EBook Gratuito

APPRENDIMENTO
Python Language

Free unaffiliated eBook created from
Stack Overflow contributors.

#python

Sommario

Di ... 1

Capitolo 1: Iniziare con Python Language ... 2

Osservazioni ... 2

Versioni ... 3

Python 3.x ... 3

Python 2.x ... 3

Examples ... 4

Iniziare ... 4

Verifica se Python è installato ... 4

Ciao, World in Python usando IDLE ... 5

Ciao file World Python ... 5

Avvia una shell Python interattiva ... 6

Altre conchiglie online ... 7

Esegui i comandi come una stringa ... 8

Conchiglie e oltre ... 8

Creazione di variabili e assegnazione di valori ... 8

Input dell'utente ... 13

IDLE - GUI Python ... 14

Risoluzione dei problemi ... 14

Tipi di dati ... 15

Tipi incorporati ... 15

booleani ... 15

Numeri ... 16

stringhe ... 16

Sequenze e collezioni ... 17

Costanti incorporate ... 17

Test del tipo di variabili ... 18

Conversione tra tipi di dati ... 19

Tipo di stringa esplicito alla definizione di valori letterali ... 19

Tipi di dati mutevoli e immutabili 19
Moduli e funzioni integrati 20
Blocco indentazione 24
Spazi contro schede 25
Tipi di raccolta 26
Aiuto Utility 31
Creare un modulo 32
Funzione stringa - str () e repr () 33
repr () 33
str () 33
Installazione di moduli esterni tramite pip 34
Trovare / installare un pacchetto 35
Aggiornamento dei pacchetti installati 35
Aggiornamento pip 35
Installazione di Python 2.7.xe 3.x 36
Capitolo 2: * args e ** kwargs 39
Osservazioni 39
h11 39
h12 39
h13 39
Examples 40
Usare * args durante la scrittura di funzioni 40
Usare ** kwargs durante la scrittura di funzioni 40
Usare * args quando si chiamano le funzioni 41
Usare ** kwargs quando si chiamano le funzioni 42
Usare * args quando si chiamano le funzioni 42
Argomenti richiesti solo per parola chiave e parola chiave 43
Popolazione dei valori di kwarg con un dizionario 43
** kwargs e valori predefiniti 43
Capitolo 3: Abstract Base Classes (abc) 44
Examples 44

Impostazione del metaclass ABCMeta ... 44
Perché / Come usare ABCMeta e @abstractmethod ... 45
Capitolo 4: accantonare ... 47
introduzione ... 47
Osservazioni ... 47
Avvertimento: ... 47
restrizioni ... 47
Examples ... 47
Codice di esempio per shelve ... 48
Per riepilogare l'interfaccia (la chiave è una stringa, i dati sono un oggetto arbitrario) ... 48
Creare un nuovo scaffale ... 48
Rispondere ... 49
Capitolo 5: Accesso agli attributi ... 51
Sintassi ... 51
Examples ... 51
Accesso di base agli attributi usando la notazione dot ... 51
Setter, Getters e Proprietà ... 51
Capitolo 6: Accesso al codice sorgente e al bytecode Python ... 54
Examples ... 54
Visualizza il bytecode di una funzione ... 54
Esplorazione dell'oggetto codice di una funzione ... 54
Mostra il codice sorgente di un oggetto ... 54
Oggetti che non sono integrati ... 54
Oggetti definiti in modo interattivo ... 55
Oggetti built-in ... 55
Capitolo 7: Accesso al database ... 57
Osservazioni ... 57
Examples ... 57
Accesso al database MySQL usando MySQLdb ... 57
SQLite ... 58
La sintassi SQLite: un'analisi approfondita ... 59
Iniziare ... 59

h21 ... 59
Attributi importanti e funzioni di Connection ... 59
Funzioni importanti del Cursor ... 60
Tipi di dati SQLite e Python ... 64
Accesso al database PostgreSQL utilizzando psycopg2 ... 64
Stabilire una connessione al database e creare una tabella ... 64
Inserimento di dati nella tabella: ... 64
Recupero dei dati della tabella: ... 65
Database Oracle ... 65
Connessione ... 67
Utilizzando sqlalchemy ... 68
Capitolo 8: Albero di sintassi astratto ... 69
Examples ... 69
Analizza le funzioni in uno script python ... 69
Capitolo 9: Alternative per cambiare dichiarazione da altre lingue ... 71
Osservazioni ... 71
Examples ... 71
Usa ciò che la lingua offre: il costrutto if / else ... 71
Usa un comando di funzioni ... 71
Usa l'introspezione di classe ... 72
Utilizzando un gestore di contesto ... 73
Capitolo 10: ambiente virtuale con virtualenvwrapper ... 75
introduzione ... 75
Examples ... 75
Crea un ambiente virtuale con virtualenvwrapper ... 75
Capitolo 11: Ambienti virtuali ... 77
introduzione ... 77
Osservazioni ... 77
Examples ... 77
Creazione e utilizzo di un ambiente virtuale ... 77
Installazione dello strumento virtualenv ... 77
Creare un nuovo ambiente virtuale ... 77

Attivazione di un ambiente virtuale esistente ... 78
Salvataggio e ripristino delle dipendenze ... 78
Uscita da un ambiente virtuale ... 78
Utilizzo di un ambiente virtuale in un host condiviso ... 79
Ambienti virtuali integrati ... 79
Installazione dei pacchetti in un ambiente virtuale ... 80
Creazione di un ambiente virtuale per una versione diversa di python ... 81
Gestione di più ambienti virtuali con virtualenvwrapper ... 81
Installazione ... 81
uso ... 82
Directory di progetto ... 82
Scopri quale ambiente virtuale stai usando ... 83
Specifica specifica della versione python da usare nello script su Unix / Linux ... 83
Utilizzo di virtualenv con guscio di pesce ... 84
Realizzare ambienti virtuali usando Anaconda ... 84
Crea un ambiente ... 85
Attiva e disattiva il tuo ambiente ... 85
Visualizza un elenco di ambienti creati ... 85
Rimuovi un ambiente ... 85
Verifica se si sta eseguendo all'interno di un ambiente virtuale ... 85
Capitolo 12: Ambito e legame variabili ... 87
Sintassi ... 87
Examples ... 87
Variabili globali ... 87
Variabili locali ... 88
Variabili non locali ... 89
Evento obbligatorio ... 89
Le funzioni ignorano l'ambito della classe durante la ricerca dei nomi ... 90
Il comando ... 91
del v ... 91
del v.name ... 91

del v[item] ... 91
del v[a:b] ... 92
Local vs Global Scope ... 92
Quali sono gli obiettivi locali e globali? ... 92
Cosa succede con le scontro sul nome? ... 92
Funzioni all'interno delle funzioni ... 93
global vs nonlocal (solo Python 3) ... 94
Capitolo 13: Analisi HTML ... 96
Examples ... 96
Trova un testo dopo un elemento in BeautifulSoup ... 96
Utilizzo dei selettori CSS in BeautifulSoup ... 96
PyQuery ... 97
Capitolo 14: Analizzare gli argomenti della riga di comando ... 98
introduzione ... 98
Examples ... 98
Ciao mondo in argparse ... 98
Esempio di base con docopt ... 99
Impostazione di argomenti mutuamente esclusivi con argparse ... 99
Utilizzo degli argomenti della riga di comando con argv ... 100
Messaggio di errore del parser personalizzato con argparse ... 101
Raggruppamento concettuale di argomenti con argparse.add_argument_group () ... 101
Esempio avanzato con docopt e docopt_dispatch ... 103
Capitolo 15: ArcPy ... 104
Osservazioni ... 104
Examples ... 104
Stampa del valore di un campo per tutte le righe della feature class nel geodatabase di fi ... 104
createDissolvedGDB per creare un file gdb nell'area di lavoro ... 104
Capitolo 16: Array ... 105
introduzione ... 105
Parametri ... 105
Examples ... 105

Introduzione di base agli array 105
Accedi a singoli elementi tramite indici 106
Aggiungi qualsiasi valore all'array usando il metodo append () 107
Inserire il valore in un array usando il metodo insert () 107
Estendi array Python usando il metodo extend () 107
Aggiungi elementi dalla lista alla matrice usando il metodo fromlist () 107
Rimuovi qualsiasi elemento dell'array usando il metodo remove () 107
Rimuovi l'ultimo elemento dell'array usando il metodo pop () 108
Recupera qualsiasi elemento attraverso il suo indice usando il metodo index () 108
Invertire un array python usando il metodo reverse () 108
Ottieni informazioni sul buffer dell'array tramite il metodo buffer_info () 108
Controlla il numero di occorrenze di un elemento usando il metodo count () 109
Converti array in string usando il metodo tostring () 109
Converti array in un elenco python con gli stessi elementi usando il metodo tolist () 109
Aggiungi una stringa al char array usando il metodo fromstring () 109
Capitolo 17: Audio 110
Examples 110
Audio con Pyglet 110
Lavorare con i file WAV 110
winsound 110
onda 110
Converti qualsiasi file sonoro con python e ffmpeg 111
Riproduzione dei beep di Windows 111
Capitolo 18: Aumentare errori / eccezioni personalizzati 112
introduzione 112
Examples 112
Eccezione personalizzata 112
Cattura eccezione personalizzata 112
Capitolo 19: Blocchi di codice, frame di esecuzione e spazi dei nomi 114
introduzione 114
Examples 114
Spazi dei nomi di codice 114

Capitolo 20: Calcolo parallelo 115
Osservazioni 115
Examples 115
Utilizzo del modulo multiprocessing per parallelizzare le attività 115
Utilizzo degli script padre e figlio per eseguire il codice in parallelo 115
Utilizzo di un'estensione C per parallelizzare le attività 116
Utilizzando il modulo PyPar per parallelizzare 116
Capitolo 21: Caratteristiche nascoste 118
Examples 118
Sovraccarico dell'operatore 118
Capitolo 22: ChemPy - pacchetto python 120
introduzione 120
Examples 120
Formule di analisi 120
Bilanciatura della stechiometria di una reazione chimica 120
Reazioni di bilanciamento 120
Equilibri chimici 121
Forza ionica 121
Cinetica chimica (sistema di equazioni differenziali ordinarie) 121
Capitolo 23: Chiama Python da C # 123
introduzione 123
Osservazioni 123
Examples 124
Script Python da chiamare con l'applicazione C # 124
Codice C # che chiama lo script Python 125
Capitolo 24: Classi 127
introduzione 127
Examples 127
Ereditarietà di base 127
Funzioni integrate che funzionano con l'ereditarietà 128
Classi e variabili di istanza 128
Metodi legati, non associati e statici 129

Classi vecchio stile e vecchio stile 132
Valori predefiniti per variabili di istanza 133
Eredità multipla 134
Descrittori e ricerche punteggiate 136
Metodi di classe: inizializzatori alternativi 136
Composizione di classe 138
Patch per scimmia 139
Elenco di tutti i membri della classe 140
Introduzione alle classi 141
Proprietà 142
Classe Singleton 144
Capitolo 25: Commenti e documentazione 147
Sintassi 147
Osservazioni 147
Examples 147
Commenti a riga singola, in linea e multilinea 147
Accesso programmatico alle docstring 147
Una funzione di esempio 148
Un'altra funzione di esempio 148
Vantaggi delle docstring rispetto ai commenti regolari 148
Scrivi la documentazione usando le docstring 149
Convenzioni sulla sintassi 149
PEP 257 149
Sfinge 150
Google Style Guide di Google 151
Capitolo 26: Comprensione delle liste 152
introduzione 152
Sintassi 152
Osservazioni 152
Examples 152
Elenco delle comprensioni 152

altro 153
Doppia iterazione 154
Mutazione sul posto e altri effetti collaterali 154
Spazio bianco nella comprensione delle liste 155
Comprensioni del dizionario 155
Espressioni del generatore 157
Casi d'uso 159
Imposta Comprensioni 159
Evita operazioni ripetitive e costose usando la clausola condizionale 160
Comprensioni che riguardano le tuple 162
Conteggio delle ricorrenze utilizzando la comprensione 162
Modifica dei tipi in un elenco 163
Capitolo 27: Concorrenza Python 164
Osservazioni 164
Examples 164
Il modulo di filettatura 164
Il modulo multiprocessing 164
Passaggio di dati tra processi di multiprocessing 165
Capitolo 28: Condizionali 168
introduzione 168
Sintassi 168
Examples 168
se, elif, e altro 168
Espressione condizionale (o "Operatore ternario") 168
Se la dichiarazione 169
Altra affermazione 169
Espressioni logiche booleane 170
E l'operatore 170
O operatore 170
Valutazione pigra 170
Test per più condizioni 171

Valori della verità 172
Usando la funzione cmp per ottenere il risultato del confronto di due oggetti 172
Valutazione dell'espressione condizionale usando le comprensioni degli elenchi 173
Verificare se un oggetto è Nessuno e assegnarlo 174
Capitolo 29: ConfigParser 175
introduzione 175
Sintassi 175
Osservazioni 175
Examples 175
Utilizzo di base 175
Creazione di file di configurazione in modo programmatico 176
Capitolo 30: confronti 177
Sintassi 177
Parametri 177
Examples 177
Maggiore o minore di 177
Non uguale a 178
Uguale a 178
Confronti a catena 179
Stile 179
Effetti collaterali 179
Confronto con `is` vs` == ` 180
Confronto di oggetti 181
Common Gotcha: Python non impone la digitazione 182
Capitolo 31: Connessione di Python a SQL Server 183
Examples 183
Connetti al server, Crea tabella, dati di query 183
Capitolo 32: Conteggio 185
Examples 185
Conteggio di tutte le occorrenze di tutti gli articoli in un iterable: collections.Counter 185
Ottenere il valore più comune (-s): collections.Counter.most_common () 185

Conteggio delle occorrenze di un elemento in una sequenza: list.count () e tuple.count () 186
Conteggio delle occorrenze di una sottostringa in una stringa: str.count () 186
Conteggio delle occorrenze nell'array numpy 186
Capitolo 33: Copia dei dati 188
Examples 188
Esecuzione di una copia superficiale 188
Esecuzione di una copia profonda 188
Esecuzione di una copia superficiale di un elenco 188
Copia un dizionario 188
Copia un set 189
Capitolo 34: Crea un ambiente virtuale con virtualenvwrapper in windows 190
Examples 190
Ambiente virtuale con virtualenvwrapper per windows 190
Capitolo 35: Creare un servizio Windows usando Python 192
introduzione 192
Examples 192
Uno script Python che può essere eseguito come servizio 192
Esecuzione di un'applicazione Web Flask come servizio 193
Capitolo 36: Creazione di pacchetti Python 195
Osservazioni 195
Examples 195
introduzione 195
Caricamento su PyPI 196
Imposta un file .pypirc 196
Registrati e carica su testpypi (opzionale) 196
analisi 197
Registrati e carica su PyPI 197
Documentazione 197
Leggimi 198
Licenze 198
Rendere eseguibile il pacchetto 198

Capitolo 37: ctypes 199
introduzione 199
Examples 199
Utilizzo di base 199
Insidie comuni 199
Impossibile caricare un file 199
Impossibile accedere a una funzione 200
Oggetto ctypes di base 200
array di tipi 201
Funzioni di avvolgimento per i tipi 202
Utilizzo complesso 202
Capitolo 38: Cuscino 204
Examples 204
Leggi il file immagine 204
Converti file in JPEG 204
Capitolo 39: Data e ora 205
Osservazioni 205
Examples 205
Analisi di una stringa in un oggetto datetime timezone aware 205
Data semplice aritmetica 205
Utilizzo di oggetti datetime di base 206
Scorri le date 206
Analisi di una stringa con un nome breve fuso orario in un oggetto datetime timezone aware 207
Costruzione di datetize sensibili al fuso orario 208
Parodia fuzzy parsing (estraendo datetime da un testo) 210
Passaggio da un fuso orario all'altro 210
Analisi di un timestamp arbitrario ISO 8601 con librerie minime 210
Conversione del timestamp in data / ora 211
Sottraendo con precisione mesi da una data 212
Calcolo delle differenze di orario 212
Ottieni un timestamp ISO 8601 213
Senza fuso orario, con microsecondi 213

Con il fuso orario, con microsecondi 213
Con il fuso orario, senza microsecondi 213
Capitolo 40: Dati binari 214
Sintassi 214
Examples 214
Formattare un elenco di valori in un oggetto byte 214
Disimballare un oggetto byte in base a una stringa di formato 214
Imballaggio di una struttura 214
Capitolo 41: Debug 216
Examples 216
The Python Debugger: debug passo dopo passo con _pdb_ 216
Via IPython e ipdb 218
Debugger remoto 218
Capitolo 42: decoratori 220
introduzione 220
Sintassi 220
Parametri 220
Examples 220
Funzione Decoratore 220
Classe decoratore 221
Metodi di decorazione 222
Avvertimento! 223
Fare un decoratore assomiglia alla funzione decorata 223
Come una funzione 223
Come classe 224
Decoratore con argomenti (fabbrica di decorazioni) 224
Funzioni del decoratore 224
Nota importante: 225
Lezioni di decoratore 225
Crea una classe singleton con un decoratore 225
Utilizzare un decoratore per cronometrare una funzione 226

Capitolo 43: Definire funzioni con argomenti lista 227
Examples 227
Funzione e chiamata 227
Capitolo 44: dentellatura 228
Examples 228
Errori di indentazione 228
Semplice esempio 228
Spazi o tabulati? 229
Come l'indentazione è analizzata 229
Capitolo 45: descrittore 231
Examples 231
Descrittore semplice 231
Conversioni bidirezionali 232
Capitolo 46: Differenza tra modulo e pacchetto 234
Osservazioni 234
Examples 234
moduli 234
Pacchi 234
Capitolo 47: Distribuzione 236
Examples 236
py2app 236
cx_Freeze 237
Capitolo 48: Distribuzione 239
Examples 239
Caricamento di un pacchetto Conda 239
Capitolo 49: Dizionario 241
Sintassi 241
Parametri 241
Osservazioni 241
Examples 241
Accesso ai valori di un dizionario 241

Il costruttore dict () 242
Evitare Eccezioni KeyError 242
Accesso a chiavi e valori 243
Introduzione al dizionario 244
creando un ditt 244
sintassi letterale 244
Detto comprensione 244
classe built-in: dict() 244
modificare un dett 245
Dizionario con valori predefiniti 245
Creare un dizionario ordinato 246
Disimballaggio dei dizionari usando l'operatore ** 246
Unione di dizionari 247
Python 3.5+ 247
Python 3.3+ 247
Python 2.x, 3.x 247
La virgola finale 247
Tutte le combinazioni di valori del dizionario 248
Iterare su un dizionario 248
Creare un dizionario 249
Esempio di dizionari 250
Capitolo 50: Django 251
introduzione 251
Examples 251
Ciao mondo con Django 251
Capitolo 51: eccezioni 254
introduzione 254
Sintassi 254
Examples 254
Sollevare le eccezioni 254
Cattura le eccezioni 254

Esecuzione del codice di pulizia con finalmente ... 255
Risollevare le eccezioni ... 255
Eccezioni a catena con rilancio da ... 256
Gerarchia delle eccezioni ... 256
Le eccezioni sono anche gli oggetti ... 259
Creazione di tipi di eccezioni personalizzate ... 259
Non prendere tutto! ... 260
Cattura più eccezioni ... 260
Esempi pratici di gestione delle eccezioni ... 261
Input dell'utente ... 261
dizionari ... 261
Altro ... 262
Capitolo 52: Eccezioni del Commonwealth ... 264
introduzione ... 264
Examples ... 264
IndentazioneErrori (o indentazione SintassiErrori) ... 264
IndentationError / SyntaxError: indent inaspettato ... 264
Esempio ... 264
IndentationError / SyntaxError: unindent non corrisponde a nessun livello di indentazione ... 265
Esempio ... 265
IndentationError: previsto un blocco rientrato ... 265
Esempio ... 265
IndentationError: uso incoerente di tabulazioni e spazi in indentazione ... 265
Esempio ... 266
Come evitare questo errore ... 266
TypeErrors ... 266
TypeError: [definizione / metodo] richiede? argomenti posizionali ma? Venne dato ... 266
Esempio ... 266
TypeError: tipi di operando non supportati per [operando]: '???' e '???' ... 267
Esempio ... 267
TypeError: '???' l'oggetto non è iterable / subscriptable: ... 267

Esempio ... 267
TypeError: '???' l'oggetto non è chiamabile ... 268
Esempio ... 268
NameError: name '???' non è definito ... 268
Semplicemente non è definito da nessuna parte nel codice ... 268
Forse è definito più tardi: ... 268
O non è stato import : ... 269
Scopi Python e la regola LEGB: ... 269
Altri errori ... 269
AssertError ... 269
KeyboardInterrupt ... 270
ZeroDivisionError ... 270
Errore di sintassi su buon codice ... 271
Capitolo 53: Elenco ... 272
introduzione ... 272
Sintassi ... 272
Osservazioni ... 272
Examples ... 272
Accedere ai valori di lista ... 272
Elenca metodi e operatori supportati ... 274
Lunghezza di una lista ... 279
Iterare su una lista ... 279
Verifica se un elemento è in una lista ... 280
Invertire gli elementi dell'elenco ... 281
Verifica se l'elenco è vuoto ... 281
Concatena e Unisci elenchi ... 281
Ogni e qualsiasi ... 283
Rimuovi i valori duplicati nella lista ... 283
Accesso ai valori nell'elenco nidificato ... 283
Confronto di liste ... 285
Inizializzazione di un elenco su un numero fisso di elementi ... 285
Capitolo 54: Elenco delle comprensioni ... 287

introduzione 287
Sintassi 287
Osservazioni 287
Examples 287
Comprensioni di liste condizionali 287
Elenco delle conclusioni con cicli annidati 289
Refactoring filter e map to list comprehensions 290
Refactoring - Riferimento rapido 291
Comprensioni di liste annidate 291
Iterate due o più liste contemporaneamente all'interno della comprensione delle liste 292
Capitolo 55: Elenco delle sezioni (selezione di parti di elenchi) 294
Sintassi 294
Osservazioni 294
Examples 294
Utilizzando il terzo argomento "step" 294
Selezione di una sottolista da una lista 294
Inversione di una lista con affettatura 295
Spostamento di un elenco utilizzando l'affettatura 295
Capitolo 56: Elenco di destrutturazione (ovvero imballaggio e disimballaggio) 297
Examples 297
Incarico distruttivo 297
Distruzione come valori 297
Distruzione come lista 297
Ignorare i valori nei compiti distruttivi 298
Ignorare le liste in compiti distruttivi 298
Argomenti della funzione di imballaggio 298
Imballaggio di un elenco di argomenti 299
Imballaggio degli argomenti delle parole chiave 299
Spacchettare gli argomenti della funzione 301
Capitolo 57: elevamento a potenza 302
Sintassi 302
Examples 302

Radice quadrata: math.sqrt () e cmath.sqrt ... 302
Esponenziazione usando i builtin: ** e pow () ... 303
Esponenziazione usando il modulo matematico: math.pow () ... 303
Funzione esponenziale: math.exp () e cmath.exp () ... 304
Funzione esponenziale meno 1: math.expm1 () ... 304
Metodi magici ed esponenziazione: builtin, matematica e cmath ... 305
Esponenziazione modulare: pow () con 3 argomenti ... 306
Radici: nth-root con esponenti frazionali ... 307
Calcolo di radici intere di grandi dimensioni ... 307
Capitolo 58: enum ... 309
Osservazioni ... 309
Examples ... 309
Creazione di un enum (Python 2.4 tramite 3.3) ... 309
Iterazione ... 309
Capitolo 59: Esecuzione di codice dinamico con `exec` e` eval` ... 310
Sintassi ... 310
Parametri ... 310
Osservazioni ... 310
Examples ... 311
Valutazione di dichiarazioni con exec ... 311
Valutare un'espressione con eval ... 311
Precompilare un'espressione per valutarla più volte ... 311
Valutare un'espressione con eval utilizzando globals personalizzati ... 311
Valutare una stringa contenente un letterale Python con ast.literal_eval ... 312
Esecuzione del codice fornito dall'utente non fidato mediante exec, eval o ast.literal_eva ... 312
Capitolo 60: Espressioni regolari (Regex) ... 313
introduzione ... 313
Sintassi ... 313
Examples ... 313
Corrisponde all'inizio di una stringa ... 313
Ricerca ... 314
Raggruppamento ... 315

Gruppi con nome ... 316
Gruppi non catturanti ... 316
Escaping Special Characters ... 317
Sostituzione ... 317
Sostituire le stringhe ... 317
Utilizzando riferimenti di gruppo ... 317
Utilizzando una funzione di sostituzione ... 318
Trova tutte le corrispondenze non sovrapposte ... 318
Modelli precompilati ... 318
Verifica dei caratteri consentiti ... 319
Dividere una stringa usando le espressioni regolari ... 319
bandiere ... 319
Parola chiave delle bandiere ... 320
Bandiere in linea ... 320
Iterare le partite usando `re.finditer` ... 321
Abbina un'espressione solo in posizioni specifiche ... 321
Capitolo 61: Eventi inviati dal server Python ... 323
introduzione ... 323
Examples ... 323
Flacone SSE ... 323
Asyncio SSE ... 323
Capitolo 62: File e cartelle I / O ... 324
introduzione ... 324
Sintassi ... 324
Parametri ... 324
Osservazioni ... 324
Evitare l'Encoding Hell multipiattaforma ... 324
Examples ... 325
Modalità file ... 325
Leggere un file riga per riga ... 327
Ottenere l'intero contenuto di un file ... 327

Scrivere su un file ... 328
Copia del contenuto di un file in un altro file ... 329
Controlla se esiste un file o percorso ... 329
Copia un albero di directory ... 330
Iterare i file (in modo ricorsivo) ... 330
Leggi un file tra un intervallo di linee ... 331
Accesso casuale ai file tramite mmap ... 331
Sostituzione del testo in un file ... 332
Verifica se un file è vuoto ... 332
Capitolo 63: Filtro ... 333
Sintassi ... 333
Parametri ... 333
Osservazioni ... 333
Examples ... 333
Uso di base del filtro ... 333
Filtro senza funzione ... 334
Filtro come controllo di cortocircuito ... 334
Funzione complementare: filterfalse, ifilterfalse ... 335
Capitolo 64: Formattazione della data ... 337
Examples ... 337
Tempo tra due date ... 337
Analisi della stringa sull'oggetto datetime ... 337
Uscita dell'oggetto datetime alla stringa ... 337
Capitolo 65: Formattazione di stringhe ... 338
introduzione ... 338
Sintassi ... 338
Osservazioni ... 338
Examples ... 338
Nozioni di base sulla formattazione delle stringhe ... 338
Allineamento e imbottitura ... 340
Formato letterale (stringa-f) ... 340
Formattazione delle stringhe con datetime ... 341

Formato utilizzando Getitem e Getattr ... 342
Formattazione mobile ... 342
Formattazione dei valori numerici ... 343
Formattazione personalizzata per una classe ... 343
Formattazione nidificata ... 345
Stringhe di imbottitura e troncatura, combinate ... 345
Segnaposto nominati ... 346
Usare un dizionario (Python 2.x) ... 346
Usare un dizionario (Python 3.2+) ... 346
Senza un dizionario: ... 346
Capitolo 66: Funzione mappa ... 347
Sintassi ... 347
Parametri ... 347
Osservazioni ... 347
Examples ... 347
Uso di base della mappa, itertools.imap e future_builtins.map ... 347
Mappare ciascun valore in un iterabile ... 348
Mappatura dei valori di diversi iterabili ... 349
Trasposizione con mappa: utilizzo di "Nessuno" come argomento di funzione (solo python 2.x ... 350
Serie e mappatura parallela ... 351
Capitolo 67: funzioni ... 354
introduzione ... 354
Sintassi ... 354
Parametri ... 354
Osservazioni ... 354
Risorse addizionali ... 355
Examples ... 355
Definire e chiamare semplici funzioni ... 355
Valori di ritorno da funzioni ... 357
Definire una funzione con argomenti ... 358
Definizione di una funzione con argomenti opzionali ... 358
avvertimento ... 359

Definizione di una funzione con più argomenti 359
Definire una funzione con un numero arbitrario di argomenti 359
Numero arbitrario di argomenti posizionali: 359
Numero arbitrario di argomenti di parole chiave 360
avvertimento 361
Nota sulla denominazione 362
Nota sull'unicità 362
Nota sulle funzioni di annidamento con argomenti opzionali 362
Definizione di una funzione con argomenti mutabili opzionali 362
Spiegazione 362
Soluzione 363
Funzioni Lambda (Inline / Anonimo) 364
Passaggio di argomenti e mutabilità 366
Chiusura 367
Funzioni ricorsive 368
Limite di ricorsione 369
Funzioni annidate 369
Iterable e dizionario disimballaggio 370
Forzare l'uso dei parametri denominati 372
Lambda ricorsivo con variabile assegnata 372
Descrizione del codice 372
Capitolo 68: Funzioni parziali 374
introduzione 374
Sintassi 374
Parametri 374
Osservazioni 374
Examples 374
Alza il potere 374
Capitolo 69: generatori 376
introduzione 376
Sintassi 376

Examples ... 376
Iterazione ... 376
La funzione next () ... 376
Invio di oggetti a un generatore ... 377
Espressioni del generatore ... 378
introduzione ... 378
Usare un generatore per trovare i numeri di Fibonacci ... 381
Sequenze infinite ... 381
Esempio classico: numeri di Fibonacci ... 382
Cedendo tutti i valori da un altro iterabile ... 382
coroutine ... 383
Resa con ricorsione: elenca in modo ricorsivo tutti i file in una directory ... 383
Iterazione su generatori in parallelo ... 384
Refactoring code-list code ... 384
Ricerca ... 385
Capitolo 70: Gestori di contesto ("con" istruzione) ... 387
introduzione ... 387
Sintassi ... 387
Osservazioni ... 387
Examples ... 388
Introduzione ai gestori di contesto e alla dichiarazione con ... 388
Assegnare a un bersaglio ... 388
Scrivi il tuo gestore di contesto ... 389
Scrivere il proprio contextmanager usando la sintassi del generatore ... 389
Più gestori di contesto ... 391
Gestisci risorse ... 391
Capitolo 71: Grafica tartaruga ... 392
Examples ... 392
Ninja Twist (Turtle Graphics) ... 392
Capitolo 72: grafico-utensile ... 393
introduzione ... 393
Examples ... 393

PyDotPlus ... 393
Installazione ... 393
PyGraphviz ... 394
Capitolo 73: hashlib ... 396
introduzione ... 396
Examples ... 396
Hash MD5 di una stringa ... 396
algoritmo fornito da OpenSSL ... 397
Capitolo 74: Heapq ... 398
Examples ... 398
Articoli più grandi e più piccoli in una collezione ... 398
Il più piccolo oggetto di una collezione ... 398
Capitolo 75: idiomi ... 400
Examples ... 400
Inizializzazione chiavi del dizionario ... 400
Commutazione delle variabili ... 400
Utilizzare il test del valore di verità ... 400
Prova per "__main__" per evitare l'esecuzione inaspettata del codice ... 401
Capitolo 76: ijson ... 402
introduzione ... 402
Examples ... 402
Semplice esempio ... 402
Capitolo 77: Il modulo base64 ... 403
introduzione ... 403
Sintassi ... 403
Parametri ... 403
Osservazioni ... 405
Examples ... 405
Codifica e decodifica Base64 ... 405
Codifica e decodifica di Base32 ... 407
Codifica e decodifica Base16 ... 407
Codifica e decodifica ASCII85 ... 408

Codifica e decodifica Base85 ... 408
Capitolo 78: Il modulo dis ... 410
Examples ... 410
Costanti nel modulo DIS ... 410
Cos'è il bytecode Python? ... 410
Disassemblare i moduli ... 410
Capitolo 79: Il modulo locale ... 412
Osservazioni ... 412
Examples ... 412
Valuta Formattazione dei dollari USA utilizzando il modulo locale ... 412
Capitolo 80: Il modulo os ... 413
introduzione ... 413
Sintassi ... 413
Parametri ... 413
Examples ... 413
Crea una directory ... 413
Ottieni la directory corrente ... 413
Determina il nome del sistema operativo ... 413
Rimuovi una directory ... 414
Segui un link simbolico (POSIX) ... 414
Modifica le autorizzazioni su un file ... 414
makedir - creazione di directory ricorsiva ... 414
Capitolo 81: Implementazioni Python non ufficiali ... 416
Examples ... 416
IronPython ... 416
Ciao mondo ... 416
link esterno ... 416
Jython ... 416
Ciao mondo ... 417
link esterno ... 417
Transcrypt ... 417

Dimensioni e velocità del codice ... 417
Integrazione con HTML ... 418
Integrazione con JavaScript e DOM ... 418
Integrazione con altre librerie JavaScript ... 418
Relazione tra Python e codice JavaScript ... 419
link esterno ... 420
Capitolo 82: Importazione di moduli ... 421
Sintassi ... 421
Osservazioni ... 421
Examples ... 421
Importare un modulo ... 421
Importazione di nomi specifici da un modulo ... 423
Importare tutti i nomi da un modulo ... 423
La variabile speciale __all__ ... 424
Importazione programmatica ... 425
Importa moduli da una posizione arbitraria del filesystem ... 425
Regole PEP8 per le importazioni ... 426
Importazione di sottomoduli ... 426
__import __ () funzione ... 426
Reimportazione di un modulo ... 427
Python 2 ... 427
Python 3 ... 427
Capitolo 83: Impostato ... 429
Sintassi ... 429
Osservazioni ... 429
Examples ... 429
Ottieni gli elementi unici di una lista ... 429
Operazioni sui set ... 430
Imposta contro multiset ... 431
Imposta le operazioni usando Methods e Builtins ... 432
Intersezione ... 432

Unione ... 432
Differenza ... 432
Differenza simmetrica ... 432
Sottoinsieme e superset ... 433
Set disgiunti ... 433
Test dell'appartenenza ... 434
Lunghezza ... 434
Set di Set ... 434
Capitolo 84: Incompatibilità che si spostano da Python 2 a Python 3 ... 435
introduzione ... 435
Osservazioni ... 435
Examples ... 436
Stampa dichiarazione vs. funzione di stampa ... 436
Stringhe: byte contro Unicode ... 437
Divisione intera ... 439
Ridurre non è più un built-in ... 441
Differenze tra le funzioni range e xrange ... 442
Compatibilità ... 443
Disimballare Iterables ... 444
Sollevamento e gestione delle eccezioni ... 446
.next () metodo sugli iteratori rinominato ... 448
Confronto tra diversi tipi ... 448
Input dell'utente ... 449
Modifiche al metodo dei dizionari ... 450
la dichiarazione exec è una funzione in Python 3 ... 451
hasattr bug di funzionalità in Python 2 ... 451
Moduli rinominati ... 452
Compatibilità ... 452
Costanti ottali ... 453
Tutte le classi sono "classi di nuovo stile" in Python 3. ... 453
Operatori rimossi <> e ``, sinonimi di! = E repr () ... 454
codifica / decodifica in esadecimale non più disponibile ... 454

funzione cmp rimossa in Python 3 ... 455
Variabili trapelate nella comprensione delle liste ... 456
carta geografica() ... 457
filter (), map () e zip () restituiscono gli iteratori anziché le sequenze ... 458
Importazioni assolute / relative ... 459
Maggiori informazioni sulle importazioni relative ... 459
File I / O ... 460
La funzione round () tie-break e return ... 461
round () tie break ... 461
round () tipo di ritorno ... 462
Vero, Falso e Nessuno ... 462
Restituisce valore quando si scrive su un oggetto file ... 463
long vs. int ... 463
Classe Valore booleano ... 464
Capitolo 85: Indicizzazione e affettatura ... 465
Sintassi ... 465
Parametri ... 465
Osservazioni ... 465
Examples ... 465
Slicing di base ... 465
Fare una copia superficiale di un array ... 466
Inversione di un oggetto ... 467
Indicizzazione delle classi personalizzate: __getitem__, __setitem__ e __delitem__ ... 467
Assegnazione delle fette ... 468
Affetta oggetti ... 469
Indicizzazione di base ... 469
Capitolo 86: iniziare con GZip ... 471
introduzione ... 471
Examples ... 471
Leggi e scrivi i file zip GNU ... 471
Capitolo 87: Input e output di base ... 472
Examples ... 472

Utilizzo di input () e raw_input () ... 472
Utilizzando la funzione di stampa ... 472
Funzione per richiedere all'utente un numero ... 473
Stampa una stringa senza una nuova riga alla fine ... 473
Leggi da stdin ... 474
Input da un file ... 474
Capitolo 88: Input, Subset e Output File di dati esterni che utilizzano Panda ... 477
introduzione ... 477
Examples ... 477
Codice di base per importare, sottoinsi e scrivere file di dati esterni usando panda ... 477
Capitolo 89: Insidie comuni ... 479
introduzione ... 479
Examples ... 479
Modifica della sequenza su cui stai iterando ... 479
Argomento predefinito mutabile ... 482
Elenco di moltiplicazione e riferimenti comuni ... 483
Intero e identità di stringa ... 487
Accesso agli attributi letterali di int ... 488
Concatenamento di o operatore ... 489
sys.argv [0] è il nome del file in esecuzione ... 490
h14 ... 490
I dizionari non sono ordinati ... 490
Global Interpreter Lock (GIL) e thread di blocco ... 491
Perdita variabile nella comprensione delle liste e nei cicli ... 492
Ritorno multiplo ... 493
Chiavi Python JSON ... 493
Capitolo 90: Interfaccia Web Server Gateway (WSGI) ... 495
Parametri ... 495
Examples ... 495
Oggetto server (metodo) ... 495
Capitolo 91: Introduzione a RabbitMQ usando AMQPStorm ... 497
Osservazioni ... 497

Examples ... 497
Come consumare messaggi da RabbitMQ ... 497
Come pubblicare messaggi su RabbitMQ ... 498
Come creare una coda in ritardo in RabbitMQ ... 499
Capitolo 92: Iterables e Iterators ... 501
Examples ... 501
Iterator vs Iterable vs Generator ... 501
Cosa può essere iterabile ... 502
Iterazione su intero iterabile ... 502
Verifica solo un elemento in iterabile ... 503
Estrarre i valori uno per uno ... 503
Iterator non è rientranti! ... 503
Capitolo 93: kivy - Framework Python multipiattaforma per lo sviluppo NUI ... 504
introduzione ... 504
Examples ... 504
Prima app ... 504
Capitolo 94: La dichiarazione del passaggio ... 507
Sintassi ... 507
Osservazioni ... 507
Examples ... 509
Ignora un'eccezione ... 509
Crea una nuova eccezione che può essere catturata ... 509
Capitolo 95: La funzione di stampa ... 510
Examples ... 510
Informazioni di base sulla stampa ... 510
Stampa i parametri ... 511
Capitolo 96: La variabile speciale __name__ ... 513
introduzione ... 513
Osservazioni ... 513
Examples ... 513
__name__ == '__main__' ... 513
Situazione 1 ... 513

Situazione 2 513
function_class_or_module .__ name__ 514
Utilizzare nella registrazione 515
Capitolo 97: Lavorare attorno al Global Interpreter Lock (GIL) 516
Osservazioni 516
Perché c'è un GIL? 516
Dettagli su come funziona GIL: 516
Benefici della GIL 516
Conseguenze del GIL 517
Riferimenti: 517
Examples 517
Multiprocessing.Pool 517
Il codice di David Beazley che mostrava problemi di threading GIL 518
Cython nogil: 519
Il codice di David Beazley che mostrava problemi di threading GIL 519
Riscritto usando nogil (SOLO FUNZIONA IN CYTHON): 519
Capitolo 98: Lavorare con gli archivi ZIP 521
Sintassi 521
Osservazioni 521
Examples 521
Aprire i file zip 521
Esaminando i contenuti di Zipfile 521
Estrazione del contenuto del file zip in una directory 522
Creare nuovi archivi 522
Capitolo 99: Lettura e scrittura CSV 524
Examples 524
Scrivere un file TSV 524
Pitone 524
File di uscita 524
Usando i panda 524
Capitolo 100: Libreria di sottoprocesso 525

Sintassi ... 525
Parametri ... 525
Examples ... 525
Chiamare i comandi esterni ... 525
Maggiore flessibilità con Popen ... 526
Avvio di un sottoprocesso ... 526
In attesa di un sottoprocesso da completare ... 526
Lettura dell'output da un sottoprocesso ... 526
Accesso interattivo ai sottoprocessi in esecuzione ... 526
Scrivere su un sottoprocesso ... 526
Leggere un flusso da un sottoprocesso ... 527
Come creare l'argomento della lista comandi ... 527
Capitolo 101: Liste collegate ... 529
introduzione ... 529
Examples ... 529
Esempio di elenco singolo collegato ... 529
Capitolo 102: Loops ... 533
introduzione ... 533
Sintassi ... 533
Parametri ... 533
Examples ... 533
Iterare sulle liste ... 533
Per i loop ... 534
Oggetti e iteratori iterabili ... 535
Rompi e continua in loop ... 535
dichiarazione di break ... 535
continue dichiarazione ... 536
Cicli annidati ... 536
Utilizzare il return da una funzione come una break ... 537
Loop con una clausola "else" ... 537
Perché dovremmo usare questo strano costrutto? ... 539

Iterare sui dizionari ... 540
Mentre Loop ... 541
La dichiarazione del passaggio ... 542
Iterazione di parti diverse di un elenco con diverse dimensioni del passo ... 542
Iterazione sull'intera lista ... 542
Iterare su sotto-lista ... 543
Il "mezzo giro" do-while ... 544
Looping e Disimballaggio ... 544
Capitolo 103: Maledizioni di base con Python ... 546
Osservazioni ... 546
Examples ... 546
Esempio di chiamata di base ... 546
La funzione helper wrapper () ... 546
Capitolo 104: Manipolazione di XML ... 548
Osservazioni ... 548
Examples ... 548
Aprire e leggere usando un ElementTree ... 548
Modifica di un file XML ... 548
Crea e crea documenti XML ... 549
Apertura e lettura di file XML di grandi dimensioni mediante iterparse (analisi incrementa ... 549
Ricerca nell'XML con XPath ... 550
Capitolo 105: Matematica complessa ... 552
Sintassi ... 552
Examples ... 552
Aritmetica complessa avanzata ... 552
Aritmetica complessa di base ... 553
Capitolo 106: Matrici multidimensionali ... 554
Examples ... 554
Elenchi in elenchi ... 554
Elenchi in elenchi in elenchi in 555
Capitolo 107: metaclassi ... 556
introduzione ... 556

Osservazioni 556
Examples 556
Metaclasses di base 556
Singleton usando metaclassi 557
Utilizzando un metaclass 558
Sintassi del metaclass 558
Compatibilità con Python 2 e 3 con six 558
Funzionalità personalizzate con metaclassi 558
Introduzione ai metaclassi 559
Cos'è un metaclass? 559
Il Metaclass più semplice 559
Un Metaclass che fa qualcosa 560
Il metaclass predefinito 560
Capitolo 108: Metodi definiti dall'utente 562
Examples 562
Creazione di oggetti metodo definiti dall'utente 562
Esempio di tartaruga 563
Capitolo 109: Metodi di stringa 564
Sintassi 564
Osservazioni 565
Examples 565
Modifica della maiuscola di una stringa 565
str.casefold() 565
str.upper() 566
str.lower() 566
str.capitalize() 566
str.title() 566
str.swapcase() 566
Utilizzo come metodi di classe str 566
Dividere una stringa in base a un delimitatore in un elenco di stringhe 567
str.split(sep=None, maxsplit=-1) 567
str.rsplit(sep=None, maxsplit=-1) 568

Sostituisci tutte le occorrenze di una sottostringa con un'altra sottostringa 568
str.replace(old, new[, count]) : 568
str.format e f-strings: formatta i valori in una stringa 569
Numero di volte in cui una sottostringa viene visualizzata in una stringa 570
str.count(sub[, start[, end]]) 570
Prova i caratteri iniziali e finali di una stringa 571
str.startswith(prefix[, start[, end]]) 571
str.endswith(prefix[, start[, end]]) 571
Test di cosa è composta una stringa 572
str.isalpha 572
str.isupper , str.islower , str.istitle 572
str.isdecimal , str.isdigit , str.isnumeric 573
str.isalnum 573
str.isspace 574
str.translate: traduzione di caratteri in una stringa 574
Eliminazione di caratteri iniziali / finali indesiderati da una stringa 575
str.strip([chars]) 575
str.rstrip([chars]) e str.lstrip([chars]) 575
Confronto tra stringhe senza confronti tra maiuscole e minuscole 576
Unisci un elenco di stringhe in un'unica stringa 577
Costanti utili del modulo stringa 577
string.ascii_letters : 577
string.ascii_lowercase : 578
string.ascii_uppercase : 578
string.digits : 578
string.hexdigits : 578
string.octaldigits : 578
string.punctuation : 578
string.whitespace : 578
string.printable : 579
Inversione di una stringa 579
Giustificare le corde 579
Conversione tra str o byte di dati e caratteri Unicode 580

La stringa contiene ... 581
Capitolo 110: Metodo Overriding ... 582
Examples ... 582
Metodo di base prioritario ... 582
Capitolo 111: mixins ... 583
Sintassi ... 583
Osservazioni ... 583
Examples ... 583
mixin ... 583
Metodi di sovrascrittura in Mixins ... 584
Capitolo 112: Modelli di progettazione ... 586
introduzione ... 586
Examples ... 586
Modello di strategia ... 586
Introduzione ai modelli di design e Singleton Pattern ... 587
delega ... 589
Capitolo 113: Modelli in python ... 592
Examples ... 592
Semplice programma di output dei dati utilizzando il modello ... 592
Modifica del delimitatore ... 592
Capitolo 114: Modulo Asyncio ... 593
Examples ... 593
Sintassi di Coroutine e Delegazione ... 593
Esecutori asincroni ... 594
Utilizzando UVLoop ... 595
Sincronizzazione primitiva: evento ... 595
Concetto ... 595
Esempio ... 596
Un Websocket semplice ... 596
Idee sbagliate comuni sull'asyncio ... 597
Capitolo 115: Modulo casuale ... 598

Sintassi 598
Examples 598
Casuale e sequenze: shuffle, scelta e campione 598
Shuffle () 598
scelta() 598
campione() 598
Creazione di numeri interi e float casuali: randint, randrange, random e uniform 599
randInt () 599
randrange () 599
casuale 600
uniforme 600
Numeri casuali riproducibili: seme e stato 600
Crea numeri casuali crittograficamente sicuri 601
Creazione di una password utente casuale 602
Decisione binaria casuale 603
Capitolo 116: Modulo collezioni 604
introduzione 604
Osservazioni 604
Examples 604
collections.Counter 604
collections.defaultdict 606
collections.OrderedDict 607
collections.namedtuple 608
collections.deque 609
collections.ChainMap 610
Capitolo 117: Modulo Deque 612
Sintassi 612
Parametri 612
Osservazioni 612
Examples 612
Deque di base usando 612

limite dimensioni deque ... 613
Metodi disponibili in deque ... 613
Larghezza Prima ricerca ... 614
Capitolo 118: Modulo di coda ... 615
introduzione ... 615
Examples ... 615
Semplice esempio ... 615
Capitolo 119: Modulo Functools ... 616
Examples ... 616
parziale ... 616
total_ordering ... 616
ridurre ... 617
lru_cache ... 617
cmp_to_key ... 618
Capitolo 120: Modulo Itertools ... 619
Sintassi ... 619
Examples ... 619
Raggruppamento di elementi da un oggetto iterabile mediante una funzione ... 619
Prendi una fetta di un generatore ... 620
itertools.product ... 621
itertools.count ... 621
itertools.takewhile ... 622
itertools.dropwhile ... 623
Zippare due iteratori finché non sono entrambi esauriti ... 624
Metodo di combinazioni nel modulo Itertools ... 624
Concatenare più iteratori insieme ... 625
itertools.repeat ... 625
Ottieni una somma cumulativa di numeri in un iterabile ... 625
Passa attraverso gli elementi in un iteratore ... 626
itertools.permutations ... 626
Capitolo 121: Modulo JSON ... 627
Osservazioni ... 627

tipi 627
Defaults 627
Tipi di serializzazione: 627
Tipi di serializzazione: 627
Serializzazione personalizzata (de-) 628
serializzazione: 628
De-serializzazione: 628
Ulteriori personalizzazioni (de-) serializzazione: 629
Examples 629
Creazione di JSON da Python dict 629
Creazione di Dict Python da JSON 629
Memorizzazione di dati in un file 630
Recupero di dati da un file 630
`load` vs` loads`, `dump` vs` dumps` 630
Chiamando `json.tool` dalla riga di comando all'output JSON pretty-print 631
Formattazione dell'output JSON 632
Impostazione dell'indentazione per ottenere risultati migliori 632
Ordinamento dei tasti alfabeticamente per ottenere risultati coerenti 632
Liberarsi degli spazi bianchi per ottenere risultati compatti 633
JSON che codifica oggetti personalizzati 633
Capitolo 122: Modulo matematico 634
Examples 634
Arrotondamento: rotondo, pavimento, ceil, trunc 634
Avvertimento! 635
Avviso sul piano, trunc e divisione intera dei numeri negativi 635
logaritmi 635
Copia di segni 636
Trigonometria 636
Calcolo della lunghezza dell'ipotenusa 636
Conversione dei gradi in / da radianti 636
Funzioni seno, coseno, tangente e inverso 636
Seno iperbolico, coseno e tangente 637

costanti ... 637
Numeri immaginari ... 638
Infinity e NaN ("non un numero") ... 638
Pow per un'esponenziazione più rapida ... 641
Numeri complessi e il modulo cmath ... 641
Capitolo 123: Modulo operatore ... 645
Examples ... 645
Operatori in alternativa a un operatore infisso ... 645
Methodcaller ... 645
Itemgetter ... 645
Capitolo 124: modulo pyautogui ... 647
introduzione ... 647
Examples ... 647
Funzioni del mouse ... 647
Funzioni della tastiera ... 647
ScreenShot e riconoscimento dell'immagine ... 647
Capitolo 125: Modulo Sqlite3 ... 648
Examples ... 648
Sqlite3: non richiede un processo server separato ... 648
Ottenere i valori dal database e la gestione degli errori ... 648
Capitolo 126: Modulo Webbrowser ... 650
introduzione ... 650
Sintassi ... 650
Parametri ... 650
Osservazioni ... 651
Examples ... 652
Apertura di un URL con Browser predefinito ... 652
Aprire un URL con diversi browser ... 652
Capitolo 127: multiprocessing ... 654
Examples ... 654
Esecuzione di due processi semplici ... 654
Utilizzando Pool e Mappa ... 655

Capitolo 128: multithreading 656
introduzione 656
Examples 656
Nozioni di base sul multithreading 656
Comunicare tra i thread 657
Creazione di un pool di worker 658
Uso avanzato di multithread 659
Stampante avanzata (logger) 659
Filo bloccabile con un ciclo while 660
Capitolo 129: Mutevole vs Immutabile (e Lavabile) in Python 662
Examples 662
Mutevole vs Immutabile 662
immutabili 662
Esercizio 663
Mutables 663
Esercizio 664
Mutevole e immutabile come argomenti 664
Esercizio 665
Capitolo 130: Neo4j e Cypher utilizzano Py2Neo 666
Examples 666
Importare e autenticare 666
Aggiunta di nodi al grafico Neo4j 666
Aggiunta di relazioni al grafico Neo4j 666
Query 1: completamento automatico sui titoli di notizie 667
Query 2: ottieni articoli di notizie per posizione in una data specifica 667
Cypher Query Samples 667
Capitolo 131: Nodo di elenco collegato 669
Examples 669
Scrivi un semplice nodo Elenco collegato in python 669
Capitolo 132: Oggetti di proprietà 670
Osservazioni 670

Examples 670
Usando il decoratore @property 670
Utilizzo del decoratore @property per le proprietà di lettura-scrittura 670
Sovrascrivere solo un getter, setter o un deleter di un oggetto proprietà 671
Usando le proprietà senza decoratori 671
Capitolo 133: Operatori bit a bit 674
introduzione 674
Sintassi 674
Examples 674
Bitwise AND 674
Bitwise OR 674
XOR bit a bit (OR esclusivo) 675
Spostamento a sinistra bit a bit 675
Spostamento a destra bit a bit 676
Bitwise NOT 676
Operazioni interne 678
Capitolo 134: Operatori booleani 679
Examples 679
e 679
o 679
non 680
Valutazione del cortocircuito 680
`e` e` or` non sono garantiti per restituire un valore booleano 681
Un semplice esempio 681
Capitolo 135: Ordinamento, minimo e massimo 682
Examples 682
Ottenere il minimo o il massimo di più valori 682
Usando l'argomento chiave 682
Argomento predefinito su max, min 682
Caso speciale: dizionari 683
In base al valore 683
Ottenere una sequenza ordinata 684

Minimo e Massimo di una sequenza ... 684
Rendi ordinabili le classi personalizzate ... 685
Estraendo N il più grande o il N più piccolo da un iterabile ... 687
Capitolo 136: os.path ... 689
introduzione ... 689
Sintassi ... 689
Examples ... 689
Join Paths ... 689
Percorso assoluto dal percorso relativo ... 689
Manipolazione dei componenti del percorso ... 690
Ottieni la directory principale ... 690
Se il percorso specificato esiste ... 690
controlla se il percorso dato è una directory, un file, un link simbolico, un punto di mou ... 690
Capitolo 137: Ottimizzazione delle prestazioni ... 692
Osservazioni ... 692
Examples ... 692
Codice di profilazione ... 692
Capitolo 138: Pallone ... 695
introduzione ... 695
Sintassi ... 695
Examples ... 695
Le basi ... 695
URL di instradamento ... 696
Metodi HTTP ... 696
File e modelli ... 697
Jinja Templating ... 698
L'oggetto della richiesta ... 699
Parametri URL ... 699
Upload di file ... 700
Biscotti ... 700
Capitolo 139: Persistenza di Python ... 701

Sintassi ... 701
Parametri ... 701
Examples ... 701
Persistenza di Python ... 701
Funzione di utilità per salvare e caricare ... 702
Capitolo 140: Pila ... 703
introduzione ... 703
Sintassi ... 703
Osservazioni ... 703
Examples ... 703
Creazione di una classe di stack con un oggetto di elenco ... 703
Analisi delle parentesi ... 705
Capitolo 141: pip: PyPI Package Manager ... 706
introduzione ... 706
Sintassi ... 706
Osservazioni ... 706
Examples ... 707
Installa pacchetti ... 707
Installa dai file dei requisiti ... 707
Disinstalla pacchetti ... 707
Per elencare tutti i pacchetti installati usando `pip` ... 708
Aggiorna pacchetti ... 708
Aggiornamento di tutti i pacchetti obsoleti su Linux ... 708
Aggiornamento di tutti i pacchetti obsoleti su Windows ... 709
Creare un file requirements.txt di tutti i pacchetti sul sistema ... 709
Crea un file requirements.txt di pacchetti solo nella virtualenv corrente ... 709
Usando una certa versione di Python con pip ... 709
Installare pacchetti non ancora sul pip come ruote ... 710
Nota sull'installazione di pre-release ... 711
Nota sull'installazione di versioni di sviluppo ... 712
Capitolo 142: Plugin ed Extension Classes ... 714

Examples ... 714
mixins ... 714
Plugin con classi personalizzate ... 715
Capitolo 143: Polimorfismo ... 717
Examples ... 717
Polimorfismo di base ... 717
Duck Typing ... 719
Capitolo 144: PostgreSQL ... 721
Examples ... 721
Iniziare ... 721
Installazione usando pip ... 721
Utilizzo di base ... 721
Capitolo 145: Precedenza dell'operatore ... 723
introduzione ... 723
Osservazioni ... 723
Examples ... 724
Esempi di precedenza di operatore semplice in python ... 724
Capitolo 146: Processi e discussioni ... 725
introduzione ... 725
Examples ... 725
Global Interpreter Lock ... 725
Esecuzione in più thread ... 727
Esecuzione in più processi ... 727
Condivisione dello stato tra thread ... 728
Condivisione dello stato tra processi ... 728
Capitolo 147: profiling ... 730
Examples ... 730
%% timeit e% timeit in IPython ... 730
funzione timeit () ... 730
riga di comando timeit ... 730
line_profiler nella riga di comando ... 731
Utilizzo di cProfile (profilo preferito) ... 731

Capitolo 148: Programmazione funzionale in Python 733
introduzione 733
Examples 733
Funzione Lambda 733
Funzione mappa 733
Ridurre la funzione 733
Funzione filtro 733
Capitolo 149: Programmazione IoT con Python e Raspberry PI 735
Examples 735
Esempio: sensore di temperatura 735
Capitolo 150: py.test 738
Examples 738
Impostazione di py.test 738
Il codice da testare 738
Il codice di prova 738
Esecuzione del test 738
Test falliti 739
Introduzione alle partite di prova 739
py.test infissi in soccorso! 740
Pulizia dopo i test. 742
Capitolo 151: pyaudio 744
introduzione 744
Osservazioni 744
Examples 744
Modalità di callback audio I / O 744
Modalità I / O audio blocco 745
Capitolo 152: pygame 747
introduzione 747
Sintassi 747
Parametri 747
Examples 747

Installare pygame 747
Il modulo mixer di Pygame 748
L'inizializzazione 748
Possibili azioni 748
canali 748
Capitolo 153: pyglet 750
introduzione 750
Examples 750
Ciao mondo in Pyglet 750
Installazione di Pyglet 750
Riproduzione audio in Pyglet 750
Utilizzo di Pyglet per OpenGL 750
Punti di disegno usando Pyglet e OpenGL 751
Capitolo 154: PyInstaller - Distribuzione del codice Python 752
Sintassi 752
Osservazioni 752
Examples 752
Installazione e configurazione 752
Utilizzando Pyinstaller 753
Raggruppamento in una cartella 753
vantaggi: 753
svantaggi 754
Raggruppamento in un singolo file 754
Capitolo 155: Python Anti-Patterns 755
Examples 755
Overzealous tranne clausola 755
Guardando prima di saltare con la funzione intensiva del processore 756
Chiavi del dizionario 756
Capitolo 156: Python ed Excel 758
Examples 758
Inserisci i dati dell'elenco in un file di Excel 758

OpenPyXL ... 758
Crea grafici Excel con xlsxwriter ... 759
Leggi i dati excel usando il modulo xlrd ... 761
Formatta i file Excel con xlsxwriter ... 762
Capitolo 157: Python HTTP Server ... 764
Examples ... 764
Esecuzione di un semplice server HTTP ... 764
Servire file ... 764
API programmatica di SimpleHTTPServer ... 766
Gestione di base di GET, POST, PUT utilizzando BaseHTTPRequestHandler ... 767
Capitolo 158: Python Lex-Yacc ... 769
introduzione ... 769
Osservazioni ... 769
Examples ... 769
Iniziare con PLY ... 769
"Ciao, mondo!" di PLY - A Simple Calculator ... 769
Parte 1: Tokenizing Input con Lex ... 771
Abbattersi ... 772
h22 ... 773
h23 ... 773
h24 ... 774
h25 ... 774
h26 ... 774
h27 ... 774
h28 ... 774
h29 ... 775
h210 ... 775
h211 ... 775
Parte 2: Parsing Input Tokenized con Yacc ... 775
Abbattersi ... 776
h212 ... 778

Capitolo 159: Python Networking 779
Osservazioni 779
Examples 779
Il più semplice esempio di client-server socket Python 779
Creazione di un server Http semplice 779
Creazione di un server TCP 780
Creazione di un server UDP 781
Avvia Simple HttpServer in una discussione e apri il browser 781
Capitolo 160: Python Serial Communication (pyserial) 783
Sintassi 783
Parametri 783
Osservazioni 783
Examples 783
Inizializza dispositivo seriale 783
Leggi dalla porta seriale 783
Verifica quali porte seriali sono disponibili sulla tua macchina 784
Capitolo 161: Python velocità del programma 785
Examples 785
Notazione 785
Elenca le operazioni 785
Operazioni di deque 786
Imposta le operazioni 787
Notifiche algoritmiche 787
Capitolo 162: Python Virtual Environment - virtualenv 789
introduzione 789
Examples 789
Installazione 789
uso 789
Installa un pacchetto nel tuo Virtualenv 790
Altri comandi virtualenv utili 790
Capitolo 163: Raccolta dei rifiuti 791
Osservazioni 791

Raccolta di rifiuti generazionale ... 791
Examples ... 793
Conteggio di riferimento ... 793
Garbage Collector per cicli di riferimento ... 794
Effetti del comando ... 795
Riutilizzo di oggetti primitivi ... 796
Visualizzazione del conto di un oggetto ... 796
Disabilitare forzatamente gli oggetti ... 796
Gestire la garbage collection ... 797
Non aspettare che la garbage collection sia pulita ... 798
Capitolo 164: raggruppa per() ... 800
introduzione ... 800
Sintassi ... 800
Parametri ... 800
Osservazioni ... 800
Examples ... 800
Esempio 1 ... 800
Esempio 2 ... 801
Esempio 3 ... 802
Esempio 4 ... 803
Capitolo 165: Rappresentazioni di stringhe di istanze di classe: metodi __str__ e __repr__ ... 805
Osservazioni ... 805
Una nota sull'implementazione di entrambi i metodi ... 805
Gli appunti ... 805
Examples ... 806
Motivazione ... 806
Il problema ... 807
La soluzione (parte 1) ... 807
La soluzione (parte 2) ... 808
Informazioni su quelle funzioni duplicate 810
Sommario ... 810

Entrambi i metodi implementati, eval-round-trip style __repr __ () 811
Capitolo 166: Raspare Web con Python 812
introduzione 812
Osservazioni 812
Utili pacchetti Python per lo scraping web (in ordine alfabetico) 812
Fare richieste e raccogliere dati 812
requests 812
requests-cache 812
scrapy 812
selenium 812
Analisi HTML 812
BeautifulSoup 813
lxml 813
Examples 813
Esempio di base di utilizzo di richieste e lxml per raschiare alcuni dati 813
Mantenimento della sessione di web scraping con le richieste 813
Raschiare usando il framework Scrapy 814
Modifica agente utente Scrapy 814
Raschiare usando BeautifulSoup4 815
Raschiatura usando Selenium WebDriver 815
Download semplice di contenuti Web con urllib.request 815
Raschiando con arricciatura 816
Capitolo 167: Registrazione 817
Examples 817
Introduzione alla registrazione di Python 817
Registrazione delle eccezioni 818
Capitolo 168: Ricerca 821
Osservazioni 821
Examples 821
Ottenere l'indice per le stringhe: str.index (), str.rindex () e str.find (), str.rfind () 821
Alla ricerca di un elemento 821
Elenco 822

tuple ... 822
Stringa ... 822
Impostato ... 822
dict ... 822
Ottenere l'elenco degli indici e le tuple: list.index (), tuple.index () ... 822
Ricerca di chiavi (s) per un valore in dict ... 823
Ottenere l'indice per le sequenze ordinate: bisect.bisect_left () ... 823
Ricerca di sequenze nidificate ... 824
Ricerca in classi personalizzate: __contains__ e __iter__ ... 825
Capitolo 169: Richieste di richieste Python ... 827
introduzione ... 827
Examples ... 827
Post semplice ... 827
Form Dati codificati ... 828
Upload di file ... 829
risposte ... 829
Autenticazione ... 830
Proxy ... 831
Capitolo 170: Riconoscimento ottico dei caratteri ... 833
introduzione ... 833
Examples ... 833
PyTesseract ... 833
PyOCR ... 833
Capitolo 171: ricorsione ... 835
Osservazioni ... 835
Examples ... 835
Somma di numeri da 1 a n ... 835
Il cosa, come e quando della ricorsione ... 835
Esplorazione di alberi con ricorsione ... 839
Aumentare la profondità massima di ricorsione ... 840
Ricorsione di coda - Cattiva pratica ... 841
Ottimizzazione della ricorsione della coda attraverso l'introspezione della pila ... 841

Capitolo 172: Ridurre 843
Sintassi 843
Parametri 843
Osservazioni 843
Examples 843
Panoramica 843
Utilizzando ridurre 844
Prodotto cumulativo 845
Variante non a corto circuito di any / all 845
Primo elemento di verità / falsy di una sequenza (o ultimo elemento se non ce n'è) 845
Capitolo 173: Scrittura in formato CSV da stringa o elenco 846
introduzione 846
Parametri 846
Osservazioni 846
Examples 846
Esempio di scrittura di base 846
Aggiunta di una stringa come nuova riga in un file CSV 847
Capitolo 174: Scrivere estensioni 848
Examples 848
Ciao mondo con estensione C 848
Passare un file aperto alle estensioni C 849
Estensione C con c ++ e Boost 849
Codice C ++ 849
Capitolo 175: Secure Shell Connection in Python 851
Parametri 851
Examples 851
connessione ssh 851
Capitolo 176: Semplici operatori matematici 852
introduzione 852
Osservazioni 852
Tipi numerici e loro metaclassi 852

Examples 852
aggiunta 852
Sottrazione 853
Moltiplicazione 853
Divisione 854
Exponentation 856
Funzioni speciali 856
logaritmi 857
Operazioni interne 857
Funzioni trigonometriche 858
Modulo 858
Capitolo 177: Serializzazione dei dati 860
Sintassi 860
Parametri 860
Osservazioni 860
Examples 861
Serializzazione tramite JSON 861
Serializzazione tramite Pickle 861
Capitolo 178: Serializzazione dei dati sottaceti 863
Sintassi 863
Parametri 863
Osservazioni 863
Tipi pickleable 863
pickle e sicurezza 863
Examples 864
Usare Pickle per serializzare e deserializzare un oggetto 864
Per serializzare l'oggetto 864
Per deserializzare l'oggetto 864
Usando oggetti pickle e byte 864
Personalizza dati sottodimensionati 865
Capitolo 179: setup.py 867

Parametri ... 867
Osservazioni ... 867
Examples ... 867
Scopo di setup.py ... 867
Aggiunta di script da riga di comando al pacchetto python ... 868
Utilizzo dei metadati del controllo del codice sorgente in setup.py ... 869
Aggiunta di opzioni di installazione ... 869
Capitolo 180: Sicurezza e crittografia ... 871
introduzione ... 871
Sintassi ... 871
Osservazioni ... 871
Examples ... 871
Calcolo di un digest di messaggi ... 871
Algoritmi di hash disponibili ... 872
Hash password sicura ... 872
File Hashing ... 873
Crittografia simmetrica usando pycrypto ... 873
Generazione di firme RSA usando pycrypto ... 874
Crittografia asimmetrica RSA usando pycrypto ... 875
Capitolo 181: Socket e crittografia / decrittografia dei messaggi tra client e server ... 877
introduzione ... 877
Osservazioni ... 877
Examples ... 880
Implementazione lato server ... 880
Implementazione lato client ... 882
Capitolo 182: Sockets ... 885
introduzione ... 885
Parametri ... 885
Examples ... 885
Invio di dati tramite UDP ... 885
Ricezione di dati tramite UDP ... 885
Invio di dati tramite TCP ... 886

TCP Socket Server multi-threaded ... 887
Socket raw su Linux ... 888
Capitolo 183: Somiglianze nella sintassi, differenze di significato: Python vs. JavaScript ... 890
introduzione ... 890
Examples ... 890
`in` con le liste ... 890
Capitolo 184: Sottocomandi CLI con output di aiuto preciso ... 891
introduzione ... 891
Osservazioni ... 891
Examples ... 891
Modo nativo (senza librerie) ... 891
argparse (formattatore di aiuto predefinito) ... 892
argparse (formattatore di aiuto personalizzato) ... 893
Capitolo 185: Sovraccarico ... 895
Examples ... 895
Metodi Magici / Dunder ... 895
Tipi di contenitori e di sequenza ... 896
Tipi chiamabili ... 897
Gestione del comportamento non implementato ... 897
Sovraccarico dell'operatore ... 898
Capitolo 186: Strumento 2to3 ... 901
Sintassi ... 901
Parametri ... 901
Osservazioni ... 902
Examples ... 902
Uso di base ... 902
Unix ... 902
finestre ... 902
Unix ... 903
finestre ... 903
Capitolo 187: SYS ... 904

introduzione ... 904
Sintassi ... 904
Osservazioni ... 904
Examples ... 904
Argomenti della riga di comando ... 904
Nome dello script ... 904
Flusso di errore standard ... 905
Termina prematuramente il processo e restituisce un codice di uscita ... 905
Capitolo 188: tempfile NamedTemporaryFile ... 906
Parametri ... 906
Examples ... 906
Crea (e scrivi in) un file temporaneo persistente noto ... 906
Capitolo 189: Test unitario ... 908
Osservazioni ... 908
Examples ... 908
Test delle eccezioni ... 908
Funzioni di simulazione con unittest.mock.create_autospec ... 909
Test Setup e Teardown in un unestest.TestCase ... 910
Asserendo sulle eccezioni ... 911
Scegliere le asserzioni all'interno di Unittests ... 912
Test unitari con pytest ... 913
Capitolo 190: The Interpreter (Command Line Console) ... 917
Examples ... 917
Ottenere aiuto generale ... 917
Riferendosi all'ultima espressione ... 917
Apertura della console Python ... 918
La variabile PYTHONSTARTUP ... 918
Argomenti della riga di comando ... 918
Ottenere aiuto su un oggetto ... 919
Capitolo 191: Tipi di dati immutabili (int, float, str, tuple e frozensets) ... 921
Examples ... 921
I singoli caratteri delle stringhe non sono assegnabili ... 921

I singoli membri di Tuple non sono assegnabili ... 921
I Frozenset sono immutabili e non assegnabili ... 921
Capitolo 192: Tipi di dati Python ... 922
introduzione ... 922
Examples ... 922
Tipo di dati numerici ... 922
Tipo di dati stringa ... 922
Elenca il tipo di dati ... 922
Tuple Data Type ... 922
Dizionario Tipo di dati ... 923
Imposta i tipi di dati ... 923
Capitolo 193: Tipo Suggerimenti ... 924
Sintassi ... 924
Osservazioni ... 924
Examples ... 924
Tipi generici ... 924
Aggiunta di tipi a una funzione ... 924
Membri e metodi della classe ... 926
Variabili e attributi ... 926
namedtuple ... 927
Digita suggerimenti per gli argomenti delle parole chiave ... 927
Capitolo 194: Tkinter ... 928
introduzione ... 928
Osservazioni ... 928
Examples ... 928
Un'applicazione tkinter minimale ... 928
Geometry Manager ... 929
Posto ... 929
pacco ... 930
Griglia ... 930
Capitolo 195: Tracciare con Matplotlib ... 932
introduzione ... 932

Examples ... 932
Una trama semplice in Matplotlib ... 932
Aggiunta di più funzioni a un grafico semplice: etichette dell'asse, titolo, segni di grad ... 933
Realizzare trame multiple nella stessa figura per sovrapposizione simile a MATLAB ... 934
Creare più grafici nella stessa figura usando la sovrapposizione della trama con comandi d ... 935
Grafici con asse X comune ma asse Y diverso: utilizzo di twinx () ... 936
Grafici con asse Y comune e asse X diverso con twiny () ... 938
Capitolo 196: Trasformazione di Panda: operazioni preforme su gruppi e concatenare i risul ... 941
Examples ... 941
Trasformazione semplice ... 941
Innanzitutto, creiamo un dataframe fittizio ... 941
Ora, utilizzeremo la funzione di transform panda per contare il numero di ordini per clien ... 941
Più risultati per gruppo ... 942
Utilizzo delle funzioni di transform che restituiscono calcoli secondari per gruppo ... 942
Capitolo 197: tuple ... 944
introduzione ... 944
Sintassi ... 944
Osservazioni ... 944
Examples ... 944
Tuple di indicizzazione ... 944
Le tuple sono immutabili ... 945
Tuple Are Element-wise hashable ed Equitable ... 945
tuple ... 946
Imballaggio e disimballaggio Tuples ... 947
Inversione di elementi ... 948
Funzioni tuple integrate ... 948
Confronto ... 948
Lunghezza della tupla ... 949
Max di una tupla ... 949
Minimo di una tupla ... 949
Convertire una lista in tupla ... 949

Concatenazione di tupla 949
Capitolo 198: Unicode 951
Examples 951
Codifica e decodifica 951
Capitolo 199: Unicode e byte 952
Sintassi 952
Parametri 952
Examples 952
Nozioni di base 952
Unicode in byte 952
Byte per unicode 953
Gestione degli errori di codifica / decodifica 953
Codifica 954
decodifica 954
Morale 954
File I / O 954
Capitolo 200: Unzipping dei file 956
introduzione 956
Examples 956
Utilizzando Python ZipFile.extractall () per decomprimere un file ZIP 956
Usando Python TarFile.extractall () per decomprimere un tarball 956
Capitolo 201: urllib 957
Examples 957
HTTP GET 957
Python 2 957
Python 3 957
HTTP POST 958
Python 2 958
Python 3 958
Decodifica i byte ricevuti in base alla codifica del tipo di contenuto 958
Capitolo 202: Utilizzo dei loop all'interno delle funzioni 960

introduzione ... 960
Examples ... 960
Dichiarazione di ritorno all'interno del loop in una funzione ... 960
Capitolo 203: Utilizzo del modulo "pip": PyPI Package Manager ... 961
introduzione ... 961
Sintassi ... 961
Examples ... 962
Esempio di utilizzo di comandi ... 962
Gestire l'eccezione di ImportError ... 962
Forza installazione ... 963
Capitolo 204: Verifica dell'esistenza e delle autorizzazioni del percorso ... 964
Parametri ... 964
Examples ... 964
Esegui i controlli usando os.access ... 964
Capitolo 205: Visualizzazione dei dati con Python ... 966
Examples ... 966
matplotlib ... 966
Seaborn ... 967
MayaVi ... 970
Plotly ... 971
Capitolo 206: WebSockets ... 974
Examples ... 974
Eco semplice con aiohttp ... 974
Classe wrapper con aiohttp ... 974
Utilizzando Autobahn come una Websocket Factory ... 975
Titoli di coda ... 978

Di

Capitolo 1: Iniziare con Python Language

Osservazioni

Python è un linguaggio di programmazione ampiamente utilizzato. È:

- **Alto livello** : Python automatizza operazioni di basso livello come la gestione della memoria. Lascia il programmatore con un controllo un po 'meno ma ha molti vantaggi, tra cui la leggibilità del codice e le espressioni minime del codice.
- **Scopo generale** : Python è progettato per essere utilizzato in tutti i contesti e ambienti. Un esempio per un linguaggio non generico è PHP: è stato progettato specificatamente come linguaggio di scripting per lo sviluppo web lato server. Al contrario, Python *può* essere utilizzato per lo sviluppo web lato server, ma anche per la creazione di applicazioni desktop.
- **Digitazione dinamica** : ogni variabile in Python può fare riferimento a qualsiasi tipo di dati. Una singola espressione può valutare dati di tipi diversi in tempi diversi. A causa di ciò, è possibile il seguente codice:

  ```
  if something:
      x = 1
  else:
      x = 'this is a string'
  print(x)
  ```

- **Fortemente digitato** : durante l'esecuzione del programma, non ti è permesso fare nulla che sia incompatibile con il tipo di dati con cui stai lavorando. Ad esempio, non ci sono conversioni nascoste da stringhe a numeri; una stringa composta da cifre non verrà mai trattata come un numero se non la si converte esplicitamente:

  ```
  1 + '1'  # raises an error
  1 + int('1')  # results with 2
  ```

- **Beginner friendly :)** : la sintassi e la struttura di Python sono molto intuitive. È di alto livello e fornisce costrutti volti a consentire la scrittura di programmi chiari sia su piccola che su larga scala. Python supporta diversi paradigmi di programmazione, compresi gli stili di programmazione orientati agli oggetti, imperativi e funzionali o procedurali. Ha una libreria standard ampia e completa e molte librerie di terze parti facili da installare.

I suoi principi di progettazione sono delineati in *The Zen of Python* .

Attualmente ci sono due principali rami di rilascio di Python che presentano alcune differenze significative. Python 2.x è la versione legacy sebbene veda ancora un uso diffuso. Python 3.x crea un insieme di modifiche incompatibili all'indietro che mirano a ridurre la duplicazione delle

funzionalità. Per assistenza per decidere quale versione è la migliore per te, consulta questo articolo .

La documentazione ufficiale di Python è anche una risorsa completa e utile, contenente documentazione per tutte le versioni di Python e tutorial per aiutarti a iniziare.

Esiste un'implementazione ufficiale della lingua fornita da Python.org, generalmente denominata CPython, e diverse implementazioni alternative del linguaggio su altre piattaforme di runtime. Questi includono IronPython (eseguendo Python sulla piattaforma .NET), Jython (sul runtime Java) e PyPy (implementando Python in un sottoinsieme di se stesso).

Versioni

Python 3.x

Versione	Data di rilascio
[3.7]	2017/05/08
3.6	2016/12/23
3.5	2015/09/13
3.4	2014/03/17
3.3	2012/09/29
3.2	2011-02-20
3.1	2009-06-26
3.0	2008-12-03

Python 2.x

Versione	Data di rilascio
2.7	2010-07-03
2.6	2008-10-02
2.5	2006-09-19
2.4	2004-11-30
2.3	2003/07/29

Versione	Data di rilascio
2.2	2001/12/21
2.1	2001/04/15
2.0	2000/10/16

Examples

Iniziare

Python è un linguaggio di programmazione di alto livello ampiamente utilizzato per la programmazione generica, creato da Guido van Rossum e rilasciato per la prima volta nel 1991. Python dispone di un sistema di tipo dinamico e gestione automatica della memoria e supporta molteplici paradigmi di programmazione, incluso l'imperativo orientato agli oggetti programmazione funzionale e stili procedurali. Ha una libreria standard ampia e completa.

Due versioni principali di Python sono attualmente in uso:

- Python 3.x è la versione corrente ed è in fase di sviluppo attivo.
- Python 2.x è la versione legacy e riceverà solo aggiornamenti di sicurezza fino al 2020. Non saranno implementate nuove funzionalità. Nota che molti progetti usano ancora Python 2, anche se la migrazione a Python 3 sta diventando più semplice.

Puoi scaricare e installare entrambe le versioni di Python qui . Vedi Python 3 vs Python 2 per un confronto tra di loro. Inoltre, alcune terze parti offrono versioni re-pacchettizzate di Python che aggiungono librerie e altre funzionalità di uso comune per semplificare l'installazione per casi di uso comune, come matematica, analisi dei dati o uso scientifico. Vedi la lista sul sito ufficiale .

Verifica se Python è installato

Per verificare che Python sia stato installato correttamente, è possibile verificarlo eseguendo il seguente comando nel terminale preferito (se si utilizza il sistema operativo Windows, è necessario aggiungere il percorso di python alla variabile di ambiente prima di utilizzarlo nel prompt dei comandi):

```
$ python --version
```

Python 3.x 3.0

Se hai installato *Python 3* ed è la tua versione di default (vedi **Risoluzione** dei **problemi** per maggiori dettagli) dovresti vedere qualcosa di simile a questo:

```
$ python --version
Python 3.6.0
```

Python 2.x 2.7

Se hai installato *Python 2* ed è la tua versione di default (vedi **Risoluzione** dei **problemi** per maggiori dettagli) dovresti vedere qualcosa di simile a questo:

```
$ python --version
Python 2.7.13
```

Se hai installato Python 3, ma `$ python --version` restituisce una versione di Python 2, hai anche Python 2 installato. Questo è spesso il caso di MacOS e di molte distribuzioni Linux. Usa `$ python3` invece di usare esplicitamente l'interprete Python 3.

Ciao, World in Python usando IDLE

IDLE è un semplice editor per Python, che viene fornito in bundle con Python.

Come creare il programma Hello, World in IDLE

- Apri IDLE sul tuo sistema preferito.
  - Nelle versioni precedenti di Windows, può essere trovato in `All Programs` dal menu di Windows.
  - In Windows 8+, cerca `IDLE` o `IDLE` nelle app presenti nel tuo sistema.
  - Sui sistemi basati su Unix (incluso Mac) è possibile aprirlo dalla shell digitando `$ idle python_file.py` .
- Si aprirà una shell con opzioni lungo la parte superiore.

Nella shell, vi è un prompt di tre parentesi angolari:

```
>>>
```

Ora scrivi il seguente codice nel prompt:

```
>>> print("Hello, World")
```

Premi `Invio` .

```
>>> print("Hello, World")
Hello, World
```

Ciao file World Python

Crea un nuovo file `hello.py` che contiene la seguente riga:

Python 3.x 3.0

```
print('Hello, World')
```

Python 2.x 2.6

Puoi utilizzare la funzione di `print` Python 3 in Python 2 con la seguente dichiarazione di `import` :

```
from __future__ import print_function
```

Python 2 ha un numero di funzionalità che possono essere facoltativamente importate da Python 3 usando il modulo `__future__` , come discusso qui .

Python 2.x 2.7

Se usi Python 2, puoi anche digitare la riga qui sotto. Si noti che questo non è valido in Python 3 e quindi non raccomandato in quanto riduce la compatibilità del codice cross-version.

```
print 'Hello, World'
```

Nel tuo terminale, vai alla directory contenente il file `hello.py` .

`python hello.py` , quindi premi il tasto `Invio` .

```
$ python hello.py
Hello, World
```

Dovresti vedere `Hello, World` stampato sulla console.

Puoi anche sostituire `hello.py` con il percorso del tuo file. Ad esempio, se hai il file nella tua home directory e il tuo utente è "utente" su Linux, puoi digitare `python /home/user/hello.py` .

Avvia una shell Python interattiva

Eseguendo (eseguendo) il comando `python` nel tuo terminale, ti viene presentata una shell Python interattiva. Questo è anche noto come Interprete Python o REPL (per 'Read Evaluate Print Loop').

```
$ python
Python 2.7.12 (default, Jun 28 2016, 08:46:01)
[GCC 6.1.1 20160602] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print 'Hello, World'
Hello, World
>>>
```

Se vuoi eseguire Python 3 dal tuo terminale, esegui il comando `python3` .

```
$ python3
Python 3.6.0 (default, Jan 13 2017, 00:00:00)
[GCC 6.1.1 20160602] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print('Hello, World')
Hello, World
>>>
```

In alternativa, avviare il prompt interattivo e caricare il file con `python -i <file.py>` .

Nella riga di comando, esegui:

```
$ python -i hello.py
"Hello World"
>>>
```

Esistono diversi modi per chiudere la shell Python:

```
>>> exit()
```

o

```
>>> quit()
```

In alternativa, `CTRL + D` chiuderà la shell e ti rimetterà sulla riga di comando del tuo terminale.

Se vuoi annullare un comando sei nel bel mezzo della digitazione e tornare a un prompt dei comandi pulito, mentre stai all'interno della shell Interpreter, usa `CTRL + C`.

Prova una shell Python interattiva online .

Altre conchiglie online

Vari siti web forniscono accesso online alle shell Python.

Le shell online possono essere utili per i seguenti scopi:

- Esegui un piccolo snippet di codice da una macchina che manca dell'installazione python (smartphone, tablet, ecc.).
- Impara o insegna Basic Python.
- Risolvi i problemi del giudice online.

Esempi:

> Dichiarazione di non responsabilità: gli autori della documentazione non sono affiliati con le risorse elencate di seguito.

- https://www.python.org/shell/ - La shell Python online ospitata dal sito Web ufficiale di Python.
- https://ideone.com/ - Ampiamente utilizzato in rete per illustrare il comportamento del frammento di codice.
- https://repl.it/languages/python3 - Compilatore, IDE e interprete online potente e semplice. Codice, compilazione ed esecuzione di codice in Python.
- https://www.tutorialspoint.com/execute_python_online.php - Una shell UNIX completa e un esploratore di progetti user-friendly.
- http://rextester.com/l/python3_online_compiler - IDE semplice e facile da usare che mostra il

tempo di esecuzione

Esegui i comandi come una stringa

Python può essere passato codice arbitrario come una stringa nella shell:

```
$ python -c 'print("Hello, World")'
Hello, World
```

Questo può essere utile quando si concatenano i risultati degli script insieme nella shell.

Conchiglie e oltre

Gestione dei pacchetti - Lo strumento consigliato da PyPA per installare i pacchetti Python è PIP . Per installare, nella riga di comando eseguire `pip install <the package name>` . Ad esempio, `pip install numpy` . (Nota: su Windows è necessario aggiungere pip alle variabili di ambiente PATH. Per evitare ciò, utilizzare `python -m pip install <the package name>`)

Conchiglie - Finora, abbiamo discusso diversi modi per eseguire codice usando la shell interattiva nativa di Python. I gusci usano il potere interpretativo di Python per sperimentare il codice in tempo reale. Le shell alternative includono IDLE , una GUI preconnessa , IPython , nota per l'estensione dell'esperienza interattiva e così via.

Programmi : per la memorizzazione a lungo termine è possibile salvare il contenuto in file .py e modificarli / eseguirli come script o programmi con strumenti esterni, ad esempio shell, IDE (come PyCharm), quaderni Jupyter , ecc. Gli utenti intermedi possono utilizzare questi strumenti; tuttavia, i metodi discussi qui sono sufficienti per iniziare.

Il tutor Python ti consente di scorrere il codice Python in modo da poter visualizzare il flusso del programma e aiutarti a capire dove il programma è andato storto.

PEP8 definisce le linee guida per la formattazione del codice Python. La formattazione del codice è importante per poter leggere rapidamente ciò che fa il codice.

Creazione di variabili e assegnazione di valori

Per creare una variabile in Python, tutto ciò che devi fare è specificare il nome della variabile, e quindi assegnargli un valore.

```
<variable name> = <value>
```

Python usa `=` per assegnare valori alle variabili. Non è necessario dichiarare una variabile in anticipo (o assegnargli un tipo di dati), assegnare un valore a una variabile stessa dichiara e inizializza la variabile con quel valore. Non c'è modo di dichiarare una variabile senza assegnargli un valore iniziale.

```
# Integer
a = 2
print(a)
# Output: 2

# Integer
b = 9223372036854775807
print(b)
# Output: 9223372036854775807

# Floating point
pi = 3.14
print(pi)
# Output: 3.14

# String
c = 'A'
print(c)
# Output: A

# String
name = 'John Doe'
print(name)
# Output: John Doe

# Boolean
q = True
print(q)
# Output: True

# Empty value or null data type
x = None
print(x)
# Output: None
```

L'assegnazione variabile funziona da sinistra a destra. Quindi il seguente ti darà un errore di sintassi.

```
0 = x
=> Output: SyntaxError: can't assign to literal
```

Non è possibile utilizzare le parole chiave di python come nome di una variabile valida. Puoi vedere l'elenco delle parole chiave per:

```
import keyword
print(keyword.kwlist)
```

Regole per la denominazione delle variabili:

1. I nomi delle variabili devono iniziare con una lettera o un trattino basso.

```
 x  = True   # valid
 _y = True   # valid

 9x = False  # starts with numeral
```

```
 => SyntaxError: invalid syntax

 $y = False #  starts with symbol
 => SyntaxError: invalid syntax
```

2. Il resto del nome della variabile può essere composto da lettere, numeri e caratteri di sottolineatura.

```
has_0_in_it = "Still Valid"
```

3. I nomi sono case sensitive.

```
x = 9
y = X*5
=>NameError: name 'X' is not defined
```

Anche se non c'è bisogno di specificare un tipo di dati quando si dichiara una variabile in Python, mentre assegnando lo spazio necessario in memoria per la variabile, l'interprete Python preleva automaticamente il più adatto tipo incorporato per esso:

```
a = 2
print(type(a))
# Output: <type 'int'>

b = 9223372036854775807
print(type(b))
# Output: <type 'int'>

pi = 3.14
print(type(pi))
# Output: <type 'float'>

c = 'A'
print(type(c))
# Output: <type 'str'>

name = 'John Doe'
print(type(name))
# Output: <type 'str'>

q = True
print(type(q))
# Output: <type 'bool'>

x = None
print(type(x))
# Output: <type 'NoneType'>
```

Ora conosci le basi del compito, prendi questa sottigliezza sull'assegnazione in python.

Quando si usa = per eseguire un'operazione di assegnazione, ciò che si trova a sinistra di = è un ***nome*** per l' ***oggetto*** sulla destra. Infine, what = does assegna il ***riferimento*** dell'oggetto a destra al ***nome*** a sinistra.

Questo è:

```
a_name = an_object  # "a_name" is now a name for the reference to the object "an_object"
```

Quindi, da molti esempi di assegnazione sopra, se selezioniamo `pi = 3.14` , allora `pi` è **un** nome (non **il** nome, poiché un oggetto può avere più nomi) per l'oggetto `3.14` . Se non capisci qualcosa di seguito, torna a questo punto e rileggilo! Inoltre, puoi dare un'occhiata a questo per una migliore comprensione.

È possibile assegnare più valori a più variabili in una riga. Si noti che ci deve essere lo stesso numero di argomenti sui lati destro e sinistro dell'operatore `=` :

```
a, b, c = 1, 2, 3
print(a, b, c)
# Output: 1 2 3

a, b, c = 1, 2
=> Traceback (most recent call last):
=>   File "name.py", line N, in <module>
=>     a, b, c = 1, 2
=> ValueError: need more than 2 values to unpack

a, b = 1, 2, 3
=> Traceback (most recent call last):
=>   File "name.py", line N, in <module>
=>     a, b = 1, 2, 3
=> ValueError: too many values to unpack
```

L'errore nell'ultimo esempio può essere eliminato assegnando i valori rimanenti a un numero uguale di variabili arbitrarie. Questa variabile fittizia può avere qualsiasi nome, ma è normale utilizzare il carattere di sottolineatura (_) per l'assegnazione di valori indesiderati:

```
a, b, _ = 1, 2, 3
print(a, b)
# Output: 1, 2
```

Si noti che il numero di _ e il numero di valori rimanenti devono essere uguali. Altrimenti 'troppi valori per decomprimere l'errore' vengono lanciati come sopra:

```
a, b, _ = 1,2,3,4
=>Traceback (most recent call last):
=>File "name.py", line N, in <module>
=>a, b, _ = 1,2,3,4
=>ValueError: too many values to unpack (expected 3)
```

È anche possibile assegnare un singolo valore a più variabili contemporaneamente.

```
a = b = c = 1
print(a, b, c)
# Output: 1 1 1
```

Quando si utilizza tale assegnazione a cascata, è importante notare che tutte e tre le variabili `a` , `b` e `c` riferiscono allo *stesso oggetto* in memoria, un oggetto `int` con il valore di 1. In altre parole, `a` , `b` e `c` sono tre nomi diversi dato allo stesso oggetto int. Assegnare un oggetto diverso a uno di essi in seguito non cambia gli altri, proprio come previsto:

```
a = b = c = 1    # all three names a, b and c refer to same int object with value 1
print(a, b, c)
# Output: 1 1 1
b = 2            # b now refers to another int object, one with a value of 2
print(a, b, c)
# Output: 1 2 1  # so output is as expected.
```

Quanto sopra vale anche per i tipi mutabili (come `list` , `dict` , ecc.) Così come è vero per i tipi immutabili (come `int` , `string` , `tuple` , ecc.):

```
x = y = [7, 8, 9]   # x and y refer to the same list object just created, [7, 8, 9]
x = [13, 8, 9]      # x now refers to a different list object just created, [13, 8, 9]
print(y)            # y still refers to the list it was first assigned
# Output: [7, 8, 9]
```

Fin qui tutto bene. Le cose sono un po 'diverse quando si tratta di *modificare* l'oggetto (diversamente *dall'assegnazione* del nome a un oggetto diverso, che abbiamo fatto sopra) quando l'assegnazione a cascata è usata per i tipi mutabili. Dai uno sguardo qui sotto e lo vedrai in prima persona:

```
x = y = [7, 8, 9]     # x and y are two different names for the same list object just created,
[7, 8, 9]
x[0] = 13             # we are updating the value of the list [7, 8, 9] through one of its
names, x in this case
print(y)              # printing the value of the list using its other name
# Output: [13, 8, 9]  # hence, naturally the change is reflected
```

Gli elenchi annidati sono validi anche in python. Ciò significa che un elenco può contenere un altro elenco come elemento.

```
x = [1, 2, [3, 4, 5], 6, 7] # this is nested list
print x[2]
# Output: [3, 4, 5]
print x[2][1]
# Output: 4
```

Infine, le variabili in Python non devono rimanere dello stesso tipo di quelle per cui sono state definite per la prima volta: puoi semplicemente usare `=` per assegnare un nuovo valore a una variabile, anche se quel valore è di un tipo diverso.

```
a = 2
print(a)
# Output: 2

a = "New value"
print(a)
```

```
# Output: New value
```

Se questo ti infastidisce, pensa al fatto che ciò che è a sinistra di `=` è solo un nome per un oggetto. Prima si chiama l'oggetto `int` con valore 2 `a` , quindi si cambia idea e si decide di assegnare il nome `a` a un oggetto `string` , con valore 'Nuovo valore'. Semplice, vero?

Input dell'utente

Input interattivo

Per ottenere input dall'utente, utilizzare la funzione di `input` (**nota** : in Python 2.x, la funzione è chiamata `raw_input` , sebbene Python 2.x abbia una propria versione di `input` completamente diversa):

Python 2.x 2.3

```
name = raw_input("What is your name? ")
# Out: What is your name? _
```

> **Nota di sicurezza** Non usare `input()` in Python2 - il testo inserito verrà valutato come se fosse un'espressione Python (equivalente a `eval(input())` in Python3), che potrebbe facilmente diventare una vulnerabilità. Vedi questo articolo per ulteriori informazioni sui rischi dell'utilizzo di questa funzione.

Python 3.x 3.0

```
name = input("What is your name? ")
# Out: What is your name? _
```

Il resto di questo esempio userà la sintassi di Python 3.

La funzione accetta un argomento stringa, che lo mostra come un prompt e restituisce una stringa. Il codice precedente fornisce un prompt, in attesa che l'utente inserisca.

```
name = input("What is your name? ")
# Out: What is your name?
```

Se l'utente digita "Bob" e gli hit entrano, il `name` della variabile verrà assegnato alla stringa `"Bob"` :

```
name = input("What is your name? ")
# Out: What is your name? Bob
print(name)
# Out: Bob
```

Si noti che l' `input` è sempre di tipo `str` , che è importante se si desidera che l'utente inserisca numeri. Pertanto, è necessario convertire lo `str` prima di provare ad usarlo come numero:

```
x = input("Write a number:")
# Out: Write a number: 10
x / 2
```

```
# Out: TypeError: unsupported operand type(s) for /: 'str' and 'int'
float(x) / 2
# Out: 5.0
```

NB: Si consiglia di utilizzare `try` / `except` blocks per rilevare le eccezioni quando si gestiscono gli input dell'utente . Ad esempio, se il tuo codice vuole lanciare `raw_input` in un `int` , e ciò che l'utente scrive è invalicabile, solleva un'eccezione `ValueError` .

IDLE - GUI Python

IDLE è l'ambiente di sviluppo e apprendimento integrato di Python ed è un'alternativa alla linea di comando. Come suggerisce il nome, IDLE è molto utile per lo sviluppo di un nuovo codice o per l'apprendimento di python. Su Windows viene fornito con l'interprete Python, ma in altri sistemi operativi potrebbe essere necessario installarlo tramite il gestore pacchetti.

Gli scopi principali di IDLE sono:

- Editor di testo multi-finestra con evidenziazione della sintassi, completamento automatico e rientro intelligente
- Shell Python con evidenziazione della sintassi
- Debugger integrato con stepping, punti di interruzione persistenti e visibilità dello stack delle chiamate
- Indentazione automatica (utile per i principianti che apprendono il rientro di Python)
- Salvare il programma Python come file .py ed eseguirli e modificarli successivamente in qualsiasi momento usando IDLE.

In IDLE, premi `F5` o `run Python Shell` per lanciare un interprete. L'utilizzo di IDLE può essere un'esperienza di apprendimento migliore per i nuovi utenti perché il codice viene interpretato come l'utente scrive.

Nota che ci sono molte alternative, vedi ad esempio questa discussione o questo elenco .

Risoluzione dei problemi

- **finestre**

  Se sei su Windows, il comando predefinito è `python` . Se si riceve un errore `"'python' is not recognized"` , la causa più probabile è che la posizione di Python non si trova nella variabile di ambiente `PATH` del sistema. È possibile accedervi facendo clic con il pulsante destro del mouse su "Risorse del computer" e selezionando "Proprietà" o spostandosi su "Sistema" tramite "Pannello di controllo". Fare clic su "Impostazioni di sistema avanzate" e quindi su "Variabili d'ambiente ...". Modifica la variabile `PATH` per includere la directory della tua installazione Python, così come la cartella Script (solitamente `C:\Python27;C:\Python27\Scripts`). Ciò richiede privilegi amministrativi e potrebbe richiedere un riavvio.

  Quando si utilizzano più versioni di Python sulla stessa macchina, una possibile soluzione è

rinominare uno dei file `python.exe` . Ad esempio, nominare una versione `python27.exe` potrebbe far diventare `python27` il comando Python per quella versione.

Puoi anche utilizzare Python Launcher per Windows, che è disponibile tramite il programma di installazione e viene fornito per impostazione predefinita. Permette di selezionare la versione di Python da eseguire usando `py -[xy]` invece di `python[xy]` . Puoi usare l'ultima versione di Python 2 eseguendo script con `py -2` e l'ultima versione di Python 3 eseguendo script con `py -3` .

- **Debian / Ubuntu / MacOS**

  Questa sezione presuppone che la posizione dell'eseguibile `python` sia stata aggiunta alla variabile d'ambiente `PATH` .

  Se sei su Debian / Ubuntu / MacOS, apri il terminale e digita `python` per Python 2.xo `python3` per Python 3.x.

  Digita `which python` per vedere quale interprete Python verrà usato.

- **Arch Linux**

  Il predefinito Python su Arch Linux (e discendenti) è Python 3, quindi usa `python` o `python3` per Python 3.xe `python2` per Python 2.x.

- **Altri sistemi**

  Python 3 è talvolta associato a `python` anziché a `python3` . Per usare Python 2 su questi sistemi su cui è installato, puoi usare `python2` .

Tipi di dati

Tipi incorporati

booleani

`bool` : un valore booleano di `True` o `False` . Le operazioni logiche come `and` , `or` , `not` possono essere eseguite sui booleani.

```
x or y    # if x is False then y otherwise x
x and y   # if x is False then x otherwise y
not x     # if x is True then False, otherwise True
```

In Python 2.xe in Python 3.x, un booleano è anche un `int` . Il tipo di `bool` è una sottoclasse del tipo `int` e `True` e `False` sono le sue uniche istanze:

```
issubclass(bool, int) # True

isinstance(True, bool) # True
```

```
isinstance(False, bool) # True
```

Se i valori booleani vengono utilizzati nelle operazioni aritmetiche, i loro valori interi (`1` e `0` per `True` e `False`) verranno utilizzati per restituire un risultato intero:

```
True + False == 1 # 1 + 0 == 1
True * True  == 1 # 1 * 1 == 1
```

Numeri

- `int` : numero intero

  ```
  a = 2
  b = 100
  c = 123456789
  d = 38563846326424324
  ```

  Gli interi in Python hanno dimensioni arbitrarie.

  Nota: nelle versioni precedenti di Python era disponibile un tipo `long` diverso da `int` . I due sono stati unificati.

- `float` : numero in virgola mobile; la precisione dipende dall'implementazione e dall'architettura di sistema, per CPython il tipo di dati `float` corrisponde a un doppio C.

  ```
  a = 2.0
  b = 100.e0
  c = 123456789.e1
  ```

- `complex` : numeri complessi

  ```
  a = 2 + 1j
  b = 100 + 10j
  ```

Gli operatori `<` , `<=` , `>` e `>=` `TypeError` un'eccezione `TypeError` quando qualsiasi operando è un numero complesso.

stringhe

Python 3.x 3.0

- `str` : una **stringa unicode** . Il tipo di `'hello'`
- `bytes` : una **stringa di byte** . Il tipo di `b'hello'`

Python 2.x 2.7

- `str` : una **stringa di byte** . Il tipo di `'hello'`
- `bytes` : sinonimo di `str`

- `unicode` : una **stringa unicode** . Il tipo di `u'hello'`

Sequenze e collezioni

Python distingue tra sequenze ordinate e raccolte non ordinate (come `set` e `dict`).

- le stringhe (`str` , `bytes` , `unicode`) sono sequenze
- `reversed` : un ordine inverso di `str` con funzione `reversed`

```
a = reversed('hello')
```

- `tuple` : una raccolta ordinata di `n` valori di qualsiasi tipo (`n >= 0`).

```
a = (1, 2, 3)
b = ('a', 1, 'python', (1, 2))
b[2] = 'something else' # returns a TypeError
```

Supporta l'indicizzazione; immutabile; lavabile se tutti i suoi membri sono lavabili

- `list` : una raccolta ordinata di `n` valori (`n >= 0`)

```
a = [1, 2, 3]
b = ['a', 1, 'python', (1, 2), [1, 2]]
b[2] = 'something else' # allowed
```

Non lavabile; mutevole.

- `set` : una collezione non ordinata di valori unici. Gli articoli devono essere lavabili .

```
a = {1, 2, 'a'}
```

- `dict` : una collezione non ordinata di coppie chiave-valore uniche; le chiavi devono essere lavabili

```
a = {1: 'one',
     2: 'two'}

b = {'a': [1, 2, 3],
     'b': 'a string'}
```

Un oggetto è lavabile se ha un valore di hash che non cambia mai durante la sua vita (ha bisogno di un `__hash__()`) e può essere paragonato ad altri oggetti (ha bisogno di un `__eq__()`). Gli oggetti lavabili che confrontano l'uguaglianza devono avere lo stesso valore di hash.

Costanti incorporate

In combinazione con i tipi di dati incorporati c'è un piccolo numero di costanti incorporate nello spazio dei nomi integrato:

- `True` : il vero valore del `bool`
- `False` : il valore falso del tipo built-in `bool`
- `None` : un oggetto singleton utilizzato per segnalare che un valore è assente.
- `Ellipsis` o `...` : utilizzato in core Python3 + ovunque e limitato l'uso in Python2.7 + come parte della notazione array. `numpy` e pacchetti correlati usano questo come riferimento "include tutto" nelle matrici.
- `NotImplemented` : un singleton utilizzato per indicare a Python che un metodo speciale non supporta gli argomenti specifici e Python proverà delle alternative se disponibili.

```
a = None # No value will be assigned. Any valid datatype can be assigned later
```

Python 3.x 3.0

`None` non ha ordini naturali. L'utilizzo degli operatori di confronto degli ordini (`<` , `<=` , `>=` , `>`) non è più supportato e genererà un `TypeError` .

Python 2.x 2.7

`None` è sempre inferiore a qualsiasi numero (`None < -32 True`).

Test del tipo di variabili

In python, possiamo controllare il tipo di dati di un oggetto usando il `type` funzione built-in.

```
a = '123'
print(type(a))
# Out: <class 'str'>
b = 123
print(type(b))
# Out: <class 'int'>
```

Nelle istruzioni condizionali è possibile testare il tipo di dati con `isinstance` . Tuttavia, di solito non è incoraggiato a fare affidamento sul tipo di variabile.

```
i = 7
if isinstance(i, int):
    i += 1
elif isinstance(i, str):
    i = int(i)
    i += 1
```

Per informazioni sulle differenze tra `type()` e `isinstance()` leggi: Differenze tra isinstance e type in Python

Per verificare se qualcosa è di `NoneType` :

```
x = None
if x is None:
    print('Not a surprise, I just defined x as None.')
```

Conversione tra tipi di dati

È possibile eseguire la conversione del tipo di dati esplicita.

Ad esempio, '123' è di tipo `str` e può essere convertito in intero usando la funzione `int` .

```
a = '123'
b = int(a)
```

La conversione da una stringa mobile come "123.456" può essere eseguita usando la funzione `float` .

```
a = '123.456'
b = float(a)
c = int(a)    # ValueError: invalid literal for int() with base 10: '123.456'
d = int(b)    # 123
```

Puoi anche convertire i tipi di sequenza o raccolta

```
a = 'hello'
list(a)  # ['h', 'e', 'l', 'l', 'o']
set(a)   # {'o', 'e', 'l', 'h'}
tuple(a) # ('h', 'e', 'l', 'l', 'o')
```

Tipo di stringa esplicito alla definizione di valori letterali

Con le etichette di una sola lettera di fronte alle virgolette puoi dire quale tipo di corda vuoi definire.

- `b'foo bar'` : risultati `bytes` in Python 3, `str` in Python 2
- `u'foo bar'` : risultati `str` in Python 3, `unicode` in Python 2
- `'foo bar'` : risultati `str`
- `r'foo bar'` : i risultati della cosiddetta stringa raw, in cui non sono necessari caratteri speciali di escape, tutto è preso alla lettera mentre hai digitato

```
normal  = 'foo\nbar'   # foo
                       # bar
escaped = 'foo\\nbar'  # foo\nbar
raw     = r'foo\nbar'  # foo\nbar
```

Tipi di dati mutevoli e immutabili

Un oggetto è chiamato *mutevole* se può essere cambiato. Ad esempio, quando si passa una lista a una funzione, l'elenco può essere modificato:

```
def f(m):
    m.append(3)  # adds a number to the list. This is a mutation.

x = [1, 2]
f(x)
x == [1, 2]  # False now, since an item was added to the list
```

Un oggetto è chiamato *immutabile* se non può essere modificato in alcun modo. Ad esempio, gli interi sono immutabili, poiché non c'è modo di cambiarli:

```
def bar():
    x = (1, 2)
    g(x)
    x == (1, 2)  # Will always be True, since no function can change the object (1, 2)
```

Nota che le **variabili** stesse sono mutabili, quindi possiamo riassegnare la *variabile* `x` , ma questo non cambia l'oggetto che `x` aveva precedentemente indicato. Ha fatto solo `x` puntare a un nuovo oggetto.

I tipi di dati le cui istanze sono mutabili sono chiamati *tipi di dati mutabili* e, analogamente, per oggetti e tipi di dati immutabili.

Esempi di tipi di dati immutabili:

- `int` , `long` , `float` , `complex`
- `str`
- `bytes`
- `tuple`
- `frozenset`

Esempi di tipi di dati mutabili:

- `bytearray`
- `list`
- `set`
- `dict`

Moduli e funzioni integrati

Un modulo è un file contenente definizioni e definizioni Python. La funzione è un pezzo di codice che esegue una logica.

```
>>> pow(2,3)    #8
```

Per controllare la funzione integrata in python possiamo usare `dir()` . Se chiamato senza un

argomento, restituisce i nomi nell'ambito corrente. Altrimenti, restituisci un elenco alfabetico di nomi che comprende (alcuni) l'attributo dell'oggetto dato e degli attributi raggiungibili da esso.

```
>>> dir(__builtins__)
[
    'ArithmeticError',
    'AssertionError',
    'AttributeError',
    'BaseException',
    'BufferError',
    'BytesWarning',
    'DeprecationWarning',
    'EOFError',
    'Ellipsis',
    'EnvironmentError',
    'Exception',
    'False',
    'FloatingPointError',
    'FutureWarning',
    'GeneratorExit',
    'IOError',
    'ImportError',
    'ImportWarning',
    'IndentationError',
    'IndexError',
    'KeyError',
    'KeyboardInterrupt',
    'LookupError',
    'MemoryError',
    'NameError',
    'None',
    'NotImplemented',
    'NotImplementedError',
    'OSError',
    'OverflowError',
    'PendingDeprecationWarning',
    'ReferenceError',
    'RuntimeError',
    'RuntimeWarning',
    'StandardError',
    'StopIteration',
    'SyntaxError',
    'SyntaxWarning',
    'SystemError',
    'SystemExit',
    'TabError',
    'True',
    'TypeError',
    'UnboundLocalError',
    'UnicodeDecodeError',
    'UnicodeEncodeError',
    'UnicodeError',
    'UnicodeTranslateError',
    'UnicodeWarning',
    'UserWarning',
    'ValueError',
    'Warning',
    'ZeroDivisionError',
    '__debug__',
    '__doc__',
```

```
    '__import__',
    '__name__',
    '__package__',
    'abs',
    'all',
    'any',
    'apply',
    'basestring',
    'bin',
    'bool',
    'buffer',
    'bytearray',
    'bytes',
    'callable',
    'chr',
    'classmethod',
    'cmp',
    'coerce',
    'compile',
    'complex',
    'copyright',
    'credits',
    'delattr',
    'dict',
    'dir',
    'divmod',
    'enumerate',
    'eval',
    'execfile',
    'exit',
    'file',
    'filter',
    'float',
    'format',
    'frozenset',
    'getattr',
    'globals',
    'hasattr',
    'hash',
    'help',
    'hex',
    'id',
    'input',
    'int',
    'intern',
    'isinstance',
    'issubclass',
    'iter',
    'len',
    'license',
    'list',
    'locals',
    'long',
    'map',
    'max',
    'memoryview',
    'min',
    'next',
    'object',
    'oct',
    'open',
```

```
    'ord',
    'pow',
    'print',
    'property',
    'quit',
    'range',
    'raw_input',
    'reduce',
    'reload',
    'repr',
    'reversed',
    'round',
    'set',
    'setattr',
    'slice',
    'sorted',
    'staticmethod',
    'str',
    'sum',
    'super',
    'tuple',
    'type',
    'unichr',
    'unicode',
    'vars',
    'xrange',
    'zip'
]
```

Per conoscere la funzionalità di qualsiasi funzione, siamo in grado di utilizzare costruito in funzione di `help` .

```
>>> help(max)
Help on built-in function max in module __builtin__:
max(...)
    max(iterable[, key=func]) -> value
    max(a, b, c, ...[, key=func]) -> value
    With a single iterable argument, return its largest item.
    With two or more arguments, return the largest argument.
```

I moduli integrati contengono funzionalità aggiuntive. Ad esempio, per ottenere la radice quadrata di un numero, è necessario includere il modulo `math` .

```
>>> import math
>>> math.sqrt(16) # 4.0
```

Per conoscere tutte le funzioni di un modulo, è possibile assegnare l'elenco delle funzioni a una variabile, quindi stampare la variabile.

```
>>> import math
>>> dir(math)

   ['__doc__', '__name__', '__package__', 'acos', 'acosh',
   'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign',
   'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1',
   'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma',
```

```
   'hypot', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10',
   'log1p', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt',
   'tan', 'tanh', 'trunc']
```

sembra che `__doc__` sia utile per fornire documentazione, ad esempio, sulle funzioni

```
>>> math.__doc__
'This module is always available.  It provides access to the\nmathematical
 functions defined by the C standard.'
```

Oltre alle funzioni, la documentazione può essere fornita anche in moduli. Quindi, se hai un file chiamato `helloWorld.py` questo modo:

```
"""This is the module docstring."""

def sayHello():
    """This is the function docstring."""
    return 'Hello World'
```

Puoi accedere alle sue docstring in questo modo:

```
>>> import helloWorld
>>> helloWorld.__doc__
'This is the module docstring.'
>>> helloWorld.sayHello.__doc__
'This is the function docstring.'
```

- Per qualsiasi tipo definito dall'utente, i suoi attributi, gli attributi della sua classe e in modo ricorsivo gli attributi delle classi base della sua classe possono essere recuperati usando dir ()

```
>>> class MyClassObject(object):
...     pass
...
>>> dir(MyClassObject)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__', '__getattribute__',
'__hash__', '__init__', '__module__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__']
```

Qualsiasi tipo di dati può essere semplicemente convertito in stringa usando una funzione incorporata chiamata `str` . Questa funzione è chiamata per impostazione predefinita quando viene passato un tipo di dati per la `print`

```
>>> str(123)    # "123"
```

Blocco indentazione

Python utilizza il rientro per definire i costrutti di controllo e di loop. Ciò contribuisce alla leggibilità di Python, tuttavia richiede al programmatore di prestare molta attenzione all'uso degli spazi bianchi. Pertanto, l'errata taratura dell'editor potrebbe comportare un codice che si comporta in

modi imprevisti.

Python utilizza il simbolo due punti (`:`) e rientro per mostrare dove blocchi di codice iniziano e finiscono (Se si proviene da un'altra lingua, non confondere questo con qualche modo essere legato alla operatore ternario). Cioè, i blocchi in Python, come le funzioni, i cicli, le clausole `if` e altri costrutti, non hanno identificatori finali. Tutti i blocchi iniziano con due punti e quindi contengono le linee rientrate sotto di esso.

Per esempio:

```
def my_function():    # This is a function definition. Note the colon (:)
    a = 2             # This line belongs to the function because it's indented
    return a          # This line also belongs to the same function
print(my_function())  # This line is OUTSIDE the function block
```

o

```
if a > b:             # If block starts here
    print(a)          # This is part of the if block
else:                 # else must be at the same level as if
    print(b)          # This line is part of the else block
```

I blocchi che contengono esattamente un'istruzione a riga singola possono essere messi sulla stessa riga, sebbene questo modulo non sia generalmente considerato di buon livello:

```
if a > b: print(a)
else: print(b)
```

Il tentativo di eseguire questa operazione con più di una singola istruzione *non* funzionerà:

```
if x > y: y = x
    print(y) # IndentationError: unexpected indent

if x > y: while y != z: y -= 1  # SyntaxError: invalid syntax
```

Un blocco vuoto causa un `IndentationError` . Usa `pass` (un comando che non fa nulla) quando hai un blocco senza contenuto:

```
def will_be_implemented_later():
    pass
```

Spazi contro schede

In breve: usa **sempre** 4 spazi per il rientro.

L'uso esclusivo delle schede è possibile ma PEP 8 , la guida di stile per il codice Python, afferma che gli spazi sono preferiti.

Python 3.x 3.0

Python 3 non consente di mescolare l'uso di tabulazioni e spazi per il rientro. In tal caso viene generato un errore in fase di compilazione: `Inconsistent use of tabs and spaces in indentation` e il programma non verrà eseguito.

Python 2.x 2.7

Python 2 consente di mescolare tab e spazi in indentazione; questo è fortemente scoraggiato. Il carattere di tabulazione completa il rientro precedente per essere un multiplo di **8** spazi . Poiché è comune che gli editor siano configurati per mostrare le schede come multipli di **4** spazi, ciò può causare bug sottili.

Citando PEP 8 :

> Quando si richiama l'interprete della riga di comando Python 2 con l'opzione `-t` , emette avvisi sul codice che mescola illegalmente tab e spazi. Quando si utilizza `-tt` questi avvisi diventano errori. Queste opzioni sono altamente raccomandate!

Molti editor hanno la configurazione "tabs to spaces". Quando si configura l'editor, è necessario distinguere tra il *carattere di* tabulazione ('\ t') e il tasto `Tab` .

- Il *carattere di* tabulazione deve essere configurato per mostrare 8 spazi, in modo che corrispondano alla semantica della lingua, almeno nei casi in cui è possibile indentazione mista (accidentale). Gli editor possono anche convertire automaticamente il carattere di tabulazione in spazi.
- Tuttavia, potrebbe essere utile configurare l'editor in modo che premendo il tasto `Tab` si inseriscano 4 spazi, invece di inserire un carattere di tabulazione.

Il codice sorgente Python scritto con una combinazione di schede e spazi o con un numero non standard di spazi di indentazione può essere reso conforme a pep8 usando autopep8 . (Un'alternativa meno potente viene fornita con la maggior parte delle installazioni Python: reindent.py)

Tipi di raccolta

Esistono numerosi tipi di raccolta in Python. Mentre tipi come `int` e `str` mantengono un singolo valore, i tipi di raccolta contengono più valori.

elenchi

Il tipo di `list` è probabilmente il tipo di raccolta più comunemente utilizzato in Python. Nonostante il suo nome, una lista è più simile a un array in altre lingue, principalmente JavaScript. In Python, una lista è semplicemente una raccolta ordinata di valori Python validi. Un elenco può essere creato racchiudendo valori, separati da virgole, tra parentesi quadre:

```
int_list = [1, 2, 3]
string_list = ['abc', 'defghi']
```

Una lista può essere vuota:

```
empty_list = []
```

Gli elementi di un elenco non sono limitati a un singolo tipo di dati, il che ha senso dato che Python è un linguaggio dinamico:

```
mixed_list = [1, 'abc', True, 2.34, None]
```

Una lista può contenere un altro elenco come suo elemento:

```
nested_list = [['a', 'b', 'c'], [1, 2, 3]]
```

Gli elementi di una lista sono accessibili tramite un *indice* o una rappresentazione numerica della loro posizione. Gli elenchi in Python sono *indicizzati a zero,* ovvero il primo elemento nell'elenco è all'indice 0, il secondo elemento è all'indice 1 e così via:

```
names = ['Alice', 'Bob', 'Craig', 'Diana', 'Eric']
print(names[0]) # Alice
print(names[2]) # Craig
```

Gli indici possono anche essere negativi, il che significa contare dalla fine della lista (`-1` è l'indice dell'ultimo elemento). Quindi, usando l'elenco dell'esempio precedente:

```
print(names[-1]) # Eric
print(names[-4]) # Bob
```

Le liste sono mutabili, quindi puoi cambiare i valori in una lista:

```
names[0] = 'Ann'
print(names)
# Outputs ['Ann', 'Bob', 'Craig', 'Diana', 'Eric']
```

Inoltre, è possibile aggiungere e / o rimuovere elementi da una lista:

Aggiungi oggetto alla fine della lista con `L.append(object)` , restituisce `None` .

```
names = ['Alice', 'Bob', 'Craig', 'Diana', 'Eric']
names.append("Sia")
print(names)
# Outputs ['Alice', 'Bob', 'Craig', 'Diana', 'Eric', 'Sia']
```

Aggiungi un nuovo elemento da elencare in un indice specifico. `L.insert(index, object)`

```
names.insert(1, "Nikki")
print(names)
# Outputs ['Alice', 'Nikki', 'Bob', 'Craig', 'Diana', 'Eric', 'Sia']
```

Rimuovi la prima occorrenza di un valore con `L.remove(value)` , restituisce `None`

```
names.remove("Bob")
```

```
print(names) # Outputs ['Alice', 'Nikki', 'Craig', 'Diana', 'Eric', 'Sia']
```

Ottieni l'indice nell'elenco del primo elemento il cui valore è x. Mostrerà un errore se non esiste tale elemento.

```
name.index("Alice")
0
```

Conta la lunghezza della lista

```
len(names)
6
```

conta il verificarsi di qualsiasi elemento nell'elenco

```
a = [1, 1, 1, 2, 3, 4]
a.count(1)
3
```

Invertire la lista

```
a.reverse()
[4, 3, 2, 1, 1, 1]
# or
a[::-1]
[4, 3, 2, 1, 1, 1]
```

Rimuovi e restituisci l'oggetto all'indice (predefinito all'ultima voce) con `L.pop([index])` , restituisce l'elemento

```
names.pop() # Outputs 'Sia'
```

Puoi scorrere gli elementi dell'elenco come di seguito:

```
for element in my_list:
    print (element)
```

Le tuple

Una `tuple` è simile a una lista tranne che è a lunghezza fissa e immutabile. Pertanto, i valori nella tupla non possono essere modificati, né i valori possono essere aggiunti o rimossi dalla tupla. Le tuple vengono comunemente utilizzate per piccole raccolte di valori che non dovranno essere modificate, ad esempio un indirizzo IP e una porta. Le tuple sono rappresentate con parentesi invece di parentesi quadre:

```
ip_address = ('10.20.30.40', 8080)
```

Le stesse regole di indicizzazione per le liste si applicano anche alle tuple. Le tuple possono anche essere annidate e i valori possono essere validi per qualsiasi Python valido.

Una tupla con un solo membro deve essere definita (notare la virgola) in questo modo:

```
one_member_tuple = ('Only member',)
```

o

```
one_member_tuple = 'Only member',   # No brackets
```

o semplicemente usando la sintassi della `tuple`

```
one_member_tuple = tuple(['Only member'])
```

dizionari

Un `dictionary` in Python è una raccolta di coppie chiave-valore. Il dizionario è circondato da parentesi graffe. Ogni coppia è separata da una virgola e la chiave e il valore sono separati da due punti. Ecco un esempio:

```
state_capitals = {
    'Arkansas': 'Little Rock',
    'Colorado': 'Denver',
    'California': 'Sacramento',
    'Georgia': 'Atlanta'
}
```

Per ottenere un valore, fare riferimento ad esso con la sua chiave:

```
ca_capital = state_capitals['California']
```

Puoi anche ottenere tutte le chiavi in un dizionario e poi scorrere su di esse:

```
for k in state_capitals.keys():
    print('{} is the capital of {}'.format(state_capitals[k], k))
```

I dizionari assomigliano molto alla sintassi JSON. Il modulo `json` nativo nella libreria standard Python può essere usato per convertire tra JSON e dizionari.

impostato

Un `set` è una raccolta di elementi senza ripetizioni e senza ordine di inserimento ma ordine ordinato. Sono utilizzati in situazioni in cui è importante solo che alcune cose siano raggruppate insieme e non in quale ordine siano state incluse. Per gruppi di dati di grandi dimensioni, è molto più veloce verificare se un elemento è presente in un `set` piuttosto che fare lo stesso per un `list` .

Definire un `set` è molto simile alla definizione di un `dictionary` :

```
first_names = {'Adam', 'Beth', 'Charlie'}
```

Oppure puoi costruire un `set` usando un `list` esistente:

```
my_list = [1,2,3]
my_set = set(my_list)
```

Controlla l'appartenenza al `set` usando `in` :

```
if name in first_names:
    print(name)
```

È possibile scorrere su un `set` esattamente come un elenco, ma ricorda: i valori saranno in un ordine arbitrario definito dall'implementazione.

defaultdict

Un `defaultdict` è un dizionario con un valore predefinito per le chiavi, in modo che le chiavi per le quali non è stato definito alcun valore siano accessibili senza errori. `defaultdict` è particolarmente utile quando i valori nel dizionario sono raccolte (elenchi, dict, ecc.) nel senso che non è necessario inizializzarli ogni volta che si utilizza una nuova chiave.

Un `defaultdict` non genererà mai un KeyError. Qualsiasi chiave che non esiste ottiene il valore predefinito restituito.

Ad esempio, considera il seguente dizionario

```
>>> state_capitals = {
    'Arkansas': 'Little Rock',
    'Colorado': 'Denver',
    'California': 'Sacramento',
    'Georgia': 'Atlanta'
}
```

Se proviamo ad accedere a una chiave inesistente, python ci restituisce un errore come segue

```
>>> state_capitals['Alabama']
Traceback (most recent call last):

  File "<ipython-input-61-236329695e6f>", line 1, in <module>
    state_capitals['Alabama']

KeyError: 'Alabama'
```

Proviamo con un `defaultdict` . Può essere trovato nel modulo delle collezioni.

```
>>> from collections import defaultdict
>>> state_capitals = defaultdict(lambda: 'Boston')
```

Quello che abbiamo fatto qui è di impostare un valore predefinito (**Boston**) nel caso in cui la chiave give non esiste. Ora compila il ditt come prima:

```
>>> state_capitals['Arkansas'] = 'Little Rock'
>>> state_capitals['California'] = 'Sacramento'
>>> state_capitals['Colorado'] = 'Denver'
```

```
>>> state_capitals['Georgia'] = 'Atlanta'
```

Se proviamo ad accedere al dict con una chiave inesistente, python ci restituirà il valore predefinito, ovvero Boston

```
>>> state_capitals['Alabama']
'Boston'
```

e restituisce i valori creati per la chiave esistente proprio come un normale `dictionary`

```
>>> state_capitals['Arkansas']
'Little Rock'
```

Aiuto Utility

Python ha diverse funzioni integrate nell'interprete. Se si desidera ottenere informazioni su parole chiave, funzioni, moduli o argomenti incorporati, aprire una console Python e immettere:

```
>>> help()
```

Riceverai informazioni inserendo le parole chiave direttamente:

```
>>> help(help)
```

o all'interno dell'utilità:

```
help> help
```

che mostrerà una spiegazione:

```
Help on _Helper in module _sitebuiltins object:

class _Helper(builtins.object)
 |  Define the builtin 'help'.
 |
 |  This is a wrapper around pydoc.help that provides a helpful message
 |  when 'help' is typed at the Python interactive prompt.
 |
 |  Calling help() at the Python prompt starts an interactive help session.
 |  Calling help(thing) prints help for the python object 'thing'.
 |
 |  Methods defined here:
 |
 |  __call__(self, *args, **kwds)
 |
 |  __repr__(self)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
```

```
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

Puoi anche richiedere sottoclassi di moduli:

```
help(pymysql.connections)
```

Puoi utilizzare la guida per accedere alle docstring dei diversi moduli che hai importato, ad esempio, prova quanto segue:

```
>>> help(math)
```

e avrai un errore

```
>>> import math
>>> help(math)
```

E ora riceverai un elenco dei metodi disponibili nel modulo, ma solo DOPO averlo importato.

Chiudete l'aiutante con `quit`

Creare un modulo

Un modulo è un file importabile contenente definizioni e istruzioni.

Un modulo può essere creato creando un file `.py` .

```
# hello.py
def say_hello():
    print("Hello!")
```

Le funzioni di un modulo possono essere utilizzate importando il modulo.

Per i moduli che hai creato, dovranno essere nella stessa directory del file in cui li stai importando. (Tuttavia, è anche possibile inserirli nella directory lib Python con i moduli pre-inclusi, ma se possibile dovrebbe essere evitato.)

```
$ python
>>> import hello
>>> hello.say_hello()
=> "Hello!"
```

I moduli possono essere importati da altri moduli.

```
# greet.py
import hello
hello.say_hello()
```

È possibile importare funzioni specifiche di un modulo.

```
# greet.py
from hello import say_hello
say_hello()
```

I moduli possono essere di tipo alias.

```
# greet.py
import hello as ai
ai.say_hello()
```

Un modulo può essere uno script eseguibile autonomo.

```
# run_hello.py
if __name__ == '__main__':
    from hello import say_hello
    say_hello()
```

Eseguirlo!

```
$ python run_hello.py
=> "Hello!"
```

Se il modulo si trova in una directory e deve essere rilevato da python, la directory deve contenere un file denominato `__init__.py` .

Funzione stringa - str () e repr ()

Esistono due funzioni che possono essere utilizzate per ottenere una rappresentazione leggibile di un oggetto.

`repr(x)` chiama `x.__repr__()` : una rappresentazione di `x` . `eval` solito converte il risultato di questa funzione nell'oggetto originale.

`str(x)` chiama `x.__str__()` : una stringa leggibile dall'uomo che descrive l'oggetto. Questo può escludere alcuni dettagli tecnici.

---

repr ()

Per molti tipi, questa funzione fa un tentativo di restituire una stringa che produce un oggetto con lo stesso valore quando viene passato a `eval()` . Altrimenti, la rappresentazione è una stringa racchiusa tra parentesi angolari che contiene il nome del tipo dell'oggetto insieme a informazioni aggiuntive. Questo spesso include il nome e l'indirizzo dell'oggetto.

str ()

Per le stringhe, restituisce la stringa stessa. La differenza tra questo e `repr(object)` è che

`str(object)` non tenta sempre di restituire una stringa accettabile da `eval()` . Piuttosto, il suo obiettivo è restituire una stringa stampabile o 'leggibile dall'uomo'. Se non viene fornito alcun argomento, restituisce la stringa vuota, `''` .

Esempio 1:

```
s = """w'o"w"""
repr(s) # Output: '\'w\\\'o"w\''
str(s)  # Output: 'w\'o"w'
eval(str(s)) == s  # Gives a SyntaxError
eval(repr(s)) == s # Output: True
```

Esempio 2:

```
import datetime
today = datetime.datetime.now()
str(today)  # Output: '2016-09-15 06:58:46.915000'
repr(today) # Output: 'datetime.datetime(2016, 9, 15, 6, 58, 46, 915000)'
```

Quando scrivi un corso, puoi sovrascrivere questi metodi per fare quello che vuoi:

```
class Represent(object):

    def __init__(self, x, y):
        self.x, self.y = x, y

    def __repr__(self):
        return "Represent(x={},y=\"{}\")".format(self.x, self.y)

    def __str__(self):
        return "Representing x as {} and y as {}".format(self.x, self.y)
```

Usando la classe sopra possiamo vedere i risultati:

```
r = Represent(1, "Hopper")
print(r)  # prints __str__
print(r.__repr__)  # prints __repr__: '<bound method Represent.__repr__ of
Represent(x=1,y="Hopper")>'
rep = r.__repr__()  # sets the execution of __repr__ to a new variable
print(rep)  # prints 'Represent(x=1,y="Hopper")'
r2 = eval(rep) # evaluates rep
print(r2)  # prints __str__ from new object
print(r2 == r)  # prints 'False' because they are different objects
```

Installazione di moduli esterni tramite pip

`pip` è tuo amico quando hai bisogno di installare qualsiasi pacchetto dalla pletora di scelte disponibili nel python package index (PyPI). `pip` è già installato se usi Python 2> = 2.7.9 o Python 3> = 3.4 scaricato da python.org. Per i computer che eseguono Linux o un altro * nix con un gestore di pacchetti nativo, i `pip` devono spesso essere installati manualmente.

Nelle istanze con Python 2 e Python 3 installati, `pip` fa spesso riferimento a Python 2 e `pip3` a

Python 3. Utilizzando `pip` verranno installati solo pacchetti per Python 2 e `pip3` installerà solo pacchetti per Python 3.

Trovare / installare un pacchetto

Cercare un pacchetto è semplice come digitare

```
$ pip search <query>
# Searches for packages whose name or summary contains <query>
```

Installare un pacchetto è semplice come digitare *(in un terminale / prompt dei comandi, non nell'interprete Python)*

```
$ pip install [package_name]           # latest version of the package

$ pip install [package_name]==x.x.x    # specific version of the package

$ pip install '[package_name]>=x.x.x'  # minimum version of the package
```

dove `xxx` è il numero di versione del pacchetto che si desidera installare.

Quando il tuo server è dietro proxy, puoi installare il pacchetto usando il seguente comando:

```
$ pip --proxy http://<server address>:<port> install
```

Aggiornamento dei pacchetti installati

Quando vengono visualizzate nuove versioni dei pacchetti installati, non vengono installati automaticamente sul sistema. Per avere una panoramica di quali dei tuoi pacchetti installati sono diventati obsoleti, esegui:

```
$ pip list --outdated
```

Per aggiornare un pacchetto specifico utilizzare

```
$ pip install [package_name] --upgrade
```

L'aggiornamento di tutti i pacchetti obsoleti non è una funzionalità standard di `pip` .

Aggiornamento pip

È possibile aggiornare l'installazione del pip esistente utilizzando i seguenti comandi

- Su Linux o macOS X:

```
$ pip install -U pip
```

Potrebbe essere necessario utilizzare `sudo` con pip su alcuni sistemi Linux

- Su Windows:

```
py -m pip install -U pip
```

o

```
python -m pip install -U pip
```

Per maggiori informazioni riguardo pip leggi qui .

Installazione di Python 2.7.xe 3.x

> **Nota** : le seguenti istruzioni sono scritte per Python 2.7 (se non specificato): le istruzioni per Python 3.x sono simili.

FINESTRE

Innanzitutto, scarica l'ultima versione di Python 2.7 dal sito web ufficiale (https://www.python.org/downloads/) . La versione è fornita come pacchetto MSI. Per installarlo manualmente, basta fare doppio clic sul file.

Di default, Python si installa in una directory:

```
C:\Python27\
```

Attenzione: l'installazione non modifica automaticamente la variabile d'ambiente PATH.

Supponendo che la tua installazione Python sia in C: \ Python27, aggiungi questo al tuo PATH:

```
C:\Python27\;C:\Python27\Scripts\
```

Ora per verificare se l'installazione di Python è valida scrivi in cmd:

```
python --version
```

Python 2.xe 3.x Side-by-Side

Per installare e utilizzare sia Python 2.x che 3.x side-by-side su una macchina Windows:

1. Installa Python 2.x utilizzando il programma di installazione MSI.
    - Assicurati che Python sia installato per tutti gli utenti.
    - Opzionale: aggiungi Python a `PATH` per rendere Python 2.x callable dalla riga di comando usando `python` .

2. Installa Python 3.x utilizzando il rispettivo programma di installazione.

    - Di nuovo, assicurati che Python sia installato per tutti gli utenti.
    - Opzionale: aggiungi Python a `PATH` per rendere Python 3.x callable dalla riga di comando usando `python` . Ciò potrebbe ignorare le impostazioni del `PATH` Python 2.x, quindi ricontrolla il `PATH` e assicurati che sia configurato in base alle tue preferenze.
    - Assicurati di installare `py launcher` per tutti gli utenti.

Python 3 installerà il launcher Python che può essere usato per lanciare Python 2.xe Python 3.x in modo intercambiabile dalla riga di comando:

```
P:\>py -3
Python 3.6.1 (v3.6.1:69c0db5, Mar 21 2017, 17:54:52) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>

C:\>py -2
Python 2.7.13 (v2.7.13:a06454b1afa1, Dec 17 2016, 20:42:59) [MSC v.1500 32 Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Per utilizzare la versione corrispondente di `pip` per una specifica versione di Python, utilizzare:

```
C:\>py -3 -m pip -V
pip 9.0.1 from C:\Python36\lib\site-packages (python 3.6)

C:\>py -2 -m pip -V
pip 9.0.1 from C:\Python27\lib\site-packages (python 2.7)
```

LINUX

Le ultime versioni di CentOS, Fedora, RedHat Enterprise (RHEL) e Ubuntu sono fornite con Python 2.7.

Per installare Python 2.7 su Linux manualmente, basta effettuare le seguenti operazioni nel terminale:

```
wget --no-check-certificate https://www.python.org/ftp/python/2.7.X/Python-2.7.X.tgz
tar -xzf Python-2.7.X.tgz
cd Python-2.7.X
./configure
make
sudo make install
```

Aggiungete anche il percorso di new python nella variabile d'ambiente PATH. Se new python è in `/root/python-2.7.X` allora esegui `export PATH = $PATH:/root/python-2.7.X`

Ora per verificare se l'installazione di Python è valida, scrivi nel terminale:

```
python --version
```

Ubuntu (dalla sorgente)

Se hai bisogno di Python 3.6 puoi installarlo dal sorgente come mostrato di seguito (Ubuntu 16.10 e 17.04 hanno versione 3.6 nel repository universale). Di seguito i passaggi da seguire per Ubuntu 16.04 e versioni precedenti:

```
sudo apt install build-essential checkinstall
sudo apt install libreadline-gplv2-dev libncursesw5-dev libssl-dev libsqlite3-dev tk-dev
libgdbm-dev libc6-dev libbz2-dev
wget https://www.python.org/ftp/python/3.6.1/Python-3.6.1.tar.xz
tar xvf Python-3.6.1.tar.xz
cd Python-3.6.1/
./configure --enable-optimizations
sudo make altinstall
```

Mac OS

Mentre parliamo, macOS viene installato con Python 2.7.10, ma questa versione è obsoleta e leggermente modificata dal normale Python.

> La versione di Python fornita con OS X è ottima per l'apprendimento ma non va bene per lo sviluppo. La versione fornita con OS X potrebbe non essere aggiornata rispetto all'attuale versione ufficiale di Python, che è considerata la versione di produzione stabile. (fonte)

Installa Homebrew :

```
/usr/bin/ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Installa Python 2.7:

```
brew install python
```

Per Python 3.x, usa invece il comando `brew install python3` .

Leggi Iniziare con Python Language online: https://riptutorial.com/it/python/topic/193/iniziare-con-python-language

Capitolo 2: * args e ** kwargs

Osservazioni

Ci sono alcune cose da notare:

1. I nomi `args` e `kwargs` sono usati per convenzione, non fanno parte delle specifiche del linguaggio. Quindi, questi sono equivalenti:

```
def func(*args, **kwargs):
    print(args)
    print(kwargs)
```

```
def func(*a, **b):
    print(a)
    print(b)
```

2. Potresti non avere più di un `args` o più di un parametro `kwargs` (tuttavia non sono necessari)

```
def func(*args1, *args2):
#   File "<stdin>", line 1
#     def test(*args1, *args2):
#                      ^
# SyntaxError: invalid syntax
```

```
def test(**kwargs1, **kwargs2):
#   File "<stdin>", line 1
#     def test(**kwargs1, **kwargs2):
#                        ^
# SyntaxError: invalid syntax
```

3. Se qualche argomento posizionale segue `*args` , sono argomenti solo per parole chiave che possono essere passati solo per nome. Una singola stella può essere usata al posto di `*args` per forzare i valori come argomenti della parola chiave senza fornire una lista di parametri variadici. Gli elenchi di parametri solo per parole chiave sono disponibili solo in Python 3.

```
def func(a, b, *args, x, y):
    print(a, b, args, x, y)

func(1, 2, 3, 4, x=5, y=6)
#>>> 1, 2, (3, 4), 5, 6
```

```
def func(a, b, *, x, y):
    print(a, b, x, y)
```

```
func(1, 2, x=5, y=6)
#>>> 1, 2, 5, 6
```

4. `**kwargs` deve arrivare per ultimo nell'elenco dei parametri.

```
def test(**kwargs, *args):
#   File "<stdin>", line 1
#     def test(**kwargs, *args):
#                        ^
# SyntaxError: invalid syntax
```

Examples

Usare * args durante la scrittura di funzioni

Puoi usare la stella * quando scrivi una funzione per raccogliere tutti gli argomenti posizionali (cioè senza nome) in una tupla:

```
def print_args(farg, *args):
   print("formal arg: %s" % farg)
   for arg in args:
       print("another positional arg: %s" % arg)
```

Metodo di chiamata:

```
print_args(1, "two", 3)
```

In questa chiamata, farg verrà assegnato come sempre, e gli altri due verranno inseriti nella tupla degli args, nell'ordine in cui sono stati ricevuti.

Usare ** kwargs durante la scrittura di funzioni

È possibile definire una funzione che accetta un numero arbitrario di argomenti di parole chiave (denominati) utilizzando la doppia stella `**` prima del nome di un parametro:

```
def print_kwargs(**kwargs):
    print(kwargs)
```

Quando chiama il metodo, Python costruirà un dizionario di tutti gli argomenti delle parole chiave e lo renderà disponibile nel corpo della funzione:

```
print_kwargs(a="two", b=3)
# prints: "{a: "two", b=3}"
```

Si noti che il parametro ** kwargs nella definizione della funzione deve sempre essere l'ultimo parametro e corrisponderà solo agli argomenti passati dopo quelli precedenti.

```
def example(a, **kw):
    print kw

example(a=2, b=3, c=4) # => {'b': 3, 'c': 4}
```

All'interno del corpo della funzione, `kwargs` viene manipolato allo stesso modo di un dizionario; per poter accedere ai singoli elementi in `kwargs` fai semplicemente scorrere come faresti con un normale dizionario:

```
def print_kwargs(**kwargs):
    for key in kwargs:
        print("key = {0}, value = {1}".format(key, kwargs[key]))
```

Ora, chiamando `print_kwargs(a="two", b=1)` mostra il seguente output:

```
print_kwargs(a = "two", b = 1)
key = a, value = "two"
key = b, value = 1
```

Usare * args quando si chiamano le funzioni

Un caso di uso comune per `*args` in una definizione di funzione è delegare l'elaborazione a una funzione di tipo wrapper o ereditata. Un tipico esempio potrebbe essere nel metodo `__init__` una classe

```
class A(object):
    def __init__(self, b, c):
        self.y = b
        self.z = c

class B(A):
    def __init__(self, a, *args, **kwargs):
        super(B, self).__init__(*args, **kwargs)
        self.x = a
```

Qui, il `a` parametro viene elaborato dalla classe figlio dopo tutti gli altri argomenti (posizionali e parole chiave) sono passati su - ed elaborati dal - la classe di base.

Per esempio:

```
b = B(1, 2, 3)
b.x  # 1
b.y  # 2
b.z  # 3
```

Quello che succede qui è la funzione di classe `B` `__init__` vede gli argomenti `1, 2, 3` . Sa che ha bisogno di prendere un argomento posizionale (`a`), quindi afferra il primo argomento passato in (`1`), quindi nell'ambito della funzione `a == 1` .

Successivamente, vede che è necessario prendere un numero arbitrario di argomenti posizionali (`*args`) in modo che tenga il resto degli argomenti posizionali passati in (`1, 2`) e li inserisca in

`*args` . Ora (nell'ambito della funzione) `args == [2, 3]` .

Quindi, chiama la funzione `__init__` classe `A` con `*args` . Python vede `*` di fronte a arg e "decomprime" la lista in argomenti. In questo esempio, quando la classe `B` 's `__init__` funzione chiama classe `A` ' s `__init__` funzione, ma passa gli argomenti `2, 3` (cioè `A(2, 3)`).

Infine, imposta la propria proprietà `x` sul primo argomento posizionale `a` , che equivale a `1` .

Usare ** kwargs quando si chiamano le funzioni

È possibile utilizzare un dizionario per assegnare valori ai parametri della funzione; utilizzando i parametri nome come chiavi nel dizionario e il valore di questi argomenti associati a ogni chiave:

```
def test_func(arg1, arg2, arg3): # Usual function with three arguments
   print("arg1: %s" % arg1)
   print("arg2: %s" % arg2)
   print("arg3: %s" % arg3)

# Note that dictionaries are unordered, so we can switch arg2 and arg3. Only the names matter.
kwargs = {"arg3": 3, "arg2": "two"}

# Bind the first argument (ie. arg1) to 1, and use the kwargs dictionary to bind the others
test_var_args_call(1, **kwargs)
```

Usare * args quando si chiamano le funzioni

L'effetto dell'utilizzo dell'operatore `*` su un argomento quando si chiama una funzione è quello di decomprimere l'elenco o un argomento di tupla

```
def print_args(arg1, arg2):
    print(str(arg1) + str(arg2))

a = [1,2]
b = tuple([3,4])

print_args(*a)
# 12
print_args(*b)
# 34
```

Si noti che la lunghezza dell'argomento speciale deve essere uguale al numero degli argomenti della funzione.

Un idioma python comune è quello di utilizzare l'operatore di disimballaggio `*` con la funzione `zip` per invertire i suoi effetti:

```
a = [1,3,5,7,9]
b = [2,4,6,8,10]

zipped = zip(a,b)
# [(1,2), (3,4), (5,6), (7,8), (9,10)]

zip(*zipped)
```

```
# (1,3,5,7,9), (2,4,6,8,10)
```

Argomenti richiesti solo per parola chiave e parola chiave

Python 3 consente di definire argomenti di funzione che possono essere assegnati solo da parole chiave, anche senza valori predefiniti. Questo viene fatto usando la stella * per consumare ulteriori parametri posizionali senza impostare i parametri della parola chiave. Tutti gli argomenti dopo * sono argomenti con parole chiave (cioè non posizionali). Si noti che se agli argomenti con parole chiave non viene assegnato un valore predefinito, sono comunque necessari quando si chiama la funzione.

```
def print_args(arg1, *args, keyword_required, keyword_only=True):
    print("first positional arg: {}".format(arg1))
    for arg in args:
        print("another positional arg: {}".format(arg))
    print("keyword_required value: {}".format(keyword_required))
    print("keyword_only value: {}".format(keyword_only))

print(1, 2, 3, 4) # TypeError: print_args() missing 1 required keyword-only argument:
'keyword_required'
print(1, 2, 3, keyword_required=4)
# first positional arg: 1
# another positional arg: 2
# another positional arg: 3
# keyword_required value: 4
# keyword_only value: True
```

Popolazione dei valori di kwarg con un dizionario

```
def foobar(foo=None, bar=None):
    return "{}{}".format(foo, bar)

values = {"foo": "foo", "bar": "bar"}

foobar(**values) # "foobar"
```

** kwargs e valori predefiniti

Per utilizzare i valori predefiniti con ** kwargs

```
def fun(**kwargs):
    print kwargs.get('value', 0)

fun()
# print 0
fun(value=1)
# print 1
```

Leggi * args e ** kwargs online: https://riptutorial.com/it/python/topic/2475/--args-e----kwargs

Capitolo 3: Abstract Base Classes (abc)

Examples

Impostazione del metaclass ABCMeta

Le classi astratte sono classi che devono essere ereditate ma evitare l'implementazione di metodi specifici, lasciando solo le firme dei metodi che le sottoclassi devono implementare.

Le classi astratte sono utili per definire e applicare le astrazioni di classe ad un livello elevato, simile al concetto di interfacce nei linguaggi tipizzati, senza la necessità di implementazione del metodo.

Un approccio concettuale alla definizione di una classe astratta consiste nello stub dei metodi di classe, quindi genera un NotImplementedError se vi si accede. Questo impedisce alle classi di bambini di accedere ai metodi padre senza sovrascriverli prima. Così:

```
class Fruit:

    def check_ripeness(self):
        raise NotImplementedError("check_ripeness method not implemented!")


class Apple(Fruit):
    pass


a = Apple()
a.check_ripeness() # raises NotImplementedError
```

La creazione di una classe astratta in questo modo impedisce l'uso improprio di metodi che non sono sovrascritti, e certamente incoraggia i metodi da definire nelle classi figlie, ma non impone la loro definizione. Con il modulo `abc` possiamo impedire l'istanziazione delle classi figlie quando non riescono a scavalcare i metodi astratti di classe dei loro genitori e antenati:

```
from abc import ABCMeta

class AbstractClass(object):
    # the metaclass attribute must always be set as a class variable
    __metaclass__ = ABCMeta

   # the abstractmethod decorator registers this method as undefined
   @abstractmethod
   def virtual_method_subclasses_must_define(self):
       # Can be left completely blank, or a base implementation can be provided
       # Note that ordinarily a blank interpretation implicitly returns `None`,
       # but by registering, this behaviour is no longer enforced.
```

Ora è possibile semplicemente eseguire la sottoclasse e l'override:

```
class Subclass(AbstractClass):
    def virtual_method_subclasses_must_define(self):
        return
```

Perché / Come usare ABCMeta e @abstractmethod

Le astratte classi base (ABC) impongono le classi derivate che implementano particolari metodi dalla classe base.

Per capire come funziona e perché dovremmo usarlo, diamo un'occhiata a un esempio che Van Rossum avrebbe apprezzato. Diciamo che abbiamo una classe Base "MontyPython" con due metodi (joke & punchline) che devono essere implementati da tutte le classi derivate.

```
class MontyPython:
    def joke(self):
        raise NotImplementedError()

    def punchline(self):
        raise NotImplementedError()

class ArgumentClinic(MontyPython):
    def joke(self):
        return "Hahahahahah"
```

Quando istanziamo un oggetto e chiamiamo i suoi due metodi, avremo un errore (come previsto) con il metodo `punchline()` .

```
 >>> sketch = ArgumentClinic()
 >>> sketch.punchline()
NotImplementedError
```

Tuttavia, questo ci consente comunque di creare un'istanza di un oggetto della classe ArgumentClinic senza ottenere un errore. In realtà non otteniamo un errore finché non cerchiamo la battuta ().

Ciò viene evitato usando il modulo Abstract Base Class (ABC). Vediamo come funziona con lo stesso esempio:

```
from abc import ABCMeta, abstractmethod

class MontyPython(metaclass=ABCMeta):
    @abstractmethod
    def joke(self):
        pass

@abstractmethod
def punchline(self):
    pass

class ArgumentClinic(MontyPython):
    def joke(self):
        return "Hahahahahah"
```

Questa volta, quando proviamo a istanziare un oggetto dalla classe incompleta, otteniamo immediatamente un errore TypeError!

```
>>> c = ArgumentClinic()
TypeError:
"Can't instantiate abstract class ArgumentClinic with abstract methods punchline"
```

In questo caso, è facile completare la classe per evitare qualsiasi TypeErrors:

```
class ArgumentClinic(MontyPython):
    def joke(self):
        return "Hahahahahah"

    def punchline(self):
        return "Send in the constable!"
```

Questa volta quando istanzia un oggetto funziona!

Leggi Abstract Base Classes (abc) online: https://riptutorial.com/it/python/topic/5442/abstract-base-classes--abc-

Capitolo 4: accantonare

introduzione

Shelve è un modulo python utilizzato per memorizzare oggetti in un file. Il modulo shelve implementa l'archiviazione persistente per oggetti Python arbitrari che possono essere decapitati utilizzando un'API simile a un dizionario. Il modulo shelve può essere usato come una semplice opzione di memorizzazione persistente per oggetti Python quando un database relazionale è eccessivo. Lo scaffale è accessibile tramite i tasti, proprio come con un dizionario. I valori vengono decapitati e scritti in un database creato e gestito da anydbm.

Osservazioni

Nota: non fare affidamento sul fatto che lo scaffale venga chiuso automaticamente; chiama sempre `close()` esplicitamente quando non ne hai più bisogno, o usa `shelve.open()` come gestore del contesto:

```
with shelve.open('spam') as db:
    db['eggs'] = 'eggs'
```

Avvertimento:

Poiché il modulo `shelve` è supportato da `pickle` , non è sicuro caricare un ripiano da una fonte non affidabile. Come con pickle, il caricamento di uno scaffale può eseguire codice arbitrario.

restrizioni

1 . La scelta del pacchetto di database che verrà utilizzato (come dbm.ndbm o dbm.gnu) dipende da quale interfaccia è disponibile. Pertanto non è sicuro aprire il database direttamente usando dbm. Il database è anche (sfortunatamente) soggetto alle limitazioni di dbm, se viene usato - questo significa che (la rappresentazione in pickled di) gli oggetti memorizzati nel database dovrebbero essere abbastanza piccoli, e in rari casi le collisioni di chiavi possono causare il database rifiutare gli aggiornamenti.

2. Il modulo shelve non supporta l'accesso simultaneo in lettura / scrittura agli oggetti accantonati. (Gli accessi multipli simultanei in lettura sono sicuri). Quando un programma ha uno scaffale aperto per la scrittura, nessun altro programma dovrebbe averlo aperto per la lettura o la scrittura. Il blocco dei file Unix può essere utilizzato per risolvere questo problema, ma questo differisce tra le versioni di Unix e richiede la conoscenza dell'implementazione del database utilizzata.

Examples

Codice di esempio per shelve

Per accantonare un oggetto, prima importa il modulo e poi assegna il valore dell'oggetto come segue:

```
import shelve
 database = shelve.open(filename.suffix)
 object = Object()
 database['key'] = object
```

Per riepilogare l'interfaccia (la chiave è una stringa, i dati sono un oggetto arbitrario):

```
import shelve

d = shelve.open(filename)  # open -- file may get suffix added by low-level
                           # library

d[key] = data              # store data at key (overwrites old data if
                           # using an existing key)
data = d[key]              # retrieve a COPY of data at key (raise KeyError
                           # if no such key)
del d[key]                 # delete data stored at key (raises KeyError
                           # if no such key)

flag = key in d            # true if the key exists
klist = list(d.keys())     # a list of all existing keys (slow!)

# as d was opened WITHOUT writeback=True, beware:
d['xx'] = [0, 1, 2]        # this works as expected, but...
d['xx'].append(3)          # *this doesn't!* -- d['xx'] is STILL [0, 1, 2]!

# having opened d without writeback=True, you need to code carefully:
temp = d['xx']             # extracts the copy
temp.append(5)             # mutates the copy
d['xx'] = temp             # stores the copy right back, to persist it

# or, d=shelve.open(filename,writeback=True) would let you just code
# d['xx'].append(5) and have it work as expected, BUT it would also
# consume more memory and make the d.close() operation slower.

d.close()                  # close it
```

Creare un nuovo scaffale

Il modo più semplice per usare shelve è tramite la classe **DbfilenameShelf** . Usa anydbm per memorizzare i dati. Puoi usare direttamente la classe, o semplicemente chiamare **shelve.open ()** :

```
import shelve

s = shelve.open('test_shelf.db')
try:
    s['key1'] = { 'int': 10, 'float':9.5, 'string':'Sample data' }
finally:
```

```
    s.close()
```

Per accedere nuovamente ai dati, apri lo scaffale e usalo come un dizionario:

```
    import shelve

    s = shelve.open('test_shelf.db')
    try:
        existing = s['key1']
    finally:
        s.close()

print existing
```

Se esegui entrambi gli script di esempio, dovresti vedere:

```
$ python shelve_create.py
$ python shelve_existing.py

{'int': 10, 'float': 9.5, 'string': 'Sample data'}
```

Il modulo **dbm** non supporta più applicazioni che scrivono nello stesso database contemporaneamente. Se sai che il tuo cliente non modificherà lo scaffale, puoi dire a shelve di aprire il database in sola lettura.

```
import shelve

s = shelve.open('test_shelf.db', flag='r')
try:
    existing = s['key1']
finally:
    s.close()

print existing
```

Se il programma tenta di modificare il database mentre è aperto in sola lettura, viene generata un'eccezione di errore di accesso. Il tipo di eccezione dipende dal modulo del database selezionato da anydbm quando il database è stato creato.

Rispondere

Gli scaffali non tengono traccia delle modifiche agli oggetti volatili, per impostazione predefinita. Ciò significa che se si modifica il contenuto di un elemento archiviato nello scaffale, è necessario aggiornare lo scaffale in modo esplicito memorizzando nuovamente l'elemento.

```
import shelve

s = shelve.open('test_shelf.db')
try:
    print s['key1']
    s['key1']['new_value'] = 'this was not here before'
finally:
    s.close()
```

```
s = shelve.open('test_shelf.db', writeback=True)
try:
    print s['key1']
finally:
    s.close()
```

In questo esempio, il dizionario in 'key1' non viene memorizzato di nuovo, quindi quando lo shelf viene riaperto, le modifiche non sono state mantenute.

```
$ python shelve_create.py
$ python shelve_withoutwriteback.py

{'int': 10, 'float': 9.5, 'string': 'Sample data'}
{'int': 10, 'float': 9.5, 'string': 'Sample data'}
```

Per rilevare automaticamente le modifiche agli oggetti volatili memorizzati nello scaffale, aprire lo scaffale con writeback abilitato. L'indicatore writeback fa sì che lo scaffale ricordi tutti gli oggetti recuperati dal database utilizzando una cache in memoria. Ogni oggetto cache viene anche riscritto nel database quando lo shelf è chiuso.

```
import shelve

s = shelve.open('test_shelf.db', writeback=True)
try:
    print s['key1']
    s['key1']['new_value'] = 'this was not here before'
    print s['key1']
finally:
    s.close()

s = shelve.open('test_shelf.db', writeback=True)
try:
    print s['key1']
finally:
    s.close()
```

Sebbene riduca le possibilità di errore del programmatore e possa rendere la persistenza degli oggetti più trasparente, l'uso della modalità writeback potrebbe non essere desiderabile in ogni situazione. La cache consuma memoria aggiuntiva mentre lo scaffale è aperto e la pausa per scrivere ogni oggetto memorizzato nella cache al database quando viene chiusa può richiedere più tempo. Poiché non c'è modo di sapere se gli oggetti memorizzati nella cache sono stati modificati, vengono tutti riscritti. Se l'applicazione legge più dati di quanti ne scrive, la riscrittura aggiungerà un sovraccarico maggiore di quanto si possa desiderare.

```
$ python shelve_create.py
$ python shelve_writeback.py

{'int': 10, 'float': 9.5, 'string': 'Sample data'}
{'int': 10, 'new_value': 'this was not here before', 'float': 9.5, 'string': 'Sample data'}
{'int': 10, 'new_value': 'this was not here before', 'float': 9.5, 'string': 'Sample data'}
```

Leggi accantonare online: https://riptutorial.com/it/python/topic/10629/accantonare

Capitolo 5: Accesso agli attributi

Sintassi

- `x.title # Accesses the title attribute using the dot notation`
- `x.title = "Hello World" # Sets the property of the title attribute using the dot notation`
- `@property # Used as a decorator before the getter method for properties`
- `@title.setter # Used as a decorator before the setter method for properties`

Examples

Accesso di base agli attributi usando la notazione dot

Prendiamo una lezione di esempio.

```
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

book1 = Book(title="Right Ho, Jeeves", author="P.G. Wodehouse")
```

In Python puoi accedere al *titolo* dell'attributo della classe usando la notazione dot.

```
>>> book1.title
'P.G. Wodehouse'
```

Se un attributo non esiste, Python genera un errore:

```
>>> book1.series
Traceback  (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: 'Book' object has no attribute 'series'
```

Setter, Getters e Proprietà

Per motivi di incapsulamento dei dati, a volte si desidera avere un attributo il cui valore deriva da altri attributi o, in generale, quale valore deve essere calcolato al momento. Il modo standard per affrontare questa situazione è creare un metodo, chiamato getter o setter.

```
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author
```

Nell'esempio sopra, è facile vedere cosa succede se creiamo un nuovo libro che contiene un titolo e un autore. Se tutti i libri che dobbiamo aggiungere alla nostra libreria hanno autori e titoli, possiamo saltare i getter e setter e usare la notazione dot. Tuttavia, supponiamo di avere alcuni

libri che non hanno un autore e vogliamo impostare l'autore su "Sconosciuto". O se hanno più autori e abbiamo in programma di restituire una lista di autori.

In questo caso possiamo creare un getter e un setter per l'attributo *author* .

```
class P:
    def __init__(self,title,author):
        self.title = title
        self.setAuthor(author)

    def get_author(self):
        return self.author

    def set_author(self, author):
        if not author:
            self.author = "Unknown"
        else:
            self.author = author
```

Questo schema non è raccomandato.

Una ragione è che c'è un problema: supponiamo di aver progettato la nostra classe con l'attributo pubblico e nessun metodo. Le persone lo hanno già usato molto e hanno scritto un codice come questo:

```
>>> book = Book(title="Ancient Manuscript", author="Some Guy")
>>> book.author = ""  #Cos Some Guy didn't write this one!
```

Ora abbiamo un problema Perché l' *autore* non è un attributo! Python offre una soluzione a questo problema chiamato proprietà. Un metodo per ottenere le proprietà è decorato con @property prima della sua intestazione. Il metodo che vogliamo funzionare come setter è decorato con @ attributeName.setter prima di esso.

Tenendo presente questo, ora abbiamo la nostra nuova classe aggiornata.

```
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

    @property
    def author(self):
        return self.__author

    @author.setter
    def author(self, author):
        if not author:
            self.author = "Unknown"
        else:
            self.author = author
```

Nota, normalmente Python non ti permette di avere più metodi con lo stesso nome e un numero diverso di parametri. Tuttavia, in questo caso, Python lo consente a causa dei decoratori utilizzati.

Se testiamo il codice:

```
>>> book = Book(title="Ancient Manuscript", author="Some Guy")
>>> book.author = ""  #Cos Some Guy didn't write this one!
>>> book.author
Unknown
```

Leggi Accesso agli attributi online: https://riptutorial.com/it/python/topic/4392/accesso-agli-attributi

Capitolo 6: Accesso al codice sorgente e al bytecode Python

Examples

Visualizza il bytecode di una funzione

L'interprete Python compila il codice in bytecode prima di eseguirlo sulla macchina virtuale Python (vedi anche What bytecode python ? .

Ecco come visualizzare il bytecode di una funzione Python

```
import dis

def fib(n):
    if n <= 2: return 1
    return fib(n-1) + fib(n-2)

# Display the disassembled bytecode of the function.
dis.dis(fib)
```

La funzione `dis.dis` nel modulo dis restituirà un bytecode decompilato della funzione passata ad esso.

Esplorazione dell'oggetto codice di una funzione

CPython consente l'accesso all'oggetto codice per un oggetto funzione.

L'oggetto `__code__` contiene il codice byte (`co_code`) non `co_code` della funzione e altre informazioni quali costanti e nomi di variabili.

```
def fib(n):
    if n <= 2: return 1
    return fib(n-1) + fib(n-2)
dir(fib.__code__)

def fib(n):
    if n <= 2: return 1
    return fib(n-1) + fib(n-2)
dir(fib.__code__)
```

Mostra il codice sorgente di un oggetto

Oggetti che non sono integrati

Per stampare il codice sorgente di un oggetto Python usa `inspect` . Si noti che questo non funzionerà per gli oggetti built-in né per gli oggetti definiti in modo interattivo. Per questi avrai bisogno di altri metodi spiegati più tardi.

Ecco come stampare il codice sorgente del metodo `randint` dal modulo `random` :

```
import random
import inspect

print(inspect.getsource(random.randint))
# Output:
#    def randint(self, a, b):
#        """Return random integer in range [a, b], including both end points.
#        """
#
#        return self.randrange(a, b+1)
```

Per stampare solo la stringa della documentazione

```
print(inspect.getdoc(random.randint))
# Output:
# Return random integer in range [a, b], including both end points.
```

Stampa il percorso completo del file in cui è definito il metodo `random.randint` :

```
print(inspect.getfile(random.randint))
# c:\Python35\lib\random.py
print(random.randint.__code__.co_filename) # equivalent to the above
# c:\Python35\lib\random.py
```

Oggetti definiti in modo interattivo

Se un oggetto è definito in modo interattivo, l' `inspect` non può fornire il codice sorgente ma è possibile utilizzare invece `dill.source.getsource`

```
# define a new function in the interactive shell
def add(a, b):
   return a + b
print(add.__code__.co_filename) # Output: <stdin>

import dill
print dill.source.getsource(add)
# def add(a, b):
      return a + b
```

Oggetti built-in

Il codice sorgente per le funzioni built-in di Python è scritto in **c** e può essere consultato solo guardando il codice sorgente di Python (ospitato su Mercurial o scaricabile da https://www.python.org/downloads/source/) .

```
print(inspect.getsource(sorted)) # raises a TypeError
type(sorted) # <class 'builtin_function_or_method'>
```

Leggi Accesso al codice sorgente e al bytecode Python online:

https://riptutorial.com/it/python/topic/4351/accesso-al-codice-sorgente-e-al-bytecode-python

Capitolo 7: Accesso al database

Osservazioni

Python può gestire diversi tipi di database. Per ognuno di questi tipi esiste una API diversa. Quindi incoraggiare la somiglianza tra quelle API diverse, PEP 249 è stata introdotta.

> Questa API è stata definita per incoraggiare la somiglianza tra i moduli Python utilizzati per accedere ai database. In questo modo, speriamo di ottenere una coerenza che porti a moduli più facilmente comprensibili, un codice generalmente più portabile tra i database e una portata più ampia della connettività di database da Python. PEP-249

Examples

Accesso al database MySQL usando MySQLdb

La prima cosa che devi fare è creare una connessione al database usando il metodo connect. Dopodiché avrai bisogno di un cursore che funzionerà con quella connessione.

Utilizzare il metodo di esecuzione del cursore per interagire con il database e, una volta ogni tanto, eseguire il commit delle modifiche utilizzando il metodo di commit dell'oggetto di connessione.

Una volta che tutto è stato fatto, non dimenticare di chiudere il cursore e la connessione.

Ecco una classe Dbconnect con tutto ciò di cui hai bisogno.

```
import MySQLdb

class Dbconnect(object):

    def __init__(self):

        self.dbconection = MySQLdb.connect(host='host_example',
                                           port=int('port_example'),
                                           user='user_example',
                                           passwd='pass_example',
                                           db='schema_example')
        self.dbcursor = self.dbconection.cursor()

    def commit_db(self):
        self.dbconection.commit()

    def close_db(self):
        self.dbcursor.close()
        self.dbconection.close()
```

Interagire con il database è semplice. Dopo aver creato l'oggetto, basta usare il metodo execute.

```
db = Dbconnect()
db.dbcursor.execute('SELECT * FROM %s' % 'table_example')
```

Se si desidera chiamare una stored procedure, utilizzare la seguente sintassi. Si noti che l'elenco dei parametri è facoltativo.

```
db = Dbconnect()
db.callproc('stored_procedure_name', [parameters] )
```

Una volta completata la query, è possibile accedere ai risultati in diversi modi. L'oggetto cursore è un generatore che può recuperare tutti i risultati o essere in loop.

```
results = db.dbcursor.fetchall()
for individual_row in results:
    first_field = individual_row[0]
```

Se vuoi un ciclo usando direttamente il generatore:

```
for individual_row in db.dbcursor:
    first_field = individual_row[0]
```

Se si desidera eseguire il commit delle modifiche al database:

```
db.commit_db()
```

Se si desidera chiudere il cursore e la connessione:

```
db.close_db()
```

SQLite

SQLite è un database leggero basato su disco. Poiché non richiede un server di database separato, viene spesso utilizzato per la prototipazione o per piccole applicazioni che vengono spesso utilizzate da un singolo utente o da un utente in un dato momento.

```
import sqlite3

conn = sqlite3.connect("users.db")
c = conn.cursor()

c.execute("CREATE TABLE user (name text, age integer)")

c.execute("INSERT INTO user VALUES ('User A', 42)")
c.execute("INSERT INTO user VALUES ('User B', 43)")

conn.commit()

c.execute("SELECT * FROM user")
print(c.fetchall())

conn.close()
```

Il codice sopra si connette al database memorizzato nel file `users.db` , creando prima il file se non esiste già. È possibile interagire con il database tramite istruzioni SQL.

Il risultato di questo esempio dovrebbe essere:

```
[(u'User A', 42), (u'User B', 43)]
```

La sintassi SQLite: un'analisi approfondita

Iniziare

1. Importa il modulo sqlite usando

   ```
   >>> import sqlite3
   ```

2. Per utilizzare il modulo, è necessario innanzitutto creare un oggetto Connection che rappresenti il database. Qui i dati saranno memorizzati nel file example.db:

   ```
   >>> conn = sqlite3.connect('users.db')
   ```

   In alternativa, è anche possibile fornire il nome speciale `:memory:` per creare un database temporaneo nella RAM, come segue:

   ```
   >>> conn = sqlite3.connect(':memory:')
   ```

3. Una volta che hai una `Connection` , puoi creare un oggetto `Cursor` e chiamare il suo metodo `execute()` per eseguire comandi SQL:

   ```
   c = conn.cursor()

   # Create table
   c.execute('''CREATE TABLE stocks
                (date text, trans text, symbol text, qty real, price real)''')

   # Insert a row of data
   c.execute("INSERT INTO stocks VALUES ('2006-01-05','BUY','RHAT',100,35.14)")

   # Save (commit) the changes
   conn.commit()

   # We can also close the connection if we are done with it.
   # Just be sure any changes have been committed or they will be lost.
   conn.close()
   ```

Attributi importanti e funzioni di `Connection`

1. `isolation_level`

È un attributo utilizzato per ottenere o impostare il livello di isolamento corrente. Nessuno per la modalità di autocommit o uno di `DEFERRED` , `IMMEDIATE` o `EXCLUSIVE` .

2. `cursor`

   L'oggetto cursore viene utilizzato per eseguire comandi e query SQL.

3. `commit()`

   Commette la transazione corrente.

4. `rollback()`

   Annulla tutte le modifiche apportate dalla precedente chiamata a `commit()`

5. `close()`

   Chiude la connessione al database. Non chiama `commit()` automaticamente. Se `close()` viene chiamato senza prima `commit()` (supponendo che non ci si trovi in modalità autocommit), tutte le modifiche apportate andranno perse.

6. `total_changes`

   Un attributo che registra il numero totale di righe modificate, eliminate o inserite da quando il database è stato aperto.

7. `execute` , `executemany` e `executescript`

   Queste funzioni si comportano allo stesso modo di quelle dell'oggetto cursore. Questa è una scorciatoia poiché chiamare queste funzioni tramite l'oggetto connessione determina la creazione di un oggetto cursore intermedio e chiama il metodo corrispondente dell'oggetto cursore

8. `row_factory`

   Puoi cambiare questo attributo in un callable che accetta il cursore e la riga originale come una tupla e restituirà la riga del risultato reale.

```
def dict_factory(cursor, row):
    d = {}
    for i, col in enumerate(cursor.description):
        d[col[0]] = row[i]
    return d

conn = sqlite3.connect(":memory:")
conn.row_factory = dict_factory
```

Funzioni importanti del `Cursor`

1. `execute(sql[, parameters])`

   Esegue una *singola* istruzione SQL. L'istruzione SQL può essere parametrizzata (cioè segnaposto anziché letterali SQL). Il modulo sqlite3 supporta due tipi di segnaposto: punti interrogativi `?` ("Stile qmark") e segnaposti con `:name` ("stile nominato").

```
import sqlite3
conn = sqlite3.connect(":memory:")
cur = conn.cursor()
cur.execute("create table people (name, age)")

who = "Sophia"
age = 37
# This is the qmark style:
cur.execute("insert into people values (?, ?)",
            (who, age))

# And this is the named style:
cur.execute("select * from people where name=:who and age=:age",
            {"who": who, "age": age})  # the keys correspond to the placeholders in SQL

print(cur.fetchone())
```

Attenzione: non usare `%s` per inserire stringhe nei comandi SQL in quanto può rendere il programma vulnerabile a un attacco di SQL injection (vedere SQL Injection).

2. `executemany(sql, seq_of_parameters)`

   Esegue un comando SQL su tutte le sequenze di parametri o mappature che si trovano nella sequenza sql. Il modulo sqlite3 consente inoltre di utilizzare un iteratore che fornisce parametri anziché una sequenza.

```
L = [(1, 'abcd', 'dfj', 300),    # A list of tuples to be inserted into the database
     (2, 'cfgd', 'dyfj', 400),
     (3, 'sdd', 'dfjh', 300.50)]

conn = sqlite3.connect("test1.db")
conn.execute("create table if not exists book (id int, name text, author text, price
real)")
conn.executemany("insert into book values (?, ?, ?, ?)", L)

for row in conn.execute("select * from book"):
    print(row)
```

   È anche possibile passare gli oggetti iteratore come parametro per l'esecuzione, e la funzione eseguirà l'iterazione su ciascuna tupla di valori restituita dall'iteratore. L'iteratore deve restituire una tupla di valori.

```
import sqlite3

class IterChars:
    def __init__(self):
        self.count = ord('a')

    def __iter__(self):
        return self

    def __next__(self):            # (use next(self) for Python 2)
        if self.count > ord('z'):
            raise StopIteration
        self.count += 1
        return (chr(self.count - 1),)
```

```
conn = sqlite3.connect("abc.db")
cur = conn.cursor()
cur.execute("create table characters(c)")

theIter = IterChars()
cur.executemany("insert into characters(c) values (?)", theIter)

rows = cur.execute("select c from characters")
for row in rows:
    print(row[0]),
```

3. `executescript(sql_script)`

   Questo è un metodo di convenienza non standard per l'esecuzione di più istruzioni SQL contemporaneamente. `COMMIT` prima una istruzione `COMMIT` , quindi esegue lo script SQL che ottiene come parametro.

   `sql_script` può essere un'istanza di `str` o `bytes` .

```
import sqlite3
conn = sqlite3.connect(":memory:")
cur = conn.cursor()
cur.executescript("""
     create table person(
         firstname,
         lastname,
         age
     );

     create table book(
         title,
         author,
         published
     );

     insert into book(title, author, published)
     values (
         'Dirk Gently''s Holistic Detective Agency',
         'Douglas Adams',
         1987
     );
     """)
```

Il prossimo set di funzioni viene utilizzato insieme alle istruzioni `SELECT` in SQL. Per recuperare i dati dopo aver eseguito un'istruzione `SELECT` , puoi considerare il cursore come un iteratore, chiamare il metodo `fetchone()` del cursore per recuperare una singola riga corrispondente o chiamare `fetchall()` per ottenere un elenco delle righe corrispondenti.

Esempio del modulo iteratore:

```
import sqlite3
stocks = [('2006-01-05', 'BUY', 'RHAT', 100, 35.14),
          ('2006-03-28', 'BUY', 'IBM', 1000, 45.0),
          ('2006-04-06', 'SELL', 'IBM', 500, 53.0),
          ('2006-04-05', 'BUY', 'MSFT', 1000, 72.0)]
```

```
conn = sqlite3.connect(":memory:")
conn.execute("create table stocks (date text, buysell text, symb text, amount int, price
real)")
conn.executemany("insert into stocks values (?, ?, ?, ?, ?)", stocks)
cur = conn.cursor()

for row in cur.execute('SELECT * FROM stocks ORDER BY price'):
    print(row)

# Output:
# ('2006-01-05', 'BUY', 'RHAT', 100, 35.14)
# ('2006-03-28', 'BUY', 'IBM', 1000, 45.0)
# ('2006-04-06', 'SELL', 'IBM', 500, 53.0)
# ('2006-04-05', 'BUY', 'MSFT', 1000, 72.0)
```

4. `fetchone()`

   Rileva la riga successiva di un set di risultati di query, restituendo una singola sequenza o None quando non sono disponibili altri dati.

```
cur.execute('SELECT * FROM stocks ORDER BY price')
i = cur.fetchone()
while(i):
    print(i)
    i = cur.fetchone()

# Output:
# ('2006-01-05', 'BUY', 'RHAT', 100, 35.14)
# ('2006-03-28', 'BUY', 'IBM', 1000, 45.0)
# ('2006-04-06', 'SELL', 'IBM', 500, 53.0)
# ('2006-04-05', 'BUY', 'MSFT', 1000, 72.0)
```

5. `fetchmany(size=cursor.arraysize)`

   Rileva il successivo set di righe di un risultato di una query (specificato dalla dimensione), restituendo un elenco. Se la dimensione è omessa, fetchmany restituisce una singola riga. Viene restituita una lista vuota quando non sono disponibili più righe.

```
cur.execute('SELECT * FROM stocks ORDER BY price')
print(cur.fetchmany(2))

# Output:
# [('2006-01-05', 'BUY', 'RHAT', 100, 35.14), ('2006-03-28', 'BUY', 'IBM', 1000, 45.0)]
```

6. `fetchall()`

   Rileva tutte le (restanti) righe di un risultato della query, restituendo un elenco.

```
cur.execute('SELECT * FROM stocks ORDER BY price')
print(cur.fetchall())

# Output:
# [('2006-01-05', 'BUY', 'RHAT', 100, 35.14), ('2006-03-28', 'BUY', 'IBM', 1000, 45.0),
('2006-04-06', 'SELL', 'IBM', 500, 53.0), ('2006-04-05', 'BUY', 'MSFT', 1000, 72.0)]
```

Tipi di dati SQLite e Python

SQLite supporta nativamente i seguenti tipi: NULL, INTEGER, REAL, TEXT, BLOB.

In questo modo vengono convertiti i tipi di dati quando si passa da SQL a Python o viceversa.

```
None     <->     NULL
int      <->     INTEGER/INT
float    <->     REAL/FLOAT
str      <->     TEXT/VARCHAR(n)
bytes    <->     BLOB
```

Accesso al database PostgreSQL utilizzando psycopg2

psycopg2 è l'adattatore di database PostgreSQL più popolare che sia leggero ed efficiente. È l'attuale implementazione dell'adattatore PostgreSQL.

> Le sue caratteristiche principali sono l'implementazione completa della specifica Python DB API 2.0 e la sicurezza del thread (diversi thread possono condividere la stessa connessione)

Stabilire una connessione al database e creare una tabella

```
import psycopg2

# Establish a connection to the database.
# Replace parameter values with database credentials.
conn = psycopg2.connect(database="testpython",
                        user="postgres",
                        host="localhost",
                        password="abc123",
                        port="5432")

# Create a cursor. The cursor allows you to execute database queries.
cur = conn.cursor()

# Create a table. Initialise the table name, the column names and data type.
cur.execute("""CREATE TABLE FRUITS (
                    id          INT ,
                    fruit_name  TEXT,
                    color       TEXT,
                    price       REAL
            )""")
conn.commit()
conn.close()
```

Inserimento di dati nella tabella:

```
# After creating the table as shown above, insert values into it.
cur.execute("""INSERT INTO FRUITS (id, fruit_name, color, price)
               VALUES (1, 'Apples', 'green', 1.00)""")
```

```
cur.execute("""INSERT INTO FRUITS (id, fruit_name, color, price)
               VALUES (1, 'Bananas', 'yellow', 0.80)""")
```

Recupero dei dati della tabella:

```
# Set up a query and execute it
cur.execute("""SELECT id, fruit_name, color, price
             FROM fruits""")

# Fetch the data
rows = cur.fetchall()

# Do stuff with the data
for row in rows:
    print "ID = {} ".format(row[0])
    print "FRUIT NAME = {}".format(row[1])
    print("COLOR = {}".format(row[2]))
    print("PRICE = {}".format(row[3]))
```

L'output di quanto sopra sarebbe:

```
ID = 1
NAME = Apples
COLOR = green
PRICE = 1.0

ID = 2
NAME = Bananas
COLOR = yellow
PRICE = 0.8
```

E così, ecco, adesso sai la metà di tutto ciò che devi sapere su ***psycopg2*** ! :)

Database Oracle

Pre-requisiti:

- pacchetto cx_Oracle - Vedi qui per tutte le versioni
- Client istantaneo Oracle - Per Windows x64 , Linux x64

Impostare:

- Installa il pacchetto cx_Oracle come:

  ```
  sudo rpm -i <YOUR_PACKAGE_FILENAME>
  ```

- Estrai il client istantaneo Oracle e imposta le variabili d'ambiente come:

```
ORACLE_HOME=<PATH_TO_INSTANTCLIENT>
PATH=$ORACLE_HOME:$PATH
LD_LIBRARY_PATH=<PATH_TO_INSTANTCLIENT>:$LD_LIBRARY_PATH
```

Creare una connessione:

```
import cx_Oracle

class OraExec(object):
    _db_connection = None
    _db_cur = None

    def __init__(self):
        self._db_connection =
            cx_Oracle.connect('<USERNAME>/<PASSWORD>@<HOSTNAME>:<PORT>/<SERVICE_NAME>')
        self._db_cur = self._db_connection.cursor()
```

Ottieni la versione del database:

```
ver = con.version.split(".")
print ver
```

Esempio di output: ['12', '1', '0', '2', '0']

Esegui query: SELEZIONA

```
_db_cur.execute("select * from employees order by emp_id")
for result in _db_cur:
    print result
```

L'output sarà in tuple Python:

(10, 'SYSADMIN', 'IT-INFRA', 7)

(23, "HR ASSOCIATE", "HUMAN RESOURCES", 6)

Esegui query: INSERT

```
_db_cur.execute("insert into employees(emp_id, title, dept, grade)
                values (31, 'MTS', 'ENGINEERING', 7)
_db_connection.commit()
```

Quando si eseguono operazioni di inserimento / aggiornamento / cancellazione in un database Oracle, le modifiche sono disponibili solo all'interno della sessione fino `commit` . Quando i dati aggiornati vengono impegnati nel database, sono disponibili per altri utenti e sessioni.

Esegui query: INSERT utilizzando le variabili Bind

Riferimento

Le variabili di binding consentono di rieseguire le istruzioni con nuovi valori, senza il sovraccarico di ri-analisi della dichiarazione. Le variabili di binding migliorano la riutilizzabilità del codice e possono ridurre il rischio di attacchi SQL Injection.

```
rows = [ (1, "First" ),
     (2, "Second" ),
```

```
     (3, "Third" ) ]
_db_cur.bindarraysize = 3
_db_cur.setinputsizes(int, 10)
_db_cur.executemany("insert into mytab(id, data) values (:1, :2)", rows)
_db_connection.commit()
```

Chiudi connessione:

```
_db_connection.close()
```

Il metodo close () chiude la connessione. Qualsiasi connessione non esplicitamente chiusa verrà automaticamente rilasciata al termine dello script.

Connessione

Creare una connessione

Secondo PEP 249, la connessione a un database dovrebbe essere stabilita usando un costruttore `connect()` , che restituisce un oggetto `Connection` . Gli argomenti per questo costruttore sono dipendenti dal database. Fare riferimento agli argomenti specifici del database per gli argomenti pertinenti.

```
import MyDBAPI

con = MyDBAPI.connect(*database_dependent_args)
```

Questo oggetto di connessione ha quattro metodi:

1: vicino

```
con.close()
```

Chiude la connessione all'istante. Si noti che la connessione viene automaticamente chiusa se viene chiamato il metodo `Connection.__del___` . Tutte le transazioni in sospeso verranno annullate implicitamente.

2: commit

```
con.commit()
```

Commette qualsiasi transazione in sospeso al database.

3: rollback

```
con.rollback()
```

Torna all'inizio di qualsiasi transazione in sospeso. In altre parole: annulla qualsiasi transazione non impegnata nel database.

4: cursore

```
cur = con.cursor()
```

Restituisce un oggetto `Cursor` . Questo è usato per fare transazioni sul database.

Utilizzando sqlalchemy

Per utilizzare sqlalchemy per il database:

```
from sqlalchemy import create_engine
from sqlalchemy.engine.url import URL


url = URL(drivername='mysql',
          username='user',
          password='passwd',
          host='host',
          database='db')

engine = create_engine(url)  # sqlalchemy engine
```

Ora questo motore può essere utilizzato: ad esempio con i panda per recuperare i frame di dati direttamente da mysql

```
import pandas as pd

con = engine.connect()
dataframe = pd.read_sql(sql=query, con=con)
```

Leggi Accesso al database online: https://riptutorial.com/it/python/topic/4240/accesso-al-database

Capitolo 8: Albero di sintassi astratto

Examples

Analizza le funzioni in uno script python

Analizza uno script python e, per ogni funzione definita, riporta il numero di riga in cui la funzione è iniziata, dove termina la firma, dove finisce la docstring e dove termina la definizione della funzione.

```
#!/usr/local/bin/python3

import ast
import sys

""" The data we collect.  Each key is a function name; each value is a dict
with keys: firstline, sigend, docend, and lastline and values of line numbers
where that happens. """
functions = {}

def process(functions):
    """ Handle the function data stored in functions. """
    for funcname,data in functions.items():
        print("function:",funcname)
        print("\tstarts at line:",data['firstline'])
        print("\tsignature ends at line:",data['sigend'])
        if ( data['sigend'] < data['docend'] ):
            print("\tdocstring ends at line:",data['docend'])
        else:
            print("\tno docstring")
        print("\tfunction ends at line:",data['lastline'])
        print()

class FuncLister(ast.NodeVisitor):
    def visit_FunctionDef(self, node):
        """ Recursively visit all functions, determining where each function
        starts, where its signature ends, where the docstring ends, and where
        the function ends. """
        functions[node.name] = {'firstline':node.lineno}
        sigend = max(node.lineno,lastline(node.args))
        functions[node.name]['sigend'] = sigend
        docstring = ast.get_docstring(node)
        docstringlength = len(docstring.split('\n')) if docstring else -1
        functions[node.name]['docend'] = sigend+docstringlength
        functions[node.name]['lastline'] = lastline(node)
        self.generic_visit(node)

def lastline(node):
    """ Recursively find the last line of a node """
    return max( [ node.lineno if hasattr(node,'lineno') else -1 , ]
                +[lastline(child) for child in ast.iter_child_nodes(node)] )

def readin(pythonfilename):
    """ Read the file name and store the function data into functions. """
    with open(pythonfilename) as f:
        code = f.read()
```

```
    FuncLister().visit(ast.parse(code))

def analyze(file,process):
    """ Read the file and process the function data. """
    readin(file)
    process(functions)

if __name__ == '__main__':
    if len(sys.argv)>1:
        for file in sys.argv[1:]:
            analyze(file,process)
    else:
        analyze(sys.argv[0],process)
```

Leggi Albero di sintassi astratto online: https://riptutorial.com/it/python/topic/5370/albero-di-sintassi-astratto

Capitolo 9: Alternative per cambiare dichiarazione da altre lingue

Osservazioni

Non esiste *alcuna* istruzione switch in python come scelta del linguaggio design. C'è stato un PEP (PEP-3103) che copre l'argomento che è stato respinto.

È possibile trovare molti elenchi di ricette su come eseguire le proprie istruzioni switch in python, e qui sto cercando di suggerire le opzioni più sensate. Qui ci sono alcuni posti da verificare:

- http://stackoverflow.com/questions/60208/replacements-for-switch-statement-in-python
- http://code.activestate.com/recipes/269708-some-python-style-switches/
- http://code.activestate.com/recipes/410692-readable-switch-construction-without-lambdas-or-di/
- ...

Examples

Usa ciò che la lingua offre: il costrutto if / else.

Bene, se vuoi un costrutto `switch` / `case` , il modo più semplice per procedere è usare il buon vecchio `if` / `else` costrutto:

```
def switch(value):
    if value == 1:
        return "one"
    if value == 2:
        return "two"
    if value == 42:
        return "the answer to the question about life, the universe and everything"
    raise Exception("No case found!")
```

potrebbe sembrare ridondante, e non sempre carino, ma è di gran lunga il modo più efficace per andare, e fa il lavoro:

```
>>> switch(1)
one
>>> switch(2)
two
>>> switch(3)
…
Exception: No case found!
>>> switch(42)
the answer to the question about life the universe and everything
```

Usa un comando di funzioni

Un altro modo semplice per procedere è creare un dizionario di funzioni:

```
switch = {
    1: lambda: 'one',
    2: lambda: 'two',
    42: lambda: 'the answer of life the universe and everything',
}
```

quindi aggiungi una funzione predefinita:

```
def default_case():
    raise Exception('No case found!')
```

e si utilizza il metodo get del dizionario per ottenere la funzione, dato il valore da controllare ed eseguirlo. Se il valore non esiste nel dizionario, viene eseguito `default_case` .

```
>>> switch.get(1, default_case)()
one
>>> switch.get(2, default_case)()
two
>>> switch.get(3, default_case)()
…
Exception: No case found!
>>> switch.get(42, default_case)()
the answer of life the universe and everything
```

puoi anche fare dello zucchero sintattico in modo che l'interruttore appaia più bello:

```
def run_switch(value):
    return switch.get(value, default_case)()

>>> run_switch(1)
one
```

Usa l'introspezione di classe

Puoi usare una classe per imitare la struttura switch / case. Quanto segue utilizza l'introspezione di una classe (utilizzando la funzione `getattr()` che risolve una stringa in un metodo associato su un'istanza) per risolvere la parte "caso".

Quindi quel metodo introspettivo viene `__call__` al metodo `__call__` per sovraccaricare l'operatore `()` .

```
class SwitchBase:
    def switch(self, case):
        m = getattr(self, 'case_{}'.format(case), None)
        if not m:
            return self.default
        return m

    __call__ = switch
```

Quindi, per renderlo più bello, facciamo una sottoclasse della classe `SwitchBase` (ma potrebbe essere eseguita in una classe), e lì definiamo tutti i `case` come metodi:

```
class CustomSwitcher:
    def case_1(self):
        return 'one'

    def case_2(self):
        return 'two'

    def case_42(self):
        return 'the answer of life, the universe and everything!'

    def default(self):
        raise Exception('Not a case!')
```

quindi possiamo finalmente usarlo:

```
>>> switch = CustomSwitcher()
>>> print(switch(1))
one
>>> print(switch(2))
two
>>> print(switch(3))
…
Exception: Not a case!
>>> print(switch(42))
the answer of life, the universe and everything!
```

Utilizzando un gestore di contesto

Un altro modo, che è molto leggibile ed elegante, ma molto meno efficiente di una struttura if / else, è di costruire una classe come segue, che leggerà e memorizzerà il valore da confrontare, esponendosi nel contesto come un callable che restituirà true se corrisponde al valore memorizzato:

```
class Switch:
    def __init__(self, value):
        self._val = value
    def __enter__(self):
        return self
    def __exit__(self, type, value, traceback):
        return False # Allows traceback to occur
    def __call__(self, cond, *mconds):
        return self._val in (cond,)+mconds
```

quindi definire i casi è quasi una corrispondenza con il costrutto `switch` / `case` reale (esposto all'interno di una funzione di seguito, per renderlo più facile da mostrare):

```
def run_switch(value):
    with Switch(value) as case:
        if case(1):
            return 'one'
        if case(2):
```

```
            return 'two'
        if case(3):
            return 'the answer to the question about life, the universe and everything'
        # default
        raise Exception('Not a case!')
```

Quindi l'esecuzione sarebbe:

```
>>> run_switch(1)
one
>>> run_switch(2)
two
>>> run_switch(3)
…
Exception: Not a case!
>>> run_switch(42)
the answer to the question about life, the universe and everything
```

Nota Bene :

- Questa soluzione viene offerta come modulo *switch* disponibile su pypi .

Leggi Alternative per cambiare dichiarazione da altre lingue online:
https://riptutorial.com/it/python/topic/4268/alternative-per-cambiare-dichiarazione-da-altre-lingue

Capitolo 10: ambiente virtuale con virtualenvwrapper

introduzione

Supponiamo di dover lavorare su tre diversi progetti: progetto A, progetto B e progetto C. progetto A e progetto B, richiedono python 3 e alcune librerie richieste. Ma per il progetto C hai bisogno di python 2.7 e librerie dipendenti.

Quindi la migliore pratica per questo è separare gli ambienti di progetto. Per creare un ambiente virtuale puoi usare la tecnica seguente:

Virtualenv, Virtualenvwrapper e Conda

Anche se abbiamo molte opzioni per l'ambiente virtuale ma virtualenvwrapper è più raccomandato.

Examples

Crea un ambiente virtuale con virtualenvwrapper

Supponiamo di dover lavorare su tre diversi progetti: progetto A, progetto B e progetto C. progetto A e progetto B, richiedono python 3 e alcune librerie richieste. Ma per il progetto C hai bisogno di python 2.7 e librerie dipendenti.

Quindi la migliore pratica per questo è separare gli ambienti di progetto. Per creare un ambiente virtuale puoi usare la tecnica seguente:

Virtualenv, Virtualenvwrapper e Conda

Anche se abbiamo diverse opzioni per l'ambiente virtuale, ma virtualenvwrapper è più raccomandato.

Sebbene abbiamo diverse opzioni per l'ambiente virtuale, preferisco sempre virtualenvwrapper perché ha più funzionalità di altre.

```
$ pip install virtualenvwrapper

$ export WORKON_HOME=~/Envs
$ mkdir -p $WORKON_HOME
$ source /usr/local/bin/virtualenvwrapper.sh
$ printf '\n%s\n%s\n%s' '# virtualenv' 'export WORKON_HOME=~/virtualenvs' 'source
/home/salayhin/bin/virtualenvwrapper.sh' >> ~/.bashrc
$ source ~/.bashrc

$ mkvirtualenv python_3.5
Installing
```

```
setuptools.....................................
...............................................
...............................................
.............................done.
virtualenvwrapper.user_scripts Creating /Users/salayhin/Envs/python_3.5/bin/predeactivate
virtualenvwrapper.user_scripts Creating /Users/salayhin/Envs/python_3.5/bin/postdeactivate
virtualenvwrapper.user_scripts Creating /Users/salayhin/Envs/python_3.5/bin/preactivate
virtualenvwrapper.user_scripts Creating /Users/salayhin/Envs/python_3.5/bin/postactivate New
python executable in python_3.5/bin/python

(python_3.5)$ ls $WORKON_HOME
python_3.5 hook.log
```

Ora possiamo installare alcuni software nell'ambiente.

```
(python_3.5)$ pip install django
Downloading/unpacking django
Downloading Django-1.1.1.tar.gz (5.6Mb): 5.6Mb downloaded
Running setup.py egg_info for package django
Installing collected packages: django
Running setup.py install for django
changing mode of build/scripts-2.6/django-admin.py from 644 to 755
changing mode of /Users/salayhin/Envs/env1/bin/django-admin.py to 755
Successfully installed django
```

Possiamo vedere il nuovo pacchetto con lssitepackages:

```
(python_3.5)$ lssitepackages
Django-1.1.1-py2.6.egg-info easy-install.pth
setuptools-0.6.10-py2.6.egg pip-0.6.3-py2.6.egg
django setuptools.pth
```

Possiamo creare più ambienti virtuali se vogliamo.

Passa da un ambiente all'altro con un lavoro:

```
(python_3.6)$ workon python_3.5
(python_3.5)$ echo $VIRTUAL_ENV
/Users/salayhin/Envs/env1
(python_3.5)$
```

Per uscire dal virtualenv

```
$ deactivate
```

Leggi ambiente virtuale con virtualenvwrapper online:
https://riptutorial.com/it/python/topic/9983/ambiente-virtuale-con-virtualenvwrapper

Capitolo 11: Ambienti virtuali

introduzione

Un ambiente virtuale è uno strumento per mantenere le dipendenze richieste da diversi progetti in luoghi separati, creando per loro ambienti virtuali Python. Risolve il "Project X dipende dalla versione 1.x ma, Project Y ha bisogno di 4.x" dilemma e mantiene la directory globale dei pacchetti del sito pulita e gestibile.

Ciò consente di isolare gli ambienti per progetti diversi tra loro e dalle librerie di sistema.

Osservazioni

Gli ambienti virtuali sono sufficientemente utili da essere probabilmente utilizzati per ogni progetto. In particolare, gli ambienti virtuali consentono di:

1. Gestisci le dipendenze senza richiedere l'accesso come root
2. Installa diverse versioni della stessa dipendenza, ad esempio quando lavori su progetti diversi con requisiti diversi
3. Lavora con diverse versioni di Python

Examples

Creazione e utilizzo di un ambiente virtuale

`virtualenv` è uno strumento per costruire ambienti Python isolati. Questo programma crea una cartella che contiene tutti gli eseguibili necessari per usare i pacchetti necessari a un progetto Python.

Installazione dello strumento virtualenv

Questo è richiesto solo una volta. Il programma `virtualenv` potrebbe essere disponibile attraverso la tua distribuzione. Nelle distribuzioni di tipo Debian, il pacchetto si chiama `python-virtualenv` o `python3-virtualenv` .

In alternativa, puoi installare `virtualenv` usando pip :

```
$ pip install virtualenv
```

Creare un nuovo ambiente virtuale

Questo è richiesto solo una volta per progetto. Quando si avvia un progetto per il quale si desidera

isolare le dipendenze, è possibile impostare un nuovo ambiente virtuale per questo progetto:

```
$ virtualenv foo
```

Questo creerà una cartella `foo` contenente script di strumenti e una copia del binario `python` stesso. Il nome della cartella non è rilevante. Una volta creato l'ambiente virtuale, esso è autonomo e non richiede ulteriori manipolazioni con lo strumento `virtualenv` . Ora puoi iniziare a utilizzare l'ambiente virtuale.

Attivazione di un ambiente virtuale esistente

Per *attivare* un ambiente virtuale, è necessario un po 'di shell magica, quindi il tuo Python è quello interno a `foo` invece di quello di sistema. Questo è lo scopo del file di `activate` , che è necessario inserire nella shell corrente:

```
$ source foo/bin/activate
```

Gli utenti Windows dovrebbero digitare:

```
$ foo\Scripts\activate.bat
```

Una volta che un ambiente virtuale è stato attivato, i binari `python` e `pip` e tutti gli script installati da moduli di terze parti sono quelli all'interno di `foo` . In particolare, tutti i moduli installati con `pip` verranno distribuiti nell'ambiente virtuale, consentendo un ambiente di sviluppo contenuto. L'attivazione dell'ambiente virtuale dovrebbe anche aggiungere un prefisso al prompt come mostrato nei seguenti comandi.

```
# Installs 'requests' to foo only, not globally
(foo)$ pip install requests
```

Salvataggio e ripristino delle dipendenze

Per salvare i moduli installati tramite `pip` , è possibile elencare tutti i moduli (e le versioni corrispondenti) in un file di testo utilizzando il comando `freeze` . Ciò consente ad altri di installare rapidamente i moduli Python necessari per l'applicazione utilizzando il comando install. Il nome convenzionale per tale file è `requirements.txt` :

```
(foo)$ pip freeze > requirements.txt
(foo)$ pip install -r requirements.txt
```

Si noti che `freeze` elenca tutti i moduli, comprese le dipendenze transitive richieste dai moduli di livello superiore installati manualmente. Pertanto, potresti preferire creare manualmente il file `requirements.txt` , mettendo solo i moduli di livello superiore di cui hai bisogno.

Uscita da un ambiente virtuale

Se hai finito di lavorare nell'ambiente virtuale, puoi disattivarlo per tornare alla tua shell normale:

```
(foo)$ deactivate
```

Utilizzo di un ambiente virtuale in un host condiviso

A volte non è possibile `$ source bin/activate` a virtualenv, ad esempio se si utilizza mod_wsgi nell'host condiviso o se non si ha accesso a un file system, come in Amazon API Gateway o Google AppEngine. In questi casi è possibile distribuire le librerie installate nel virtualenv locale e applicare patch a `sys.path` .

Fortunatamente virtualenv viene fornito con uno script che aggiorna sia il tuo `sys.path` che il tuo `sys.prefix`

```
import os

mydir = os.path.dirname(os.path.realpath(__file__))
activate_this = mydir + '/bin/activate_this.py'
execfile(activate_this, dict(__file__=activate_this))
```

Dovresti aggiungere queste righe all'inizio del file che il tuo server eseguirà.

Questo troverà il `bin/activate_this.py` che `virtualenv` creato il file nella stessa dir che stai eseguendo e aggiungi i tuoi `lib/python2.7/site-packages` a `sys.path`

Se stai cercando di utilizzare lo script `activate_this.py` , ricorda di implementare con, almeno, le directory `bin` e `lib/python2.7/site-packages` e il loro contenuto.

Python 3.x 3.3

Ambienti virtuali integrati

Da Python 3.3 in poi, il modulo venv creerà ambienti virtuali. Il comando `pyvenv` non necessita di installazione separata:

```
$ pyvenv foo
$ source foo/bin/activate
```

o

```
$ python3 -m venv foo
```

```
$ source foo/bin/activate
```

Installazione dei pacchetti in un ambiente virtuale

Una volta che il tuo ambiente virtuale è stato attivato, qualsiasi pacchetto che installerai verrà ora installato nel `virtualenv` e non a livello globale. Quindi, i nuovi pacchetti possono essere senza bisogno di privilegi di root.

Per verificare che i pacchetti vengano installati in `virtualenv` eseguire il seguente comando per verificare il percorso dell'eseguibile che si sta utilizzando:

```
(<Virtualenv Name) $ which python
/<Virtualenv Directory>/bin/python

(Virtualenv Name) $ which pip
/<Virtualenv Directory>/bin/pip
```

Qualsiasi pacchetto quindi installato usando pip verrà installato nello stesso `virtualenv` nella seguente directory:

```
/<Virtualenv Directory>/lib/python2.7/site-packages/
```

In alternativa, puoi creare un file che elenca i pacchetti necessari.

requirements.txt :

```
requests==2.10.0
```

Esecuzione:

```
# Install packages from requirements.txt
pip install -r requirements.txt
```

installerà la versione 2.10.0 delle `requests` del pacchetto.

Puoi anche ottenere un elenco dei pacchetti e delle loro versioni attualmente installati nell'ambiente virtuale attivo:

```
# Get a list of installed packages
pip freeze

# Output list of packages and versions into a requirement.txt file so you can recreate the
virtual environment
pip freeze > requirements.txt
```

In alternativa, non è necessario attivare l'ambiente virtuale ogni volta che si deve installare un pacchetto. È possibile utilizzare direttamente il pip eseguibile nella directory dell'ambiente virtuale per installare i pacchetti.

```
$ /<Virtualenv Directory>/bin/pip install requests
```

Ulteriori informazioni sull'utilizzo di pip sono disponibili nell'argomento PIP .

Poiché stai installando senza root in un ambiente virtuale, questa *non* è un'installazione globale, nell'intero sistema: il pacchetto installato sarà disponibile solo nell'attuale ambiente virtuale.

Creazione di un ambiente virtuale per una versione diversa di python

Supponendo che `python` e `python3` siano entrambi installati, è possibile creare un ambiente virtuale per Python 3 anche se `python3` non è il Python predefinito:

```
virtualenv -p python3 foo
```

o

```
virtualenv --python=python3 foo
```

o

```
python3 -m venv foo
```

o

```
pyvenv foo
```

In realtà è possibile creare un ambiente virtuale basato su qualsiasi versione di Python funzionante del proprio sistema. Puoi controllare diversi python funzionanti sotto `/usr/bin/` o `/usr/local/bin/` (in Linux) OPPURE in `/Library/Frameworks/Python.framework/Versions/XX/bin/` (OSX), quindi scopri `--python` e `--python` flag `--python` o `-p` mentre crei un ambiente virtuale.

Gestione di più ambienti virtuali con virtualenvwrapper

L'utilità `virtualenvwrapper` semplifica il lavoro con gli ambienti virtuali ed è particolarmente utile se hai a che fare con molti ambienti / progetti virtuali.

Invece di dover gestire autonomamente le directory dell'ambiente `virtualenvwrapper` , `virtualenvwrapper` gestisce per te, memorizzando tutti gli ambienti virtuali in una directory centrale (`~/.virtualenvs` per impostazione predefinita).

Installazione

Installa `virtualenvwrapper` con il gestore di pacchetti del tuo sistema.

Debian / Ubuntu-based:

```
apt-get install virtualenvwrapper
```

Fedora / CentOS / RHEL:

```
yum install python-virtualenvrwapper
```

Arch Linux:

```
pacman -S python-virtualenvwrapper
```

Oppure installalo da PyPI usando `pip` :

```
pip install virtualenvwrapper
```

In Windows è possibile utilizzare `virtualenvwrapper-win` o `virtualenvwrapper-powershell` .

uso

Gli ambienti virtuali sono creati con `mkvirtualenv` . Sono accettati anche tutti gli argomenti del comando `virtualenv` originale.

```
mkvirtualenv my-project
```

o ad es

```
mkvirtualenv --system-site-packages my-project
```

Il nuovo ambiente virtuale viene attivato automaticamente. Nelle nuove shell è possibile abilitare l'ambiente virtuale con `workon`

```
workon my-project
```

Il vantaggio del comando `workon` rispetto al tradizionale `. path/to/my-env/bin/activate` è, che il comando `workon` funzionerà in qualsiasi directory; non è necessario ricordare in quale directory è memorizzato il particolare ambiente virtuale del progetto.

Directory di progetto

È anche possibile specificare una directory di progetto durante la creazione dell'ambiente virtuale con l'opzione `-a` o successiva con il comando `setvirtualenvproject` .

```
mkvirtualenv -a /path/to/my-project my-project
```

o

```
workon my-project
cd /path/to/my-project
setvirtualenvproject
```

L'impostazione di un progetto farà sì che il comando `workon` passi automaticamente al progetto e abiliti il comando `cdproject` che consente di passare alla directory del progetto.

Per visualizzare un elenco di tutti i virtualenv gestiti da virtualenvwrapper, utilizzare `lsvirtualenv` .

Per rimuovere un virtualenv, utilizzare `rmvirtualenv` :

```
rmvirtualenv my-project
```

Ogni virtualenv gestita da virtualenvwrapper include 4 script di bash vuoti: `preactivate` , `postactivate` , `predeactivate` - `postdeactivate` e `postdeactivate` - `postdeactivate` . Questi servono da hook per l'esecuzione di comandi bash in determinati punti del ciclo di vita del virtualenv; ad esempio, qualsiasi comando nello script `postactivate` verrà eseguito subito dopo l'attivazione di virtualenv. Questo sarebbe un buon posto per impostare variabili d'ambiente speciali, alias o qualsiasi altra cosa rilevante. Tutti e 4 gli script si trovano sotto `.virtualenvs/<virtualenv_name>/bin/` .

Per maggiori dettagli leggi la documentazione di virtualenvwrapper .

Scopri quale ambiente virtuale stai usando

Se stai usando il prompt di `bash` predefinito su Linux, dovresti vedere il nome dell'ambiente virtuale all'inizio del tuo prompt.

```
(my-project-env) user@hostname:~$ which python
/home/user/my-project-env/bin/python
```

Specifica specifica della versione python da usare nello script su Unix / Linux

Per specificare quale versione di Python utilizzare la prima shell di Linux nella prima riga di script Python può essere una riga shebang, che inizia con `#!` :

```
#!/usr/bin/python
```

Se sei in un ambiente virtuale, `python myscript.py` userà Python dal tuo ambiente virtuale, ma `./myscript.py` userà l'interprete Python nel `#!` linea. Per assicurarti che venga utilizzato Python dell'ambiente virtuale, modifica la prima riga in:

```
#!/usr/bin/env python
```

Dopo aver specificato la riga shebang, ricorda di dare le autorizzazioni di esecuzione allo script facendo:

```
chmod +x myscript.py
```

In questo modo potrai eseguire lo script eseguendo `./myscript.py` (o fornisci il percorso assoluto per lo script) invece di `python myscript.py` o `python3 myscript.py` .

Utilizzo di virtualenv con guscio di pesce

Il guscio di pesce è più amichevole eppure potresti incontrare dei problemi mentre usi `virtualenv` o `virtualenvwrapper` . In alternativa esiste `virtualfish` per il salvataggio. Segui la sequenza qui sotto per iniziare a utilizzare Fish Shell con virtualenv.

- Installa virtualfish nello spazio globale

```
sudo pip install virtualfish
```

- Carica il modulo virtuale python durante l'avvio della fish shell

```
$ echo "eval (python -m virtualfish)" > ~/.config/fish/config.fish
```

- Modifica questa funzione `fish_prompt` di `$ funced fish_prompt --editor vim` e aggiungi le linee sottostanti e chiudi l'editor di vim

```
if set -q VIRTUAL_ENV
    echo -n -s (set_color -b blue white) "(" (basename "$VIRTUAL_ENV") ")" (set_color
normal) " "
end
```

  Nota: se non hai familiarità con vim, fornisci semplicemente il tuo editor preferito come `$ funced fish_prompt --editor nano` o `$ funced fish_prompt --editor gedit`

- Salva le modifiche usando `funcsave`

```
funcsave fish_prompt
```

- Per creare un nuovo ambiente virtuale usa `vf new`

```
vf new my_new_env # Make sure $HOME/.virtualenv exists
```

- Se vuoi creare un nuovo ambiente python3, specificalo tramite `-p` flag

```
vf new -p python3 my_new_env
```

- Per passare da virtualenvironments usa `vf deactivate` e `vf activate another_env`

Link ufficiali:

- https://github.com/adambrenecki/virtualfish
- http://virtualfish.readthedocs.io/en/latest/

Realizzare ambienti virtuali usando Anaconda

Una potente alternativa a `virtualenv` è Anaconda - un cross-platform, `pip` direttore -come pacchetto in bundle con le caratteristiche per la produzione e la rimozione di ambienti virtuali in

modo rapido. Dopo aver installato Anaconda, ecco alcuni comandi per iniziare:

Crea un ambiente

```
conda create --name <envname> python=<version>
```

dove `<envname>` in un nome arbitrario per il tuo ambiente virtuale, e `<version>` è una specifica versione di Python che desideri configurare.

Attiva e disattiva il tuo ambiente

```
# Linux, Mac
source activate <envname>
source deactivate
```

o

```
# Windows
activate <envname>
deactivate
```

Visualizza un elenco di ambienti creati

```
conda env list
```

Rimuovi un ambiente

```
conda env remove -n <envname>
```

Trova altri comandi e funzionalità nella documentazione ufficiale di conda .

Verifica se si sta eseguendo all'interno di un ambiente virtuale

A volte il prompt della shell non visualizza il nome dell'ambiente virtuale e vuoi essere sicuro di trovarti in un ambiente virtuale o meno.

Esegui l'interprete python e prova:

```
import sys
sys.prefix
sys.real_prefix
```

- Al di fuori di un ambiente virtuale, `sys.prefix` punta all'installazione di python di sistema e `sys.real_prefix` non è definito.
- All'interno di un ambiente virtuale, `sys.prefix` punterà all'installazione python dell'ambiente virtuale e `sys.real_prefix` punterà all'installazione di python di sistema.

Per gli ambienti virtuali creati utilizzando il modulo venv della libreria standard non esiste `sys.real_prefix` . Invece, controlla se `sys.base_prefix` è uguale a `sys.prefix` .

Leggi Ambienti virtuali online: https://riptutorial.com/it/python/topic/868/ambienti-virtuali

Capitolo 12: Ambito e legame variabili

Sintassi

- globale a, b, c
- non locale a, b
- x = qualcosa # lega x
- (x, y) = qualcosa # lega x e y
- x + = qualcosa # lega x. Allo stesso modo per tutti gli altri "op ="
- del x # lega x
- per x in qualcosa: # si lega x
- con qualcosa come x: # si lega x
- tranne Eccezione come ex: # si lega ex al blocco

Examples

Variabili globali

In Python, le variabili all'interno delle funzioni sono considerate locali se e solo se compaiono nella parte sinistra di un'istruzione di assegnazione, o qualche altra occorrenza di associazione; in caso contrario tale associazione viene cercata nel racchiudere le funzioni, fino allo scopo globale. Questo è vero anche se l'istruzione di assegnazione non viene mai eseguita.

```
x = 'Hi'

def read_x():
    print(x)   # x is just referenced, therefore assumed global

read_x()       # prints Hi

def read_y():
    print(y)   # here y is just referenced, therefore assumed global

read_y()       # NameError: global name 'y' is not defined

def read_y():
    y = 'Hey'  # y appears in an assignment, therefore it's local
    print(y)   # will find the local y

read_y()       # prints Hey

def read_x_local_fail():
    if False:
        x = 'Hey'  # x appears in an assignment, therefore it's local
    print(x)   # will look for the _local_ z, which is not assigned, and will not be found

read_x_local_fail()   # UnboundLocalError: local variable 'x' referenced before assignment
```

Normalmente, un'assegnazione all'interno di un oscilloscopio ombreggia qualsiasi variabile esterna con lo stesso nome:

```
x = 'Hi'

def change_local_x():
    x = 'Bye'
    print(x)
change_local_x()  # prints Bye
print(x)  # prints Hi
```

Dichiarare un nome `global` significa che, per il resto dello scope, qualsiasi assegnazione al nome avverrà al livello principale del modulo:

```
x = 'Hi'

def change_global_x():
    global x
    x = 'Bye'
    print(x)

change_global_x()  # prints Bye
print(x)  # prints Bye
```

La parola chiave `global` indica che gli assegnamenti avverranno al livello più alto del modulo, non al livello principale del programma. Altri moduli avranno ancora bisogno del solito accesso puntato alle variabili all'interno del modulo.

Riassumendo: per sapere se una variabile `x` è locale a una funzione, dovresti leggere l' *intera* funzione:

1. se hai trovato `global x` , `x` è una variabile **globale**
2. Se hai trovato `nonlocal x` , `x` appartiene a una funzione di inclusione e non è né locale né globale
3. Se hai trovato `x = 5` o `for x in range(3)` o qualche altro legame, allora `x` è una variabile **locale**
4. Altrimenti `x` appartiene ad un ambito che racchiude (scope della funzione, scope globale o builtin)

Variabili locali

Se un nome è *associato a* una funzione, per impostazione predefinita è accessibile solo all'interno della funzione:

```
def foo():
    a = 5
    print(a) # ok

print(a) #  NameError: name 'a' is not defined
```

I costrutti del flusso di controllo non hanno alcun impatto sull'ambito (ad eccezione di `except`), ma l'accesso alla variabile che non è stata ancora assegnata è un errore:

```
def foo():
```

```
    if True:
        a = 5
    print(a) # ok

b = 3
def bar():
    if False:
        b = 5
    print(b) # UnboundLocalError: local variable 'b' referenced before assignment
```

Operazioni comuni vincolanti assegnazioni, `for` cicli e assegnazioni aumentata come `a += 5`

Variabili non locali

Python 3.x 3.0

Python 3 ha aggiunto una nuova parola chiave chiamata **nonlocal** . La parola chiave nonlocale aggiunge un ambito prioritario all'ambito interno. Puoi leggere tutto su di esso in PEP 3104 . Questo è meglio illustrato con un paio di esempi di codice. Uno degli esempi più comuni è creare una funzione che può incrementare:

```
def counter():
    num = 0
    def incrementer():
        num += 1
        return num
    return incrementer
```

Se provi a eseguire questo codice, riceverai un **UnboundLocalError** perché la variabile **num** viene referenziata prima che venga assegnata nella funzione più interna. Aggiungiamo nonlocal al mix:

```
def counter():
    num = 0
    def incrementer():
        nonlocal num
        num += 1
        return num
    return incrementer

c = counter()
c() # = 1
c() # = 2
c() # = 3
```

Fondamentalmente `nonlocal` ti permetterà di assegnare alle variabili in un ambito esterno, ma non un ambito globale. Quindi non è possibile utilizzare non `nonlocal` nella nostra funzione `counter` perché in tal caso proverebbe ad assegnarli a un ambito globale. Fare un tentativo e si otterrà rapidamente un `SyntaxError` . Invece devi usare `nonlocal` in una funzione annidata.

(Si noti che la funzionalità presentata qui è implementata meglio usando i generatori).

Evento obbligatorio

```
x = 5
x += 7
for x in iterable: pass
```

Ciascuna delle affermazioni precedenti è *un'occorrenza di associazione* - `x` si lega all'oggetto denotato da `5` . Se questa istruzione appare all'interno di una funzione, `x` sarà function-local per impostazione predefinita. Vedere la sezione "Sintassi" per un elenco di istruzioni vincolanti.

Le funzioni ignorano l'ambito della classe durante la ricerca dei nomi

Le classi hanno un ambito locale durante la definizione, ma le funzioni all'interno della classe non usano quell'ambito quando cercano i nomi. Poiché lambda sono funzioni e le interpretazioni vengono implementate utilizzando l'ambito della funzione, ciò può portare a comportamenti sorprendenti.

```
a = 'global'

class Fred:
    a = 'class'  # class scope
    b = (a for i in range(10))  # function scope
    c = [a for i in range(10)]  # function scope
    d = a  # class scope
    e = lambda: a  # function scope
    f = lambda a=a: a  # default argument uses class scope

    @staticmethod  # or @classmethod, or regular instance method
    def g():  # function scope
        return a

print(Fred.a)  # class
print(next(Fred.b))  # global
print(Fred.c[0])  # class in Python 2, global in Python 3
print(Fred.d)  # class
print(Fred.e())  # global
print(Fred.f())  # class
print(Fred.g()) # global
```

Gli utenti che non hanno familiarità con il funzionamento di questo scope potrebbero aspettarsi che `b` , `c` ed `e` stampino la `class` .

Da PEP 227 :

> I nomi nella portata della classe non sono accessibili. I nomi sono risolti nello scope della funzione di chiusura più interna. Se una definizione di classe si verifica in una catena di ambiti nidificati, il processo di risoluzione ignora le definizioni di classe.

Dalla documentazione di Python su denominazione e associazione :

> L'ambito dei nomi definiti in un blocco di classe è limitato al blocco di classe; non si estende ai blocchi di codice dei metodi - questo include le comprensioni e le espressioni del generatore poiché sono implementati usando un ambito di funzione. Ciò significa che il seguente non funzionerà:

```
class A:
    a = 42
    b = list(a + i for i in range(10))
```

Questo esempio utilizza i riferimenti di questa risposta di Martijn Pieters, che contiene un'analisi più approfondita di questo comportamento.

Il comando

Questo comando ha diverse forme correlate ma distinte.

`del v`

Se `v` è una variabile, il comando `del v` rimuove la variabile dal suo ambito. Per esempio:

```
x = 5
print(x) # out: 5
del x
print(x) # NameError: name 'f' is not defined
```

> Si noti che `del` è *un'occorrenza di associazione* , il che significa che, a meno che non sia specificato esplicitamente altrimenti (usando `nonlocal` o `global`), `del v` renderà `v` local allo scope corrente. Se si intende eliminare `v` in un ambito esterno, utilizzare `nonlocal v` `global v` o `global v` nello stesso ambito dell'istruzione `del v` .

In tutto quanto segue, l'intenzione di un comando è un comportamento predefinito, ma non viene applicato dalla lingua. Una classe potrebbe essere scritta in modo tale da invalidare questa intenzione.

`del v.name`

Questo comando attiva una chiamata a `v.__delattr__(name)` .

L'intenzione è di rendere il `name` dell'attributo `name` disponibile. Per esempio:

```
class A:
    pass

a = A()
a.x = 7
print(a.x) # out: 7
del a.x
print(a.x) # error: AttributeError: 'A' object has no attribute 'x'
```

`del v[item]`

Questo comando attiva una chiamata a `v.__delitem__(item)` .

L'intenzione è che l' `item` non apparterrà alla mappatura implementata dall'oggetto `v` . Per

esempio:

```
x = {'a': 1, 'b': 2}
del x['a']
print(x) #  out: {'b': 2}
print(x['a']) # error: KeyError: 'a'
```

del v[a:b]

Questo in realtà chiama `v.__delslice__(a, b)` .

L'intenzione è simile a quella descritta sopra, ma con sezioni: intervalli di elementi anziché un singolo elemento. Per esempio:

```
x = [0, 1, 2, 3, 4]
del x[1:3]
print(x) #  out: [0, 3, 4]
```

Vedi anche Garbage Collection # Il comando del .

Local vs Global Scope

Quali sono gli obiettivi locali e globali?

Tutte le variabili di Python che sono accessibili ad un certo punto nel codice sono in *ambito locale* o nell'ambito *globale* .

La spiegazione è che l'ambito locale include tutte le variabili definite nella funzione corrente e l'ambito globale include variabili definite al di fuori della funzione corrente.

```
foo = 1  # global

def func():
    bar = 2  # local
    print(foo)  # prints variable foo from global scope
    print(bar)  # prints variable bar from local scope
```

Si può ispezionare quali variabili sono in quale ambito. Le funzioni built-in `locals()` e `globals()` restituiscono l'intero scope come dizionari.

```
foo = 1

def func():
    bar = 2
    print(globals().keys())  # prints all variable names in global scope
    print(locals().keys())  # prints all variable names in local scope
```

Cosa succede con le scontro sul nome?

```
foo = 1

def func():
    foo = 2  # creates a new variable foo in local scope, global foo is not affected

    print(foo)  # prints 2

    # global variable foo still exists, unchanged:
    print(globals()['foo'])  # prints 1
    print(locals()['foo'])  # prints 2
```

Per modificare una variabile globale, utilizzare la parola chiave `global` :

```
foo = 1

def func():
    global foo
    foo = 2  # this modifies the global foo, rather than creating a local variable
```

L'ambito è definito per l'intero corpo della funzione!

Ciò che significa è che una variabile non sarà mai globale per metà della funzione e locale in seguito, o viceversa.

```
foo = 1

def func():
    # This function has a local variable foo, because it is defined down below.
    # So, foo is local from this point. Global foo is hidden.

    print(foo) # raises UnboundLocalError, because local foo is not yet initialized
    foo = 7
    print(foo)
```

Allo stesso modo, la oposite:

```
foo = 1

def func():
    # In this function, foo is a global variable from the begining

    foo = 7  # global foo is modified

    print(foo)  # 7
    print(globals()['foo'])  # 7

    global foo  # this could be anywhere within the function
    print(foo)  # 7
```

Funzioni all'interno delle funzioni

Possono esserci molti livelli di funzioni annidati all'interno delle funzioni, ma all'interno di una funzione c'è solo un ambito locale per quella funzione e l'ambito globale. Non ci sono ambiti

intermedi.

```
foo = 1

def f1():
    bar = 1

    def f2():
        baz = 2
        # here, foo is a global variable, baz is a local variable
        # bar is not in either scope
        print(locals().keys())  # ['baz']
        print('bar' in locals())  # False
        print('bar' in globals())  # False

    def f3():
        baz = 3
        print(bar)  # bar from f1 is referenced so it enters local scope of f3 (closure)
        print(locals().keys())  # ['bar', 'baz']
        print('bar' in locals())  # True
        print('bar' in globals())  # False

    def f4():
        bar = 4  # a new local bar which hides bar from local scope of f1
        baz = 4
        print(bar)
        print(locals().keys())  # ['bar', 'baz']
        print('bar' in locals())  # True
        print('bar' in globals())  # False
```

`global` vs `nonlocal` (solo Python 3)

Entrambe queste parole chiave sono utilizzate per ottenere l'accesso in scrittura a variabili che non sono locali alle funzioni correnti.

La parola chiave `global` dichiara che un nome dovrebbe essere trattato come una variabile globale.

```
foo = 0  # global foo

def f1():
    foo = 1  # a new foo local in f1

    def f2():
        foo = 2  # a new foo local in f2

        def f3():
            foo = 3  # a new foo local in f3
            print(foo)  # 3
            foo = 30  # modifies local foo in f3 only

        def f4():
            global foo
            print(foo)  # 0
            foo = 100  # modifies global foo
```

D'altra parte, `nonlocal` (vedi Variabili `nonlocal`), disponibile in Python 3, prende una variabile *locale* da un ambito che racchiude l'ambito locale della funzione corrente.

Dalla documentazione Python su `nonlocal` :

> L'istruzione nonlocale fa sì che gli identificatori elencati facciano riferimento a variabili associate in precedenza nello scope che racchiude il più vicino escludendo le globali.

Python 3.x 3.0

```
def f1():

    def f2():
        foo = 2  # a new foo local in f2

        def f3():
            nonlocal foo  # foo from f2, which is the nearest enclosing scope
            print(foo)  # 2
            foo = 20  # modifies foo from f2!
```

Leggi Ambito e legame variabili online: https://riptutorial.com/it/python/topic/263/ambito-e-legame-variabili

Capitolo 13: Analisi HTML

Examples

Trova un testo dopo un elemento in BeautifulSoup

Immagina di avere il seguente codice HTML:

```
<div>
    <label>Name:</label>
    John Smith
</div>
```

E devi localizzare il testo "John Smith" dopo l'elemento `label` .

In questo caso, è possibile individuare l'elemento `label` base al testo e quindi utilizzare la proprietà `.next_sibling` :

```
from bs4 import BeautifulSoup

data = """
<div>
    <label>Name:</label>
    John Smith
</div>
"""

soup = BeautifulSoup(data, "html.parser")

label = soup.find("label", text="Name:")
print(label.next_sibling.strip())
```

Stampa `John Smith` .

Utilizzo dei selettori CSS in BeautifulSoup

BeautifulSoup ha un supporto limitato per i selettori CSS , ma copre quelli più comunemente usati. Usa il metodo `select()` per trovare più elementi e `select_one()` per trovare un singolo elemento.

Esempio di base:

```
from bs4 import BeautifulSoup

data = """
<ul>
    <li class="item">item1</li>
    <li class="item">item2</li>
    <li class="item">item3</li>
</ul>
"""
```

```
soup = BeautifulSoup(data, "html.parser")

for item in soup.select("li.item"):
    print(item.get_text())
```

stampe:

```
item1
item2
item3
```

PyQuery

pyquery è una libreria jquery per Python. Ha un ottimo supporto per i selettori di css.

```
from pyquery import PyQuery

html = """
<h1>Sales</h1>
<table id="table">
<tr>
    <td>Lorem</td>
    <td>46</td>
</tr>
<tr>
    <td>Ipsum</td>
    <td>12</td>
</tr>
<tr>
    <td>Dolor</td>
    <td>27</td>
</tr>
<tr>
    <td>Sit</td>
    <td>90</td>
</tr>
</table>
"""

doc = PyQuery(html)

title = doc('h1').text()

print title

table_data = []

rows = doc('#table > tr')
for row in rows:
    name = PyQuery(row).find('td').eq(0).text()
    value = PyQuery(row).find('td').eq(1).text()

    print "%s\t  %s" % (name, value)
```

Leggi Analisi HTML online: https://riptutorial.com/it/python/topic/1384/analisi-html

Capitolo 14: Analizzare gli argomenti della riga di comando

introduzione

La maggior parte degli strumenti a riga di comando si basano su argomenti passati al programma al momento dell'esecuzione. Invece di richiedere input, questi programmi si aspettano che vengano impostati dati o flag specifici (che diventano booleani). Ciò consente sia all'utente che ad altri programmi di eseguire il file Python passandogli i dati all'avvio. Questa sezione spiega e dimostra l'implementazione e l'uso degli argomenti della riga di comando in Python.

Examples

Ciao mondo in argparse

Il seguente programma dice ciao all'utente. Prende un argomento posizionale, il nome dell'utente e può anche essere detto il saluto.

```
import argparse

parser = argparse.ArgumentParser()

parser.add_argument('name',
    help='name of user'
)

parser.add_argument('-g', '--greeting',
    default='Hello',
    help='optional alternate greeting'
)

args = parser.parse_args()

print("{greeting}, {name}!".format(
       greeting=args.greeting,
       name=args.name)
)
```

```
$ python hello.py --help
usage: hello.py [-h] [-g GREETING] name

positional arguments:
  name                  name of user

optional arguments:
  -h, --help            show this help message and exit
  -g GREETING, --greeting GREETING
                        optional alternate greeting
```

```
$ python hello.py world
```

```
Hello, world!
$ python hello.py John -g Howdy
Howdy, John!
```

Per maggiori dettagli si prega di leggere la documentazione di argparse .

Esempio di base con docopt

docopt trasforma l'argomento della riga di comando in analisi. Invece di analizzare gli argomenti, basta **scrivere la stringa di utilizzo** per il proprio programma e docopt **analizza la stringa di utilizzo** e la utilizza per estrarre gli argomenti della riga di comando.

```
"""
Usage:
    script_name.py [-a] [-b] <path>

Options:
    -a            Print all the things.
    -b            Get more bees into the path.
"""
from docopt import docopt


if __name__ == "__main__":
    args = docopt(__doc__)
    import pprint; pprint.pprint(args)
```

Esecuzioni di esempio:

```
$ python script_name.py
Usage:
    script_name.py [-a] [-b] <path>
$ python script_name.py something
{'-a': False,
 '-b': False,
 '<path>': 'something'}
$ python script_name.py something -a
{'-a': True,
 '-b': False,
 '<path>': 'something'}
$ python script_name.py -b something -a
{'-a': True,
 '-b': True,
 '<path>': 'something'}
```

Impostazione di argomenti mutuamente esclusivi con argparse

Se vuoi che due o più argomenti si escludano a vicenda. È possibile utilizzare la funzione `argparse.ArgumentParser.add_mutually_exclusive_group()` . Nell'esempio seguente, possono esistere foo o bar ma non entrambi allo stesso tempo.

```
import argparse

parser = argparse.ArgumentParser()
```

```
group = parser.add_mutually_exclusive_group()
group.add_argument("-f", "--foo")
group.add_argument("-b", "--bar")
args = parser.parse_args()
print "foo = ", args.foo
print "bar = ", args.bar
```

Se si tenta di eseguire lo script specificando entrambi `--bar` argomenti `--foo` e `--bar` , lo script si lamenterà con il messaggio seguente.

```
error: argument -b/--bar: not allowed with argument -f/--foo
```

Utilizzo degli argomenti della riga di comando con argv

Ogni volta che uno script Python viene richiamato dalla riga di comando, l'utente può fornire ulteriori **argomenti della riga di comando** che verranno passati allo script. Questi argomenti saranno disponibili per il programmatore dalla variabile di sistema `sys.argv` ("argv" è un nome tradizionale utilizzato nella maggior parte dei linguaggi di programmazione e significa " **arg** ument **v** ector").

Per convenzione, il primo elemento nell'elenco `sys.argv` è il nome dello stesso script Python, mentre il resto degli elementi sono i token passati dall'utente quando si richiama lo script.

```
# cli.py
import sys
print(sys.argv)

$ python cli.py
=> ['cli.py']

$ python cli.py fizz
=> ['cli.py', 'fizz']

$ python cli.py fizz buzz
=> ['cli.py', 'fizz', 'buzz']
```

Ecco un altro esempio di come usare `argv` . Prima rimuoviamo l'elemento iniziale di sys.argv perché contiene il nome dello script. Quindi combiniamo il resto degli argomenti in una singola frase e infine stampiamo quella frase anteponendo il nome dell'utente attualmente loggato (in modo che emuli un programma di chat).

```
import getpass
import sys

words = sys.argv[1:]
sentence = " ".join(words)
print("[%s] %s" % (getpass.getuser(), sentence))
```

L'algoritmo comunemente usato quando "manualmente" l'analisi di un numero di argomenti non posizionali è quello di scorrere l'elenco `sys.argv` . Un modo è quello di andare oltre la lista e pop ogni elemento di esso:

```
# reverse and copy sys.argv
argv = reversed(sys.argv)
# extract the first element
arg = argv.pop()
# stop iterating when there's no more args to pop()
while len(argv) > 0:
    if arg in ('-f', '--foo'):
        print('seen foo!')
    elif arg in ('-b', '--bar'):
        print('seen bar!')
    elif arg in ('-a', '--with-arg'):
        arg = arg.pop()
        print('seen value: {}'.format(arg))
    # get the next value
    arg = argv.pop()
```

Messaggio di errore del parser personalizzato con argparse

È possibile creare messaggi di errore parser in base alle esigenze del proprio script. Questo è attraverso la funzione `argparse.ArgumentParser.error` . L'esempio seguente mostra lo script stampa di un utilizzo e un messaggio di errore a `stderr` quando `--foo` è dato, ma non `--bar` .

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument("-f", "--foo")
parser.add_argument("-b", "--bar")
args = parser.parse_args()
if args.foo and args.bar is None:
    parser.error("--foo requires --bar. You did not specify bar.")

print "foo =", args.foo
print "bar =", args.bar
```

Supponendo che il nome dello script sia sample.py, eseguiamo: `python sample.py --foo ds_in_fridge`

Lo script si lamenterà con quanto segue:

```
usage: sample.py [-h] [-f FOO] [-b BAR]
sample.py: error: --foo requires --bar. You did not specify bar.
```

Raggruppamento concettuale di argomenti con argparse.add_argument_group ()

Quando crei Argparse ArgumentParser () ed esegui il tuo programma con '-h' ottieni un messaggio di utilizzo automatico che spiega quali argomenti puoi eseguire con il tuo software. Per impostazione predefinita, gli argomenti posizionali e gli argomenti condizionali sono separati in due categorie, ad esempio, qui c'è un piccolo script (example.py) e l'output quando si esegue `python example.py -h` .

```
import argparse
```

```
parser = argparse.ArgumentParser(description='Simple example')
parser.add_argument('name', help='Who to greet', default='World')
parser.add_argument('--bar_this')
parser.add_argument('--bar_that')
parser.add_argument('--foo_this')
parser.add_argument('--foo_that')
args = parser.parse_args()
```

```
usage: example.py [-h] [--bar_this BAR_THIS] [--bar_that BAR_THAT]
                      [--foo_this FOO_THIS] [--foo_that FOO_THAT]
                      name

Simple example

positional arguments:
  name                  Who to greet

optional arguments:
  -h, --help            show this help message and exit
  --bar_this BAR_THIS
  --bar_that BAR_THAT
  --foo_this FOO_THIS
  --foo_that FOO_THAT
```

Ci sono alcune situazioni in cui vuoi separare i tuoi argomenti in ulteriori sezioni concettuali per aiutare il tuo utente. Ad esempio, potresti desiderare di avere tutte le opzioni di input in un gruppo e tutte le opzioni di output in un'altra. L'esempio sopra può essere regolato per separare gli `--foo_*` dagli argomenti `--bar_*` modo.

```
import argparse

parser = argparse.ArgumentParser(description='Simple example')
parser.add_argument('name', help='Who to greet', default='World')
# Create two argument groups
foo_group = parser.add_argument_group(title='Foo options')
bar_group = parser.add_argument_group(title='Bar options')
# Add arguments to those groups
foo_group.add_argument('--bar_this')
foo_group.add_argument('--bar_that')
bar_group.add_argument('--foo_this')
bar_group.add_argument('--foo_that')
args = parser.parse_args()
```

Che produce questo output quando `python example.py -h` viene eseguito:

```
usage: example.py [-h] [--bar_this BAR_THIS] [--bar_that BAR_THAT]
                      [--foo_this FOO_THIS] [--foo_that FOO_THAT]
                      name

Simple example

positional arguments:
  name                  Who to greet

optional arguments:
  -h, --help            show this help message and exit
```

```
Foo options:
  --bar_this BAR_THIS
  --bar_that BAR_THAT

Bar options:
  --foo_this FOO_THIS
  --foo_that FOO_THAT
```

Esempio avanzato con docopt e docopt_dispatch

Come con docopt, con [docopt_dispatch] costruisci il tuo `--help` nella variabile `__doc__` del tuo modulo entry-point. Lì, si chiama `dispatch` con la stringa doc come argomento, quindi può eseguire il parser su di esso.

Fatto questo, invece di gestire manualmente gli argomenti (che di solito finiscono in una struttura ciclomatic se / else alta), lo si lascia alla distribuzione dando solo come si desidera gestire l'insieme di argomenti.

Questo è il motivo per cui è utilizzato il decoratore `dispatch.on` : gli si fornisce l'argomento o la sequenza di argomenti che devono attivare la funzione e tale funzione verrà eseguita con i valori corrispondenti come parametri.

```
"""Run something in development or production mode.

Usage: run.py --development <host> <port>
       run.py --production <host> <port>
       run.py items add <item>
       run.py items delete <item>

"""
from docopt_dispatch import dispatch

@dispatch.on('--development')
def development(host, port, **kwargs):
    print('in *development* mode')

@dispatch.on('--production')
def development(host, port, **kwargs):
    print('in *production* mode')

@dispatch.on('items', 'add')
def items_add(item, **kwargs):
    print('adding item...')

@dispatch.on('items', 'delete')
def items_delete(item, **kwargs):
    print('deleting item...')

if __name__ == '__main__':
    dispatch(__doc__)
```

Leggi Analizzare gli argomenti della riga di comando online:
https://riptutorial.com/it/python/topic/1382/analizzare-gli-argomenti-della-riga-di-comando

Capitolo 15: ArcPy

Osservazioni

Questo esempio utilizza un cursore di ricerca dal modulo Data Access (da) di ArcPy.

Non confondere la sintassi arcpy.da.SearchCursor con il file arcpy.SearchCursor () precedente e più lento.

Il modulo di accesso ai dati (arcpy.da) è disponibile solo da ArcGIS 10.1 per Desktop.

Examples

Stampa del valore di un campo per tutte le righe della feature class nel geodatabase di file utilizzando il cursore di ricerca

Per stampare un campo test (TestField) da una feature feature di prova (TestFC) in un file di test geodatabase (Test.gdb) situato in una cartella temporanea (C: \ Temp):

```
with arcpy.da.SearchCursor(r"C:\Temp\Test.gdb\TestFC",["TestField"]) as cursor:
    for row in cursor:
        print row[0]
```

createDissolvedGDB per creare un file gdb nell'area di lavoro

```
def createDissolvedGDB(workspace, gdbName):
    gdb_name = workspace + "/" + gdbName + ".gdb"

    if(arcpy.Exists(gdb_name):
        arcpy.Delete_management(gdb_name)
        arcpy.CreateFileGDB_management(workspace, gdbName, "")
    else:
        arcpy.CreateFileGDB_management(workspace, gdbName, "")

    return gdb_name
```

Leggi ArcPy online: https://riptutorial.com/it/python/topic/4693/arcpy

Capitolo 16: Array

introduzione

Gli "array" in Python non sono gli array nei linguaggi di programmazione convenzionali come C e Java, ma più vicini alle liste. Una lista può essere una raccolta di elementi omogenei o eterogenei e può contenere inte, stringhe o altri elenchi.

Parametri

Parametro	Dettagli
`b`	Rappresenta numero intero con segno di dimensione 1 byte
`B`	Rappresenta un numero intero senza segno di dimensione 1 byte
`c`	Rappresenta il carattere di dimensione 1 byte
`u`	Rappresenta un carattere unicode di dimensione 2 byte
`h`	Rappresenta numero intero con segno di dimensione 2 byte
`H`	Rappresenta numero intero senza segno di dimensione 2 byte
`i`	Rappresenta numero intero con segno di dimensione 2 byte
`I`	Rappresenta numero intero senza segno di dimensione 2 byte
`w`	Rappresenta il carattere unicode di dimensione 4 byte
`l`	Rappresenta numero intero con segno di dimensione 4 byte
`L`	Rappresenta un numero intero senza segno di dimensione 4 byte
`f`	Rappresenta il punto mobile di dimensione 4 byte
`d`	Rappresenta il punto mobile di dimensioni 8 byte

Examples

Introduzione di base agli array

Una matrice è una struttura di dati che memorizza valori dello stesso tipo di dati. In Python, questa è la principale differenza tra array e liste.

Mentre gli elenchi python possono contenere valori corrispondenti a diversi tipi di dati, gli array in

python possono contenere solo valori corrispondenti allo stesso tipo di dati. In questo tutorial, comprenderemo gli array Python con alcuni esempi.

Se sei nuovo in Python, inizia con l'articolo Introduzione di Python.

Per utilizzare gli array in linguaggio Python, è necessario importare il modulo `array` standard. Questo perché l'array non è un tipo di dati fondamentale come stringhe, interi, ecc. Ecco come è possibile importare il modulo `array` in python:

```
from array import *
```

Una volta importato il modulo `array` , è possibile dichiarare un array. Ecco come lo fai:

```
arrayIdentifierName = array(typecode, [Initializers])
```

Nella dichiarazione precedente, `arrayIdentifierName` è il nome dell'array, `typecode` consente a python di conoscere il tipo di array e `Initializers` sono i valori con cui viene inizializzato l'array.

I codici di errore sono i codici utilizzati per definire il tipo di valori di matrice o il tipo di matrice. La tabella nella sezione parametri mostra i possibili valori che puoi usare quando dichiari un array e il suo tipo.

Ecco un esempio reale di dichiarazione dell'array Python:

```
my_array = array('i',[1,2,3,4])
```

Nell'esempio sopra, il codice di battitura usato è `i` . Questo codice di errore rappresenta il numero intero con segno la cui dimensione è 2 byte.

Ecco un semplice esempio di un array contenente 5 numeri interi

```
from array import *
my_array = array('i', [1,2,3,4,5])
for i in my_array:
    print(i)
# 1
# 2
# 3
# 4
# 5
```

Accedi a singoli elementi tramite indici

È possibile accedere a singoli elementi tramite gli indici. Gli array Python sono a zero indici. Ecco un esempio:

```
my_array = array('i', [1,2,3,4,5])
print(my_array[1])
# 2
print(my_array[2])
```

```
# 3
print(my_array[0])
# 1
```

Aggiungi qualsiasi valore all'array usando il metodo append ()

```
my_array = array('i', [1,2,3,4,5])
my_array.append(6)
# array('i', [1, 2, 3, 4, 5, 6])
```

Si noti che il valore `6` stato aggiunto ai valori di array esistenti.

Inserire il valore in un array usando il metodo insert ()

Possiamo usare il metodo `insert()` per inserire un valore in qualsiasi indice dell'array. Ecco un esempio:

```
my_array = array('i', [1,2,3,4,5])
my_array.insert(0,0)
#array('i', [0, 1, 2, 3, 4, 5])
```

Nell'esempio precedente, il valore 0 è stato inserito nell'indice 0. Si noti che il primo argomento è l'indice mentre il secondo argomento è il valore.

Estendi array Python usando il metodo extend ()

Un array python può essere esteso con più di un valore usando il metodo `extend()` . Ecco un esempio:

```
my_array = array('i', [1,2,3,4,5])
my_extnd_array = array('i', [7,8,9,10])
my_array.extend(my_extnd_array)
# array('i', [1, 2, 3, 4, 5, 7, 8, 9, 10])
```

Vediamo che l'array my_array è stato esteso con i valori di `my_extnd_array` .

Aggiungi elementi dalla lista alla matrice usando il metodo fromlist ()

Ecco un esempio:

```
my_array = array('i', [1,2,3,4,5])
c=[11,12,13]
my_array.fromlist(c)
# array('i', [1, 2, 3, 4, 5, 11, 12, 13])
```

Quindi vediamo che i valori 11,12 e 13 sono stati aggiunti dalla lista `c` a `my_array` .

Rimuovi qualsiasi elemento dell'array usando il metodo remove ()

Ecco un esempio:

```
my_array = array('i', [1,2,3,4,5])
my_array.remove(4)
# array('i', [1, 2, 3, 5])
```

Vediamo che l'elemento 4 è stato rimosso dall'array.

Rimuovi l'ultimo elemento dell'array usando il metodo pop ()

`pop` rimuove l'ultimo elemento dalla matrice. Ecco un esempio:

```
my_array = array('i', [1,2,3,4,5])
my_array.pop()
# array('i', [1, 2, 3, 4])
```

Quindi vediamo che l'ultimo elemento (`5`) è stato estratto dalla matrice.

Recupera qualsiasi elemento attraverso il suo indice usando il metodo index ()

`index()` restituisce il primo indice del valore corrispondente. Ricorda che gli array sono a zero indice.

```
my_array = array('i', [1,2,3,4,5])
print(my_array.index(5))
# 5
my_array = array('i', [1,2,3,3,5])
print(my_array.index(3))
# 3
```

Si noti nel secondo esempio che è stato restituito un solo indice, anche se il valore esiste due volte nella matrice

Invertire un array python usando il metodo reverse ()

Il metodo `reverse()` fa ciò che il nome dice che farà - inverte l'array. Ecco un esempio:

```
my_array = array('i', [1,2,3,4,5])
my_array.reverse()
# array('i', [5, 4, 3, 2, 1])
```

Ottieni informazioni sul buffer dell'array tramite il metodo buffer_info ()

Questo metodo fornisce l'indirizzo iniziale del buffer dell'array in memoria e il numero di elementi nell'array. Ecco un esempio:

```
my_array = array('i', [1,2,3,4,5])
my_array.buffer_info()
```

```
(33881712, 5)
```

Controlla il numero di occorrenze di un elemento usando il metodo count ()

`count()` restituirà il numero di volte e l'elemento appare in una matrice. Nell'esempio seguente vediamo che il valore `3` verifica due volte.

```
my_array = array('i', [1,2,3,3,5])
my_array.count(3)
# 2
```

Converti array in string usando il metodo tostring ()

`tostring()` converte l'array in una stringa.

```
my_char_array = array('c', ['g','e','e','k'])
# array('c', 'geek')
print(my_char_array.tostring())
# geek
```

Converti array in un elenco python con gli stessi elementi usando il metodo tolist ()

Quando hai bisogno di un oggetto `list` Python, puoi utilizzare il metodo `tolist()` per convertire la tua matrice in una lista.

```
my_array = array('i', [1,2,3,4,5])
c = my_array.tolist()
# [1, 2, 3, 4, 5]
```

Aggiungi una stringa al char array usando il metodo fromstring ()

Puoi aggiungere una stringa a un array di caratteri usando `fromstring()`

```
my_char_array = array('c', ['g','e','e','k'])
my_char_array.fromstring("stuff")
print(my_char_array)
#array('c', 'geekstuff')
```

Leggi Array online: https://riptutorial.com/it/python/topic/4866/array

Capitolo 17: Audio

Examples

Audio con Pyglet

```
import pyglet
audio = pyglet.media.load("audio.wav")
audio.play()
```

Per ulteriori informazioni, vedere pyglet

Lavorare con i file WAV

winsound

- Ambiente Windows

```
import winsound
winsound.PlaySound("path_to_wav_file.wav", winsound.SND_FILENAME)
```

onda

- Supporto mono / stereo
- Non supporta compressione / decompressione

```
import wave
with wave.open("path_to_wav_file.wav", "rb") as wav_file:    # Open WAV file in read-only
mode.
    # Get basic information.
    n_channels = wav_file.getnchannels()      # Number of channels. (1=Mono, 2=Stereo).
    sample_width = wav_file.getsampwidth()    # Sample width in bytes.
    framerate = wav_file.getframerate()       # Frame rate.
    n_frames = wav_file.getnframes()          # Number of frames.
    comp_type = wav_file.getcomptype()        # Compression type (only supports "NONE").
    comp_name = wav_file.getcompname()        # Compression name.

    # Read audio data.
    frames = wav_file.readframes(n_frames)    # Read n_frames new frames.
    assert len(frames) == sample_width * n_frames

# Duplicate to a new WAV file.
with wave.open("path_to_new_wav_file.wav", "wb") as wav_file:    # Open WAV file in write-only
mode.
    # Write audio data.
    params = (n_channels, sample_width, framerate, n_frames, comp_type, comp_name)
    wav_file.setparams(params)
    wav_file.writeframes(frames)
```

Converti qualsiasi file sonoro con python e ffmpeg

```
from subprocess import check_call

ok = check_call(['ffmpeg','-i','input.mp3','output.wav'])
if ok:
    with open('output.wav', 'rb') as f:
        wav_file = f.read()
```

Nota:

- http://superuser.com/questions/507386/why-would-i-choose-libav-over-ffmpeg-or-is-there-even-a-difference
- Quali sono le differenze e le somiglianze tra ffmpeg, libav e avconv?

Riproduzione dei beep di Windows

Windows fornisce un'interfaccia esplicita attraverso la quale il modulo `winsound` ti consente di suonare beep grezzi a una data frequenza e durata.

```
import winsound
freq = 2500 # Set frequency To 2500 Hertz
dur = 1000 # Set duration To 1000 ms == 1 second
winsound.Beep(freq, dur)
```

Leggi Audio online: https://riptutorial.com/it/python/topic/8189/audio

Capitolo 18: Aumentare errori / eccezioni personalizzati

introduzione

Python ha molte eccezioni built-in che costringono il tuo programma a generare un errore quando qualcosa in esso va storto.

Tuttavia, a volte potrebbe essere necessario creare eccezioni personalizzate che servono al tuo scopo.

In Python, gli utenti possono definire tali eccezioni creando una nuova classe. Questa classe di eccezioni deve essere derivata, direttamente o indirettamente, dalla classe Exception. La maggior parte delle eccezioni built-in derivano anche da questa classe.

Examples

Eccezione personalizzata

Qui, abbiamo creato un'eccezione definita dall'utente, CustomError, derivata dalla classe Exception. Questa nuova eccezione può essere sollevata, come altre eccezioni, usando l'istruzione raise con un messaggio di errore opzionale.

```
class CustomError(Exception):
       pass

x = 1

if x == 1:
    raise CustomError('This is custom error')
```

Produzione:

```
Traceback (most recent call last):
  File "error_custom.py", line 8, in <module>
    raise CustomError('This is custom error')
__main__.CustomError: This is custom error
```

Cattura eccezione personalizzata

Questo esempio mostra come catturare l'eccezione personalizzata

```
class CustomError(Exception):
     pass

try:
    raise CustomError('Can you catch me ?')
```

```
except CustomError as e:
    print ('Catched CustomError :{}'.format(e))
except Exception as e:
    print ('Generic exception: {}'.format(e))
```

Produzione:

```
Catched CustomError :Can you catch me ?
```

Leggi Aumentare errori / eccezioni personalizzati online:
https://riptutorial.com/it/python/topic/10882/aumentare-errori---eccezioni-personalizzati

Capitolo 19: Blocchi di codice, frame di esecuzione e spazi dei nomi

introduzione

Un blocco di codice è un pezzo di testo di programma Python che può essere eseguito come un'unità, come un modulo, una definizione di classe o un corpo di una funzione. Alcuni blocchi di codice (come i moduli) vengono normalmente eseguiti solo una volta, altri (come i corpi delle funzioni) possono essere eseguiti molte volte. I blocchi di codice possono contenere testualmente altri blocchi di codice. I blocchi di codice possono richiamare altri blocchi di codice (che possono o non possono essere contenuti testualmente in essi) come parte della loro esecuzione, ad esempio invocando (chiamando) una funzione.

Examples

Spazi dei nomi di codice

Tipo di blocco di codice	Spazio dei nomi globale	Spazio dei nomi locale
Modulo	ns per il modulo	uguale a globale
Script (file o comando)	ns per `__main__`	uguale a globale
Comando interattivo	ns per `__main__`	uguale a globale
Definizione della classe	ns globale del blocco contenitore	nuovo spazio dei nomi
Corpo della funzione	ns globale del blocco contenitore	nuovo spazio dei nomi
Stringa passata alla dichiarazione `exec`	ns globale del blocco contenitore	spazio dei nomi locale del blocco contenitore
Stringa passata a `eval()`	ns globale di chi chiama	local ns of caller
File letto da `execfile()`	ns globale di chi chiama	local ns of caller
Espressione letta da `input()`	ns globale di chi chiama	local ns of caller

Leggi Blocchi di codice, frame di esecuzione e spazi dei nomi online: https://riptutorial.com/it/python/topic/10741/blocchi-di-codice--frame-di-esecuzione-e-spazi-dei-nomi

Capitolo 20: Calcolo parallelo

Osservazioni

A causa del GIL (Global interpreter lock), solo un'istanza dell'interprete python viene eseguita in un singolo processo. Quindi, in generale, l'utilizzo di multi-threading migliora solo i calcoli associati all'IO, non quelli legati alla CPU. Il modulo `multiprocessing` è consigliato se si desidera parallelizzare le attività legate alla CPU.

GIL si applica a CPython, l'implementazione più popolare di Python, oltre a PyPy. Altre implementazioni come Jython e IronPython non hanno GIL .

Examples

Utilizzo del modulo multiprocessing per parallelizzare le attività

```
import multiprocessing

def fib(n):
    """computing the Fibonacci in an inefficient way
    was chosen to slow down the CPU."""
    if n <= 2:
        return 1
    else:
        return fib(n-1)+fib(n-2)
p = multiprocessing.Pool()
print(p.map(fib,[38,37,36,35,34,33]))

# Out: [39088169, 24157817, 14930352, 9227465, 5702887, 3524578]
```

Poiché l'esecuzione di ogni chiamata a `fib` avviene in parallelo, il tempo di esecuzione dell'esempio completo è **1,8 × più veloce** rispetto a se eseguito in modo sequenziale su un processore doppio.

Python 2.2+

Utilizzo degli script padre e figlio per eseguire il codice in parallelo

child.py

```
import time

def main():
    print "starting work"
    time.sleep(1)
    print "work work work work work"
    time.sleep(1)
    print "done working"

if __name__ == '__main__':
```

```
    main()
```

parent.py

```
import os

def main():
    for i in range(5):
        os.system("python child.py &")

if __name__ == '__main__':
    main()
```

Questo è utile per attività di richiesta / risposta HTTP parallele e indipendenti o selezione / inserimenti del database. Gli argomenti della riga di comando possono essere dati anche allo script **child.py** . La sincronizzazione tra gli script può essere ottenuta da tutti gli script che controllano regolarmente un server separato (come un'istanza Redis).

Utilizzo di un'estensione C per parallelizzare le attività

L'idea qui è di spostare i lavori ad alta intensità di calcolo su C (usando macro speciali), indipendentemente da Python, e rilasciare il codice C al GIL mentre funziona.

```
#include "Python.h"
...
PyObject *pyfunc(PyObject *self, PyObject *args) {
    ...
    Py_BEGIN_ALLOW_THREADS
    // Threaded C code
    ...
    Py_END_ALLOW_THREADS
    ...
}
```

Utilizzando il modulo PyPar per parallelizzare

PyPar è una libreria che utilizza l'interfaccia di passaggio dei messaggi (MPI) per fornire il parallelismo in Python. Un semplice esempio in PyPar (come visto su https://github.com/daleroberts/pypar) assomiglia a questo:

```
import pypar as pp

ncpus = pp.size()
rank = pp.rank()
node = pp.get_processor_name()

print 'I am rank %d of %d on node %s' % (rank, ncpus, node)

if rank == 0:
  msh = 'P0'
  pp.send(msg, destination=1)
  msg = pp.receive(source=rank-1)
  print 'Processor 0 received message "%s" from rank %d' % (msg, rank-1)
```

```
else:
  source = rank-1
  destination = (rank+1) % ncpus
  msg = pp.receive(source)
  msg = msg + 'P' + str(rank)
  pypar.send(msg, destination)
pp.finalize()
```

Leggi Calcolo parallelo online: https://riptutorial.com/it/python/topic/542/calcolo-parallelo

Capitolo 21: Caratteristiche nascoste

Examples

Sovraccarico dell'operatore

Tutto in Python è un oggetto. Ogni oggetto ha alcuni metodi interni speciali che usa per interagire con altri oggetti. In generale, questi metodi seguono la convenzione di denominazione `__action__` . Collettivamente, questo è definito come il modello di dati Python .

Puoi sovraccaricare *uno* di questi metodi. Questo è comunemente usato nel sovraccarico dell'operatore in Python. Di seguito è riportato un esempio di overloading dell'operatore utilizzando il modello di dati Python. La classe `Vector` crea un vettore semplice di due variabili. Aggiungeremo il supporto appropriato per le operazioni matematiche di due vettori utilizzando l'overloading dell'operatore.

```
class Vector(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, v):
        # Addition with another vector.
        return Vector(self.x + v.x, self.y + v.y)

    def __sub__(self, v):
        # Subtraction with another vector.
        return Vector(self.x - v.x, self.y - v.y)

    def __mul__(self, s):
        # Multiplication with a scalar.
        return Vector(self.x * s, self.y * s)

    def __div__(self, s):
        # Division with a scalar.
        float_s = float(s)
        return Vector(self.x / float_s, self.y / float_s)

    def __floordiv__(self, s):
        # Division with a scalar (value floored).
        return Vector(self.x // s, self.y // s)

    def __repr__(self):
        # Print friendly representation of Vector class. Else, it would
        # show up like, <__main__.Vector instance at 0x01DDDDC8>.
        return '<Vector (%f, %f)>' % (self.x, self.y, )

a = Vector(3, 5)
b = Vector(2, 7)

print a + b # Output: <Vector (5.000000, 12.000000)>
print b - a # Output: <Vector (-1.000000, 2.000000)>
print b * 1.3 # Output: <Vector (2.600000, 9.100000)>
print a // 17 # Output: <Vector (0.000000, 0.000000)>
```

```
print a / 17 # Output: <Vector (0.176471, 0.294118)>
```

L'esempio sopra mostra l'overloading degli operatori numerici di base. Un elenco completo può essere trovato qui .

Leggi Caratteristiche nascoste online: https://riptutorial.com/it/python/topic/946/caratteristiche-nascoste

Capitolo 22: ChemPy - pacchetto python

introduzione

ChemPy è un pacchetto python progettato principalmente per risolvere e risolvere problemi di chimica fisica, analitica e inorganica. È un toolkit Python gratuito e open source per applicazioni di chimica, ingegneria chimica e scienze dei materiali.

Examples

Formule di analisi

```
from chempy import Substance
ferricyanide = Substance.from_formula('Fe(CN)6-3')
ferricyanide.composition == {0: -3, 26: 1, 6: 6, 7: 6}
True
print(ferricyanide.unicode_name)
Fe(CN)₆³⁻
 print(ferricyanide.latex_name + ", " + ferricyanide.html_name)
Fe(CN)_{6}^{3-}, Fe(CN)<sub>6</sub><sup>3-</sup>
 print('%.3f' % ferricyanide.mass)
211.955
```

Nella composizione, i numeri atomici (e 0 per la carica) sono usati come chiavi e il conteggio di ciascun tipo diventa valore rispettivo.

Bilanciatura della stechiometria di una reazione chimica

```
 from chempy import balance_stoichiometry  # Main reaction in NASA's booster rockets:
 reac, prod = balance_stoichiometry({'NH4ClO4', 'Al'}, {'Al2O3', 'HCl', 'H2O', 'N2'})
 from pprint import pprint
 pprint(reac)
{'Al': 10, 'NH4ClO4': 6}
 pprint(prod)
{'Al2O3': 5, 'H2O': 9, 'HCl': 6, 'N2': 3}
 from chempy import mass_fractions
 for fractions in map(mass_fractions, [reac, prod]):
...     pprint({k: '{0:.3g} wt%'.format(v*100) for k, v in fractions.items()})
...
{'Al': '27.7 wt%', 'NH4ClO4': '72.3 wt%'}
{'Al2O3': '52.3 wt%', 'H2O': '16.6 wt%', 'HCl': '22.4 wt%', 'N2': '8.62 wt%'}
```

Reazioni di bilanciamento

```
from chempy import Equilibrium
 from sympy import symbols
 K1, K2, Kw = symbols('K1 K2 Kw')
 e1 = Equilibrium({'MnO4-': 1, 'H+': 8, 'e-': 5}, {'Mn+2': 1, 'H2O': 4}, K1)
 e2 = Equilibrium({'O2': 1, 'H2O': 2, 'e-': 4}, {'OH-': 4}, K2)
 coeff = Equilibrium.eliminate([e1, e2], 'e-')
```

```
 coeff
[4, -5]
 redox = e1*coeff[0] + e2*coeff[1]
 print(redox)
20 OH- + 32 H+ + 4 MnO4- = 26 H2O + 4 Mn+2 + 5 O2; K1**4/K2**5
 autoprot = Equilibrium({'H2O': 1}, {'H+': 1, 'OH-': 1}, Kw)
 n = redox.cancel(autoprot)
 n
20
 redox2 = redox + n*autoprot
 print(redox2)
12 H+ + 4 MnO4- = 4 Mn+2 + 5 O2 + 6 H2O; K1**4*Kw**20/K2**5
```

Equilibri chimici

```
 from chempy import Equilibrium
 from chempy.chemistry import Species
 water_autop = Equilibrium({'H2O'}, {'H+', 'OH-'}, 10**-14)  # unit "molar" assumed
 ammonia_prot = Equilibrium({'NH4+'}, {'NH3', 'H+'}, 10**-9.24)  # same here
 from chempy.equilibria import EqSystem
 substances = map(Species.from_formula, 'H2O OH- H+ NH3 NH4+'.split())
 eqsys = EqSystem([water_autop, ammonia_prot], substances)
 print('\n'.join(map(str, eqsys.rxns)))  # "rxns" short for "reactions"
H2O = H+ + OH-; 1e-14
NH4+ = H+ + NH3; 5.75e-10
 from collections import defaultdict
 init_conc = defaultdict(float, {'H2O': 1, 'NH3': 0.1})
 x, sol, sane = eqsys.root(init_conc)
 assert sol['success'] and sane
 print(sorted(sol.keys()))  # see package "pyneqsys" for more info
['fun', 'intermediate_info', 'internal_x_vecs', 'nfev', 'njev', 'success', 'x', 'x_vecs']
 print(', '.join('%.2g' % v for v in x))
1, 0.0013, 7.6e-12, 0.099, 0.0013
```

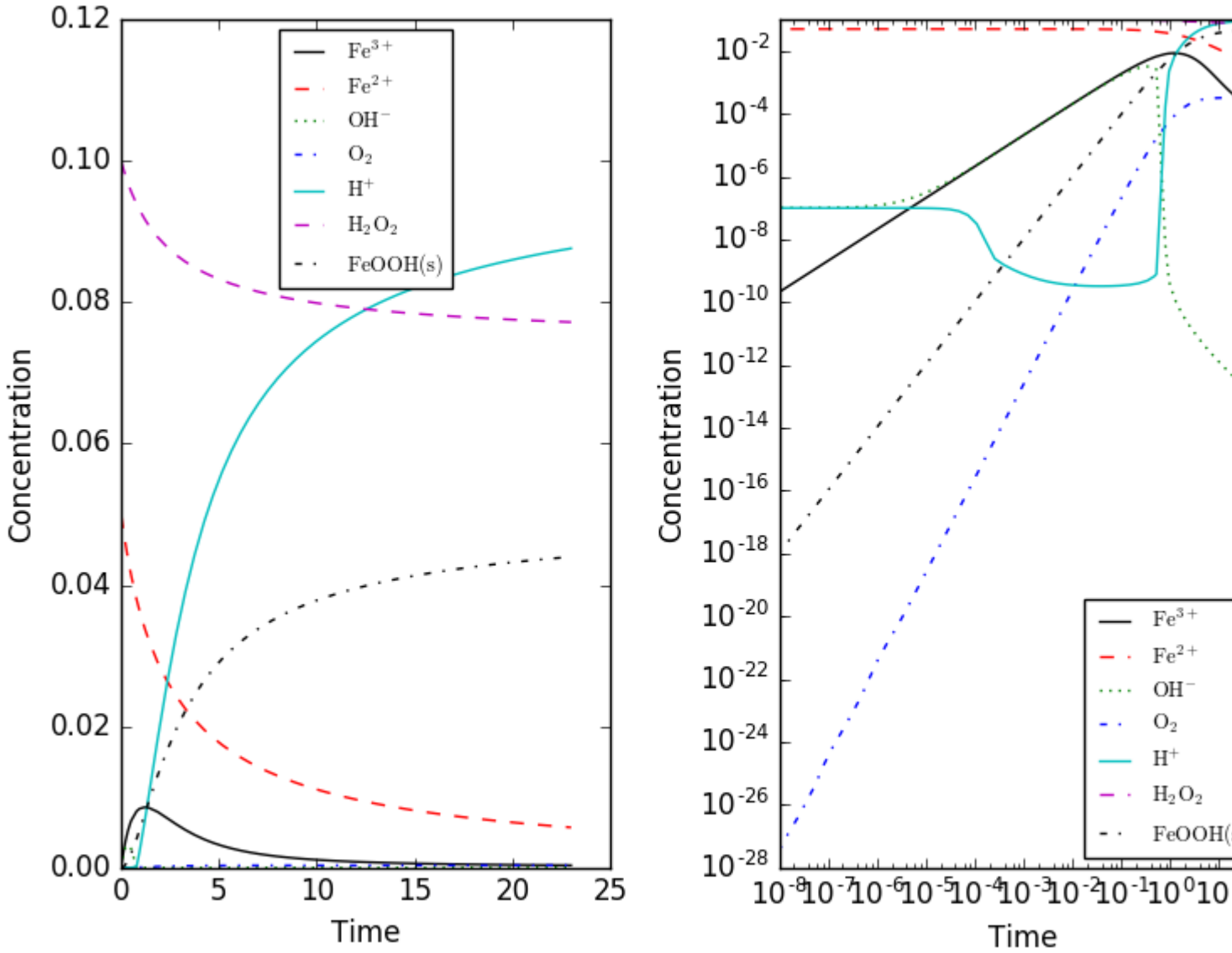
Forza ionica

```
 from chempy.electrolytes import ionic_strength
     ionic_strength({'Fe+3': 0.050, 'ClO4-': 0.150}) == .3
    True
```

Cinetica chimica (sistema di equazioni differenziali ordinarie)

```
from chempy import ReactionSystem  # The rate constants below are arbitrary
 rsys = ReactionSystem.from_string("""2 Fe+2 + H2O2 -> 2 Fe+3 + 2 OH-; 42
     2 Fe+3 + H2O2 -> 2 Fe+2 + O2 + 2 H+; 17
     H+ + OH- -> H2O; 1e10
     H2O -> H+ + OH-; 1e-4
     Fe+3 + 2 H2O -> FeOOH(s) + 3 H+; 1
     FeOOH(s) + 3 H+ -> Fe+3 + 2 H2O; 2.5""")  # "[H2O]" = 1.0 (actually 55.4 at RT)
 from chempy.kinetics.ode import get_odesys
 odesys, extra = get_odesys(rsys)
 from collections import defaultdict
 import numpy as np
 tout = sorted(np.concatenate((np.linspace(0, 23), np.logspace(-8, 1))))
 c0 = defaultdict(float, {'Fe+2': 0.05, 'H2O2': 0.1, 'H2O': 1.0, 'H+': 1e-7, 'OH-': 1e-7})
 result = odesys.integrate(tout, c0, atol=1e-12, rtol=1e-14)
```

```
 import matplotlib.pyplot as plt
 _ = plt.subplot(1, 2, 1)
 _ = result.plot(names=[k for k in rsys.substances if k != 'H2O'])
 _ = plt.legend(loc='best', prop={'size': 9}); _ = plt.xlabel('Time'); _ =
plt.ylabel('Concentration')
 _ = plt.subplot(1, 2, 2)
 _ = result.plot(names=[k for k in rsys.substances if k != 'H2O'], xscale='log', yscale='log')
 _ = plt.legend(loc='best', prop={'size': 9}); _ = plt.xlabel('Time'); _ =
plt.ylabel('Concentration')
 _ = plt.tight_layout()
 plt.show()
```



Leggi ChemPy - pacchetto python online: https://riptutorial.com/it/python/topic/10625/chempy---pacchetto-python

Capitolo 23: Chiama Python da C

introduzione

La documentazione fornisce un'implementazione esemplificativa della comunicazione tra processi tra C # e gli script Python.

Osservazioni

Si noti che nell'esempio sopra i dati sono serializzati usando la libreria **MongoDB.Bson** che può essere installata tramite il gestore di NuGet.

Altrimenti, puoi usare qualsiasi libreria di serializzazione JSON a tua scelta.

Di seguito sono riportati i passaggi di implementazione della comunicazione tra processi:

- Gli argomenti di input sono serializzati nella stringa JSON e salvati in un file di testo temporaneo:

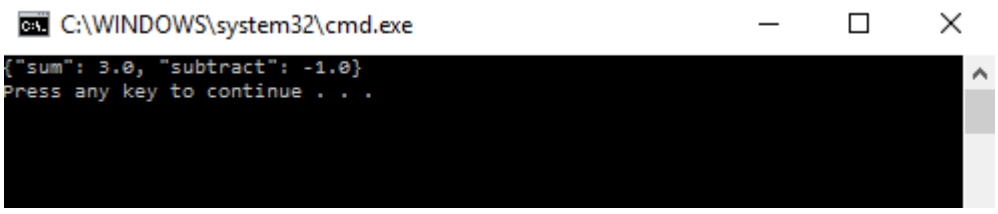
```
 BsonDocument argsBson = BsonDocument.Parse("{ 'x' : '1', 'y' : '2' }");
 string argsFile = string.Format("{0}\\{1}.txt", Path.GetDirectoryName(pyScriptPath),
Guid.NewGuid());
```

- Python interpreter python.exe esegue lo script python che legge la stringa JSON da un file di testo temporaneo e gli argomenti di input backs-out:

```
filename = sys.argv[ 1 ]
with open( filename ) as data_file:
   input_args = json.loads( data_file.read() )

x, y = [ float(input_args.get( key )) for key in [ 'x', 'y' ] ]
```

- Lo script Python viene eseguito e il dizionario di output viene serializzato nella stringa JSON e stampato nella finestra di comando:

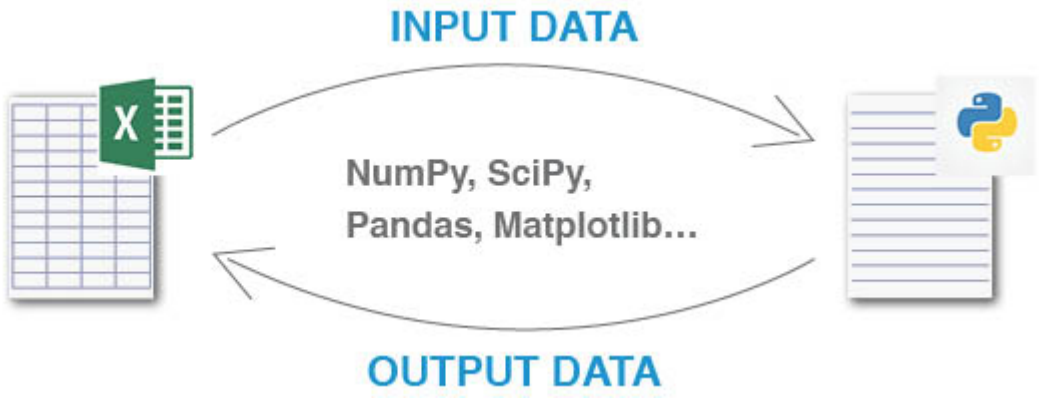
```
print json.dumps( { 'sum' : x + y , 'subtract' : x - y } )
```



- Leggi la stringa JSON di output dall'applicazione C #:

```
using (StreamReader myStreamReader = process.StandardOutput)
```

```
{
   outputString = myStreamReader.ReadLine();
   process.WaitForExit();
}
```



Sto utilizzando la comunicazione tra processi C # e gli script Python in uno dei miei progetti che consente di chiamare gli script Python direttamente dai fogli di calcolo di Excel.

Il progetto utilizza il componente aggiuntivo ExcelDNA per C #: associazione Excel.

Il codice sorgente è memorizzato nel repository GitHub.

Di seguito sono riportati i collegamenti alle pagine wiki che forniscono una panoramica del progetto e aiutano a iniziare in 4 semplici passaggi .

- Iniziare
- Panoramica dell'implementazione
- Esempi
- Object-Wizard
- funzioni

Spero che tu possa trovare utile l'esempio e il progetto.

Examples

Script Python da chiamare con l'applicazione C

```
import sys
import json

# load input arguments from the text file
filename = sys.argv[ 1 ]
with open( filename ) as data_file:
    input_args = json.loads( data_file.read() )

# cast strings to floats
x, y = [ float(input_args.get( key )) for key in [ 'x', 'y' ] ]
```

```
print json.dumps( { 'sum' : x + y , 'subtract' : x - y } )
```

Codice C # che chiama lo script Python

```
using MongoDB.Bson;
using System;
using System.Diagnostics;
using System.IO;

namespace python_csharp
{
    class Program
    {
        static void Main(string[] args)
        {
            // full path to .py file
            string pyScriptPath = "...../sum.py";
            // convert input arguments to JSON string
            BsonDocument argsBson = BsonDocument.Parse("{ 'x' : '1', 'y' : '2' }");

            bool saveInputFile = false;

            string argsFile = string.Format("{0}\\{1}.txt",
Path.GetDirectoryName(pyScriptPath), Guid.NewGuid());

            string outputString = null;
            // create new process start info
            ProcessStartInfo prcStartInfo = new ProcessStartInfo
            {
                // full path of the Python interpreter 'python.exe'
                FileName = "python.exe", // string.Format(@"""{0}""", "python.exe"),
                UseShellExecute = false,
                RedirectStandardOutput = true,
                CreateNoWindow = false
            };

            try
            {
                // write input arguments to .txt file
                using (StreamWriter sw = new StreamWriter(argsFile))
                {
                    sw.WriteLine(argsBson);
                    prcStartInfo.Arguments = string.Format("{0} {1}",
string.Format(@"""{0}""", pyScriptPath), string.Format(@"""{0}""", argsFile));
                }
                // start process
                using (Process process = Process.Start(prcStartInfo))
                {
                    // read standard output JSON string
                    using (StreamReader myStreamReader = process.StandardOutput)
                    {
                        outputString = myStreamReader.ReadLine();
                        process.WaitForExit();
                    }
                }
            }
            finally
            {
                // delete/save temporary .txt file
```

```
                if (!saveInputFile)
                {
                    File.Delete(argsFile);
                }
            }
            Console.WriteLine(outputString);
        }
    }
}
```

Leggi Chiama Python da C # online: https://riptutorial.com/it/python/topic/10759/chiama-python-da-c-sharp

Capitolo 24: Classi

introduzione

Python si offre non solo come un popolare linguaggio di scripting, ma supporta anche il paradigma di programmazione orientato agli oggetti. Le classi descrivono i dati e forniscono metodi per manipolare tali dati, tutti racchiusi in un singolo oggetto. Inoltre, le classi consentono l'astrazione separando i dettagli di implementazione concreti dalle rappresentazioni astratte dei dati.

Il codice che utilizza le classi è generalmente più facile da leggere, capire e mantenere.

Examples

Ereditarietà di base

L'ereditarietà di Python si basa su idee simili utilizzate in altri linguaggi object oriented come Java, C ++, ecc. Una nuova classe può essere derivata da una classe esistente come segue.

```
class BaseClass(object):
    pass

class DerivedClass(BaseClass):
    pass
```

Il `BaseClass` è il già esistente *(genitore)* di classe, e la `DerivedClass` è il nuovo *(bambino)* classe che eredita (o *sottoclassi)* attributi da `BaseClass` . **Nota** : A partire da Python 2.2, tutte le classi ereditano implicitamente dalla classe `object` , che è la classe base per tutti i tipi built-in.

Definiamo una classe `Rectangle` genitore nell'esempio seguente, che eredita implicitamente `object` :

```
class Rectangle():
    def __init__(self, w, h):
        self.w = w
        self.h = h

    def area(self):
        return self.w * self.h

    def perimeter(self):
        return 2 * (self.w + self.h)
```

La classe `Rectangle` può essere utilizzata come classe base per definire una classe `Square` , poiché un quadrato è un caso speciale di rettangolo.

```
class Square(Rectangle):
    def __init__(self, s):
        # call parent constructor, w and h are both s
```

```
        super(Square, self).__init__(s, s)
        self.s = s
```

La classe `Square` erediterà automaticamente tutti gli attributi della classe `Rectangle` e della classe dell'oggetto. `super()` è usato per chiamare il metodo `__init__()` della classe `Rectangle` , essenzialmente chiamando qualsiasi metodo sovrascritto della classe base. **Nota** : in Python 3, `super()` non richiede argomenti.

Gli oggetti classe derivati possono accedere e modificare gli attributi delle sue classi base:

```
r.area()
# Output: 12
r.perimeter()
# Output: 14

s.area()
# Output: 4
s.perimeter()
# Output: 8
```

Funzioni integrate che funzionano con l'ereditarietà

`issubclass(DerivedClass, BaseClass)` : restituisce `True` se `DerivedClass` è una sottoclasse di `BaseClass`

`isinstance(s, Class)` : restituisce `True` se s è un'istanza di `Class` o una delle classi derivate di `Class`

```
# subclass check
issubclass(Square, Rectangle)
# Output: True

# instantiate
r = Rectangle(3, 4)
s = Square(2)

isinstance(r, Rectangle)
# Output: True
isinstance(r, Square)
# Output: False
# A rectangle is not a square

isinstance(s, Rectangle)
# Output: True
# A square is a rectangle
isinstance(s, Square)
# Output: True
```

Classi e variabili di istanza

Le variabili di istanza sono univoche per ogni istanza, mentre le variabili di classe sono condivise

da tutte le istanze.

```
class C:
    x = 2  # class variable

    def __init__(self, y):
        self.y = y  # instance variable

C.x
# 2
C.y
# AttributeError: type object 'C' has no attribute 'y'

c1 = C(3)
c1.x
# 2
c1.y
# 3

c2 = C(4)
c2.x
# 2
c2.y
# 4
```

È possibile accedere alle variabili di classe sulle istanze di questa classe, ma assegnando all'attributo class verrà creata una variabile di istanza che ombreggia la variabile di classe

```
c2.x = 4
c2.x
# 4
C.x
# 2
```

Si noti che *mutando* variabili di classe dalle istanze può portare ad alcune conseguenze inaspettate.

```
class D:
    x = []
    def __init__(self, item):
        self.x.append(item)  # note that this is not an assigment!

d1 = D(1)
d2 = D(2)

d1.x
# [1, 2]
d2.x
# [1, 2]
D.x
# [1, 2]
```

Metodi legati, non associati e statici

L'idea dei metodi bound e nonbound è stata rimossa in Python 3 . In Python 3 quando dichiari un metodo all'interno di una classe, stai usando una parola chiave `def` , creando così un oggetto

funzione. Questa è una funzione regolare e la classe circostante funziona come il suo spazio dei nomi. Nell'esempio seguente dichiariamo il metodo `f` all'interno della classe `A` , e diventa una funzione `Af` :

Python 3.x 3.0

```
class A(object):
    def f(self, x):
        return 2 * x
A.f
# <function A.f at ...>  (in Python 3.x)
```

In Python 2 il comportamento era diverso: gli oggetti funzione all'interno della classe venivano implicitamente sostituiti con oggetti di tipo `instancemethod` , che venivano chiamati *metodi* non associati perché non erano associati a nessuna particolare istanza di classe. Era possibile accedere alla funzione sottostante usando la proprietà `.__func__` .

Python 2.x 2.3

```
A.f
# <unbound method A.f>   (in Python 2.x)
A.f.__class__
# <type 'instancemethod'>
A.f.__func__
# <function f at ...>
```

Questi ultimi comportamenti sono confermati dall'ispezione: i metodi sono riconosciuti come funzioni in Python 3, mentre la distinzione è confermata in Python 2.

Python 3.x 3.0

```
import inspect

inspect.isfunction(A.f)
# True
inspect.ismethod(A.f)
# False
```

Python 2.x 2.3

```
import inspect

inspect.isfunction(A.f)
# False
inspect.ismethod(A.f)
# True
```

In entrambe le versioni della funzione / metodo Python, `Af` può essere chiamato direttamente, a condizione di passare un'istanza di classe `A` come primo argomento.

```
A.f(1, 7)
# Python 2: TypeError: unbound method f() must be called with
```

```
#                       A instance as first argument (got int instance instead)
# Python 3: 14
a = A()
A.f(a, 20)
# Python 2 & 3: 40
```

Ora supponiamo `a` è un'istanza della classe `A` , che cosa è `af` allora? Beh, intuitivamente questo dovrebbe essere lo stesso metodo `f` di classe `A` , solo che dovrebbe in qualche modo "sapere" che è stato applicato all'oggetto `a` - in Python questo si chiama il metodo *legato* a `a` .

I dettagli essenziali sono i seguenti: la scrittura `af` invoca la magia `__getattribute__` metodo `a` , che controlla innanzitutto se `a` ha un attributo denominato `f` (che non), quindi controlla la classe `A` se contiene un metodo con un tale nome (lo fa), e crea un nuovo oggetto `m` di tipo `method` che ha il riferimento al `Af` originale in `m.__func__` , e un riferimento all'oggetto `a` in `m.__self__` . Quando questo oggetto viene chiamato come funzione, fa semplicemente quanto segue: `m(...) => m.__func__(m.__self__, ...)` . Quindi questo oggetto è chiamato **metodo vincolato** perché quando viene invocato esso sa di fornire l'oggetto a cui era associato come primo argomento. (Queste cose funzionano allo stesso modo in Python 2 e 3).

```
a = A()
a.f
# <bound method A.f of <__main__.A object at ...>>
a.f(2)
# 4

# Note: the bound method object a.f is recreated *every time* you call it:
a.f is a.f  # False
# As a performance optimization you can store the bound method in the object's
# __dict__, in which case the method object will remain fixed:
a.f = a.f
a.f is a.f  # True
```

Infine, Python ha **metodi di classe** e **metodi statici** - tipi speciali di metodi. I metodi di classe funzionano allo stesso modo dei metodi regolari, tranne che quando invocati su un oggetto si collegano alla *classe* dell'oggetto anziché all'oggetto. Quindi `m.__self__ = type(a)` . Quando chiamate tale metodo associato, passa la classe di `a` come primo argomento. I metodi statici sono ancora più semplici: non legano nulla e restituiscono semplicemente la funzione sottostante senza alcuna trasformazione.

```
class D(object):
    multiplier = 2

    @classmethod
    def f(cls, x):
        return cls.multiplier * x

    @staticmethod
    def g(name):
        print("Hello, %s" % name)

D.f
# <bound method type.f of <class '__main__.D'>>
D.f(12)
# 24
```

```
D.g
# <function D.g at ...>
D.g("world")
# Hello, world
```

Si noti che i metodi di classe sono associati alla classe anche quando si accede all'istanza:

```
d = D()
d.multiplier = 1337
(D.multiplier, d.multiplier)
# (2, 1337)
d.f
# <bound method D.f of <class '__main__.D'>>
d.f(10)
# 20
```

Vale la pena notare che al livello più basso, le funzioni, i metodi, i metodi statici, ecc. Sono in realtà dei descrittori che invocano `__get__` , `__set__` e facoltativamente `__del__` metodi speciali. Per maggiori dettagli su classmethods e staticmethods:

- Qual è la differenza tra @staticmethod e @classmethod in Python?
- Significato di @classmethod e @staticmethod per principianti?

Classi vecchio stile e vecchio stile

Python 2.x 2.2.0

Le classi di *nuovo stile* sono state introdotte in Python 2.2 per unificare *classi* e *tipi* . Essi ereditano dal tipo di `object` livello superiore. *Una classe di nuovo stile è un tipo definito dall'utente* ed è molto simile ai tipi built-in.

```
# new-style class
class New(object):
    pass

# new-style instance
new = New()

new.__class__
# <class '__main__.New'>
type(new)
# <class '__main__.New'>
issubclass(New, object)
# True
```

Le classi *vecchio stile* **non** ereditano `object` . Le istanze di vecchio stile vengono sempre implementate con un tipo di `instance` incorporato.

```
# old-style class
class Old:
    pass

# old-style instance
```

```
old = Old()

old.__class__
# <class __main__.Old at ...>
type(old)
# <type 'instance'>
issubclass(Old, object)
# False
```

Python 3.x 3.0.0

In Python 3, le classi vecchio stile sono state rimosse.

Le classi di nuovo stile in Python 3 ereditano implicitamente `object` , quindi non è più necessario specificare `MyClass(object)` .

```
class MyClass:
    pass

my_inst = MyClass()

type(my_inst)
# <class '__main__.MyClass'>
my_inst.__class__
# <class '__main__.MyClass'>
issubclass(MyClass, object)
# True
```

Valori predefiniti per variabili di istanza

Se la variabile contiene un valore di un tipo immutabile (ad es. Una stringa), allora va bene assegnare un valore predefinito come questo

```
class Rectangle(object):
    def __init__(self, width, height, color='blue'):
        self.width = width
        self.height = height
        self.color = color

    def area(self):
        return self.width  * self.height

# Create some instances of the class
default_rectangle = Rectangle(2, 3)
print(default_rectangle.color) # blue

red_rectangle = Rectangle(2, 3, 'red')
print(red_rectangle.color) # red
```

È necessario prestare attenzione durante l'inizializzazione di oggetti mutabili come gli elenchi nel costruttore. Considera il seguente esempio:

```
class Rectangle2D(object):
    def __init__(self, width, height, pos=[0,0], color='blue'):
        self.width = width
```

```
        self.height = height
        self.pos = pos
        self.color = color

r1 = Rectangle2D(5,3)
r2 = Rectangle2D(7,8)
r1.pos[0] = 4
r1.pos # [4, 0]
r2.pos # [4, 0] r2's pos has changed as well
```

Questo comportamento è causato dal fatto che in Python i parametri predefiniti sono vincolati all'esecuzione della funzione e non alla dichiarazione di funzione. Per ottenere una variabile di istanza predefinita che non è condivisa tra le istanze, si dovrebbe usare un costrutto come questo:

```
class Rectangle2D(object):
    def __init__(self, width, height, pos=None, color='blue'):
        self.width = width
        self.height = height
        self.pos = pos or [0, 0] # default value is [0, 0]
        self.color = color

r1 = Rectangle2D(5,3)
r2 = Rectangle2D(7,8)
r1.pos[0] = 4
r1.pos # [4, 0]
r2.pos # [0, 0] r2's pos hasn't changed
```

Vedi anche Argomenti predefiniti mutabili e "Almost Astonishment" e l'argomento Mutable Default .

Eredità multipla

Python utilizza l'algoritmo di linearizzazione C3 per determinare l'ordine in cui risolvere gli attributi di classe, inclusi i metodi. Questo è noto come il metodo Resolution Order (MRO).

Ecco un semplice esempio:

```
class Foo(object):
    foo = 'attr foo of Foo'


class Bar(object):
    foo = 'attr foo of Bar' # we won't see this.
    bar = 'attr bar of Bar'

class FooBar(Foo, Bar):
    foobar = 'attr foobar of FooBar'
```

Ora, se istanziamo FooBar, se guardiamo l'attributo foo, vediamo che l'attributo di Foo viene trovato per primo

```
fb = FooBar()
```

e

```
>>> fb.foo
'attr foo of Foo'
```

Ecco il MRO di FooBar:

```
>>> FooBar.mro()
[<class '__main__.FooBar'>, <class '__main__.Foo'>, <class '__main__.Bar'>, <type 'object'>]
```

Si può semplicemente affermare che l'algoritmo MRO di Python è

1. Profondità prima (es. `FooBar` quindi `Foo`) a meno che
2. un genitore condiviso (`object`) è bloccato da un bambino (`Bar`) e
3. non sono consentite relazioni circolari.

Ad esempio, Bar non può ereditare da FooBar mentre FooBar eredita da Bar.

Per un esempio completo in Python, vedere la voce wikipedia .

Un'altra potente caratteristica nell'ereditarietà è `super` . super può recuperare le caratteristiche delle classi genitore.

```
class Foo(object):
    def foo_method(self):
        print "foo Method"

class Bar(object):
    def bar_method(self):
        print "bar Method"

class FooBar(Foo, Bar):
    def foo_method(self):
        super(FooBar, self).foo_method()
```

Eredità multipla con metodo init di classe, quando ogni classe ha il proprio metodo init, quindi proviamo per l'ineritanza multipla, quindi solo il metodo init viene chiamato di classe che è ereditato per primo.

per il seguente esempio, solo il metodo **init della** classe Foo viene chiamato come chiamata della classe **Bar** init

```
    class Foo(object):
        def __init__(self):
            print "foo init"

    class Bar(object):
        def __init__(self):
            print "bar init"

    class FooBar(Foo, Bar):
        def __init__(self):
            print "foobar init"
            super(FooBar, self).__init__()

    a = FooBar()
```

Produzione:

```
    foobar init
    foo init
```

Ma ciò non significa che la classe **Bar** non sia ereditaria. L'istanza della classe **FooBar** finale è anche un'istanza della classe **Bar** e della classe **Foo** .

```
print isinstance(a,FooBar)
print isinstance(a,Foo)
print isinstance(a,Bar)
```

Produzione:

```
True
True
True
```

Descrittori e ricerche punteggiate

I descrittori sono oggetti che sono (solitamente) attributi di classi e che hanno uno qualsiasi dei metodi speciali `__get__` , `__set__` o `__delete__` .

I descrittori di dati hanno uno qualsiasi di `__set__` o `__delete__`

Questi possono controllare la ricerca `staticmethod` su un'istanza e sono utilizzati per implementare funzioni, `staticmethod` , `classmethod` e `property` . Una ricerca tratteggiata (ad esempio, l'istanza `foo` della classe `Foo` osserva la `bar` attributi `bar` ovvero `foo.bar`), utilizza il seguente algoritmo:

1. `bar` è guardato in classe, `Foo` . Se è presente ed è un **descrittore di dati** , viene utilizzato il descrittore di dati. Ecco come la `property` è in grado di controllare l'accesso ai dati in un'istanza e le istanze non possono sovrascriverli. Se un **descrittore di dati** non è lì, allora

2. `bar` viene cercata nell'istanza `__dict__` . Questo è il motivo per cui possiamo eseguire l'override o bloccare i metodi chiamati da un'istanza con una ricerca punteggiata. Se la `bar` esiste nell'istanza, viene utilizzata. Altrimenti, noi allora

3. guarda nella classe `Foo` per `bar` . Se si tratta di un **descrittore** , viene utilizzato il protocollo descrittore. In questo modo vengono implementate le funzioni (in questo contesto, metodi non `classmethod`), `classmethod` e `staticmethod` . Altrimenti restituisce semplicemente l'oggetto lì, oppure esiste un `AttributeError`

Metodi di classe: inizializzatori alternativi

I metodi di classe presentano metodi alternativi per creare istanze di classi. Per illustrare, diamo un'occhiata a un esempio.

Supponiamo di avere una classe `Person` relativamente semplice:

```
class Person(object):

    def __init__(self, first_name, last_name, age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age
        self.full_name = first_name + " " + last_name

    def greet(self):
        print("Hello, my name is " + self.full_name + ".")
```

Potrebbe essere utile avere un modo per creare istanze di questa classe specificando un nome completo invece di nome e cognome separatamente. Un modo per farlo sarebbe quello di avere `last_name` essere un parametro facoltativo, e partendo dal presupposto che se non viene fornito, abbiamo passato il nome completo in:

```
class Person(object):

    def __init__(self, first_name, age, last_name=None):
        if last_name is None:
            self.first_name, self.last_name = first_name.split(" ", 2)
        else:
            self.first_name = first_name
            self.last_name = last_name

        self.full_name = self.first_name + " " + self.last_name
        self.age = age

    def greet(self):
        print("Hello, my name is " + self.full_name + ".")
```

Tuttavia, ci sono due problemi principali con questo bit di codice:

1. I parametri `first_name` e `last_name` ora sono fuorvianti, poiché è possibile immettere un nome completo per `first_name` . Inoltre, se ci sono più casi e / o più parametri che hanno questo tipo di flessibilità, la ramificazione if / elif / else può diventare fastidiosa velocemente.

2. Non è altrettanto importante, ma vale comunque la pena segnalarlo: cosa succede se `last_name` è `None` , ma `first_name` non si divide in due o più cose tramite spazi? Abbiamo ancora un altro livello di convalida dell'input e / o gestione delle eccezioni ...

Inserisci i metodi di classe. Invece di avere un singolo inizializzatore, creeremo un inizializzatore separato, chiamato `from_full_name` , e lo `classmethod` decoratore `classmethod` (incorporato).

```
class Person(object):

    def __init__(self, first_name, last_name, age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age
        self.full_name = first_name + " " + last_name

    @classmethod
    def from_full_name(cls, name, age):
        if " " not in name:
```

```
            raise ValueError
        first_name, last_name = name.split(" ", 2)
        return cls(first_name, last_name, age)

    def greet(self):
        print("Hello, my name is " + self.full_name + ".")
```

Notate `cls` anziché `self` come primo argomento a `from_full_name` . I metodi di classe sono applicati alla classe generale, *non* un'istanza di una data classe (che è ciò che di solito indica il `self`). Quindi, se `cls` è la nostra classe `Person` , il valore restituito dal metodo di classe `from_full_name` è `Person(first_name, last_name, age)` , che utilizza `__init__` di `Person` per creare un'istanza della classe `Person` . In particolare, se dovessimo creare una sottoclasse `Employee` of `Person` , allora `from_full_name` funzionerebbe anche nella classe `Employee` .

Per dimostrare che funziona come previsto, creiamo istanze di `Person` in più di un modo senza la ramificazione in `__init__` :

```
In [2]: bob = Person("Bob", "Bobberson", 42)

In [3]: alice = Person.from_full_name("Alice Henderson", 31)

In [4]: bob.greet()
Hello, my name is Bob Bobberson.

In [5]: alice.greet()
Hello, my name is Alice Henderson.
```

Altre referenze:

- Python @classmethod e @staticmethod per principianti?
- https://docs.python.org/2/library/functions.html#classmethod
- https://docs.python.org/3.5/library/functions.html#classmethod

Composizione di classe

La composizione di classe consente relazioni esplicite tra oggetti. In questo esempio, le persone vivono in città che appartengono ai paesi. La composizione consente alle persone di accedere al numero di tutte le persone che vivono nel loro paese:

```
class Country(object):
    def __init__(self):
        self.cities=[]

    def addCity(self,city):
        self.cities.append(city)


class City(object):
    def __init__(self, numPeople):
        self.people = []
        self.numPeople = numPeople
```

```
    def addPerson(self, person):
        self.people.append(person)

    def join_country(self,country):
        self.country = country
        country.addCity(self)

        for i in range(self.numPeople):
                person(i).join_city(self)


class Person(object):
    def __init__(self, ID):
        self.ID=ID

    def join_city(self, city):
        self.city = city
        city.addPerson(self)

    def people_in_my_country(self):
        x= sum([len(c.people) for c in self.city.country.cities])
        return x

US=Country()
NYC=City(10).join_country(US)
SF=City(5).join_country(US)

print(US.cities[0].people[0].people_in_my_country())

# 15
```

Patch per scimmia

In questo caso, "patch di scimmia" significa aggiungere una nuova variabile o un metodo a una classe dopo che è stata definita. Ad esempio, diciamo che abbiamo definito la classe `A` come

```
class A(object):
    def __init__(self, num):
        self.num = num

    def __add__(self, other):
        return A(self.num + other.num)
```

Ma ora vogliamo aggiungere un'altra funzione più avanti nel codice. Supponiamo che questa funzione sia la seguente.

```
def get_num(self):
    return self.num
```

Ma come aggiungiamo questo come metodo in `A` ? È semplice, in pratica, inseriamo semplicemente tale funzione in `A` con un'istruzione di assegnazione.

```
A.get_num = get_num
```

Perché funziona? Poiché le funzioni sono oggetti proprio come qualsiasi altro oggetto, e i metodi sono funzioni che appartengono alla classe.

La funzione `get_num` deve essere disponibile per tutte le esistenti (già create) anche per le nuove istanze di `A`

Queste aggiunte sono disponibili su tutte le istanze di quella classe (o delle sue sottoclassi) automaticamente. Per esempio:

```
foo = A(42)

A.get_num = get_num

bar = A(6);

foo.get_num() # 42

bar.get_num() # 6
```

Nota che, a differenza di altri linguaggi, questa tecnica non funziona per certi tipi built-in e non è considerata di buon stile.

Elenco di tutti i membri della classe

La funzione `dir()` può essere utilizzata per ottenere un elenco dei membri di una classe:

```
dir(Class)
```

Per esempio:

```
>>> dir(list)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__',
'__iadd__', '__imul__', '__init__', '__iter__', '__le__', '__len__', '__lt__', '__mul__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__',
'__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'append', 'clear',
'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

È comune cercare solo membri "non magici". Questo può essere fatto usando una semplice comprensione che elenca membri con nomi che non iniziano con __ :

```
>>> [m for m in dir(list) if not m.startswith('__')]
['append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse',
'sort']
```

Avvertenze:

Le classi possono definire un `__dir__()` . Se questo metodo esiste chiamando `dir()` chiamerà `__dir__()` , altrimenti Python proverà a creare un elenco di membri della classe. Ciò significa che la funzione dir può avere risultati imprevisti. Due citazioni di importanza dalla documentazione ufficiale di Python :

Se l'oggetto non fornisce **dir** (), la funzione fa del suo meglio per raccogliere informazioni dall'attributo **dict** dell'oggetto, se definito, e dal suo oggetto type. L'elenco risultante non è necessariamente completo e potrebbe essere inaccurato quando l'oggetto ha un **getattr** personalizzato ().

Nota: poiché dir () viene fornito principalmente come utilità per l'uso a un prompt interattivo, tenta di fornire un set di nomi interessante più di quello che prova a fornire un insieme di nomi rigorosamente o coerentemente definito e il suo comportamento dettagliato può cambiare stampa. Ad esempio, gli attributi metaclass non sono nella lista dei risultati quando l'argomento è una classe.

Introduzione alle classi

Una classe, funziona come un modello che definisce le caratteristiche di base di un particolare oggetto. Ecco un esempio:

```
class Person(object):
     """A simple class."""                          # docstring
     species = "Homo Sapiens"                       # class attribute

     def __init__(self, name):                      # special method
         """This is the initializer. It's a special
         method (see below).
         """
         self.name = name                           # instance attribute

     def __str__(self):                             # special method
         """This method is run when Python tries
         to cast the object to a string. Return
         this string when using print(), etc.
         """
         return self.name

     def rename(self, renamed):                     # regular method
         """Reassign and print the name attribute."""
         self.name = renamed
         print("Now my name is {}".format(self.name))
```

Ci sono alcune cose da notare quando si guarda all'esempio sopra.

1. La classe è composta da *attributi* (dati) e *metodi* (funzioni).
2. Attributi e metodi sono semplicemente definiti come normali variabili e funzioni.
3. Come indicato nella docstring corrispondente, il metodo `__init__()` è chiamato *inizializzatore* . È equivalente al costruttore in altri linguaggi orientati agli oggetti ed è il metodo che viene eseguito per la prima volta quando si crea un nuovo oggetto o una nuova istanza della classe.
4. Gli attributi che si applicano all'intera classe sono definiti per primi e sono chiamati *attributi di classe* .
5. Gli attributi che si applicano a un'istanza specifica di una classe (un oggetto) sono chiamati *attributi di istanza* . Sono generalmente definiti all'interno di `__init__()` ; questo non è necessario, ma è raccomandato (poiché gli attributi definiti al di fuori di `__init__()` corrono il rischio di essere consultati prima che vengano definiti).

6. Ogni metodo, incluso nella definizione della classe, passa l'oggetto in questione come primo parametro. La parola `self` viene usata per questo parametro (l'uso di `self` è in realtà per convenzione, poiché la parola `self` non ha alcun significato intrinseco in Python, ma questa è una delle convenzioni più rispettate di Python e dovresti sempre seguirla).
7. Quelli che sono abituati alla programmazione orientata agli oggetti in altre lingue possono essere sorpresi da alcune cose. Uno è che Python non ha alcun concetto reale di elementi `private` , quindi tutto, per impostazione predefinita, imita il comportamento della parola chiave `public` C ++ / Java. Per ulteriori informazioni, consultare l'esempio "Membri della classe privata" in questa pagina.
8. Alcuni dei metodi della classe hanno la seguente forma: `__functionname__(self, other_stuff)` . Tutti questi metodi sono chiamati "metodi magici" e sono una parte importante delle classi in Python. Ad esempio, l'overloading dell'operatore in Python è implementato con metodi magici. Per ulteriori informazioni, consultare la documentazione pertinente .

Ora facciamo alcuni esempi della nostra classe `Person` !

```
>>> # Instances
>>> kelly = Person("Kelly")
>>> joseph = Person("Joseph")
>>> john_doe = Person("John Doe")
```

Al momento abbiamo tre oggetti `Person` , `kelly` , `joseph` e `john_doe` .

Possiamo accedere agli attributi della classe da ogni istanza usando l'operatore punto `.` Notare nuovamente la differenza tra attributi di classe e istanza:

```
>>> # Attributes
>>> kelly.species
'Homo Sapiens'
>>> john_doe.species
'Homo Sapiens'
>>> joseph.species
'Homo Sapiens'
>>> kelly.name
'Kelly'
>>> joseph.name
'Joseph'
```

Possiamo eseguire i metodi della classe usando lo stesso operatore punto `.` :

```
>>> # Methods
>>> john_doe.__str__()
'John Doe'
>>>  print(john_doe)
'John Doe'
>>>  john_doe.rename("John")
'Now my name is John'
```

Proprietà

Le classi Python supportano le **proprietà** , che assomigliano a variabili di oggetti regolari, ma con

la possibilità di allegare comportamenti e documentazione personalizzati.

```
class MyClass(object):

    def __init__(self):
       self._my_string = ""

    @property
    def string(self):
        """A profoundly important string."""
        return self._my_string

    @string.setter
    def string(self, new_value):
        assert isinstance(new_value, str), \
               "Give me a string, not a %r!" % type(new_value)
        self._my_string = new_value

    @string.deleter
    def x(self):
        self._my_string = None
```

Sembra che l'oggetto della classe `MyClass` abbia una proprietà `.string` , tuttavia il suo comportamento è ora strettamente controllato:

```
mc = MyClass()
mc.string = "String!"
print(mc.string)
del mc.string
```

Oltre alla sintassi utile come sopra, la sintassi della proprietà consente la convalida o altri incrementi da aggiungere a tali attributi. Questo potrebbe essere particolarmente utile con le API pubbliche - dove un livello di aiuto dovrebbe essere dato all'utente.

Un altro uso comune delle proprietà è di consentire alla classe di presentare "attributi virtuali" - attributi che non sono effettivamente memorizzati ma che vengono calcolati solo quando richiesto.

```
class Character(object):
    def __init__(name, max_hp):
        self._name = name
        self._hp = max_hp
        self._max_hp = max_hp

    # Make hp read only by not providing a set method
    @property
    def hp(self):
        return self._hp

    # Make name read only by not providing a set method
    @property
    def name(self):
        return self.name

    def take_damage(self, damage):
        self.hp -= damage
        self.hp = 0 if self.hp <0 else self.hp
```

```
    @property
    def is_alive(self):
        return self.hp != 0

    @property
    def is_wounded(self):
        return self.hp < self.max_hp if self.hp > 0 else False

    @property
    def is_dead(self):
        return not self.is_alive

bilbo = Character('Bilbo Baggins', 100)
bilbo.hp
# out : 100
bilbo.hp = 200
# out : AttributeError: can't set attribute
# hp attribute is read only.

bilbo.is_alive
# out : True
bilbo.is_wounded
# out : False
bilbo.is_dead
# out : False

bilbo.take_damage( 50 )

bilbo.hp
# out : 50

bilbo.is_alive
# out : True
bilbo.is_wounded
# out : True
bilbo.is_dead
# out : False

bilbo.take_damage( 50 )
bilbo.hp
# out : 0

bilbo.is_alive
# out : False
bilbo.is_wounded
# out : False
bilbo.is_dead
# out : True
```

Classe Singleton

Un singleton è un pattern che limita l'istanza di una classe a un'istanza / oggetto. Per maggiori informazioni sui modelli di design singleton Python, vedere qui .

```
class Singleton:
    def __new__(cls):
        try:
            it = cls.__it__
```

```
        except AttributeError:
            it = cls.__it__ = object.__new__(cls)
        return it

    def __repr__(self):
        return '<{}>'.format(self.__class__.__name__.upper())

    def __eq__(self, other):
         return other is self
```

Un altro metodo è quello di decorare la tua classe. Seguendo l'esempio di questa risposta, crea una classe Singleton:

```
class Singleton:
    """
    A non-thread-safe helper class to ease implementing singletons.
    This should be used as a decorator -- not a metaclass -- to the
    class that should be a singleton.

    The decorated class can define one `__init__` function that
    takes only the `self` argument. Other than that, there are
    no restrictions that apply to the decorated class.

    To get the singleton instance, use the `Instance` method. Trying
    to use `__call__` will result in a `TypeError` being raised.

    Limitations: The decorated class cannot be inherited from.

    """

    def __init__(self, decorated):
        self._decorated = decorated

    def Instance(self):
        """
        Returns the singleton instance. Upon its first call, it creates a
        new instance of the decorated class and calls its `__init__` method.
        On all subsequent calls, the already created instance is returned.

        """
        try:
            return self._instance
        except AttributeError:
            self._instance = self._decorated()
            return self._instance

    def __call__(self):
        raise TypeError('Singletons must be accessed through `Instance()`.')

    def __instancecheck__(self, inst):
        return isinstance(inst, self._decorated)
```

Per usare puoi usare il metodo `Instance`

```
@Singleton
class Single:
    def __init__(self):
        self.name=None
```

```
        self.val=0
    def getName(self):
        print(self.name)

x=Single.Instance()
y=Single.Instance()
x.name='I\'m single'
x.getName() # outputs I'm single
y.getName() # outputs I'm single
```

Leggi Classi online: https://riptutorial.com/it/python/topic/419/classi

Capitolo 25: Commenti e documentazione

Sintassi

- # Questo è un commento a riga singola
- print ("") # Questo è un commento in linea
- """
  Questo è
  un commento su più righe
  """

Osservazioni

Gli sviluppatori dovrebbero seguire le linee guida PEP257 - Convenzioni di docstring . In alcuni casi, le guide di stile (come quelle di Google Style Guide) o il rendering di documentazione di terze parti (come la Sfinge) possono dettagliare ulteriori convenzioni per le docstring.

Examples

Commenti a riga singola, in linea e multilinea

I commenti sono usati per spiegare il codice quando il codice di base non è chiaro.

Python ignora i commenti, quindi non eseguirà il codice, o creerà errori di sintassi per semplici frasi inglesi.

I commenti a riga singola iniziano con il carattere hash (`#`) e terminano alla fine della riga.

- Commento a riga singola:

```
# This is a single line comment in Python
```

- Commento in linea:

```
print("Hello World")  # This line prints "Hello World"
```

- I commenti che si estendono su più righe hanno `"""` o `'''` su entrambe le estremità. È lo stesso di una stringa multilinea, ma possono essere utilizzati come commenti:

```
"""
This type of comment spans multiple lines.
These are mostly used for documentation of functions, classes and modules.
"""
```

Accesso programmatico alle docstring

Le docstring sono, a differenza dei normali commenti, memorizzate come attributo della funzione che documentano, il che significa che è possibile accedervi a livello di programmazione.

Una funzione di esempio

```
def func():
    """This is a function that does nothing at all"""
    return
```

È possibile accedere alla docstring utilizzando l'attributo `__doc__` :

```
print(func.__doc__)
```

```
    Questa è una funzione che non fa nulla
```

```
help(func)
```

```
    Aiuto sulla funzione func nel modulo __main__ :

    func()

    Questa è una funzione che non fa nulla
```

Un'altra funzione di esempio

`function.__doc__` è solo la docstring effettiva come una stringa, mentre la funzione `help` fornisce informazioni generali su una funzione, inclusa la docstring. Ecco un esempio più utile:

```
def greet(name, greeting="Hello"):
    """Print a greeting to the user `name`

    Optional parameter `greeting` can change what they're greeted with."""

    print("{} {}".format(greeting, name))
```

```
help(greet)
```

```
    Aiuto sulla funzione greet nel modulo __main__ :

    greet(name, greeting='Hello')

    Stampa un saluto al name utente
    Il greeting facoltativo dei parametri può modificare ciò con cui vengono salutati.
```

Vantaggi delle docstring rispetto ai commenti regolari

Non è sufficiente inserire alcuna docstring o un commento regolare in una funzione.

```
def greet(name, greeting="Hello"):
    # Print a greeting to the user `name`
    # Optional parameter `greeting` can change what they're greeted with.

    print("{} {}".format(greeting, name))
```

```
print(greet.__doc__)
```

Nessuna

```
help(greet)
```

Aiuto sulla funzione di benvenuto nel modulo **principale** :

```
greet(name, greeting='Hello')
```

Scrivi la documentazione usando le docstring

Una docstring è un commento su più righe usato per documentare moduli, classi, funzioni e metodi. Deve essere la prima affermazione del componente che descrive.

```
def hello(name):
    """Greet someone.

    Print a greeting ("Hello") for the person with the given name.
    """

    print("Hello "+name)
```

```
class Greeter:
    """An object used to greet people.

    It contains multiple greeting functions for several languages
    and times of the  day.
    """
```

È possibile accedere al valore di docstring all'interno del programma ed è, ad esempio, utilizzato dal comando `help` .

Convenzioni sulla sintassi

PEP 257

PEP 257 definisce uno standard di sintassi per i commenti di docstring. In pratica consente due tipi:

- Docstrings a una riga:

Secondo PEP 257, dovrebbero essere utilizzati con funzioni brevi e semplici. Tutto è posto in una riga, ad esempio:

```
def hello():
    """Say hello to your friends."""
    print("Hello my friends!")
```

La docstring terminerà con un periodo, il verbo dovrebbe essere nella forma imperativa.

- Docstrings multilinea:

La docintazione su più righe dovrebbe essere utilizzata per funzioni, moduli o classi più lunghi e complessi.

```
def hello(name, language="en"):
    """Say hello to a person.

    Arguments:
    name: the name of the person
    language: the language in which the person should be greeted
    """

    print(greeting[language]+" "+name)
```

Iniziano con un breve sommario (equivalente al contenuto di una docstring su una riga) che può trovarsi sulla stessa riga delle virgolette o sulla riga successiva, fornire ulteriori dettagli e parametri di lista e valori di ritorno.

Nota PEP 257 definisce quali informazioni dovrebbero essere fornite all'interno di una docstring, non definisce in quale formato deve essere fornita. Questo è stato il motivo per cui le altre parti e gli strumenti di analisi della documentazione hanno specificato i propri standard per la documentazione, alcuni dei quali sono elencati di seguito e in questa domanda .

Sfinge

Sfinge è uno strumento per generare documentazione basata su HTML per progetti Python basati su docstring. Il suo linguaggio di markup usato è reStructuredText . Definiscono i propri standard per la documentazione, pythonhosted.org ospita una descrizione molto buona di loro . Il formato Sfinge è ad esempio utilizzato da pyCharm IDE .

Una funzione sarebbe documentata in questo modo usando il formato Sphinx / reStructuredText:

```
def hello(name, language="en"):
    """Say hello to a person.

    :param name: the name of the person
    :type name: str
    :param language: the language in which the person should be greeted
    :type language: str
    :return: a number
    :rtype: int
    """

    print(greeting[language]+" "+name)
    return 4
```

Google Style Guide di Google

Google ha pubblicato Google Python Style Guide che definisce le convenzioni di codifica per Python, inclusi i commenti alla documentazione. In confronto alla Sfinge / REST molte persone dicono che la documentazione secondo le linee guida di Google è meglio leggibile.

La pagina pythonhosted.org menzionata sopra fornisce anche alcuni esempi di buona documentazione secondo la Google Style Guide.

Usando il plugin Napoleon , Sphinx può anche analizzare la documentazione nel formato conforme alla Guida dello stile di Google.

Una funzione sarebbe documentata in questo modo utilizzando il formato della Guida di stile di Google:

```
def hello(name, language="en"):
    """Say hello to a person.

    Args:
        name: the name of the person as string
        language: the language code string

    Returns:
        A number.
    """

    print(greeting[language]+" "+name)
    return 4
```

Leggi Commenti e documentazione online: https://riptutorial.com/it/python/topic/4144/commenti-e-documentazione

Capitolo 26: Comprensione delle liste

introduzione

Le comprensioni delle liste in Python sono costrutti concisi e sintattici. Possono essere utilizzati per generare elenchi da altri elenchi applicando funzioni a ciascun elemento nell'elenco. La seguente sezione spiega e dimostra l'uso di queste espressioni.

Sintassi

- [x + 1 per x in (1, 2, 3)] # list comprehension, dà [2, 3, 4]
- (x + 1 per x in (1, 2, 3)) espressione del generatore #, darà 2, quindi 3, quindi 4
- [x per x in (1, 2, 3) se x% 2 == 0] # list comprehension with filter, dà [2]
- [x + 1 if x% 2 == 0 else x per x in (1, 2, 3)] # list comprehension with ternary
- [x + 1 if x% 2 == 0 else x per x nel range (-3,4) se x> 0] # list comprehension with ternary e filtering
- {x per x in (1, 2, 2, 3)} # set comprehension, dà {1, 2, 3}
- {k: v per k, v in [('a', 1), ('b', 2)]} # dict comprensione, dà {'a': 1, 'b': 2} (python 2.7+ e Solo 3.0+)
- [x + y per x in [1, 2] per y in [10, 20]] # Anelli nidificati, restituisce [11, 21, 12, 22]
- [x + y per x in [1, 2, 3] se x> 2 per y in [3, 4, 5]] # Condizione verificata al 1 ° ciclo
- [x + y per x in [1, 2, 3] per y in [3, 4, 5] se x> 2] # Condizione selezionata al 2 per ciclo
- [x per x in xrange (10) if x% 2 == 0] # Condition checked if numbers in loop sono numeri dispari

Osservazioni

Le comprensioni sono costrutti sintattici che definiscono strutture dati o espressioni uniche per un particolare linguaggio. L'uso corretto delle comprensioni reinterpreta questi in espressioni facilmente comprensibili. Come espressioni, possono essere utilizzate:

- nella parte destra dei compiti
- come argomenti per funzionare chiamate
- nel corpo di una funzione lambda
- come dichiarazioni autonome. (Ad esempio: `[print(x) for x in range(10)]`)

Examples

Elenco delle comprensioni

Una comprensione delle liste crea una nuova `list` applicando un'espressione a ciascun elemento di un iterabile . La forma più elementare è:

```
[ <expression> for <element> in <iterable> ]
```

C'è anche una condizione opzionale "se":

```
[ <expression> for <element> in <iterable> if <condition> ]
```

Ogni `<element>` in `<iterable>` è collegato a `<expression>` se la (`<condition>`) `<condition>` valutata come vera . Tutti i risultati vengono restituiti contemporaneamente nella nuova lista. Le espressioni del generatore vengono valutate pigramente, ma la comprensione delle liste valuta immediatamente l'intero iteratore - consumando memoria proporzionale alla lunghezza dell'iteratore.

Per creare un `list` di interi quadrati:

```
squares = [x * x for x in (1, 2, 3, 4)]
# squares: [1, 4, 9, 16]
```

L'espressione `for` imposta `x` per ciascun valore a sua volta da `(1, 2, 3, 4)` . Il risultato dell'espressione `x * x` è aggiunto ad una `list` interna. L' `list` interno viene assegnato ai `squares` variabili quando completato.

Oltre ad un aumento di velocità (come spiegato qui), una comprensione di lista è approssimativamente equivalente al seguente ciclo:

```
squares = []
for x in (1, 2, 3, 4):
    squares.append(x * x)
# squares: [1, 4, 9, 16]
```

L'espressione applicata a ciascun elemento può essere complessa quanto necessaria:

```
# Get a list of uppercase characters from a string
[s.upper() for s in "Hello World"]
# ['H', 'E', 'L', 'L', 'O', ' ', 'W', 'O', 'R', 'L', 'D']

# Strip off any commas from the end of strings in a list
[w.strip(',') for w in ['these,', 'words,,', 'mostly', 'have,commas,']]
# ['these', 'words', 'mostly', 'have,commas']

# Organize letters in words more reasonably - in an alphabetical order
sentence = "Beautiful is better than ugly"
["".join(sorted(word, key = lambda x: x.lower())) for word in sentence.split()]
# ['aBefiltuu', 'is', 'beertt', 'ahnt', 'gluy']
```

---

altro

`else` può essere usato nei costrutti di comprensione delle liste, ma attenzione alla sintassi. Le clausole if / else devono essere usate prima `for` ciclo, non dopo:

```
# create a list of characters in apple, replacing non vowels with '*'
# Ex - 'apple' --> ['a', '*', '*', '*' ,'e']
```

```
[x for x in 'apple' if x in 'aeiou' else '*']
#SyntaxError: invalid syntax

# When using if/else together use them before the loop
[x if x in 'aeiou' else '*' for x in 'apple']
#['a', '*', '*', '*', 'e']
```

Nota: utilizza un costrutto linguistico diverso, un'espressione condizionale , che a sua volta non fa parte della sintassi di comprensione . Mentre il `if` dopo il `for…in` è una parte delle list comprehensions e viene usato per *filtrare* elementi dalla sorgente iterabile.

Doppia iterazione

L'ordine di doppia iterazione `[... for x in ... for y in ...]` è naturale o contro-intuitivo. La regola generale è quella di seguire un equivalente `for` ciclo:

```
def foo(i):
    return i, i + 0.5

for i in range(3):
    for x in foo(i):
        yield str(x)
```

Questo diventa:

```
[str(x)
    for i in range(3)
        for x in foo(i)
]
```

Questo può essere compresso in una riga come `[str(x) for i in range(3) for x in foo(i)]`

Mutazione sul posto e altri effetti collaterali

Prima di utilizzare la comprensione delle liste, comprendere la differenza tra le funzioni chiamate per i loro effetti collaterali (*mutanti* o funzioni sul *posto*) che in genere restituiscono `None` e le funzioni che restituiscono un valore interessante.

Molte funzioni (in particolare le funzioni *pure*) prendono semplicemente un oggetto e restituiscono un oggetto. Una funzione sul *posto* modifica l'oggetto esistente, che è chiamato *effetto collaterale* . Altri esempi includono operazioni di input e output come la stampa.

`list.sort()` ordina un elenco *sul posto* (nel senso che modifica l'elenco originale) e restituisce il valore `None` . Pertanto, non funzionerà come previsto in una comprensione di lista:

```
[x.sort() for x in [[2, 1], [4, 3], [0, 1]]]
# [None, None, None]
```

Invece, `sorted()` restituisce una `list` ordinata piuttosto che ordinare sul posto:

```
[sorted(x) for x in [[2, 1], [4, 3], [0, 1]]]
# [[1, 2], [3, 4], [0, 1]]
```

È possibile utilizzare la comprensione per gli effetti collaterali, come I / O o le funzioni sul posto. Tuttavia, un ciclo for è solitamente più leggibile. Mentre questo funziona in Python 3:

```
[print(x) for x in (1, 2, 3)]
```

Usa invece:

```
for x in (1, 2, 3):
    print(x)
```

In alcune situazioni, le funzioni di effetti collaterali *sono* adatte per la comprensione delle liste. `random.randrange()` ha l'effetto collaterale di cambiare lo stato del generatore di numeri casuali, ma restituisce anche un valore interessante. Inoltre, `next()` può essere chiamato su un iteratore.

Il seguente generatore di valori casuali non è puro, ma ha senso quando il generatore casuale viene reimpostato ogni volta che viene valutata l'espressione:

```
from random import randrange
[randrange(1, 7) for _ in range(10)]
# [2, 3, 2, 1, 1, 5, 2, 4, 3, 5]
```

Spazio bianco nella comprensione delle liste

Comprensioni di lista più complicate possono raggiungere una lunghezza indesiderata o diventare meno leggibili. Sebbene meno comune negli esempi, è possibile rompere una comprensione di lista in più linee in questo modo:

```
[
    x for x
    in 'foo'
    if x not in 'bar'
]
```

Comprensioni del dizionario

Una comprensione del dizionario è simile alla comprensione di una lista, tranne per il fatto che produce un oggetto dizionario invece di una lista.

Un esempio di base:

Python 2.x 2.7

```
{x: x * x for x in (1, 2, 3, 4)}
# Out: {1: 1, 2: 4, 3: 9, 4: 16}
```

che è solo un altro modo di scrivere:

```
dict((x, x * x) for x in (1, 2, 3, 4))
# Out: {1: 1, 2: 4, 3: 9, 4: 16}
```

Come per la comprensione di una lista, possiamo usare una dichiarazione condizionale all'interno della comprensione del ditt per produrre solo gli elementi dict che soddisfano qualche criterio.

Python 2.x 2.7

```
{name: len(name) for name in ('Stack', 'Overflow', 'Exchange') if len(name) > 6}
# Out: {'Exchange': 8, 'Overflow': 8}
```

Oppure, riscritto usando un'espressione di generatore.

```
dict((name, len(name)) for name in ('Stack', 'Overflow', 'Exchange') if len(name) > 6)
# Out: {'Exchange': 8, 'Overflow': 8}
```

A partire da un dizionario e usando la comprensione del dizionario come filtro della coppia chiave-valore

Python 2.x 2.7

```
initial_dict = {'x': 1, 'y': 2}
{key: value for key, value in initial_dict.items() if key == 'x'}
# Out: {'x': 1}
```

Commutazione chiave e valore del dizionario (dizionario invertito)

Se hai un dict contenente semplici valori *hashable* (i valori duplicati potrebbero avere risultati imprevisti):

```
my_dict = {1: 'a', 2: 'b', 3: 'c'}
```

e volevi scambiare le chiavi e i valori, puoi adottare diversi approcci a seconda del tuo stile di codifica:

- `swapped = {v: k for k, v in my_dict.items()}`
- `swapped = dict((v, k) for k, v in my_dict.iteritems())`
- `swapped = dict(zip(my_dict.values(), my_dict))`
- `swapped = dict(zip(my_dict.values(), my_dict.keys()))`
- `swapped = dict(map(reversed, my_dict.items()))`

```
print(swapped)
```

```
# Out: {a: 1, b: 2, c: 3}
```

Python 2.x 2.3

Se il tuo dizionario è grande, considera l' *importazione di itertools* e utilizza `izip` o `imap` .

Unione di dizionari

Combina dizionari e opzionalmente sostituisci i vecchi valori con una comprensione del dizionario annidata.

```
dict1 = {'w': 1, 'x': 1}
dict2 = {'x': 2, 'y': 2, 'z': 2}

{k: v for d in [dict1, dict2] for k, v in d.items()}
# Out: {'w': 1, 'x': 2, 'y': 2, 'z': 2}
```

Tuttavia, il disimballaggio del dizionario (PEP 448) potrebbe essere preferito.

Python 3.x 3.5

```
{**dict1, **dict2}
# Out: {'w': 1, 'x': 2, 'y': 2, 'z': 2}
```

Nota : le comprensibilità del dizionario sono state aggiunte in Python 3.0 e supportate a 2.7+, a differenza delle list comprehensions, che sono state aggiunte in 2.0. Le versioni <2.7 possono usare le espressioni del generatore e il `dict()` incorporato per simulare il comportamento delle comprensioni del dizionario.

Espressioni del generatore

Le espressioni del generatore sono molto simili alle list comprehensions. La differenza principale è che non crea una serie completa di risultati contemporaneamente; crea un oggetto generatore che può quindi essere iterato sopra.

Ad esempio, vedere la differenza nel seguente codice:

```
# list comprehension
[x**2 for x in range(10)]
# Output: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Python 2.x 2.4

```
# generator comprehension
(x**2 for x in xrange(10))
# Output: <generator object <genexpr> at 0x11b4b7c80>
```

Questi sono due oggetti molto diversi:

- la comprensione della lista restituisce un oggetto `list` mentre la comprensione del

generatore restituisce un `generator` .

- `generator` oggetti del `generator` non possono essere indicizzati e si avvale della funzione `next` per ottenere gli oggetti in ordine.

Nota : usiamo `xrange` poiché crea anche un oggetto generatore. Se usassimo l'intervallo, sarebbe creata una lista. Inoltre, `xrange` esiste solo nella versione successiva di python 2. In python 3, l' `range` restituisce solo un generatore. Per ulteriori informazioni, vedere l'esempio delle *differenze tra intervallo e xrange* .

Python 2.x 2.4

```
g = (x**2 for x in xrange(10))
print(g[0])
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'generator' object has no attribute '__getitem__'
```

```
g.next()  # 0
g.next()  # 1
g.next()  # 4
...
g.next()  # 81

g.next()  # Throws StopIteration Exception
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Python 3.x 3.0

NOTA: la funzione `g.next()` dovrebbe essere sostituita da `next(g)` e `xrange` con `range` poiché `Iterator.next()` e `xrange()` non esistono in Python 3.

Sebbene entrambi possano essere ripetuti in modo simile:

```
for i in [x**2 for x in range(10)]:
    print(i)

"""
Out:
0
1
4
...
81
"""
```

Python 2.x 2.4

```
for i in (x**2 for x in xrange(10)):
    print(i)

"""
Out:
0
1
4
.
.
.
81
"""
```

Casi d'uso

Le espressioni del generatore vengono valutate pigramente, il che significa che generano e restituiscono ciascun valore solo quando il generatore viene iterato. Ciò è spesso utile durante l'iterazione di set di dati di grandi dimensioni, evitando la necessità di creare un duplicato del set di dati nella memoria:

```
for square in (x**2 for x in range(1000000)):
    #do something
```

Un altro caso d'uso comune è quello di evitare di iterare su un intero iterabile se ciò non è necessario. In questo esempio, un elemento viene recuperato da un'API remota con ogni iterazione di `get_objects()` . Migliaia di oggetti possono esistere, devono essere recuperati uno alla volta e abbiamo solo bisogno di sapere se esiste un oggetto che corrisponde a un modello. Usando un'espressione di generatore, quando incontriamo un oggetto che corrisponde al modello.

```
def get_objects():
    """Gets objects from an API one by one"""
    while True:
        yield get_next_item()

def object_matches_pattern(obj):
    # perform potentially complex calculation
    return matches_pattern

def right_item_exists():
    items = (object_matched_pattern(each) for each in get_objects())
    for item in items:
        if item.is_the_right_one:


            return True
    return False
```

Imposta Comprensioni

La comprensione dell'insieme è simile alla comprensione di elenchi e dizionari , ma produce un insieme , che è una collezione non ordinata di elementi unici.

Python 2.x 2.7

```
# A set containing every value in range(5):
{x for x in range(5)}
# Out: {0, 1, 2, 3, 4}

# A set of even numbers between 1 and 10:
{x for x in range(1, 11) if x % 2 == 0}
# Out: {2, 4, 6, 8, 10}

# Unique alphabetic characters in a string of text:
text = "When in the Course of human events it becomes necessary for one people..."
{ch.lower() for ch in text if ch.isalpha()}
# Out: set(['a', 'c', 'b', 'e', 'f', 'i', 'h', 'm', 'l', 'o',
#           'n', 'p', 's', 'r', 'u', 't', 'w', 'v', 'y'])
```

Dimostrazione dal vivo

Tieni presente che i set non sono ordinati. Ciò significa che l'ordine dei risultati nell'insieme può essere diverso da quello presentato negli esempi precedenti.

Nota : la comprensione dei set è disponibile da python 2.7+, a differenza delle list comprehensions, che sono state aggiunte in 2.0. In Python 2.2 a Python 2.6, la funzione `set()` può essere utilizzata con un'espressione generatore per produrre lo stesso risultato:

Python 2.x 2.2

```
set(x for x in range(5))
# Out: {0, 1, 2, 3, 4}
```

Evita operazioni ripetitive e costose usando la clausola condizionale

Considera la sotto lista di comprensione:

```
>>> def f(x):
...     import time
...     time.sleep(.1)       # Simulate expensive function
...     return x**2

>>> [f(x) for x in range(1000) if f(x) > 10]
[16, 25, 36, ...]
```

Ciò comporta due chiamate a `f(x)` per 1.000 valori di `x` : una chiamata per la generazione del valore e l'altra per il controllo della condizione `if` . Se `f(x)` è un'operazione particolarmente costosa, ciò può avere implicazioni significative sulle prestazioni. Peggio ancora, se chiamare `f()` ha effetti collaterali, può avere risultati sorprendenti.

Invece, dovresti valutare l'operazione costosa solo una volta per ogni valore di `x` generando un iterabile intermedio (espressione del generatore) come segue:

```
>>> [v for v in (f(x) for x in range(1000)) if v > 10]
[16, 25, 36, ...]
```

Oppure, usando l'equivalente della mappa integrata:

```
>>> [v for v in map(f, range(1000)) if v > 10]
[16, 25, 36, ...]
```

Un altro modo che potrebbe risultare in un codice più leggibile consiste nel mettere il risultato parziale (`v` nell'esempio precedente) in un iterabile (come un elenco o una tupla) e quindi scorrere su di esso. Dato che `v` sarà l'unico elemento nel iterabile, il risultato è che ora abbiamo un riferimento all'output della nostra funzione lenta calcolata una sola volta:

```
>>> [v for x in range(1000) for v in [f(x)] if v > 10]
[16, 25, 36, ...]
```

Tuttavia, in pratica, la logica del codice può essere più complicata ed è importante tenerlo leggibile. In generale, una funzione di generatore separata è raccomandata su un one-liner complesso:

```
>>> def process_prime_numbers(iterable):
...     for x in iterable:
...         if is_prime(x):
...             yield f(x)
...
>>> [x for x in process_prime_numbers(range(1000)) if x > 10]
[11, 13, 17, 19, ...]
```

Un altro modo per evitare che il calcolo `f(x)` più volte è quello di utilizzare il `@functools.lru_cache()` (Python 3.2+) decoratore su `f(x)` . In questo modo, poiché l'output di `f` per l'input `x` è già stato calcolato una volta, la seconda funzione invocata dalla comprensione dell'elenco originale sarà veloce quanto una ricerca di dizionario. Questo approccio utilizza la memoizzazione per migliorare l'efficienza, che è paragonabile all'uso di espressioni generatrici.

Di 'che devi appiattire una lista

```
l = [[1, 2, 3], [4, 5, 6], [7], [8, 9]]
```

Alcuni dei metodi potrebbero essere:

```
reduce(lambda x, y: x+y, l)

sum(l, [])

list(itertools.chain(*l))
```

Tuttavia la comprensione delle liste fornirebbe la migliore complessità temporale.

```
[item for sublist in l for item in sublist]
```

Le scorciatoie basate su + (incluso l'uso implicito in somma) sono, per necessità, O (L ^ 2) quando

ci sono sottoliste L - poiché l'elenco dei risultati intermedi continua ad allungarsi, ad ogni passaggio viene ottenuto un nuovo elenco di risultati intermedi assegnato e tutti gli elementi nel risultato intermedio precedente devono essere copiati sopra (così come alcuni nuovi aggiunti alla fine). Quindi (per semplicità e senza effettiva perdita di generalità) dite di avere L Liste di elementi I ciascuno: i primi elementi I vengono copiati avanti e indietro L-1 volte, il secondo I elementi L-2 volte, e così via; il numero totale di copie è I volte la somma di x per x da 1 a L esclusa, cioè I * (L ** 2) / 2.

La comprensione dell'elenco genera solo una lista, una volta, e copia ogni oggetto (dalla sua posizione originale di residenza alla lista dei risultati) anche esattamente una volta.

Comprensioni che riguardano le tuple

La clausola `for` di una list list può specificare più di una variabile:

```
[x + y for x, y in [(1, 2), (3, 4), (5, 6)]]
# Out: [3, 7, 11]

[x + y for x, y in zip([1, 3, 5], [2, 4, 6])]
# Out: [3, 7, 11]
```

Questo è come regolare `for` loop:

```
for x, y in [(1,2), (3,4), (5,6)]:
    print(x+y)
# 3
# 7
# 11
```

Nota comunque, se l'espressione che inizia la comprensione è una tupla, allora deve essere tra parentesi:

```
[x, y for x, y in [(1, 2), (3, 4), (5, 6)]]
# SyntaxError: invalid syntax

[(x, y) for x, y in [(1, 2), (3, 4), (5, 6)]]
# Out: [(1, 2), (3, 4), (5, 6)]
```

Conteggio delle ricorrenze utilizzando la comprensione

Quando vogliamo contare il numero di elementi in un iterabile, che soddisfano alcune condizioni, possiamo usare la comprensione per produrre una sintassi idiomatica:

```
# Count the numbers in `range(1000)` that are even and contain the digit `9`:
print (sum(
    1 for x in range(1000)
    if x % 2 == 0 and
    '9' in str(x)
))
# Out: 95
```

Il concetto di base può essere riassunto come:

1. Iterare sugli elementi `range(1000)` .
2. Concatenate tutto il necessario `if` le condizioni.
3. Usa 1 come *espressione* per restituire 1 per ogni oggetto che soddisfa le condizioni.
4. Riassumi tutti i `1` s per determinare il numero di articoli che soddisfano le condizioni.

Nota : qui non stiamo raccogliendo i `1` s in una lista (notare l'assenza di parentesi quadre), ma stiamo passando quelli direttamente alla funzione `sum` che li sta sommando. Questo è chiamato *un'espressione generatore* , che è simile a una comprensione.

Modifica dei tipi in un elenco

I dati quantitativi vengono spesso letti come stringhe che devono essere convertite in tipi numerici prima dell'elaborazione. I tipi di tutti gli elementi dell'elenco possono essere convertiti con una funzione Comprehension elenco o `map()` .

```
# Convert a list of strings to integers.
items = ["1","2","3","4"]
[int(item) for item in items]
# Out: [1, 2, 3, 4]

# Convert a list of strings to float.
items = ["1","2","3","4"]
map(float, items)
# Out:[1.0, 2.0, 3.0, 4.0]
```

Leggi Comprensione delle liste online: https://riptutorial.com/it/python/topic/196/comprensione-delle-liste

Capitolo 27: Concorrenza Python

Osservazioni

Gli sviluppatori Python hanno fatto in modo che l'API tra `threading` e `multiprocessing` sia simile, in modo che il passaggio tra le due varianti sia più semplice per i programmatori.

Examples

Il modulo di filettatura

```
from __future__ import print_function
import threading
def counter(count):
    while count > 0:
        print("Count value", count)
        count -= 1
    return

t1 = threading.Thread(target=countdown,args=(10,))
t1.start()
t2 = threading.Thread(target=countdown,args=(20,))
t2.start()
```

In alcune implementazioni di Python come CPython, il vero parallelismo non si ottiene usando i thread a causa dell'uso di ciò che è noto come GIL o **G** lobal **I** nterpreter **L** ock.

Ecco un'eccellente panoramica della concorrenza Python:

Concorrenza Python di David Beazley (YouTube)

Il modulo multiprocessing

```
from __future__ import print_function
import multiprocessing


def countdown(count):
    while count > 0:
        print("Count value", count)
        count -= 1
    return

if __name__ == "__main__":
    p1 = multiprocessing.Process(target=countdown, args=(10,))
    p1.start()

    p2 = multiprocessing.Process(target=countdown, args=(20,))
    p2.start()

    p1.join()
```

```
    p2.join()
```

Qui, ogni funzione viene eseguita in un nuovo processo. Poiché una nuova istanza di Python VM sta eseguendo il codice, non esiste `GIL` e il parallelismo viene eseguito su più core.

Il metodo `Process.start` avvia questo nuovo processo ed esegue la funzione passata nell'argomento di `target` con gli argomenti `args` . Il metodo `Process.join` attende la fine dell'esecuzione dei processi `p1` e `p2` .

I nuovi processi vengono lanciati in modo diverso a seconda della versione di python e della piastraforma su cui è in esecuzione il codice, *ad esempio* :

- Windows utilizza `spawn` per creare il nuovo processo.
- Con i sistemi unix e la versione precedente alla 3.3, i processi vengono creati utilizzando una `fork` .
  Si noti che questo metodo non rispetta l'utilizzo POSIX della forcella e quindi porta a comportamenti imprevisti, specialmente quando si interagisce con altre librerie di multiprocessing.
- Con il sistema unix e la versione 3.4+, è possibile scegliere di avviare i nuovi processi con `fork` , `forkserver` o `spawn` utilizzando `multiprocessing.set_start_method` all'inizio del programma. `forkserver` metodi di `forkserver` e `spawn` sono più lenti della forking ma evitano comportamenti imprevisti.

Utilizzo della forcella POSIX :

> Dopo un fork in un programma multithread, il bambino può chiamare in modo sicuro solo le funzioni sicure del segnale asincrono fino al momento in cui chiama execve.
> (vedi)

Usando fork, verrà lanciato un nuovo processo con lo stesso esatto stato per tutto il mutex corrente ma verrà lanciato solo il `MainThread` . Questo non è sicuro in quanto potrebbe portare a condizioni di gara *ad esempio* :

- Se si usa un `Lock` in `MainThread` e lo si passa ad un altro thread che si suppone lo blocchi ad un certo punto. Se la `fork` contemporaneamente, il nuovo processo inizierà con un blocco bloccato che non verrà mai rilasciato poiché il secondo thread non esiste in questo nuovo processo.

In realtà, questo tipo di comportamento non dovrebbe verificarsi in Python puro poiché il `multiprocessing` gestisce correttamente, ma se si interagisce con altre librerie, questo tipo di comportamento può verificarsi, portando ad un arresto anomalo del sistema (ad esempio con numpy / accelerato su macOS).

Passaggio di dati tra processi di multiprocessing

Poiché i dati sono sensibili quando vengono gestiti tra due thread (si pensi che la lettura simultanea e la scrittura simultanea possano essere in conflitto tra loro, causando condizioni di competizione), è stata creata una serie di oggetti unici per facilitare il passaggio di dati avanti e

indietro tra i thread. Qualsiasi operazione veramente atomica può essere utilizzata tra i thread, ma è sempre sicuro attenersi a Queue.

```
import multiprocessing
import queue
my_Queue=multiprocessing.Queue()
#Creates a queue with an undefined maximum size
#this can be dangerous as the queue becomes increasingly large
#it will take a long time to copy data to/from each read/write thread
```

La maggior parte delle persone suggerirà che quando si usa la coda, si inseriscano sempre i dati della coda in una prova: tranne: blocco invece di usare vuoto. Tuttavia, per le applicazioni in cui non importa se si ignora un ciclo di scansione (i dati possono essere inseriti nella coda mentre si `queue.Empty==True` stati dalla `queue.Empty==True` to `queue.Empty==False`) di solito è meglio mettere i read e scrivere l'accesso in quello che io chiamo un blocco di Iftry, perché una dichiarazione di "se" è tecnicamente più performante di quella che cattura l'eccezione.

```
import multiprocessing
import queue
'''Import necessary Python standard libraries, multiprocessing for classes and queue for the
queue exceptions it provides'''
def Queue_Iftry_Get(get_queue, default=None, use_default=False, func=None, use_func=False):
    '''This global method for the Iftry block is provided for it's reuse and
standard functionality, the if also saves on performance as opposed to catching
 the exception, which is expencive.
        It also allows the user to specify a function for the outgoing data to use,
 and a default value to return if the function cannot return the value from the queue'''
        if get_queue.empty():
            if use_default:
                return default
        else:
            try:
                value = get_queue.get_nowait()
            except queue.Empty:
                if use_default:
                    return default
            else:
                if use_func:
                    return func(value)
                else:
                    return value
    def Queue_Iftry_Put(put_queue, value):
        '''This global method for the Iftry block is provided because of its reuse
and
standard functionality, the If also saves on performance as opposed to catching
 the exception, which is expensive.
        Return True if placing value in the queue was successful. Otherwise, false'''
        if put_queue.full():
            return False
        else:
            try:
                put_queue.put_nowait(value)
            except queue.Full:
                return False
            else:
                return True
```

Leggi Concorrenza Python online: https://riptutorial.com/it/python/topic/3357/concorrenza-python

Capitolo 28: Condizionali

introduzione

Le espressioni condizionali, che includono parole chiave come if, elif e altro, forniscono ai programmi Python la possibilità di eseguire azioni diverse a seconda della condizione booleana: True o False. Questa sezione copre l'uso di condizionali Python, logica booleana e dichiarazioni ternarie.

Sintassi

- <espressione> se <condizionale> altro <espressione> # Operatore ternario

Examples

se, elif, e altro

In Python puoi definire una serie di condizionali usando `if` per il primo, `elif` per il resto, fino al finale (opzionale) `else` per qualsiasi cosa non catturata dagli altri condizionali.

```
number = 5

if number > 2:
    print("Number is bigger than 2.")
elif number < 2:  # Optional clause (you can have multiple elifs)
    print("Number is smaller than 2.")
else:  # Optional clause (you can only have one else)
    print("Number is 2.")
```

Il `Number is bigger than 2` uscite `Number is bigger than 2`

Usando `else if` invece di `elif` si innesca un errore di sintassi e non è permesso.

Espressione condizionale (o "Operatore ternario")

L'operatore ternario viene utilizzato per espressioni condizionali in linea. È meglio utilizzato in operazioni semplici e concise che sono facilmente leggibili.

- L'ordine degli argomenti è diverso da molti altri linguaggi (come C, Ruby, Java, ecc.), Che possono portare a bug quando le persone che non hanno familiarità con il comportamento "sorprendente" di Python lo usano (potrebbero invertire l'ordine).
- Alcuni lo trovano "ingombrante", poiché va contro il normale flusso del pensiero (pensando prima alla condizione e poi agli effetti).

```
n = 5
```

```
"Greater than 2" if n > 2 else "Smaller than or equal to 2"
# Out: 'Greater than 2'
```

Il risultato di questa espressione sarà come viene letto in inglese - se l'espressione condizionale è True, allora valuterà l'espressione sul lato sinistro, altrimenti, il lato destro.

Le operazioni di Tenary possono anche essere annidate, come qui:

```
n = 5
"Hello" if n > 10 else "Goodbye" if n > 5 else "Good day"
```

Forniscono anche un metodo per includere condizionali nelle funzioni lambda .

Se la dichiarazione

```
if condition:
    body
```

Le istruzioni `if` controllano la condizione. Se valuta su `True` , esegue il corpo dell'istruzione `if` . Se valuta su `False` , salta il corpo.

```
if True:
    print "It is true!"
>> It is true!

if False:
    print "This won't get printed.."
```

La condizione può essere qualsiasi espressione valida:

```
if 2 + 2 == 4:
    print "I know math!"
>> I know math!
```

Altra affermazione

```
if condition:
    body
else:
    body
```

L'istruzione else eseguirà il suo corpo solo se le istruzioni condizionali precedenti valgono tutte su False.

```
if True:
    print "It is true!"
else:
    print "This won't get printed.."

# Output: It is true!
```

```
if False:
    print "This won't get printed.."
else:
    print "It is false!"

# Output: It is false!
```

Espressioni logiche booleane

Le espressioni logiche booleane, oltre a valutare `True` o `False` , restituiscono il *valore* interpretato come `True` o `False` . È un modo Pythonic per rappresentare la logica che altrimenti richiederebbe un test if-else.

E l'operatore

L'operatore `and` valuta tutte le espressioni e restituisce l'ultima espressione se tutte le espressioni vengono valutate su `True` . Altrimenti restituisce il primo valore che restituisce `False` :

```
>>> 1 and 2
2

>>> 1 and 0
0

>>> 1 and "Hello World"
"Hello World"

>>> "" and "Pancakes"
""
```

O operatore

L'operatore `or` valuta le espressioni da sinistra a destra e restituisce il primo valore che restituisce `True` o l'ultimo valore (se nessuno è `True`).

```
>>> 1 or 2
1

>>> None or 1
1

>>> 0 or []
[]
```

Valutazione pigra

Quando usi questo approccio, ricorda che la valutazione è pigra. Le espressioni che non devono essere valutate per determinare il risultato non vengono valutate. Per esempio:

```
>>> def print_me():
        print('I am here!')
>>> 0 and print_me()
0
```

Nell'esempio precedente, `print_me` non viene mai eseguito perché Python può determinare che l'intera espressione è `False` quando incontra lo `0` (`False`). Tieni questo a mente se `print_me` deve essere eseguito per servire la logica del tuo programma.

Test per più condizioni

Un errore comune durante il controllo di più condizioni consiste nell'applicare la logica in modo errato.

Questo esempio sta tentando di verificare se due variabili sono ciascuna maggiore di 2. L'istruzione viene valutata come - `if (a) and (b > 2)` . Questo produce un risultato inaspettato perché `bool(a)` valutato come `True` quando `a` non è zero.

```
>>> a = 1
>>> b = 6
>>> if a and b > 2:
...     print('yes')
... else:
...     print('no')

yes
```

Ogni variabile deve essere confrontata separatamente.

```
>>> if a > 2 and b > 2:
...     print('yes')
... else:
...     print('no')

no
```

Un altro errore simile viene fatto quando si controlla se una variabile è uno di più valori. L'affermazione in questo esempio è valutata come - `if (a == 3) or (4) or (6)` . Questo produce un risultato inaspettato perché `bool(4)` e `bool(6)` valutano ciascuno in `True`

```
>>> a = 1
>>> if a == 3 or 4 or 6:
...     print('yes')
... else:
...     print('no')

yes
```

Anche in questo caso ogni confronto deve essere effettuato separatamente

```
>>> if a == 3 or a == 4 or a == 6:
...     print('yes')
... else:
...     print('no')

no
```

Usare l'operatore in è il modo canonico per scrivere questo.

```
>>> if a in (3, 4, 6):
...     print('yes')
... else:
...     print('no')

no
```

Valori della verità

I seguenti valori sono considerati falsi, in quanto valutano `False` se applicati a un operatore booleano.

- Nessuna
- falso
- `0` , o qualsiasi valore numerico equivalente a zero, ad esempio `0L` , `0.0` , `0j`
- Sequenze vuote: `''` , `""` , `()` , `[]`
- Mappature vuote: `{}`
- Tipi definiti dall'utente in cui i metodi `__bool__` o `__len__` restituiscono 0 o `False`

Tutti gli altri valori in Python valutano `True` .

Nota: un errore comune è semplicemente controllare la Falsità di un'operazione che restituisce valori Falsey diversi in cui la differenza è importante. Ad esempio, usando `if foo()` piuttosto che il più esplicito `if foo() is None`

Usando la funzione cmp per ottenere il risultato del confronto di due oggetti

Python 2 include una funzione `cmp` che consente di determinare se un oggetto è inferiore, uguale o superiore a un altro oggetto. Questa funzione può essere utilizzata per selezionare una scelta da un elenco basato su una di queste tre opzioni.

Supponiamo di dover stampare `'greater than'` se `x > y` , `'less than'` se `x < y` e `'equal'` se `x == y` .

```
['equal', 'greater than', 'less than', ][cmp(x,y)]

# x,y = 1,1 output: 'equal'
# x,y = 1,2 output: 'less than'
# x,y = 2,1 output: 'greater than'
```

`cmp(x,y)` restituisce i seguenti valori

Confronto	Risultato
x <y	-1
x == y	0
x> y	1

Questa funzione è stata rimossa su Python 3. È possibile utilizzare la `cmp_to_key(func)` aiuto `cmp_to_key(func)` situata in `functools` in Python 3 per convertire le vecchie funzioni di confronto in funzioni chiave.

Valutazione dell'espressione condizionale usando le comprensioni degli elenchi

Python ti permette di hackerare le list comprehensions per valutare le espressioni condizionali.

Per esempio,

```
[value_false, value_true][<conditional-test>]
```

Esempio:

```
>> n = 16
>> print [10, 20][n <= 15]
10
```

Qui `n<=15` restituisce `False` (che equivale a 0 in Python). Quindi, cosa sta valutando Python è:

```
[10, 20][n <= 15]
==> [10, 20][False]
==> [10, 20][0]     #False==0, True==1 (Check Boolean Equivalencies in Python)
==> 10
```

Python 2.x 2.7

Il metodo `__cmp__` integrato ha restituito 3 possibili valori: 0, 1, -1, dove cmp (x, y) ha restituito 0: se entrambi gli oggetti erano uguali 1: x> y -1: x <y

Questo può essere usato con le list comprehensions per restituire il primo (ovvero indice 0), il secondo (cioè indice 1) e l'ultimo (cioè indice -1) elemento della lista. Dandoci un condizionale di questo tipo:

```
[value_equals, value_greater, value_less][<conditional-test>]
```

Infine, in tutti gli esempi sopra Python valuta entrambi i rami prima di sceglierne uno. Per valutare solo il ramo scelto:

```
[lambda: value_false, lambda: value_true][<test>]()
```

dove l'aggiunta di `()` alla fine assicura che le funzioni lambda vengano solo chiamate / valutate alla fine. Pertanto, valutiamo solo il ramo scelto.

Esempio:

```
count = [lambda:0, lambda:N+1][count==N]()
```

Verificare se un oggetto è Nessuno e assegnarlo

Spesso vorrai assegnare qualcosa a un oggetto se è `None` , a indicare che non è stato assegnato. Useremo un `aDate` .

Il modo più semplice per farlo è usare il test `is None` .

```
if aDate is None:
    aDate=datetime.date.today()
```

(Nota che è più Pythonico a dire `is None` invece di `== None` .)

Ma questo può essere ottimizzato un po 'sfruttando l'idea che `not None` valuterà a `True` in un'espressione booleana. Il seguente codice è equivalente:

```
if not aDate:
    aDate=datetime.date.today()
```

Ma c'è un modo più pitonico. Il seguente codice è anche equivalente:

```
aDate=aDate or datetime.date.today()
```

Questo fa una valutazione del cortocircuito . Se `aDate` è inizializzato e `not None` è `not None` , viene assegnato a se stesso senza alcun effetto netto. Se `is None` , quindi `datetime.date.today()` viene assegnato a `aDate` .

Leggi Condizionali online: https://riptutorial.com/it/python/topic/1111/condizionali

Capitolo 29: ConfigParser

introduzione

Questo modulo fornisce la classe ConfigParser che implementa un linguaggio di configurazione di base nei file INI. Puoi usarlo per scrivere programmi Python che possono essere facilmente personalizzati dagli utenti finali.

Sintassi

- Ogni nuova riga contiene una nuova coppia di valori chiave separati dal segno =
- Le chiavi possono essere separate in sezioni
- Nel file INI, ogni titolo di sezione è scritto tra parentesi: []

Osservazioni

Tutti i valori restituiti da `ConfigParser.ConfigParser().get` sono stringhe. Può essere convertito in tipi più comuni grazie a `eval`

Examples

Utilizzo di base

In config.ini:

```
[DEFAULT]
debug = True
name = Test
password = password

[FILES]
path = /path/to/file
```

In Python:

```
from ConfigParser import ConfigParser
config = ConfigParser()

#Load configuration file
config.read("config.ini")

# Access the key "debug" in "DEFAULT" section
config.get("DEFAULT", "debug")
# Return 'True'

# Access the key "path" in "FILES" destion
config.get("FILES", "path")
# Return '/path/to/file'
```

Creazione di file di configurazione in modo programmatico

Il file di configurazione contiene sezioni, ogni sezione contiene chiavi e valori. il modulo configparser può essere usato per leggere e scrivere i file di configurazione. Creazione del file di configurazione: -

```
import configparser
config = configparser.ConfigParser()
config['settings']={'resolution':'320x240',
                    'color':'blue'}
with open('example.ini', 'w') as configfile:
    config.write(configfile)
```

Il file di output contiene sotto la struttura

```
[settings]
resolution = 320x240
color = blue
```

Se si desidera modificare un campo particolare, ottenere il campo e assegnare il valore

```
settings=config['settings']
settings['color']='red'
```

Leggi ConfigParser online: https://riptutorial.com/it/python/topic/9186/configparser

Capitolo 30: confronti

Sintassi

- ! = - Non è uguale a
- == - È uguale a
- `>` - maggiore di
- `<` - meno di
- `>=` - maggiore o uguale a
- `<=` - inferiore o uguale a
- `is` - verifica se gli oggetti sono esattamente lo stesso oggetto
- is not = test se gli oggetti non sono esattamente lo stesso oggetto

Parametri

Parametro	Dettagli
X	Primo oggetto da confrontare
y	Secondo oggetto da confrontare

Examples

Maggiore o minore di

```
x > y
x < y
```

Questi operatori confrontano due tipi di valori, sono inferiori e maggiori degli operatori. Per i numeri questo semplicemente confronta i valori numerici per vedere quale è più grande:

```
12 > 4
# True
12 < 4
# False
1 < 4
# True
```

Per gli archi si confronteranno lessicograficamente, che è simile all'ordine alfabetico ma non

esattamente lo stesso.

```
"alpha" < "beta"
# True
"gamma" > "beta"
# True
"gamma" < "OMEGA"
# False
```

In questi confronti, le lettere minuscole sono considerate "maggiori di" maiuscole, motivo per cui `"gamma" < "OMEGA"` è falso. Se fossero tutti maiuscoli, restituirebbe il risultato ordinato in ordine alfabetico:

```
"GAMMA" < "OMEGA"
# True
```

Ogni tipo definisce il suo calcolo con `<` e `>` operatori in modo diverso, quindi è necessario indagare su cosa gli operatori intendono con un determinato tipo prima di utilizzarlo.

Non uguale a

```
x != y
```

Ciò restituisce `True` se `x` e `y` non sono uguali e altrimenti restituisce `False` .

```
12 != 1
# True
12 != '12'
# True
'12' != '12'
# False
```

Uguale a

```
x == y
```

Questa espressione valuta se `x` ed `y` sono lo stesso valore e restituisce il risultato come valore booleano. Generalmente sia il tipo che il valore devono corrispondere, quindi l'int `12` non è uguale alla stringa `'12'` .

```
12 == 12
# True
12 == 1
# False
'12' == '12'
# True
'spam' == 'spam'
# True
'spam' == 'spam '
# False
'12' == 12
```

```
# False
```

Nota che ogni tipo deve definire una funzione che verrà utilizzata per valutare se due valori sono gli stessi. Per i tipi built-in, queste funzioni si comportano come ci si aspetterebbe e valutano solo le cose in base all'essere lo stesso valore. Tuttavia, i tipi personalizzati potrebbero definire test di uguaglianza come qualsiasi cosa vorrebbero, incluso il ritorno sempre `True` o il ritorno di `False` .

Confronti a catena

È possibile confrontare più articoli con più operatori di confronto con il confronto a catena. Per esempio

```
x > y > z
```

è solo una breve forma di:

```
x > y and y > z
```

Verrà valutato su `True` solo se entrambi i confronti sono `True` .

La forma generale è

```
a OP b OP c OP d ...
```

Dove `OP` rappresenta una delle operazioni di confronto multiple che è possibile utilizzare e le lettere rappresentano espressioni valide arbitrarie.

> Si noti che `0 != 1 != 0` restituisce `True` , anche se `0 != 0` è `False` . A differenza della comune notazione matematica in cui `x != y != z` significa che `x` , `y` e `z` hanno valori diversi. Il concatenamento `==` operazioni ha il significato naturale nella maggior parte dei casi, poiché l'uguaglianza è generalmente transitiva.

Stile

Non esiste un limite teorico su quanti articoli e operazioni di confronto si utilizzano a patto di avere una sintassi corretta:

```
1 > -1 < 2 > 0.5 < 100 != 24
```

Quanto sopra restituisce `True` se ogni confronto restituisce `True` . Tuttavia, l'utilizzo del concatenamento concatenato non è un buon stile. Un buon collegamento sarà "direzionale", non più complicato di

```
1 > x > -4 > y != 8
```

Effetti collaterali

Non appena un confronto restituisce `False` , l'espressione viene valutata immediatamente su `False` , ignorando tutti i confronti rimanenti.

Si noti che l'espressione `exp` in `a > exp > b` sarà valutata solo una volta, mentre nel caso di

```
a > exp and exp > b
```

`exp` sarà calcolato due volte se `a > exp` è vero.

Confronto con `is` vs` == `

Un errore comune è confondere gli operatori di confronto di uguaglianza `is` e `==` .

`a == b` confronta il valore di `a` e `b` .

`a is b` confronterà le *identità* di `a` e `b` .

Illustrare:

```
a = 'Python is fun!'
b = 'Python is fun!'
a == b # returns True
a is b # returns False

a = [1, 2, 3, 4, 5]
b = a      # b references a
a == b     # True
a is b     # True
b = a[:]   # b now references a copy of a
a == b     # True
a is b     # False [!!]
```

Fondamentalmente, `is` può essere pensato come abbreviazione di `id(a) == id(b)` .

Oltre a ciò, ci sono stranezze dell'ambiente di runtime che complicano ulteriormente le cose. Le stringhe corte e gli interi piccoli restituiscono `True` se confrontati con `is` , a causa del tentativo della macchina Python di utilizzare meno memoria per oggetti identici.

```
a = 'short'
b = 'short'
c = 5
d = 5
a is b # True
c is d # True
```

Ma stringhe più lunghe e interi più grandi verranno memorizzati separatamente.

```
a = 'not so short'
b = 'not so short'
```

```
c = 1000
d = 1000
a is b # False
c is d # False
```

È necessario utilizzare `is` per verificare per `None` :

```
if myvar is not None:
    # not None
    pass
if myvar is None:
    # None
    pass
```

L'uso di `is` è di testare un "sentinel" (cioè un oggetto unico).

```
sentinel = object()
def myfunc(var=sentinel):
    if var is sentinel:
        # value wasn't provided
        pass
    else:
        # value was provided
        pass
```

Confronto di oggetti

Per confrontare l'uguaglianza delle classi personalizzate, è possibile sovrascrivere `==` e `!=` Definendo i metodi `__eq__` e `__ne__` . Puoi anche eseguire l'override di `__lt__` (`<`), `__le__` (`<=`), `__gt__` (`>`) e `__ge__` (`>`). Nota che devi solo sostituire due metodi di confronto e Python può gestire il resto (`==` è lo stesso di `not <` e `not >` , ecc.)

```
class Foo(object):
    def __init__(self, item):
        self.my_item = item
    def __eq__(self, other):
        return self.my_item == other.my_item

a = Foo(5)
b = Foo(5)
a == b     # True
a != b     # False
a is b     # False
```

Si noti che questo semplice confronto presuppone che l' `other` (l'oggetto confrontato con) sia lo stesso tipo di oggetto. Il confronto con un altro tipo genera un errore:

```
class Bar(object):
    def __init__(self, item):
        self.other_item = item
    def __eq__(self, other):
        return self.other_item == other.other_item
    def __ne__(self, other):
        return self.other_item != other.other_item
```

```
c = Bar(5)
a == c    # throws AttributeError: 'Foo' object has no attribute 'other_item'
```

Controllare `isinstance()` o simili aiuterà a prevenire questo (se lo si desidera).

Common Gotcha: Python non impone la digitazione

In molti altri linguaggi, se si esegue quanto segue (esempio Java)

```
if("asgdsrf" == 0) {
    //do stuff
}
```

... avrai un errore Non puoi semplicemente confrontare le stringhe con numeri come questo. In Python, questa è una dichiarazione perfettamente legale: si risolverà su `False` .

Un tipico trucchetto è il seguente

```
myVariable = "1"
if 1 == myVariable:
    #do stuff
```

Questo confronto valuterà `False` senza errori, ogni volta, potenzialmente nascondendo un bug o rompere un condizionale.

Leggi confronti online: https://riptutorial.com/it/python/topic/248/confronti

Capitolo 31: Connessione di Python a SQL Server

Examples

Connetti al server, Crea tabella, dati di query

Installa il pacchetto:

```
$ pip install pymssql
```

```
import pymssql

SERVER = "servername"
USER = "username"
PASSWORD = "password"
DATABASE = "dbname"

connection = pymssql.connect(server=SERVER, user=USER,
                password=PASSWORD, database=DATABASE)

cursor = connection.cursor() # to access field as dictionary use cursor(as_dict=True)
cursor.execute("SELECT TOP 1 * FROM TableName")
row = cursor.fetchone()

######## CREATE TABLE ########
cursor.execute("""
CREATE TABLE posts (
    post_id INT PRIMARY KEY NOT NULL,
    message TEXT,
    publish_date DATETIME
)
""")

######## INSERT DATA IN TABLE ########
cursor.execute("""
    INSERT INTO posts VALUES(1, "Hey There", "11.23.2016")
""")
# commit your work to database
connection.commit()

######## ITERATE THROUGH RESULTS  ########
cursor.execute("SELECT TOP 10 * FROM posts ORDER BY publish_date DESC")
for row in cursor:
    print("Message: " + row[1] + " | " + "Date: " + row[2])
    # if you pass as_dict=True to cursor
    # print(row["message"])

connection.close()
```

Puoi fare qualsiasi cosa se il tuo lavoro è legato alle espressioni SQL, basta passare queste espressioni al metodo di esecuzione (operazioni CRUD).

Perché con istruzione, chiamata stored procedure, gestione degli errori o altro controllo di esempio: pymssql.org

Leggi Connessione di Python a SQL Server online: https://riptutorial.com/it/python/topic/7985/connessione-di-python-a-sql-server

Capitolo 32: Conteggio

Examples

Conteggio di tutte le occorrenze di tutti gli articoli in un iterable: collections.Counter

```
from collections import Counter

c = Counter(["a", "b", "c", "d", "a", "b", "a", "c", "d"])
c
# Out: Counter({'a': 3, 'b': 2, 'c': 2, 'd': 2})
c["a"]
# Out: 3

c[7]     # not in the list (7 occurred 0 times!)
# Out: 0
```

Le `collections.Counter` può essere utilizzato per qualsiasi iterable e conta ogni occorrenza per ogni elemento.

Nota : un'eccezione è se un `dict` o altre `collections.Mapping` classe simile a Mapping viene fornita, quindi non le conteggia, ma crea un contatore con questi valori:

```
Counter({"e": 2})
# Out: Counter({"e": 2})

Counter({"e": "e"})        # warning Counter does not verify the values are int
# Out: Counter({"e": "e"})
```

Ottenere il valore più comune (-s): collections.Counter.most_common ()

Il conteggio delle *chiavi* di una `Mapping` non è possibile con `collections.Counter` ma possiamo contare i *valori* :

```
from collections import Counter
adict = {'a': 5, 'b': 3, 'c': 5, 'd': 2, 'e':2, 'q': 5}
Counter(adict.values())
# Out: Counter({2: 2, 3: 1, 5: 3})
```

Gli elementi più comuni sono disponibili con il `most_common` most_common:

```
# Sorting them from most-common to least-common value:
Counter(adict.values()).most_common()
# Out: [(5, 3), (2, 2), (3, 1)]

# Getting the most common value
Counter(adict.values()).most_common(1)
# Out: [(5, 3)]
```

```
# Getting the two most common values
Counter(adict.values()).most_common(2)
# Out: [(5, 3), (2, 2)]
```

Conteggio delle occorrenze di un elemento in una sequenza: list.count () e tuple.count ()

```
alist = [1, 2, 3, 4, 1, 2, 1, 3, 4]
alist.count(1)
# Out: 3

atuple = ('bear', 'weasel', 'bear', 'frog')
atuple.count('bear')
# Out: 2
atuple.count('fox')
# Out: 0
```

Conteggio delle occorrenze di una sottostringa in una stringa: str.count ()

```
astring = 'thisisashorttext'
astring.count('t')
# Out: 4
```

Funziona anche per sottostringhe più lunghe di un carattere:

```
astring.count('th')
# Out: 1
astring.count('is')
# Out: 2
astring.count('text')
# Out: 1
```

che non sarebbe possibile con `collections.Counter` che conta solo caratteri singoli:

```
from collections import Counter
Counter(astring)
# Out: Counter({'a': 1, 'e': 1, 'h': 2, 'i': 2, 'o': 1, 'r': 1, 's': 3, 't': 4, 'x': 1})
```

Conteggio delle occorrenze nell'array numpy

Per contare le occorrenze di un valore in una matrice numpy. Questo funzionerà:

```
>>> import numpy as np
>>> a=np.array([0,3,4,3,5,4,7])
>>> print np.sum(a==3)
2
```

La logica è che l'istruzione booleana produce una matrice in cui tutte le occorrenze dei valori richiesti sono 1 e tutte le altre sono zero. Quindi sommando questi dà il numero di occorrenze. Questo funziona per array di qualsiasi forma o dtype.

Ci sono due metodi che uso per contare le occorrenze di tutti i valori unici in numpy. Unico e bincount. Unico appiattisce automaticamente gli array multidimensionali, mentre bincount funziona solo con gli array 1d contenenti solo numeri interi positivi.

```
>>> unique,counts=np.unique(a,return_counts=True)
>>> print unique,counts # counts[i] is equal to occurrences of unique[i] in a
[0 3 4 5 7] [1 2 2 1 1]
>>> bin_count=np.bincount(a)
>>> print bin_count # bin_count[i] is equal to occurrences of i in a
[1 0 0 2 2 1 0 1]
```

Se i tuoi dati sono array numpy, in genere è molto più veloce usare metodi numpy per convertire i tuoi dati in metodi generici.

Leggi Conteggio online: https://riptutorial.com/it/python/topic/476/conteggio

Capitolo 33: Copia dei dati

Examples

Esecuzione di una copia superficiale

Una copia superficiale è una copia di una raccolta senza eseguire una copia dei suoi elementi.

```
>>> import copy
>>> c = [[1,2]]
>>> d = copy.copy(c)
>>> c is d
False
>>> c[0] is d[0]
True
```

Esecuzione di una copia profonda

Se hai elenchi annidati, è opportuno clonare anche gli elenchi annidati. Questa azione è chiamata copia profonda.

```
>>> import copy
>>> c = [[1,2]]
>>> d = copy.deepcopy(c)
>>> c is d
False
>>> c[0] is d[0]
False
```

Esecuzione di una copia superficiale di un elenco

Puoi creare copie superficiali degli elenchi usando le sezioni.

```
>>> l1 = [1,2,3]
>>> l2 = l1[:]     # Perform the shallow copy.
>>> l2
[1,2,3]
>>> l1 is l2
False
```

Copia un dizionario

Un oggetto dizionario ha la `copy` del metodo. Esegue una copia superficiale del dizionario.

```
>>> d1 = {1:[]}
>>> d2 = d1.copy()
>>> d1 is d2
False
>>> d1[1] is d2[1]
```

```
True
```

Copia un set

Anche i set hanno un metodo di `copy` . È possibile utilizzare questo metodo per eseguire una copia superficiale.

```
>>> s1 = {()}
>>> s2 = s1.copy()
>>> s1 is s2
False
>>> s2.add(3)
>>> s1
{[]}
>>> s2
{3,[]}
```

Leggi Copia dei dati online: https://riptutorial.com/it/python/topic/920/copia-dei-dati

Capitolo 34: Crea un ambiente virtuale con virtualenvwrapper in windows

Examples

Ambiente virtuale con virtualenvwrapper per windows

Supponiamo di dover lavorare su tre diversi progetti: progetto A, progetto B e progetto C. progetto A e progetto B, richiedono python 3 e alcune librerie richieste. Ma per il progetto C hai bisogno di python 2.7 e librerie dipendenti.

Quindi la migliore pratica per questo è separare gli ambienti di progetto. Per creare un ambiente virtuale python separato è necessario seguire i seguenti passaggi:

Passo 1: Installa pip con questo comando: `python -m pip install -U pip`

Passo 2: Quindi installare il pacchetto "virtualenvwrapper-win" utilizzando il comando (il comando può essere eseguito da Windows Power Shell):

```
pip install virtualenvwrapper-win
```

Passaggio 3: creare un nuovo ambiente virtualenv utilizzando il comando: `mkvirtualenv python_3.5`

Passaggio 4: attivare l'ambiente utilizzando il comando:

```
workon < environment name>
```

Comandi principali per virtualenvwrapper:

```
mkvirtualenv <name>
Create a new virtualenv environment named <name>. The environment will be created in
WORKON_HOME.

lsvirtualenv
List all of the enviornments stored in WORKON_HOME.

rmvirtualenv <name>
Remove the environment <name>. Uses folder_delete.bat.

workon [<name>]
If <name> is specified, activate the environment named <name> (change the working virtualenv
to <name>). If a project directory has been defined, we will change into it. If no argument is
specified, list the available environments. One can pass additional option -c after virtualenv
name to cd to virtualenv directory if no projectdir is set.

deactivate
Deactivate the working virtualenv and switch back to the default system Python.

add2virtualenv <full or relative path>
```

```
If a virtualenv environment is active, appends <path> to virtualenv_path_extensions.pth inside
the environment's site-packages, which effectively adds <path> to the environment's
PYTHONPATH. If a virtualenv environment is not active, appends <path> to
virtualenv_path_extensions.pth inside the default Python's site-packages. If <path> doesn't
exist, it will be created.
```

Leggi Crea un ambiente virtuale con virtualenvwrapper in windows online: https://riptutorial.com/it/python/topic/9984/crea-un-ambiente-virtuale-con-virtualenvwrapper-in-windows

Capitolo 35: Creare un servizio Windows usando Python

introduzione

I processi senza testa (senza interfaccia utente) in Windows sono denominati Servizi. Possono essere controllati (avviati, fermati, ecc.) Utilizzando i controlli standard di Windows come la console di comando, Powershell o la scheda Servizi in Task Manager. Un buon esempio potrebbe essere un'applicazione che fornisce servizi di rete, come un'applicazione web o forse un'applicazione di backup che esegue varie attività di archiviazione in background. Esistono diversi modi per creare e installare un'applicazione Python come servizio in Windows.

Examples

Uno script Python che può essere eseguito come servizio

I moduli utilizzati in questo esempio fanno parte di pywin32 (estensioni Python per Windows). A seconda di come hai installato Python, potrebbe essere necessario installarlo separatamente.

```
import win32serviceutil
import win32service
import win32event
import servicemanager
import socket


class AppServerSvc (win32serviceutil.ServiceFramework):
    _svc_name_ = "TestService"
    _svc_display_name_ = "Test Service"

    def __init__(self,args):
        win32serviceutil.ServiceFramework.__init__(self,args)
        self.hWaitStop = win32event.CreateEvent(None,0,0,None)
        socket.setdefaulttimeout(60)

    def SvcStop(self):
        self.ReportServiceStatus(win32service.SERVICE_STOP_PENDING)
        win32event.SetEvent(self.hWaitStop)

    def SvcDoRun(self):
        servicemanager.LogMsg(servicemanager.EVENTLOG_INFORMATION_TYPE,
                              servicemanager.PYS_SERVICE_STARTED,
                              (self._svc_name_,''))
        self.main()

    def main(self):
        pass

if __name__ == '__main__':
    win32serviceutil.HandleCommandLine(AppServerSvc)
```

Questo è solo il boilerplate. Il tuo codice applicazione, probabilmente invocando uno script separato, andrebbe nella funzione main ().

Sarà inoltre necessario installarlo come servizio. La migliore soluzione al momento sembra essere quella di utilizzare il gestore di servizi non sucking . Ciò consente di installare un servizio e fornisce una GUI per configurare la riga di comando che il servizio esegue. Per Python puoi fare questo, che crea il servizio in un colpo solo:

```
nssm install MyServiceName c:\python27\python.exe c:\temp\myscript.py
```

Dove my_script.py è lo script boilerplate sopra, modificato per richiamare lo script o il codice dell'applicazione nella funzione main (). Si noti che il servizio non esegue direttamente lo script Python, esegue l'interprete Python e lo passa lo script principale sulla riga di comando.

In alternativa è possibile utilizzare gli strumenti forniti nel Resource Kit di Windows Server per la versione del sistema operativo in uso, quindi creare il servizio.

Esecuzione di un'applicazione Web Flask come servizio

Questa è una variante dell'esempio generico. Devi solo importare lo script dell'app e invocare il suo metodo `run()` nella funzione `main()` del servizio. In questo caso stiamo anche utilizzando il modulo multiprocessing a causa di un problema durante l'accesso a `WSGIRequestHandler` .

```
import win32serviceutil
import win32service
import win32event
import servicemanager
from multiprocessing import Process

from app import app


class Service(win32serviceutil.ServiceFramework):
    _svc_name_ = "TestService"
    _svc_display_name_ = "Test Service"
    _svc_description_ = "Tests Python service framework by receiving and echoing messages over
a named pipe"

    def __init__(self, *args):
        super().__init__(*args)

    def SvcStop(self):
        self.ReportServiceStatus(win32service.SERVICE_STOP_PENDING)
        self.process.terminate()
        self.ReportServiceStatus(win32service.SERVICE_STOPPED)

    def SvcDoRun(self):
        self.process = Process(target=self.main)
        self.process.start()
        self.process.run()

    def main(self):
        app.run()
```

```
if __name__ == '__main__':
    win32serviceutil.HandleCommandLine(Service)
```

Adattato da http://stackoverflow.com/a/25130524/318488

Leggi Creare un servizio Windows usando Python online:
https://riptutorial.com/it/python/topic/9065/creare-un-servizio-windows-usando-python

Capitolo 36: Creazione di pacchetti Python

Osservazioni

Il progetto di esempio pypa contiene un modello `setup.py` completo e facilmente modificabile che dimostra una vasta gamma di funzionalità che gli strumenti di configurazione hanno da offrire.

Examples

introduzione

Ogni pacchetto richiede un file `setup.py` che descrive il pacchetto.

Considera la seguente struttura di directory per un pacchetto semplice:

```
+-- package_name
|       |
|       +-- __init__.py
|
+-- setup.py
```

`__init__.py` contiene solo la riga `def foo(): return 100` .

Il seguente `setup.py` definirà il pacchetto:

```
from setuptools import setup


setup(
    name='package_name',                    # package name
    version='0.1',                          # version
    description='Package Description',      # short description
    url='http://example.com',               # package URL
    install_requires=[],                    # list of packages this package depends
                                            # on.
    packages=['package_name'],              # List of module names that installing
                                            # this package will provide.
)
```

virtualenv è ottimo per testare le installazioni dei pacchetti senza modificare gli altri ambienti Python:

```
$ virtualenv .virtualenv
...
$ source .virtualenv/bin/activate
$ python setup.py install
running install
...
Installed .../package_name-0.1-....egg
...
```

```
$ python
>>> import package_name
>>> package_name.foo()
100
```

Caricamento su PyPI

Una volta che `setup.py` è completamente funzionante (vedi Introduzione), è molto semplice caricare il pacchetto su PyPI .

Imposta un file .pypirc

Questo file memorizza login e password per autenticare i tuoi account. In genere è memorizzato nella tua directory home.

```
# .pypirc file

[distutils]
index-servers =
  pypi
  pypitest

[pypi]
repository=https://pypi.python.org/pypi
username=your_username
password=your_password

[pypitest]
repository=https://testpypi.python.org/pypi
username=your_username
password=your_password
```

È più sicuro usare lo `twine` per caricare i pacchetti, quindi assicurati che sia installato.

```
$ pip install twine
```

Registrati e carica su testpypi (opzionale)

Nota : PyPI non consente la sovrascrittura dei pacchetti caricati , quindi è prudente testare prima la distribuzione su un server di test dedicato, ad esempio testpypi. Questa opzione sarà discussa. Prendi in considerazione uno schema di controllo delle versioni per il tuo pacchetto prima del caricamento, come la versione del calendario o il controllo delle versioni semantiche .

Accedi o crea un nuovo account su testpypi . La registrazione è richiesta solo la prima volta, anche se la registrazione più di una volta non è dannosa.

```
$ python setup.py register -r pypitest
```

Mentre si trova nella directory principale del pacchetto:

```
$ twine upload dist/* -r pypitest
```

Il tuo pacchetto dovrebbe ora essere accessibile tramite il tuo account.

analisi

Crea un ambiente virtuale di prova. Prova a `pip install` il pacchetto da testpypi o PyPI.

```
# Using virtualenv
$ mkdir testenv
$ cd testenv
$ virtualenv .virtualenv
...
$ source .virtualenv/bin/activate
# Test from testpypi
(.virtualenv)  pip install --verbose --extra-index-url https://testpypi.python.org/pypi
package_name
...
# Or test from PyPI
(.virtualenv) $ pip install package_name
...

(.virtualenv) $ python
Python 3.5.1 (default, Jan 27 2016, 19:16:39)
[GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import package_name
>>> package_name.foo()
100
```

In caso di successo, il pacchetto è meno impor- tante. Potresti considerare di testare la tua API anche prima del tuo caricamento finale su PyPI. Se il pacchetto non è riuscito durante il test, non preoccuparti. Puoi ancora aggiustarlo, ricaricarlo su testpypi e provare di nuovo.

Registrati e carica su PyPI

Assicurarsi che sia stato installato il `twine` :

```
$ pip install twine
```

Accedere o creare un nuovo account su PyPI .

```
$ python setup.py register -r pypi
$ twine upload dist/*
```

Questo è tutto! Il tuo pacco è ora in diretta .

Se scopri un bug, carica semplicemente una nuova versione del tuo pacchetto.

Documentazione

Non dimenticare di includere almeno una sorta di documentazione per il tuo pacchetto. PyPi prende come lingua di formattazione predefinita reStructuredText .

Leggimi

Se il tuo pacchetto non ha una grande documentazione, includi cosa può aiutare gli altri utenti nel file `README.rst` . Quando il file è pronto, è necessario un altro per dire a PyPi di mostrarlo.

Crea il file `setup.cfg` e inserisci queste due righe:

```
[metadata]
description-file = README.rst
```

Nota che se provi a mettere il file Markdown nel tuo pacchetto, PyPi lo leggerà come un puro file di testo senza alcuna formattazione.

Licenze

Spesso è più che benvenuto inserire un file `LICENSE.txt` nel pacchetto con una delle licenze OpenSource per dire agli utenti se possono usare il tuo pacchetto, ad esempio in progetti commerciali o se il tuo codice è utilizzabile con la loro licenza.

In modo più leggibile, alcune licenze sono spiegate in TL; DR .

Rendere eseguibile il pacchetto

Se il tuo pacchetto non è solo una libreria, ma ha un pezzo di codice che può essere usato come vetrina o come applicazione standalone quando il tuo pacchetto è installato, metti quel pezzo di codice nel file `__main__.py` .

Mettere il `__main__.py` nella `package_name` cartella. In questo modo sarai in grado di eseguirlo direttamente dalla console:

```
python -m package_name
```

Se non è disponibile `__main__.py` file `__main__.py` , il pacchetto non verrà eseguito con questo comando e questo errore verrà stampato:

> python: nessun modulo chiamato `package_name.__main__`; 'package_name' è un pacchetto e non può essere eseguito direttamente.

Leggi Creazione di pacchetti Python online: https://riptutorial.com/it/python/topic/1381/creazione-di-pacchetti-python

Capitolo 37: ctypes

introduzione

`ctypes` è una libreria incorporata in Python che richiama le funzioni esportate da librerie compilate native.

Nota: poiché questa libreria gestisce il codice compilato, è relativamente dipendente dal SO.

Examples

Utilizzo di base

Diciamo che vogliamo usare la funzione `ntohl libc` .

Per prima cosa, dobbiamo caricare `libc.so` :

```
>>> from ctypes import *
>>> libc = cdll.LoadLibrary('libc.so.6')
>>> libc
<CDLL 'libc.so.6', handle baadf00d at 0xdeadbeef>
```

Quindi, otteniamo l'oggetto funzione:

```
>>> ntohl = libc.ntohl
>>> ntohl
<_FuncPtr object at 0xbaadf00d>
```

E ora, possiamo semplicemente invocare la funzione:

```
>>> ntohl(0x6C)
1811939328
>>> hex(_)
'0x6c000000'
```

Che fa esattamente quello che ci aspettiamo che faccia.

Insidie comuni

Impossibile caricare un file

Il primo possibile errore non riesce a caricare la libreria. In tal caso viene sollevato di solito un OSError.

Questo perché il file non esiste (o non può essere trovato dal sistema operativo):

```
>>> cdll.LoadLibrary("foobar.so")
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "/usr/lib/python3.5/ctypes/__init__.py", line 425, in LoadLibrary
    return self._dlltype(name)
File "/usr/lib/python3.5/ctypes/__init__.py", line 347, in __init__
    self._handle = _dlopen(self._name, mode)
OSError: foobar.so: cannot open shared object file: No such file or directory
```

Come puoi vedere, l'errore è chiaro e piuttosto indicativo.

La seconda ragione è che il file è stato trovato, ma non è del formato corretto.

```
>>> cdll.LoadLibrary("libc.so")
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "/usr/lib/python3.5/ctypes/__init__.py", line 425, in LoadLibrary
    return self._dlltype(name)
File "/usr/lib/python3.5/ctypes/__init__.py", line 347, in __init__
    self._handle = _dlopen(self._name, mode)
OSError: /usr/lib/i386-linux-gnu/libc.so: invalid ELF header
```

In questo caso, il file è un file di script e non un file `.so` . Ciò potrebbe anche accadere quando si tenta di aprire un file `.dll` su una macchina Linux o un file 64bit su un interprete python a 32 bit. Come puoi vedere, in questo caso l'errore è un po 'più vago e richiede alcuni scavi.

Impossibile accedere a una funzione

Supponendo di aver caricato correttamente il file `.so` , dobbiamo quindi accedere alla nostra funzione come abbiamo fatto nel primo esempio.

Quando viene utilizzata una funzione non esistente, viene sollevato un `AttributeError` :

```
>>> libc.foo
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "/usr/lib/python3.5/ctypes/__init__.py", line 360, in __getattr__
    func = self.__getitem__(name)
File "/usr/lib/python3.5/ctypes/__init__.py", line 365, in __getitem__
    func = self._FuncPtr((name_or_ordinal, self))
AttributeError: /lib/i386-linux-gnu/libc.so.6: undefined symbol: foo
```

Oggetto ctypes di base

L'oggetto più elementare è un int:

```
>>> obj = ctypes.c_int(12)
>>> obj
c_long(12)
```

Ora, `obj` riferisce a un blocco di memoria contenente il valore 12.

È possibile accedere direttamente a questo valore e persino modificarlo:

```
>>> obj.value
12
>>> obj.value = 13
>>> obj
c_long(13)
```

Poiché `obj` riferisce a un blocco di memoria, possiamo anche scoprire la sua dimensione e posizione:

```
>>> sizeof(obj)
4
>>> hex(addressof(obj))
'0xdeadbeef'
```

array di tipi

Come ogni buon programmatore C sa, un singolo valore non ti porterà lontano. Ciò che ci farà davvero andare avanti sono gli array!

```
>>> c_int * 16
<class '__main__.c_long_Array_16'>
```

Questo non è un array reale, ma è dannatamente vicino! Abbiamo creato una classe che denota un array di 16 `int` s.

Ora tutto ciò che dobbiamo fare è inizializzarlo:

```
>>> arr = (c_int * 16)(*range(16))
>>> arr
<__main__.c_long_Array_16 object at 0xbaddcafe>
```

Ora `arr` è un array reale che contiene i numeri da 0 a 15.

Possono essere consultati come qualsiasi elenco:

```
>>> arr[5]
5
>>> arr[5] = 20
>>> arr[5]
20
```

E proprio come qualsiasi altro oggetto `ctypes` , ha anche una dimensione e una posizione:

```
>>> sizeof(arr)
64 # sizeof(c_int) * 16
>>> hex(addressof(arr))
'0xc000l0ff'
```

Funzioni di avvolgimento per i tipi

In alcuni casi, una funzione C accetta un puntatore a funzione. Come utenti accaniti di `ctypes` , vorremmo usare quelle funzioni e persino passare la funzione python come argomenti.

Definiamo una funzione:

```
>>> def max(x, y):
        return x if x >= y else y
```

Ora, quella funzione accetta due argomenti e restituisce un risultato dello stesso tipo. Per l'esempio, supponiamo che type sia un int.

Come abbiamo fatto sull'esempio dell'array, possiamo definire un oggetto che denota quel prototipo:

```
>>> CFUNCTYPE(c_int, c_int, c_int)
<CFunctionType object at 0xdeadbeef>
```

Quel prototipo denota una funzione che restituisce un `c_int` (il primo argomento) e accetta due argomenti `c_int` (gli altri argomenti).

Ora avvolgiamo la funzione:

```
>>> CFUNCTYPE(c_int, c_int, c_int)(max)
<CFunctionType object at 0xdeadbeef>
```

Prototipi di funzione hanno in più l'utilizzo: possono avvolgere `ctypes` funzione (come `libc.ntohl`) e verificare che gli argomenti corretti vengono utilizzati quando si richiama la funzione.

```
>>> libc.ntohl() # garbage in - garbage out
>>> CFUNCTYPE(c_int, c_int)(libc.ntohl)()
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: this function takes at least 1 argument (0 given)
```

Utilizzo complesso

Combiniamo tutti gli esempi sopra in uno scenario complesso: usando la funzione `lfind` `libc` .

Per maggiori dettagli sulla funzione, leggi la pagina man . Vi esorto a leggerlo prima di andare avanti.

Innanzitutto, definiremo i prototipi corretti:

```
>>> compar_proto = CFUNCTYPE(c_int, POINTER(c_int), POINTER(c_int))
>>> lfind_proto = CFUNCTYPE(c_void_p, c_void_p, c_void_p, POINTER(c_uint), c_uint,
compar_proto)
```

Quindi, creiamo le variabili:

```
>>> key = c_int(12)
>>> arr = (c_int * 16)(*range(16))
>>> nmemb = c_uint(16)
```

E ora definiamo la funzione di confronto:

```
>>> def compar(x, y):
        return x.contents.value - y.contents.value
```

Si noti che `x y` sono `POINTER(c_int)` , quindi è necessario dereferenziarli e prendere i loro valori per confrontare effettivamente il valore memorizzato nella memoria.

Ora possiamo combinare tutto insieme:

```
>>> lfind = lfind_proto(libc.lfind)
>>> ptr = lfind(byref(key), byref(arr), byref(nmemb), sizeof(c_int), compar_proto(compar))
```

`ptr` è il puntatore vuoto restituito. Se la `key` non è stata trovata in `arr` , il valore sarebbe `None` , ma in questo caso abbiamo ottenuto un valore valido.

Ora possiamo convertirlo e accedere al valore:

```
>>> cast(ptr, POINTER(c_int)).contents
c_long(12)
```

Inoltre, possiamo vedere che `ptr` punta al valore corretto all'interno di `arr` :

```
>>> addressof(arr) + 12 * sizeof(c_int) == ptr
True
```

Leggi ctypes online: https://riptutorial.com/it/python/topic/9050/ctypes

Capitolo 38: Cuscino

Examples

Leggi il file immagine

```
from PIL import Image

im = Image.open("Image.bmp")
```

Converti file in JPEG

```
from __future__ import print_function
import os, sys
from PIL import Image

for infile in sys.argv[1:]:
    f, e = os.path.splitext(infile)
    outfile = f + ".jpg"
    if infile != outfile:
        try:
            Image.open(infile).save(outfile)
        except IOError:
            print("cannot convert", infile)
```

Leggi Cuscino online: https://riptutorial.com/it/python/topic/6841/cuscino

Capitolo 39: Data e ora

Osservazioni

Python fornisce sia metodi incorporati che librerie esterne per creare, modificare, analizzare e manipolare date e ore.

Examples

Analisi di una stringa in un oggetto datetime timezone aware

Python 3.2+ supporta il formato `%z` quando analizza una stringa in un oggetto `datetime` .

> Scostamento UTC nel formato `+HHMM` o `-HHMM` (stringa vuota se l'oggetto è ingenuo).

Python 3.x 3.2

```
import datetime
dt = datetime.datetime.strptime("2016-04-15T08:27:18-0500", "%Y-%m-%dT%H:%M:%S%z")
```

Per altre versioni di Python, è possibile utilizzare una libreria esterna come `dateutil` , che rende `dateutil` analisi di una stringa con fuso orario in un oggetto `datetime` .

```
import dateutil.parser
dt = dateutil.parser.parse("2016-04-15T08:27:18-0500")
```

La variabile `dt` è ora un oggetto `datetime` con il seguente valore:

```
datetime.datetime(2016, 4, 15, 8, 27, 18, tzinfo=tzoffset(None, -18000))
```

Data semplice aritmetica

Le date non esistono in isolamento. È normale che tu debba trovare la quantità di tempo tra le date o determinare quale sarà la data di domani. Questo può essere realizzato usando oggetti `timedelta`

```
import datetime

today = datetime.date.today()
print('Today:', today)

yesterday = today - datetime.timedelta(days=1)
print('Yesterday:', yesterday)

tomorrow = today + datetime.timedelta(days=1)
print('Tomorrow:', tomorrow)
```

```
print('Time between tomorrow and yesterday:', tomorrow - yesterday)
```

Ciò produrrà risultati simili a:

```
Today: 2016-04-15
Yesterday: 2016-04-14
Tomorrow: 2016-04-16
Difference between tomorrow and yesterday: 2 days, 0:00:00
```

Utilizzo di oggetti datetime di base

Il modulo datetime contiene tre tipi principali di oggetti: data, ora e data / ora.

```
import datetime

# Date object
today = datetime.date.today()
new_year = datetime.date(2017, 01, 01) #datetime.date(2017, 1, 1)

# Time object
noon = datetime.time(12, 0, 0) #datetime.time(12, 0)

# Current datetime
now = datetime.datetime.now()

# Datetime object
millenium_turn = datetime.datetime(2000, 1, 1, 0, 0, 0) #datetime.datetime(2000, 1, 1, 0, 0)
```

Le operazioni aritmetiche per questi oggetti sono supportate solo nello stesso tipo di dati e l'esecuzione di operazioni aritmetiche semplici con istanze di tipi diversi comporterà un'eccezione TypeError.

```
# subtraction of noon from today
noon-today
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for -: 'datetime.time' and 'datetime.date'
However, it is straightforward to convert between types.

# Do this instead
print('Time since the millenium at midnight: ',
      datetime.datetime(today.year, today.month, today.day) - millenium_turn)

# Or this
print('Time since the millenium at noon: ',
      datetime.datetime.combine(today, noon) - millenium_turn)
```

Scorri le date

A volte si desidera eseguire un'iterazione su un intervallo di date da una data di inizio a una data di fine. Puoi farlo usando la libreria `datetime` e `timedelta` oggetto `timedelta` :

```
import datetime
```

```
# The size of each step in days
day_delta = datetime.timedelta(days=1)

start_date = datetime.date.today()
end_date = start_date + 7*day_delta

for i in range((end_date - start_date).days):
    print(start_date + i*day_delta)
```

Che produce:

```
2016-07-21
2016-07-22
2016-07-23
2016-07-24
2016-07-25
2016-07-26
2016-07-27
```

Analisi di una stringa con un nome breve fuso orario in un oggetto datetime timezone aware

Utilizzando la libreria `dateutil` come nell'esempio precedente `dateutil` timestamp sensibili al fuso orario , è anche possibile analizzare i timestamp con un nome di fuso orario "breve" specificato.

Per le date formattate con i nomi di zona breve tempo o abbreviazioni, che sono generalmente ambigua (ad esempio CST, che potrebbe essere centrale Standard Time, Cina Standard Time, Cuba Standard Time, ecc - più può essere trovato qui) o non necessariamente disponibili in un database standard , è necessario specificare una mappatura tra l'abbreviazione del fuso orario e l'oggetto `tzinfo` .

```
from dateutil import tz
from dateutil.parser import parse

ET = tz.gettz('US/Eastern')
CT = tz.gettz('US/Central')
MT = tz.gettz('US/Mountain')
PT = tz.gettz('US/Pacific')

us_tzinfos = {'CST': CT, 'CDT': CT,
              'EST': ET, 'EDT': ET,
              'MST': MT, 'MDT': MT,
              'PST': PT, 'PDT': PT}

dt_est = parse('2014-01-02 04:00:00 EST', tzinfos=us_tzinfos)
dt_pst = parse('2016-03-11 16:00:00 PST', tzinfos=us_tzinfos)
```

Dopo aver eseguito questo:

```
dt_est
# datetime.datetime(2014, 1, 2, 4, 0, tzinfo=tzfile('/usr/share/zoneinfo/US/Eastern'))
dt_pst
# datetime.datetime(2016, 3, 11, 16, 0, tzinfo=tzfile('/usr/share/zoneinfo/US/Pacific'))
```

Vale la pena notare che se si utilizza un `pytz` fuso orario con questo metodo, *non* sarà adeguatamente localizzato:

```
from dateutil.parser import parse
import pytz

EST = pytz.timezone('America/New_York')
dt = parse('2014-02-03 09:17:00 EST', tzinfos={'EST': EST})
```

Questo semplicemente associa il `pytz` orario `pytz` al datetime:

```
dt.tzinfo # Will be in Local Mean Time!
# <DstTzInfo 'America/New_York' LMT-1 day, 19:04:00 STD>
```

Se si utilizza questo metodo, è consigliabile `localize` la parte ingenua del datetime dopo l'analisi:

```
dt_fixed = dt.tzinfo.localize(dt.replace(tzinfo=None))
dt_fixed.tzinfo # Now it's EST.
# <DstTzInfo 'America/New_York' EST-1 day, 19:00:00 STD>)
```

Costruzione di datetize sensibili al fuso orario

Di default tutti gli oggetti `datetime` sono ingenui. Per renderli sensibili al fuso orario, è necessario allegare un oggetto `tzinfo` , che fornisce l'offset UTC e l'abbreviazione del fuso orario in funzione della data e dell'ora.

Fissi i fusi orari di offset

Per i fusi orari che sono un offset fisso da UTC, in Python 3.2+, il modulo `datetime` fornisce la classe `timezone` , un'implementazione concreta di `tzinfo` , che accetta un parametro nome `timedelta` e (facoltativo):

Python 3.x 3.2

```
from datetime import datetime, timedelta, timezone
JST = timezone(timedelta(hours=+9))

dt = datetime(2015, 1, 1, 12, 0, 0, tzinfo=JST)
print(dt)
# 2015-01-01 12:00:00+09:00

print(dt.tzname())
# UTC+09:00

dt = datetime(2015, 1, 1, 12, 0, 0, tzinfo=timezone(timedelta(hours=9), 'JST'))
print(dt.tzname)
# 'JST'
```

Per le versioni di Python precedenti alla 3.2, è necessario utilizzare una libreria di terze parti, come `dateutil` . `dateutil` fornisce una classe equivalente, `tzoffset` , che (a partire dalla versione 2.5.3) prende gli argomenti del modulo `dateutil.tz.tzoffset(tzname, offset)` , dove l' `offset` è specificato in secondi:

Python 3.x 3.2
Python 2.x 2.7

```
from datetime import datetime, timedelta
from dateutil import tz

JST = tz.tzoffset('JST', 9 * 3600) # 3600 seconds per hour
dt = datetime(2015, 1, 1, 12, 0, tzinfo=JST)
print(dt)
# 2015-01-01 12:00:00+09:00
print(dt.tzname)
# 'JST'
```

Zone con ora legale

Per le zone con ora legale, le librerie standard Python non forniscono una classe standard, quindi è necessario utilizzare una libreria di terze parti. `pytz` e `dateutil` sono librerie popolari che forniscono classi di fuso orario.

Oltre ai fusi orari statici, `dateutil` fornisce classi di fuso orario che utilizzano l'ora legale (consultare la documentazione del modulo `tz`). È possibile utilizzare il metodo `tz.gettz()` per ottenere un oggetto del fuso orario, che può quindi essere passato direttamente al costruttore `datetime` :

```
from datetime import datetime
from dateutil import tz
local = tz.gettz() # Local time
PT = tz.gettz('US/Pacific') # Pacific time

dt_l = datetime(2015, 1, 1, 12, tzinfo=local) # I am in EST
dt_pst = datetime(2015, 1, 1, 12, tzinfo=PT)
dt_pdt = datetime(2015, 7, 1, 12, tzinfo=PT) # DST is handled automatically
print(dt_l)
# 2015-01-01 12:00:00-05:00
print(dt_pst)
# 2015-01-01 12:00:00-08:00
print(dt_pdt)
# 2015-07-01 12:00:00-07:00
```

ATTENZIONE : A partire dalla versione 2.5.3, `dateutil` non gestisce correttamente i datetimes ambigui e sarà sempre predefinito alla data *successiva* . Non v'è alcun modo per costruire un oggetto con un `dateutil` fuso orario che rappresenta, per esempio `2015-11-01 1:30 EDT-4` , poiché questo è *in* un periodo di transizione legale.

Tutti i casi limite sono gestiti correttamente quando si utilizza `pytz` , ma i `pytz` orari `pytz` *non* devono essere collegati direttamente ai fusi orari tramite il costruttore. Invece, un `pytz` orario di `pytz` dovrebbe essere collegato usando il metodo di `localize` del fuso orario:

```
from datetime import datetime, timedelta
import pytz

PT = pytz.timezone('US/Pacific')
dt_pst = PT.localize(datetime(2015, 1, 1, 12))
dt_pdt = PT.localize(datetime(2015, 11, 1, 0, 30))
print(dt_pst)
# 2015-01-01 12:00:00-08:00
```

```
print(dt_pdt)
# 2015-11-01 00:30:00-07:00
```

Tenere presente che se si esegue l'aritmetica data / ora su un fuso orario `pytz` , è necessario eseguire i calcoli in UTC (se si desidera il tempo trascorso assoluto) oppure è necessario chiamare `normalize()` sul risultato:

```
dt_new = dt_pdt + timedelta(hours=3) # This should be 2:30 AM PST
print(dt_new)
# 2015-11-01 03:30:00-07:00
dt_corrected = PT.normalize(dt_new)
print(dt_corrected)
# 2015-11-01 02:30:00-08:00
```

Parodia fuzzy parsing (estraendo datetime da un testo)

È possibile estrarre una data da un testo utilizzando l' analizzatore `dateutil` in modalità "fuzzy", in cui i componenti della stringa non riconosciuti come parte di una data vengono ignorati.

```
from dateutil.parser import parse

dt = parse("Today is January 1, 2047 at 8:21:00AM", fuzzy=True)
print(dt)
```

`dt` ora è un *oggetto* `datetime` e `datetime.datetime(2047, 1, 1, 8, 21)` stampato.

Passaggio da un fuso orario all'altro

Per passare da un fuso orario all'altro, sono necessari oggetti datetime che sono sensibili al fuso orario.

```
from datetime import datetime
from dateutil import tz

utc = tz.tzutc()
local = tz.tzlocal()

utc_now = datetime.utcnow()
utc_now # Not timezone-aware.

utc_now = utc_now.replace(tzinfo=utc)
utc_now # Timezone-aware.

local_now = utc_now.astimezone(local)
local_now # Converted to local time.
```

Analisi di un timestamp arbitrario ISO 8601 con librerie minime

Python ha solo un supporto limitato per l'analisi di timestamp ISO 8601. Per `strptime` riguarda la `strptime` è necessario sapere esattamente in che formato si trova. Come complicazione, la stringificazione di un `datetime` è un timestamp ISO 8601, con spazio come separatore e frazione di

6 cifre:

```
str(datetime.datetime(2016, 7, 22, 9, 25, 59, 555555))
# '2016-07-22 09:25:59.555555'
```

ma se la frazione è 0, non viene prodotta alcuna parte frazionaria

```
str(datetime.datetime(2016, 7, 22, 9, 25, 59, 0))
# '2016-07-22 09:25:59'
```

Ma queste 2 forme hanno bisogno di un formato *diverso* per la `strptime` . Inoltre, `strptime' does not support at all parsing minute timezones that have a : in it, thus` 2016-07-22 09: 25: 59 + 0300 `can be parsed, but the standard format` 2016-07-22 09:25:59 +03: 00` non può.

Esiste una libreria a file singolo chiamata `iso8601` che analizza correttamente i timestamp ISO 8601 e solo loro.

Supporta frazioni e fusi orari e il separatore `T` tutto con un'unica funzione:

```
import iso8601
iso8601.parse_date('2016-07-22 09:25:59')
# datetime.datetime(2016, 7, 22, 9, 25, 59, tzinfo=<iso8601.Utc>)
iso8601.parse_date('2016-07-22 09:25:59+03:00')
# datetime.datetime(2016, 7, 22, 9, 25, 59, tzinfo=<FixedOffset '+03:00' ...>)
iso8601.parse_date('2016-07-22 09:25:59Z')
# datetime.datetime(2016, 7, 22, 9, 25, 59, tzinfo=<iso8601.Utc>)
iso8601.parse_date('2016-07-22T09:25:59.000111+03:00')
# datetime.datetime(2016, 7, 22, 9, 25, 59, 111, tzinfo=<FixedOffset '+03:00' ...>)
```

Se non è impostato un fuso orario, `iso8601.parse_date` valore predefinito UTC. La zona predefinita può essere modificata con l'argomento della parola chiave `default_zone` . In particolare, se questo è `None` invece del valore predefinito, allora quei timestamp che non hanno un fuso orario esplicito vengono restituiti come dati naive invece:

```
iso8601.parse_date('2016-07-22T09:25:59', default_timezone=None)
# datetime.datetime(2016, 7, 22, 9, 25, 59)
iso8601.parse_date('2016-07-22T09:25:59Z', default_timezone=None)
# datetime.datetime(2016, 7, 22, 9, 25, 59, tzinfo=<iso8601.Utc>)
```

Conversione del timestamp in data / ora

Il modulo `datetime` può convertire un `timestamp` POSIX in un oggetto `datetime` ITC.

The Epoch è il 1 gennaio 1970 a mezzanotte.

```
import time
from datetime import datetime
seconds_since_epoch=time.time()  #1469182681.709

utc_date=datetime.utcfromtimestamp(seconds_since_epoch) #datetime.datetime(2016, 7, 22, 10,
18, 1, 709000)
```

Sottraendo con precisione mesi da una data

Utilizzando il modulo del `calendar`

```
import calendar
from datetime import date

def monthdelta(date, delta):
    m, y = (date.month+delta) % 12, date.year + ((date.month)+delta-1) // 12
    if not m: m = 12
    d = min(date.day, calendar.monthrange(y, m)[1])
    return date.replace(day=d,month=m, year=y)

next_month = monthdelta(date.today(), 1) #datetime.date(2016, 10, 23)
```

Utilizzando il modulo `dateutils`

```
import datetime
import dateutil.relativedelta

d = datetime.datetime.strptime("2013-03-31", "%Y-%m-%d")
d2 = d - dateutil.relativedelta.relativedelta(months=1)  #datetime.datetime(2013, 2, 28, 0, 0)
```

Calcolo delle differenze di orario

il modulo `timedelta` è utile per calcolare le differenze tra i tempi:

```
from datetime import datetime, timedelta
now = datetime.now()
then = datetime(2016, 5, 23)     # datetime.datetime(2016, 05, 23, 0, 0, 0)
```

La specifica del tempo è facoltativa quando si crea un nuovo oggetto `datetime`

```
delta = now-then
```

`delta` è di tipo `timedelta`

```
print(delta.days)
# 60
print(delta.seconds)
# 40826
```

Per ottenere il giorno dopo e il giorno prima della data, potremmo usare:

n giorno dopo data:

```
def get_n_days_after_date(date_format="%d %B %Y", add_days=120):

    date_n_days_after = datetime.datetime.now() + timedelta(days=add_days)
    return date_n_days_after.strftime(date_format)
```

n giorno prima della data:

```
def get_n_days_before_date(self, date_format="%d %B %Y", days_before=120):

       date_n_days_ago = datetime.datetime.now() - timedelta(days=days_before)
       return date_n_days_ago.strftime(date_format)
```

Ottieni un timestamp ISO 8601

Senza fuso orario, con microsecondi

```
from datetime import datetime

datetime.now().isoformat()
# Out: '2016-07-31T23:08:20.886783'
```

Con il fuso orario, con microsecondi

```
from datetime import datetime
from dateutil.tz import tzlocal

datetime.now(tzlocal()).isoformat()
# Out: '2016-07-31T23:09:43.535074-07:00'
```

Con il fuso orario, senza microsecondi

```
from datetime import datetime
from dateutil.tz import tzlocal

datetime.now(tzlocal()).replace(microsecond=0).isoformat()
# Out: '2016-07-31T23:10:30-07:00'
```

Vedere ISO 8601 per ulteriori informazioni sul formato ISO 8601.

Leggi Data e ora online: https://riptutorial.com/it/python/topic/484/data-e-ora

Capitolo 40: Dati binari

Sintassi

- pack (fmt, v1, v2, ...)
- scompattare (fmt, buffer)

Examples

Formattare un elenco di valori in un oggetto byte

```
from struct import pack

print(pack('I3c', 123, b'a', b'b', b'c'))  # b'{\x00\x00\x00abc'
```

Disimballare un oggetto byte in base a una stringa di formato

```
from struct import unpack

print(unpack('I3c', b'{\x00\x00\x00abc'))  # (123, b'a', b'b', b'c')
```

Imballaggio di una struttura

Il modulo " **struct** " fornisce la possibilità di impacchettare oggetti python come blocchi contigui di byte o di dissimulare un blocco di byte in strutture python.

La funzione pack accetta una stringa di formato e uno o più argomenti e restituisce una stringa binaria. Questo sembra molto simile alla formattazione di una stringa, tranne per il fatto che l'output non è una stringa ma una porzione di byte.

```
import struct
import sys
print "Native byteorder: ", sys.byteorder
# If no byteorder is specified, native byteorder is used
buffer = struct.pack("ihb", 3, 4, 5)
print "Byte chunk: ", repr(buffer)
print "Byte chunk unpacked: ", struct.unpack("ihb", buffer)
# Last element as unsigned short instead of unsigned char ( 2 Bytes)
buffer = struct.pack("ihh", 3, 4, 5)
print "Byte chunk: ", repr(buffer)
```

Produzione:

> Byte byte nativo: piccolo byte Byte: '\ x03 \ x00 \ x00 \ x00 \ x04 \ x00 \ x05' Blocco byte scompattato: (3, 4, 5) Bit byte: '\ x03 \ x00 \ x00 \ x00 \ x04 \ x04 \ x00 \ x05 \ x00'

È possibile utilizzare l'ordine dei byte di rete con i dati ricevuti dalla rete o dai dati del pacchetto

per inviarlo alla rete.

```
import struct
# If no byteorder is specified, native byteorder is used
buffer = struct.pack("hhh", 3, 4, 5)
print "Byte chunk native byte order: ", repr(buffer)
buffer = struct.pack("!hhh", 3, 4, 5)
print "Byte chunk network byte order: ", repr(buffer)
```

Produzione:

> Ordine byte byte del byte byte: '\ x03 \ x00 \ x04 \ x00 \ x05 \ x00'
>
> Ordine dei byte della rete a byte byte: '\ x00 \ x03 \ x00 \ x04 \ x00 \ x05'

È possibile ottimizzare evitando il sovraccarico di allocare un nuovo buffer fornendo un buffer creato in precedenza.

```
import struct
from ctypes import create_string_buffer
bufferVar = create_string_buffer(8)
bufferVar2 = create_string_buffer(8)
# We use a buffer that has already been created
# provide format, buffer, offset and data
struct.pack_into("hhh", bufferVar, 0, 3, 4, 5)
print "Byte chunk: ", repr(bufferVar.raw)
struct.pack_into("hhh", bufferVar2, 2, 3, 4, 5)
print "Byte chunk: ", repr(bufferVar2.raw)
```

Produzione:

> Blocco byte: '\ x03 \ x00 \ x04 \ x00 \ x05 \ x00 \ x00 \ x00'
>
> Blocco byte: '\ x00 \ x00 \ x03 \ x00 \ x04 \ x00 \ x05 \ x00'

Leggi Dati binari online: https://riptutorial.com/it/python/topic/2978/dati-binari

Capitolo 41: Debug

Examples

The Python Debugger: debug passo dopo passo con _pdb_

La libreria standard Python include una libreria di debug interattiva chiamata *pdb* . *pdb* ha capacità estese, il più comunemente usato è la capacità di "passare attraverso" un programma.

Per accedere immediatamente al debugging step-through, utilizzare:

```
python -m pdb <my_file.py>
```

Questo avvierà il debugger nella prima riga del programma.

Di solito vorrete indirizzare una sezione specifica del codice per il debug. Per fare ciò importiamo la libreria *pdb* e usiamo *set_trace ()* per interrompere il flusso di questo codice di esempio in difficoltà.

```
import pdb

def divide(a, b):
    pdb.set_trace()
    return a/b
    # What's wrong with this? Hint: 2 != 3

print divide(1, 2)
```

L'esecuzione di questo programma avvierà il debugger interattivo.

```
python foo.py
> ~/scratch/foo.py(5)divide()
-> return a/b
(Pdb)
```

Spesso questo comando viene utilizzato su una riga in modo che possa essere commentato con un singolo carattere #

```
 import pdf; pdb.set_trace()
```

Al prompt *(Pdb)* possono essere inseriti i comandi. Questi comandi possono essere comandi di debugger o python. Per stampare variabili possiamo usare *p* dal debugger, o la *stampa* di python.

```
(Pdb) p a
1
(Pdb) print a
1
```

Per vedere l'elenco di tutte le variabili locali utilizzare

```
locals
```

funzione incorporata

Questi sono buoni comandi di debug per sapere:

```
b <n> | <f>: set breakpoint at line *n* or function named *f*.
# b 3
# b divide
b: show all breakpoints.
c: continue until the next breakpoint.
s: step through this line (will enter a function).
n: step over this line (jumps over a function).
r: continue until the current function returns.
l: list a window of code around this line.
p <var>: print variable named *var*.
# p x
q: quit debugger.
bt: print the traceback of the current execution call stack
up: move your scope up the function call stack to the caller of the current function
down: Move your scope back down the function call stack one level
step: Run the program until the next line of execution in the program, then return control
back to the debugger
next: run the program until the next line of execution in the current function, then return
control back to the debugger
return: run the program until the current function returns, then return control back to the
debugger
continue: continue running the program until the next breakpoint (or set_trace si called
again)
```

Il debugger può anche valutare Python in modo interattivo:

```
-> return a/b
(Pdb) p a+b
3
(Pdb) [ str(m) for m in [a,b]]
['1', '2']
(Pdb) [ d for d in xrange(5)]
[0, 1, 2, 3, 4]
```

Nota:

Se uno qualsiasi dei nomi delle variabili coincide con i comandi del debugger, usa un punto esclamativo ' **!** 'prima della var fare esplicitamente riferimento alla variabile e non al comando debugger. Ad esempio, spesso potrebbe accadere che tu usi il nome variabile ' **c** ' per un contatore e potresti volerlo stampare nel debugger. un semplice comando ' **c** ' continuerà l'esecuzione fino al successivo punto di interruzione. Usa invece ' **! C** ' per stampare il valore della variabile come segue:

```
(Pdb) !c
4
```

Via IPython e ipdb

Se sono installati IPython (o Jupyter), il debugger può essere richiamato usando:

```
import ipdb
ipdb.set_trace()
```

Al raggiungimento, il codice uscirà e stamperà:

```
 /home/usr/ook.py(3)<module>()
      1 import ipdb
      2 ipdb.set_trace()
----> 3 print("Hello world!")

ipdb>
```

Chiaramente, questo significa che uno deve modificare il codice. C'è un modo più semplice:

```
from IPython.core import ultratb
sys.excepthook = ultratb.FormattedTB(mode='Verbose',
                                     color_scheme='Linux',
                                     call_pdb=1)
```

Ciò causerà la chiamata del debugger se è stata sollevata un'eccezione non rilevata.

Debugger remoto

Alcune volte è necessario eseguire il debug del codice Python che viene eseguito da un altro processo e, in questo caso, `rpdb` è utile.

> rpdb è un wrapper su pdb che reindirizza stdin e stdout a un gestore di socket. Di default apre il debugger sulla porta 4444

Uso:

```
# In the Python file you want to debug.
import rpdb
rpdb.set_trace()
```

E quindi è necessario eseguire questo nel terminale per connettersi a questo processo.

```
# Call in a terminal to see the output
$ nc 127.0.0.1 4444
```

E riceverai il pdb promt

```
> /home/usr/ook.py(3)<module>()
-> print("Hello world!")
(Pdb)
```

Leggi Debug online: https://riptutorial.com/it/python/topic/2077/debug

Capitolo 42: decoratori

introduzione

Le funzioni del decoratore sono schemi di progettazione del software. Modificano dinamicamente la funzionalità di una funzione, metodo o classe senza dover utilizzare direttamente le sottoclassi o modificare il codice sorgente della funzione decorata. Se utilizzati correttamente, i decoratori possono diventare strumenti potenti nel processo di sviluppo. Questo argomento riguarda l'implementazione e le applicazioni delle funzioni di decoratore in Python.

Sintassi

- def decorator_function (f): pass # definisce un decoratore chiamato decorator_function
- @decorator_function
  def decorated_function (): passa # la funzione è ora spostata (decorata da) decorator_function
- decorated_function = decorator_function (decorated_function) # equivale a usare lo zucchero sintattico `@decorator_function`

Parametri

Parametro	Dettagli
f	La funzione da decorare (avvolta)

Examples

Funzione Decoratore

I decoratori aumentano il comportamento di altre funzioni o metodi. Qualsiasi funzione che prende una funzione come parametro e restituisce una funzione aumentata può essere utilizzata come **decoratore** .

```
# This simplest decorator does nothing to the function being decorated. Such
# minimal decorators can occasionally be used as a kind of code markers.
def super_secret_function(f):
    return f

@super_secret_function
def my_function():
    print("This is my secret function.")
```

La `@` -notation è zucchero sintattico che è equivalente al seguente:

```
my_function = super_secret_function(my_function)
```

È importante tenerlo presente per capire come funzionano i decoratori. Questa sintassi "unsugared" rende chiaro il motivo per cui la funzione decoratore prende una funzione come argomento e perché dovrebbe restituire un'altra funzione. Dimostra anche cosa succederebbe se *non si* restituisse una funzione:

```
def disabled(f):
    """
    This function returns nothing, and hence removes the decorated function
    from the local scope.
    """
    pass

@disabled
def my_function():
    print("This function can no longer be called...")

my_function()
# TypeError: 'NoneType' object is not callable
```

Pertanto, di solito definiamo una *nuova funzione* all'interno del decoratore e la restituiamo. Questa nuova funzione dovrebbe prima fare qualcosa che deve fare, quindi chiamare la funzione originale e infine elaborare il valore di ritorno. Considera questa semplice funzione di decoratore che stampa gli argomenti ricevuti dalla funzione originale, quindi li chiama.

```
#This is the decorator
def print_args(func):
    def inner_func(*args, **kwargs):
        print(args)
        print(kwargs)
        return func(*args, **kwargs) #Call the original function with its arguments.
    return inner_func

@print_args
def multiply(num_a, num_b):
    return num_a * num_b

print(multiply(3, 5))
#Output:
# (3,5) - This is actually the 'args' that the function receives.
# {} - This is the 'kwargs', empty because we didn't specify keyword arguments.
# 15 - The result of the function.
```

Classe decoratore

Come accennato nell'introduzione, un decoratore è una funzione che può essere applicata a un'altra funzione per aumentarne il comportamento. Lo zucchero sintattico è equivalente al seguente: `my_func = decorator(my_func)` . Ma cosa succede se il `decorator` era invece una classe? La sintassi funzionerebbe ancora, tranne che ora `my_func` viene sostituito con un'istanza della classe `decorator` . Se questa classe implementa il metodo magico `__call__()` , allora sarebbe ancora possibile usare `my_func` come se fosse una funzione:

```
class Decorator(object):
    """Simple decorator class."""

    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):
        print('Before the function call.')
        res = self.func(*args, **kwargs)
        print('After the function call.')
        return res

@Decorator
def testfunc():
    print('Inside the function.')

testfunc()
# Before the function call.
# Inside the function.
# After the function call.
```

Nota che una funzione decorata con un decoratore di classe non sarà più considerata una "funzione" dal punto di vista del controllo dei caratteri:

```
import types
isinstance(testfunc, types.FunctionType)
# False
type(testfunc)
# <class '__main__.Decorator'>
```

Metodi di decorazione

Per i metodi di decorazione è necessario definire un `__get__` __get__ aggiuntivo:

```
from types import MethodType

class Decorator(object):
    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):
        print('Inside the decorator.')
        return self.func(*args, **kwargs)

    def __get__(self, instance, cls):
        # Return a Method if it is called on an instance
        return self if instance is None else MethodType(self, instance)

class Test(object):
    @Decorator
    def __init__(self):
        pass

a = Test()
```

Dentro il decoratore.

Avvertimento!

I decoratori di classe producono solo un'istanza per una funzione specifica, quindi decorare un metodo con un decoratore di classe condividerà lo stesso decoratore tra tutte le istanze di quella classe:

```
from types import MethodType

class CountCallsDecorator(object):
    def __init__(self, func):
        self.func = func
        self.ncalls = 0    # Number of calls of this method

    def __call__(self, *args, **kwargs):
        self.ncalls += 1   # Increment the calls counter
        return self.func(*args, **kwargs)

    def __get__(self, instance, cls):
        return self if instance is None else MethodType(self, instance)

class Test(object):
    def __init__(self):
        pass

    @CountCallsDecorator
    def do_something(self):
        return 'something was done'

a = Test()
a.do_something()
a.do_something.ncalls   # 1
b = Test()
b.do_something()
b.do_something.ncalls   # 2
```

Fare un decoratore assomiglia alla funzione decorata

I decoratori normalmente spogliano i metadati della funzione perché non sono gli stessi. Ciò può causare problemi quando si utilizza la meta-programmazione per accedere in modo dinamico ai metadati delle funzioni. I metadati includono anche le docstring della funzione e il suo nome. `functools.wraps` rende la funzione decorata simile alla funzione originale copiando diversi attributi alla funzione wrapper.

```
from functools import wraps
```

I due metodi di avvolgere un decoratore stanno ottenendo la stessa cosa nascondendo che la funzione originale è stata decorata. Non vi è alcun motivo per preferire la versione della funzione alla versione della classe a meno che non si stia già utilizzando uno sull'altro.

Come una funzione

```
def decorator(func):
    # Copies the docstring, name, annotations and module to the decorator
    @wraps(func)
    def wrapped_func(*args, **kwargs):
        return func(*args, **kwargs)
    return wrapped_func

@decorator
def test():
    pass

test.__name__
```

'test'

Come classe

```
class Decorator(object):
    def __init__(self, func):
        # Copies name, module, annotations and docstring to the instance.
        self._wrapped = wraps(func)(self)

    def __call__(self, *args, **kwargs):
        return self._wrapped(*args, **kwargs)

@Decorator
def test():
    """Docstring of test."""
    pass

test.__doc__
```

"Docstring of test".

Decoratore con argomenti (fabbrica di decorazioni)

Un decoratore accetta un solo argomento: la funzione da decorare. Non c'è modo di passare altri argomenti.

Ma spesso si desiderano ulteriori argomenti. Il trucco è quindi di creare una funzione che accetta argomenti arbitrari e restituisce un decoratore.

Funzioni del decoratore

```
def decoratorfactory(message):
    def decorator(func):
        def wrapped_func(*args, **kwargs):
```

```
            print('The decorator wants to tell you: {}'.format(message))
            return func(*args, **kwargs)
        return wrapped_func
    return decorator

@decoratorfactory('Hello World')
def test():
    pass

test()
```

L'arredatore vuole dirti: Hello World

Nota importante:

Con queste fabbriche di decoratori **devi** chiamare il decoratore con una coppia di parentesi:

```
@decoratorfactory # Without parentheses
def test():
    pass

test()
```

TypeError: decorator () mancante 1 argomento posizionale richiesto: 'func'

Lezioni di decoratore

```
def decoratorfactory(*decorator_args, **decorator_kwargs):

    class Decorator(object):
        def __init__(self, func):
            self.func = func

        def __call__(self, *args, **kwargs):
            print('Inside the decorator with arguments {}'.format(decorator_args))
            return self.func(*args, **kwargs)

    return Decorator

@decoratorfactory(10)
def test():
    pass

test()
```

All'interno del decoratore con argomenti (10,)

Crea una classe singleton con un decoratore

Un singleton è un pattern che limita l'istanza di una classe a un'istanza / oggetto. Usando un decoratore, possiamo definire una classe come un singleton forzando la classe a restituire

un'istanza esistente della classe o creare una nuova istanza (se non esiste).

```
def singleton(cls):
    instance = [None]
    def wrapper(*args, **kwargs):
        if instance[0] is None:
            instance[0] = cls(*args, **kwargs)
        return instance[0]

    return wrapper
```

Questo decoratore può essere aggiunto a qualsiasi dichiarazione di classe e si assicurerà che al massimo venga creata una sola istanza della classe. Qualsiasi chiamata successiva restituirà l'istanza di classe già esistente.

```
@singleton
class SomeSingletonClass:
    x = 2
    def __init__(self):
        print("Created!")

instance = SomeSingletonClass()  # prints: Created!
instance = SomeSingletonClass()  # doesn't print anything
print(instance.x)                # 2

instance.x = 3
print(SomeSingletonClass().x)    # 3
```

Quindi non importa se ti riferisci all'istanza della classe tramite la tua variabile locale o se crei un'altra "istanza", ottieni sempre lo stesso oggetto.

Utilizzare un decoratore per cronometrare una funzione

```
import time
def timer(func):
    def inner(*args, **kwargs):
        t1 = time.time()
        f = func(*args, **kwargs)
        t2 = time.time()
        print 'Runtime took {0} seconds'.format(t2-t1)
        return f
    return inner

@timer
def example_function():
    #do stuff


example_function()
```

Leggi decoratori online: https://riptutorial.com/it/python/topic/229/decoratori

Capitolo 43: Definire funzioni con argomenti lista

Examples

Funzione e chiamata

Gli elenchi come argomenti sono solo un'altra variabile:

```
def func(myList):
    for item in myList:
        print(item)
```

e può essere passato nella chiamata alla funzione stessa:

```
func([1,2,3,5,7])

1
2
3
5
7
```

O come variabile:

```
aList = ['a','b','c','d']
func(aList)

a
b
c
d
```

Leggi Definire funzioni con argomenti lista online:
https://riptutorial.com/it/python/topic/7744/definire-funzioni-con-argomenti-lista

Capitolo 44: dentellatura

Examples

Errori di indentazione

La spaziatura dovrebbe essere uniforme e uniforme. Il rientro improprio può causare un `IndentationError` o fare in modo che il programma faccia qualcosa di inaspettato. Nell'esempio seguente viene `IndentationError` un `IndentationError` :

```
a = 7
if a > 5:
  print "foo"
else:
  print "bar"
 print "done"
```

O se la linea che segue i due punti non è rientrata, verrà generato anche un `IndentationError` :

```
if True:
print "true"
```

Se si aggiunge indentazione a cui non appartiene, verrà generato un `IndentationError` :

```
if  True:
    a = 6
        b = 5
```

Se si dimentica di annullare la funzionalità potrebbe essere perso. In questo esempio, viene restituito `None` invece del `False` atteso:

```
def isEven(a):
    if a%2 ==0:
        return True
        #this next line should be even with the if
        return False
print isEven(7)
```

Semplice esempio

Per Python, Guido van Rossum ha basato il raggruppamento di affermazioni sul rientro. Le ragioni di ciò sono spiegate nella prima sezione delle "Domande frequenti su Python di design e cronologia" . I due punti, `:` , vengono utilizzati per dichiarare un blocco di codice rientrato , come nell'esempio seguente:

```
class ExampleClass:
    #Every function belonging to a class must be indented equally
    def __init__(self):
```

```
        name = "example"

    def someFunction(self, a):
        #Notice everything belonging to a function must be indented
        if a > 5:
            return True
        else:
            return False

#If a function is not indented to the same level it will not be considers as part of the
parent class
def separateFunction(b):
    for i in b:
    #Loops are also indented and nested conditions start a new indentation
        if i == 1:
            return True
    return False

separateFunction([2,3,5,6,1])
```

Spazi o tabulati?

Il rientro consigliato è di 4 spazi ma è possibile utilizzare tabulazioni o spazi purché coerenti. ***Non mischiare tab e spazi in Python*** poiché ciò causerà un errore in Python 3 e può causare errori in Python 2 .

Come l'indentazione è analizzata

Lo spazio bianco viene gestito dall'analizzatore lessicale prima di essere analizzato.

L'analizzatore lessicale utilizza una pila per memorizzare i livelli di indentazione. All'inizio, la pila contiene solo il valore 0, che è la posizione più a sinistra. Ogni volta che inizia un blocco nidificato, il nuovo livello di indentazione viene inserito nello stack e un token "INDENT" viene inserito nel flusso di token che viene passato al parser. Non può mai esserci più di un gettone "INDENT" in una riga (`IndentationError`).

Quando viene rilevata una riga con un livello di indentazione più piccolo, i valori vengono estratti dallo stack fino a quando un valore è in cima, che è uguale al nuovo livello di indentazione (se non viene trovato nessuno, si verifica un errore di sintassi). Per ogni valore spuntato, viene generato un token "DEDENT". Ovviamente, possono essere presenti più token "DEDENT" in fila.

L'analizzatore lessicale salta le righe vuote (quelle che contengono solo spazi bianchi e probabilmente commenti) e non genererà mai token "INDENT" o "DEDENT" per loro.

Alla fine del codice sorgente, i token "DEDENT" vengono generati per ogni livello di rientro rimasto nello stack, fino a quando non viene lasciato solo lo 0.

Per esempio:

```
if foo:
    if bar:
        x = 42
```

```
else:
    print foo
```

viene analizzato come:

```
<if> <foo> <:>                    [0]
<INDENT> <if> <bar> <:>           [0, 4]
<INDENT> <x> <=> <42>             [0, 4, 8]
<DEDENT> <DEDENT> <else> <:>      [0]
<INDENT> <print> <foo>            [0, 2]
<DEDENT>
```

Il parser gestisce i token "INDENT" e "DEDENT" come delimitatori di blocchi.

Leggi dentellatura online: https://riptutorial.com/it/python/topic/2597/dentellatura

Capitolo 45: descrittore

Examples

Descrittore semplice

Esistono due diversi tipi di descrittori. I descrittori di dati sono definiti come oggetti che definiscono sia un `__get__()` sia un `__set__()` , mentre i descrittori non di dati definiscono solo un `__get__()` . Questa distinzione è importante quando si considerano le sostituzioni e lo spazio dei nomi del dizionario di un'istanza. Se un descrittore di dati e una voce nel dizionario di un'istanza condividono lo stesso nome, il descrittore di dati avrà la precedenza. Tuttavia, se invece un descrittore non di dati e una voce nel dizionario di un'istanza condividono lo stesso nome, la voce del dizionario dell'istanza avrà la precedenza.

Per creare un descrittore di dati di sola lettura, definisci sia **get** () che **set** () con **set** () generando un AttributeError quando chiamato. Definire il metodo **set** () con un segnaposto che genera un'eccezione è sufficiente per renderlo un descrittore di dati.

```
descr.__get__(self, obj, type=None) --> value
descr.__set__(self, obj, value) --> None
descr.__delete__(self, obj) --> None
```

Un esempio implementato:

```
class DescPrinter(object):
    """A data descriptor that logs activity."""
    _val = 7

    def __get__(self, obj, objtype=None):
        print('Getting ...')
        return self._val

    def __set__(self, obj, val):
        print('Setting', val)
        self._val = val

    def __delete__(self, obj):
        print('Deleting ...')
        del self._val


class Foo():
    x = DescPrinter()

i = Foo()
i.x
# Getting ...
# 7

i.x = 100
# Setting 100
i.x
```

```
# Getting ...
# 100

del i.x
# Deleting ...
i.x
# Getting ...
# 7
```

Conversioni bidirezionali

Gli oggetti descrittori possono consentire agli attributi degli oggetti correlati di reagire automaticamente alle modifiche.

Supponiamo di voler modellare un oscillatore con una determinata frequenza (in Hertz) e un periodo (in secondi). Quando aggiorniamo la frequenza, vogliamo che il periodo si aggiorni e quando aggiorniamo il periodo vogliamo che la frequenza si aggiorni:

```
 >>> oscillator = Oscillator(freq=100.0)  # Set frequency to 100.0 Hz
>>> oscillator.period  # Period is 1 / frequency, i.e. 0.01 seconds
0.01
>>> oscillator.period = 0.02  # Set period to 0.02 seconds
>>> oscillator.freq # The frequency is automatically adjusted
50.0
>>> oscillator.freq = 200.0  # Set the frequency to 200.0 Hz
>>> oscillator.period  # The period is automatically adjusted
0.005
```

Selezioniamo uno dei valori (frequenza, in Hertz) come "ancora", cioè quello che può essere impostato senza conversione e scrivere per esso una classe descrittore:

```
class Hertz(object):
    def __get__(self, instance, owner):
        return self.value

    def __set__(self, instance, value):
        self.value = float(value)
```

Il valore "altro" (periodo, in secondi) è definito in termini di ancoraggio. Scriviamo una classe descrittore che esegue le nostre conversioni:

```
class Second(object):
    def __get__(self, instance, owner):
        # When reading period, convert from frequency
        return 1 / instance.freq

    def __set__(self, instance, value):
        # When setting period, update the frequency
        instance.freq = 1 / float(value)
```

Ora possiamo scrivere la classe Oscillator:

```
class Oscillator(object):
```

```
    period = Second()  # Set the other value as a class attribute

    def __init__(self, freq):
        self.freq = Hertz()  # Set the anchor value as an instance attribute
        self.freq = freq  # Assign the passed value - self.period will be adjusted
```

Leggi descrittore online: https://riptutorial.com/it/python/topic/3405/descrittore

Capitolo 46: Differenza tra modulo e pacchetto

Osservazioni

È possibile inserire un pacchetto Python in un file ZIP e utilizzarlo in questo modo se aggiungi queste righe all'inizio del tuo script:

```
import sys
sys.path.append("package.zip")
```

Examples

moduli

Un modulo è un singolo file Python che può essere importato. L'utilizzo di un modulo è simile a questo:

`module.py`

```
def hi():
    print("Hello world!")
```

`my_script.py`

```
import module
module.hi()
```

in un interprete

```
>>> from module import hi
>>> hi()
# Hello world!
```

Pacchi

Un pacchetto è composto da più file Python (o moduli) e può anche includere librerie scritte in C o C ++. Invece di essere un singolo file, è un'intera struttura di cartelle che potrebbe apparire come questa:

`package` cartelle

- `__init__.py`
- `dog.py`
- `hi.py`

`__init__.py`

```
from package.dog import woof
from package.hi import hi
```

`dog.py`

```
def woof():
    print("WOOF!!!")
```

`hi.py`

```
def hi():
    print("Hello world!")
```

Tutti i pacchetti Python devono contenere un file `__init__.py` . Quando si importa un pacchetto nello script (`import package`), verrà eseguito lo script `__init__.py` , che consente di accedere a tutte le funzioni del pacchetto. In questo caso, ti consente di utilizzare le funzioni `package.hi` e `package.woof` .

Leggi Differenza tra modulo e pacchetto online:
https://riptutorial.com/it/python/topic/3142/differenza-tra-modulo-e-pacchetto

Capitolo 47: Distribuzione

Examples

py2app

Per usare il framework py2app devi prima installarlo. Fallo aprendo il terminale e inserendo il seguente comando:

```
sudo easy_install -U py2app
```

Puoi anche `pip` installare i pacchetti come:

```
pip install py2app
```

Quindi crea il file di installazione per il tuo script python:

```
py2applet --make-setup MyApplication.py
```

Modifica le impostazioni del file di installazione a tuo piacimento, questo è il valore predefinito:

```
"""
This is a setup.py script generated by py2applet

Usage:
    python setup.py py2app
"""

from setuptools import setup

APP = ['test.py']
DATA_FILES = []
OPTIONS = {'argv_emulation': True}

setup(
    app=APP,
    data_files=DATA_FILES,
    options={'py2app': OPTIONS},
    setup_requires=['py2app'],
)
```

Per aggiungere un file di icona (questo file deve avere un'estensione .icns) o includere immagini nell'applicazione come riferimento, modificare le opzioni come mostrato:

```
DATA_FILES = ['myInsertedImage.jpg']
OPTIONS = {'argv_emulation': True, 'iconfile': 'myCoolIcon.icns'}
```

Infine, inserisci questo nel terminale:

```
python setup.py py2app
```

Lo script dovrebbe essere eseguito e troverai la tua applicazione finita nella cartella dist.

Utilizzare le seguenti opzioni per una maggiore personalizzazione:

```
optimize (-O)          optimization level: -O1 for "python -O", -O2 for
                       "python -OO", and -O0 to disable [default: -O0]

includes (-i)          comma-separated list of modules to include

packages (-p)          comma-separated list of packages to include

extension              Bundle extension [default:.app for app, .plugin for
                       plugin]

extra-scripts          comma-separated list of additional scripts to include
                       in an application or plugin.
```

cx_Freeze

Installa cx_Freeze da qui

Decomprimere la cartella ed eseguire questi comandi da quella directory:

```
python setup.py build
sudo python setup.py install
```

Crea una nuova directory per il tuo script python e crea un file **"setup.py"** nella stessa directory con il seguente contenuto:

```
application_title = "My Application" # Use your own application name
main_python_file = "my_script.py" # Your python script

import sys

from cx_Freeze import setup, Executable

base = None
if sys.platform == "win32":
    base = "Win32GUI"

includes = ["atexit","re"]

setup(
        name = application_title,
        version = "0.1",
        description = "Your Description",
        options = {"build_exe" : {"includes" : includes }},
        executables = [Executable(main_python_file, base = base)])
```

Ora esegui il tuo setup.py dal terminale:

```
python setup.py bdist_mac
```

NOTA: su El Capitan questo dovrà essere eseguito come root con la modalità SIP disabilitata.

Leggi Distribuzione online: https://riptutorial.com/it/python/topic/2026/distribuzione

Capitolo 48: Distribuzione

Examples

Caricamento di un pacchetto Conda

Prima di iniziare devi avere:

Anaconda installato sul tuo sistema Account su Binstar Se non stai usando Anaconda 1.6+ installa il client della riga di comando binstar :

```
$ conda install binstar
$ conda update binstar
```

Se non si utilizza Anaconda, Binstar è disponibile anche su pypi:

```
$ pip install binstar
```

Ora possiamo accedere:

```
$ binstar login
```

Verifica il tuo login con il comando whoami:

```
$ binstar whoami
```

Stiamo per caricare un pacchetto con una semplice funzione 'ciao mondo'. Per iniziare, inizia a prendere il mio repository di pacchetti dimostrativi da Github:

```
$ git clone https://github.com/<NAME>/<Package>
```

Questa è una piccola directory che assomiglia a questo:

```
package/
        setup.py
        test_package/
           __init__.py
           hello.py
           bld.bat
           build.sh
           meta.yaml
```

`Setup.py` è il file di sviluppo standard di python e `hello.py` ha la nostra singola funzione hello_world ().

`bld.bat` , `build.sh` e `meta.yaml` sono script e metadati per il pacchetto `Conda` . Puoi leggere la pagina di build di Conda per maggiori informazioni su questi tre file e il loro scopo.

Ora creiamo il pacchetto eseguendo:

```
$ conda build test_package/
```

Questo è tutto ciò che serve per creare un pacchetto Conda.

Il passaggio finale è il caricamento su binstar copiando e incollando l'ultima riga della stampa dopo aver eseguito il comando test_package / comando conda build. Sul mio sistema il comando è:

```
$ binstar upload /home/xavier/anaconda/conda-bld/linux-64/test_package-0.1.0-py27_0.tar.bz2
```

Poiché è la prima volta che crei un pacchetto e rilasci, ti verrà richiesto di compilare alcuni campi di testo che potrebbero in alternativa essere fatti tramite l'app web.

Si vedrà un `done` stampato fuori per confermare di aver caricato con successo il pacchetto Conda a Binstar.

Leggi Distribuzione online: https://riptutorial.com/it/python/topic/4064/distribuzione

Capitolo 49: Dizionario

Sintassi

- mydict = {}
- mydict [k] = valore
- value = mydict [k]
- valore = mydict.get (k)
- value = mydict.get (k, "valore_predefinito")

Parametri

Parametro	Dettagli
chiave	La chiave desiderata per la ricerca
valore	Il valore da impostare o restituire

Osservazioni

Elementi utili da ricordare quando si crea un dizionario:

- Ogni chiave deve essere **unica** (altrimenti verrà ignorata)
- Ogni chiave deve essere **hashable** (può utilizzare la `hash` funzione di hash che, altrimenti `TypeError` sarà gettato)
- Non c'è un ordine particolare per le chiavi.

Examples

Accesso ai valori di un dizionario

```
dictionary = {"Hello": 1234, "World": 5678}
print(dictionary["Hello"])
```

Il codice precedente stamperà `1234` .

La stringa `"Hello"` in questo esempio è chiamata *chiave* . È usato per cercare un valore nel `dict` posizionando la chiave tra parentesi quadre.

Il numero `1234` è visto dopo i rispettivi due punti nella definizione del `dict` . Questo è chiamato il *valore* che `"Hello"` *mappe al* presente `dict` .

La ricerca di un valore come questo con una chiave che non esiste solleverà un'eccezione `KeyError` , interrompendo l'esecuzione se non `KeyError` . Se vogliamo accedere a un valore senza

rischiare un `KeyError` , possiamo usare il metodo `dictionary.get` . Per impostazione predefinita, se la chiave non esiste, il metodo restituirà `None` . Possiamo passargli un secondo valore da restituire invece di `None` in caso di una ricerca fallita.

```
w = dictionary.get("whatever")
x = dictionary.get("whatever", "nuh-uh")
```

In questo esempio `w` otterrà il valore `None` e `x` otterrà il valore `"nuh-uh"` .

Il costruttore dict ()

Il costruttore `dict()` può essere usato per creare dizionari da argomenti di parole chiave, o da un singolo iterabile di coppie chiave-valore, o da un singolo dizionario e argomenti di parole chiave.

```
dict(a=1, b=2, c=3)                    # {'a': 1, 'b': 2, 'c': 3}
dict([('d', 4), ('e', 5), ('f', 6)])   # {'d': 4, 'e': 5, 'f': 6}
dict([('a', 1)], b=2, c=3)             # {'a': 1, 'b': 2, 'c': 3}
dict({'a' : 1, 'b' : 2}, c=3)          # {'a': 1, 'b': 2, 'c': 3}
```

Evitare Eccezioni KeyError

Un errore comune quando si utilizzano i dizionari è di accedere a una chiave inesistente. Ciò in genere `KeyError` un'eccezione `KeyError`

```
mydict = {}
mydict['not there']
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'not there'
```

Un modo per evitare errori chiave consiste nell'utilizzare il metodo `dict.get` , che consente di specificare un valore predefinito da restituire nel caso di una chiave assente.

```
value = mydict.get(key, default_value)
```

Che restituisce `mydict[key]` se esiste, ma restituisce altrimenti `default_value` . Nota che questo non aggiunge la `key` a `mydict` . Quindi, se si desidera mantenere quella coppia chiave-valore, si dovrebbe utilizzare `mydict.setdefault(key, default_value)` , che *non* memorizzare la coppia chiave-valore.

```
mydict = {}
print(mydict)
# {}
print(mydict.get("foo", "bar"))
# bar
print(mydict)
# {}
print(mydict.setdefault("foo", "bar"))
# bar
```

```
print(mydict)
# {'foo': 'bar'}
```

Un modo alternativo per affrontare il problema è l'eccezione

```
try:
    value = mydict[key]
except KeyError:
    value = default_value
```

Puoi anche controllare se la chiave è `in` dizionario.

```
if key in mydict:
    value = mydict[key]
else:
    value = default_value
```

Si noti, tuttavia, che negli ambienti a più thread è possibile che la chiave venga rimossa dal dizionario dopo il controllo, creando una condizione di competizione in cui è ancora possibile lanciare l'eccezione.

Un'altra opzione è usare una sottoclasse di dict, collections.defaultdict, che ha un default_factory per creare nuove voci nel dict quando viene dato un new_key.

Accesso a chiavi e valori

Quando si lavora con i dizionari, è spesso necessario accedere a tutte le chiavi e i valori del dizionario, in un ciclo `for` , in una list comprehension o semplicemente come un elenco semplice.

Dato un dizionario come:

```
mydict = {
    'a': '1',
    'b': '2'
}
```

Puoi ottenere un elenco di chiavi usando il metodo `keys()` :

```
print(mydict.keys())
# Python2: ['a', 'b']
# Python3: dict_keys(['b', 'a'])
```

Se invece vuoi un elenco di valori, usa il metodo `values()` :

```
print(mydict.values())
# Python2: ['1', '2']
# Python3: dict_values(['2', '1'])
```

Se vuoi lavorare sia con la chiave che con il suo valore corrispondente, puoi usare il metodo `items()` :

```
print(mydict.items())
# Python2: [('a', '1'), ('b', '2')]
# Python3: dict_items([('b', '2'), ('a', '1')])
```

NOTA: poiché un `dict` non è ordinato, le `keys()` , `values()` e `items()` non hanno alcun ordinamento. Utilizzare `sort()` , `sorted()` , o un `OrderedDict` se vi preoccupate per l'ordine che questi metodi restituiscono.

Python 2/3 Differenza: In Python 3, questi metodi restituiscono oggetti iterabili speciali, non elenchi e sono l'equivalente dei `iterkeys()` , `itervalues()` e `iteritems()` di Python 2. Questi oggetti possono essere utilizzati come liste per la maggior parte, anche se ci sono alcune differenze. Vedi PEP 3106 per maggiori dettagli.

Introduzione al dizionario

Un dizionario è un esempio di un *archivio di valori chiave* noto anche come *Mapping* in Python. Permette di memorizzare e recuperare elementi facendo riferimento a una chiave. Poiché i dizionari fanno riferimento alla chiave, hanno una ricerca molto veloce. Poiché vengono principalmente utilizzati per referenziare gli elementi per chiave, non vengono ordinati.

creando un ditt

I dizionari possono essere avviati in molti modi:

sintassi letterale

```
d = {}                          # empty dict
d = {'key': 'value'}            # dict with initial values
```

Python 3.x 3.5

```
# Also unpacking one or multiple dictionaries with the literal syntax is possible

# makes a shallow copy of otherdict
d = {**otherdict}
# also updates the shallow copy with the contents of the yetanotherdict.
d = {**otherdict, **yetanotherdict}
```

Detto comprensione

```
d = {k:v for k,v in [('key', 'value',)]}
```

vedi anche: Comprensioni

classe built-in: `dict()`

```
d = dict()                      # emtpy dict
d = dict(key='value')           # explicit keyword arguments
d = dict([('key', 'value')])  # passing in a list of key/value pairs
# make a shallow copy of another dict (only possible if keys are only strings!)
d = dict(**otherdict)
```

modificare un dett

Per aggiungere elementi a un dizionario, è sufficiente creare una nuova chiave con un valore:

```
d['newkey'] = 42
```

È anche possibile aggiungere `list` e `dictionary` come valore:

```
d['new_list'] = [1, 2, 3]
d['new_dict'] = {'nested_dict': 1}
```

Per eliminare un elemento, elimina la chiave dal dizionario:

```
del d['newkey']
```

Dizionario con valori predefiniti

Disponibile nella libreria standard come `defaultdict`

```
from collections import defaultdict

d = defaultdict(int)
d['key']                          # 0
d['key'] = 5
d['key']                          # 5

d = defaultdict(lambda: 'empty')
d['key']                          # 'empty'
d['key'] = 'full'
d['key']                          # 'full'
```

[*] In alternativa, se devi utilizzare la classe `dict` incorporata, l' `using dict.setdefault()` ti consente di creare un valore predefinito ogni volta che accedi a una chiave che non esisteva prima:

```
>>> d = {}
{}
>>> d.setdefault('Another_key', []).append("This worked!")
>>> d
{'Another_key': ['This worked!']}
```

Tieni presente che se hai molti valori da aggiungere, `dict.setdefault()` creerà una nuova istanza del valore iniziale (in questo esempio a `[]`) ogni volta che viene chiamato, il che potrebbe creare carichi di lavoro non necessari.

[*] *Python Cookbook, 3a edizione, di David Beazley e Brian K. Jones (O'Reilly). Copyright 2013 David Beazley e Brian Jones, 978-1-449-34037-7.*

Creare un dizionario ordinato

È possibile creare un dizionario ordinato che seguirà un determinato ordine durante l'iterazione sui tasti del dizionario.

Usa `OrderedDict` dal modulo delle `collections` . Ciò restituirà sempre gli elementi del dizionario nell'ordine di inserimento originale quando viene ripetuto.

```
from collections import OrderedDict

d = OrderedDict()
d['first'] = 1
d['second'] = 2
d['third'] = 3
d['last'] = 4

# Outputs "first 1", "second 2", "third 3", "last 4"
for key in d:
    print(key, d[key])
```

Disimballaggio dei dizionari usando l'operatore **

È possibile utilizzare l'operatore di disimballaggio dell'argomento parola chiave `**` per distribuire le coppie chiave-valore in un dizionario negli argomenti di una funzione. Un esempio semplificato dalla documentazione ufficiale :

```
>>>
>>> def parrot(voltage, state, action):
...     print("This parrot wouldn't", action, end=' ')
...     print("if you put", voltage, "volts through it.", end=' ')
...     print("E's", state, "!")
...
>>> d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}
>>> parrot(**d)

This parrot wouldn't VOOM if you put four million volts through it. E's bleedin' demised !
```

A partire da Python 3.5 è anche possibile utilizzare questa sintassi per unire un numero arbitrario di oggetti `dict` .

```
>>> fish = {'name': "Nemo", 'hands': "fins", 'special': "gills"}
>>> dog = {'name': "Clifford", 'hands': "paws", 'color': "red"}
>>> fishdog = {**fish, **dog}
>>> fishdog

{'hands': 'paws', 'color': 'red', 'name': 'Clifford', 'special': 'gills'}
```

Come dimostra questo esempio, le chiavi duplicate si associano al loro ultimo valore (ad esempio "Clifford" sovrascrive "Nemo").

Unione di dizionari

Considera i seguenti dizionari:

```
>>> fish = {'name': "Nemo", 'hands': "fins", 'special': "gills"}
>>> dog = {'name': "Clifford", 'hands': "paws", 'color': "red"}
```

Python 3.5+

```
>>> fishdog = {**fish, **dog}
>>> fishdog
{'hands': 'paws', 'color': 'red', 'name': 'Clifford', 'special': 'gills'}
```

Come dimostra questo esempio, le chiavi duplicate si associano al loro ultimo valore (ad esempio "Clifford" sovrascrive "Nemo").

Python 3.3+

```
>>> from collections import ChainMap
>>> dict(ChainMap(fish, dog))
{'hands': 'fins', 'color': 'red', 'special': 'gills', 'name': 'Nemo'}
```

Con questa tecnica il valore più importante ha la precedenza per un dato tasto piuttosto che per l'ultimo ("Clifford" viene buttato in favore di "Nemo").

Python 2.x, 3.x

```
>>> from itertools import chain
>>> dict(chain(fish.items(), dog.items()))
{'hands': 'paws', 'color': 'red', 'name': 'Clifford', 'special': 'gills'}
```

Questo usa il valore ultimo, come con la tecnica basata su `**` per l'unione ("Clifford" sovrascrive "Nemo").

```
>>> fish.update(dog)
>>> fish
{'color': 'red', 'hands': 'paws', 'name': 'Clifford', 'special': 'gills'}
```

`dict.update` usa il secondo dict per sovrascrivere quello precedente.

La virgola finale

Come gli elenchi e le tuple, puoi includere una virgola finale nel dizionario.

```
role = {"By day": "A typical programmer",
        "By night": "Still a typical programmer", }
```

PEP 8 indica che è necessario lasciare uno spazio tra la virgola finale e quella di chiusura.

Tutte le combinazioni di valori del dizionario

```
options = {
    "x": ["a", "b"],
    "y": [10, 20, 30]
}
```

Dato un dizionario come quello mostrato sopra, dove c'è una lista che rappresenta un insieme di valori da esplorare per la chiave corrispondente. Supponiamo di voler esplorare `"x"="a"` con `"y"=10` , quindi `"x"="a"` con `"y"=10` e così via fino a quando non hai esplorato tutte le possibili combinazioni.

Puoi creare una lista che restituisca tutte queste combinazioni di valori usando il seguente codice.

```
import itertools

options = {
    "x": ["a", "b"],
    "y": [10, 20, 30]}

keys = options.keys()
values = (options[key] for key in keys)
combinations = [dict(zip(keys, combination)) for combination in itertools.product(*values)]
print combinations
```

Questo ci dà il seguente elenco memorizzato nelle `combinations` variabili:

```
[{'x': 'a', 'y': 10},
 {'x': 'b', 'y': 10},
 {'x': 'a', 'y': 20},
 {'x': 'b', 'y': 20},
 {'x': 'a', 'y': 30},
 {'x': 'b', 'y': 30}]
```

Iterare su un dizionario

Se si utilizza un dizionario come iteratore (ad esempio in una dichiarazione `for`), attraversa le **chiavi** del dizionario. Per esempio:

```
d = {'a': 1, 'b': 2, 'c':3}
for key in d:
    print(key, d[key])
# c 3
# b 2
# a 1
```

Lo stesso è vero quando usato in una comprensione

```
print([key for key in d])
# ['c', 'b', 'a']
```

Python 3.x 3.0

Il metodo `items()` può essere utilizzato per ripetere contemporaneamente sia la **chiave** che il **valore** :

```
for key, value in d.items():
    print(key, value)
# c 3
# b 2
# a 1
```

Mentre il metodo `values()` può essere usato per iterare solo sui valori, come ci si aspetterebbe:

```
for key, value in d.values():
    print(key, value)
    # 3
    # 2
    # 1
```

Python 2.x 2.2

Qui, i metodi `keys()` , `values()` e `items()` restituiscono liste, e ci sono i tre metodi aggiuntivi `iterkeys()` `itervalues()` e `iteritems()` per restituire iteratori.

Creare un dizionario

Regole per creare un dizionario:

- Ogni chiave deve essere **unica** (altrimenti verrà ignorata)
- Ogni chiave deve essere **hashable** (può utilizzare la `hash` funzione di hash che, altrimenti `TypeError` sarà gettato)
- Non c'è un ordine particolare per le chiavi.

```
# Creating and populating it with values
stock = {'eggs': 5, 'milk': 2}

# Or creating an empty dictionary
dictionary = {}

# And populating it after
dictionary['eggs'] = 5
dictionary['milk'] = 2

# Values can also be lists
mydict = {'a': [1, 2, 3], 'b': ['one', 'two', 'three']}

# Use list.append() method to add new elements to the values list
mydict['a'].append(4)   # => {'a': [1, 2, 3, 4], 'b': ['one', 'two', 'three']}
```

```
mydict['b'].append('four')  # => {'a': [1, 2, 3, 4], 'b': ['one', 'two', 'three', 'four']}

# We can also create a dictionary using a list of two-items tuples
iterable = [('eggs', 5), ('milk', 2)]
dictionary = dict(iterables)

# Or using keyword argument:
dictionary = dict(eggs=5, milk=2)

# Another way will be to use the dict.fromkeys:
dictionary = dict.fromkeys((milk, eggs))  # => {'milk': None, 'eggs': None}
dictionary = dict.fromkeys((milk, eggs), (2, 5))  # => {'milk': 2, 'eggs': 5}
```

Esempio di dizionari

I dizionari mappano le chiavi ai valori.

```
car = {}
car["wheels"] = 4
car["color"] = "Red"
car["model"] = "Corvette"
```

I valori del dizionario sono accessibili tramite le loro chiavi.

```
print "Little " + car["color"] + " " + car["model"] + "!"
# This would print out "Little Red Corvette!"
```

I dizionari possono anche essere creati in uno stile JSON:

```
car = {"wheels": 4, "color": "Red", "model": "Corvette"}
```

I valori del dizionario possono essere ripetuti su:

```
for key in car:
  print key + ": " + car[key]

# wheels: 4
# color: Red
# model: Corvette
```

Leggi Dizionario online: https://riptutorial.com/it/python/topic/396/dizionario

Capitolo 50: Django

introduzione

Django è un framework Web Python di alto livello che incoraggia lo sviluppo rapido e il design pulito e pragmatico. Creato da sviluppatori esperti, si prende cura di gran parte del fastidio dello sviluppo Web, quindi puoi concentrarti sulla scrittura della tua app senza dover reinventare la ruota. È gratuito e open source.

Examples

Ciao mondo con Django

Fai un semplice esempio `Hello World` usando il tuo Django.

assicuriamoci prima di aver installato django sul tuo PC.

apri un terminale e digita: python -c "import django"
-> se non arriva nessun errore che significa che Django è già installato.

Ora creiamo un progetto in django. Per questo scrivi sotto il comando sul terminale:
django-admin startproject HelloWorld

Il comando sopra creerà una directory chiamata HelloWorld.
La struttura della directory sarà simile a:
Ciao mondo
| --helloworld
| | - **init** .py
| | --settings.py
| | --urls.py
| | --wsgi.py
| --manage.py

Scrittura di viste (riferimento dalla documentazione di django)

Una funzione di visualizzazione, o vista in breve, è semplicemente una funzione Python che accetta una richiesta Web e restituisce una risposta Web. Questa risposta può essere il contenuto HTML di una pagina Web o di qualsiasi cosa. La documentazione dice che possiamo scrivere funzioni di visualizzazione ovunque, ma è meglio scrivere in views.py nella nostra directory di progetto.

Ecco una vista che restituisce un messaggio ciao mondo. (Views.py)

```
from django.http import HttpResponse

define helloWorld(request):
```

```
    return HttpResponse("Hello World!! Django Welcomes You.")
```

capiamo il codice, passo dopo passo.

- Per prima cosa, importiamo la classe HttpResponse dal modulo django.http.
- Successivamente, definiamo una funzione chiamata helloWorld. Questa è la funzione di visualizzazione. Ogni funzione di visualizzazione accetta un oggetto HttpRequest come primo parametro, che di solito è denominato richiesta.

  Si noti che il nome della funzione di visualizzazione non ha importanza; non deve essere chiamato in un certo modo perché Django lo riconosca. l'abbiamo chiamato helloWorld qui, in modo che sia chiaro cosa fa.
- La vista restituisce un oggetto HttpResponse che contiene la risposta generata. Ogni funzione di visualizzazione è responsabile della restituzione di un oggetto HttpResponse.

Per maggiori informazioni sulle viste django clicca qui

Mappatura degli URL alle viste
Per visualizzare questa vista su un particolare URL, devi creare un URLconf;

Prima di ciò, comprendiamo come le richieste dei processi di django.

- Django determina il modulo URLconf di root da utilizzare.
- Django carica il modulo Python e cerca la variabile urlpatterns. Questo dovrebbe essere un elenco Python delle istanze di django.conf.urls.url ().
- Django scorre attraverso ogni pattern URL, in ordine, e si ferma sul primo che corrisponde all'URL richiesto.
- Una volta che una delle regex corrisponde, Django importa e chiama la vista data, che è una semplice funzione Python.

Ecco come il nostro URLconf si assomiglia:

```
from django.conf.urls import url
from . import views #import the views.py from current directory

urlpatterns = [
   url(r'^helloworld/$', views.helloWorld),
]
```

Per maggiori informazioni su django Urls clicca qui

Ora cambia la directory in HelloWorld e scrivi sotto il comando sul terminale.
python manage.py runserver

per impostazione predefinita il server verrà eseguito a 127.0.0.1:8000

Apri il browser e digita 127.0.0.1:8000/helloworld/. La pagina ti mostrerà "Ciao mondo !! Django ti dà il benvenuto".

Leggi Django online: https://riptutorial.com/it/python/topic/8994/django

Capitolo 51: eccezioni

introduzione

Gli errori rilevati durante l'esecuzione sono chiamati eccezioni e non sono incondizionatamente fatali. La maggior parte delle eccezioni non sono gestite dai programmi; è possibile scrivere programmi che gestiscono eccezioni selezionate. Ci sono funzioni specifiche in Python per gestire le eccezioni e la logica delle eccezioni. Inoltre, le eccezioni hanno una gerarchia di tipo ricco, che eredita dal tipo `BaseException` .

Sintassi

- aumentare l' *eccezione*
- raise # re-raise un'eccezione che è già stata sollevata
- genera *un'eccezione* dalla *causa* # Python 3 - imposta la causa dell'eccezione
- genera *un'eccezione* da None # Python 3 - sopprime tutto il contesto di eccezione
- provare:
- tranne *[tipi di eccezioni] [* come *identificatore]* :
- altro:
- finalmente:

Examples

Sollevare le eccezioni

Se il tuo codice rileva una condizione che non sa come gestire, come un parametro errato, dovrebbe sollevare l'eccezione appropriata.

```
def even_the_odds(odds):
    if odds % 2 != 1:
        raise ValueError("Did not get an odd number")

    return odds + 1
```

Cattura le eccezioni

Usa `try...except:` per rilevare eccezioni. Dovresti specificare un'eccezione precisa che puoi:

```
try:
    x = 5 / 0
except ZeroDivisionError as e:
    # `e` is the exception object
    print("Got a divide by zero! The exception was:", e)
    # handle exceptional case
    x = 0
finally:
```

```
    print "The END"
    # it runs no matter what execute.
```

La classe di eccezione specificata - in questo caso `ZeroDivisionError` - `ZeroDivisionError` qualsiasi eccezione che sia di quella classe o di qualsiasi sottoclasse di tale eccezione.

Ad esempio, `ZeroDivisionError` è una sottoclasse di `ArithmeticError` :

```
>>> ZeroDivisionError.__bases__
(<class 'ArithmeticError'>,)
```

E così, il seguente catturerà ancora l' `ZeroDivisionError` :

```
try:
    5 / 0
except ArithmeticError:
    print("Got arithmetic error")
```

Esecuzione del codice di pulizia con finalmente

A volte, è possibile che si verifichi che qualcosa si verifichi indipendentemente dall'eccezione che si è verificata, ad esempio, se è necessario ripulire alcune risorse.

La `finally` del blocco di un `try` clausola di accadrà indipendentemente dal fatto che le eccezioni sono state sollevate.

```
resource = allocate_some_expensive_resource()
try:
    do_stuff(resource)
except SomeException as e:
    log_error(e)
    raise  # re-raise the error
finally:
    free_expensive_resource(resource)
```

Questo schema è spesso gestito meglio con i gestori di contesto (usando l'istruzione `with`).

Risollevare le eccezioni

A volte si desidera rilevare un'eccezione solo per ispezionarla, ad esempio per scopi di registrazione. Dopo l'ispezione, si desidera che l'eccezione continui a propagarsi come prima.

In questo caso, usa semplicemente l'istruzione `raise` senza parametri.

```
try:
    5 / 0
except ZeroDivisionError:
    print("Got an error")
    raise
```

Tieni presente, tuttavia, che qualcuno più in alto nella pila dei chiamanti può ancora cogliere

l'eccezione e gestirla in qualche modo. Il risultato finale potrebbe essere un fastidio in questo caso perché accadrà in ogni caso (catturato o non catturato). Quindi potrebbe essere un'idea migliore sollevare un'eccezione diversa, contenente il tuo commento sulla situazione e l'eccezione originale:

```
try:
    5 / 0
except ZeroDivisionError as e:
    raise ZeroDivisionError("Got an error", e)
```

Ma questo ha l'inconveniente di ridurre la traccia di eccezioni esattamente a questo `raise` mentre il `raise` senza argomento mantiene la traccia di eccezione originale.

In Python 3 puoi mantenere lo stack originale usando la sintassi `raise` - `from` :

```
    raise ZeroDivisionError("Got an error") from e
```

Eccezioni a catena con rilancio da

Durante il processo di gestione di un'eccezione, potresti sollevare un'altra eccezione. Ad esempio, se si ottiene un `IOError` durante la lettura da un file, è possibile che si desideri sollevare un errore specifico dell'applicazione da presentare agli utenti della libreria.

Python 3.x 3.0

È possibile concatenare le eccezioni per mostrare come è stata gestita la gestione delle eccezioni:

```
>>> try:
    5 / 0
except ZeroDivisionError as e:
    raise ValueError("Division failed") from e

Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: division by zero

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
ValueError: Division failed
```

Gerarchia delle eccezioni

La gestione delle eccezioni si verifica in base a una gerarchia di eccezioni, determinata dalla struttura di ereditarietà delle classi di eccezioni.

Ad esempio, `IOError` e `OSError` sono entrambe sottoclassi di `EnvironmentError` . Il codice che cattura un `IOError` non cattura un `OSError` . Tuttavia, il codice che cattura un `EnvironmentError` catturerà sia `IOError` che `OSError` .

La gerarchia delle eccezioni built-in:

Python 2.x 2.3

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StandardError
      |    +-- BufferError
      |    +-- ArithmeticError
      |    |    +-- FloatingPointError
      |    |    +-- OverflowError
      |    |    +-- ZeroDivisionError
      |    +-- AssertionError
      |    +-- AttributeError
      |    +-- EnvironmentError
      |    |    +-- IOError
      |    |    +-- OSError
      |    |         +-- WindowsError (Windows)
      |    |         +-- VMSError (VMS)
      |    +-- EOFError
      |    +-- ImportError
      |    +-- LookupError
      |    |    +-- IndexError
      |    |    +-- KeyError
      |    +-- MemoryError
      |    +-- NameError
      |    |    +-- UnboundLocalError
      |    +-- ReferenceError
      |    +-- RuntimeError
      |    |    +-- NotImplementedError
      |    +-- SyntaxError
      |    |    +-- IndentationError
      |    |         +-- TabError
      |    +-- SystemError
      |    +-- TypeError
      |    +-- ValueError
      |         +-- UnicodeError
      |              +-- UnicodeDecodeError
      |              +-- UnicodeEncodeError
      |              +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
       +-- ImportWarning
       +-- UnicodeWarning
       +-- BytesWarning
```

Python 3.x 3.0

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
```

```
+-- GeneratorExit
+-- Exception
     +-- StopIteration
     +-- StopAsyncIteration
     +-- ArithmeticError
     |    +-- FloatingPointError
     |    +-- OverflowError
     |    +-- ZeroDivisionError
     +-- AssertionError
     +-- AttributeError
     +-- BufferError
     +-- EOFError
     +-- ImportError
     +-- LookupError
     |    +-- IndexError
     |    +-- KeyError
     +-- MemoryError
     +-- NameError
     |    +-- UnboundLocalError
     +-- OSError
     |    +-- BlockingIOError
     |    +-- ChildProcessError
     |    +-- ConnectionError
     |    |    +-- BrokenPipeError
     |    |    +-- ConnectionAbortedError
     |    |    +-- ConnectionRefusedError
     |    |    +-- ConnectionResetError
     |    +-- FileExistsError
     |    +-- FileNotFoundError
     |    +-- InterruptedError
     |    +-- IsADirectoryError
     |    +-- NotADirectoryError
     |    +-- PermissionError
     |    +-- ProcessLookupError
     |    +-- TimeoutError
     +-- ReferenceError
     +-- RuntimeError
     |    +-- NotImplementedError
     |    +-- RecursionError
     +-- SyntaxError
     |    +-- IndentationError
     |         +-- TabError
     +-- SystemError
     +-- TypeError
     +-- ValueError
     |    +-- UnicodeError
     |         +-- UnicodeDecodeError
     |         +-- UnicodeEncodeError
     |         +-- UnicodeTranslateError
     +-- Warning
          +-- DeprecationWarning
          +-- PendingDeprecationWarning
          +-- RuntimeWarning
          +-- SyntaxWarning
          +-- UserWarning
          +-- FutureWarning
          +-- ImportWarning
          +-- UnicodeWarning
          +-- BytesWarning
          +-- ResourceWarning
```

Le eccezioni sono anche gli oggetti

Le eccezioni sono solo oggetti Python regolari che ereditano dalla `BaseException` . Uno script Python può usare l'istruzione `raise` per interrompere l'esecuzione, facendo in modo che Python stampi una traccia dello stack dello stack di chiamate in quel punto e una rappresentazione dell'istanza di eccezione. Per esempio:

```
>>> def failing_function():
...     raise ValueError('Example error!')
>>> failing_function()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in failing_function
ValueError: Example error!
```

che dice che un `'Example error!' ValueError` con il messaggio `'Example error!'` è stato sollevato dal nostro `failing_function()` , che è stato eseguito nell'interprete.

Il codice chiamante può scegliere di gestire tutti i tipi di eccezioni che possono essere sollevate da una chiamata:

```
>>> try:
...     failing_function()
... except ValueError:
...     print('Handled the error')
Handled the error
```

È possibile ottenere gli oggetti eccezione assegnandoli nella parte `except...` del codice di gestione delle eccezioni:

```
>>> try:
...     failing_function()
... except ValueError as e:
...     print('Caught exception', repr(e))
Caught exception ValueError('Example error!',)
```

Un elenco completo delle eccezioni Python integrate con le loro descrizioni può essere trovato nella documentazione di Python: https://docs.python.org/3.5/library/exceptions.html . Ed ecco l'elenco completo organizzato gerarchicamente: Gerarchia delle eccezioni .

Creazione di tipi di eccezioni personalizzate

Creare una classe che eredita da `Exception` :

```
class FooException(Exception):
    pass
try:
    raise FooException("insert description here")
except FooException:
    print("A FooException was raised.")
```

o un altro tipo di eccezione:

```
class NegativeError(ValueError):
      pass

def foo(x):
    # function that only accepts positive values of x
    if x < 0:
        raise NegativeError("Cannot process negative numbers")
    ...  # rest of function body
try:
    result = foo(int(input("Enter a positive integer: ")))  # raw_input in Python 2.x
except NegativeError:
    print("You entered a negative number!")
else:
    print("The result was " + str(result))
```

Non prendere tutto!

Mentre è spesso allettante catturare ogni `Exception` :

```
try:
    very_difficult_function()
except Exception:
    # log / try to reconnect / exit gratiously
finally:
    print "The END"
    # it runs no matter what execute.
```

O anche tutto (incluso `BaseException` e tutti i suoi figli, inclusa l' `Exception`):

```
try:
    even_more_difficult_function()
except:
    pass  # do whatever needed
```

Nella maggior parte dei casi è una cattiva pratica. Potrebbe prendere più del previsto, come `SystemExit` , `KeyboardInterrupt` e `MemoryError` , ognuno dei quali dovrebbe essere gestito in modo diverso dal solito sistema o errori logici. Significa anche che non c'è una chiara comprensione di ciò che il codice interno potrebbe fare male e come recuperare correttamente da quella condizione. Se stai rilevando ogni errore, non sai che errore si è verificato o come risolverlo.

Questo è più comunemente indicato come "bug masking" e dovrebbe essere evitato. Lascia che il tuo programma si blocchi, invece di fallire silenziosamente o, peggio ancora, fallire a un livello più profondo di esecuzione. (Immagina che sia un sistema transazionale)

Di solito questi costrutti sono usati al livello più esterno del programma e registreranno i dettagli dell'errore in modo che il bug possa essere corretto, o l'errore può essere gestito in modo più specifico.

Cattura più eccezioni

Ci sono alcuni modi per catturare più eccezioni .

Il primo è creando una tupla dei tipi di eccezioni che si desidera catturare e gestire allo stesso modo. Questo esempio farà sì che il codice ignori le eccezioni `KeyError` e `AttributeError` .

```
try:
    d = {}
    a = d[1]
    b = d.non_existing_field
except (KeyError, AttributeError) as e:
    print("A KeyError or an AttributeError exception has been caught.")
```

Se desideri gestire diverse eccezioni in modi diversi, puoi fornire un blocco di eccezioni separato per ogni tipo. In questo esempio, rileviamo ancora `KeyError` e `AttributeError` , ma gestiamo le eccezioni in modi diversi.

```
try:
    d = {}
    a = d[1]
    b = d.non_existing_field
except KeyError as e:
    print("A KeyError has occurred. Exception message:", e)
except AttributeError as e:
    print("An AttributeError has occurred. Exception message:", e)
```

Esempi pratici di gestione delle eccezioni

Input dell'utente

Immagina di voler che un utente inserisca un numero tramite `input` . Vuoi assicurarti che l'input sia un numero. Puoi usare `try` / `except` per questo:

Python 3.x 3.0

```
while True:
    try:
        nb = int(input('Enter a number: '))
        break
    except ValueError:
        print('This is not a number, try again.')
```

Nota: Python 2.x `raw_input` invece `raw_input` ; la funzione `input` esiste in Python 2.x ma ha semantica diversa. Nell'esempio sopra, l' `input` accetta anche espressioni come `2 + 2` che valutano un numero.

Se l'input non può essere convertito in un numero intero, viene sollevata `ValueError` . Puoi prenderlo con l' `except` . Se non viene sollevata alcuna eccezione, `break` salti fuori dal ciclo. Dopo il ciclo, `nb` contiene un numero intero.

dizionari

Immaginate effettua l'iterazione di un elenco di numeri interi consecutivi, come `range(n)` , e si dispone di un elenco di dizionari `d` che contiene le informazioni sulle cose da fare quando si incontrano alcuni interi particolari, dici *saltare il* `d[i]` *i prossimi.*

```
d = [{7: 3}, {25: 9}, {38: 5}]

for i in range(len(d)):
    do_stuff(i)
    try:
       dic = d[i]
       i += dic[i]
    except KeyError:
       i += 1
```

Un `KeyError` verrà generato quando si tenta di ottenere un valore da un dizionario per una chiave che non esiste.

Altro

Il codice in un altro blocco verrà eseguito solo se non sono state sollevate eccezioni dal codice nel blocco `try` . Questo è utile se hai un codice che non vuoi eseguire se viene lanciata un'eccezione, ma non vuoi che vengano catturate le eccezioni generate da quel codice.

Per esempio:

```
try:
    data = {1: 'one', 2: 'two'}
    print(data[1])
except KeyError as e:
    print('key not found')
else:
    raise ValueError()
# Output: one
# Output: ValueError
```

Nota che questo tipo di `else:` non può essere combinato con un `if` avvia la clausola else in un `elif` . Se si dispone di un seguito `if` ha bisogno per rimanere rientrato sotto che `else:` :

```
try:
    ...
except ...:
    ...
else:
    if ...:
        ...
    elif ...:
        ...
    else:
        ...
```

Leggi eccezioni online: https://riptutorial.com/it/python/topic/1788/eccezioni

Capitolo 52: Eccezioni del Commonwealth

introduzione

Qui in Stack Overflow spesso vediamo duplicati che parlano degli stessi errori: `"ImportError: No module named '??????'`, `SyntaxError: invalid syntax` o `NameError: name '???' is not defined` tratta di uno sforzo per ridurli e avere una documentazione a cui collegarsi.

Examples

IndentazioneErrori (o indentazione SintassiErrori)

Nella maggior parte delle altre lingue il rientro non è obbligatorio, ma in Python (e in altre lingue: prime versioni di FORTRAN, Makefile, Whitespace (linguaggio esoterico), ecc.) Non è così, cosa può essere fonte di confusione se vieni da un'altra lingua, se stavi copiando il codice da un esempio al tuo, o semplicemente se sei nuovo.

IndentationError / SyntaxError: indent inaspettato

Questa eccezione viene sollevata quando il livello di indentazione aumenta senza motivo.

Esempio

Non c'è motivo di aumentare il livello qui:

Python 2.x 2.0 2.7

```
print "This line is ok"
    print "This line isn't ok"
```

Python 3.x 3.0

```
print("This line is ok")
    print("This line isn't ok")
```

Qui ci sono due errori: l'ultimo e che il rientro non corrisponde a nessun livello di indentazione. Tuttavia ne viene mostrato solo uno:

Python 2.x 2.0 2.7

```
print "This line is ok"
 print "This line isn't ok"
```

Python 3.x 3.0

```
 print("This line is ok")
  print("This line isn't ok")
```

IndentationError / SyntaxError: unindent non corrisponde a nessun livello di indentazione esterno

Sembra che tu non abbia distrutto completamente.

Esempio

Python 2.x 2.0 2.7

```
def foo():
    print "This should be part of foo()"
   print "ERROR!"
print "This is not a part of foo()"
```

Python 3.x 3.0

```
 print("This line is ok")
  print("This line isn't ok")
```

IndentationError: previsto un blocco rientrato

Dopo i due punti (e poi una nuova riga) il livello di indentazione deve aumentare. Questo errore viene generato quando ciò non è accaduto.

Esempio

```
if ok:
doStuff()
```

Nota : usa la parola chiave `pass` (che non fa assolutamente nulla) per mettere un `if` , `else` , `except` , `class` , `method` o `definition` ma non dire cosa succederà se chiamato / condizione è vera (ma fallo più tardi, o nel caso di `except` : non fare niente):

```
def foo():
    pass
```

IndentationError: uso incoerente di tabulazioni e spazi in indentazione

Esempio

```
def foo():
    if ok:
      return "Two != Four != Tab"
        return "i dont care i do whatever i want"
```

Come evitare questo errore

Non usare le schede. È scoraggiato da `PEP8` , la guida di stile per Python.

1. Imposta il tuo editor per utilizzare 4 **spazi** per il rientro.
2. Fai una ricerca e sostituisci per sostituire tutte le schede con 4 spazi.
3. Assicurati che il tuo editor sia impostato per **visualizzare le** schede come 8 spazi, in modo che tu possa facilmente realizzare quell'errore e correggerlo.

Vedi questa domanda se vuoi saperne di più.

TypeErrors

Queste eccezioni sono causate quando il tipo di alcuni oggetti dovrebbe essere diverso

TypeError: [definizione / metodo] richiede? argomenti posizionali ma? Venne dato

Una funzione o un metodo è stato chiamato con più (o meno) argomenti rispetto a quelli che può accettare.

Esempio

Se vengono forniti più argomenti:

```
def foo(a): return a
foo(a,b,c,d) #And a,b,c,d are defined
```

Se vengono forniti meno argomenti:

```
def foo(a,b,c,d): return a += b + c + d
```

```
foo(a) #And a is defined
```

Nota : se si desidera utilizzare un numero sconosciuto di argomenti, è possibile utilizzare `*args` o `**kwargs` . Vedi * args e ** kwargs

---

TypeError: tipi di operando non supportati per [operando]: '???' e '???'

Alcuni tipi non possono essere gestiti insieme, a seconda dell'operando.

Esempio

Ad esempio: `+` è usato per concatenare e aggiungere, ma non puoi usarne nessuno per entrambi i tipi. Ad esempio, provare a creare un `set` concatenando (`+` ing) `'set1'` e `'tuple1'` restituisce l'errore. Codice:

```
set1, tuple1 = {1,2}, (3,4)
a = set1 + tuple1
```

Alcuni tipi (es .: `int` e `string`) usano entrambi `+` ma per cose diverse:

```
b = 400 + 'foo'
```

Oppure potrebbero non essere nemmeno usati per nulla:

```
c = ["a","b"] - [1,2]
```

Ma puoi ad esempio aggiungere un `float` a un `int` :

```
d = 1 + 1.0
```

TypeError: '???' l'oggetto non è iterable / subscriptable:

Perché un oggetto sia iterabile può prendere indici sequenziali a partire da zero fino a quando gli indici non sono più validi e viene sollevato un `IndexError` (Più tecnicamente: deve avere un metodo `__iter__` che restituisce un `__iterator__` , o che definisce un metodo `__getitem__` che fa ciò che è stato precedentemente menzionato).

Esempio

Qui stiamo dicendo che la `bar` è l'oggetto zeroth di 1. Sciocchezze:

```
foo = 1
bar = foo[0]
```

Questa è una versione più discreta: In questo esempio `for` tentativi per impostare `x` per `amount[0]` , il primo elemento in un iterabile ma non può perché quantità è un int:

```
amount = 10
for x in amount: print(x)
```

TypeError: '???' l'oggetto non è chiamabile

Stai definendo una variabile e chiamandola in un secondo momento (come quello che fai con una funzione o un metodo)

Esempio

```
foo = "notAFunction"
foo()
```

NameError: name '???' non è definito

Viene generato quando si tenta di utilizzare una variabile, un metodo o una funzione non inizializzata (almeno non prima). In altre parole, viene generato quando non viene trovato un nome locale o globale richiesto. È possibile che tu abbia sbagliato il nome dell'oggetto o dimenticato di `import` qualcosa. Inoltre forse è in un altro ambito. Copriremo quelli con esempi separati.

Semplicemente non è definito da nessuna parte nel codice

È possibile che tu abbia dimenticato di inizializzarlo, specialmente se è una costante

```
foo   # This variable is not defined
bar() # This function is not defined
```

Forse è definito più tardi:

```
baz()

def baz():
    pass
```

O non è stato `import` :

```
#needs import math

def sqrt():
    x = float(input("Value: "))
    return math.sqrt(x)
```

Scopi Python e la regola LEGB:

La cosiddetta regola LEGB parla degli ambiti Python. Il suo nome si basa sui diversi ambiti, ordinati dalle priorità corrispondenti:

```
Local → Enclosed → Global → Built-in.
```

- **L** ocal: variabili non dichiarate globali o assegnate in una funzione.
- **E** nclosing: variabili definite in una funzione che è racchiusa in un'altra funzione.
- **G** lobal: variabili dichiarate globali o assegnate al livello superiore di un file.
- **B** uilt-in: variabili preassegnate nel modulo nomi incorporato.

Come esempio:

```
for i in range(4):
    d = i * 2
print(d)
```

`d` è accessibile perché il ciclo `for` non segna un nuovo scope, ma se lo facesse avremmo un errore e il suo comportamento sarebbe simile a:

```
def noaccess():
    for i in range(4):
        d = i * 2
noaccess()
print(d)
```

Python dice `NameError: name 'd' is not defined`

Altri errori

AssertError

`assert` dichiarazione esiste in quasi ogni linguaggio di programmazione. Quando lo fai:

```
assert condition
```

o:

```
assert condition, message
```

È equivalente a questo:

```
if __debug__:
    if not condition: raise AssertionError(message)
```

Le asserzioni possono includere un messaggio opzionale e puoi disabilitarle quando hai finito il debug.

Nota : il **debug della** variabile integrato è True in circostanze normali, False quando viene richiesta l'ottimizzazione (opzione riga di comando -O). Le assegnazioni di **debug** sono illegali. Il valore per la variabile built-in viene determinato all'avvio dell'interprete.

KeyboardInterrupt

Errore generato quando l'utente preme il tasto interrupt, normalmente `Ctrl` + `C` o `del` .

ZeroDivisionError

Hai provato a calcolare `1/0` che non è definito. Vedi questo esempio per trovare i divisori di un numero:

Python 2.x 2.0 2.7

```
div = float(raw_input("Divisors of: "))
for x in xrange(div+1): #includes the number itself and zero
    if div/x == div//x:
        print x, "is a divisor of", div
```

Python 3.x 3.0

```
div = int(input("Divisors of: "))
for x in range(div+1): #includes the number itself and zero
    if div/x == div//x:
        print(x, "is a divisor of", div)
```

Solleva `ZeroDivisionError` perché il ciclo `for` assegna quel valore a `x` . Invece dovrebbe essere:

Python 2.x 2.0 2.7

```
div = float(raw_input("Divisors of: "))
for x in xrange(1,div+1): #includes the number itself but not zero
    if div/x == div//x:
        print x, "is a divisor of", div
```

Python 3.x 3.0

```
div = int(input("Divisors of: "))
for x in range(1,div+1): #includes the number itself but not zero
    if div/x == div//x:
        print(x, "is a divisor of", div)
```

Errore di sintassi su buon codice

La maggior parte delle volte un SyntaxError che punta a una linea non interessante significa che c'è un problema sulla linea (in questo esempio, è una parentesi mancante):

```
def my_print():
    x = (1 + 1
    print(x)
```

ritorna

```
  File "<input>", line 3
    print(x)
        ^
SyntaxError: invalid syntax
```

Il motivo più comune di questo problema è parentesi / parentesi non corrispondenti, come mostra l'esempio.

C'è un avvertimento importante per le dichiarazioni di stampa in Python 3:

Python 3.x 3.0

```
 >>> print "hello world"
  File "<stdin>", line 1
    print "hello world"
                      ^
SyntaxError: invalid syntax
```

Perché la dichiarazione di `print` stata sostituita con la funzione `print()` , quindi si desidera:

```
print("hello world")  # Note this is valid for both Py2 & Py3
```

Leggi Eccezioni del Commonwealth online: https://riptutorial.com/it/python/topic/9300/eccezioni-del-commonwealth

Capitolo 53: Elenco

introduzione

Python **List** è una struttura di dati generale ampiamente utilizzata nei programmi Python. Si trovano in altre lingue, spesso definite *matrici dinamiche* . Sono entrambi *mutabili* e un tipo di dati di *sequenza* che consente loro di essere *indicizzati* e *tagliati* . L'elenco può contenere diversi tipi di oggetti, inclusi altri oggetti elenco.

Sintassi

- [valore, valore, ...]
- lista ([iterable])

Osservazioni

`list` è un particolare tipo di iterabile, ma non è l'unico che esiste in Python. A volte è preferibile utilizzare `set` , `tuple` o `dictionary`

`list` è il nome dato in Python agli array dinamici (simile al `vector<void*>` da C ++ o `ArrayList<Object>` di Java `ArrayList<Object>`). Non è una lista collegata.

L'accesso agli elementi avviene in tempo costante ed è molto veloce. L'aggiunta di elementi alla fine dell'elenco viene ammortizzata in tempo costante, ma una volta ogni tanto può comportare l'allocazione e la copia dell'intera `list` .

La comprensione delle liste è legata alle liste.

Examples

Accedere ai valori di lista

Gli elenchi Python sono indicizzati a zero e si comportano come matrici in altre lingue.

```
lst = [1, 2, 3, 4]
lst[0]  # 1
lst[1]  # 2
```

Il tentativo di accedere a un indice al di fuori dei limiti dell'elenco `IndexError` un `IndexError` .

```
lst[4]  # IndexError: list index out of range
```

Gli indici negativi sono interpretati come conteggi dalla *fine* della lista.

```
lst[-1]  # 4
```

```
lst[-2]  # 3
lst[-5]  # IndexError: list index out of range
```

Questo è funzionalmente equivalente a

```
lst[len(lst)-1]  # 4
```

Gli elenchi consentono di utilizzare la *notazione delle* `lst[start:end:step]` come `lst[start:end:step]` . L'output della notazione di sezione è un nuovo elenco contenente elementi dall'indice `start` a `end-1` . Se le opzioni vengono omessi `start` default è all'inizio della lista, `end` alla fine della lista e `step` a 1:

```
lst[1:]      # [2, 3, 4]
lst[:3]      # [1, 2, 3]
lst[::2]     # [1, 3]
lst[::-1]    # [4, 3, 2, 1]
lst[-1:0:-1] # [4, 3, 2]
lst[5:8]     # [] since starting index is greater than length of lst, returns empty list
lst[1:10]    # [2, 3, 4] same as omitting ending index
```

Con questo in mente, puoi stampare una versione invertita della lista chiamando

```
lst[::-1]    # [4, 3, 2, 1]
```

Quando si utilizzano lunghezze di passi di importi negativi, l'indice di partenza deve essere maggiore dell'indice finale altrimenti il risultato sarà una lista vuota.

```
lst[3:1:-1] # [4, 3]
```

L'utilizzo degli indici step negativi è equivalente al seguente codice:

```
reversed(lst)[0:2] # 0 = 1 -1
                   # 2 = 3 -1
```

Gli indici utilizzati sono 1 in meno di quelli utilizzati nell'indicizzazione negativa e sono invertiti.

Affettatura avanzata

Quando gli elenchi vengono `__getitem__()` viene chiamato il metodo `__getitem__()` dell'oggetto elenco, con un oggetto `slice` . Python ha un metodo slice incorporato per generare oggetti slice. Possiamo usarlo per *memorizzare* una sezione e riutilizzarla in un secondo momento in questo modo,

```
data = 'chandan purohit    22 2000'  #assuming data fields of fixed length
name_slice = slice(0,19)
age_slice = slice(19,21)
salary_slice = slice(22,None)

#now we can have more readable slices
print(data[name_slice]) #chandan purohit
```

```
print(data[age_slice]) #'22'
print(data[salary_slice]) #'2000'
```

Questo può essere di grande utilità fornendo funzionalità di slicing ai nostri oggetti sovrascrivendo `__getitem__` nella nostra classe.

Elenca metodi e operatori supportati

A partire da una lista data `a` :

```
a = [1, 2, 3, 4, 5]
```

1. `append(value)` - aggiunge un nuovo elemento alla fine dell'elenco.

   ```
   # Append values 6, 7, and 7 to the list
   a.append(6)
   a.append(7)
   a.append(7)
   # a: [1, 2, 3, 4, 5, 6, 7, 7]

   # Append another list
   b = [8, 9]
   a.append(b)
   # a: [1, 2, 3, 4, 5, 6, 7, 7, [8, 9]]

   # Append an element of a different type, as list elements do not need to have the same
   type
   my_string = "hello world"
   a.append(my_string)
   # a: [1, 2, 3, 4, 5, 6, 7, 7, [8, 9], "hello world"]
   ```

   Si noti che il metodo `append()` aggiunge solo un nuovo elemento alla fine dell'elenco. Se si aggiunge un elenco a un altro elenco, l'elenco che si aggiunge diventa un singolo elemento alla fine del primo elenco.

   ```
   # Appending a list to another list
   a = [1, 2, 3, 4, 5, 6, 7, 7]
   b = [8, 9]
   a.append(b)
   # a: [1, 2, 3, 4, 5, 6, 7, 7, [8, 9]]
   a[8]
   # Returns: [8,9]
   ```

2. `extend(enumerable)` : estende l'elenco aggiungendo elementi da un'altra enumerabile.

   ```
   a = [1, 2, 3, 4, 5, 6, 7, 7]
   b = [8, 9, 10]

   # Extend list by appending all elements from b
   a.extend(b)
   # a: [1, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10]

   # Extend list with elements from a non-list enumerable:
   ```

```
a.extend(range(3))
# a: [1, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10, 0, 1, 2]
```

Gli elenchi possono anche essere concatenati con l'operatore `+` . Nota che questo non modifica nessuno degli elenchi originali:

```
a = [1, 2, 3, 4, 5, 6] + [7, 7] + b
# a: [1, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10]
```

3. `index(value, [startIndex])` - ottiene l'indice della prima occorrenza del valore di input. Se il valore di input non è presente nell'elenco, viene sollevata un'eccezione `ValueError` . Se viene fornito un secondo argomento, la ricerca viene avviata a quell'indice specificato.

```
a.index(7)
# Returns: 6

a.index(49) # ValueError, because 49 is not in a.

a.index(7, 7)
# Returns: 7

a.index(7, 8) # ValueError, because there is no 7 starting at index 8
```

4. `insert(index, value)` - inserisce il `value` appena prima `index` specificato. Quindi, dopo l'inserimento, il nuovo elemento occupa l' `index` posizione.

```
a.insert(0, 0)  # insert 0 at position 0
a.insert(2, 5)  # insert 5 at position 2
# a: [0, 1, 5, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10]
```

5. `pop([index])` - rimuove e restituisce l'oggetto `index` . Senza argomenti rimuove e restituisce l'ultimo elemento della lista.

```
a.pop(2)
# Returns: 5
# a: [0, 1, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10]
a.pop(8)
# Returns: 7
# a: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# With no argument:
a.pop()
# Returns: 10
# a: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

6. `remove(value)` - rimuove la prima occorrenza del valore specificato. Se il valore fornito non può essere trovato, viene sollevata `ValueError` .

```
a.remove(0)
a.remove(9)
# a: [1, 2, 3, 4, 5, 6, 7, 8]
a.remove(10)
```

```
# ValueError, because 10 is not in a
```

7. `reverse()` - inverte l'elenco sul posto e restituisce `None` .

```
a.reverse()
# a: [8, 7, 6, 5, 4, 3, 2, 1]
```

Ci sono anche altri modi per invertire una lista .

8. `count(value)` : conta il numero di occorrenze di alcuni valori nell'elenco.

```
a.count(7)
# Returns: 2
```

9. `sort()` - ordina l'elenco in ordine numerico e lessicografico e restituisce `None` .

```
a.sort()
# a = [1, 2, 3, 4, 5, 6, 7, 8]
# Sorts the list in numerical order
```

Le liste possono anche essere invertite se ordinate usando il `reverse=True` flag nel metodo `sort()` .

```
a.sort(reverse=True)
# a = [8, 7, 6, 5, 4, 3, 2, 1]
```

Se si desidera ordinare per attributi di elementi, è possibile utilizzare l'argomento della parola `key` :

```
import datetime

class Person(object):
    def __init__(self, name, birthday, height):
        self.name = name
        self.birthday = birthday
        self.height = height

    def __repr__(self):
        return self.name

l = [Person("John Cena", datetime.date(1992, 9, 12), 175),
     Person("Chuck Norris", datetime.date(1990, 8, 28), 180),
     Person("Jon Skeet", datetime.date(1991, 7, 6), 185)]

l.sort(key=lambda item: item.name)
# l: [Chuck Norris, John Cena, Jon Skeet]

l.sort(key=lambda item: item.birthday)
# l: [Chuck Norris, Jon Skeet, John Cena]

l.sort(key=lambda item: item.height)
# l: [John Cena, Chuck Norris, Jon Skeet]
```

In caso di lista di dicts il concetto è lo stesso:

```
import datetime

l = [{'name':'John Cena', 'birthday': datetime.date(1992, 9, 12),'height': 175},
 {'name': 'Chuck Norris', 'birthday': datetime.date(1990, 8, 28),'height': 180},
 {'name': 'Jon Skeet', 'birthday': datetime.date(1991, 7, 6), 'height': 185}]

l.sort(key=lambda item: item['name'])
# l: [Chuck Norris, John Cena, Jon Skeet]

l.sort(key=lambda item: item['birthday'])
# l: [Chuck Norris, Jon Skeet, John Cena]

l.sort(key=lambda item: item['height'])
# l: [John Cena, Chuck Norris, Jon Skeet]
```

Ordina per sottotit:

```
import datetime

l = [{'name':'John Cena', 'birthday': datetime.date(1992, 9, 12),'size': {'height': 175,
'weight': 100}},
 {'name': 'Chuck Norris', 'birthday': datetime.date(1990, 8, 28),'size' : {'height': 180,
'weight': 90}},
 {'name': 'Jon Skeet', 'birthday': datetime.date(1991, 7, 6), 'size': {'height': 185,
'weight': 110}}]

l.sort(key=lambda item: item['size']['height'])
# l: [John Cena, Chuck Norris, Jon Skeet]
```

Un modo migliore per ordinare usando `attrgetter` e `itemgetter`

Le liste possono anche essere ordinate usando le funzioni `attrgetter` e `itemgetter` dal modulo operatore. Questi possono aiutare a migliorare la leggibilità e la riusabilità. Ecco alcuni esempi,

```
from operator import itemgetter,attrgetter

people = [{'name':'chandan','age':20,'salary':2000},
          {'name':'chetan','age':18,'salary':5000},
          {'name':'guru','age':30,'salary':3000}]
by_age = itemgetter('age')
by_salary = itemgetter('salary')

people.sort(key=by_age) #in-place sorting by age
people.sort(key=by_salary) #in-place sorting by salary
```

`itemgetter` può anche avere un indice. Questo è utile se vuoi ordinare in base agli indici di una tupla.

```
list_of_tuples = [(1,2), (3,4), (5,0)]
list_of_tuples.sort(key=itemgetter(1))
print(list_of_tuples) #[(5, 0), (1, 2), (3, 4)]
```

Utilizzare l' `attrgetter` se si desidera ordinare per attributi di un oggetto,

```
persons = [Person("John Cena", datetime.date(1992, 9, 12), 175),
           Person("Chuck Norris", datetime.date(1990, 8, 28), 180),
           Person("Jon Skeet", datetime.date(1991, 7, 6), 185)] #reusing Person class from
above example

person.sort(key=attrgetter('name')) #sort by name
by_birthday = attrgetter('birthday')
person.sort(key=by_birthday) #sort by birthday
```

10. `clear()` - rimuove tutti gli elementi dall'elenco

    ```
    a.clear()
    # a = []
    ```

11. **Replicazione** : moltiplicando una lista esistente per un intero, si otterrà una lista più ampia consistente in tante copie dell'originale. Questo può essere utile ad esempio per l'inizializzazione della lista:

    ```
    b = ["blah"] * 3
    # b = ["blah", "blah", "blah"]
    b = [1, 3, 5] * 5
    # [1, 3, 5, 1, 3, 5, 1, 3, 5, 1, 3, 5, 1, 3, 5]
    ```

    Fai attenzione a questo se il tuo elenco contiene riferimenti ad oggetti (ad esempio un elenco di elenchi), vedi Errori comuni - Moltiplicazione elenco e riferimenti comuni .

12. **Cancellazione elemento** : è possibile eliminare più elementi nell'elenco utilizzando la parola chiave `del` e la notazione di sezione:

    ```
    a = list(range(10))
    del a[::2]
    # a = [1, 3, 5, 7, 9]
    del a[-1]
    # a = [1, 3, 5, 7]
    del a[:]
    # a = []
    ```

13. **copiatura**

    L'assegnazione di default "=" assegna un riferimento della lista originale al nuovo nome. Cioè, il nome originale e il nuovo nome puntano entrambi allo stesso oggetto elenco. Le modifiche apportate tramite una di esse si rifletteranno in un'altra. Questo spesso non è ciò che intendevi.

    ```
    b = a
    a.append(6)
    # b: [1, 2, 3, 4, 5, 6]
    ```

    Se vuoi creare una copia della lista hai sotto le opzioni.

    Puoi tagliarlo:

```
new_list = old_list[:]
```

Puoi usare la funzione built-in list ():

```
new_list = list(old_list)
```

Puoi usare generici copy.copy ():

```
import copy
new_list = copy.copy(old_list) #inserts references to the objects found in the original.
```

Questo è un po 'più lento di list () perché deve prima trovare il datatype di old_list.

Se la lista contiene oggetti e vuoi copiarli, usa generico copy.deepcopy ():

```
import copy
new_list = copy.deepcopy(old_list) #inserts copies of the objects found in the original.
```

Ovviamente il metodo più lento e che richiede più memoria, ma a volte inevitabile.

Python 3.x 3.0

`copy()` - Restituisce una copia superficiale dell'elenco

```
    aa = a.copy()
    # aa = [1, 2, 3, 4, 5]
```

Lunghezza di una lista

Usa `len()` per ottenere la lunghezza unidimensionale di una lista.

```
len(['one', 'two'])  # returns 2

len(['one', [2, 3], 'four'])  # returns 3, not 4
```

`len()` funziona anche su stringhe, dizionari e altre strutture di dati simili alle liste.

Nota che `len()` è una funzione built-in, non un metodo di un oggetto lista.

Si noti inoltre che il costo di `len()` è `O(1)` , il che significa che richiederà la stessa quantità di tempo per ottenere la lunghezza di un elenco indipendentemente dalla sua lunghezza.

Iterare su una lista

Python supporta l'uso di un ciclo `for` direttamente su un elenco:

```
my_list = ['foo', 'bar', 'baz']
for item in my_list:
    print(item)
```

```
# Output: foo
# Output: bar
# Output: baz
```

Puoi anche ottenere la posizione di ogni oggetto allo stesso tempo:

```
for (index, item) in enumerate(my_list):
    print('The item in position {} is: {}'.format(index, item))

# Output: The item in position 0 is: foo
# Output: The item in position 1 is: bar
# Output: The item in position 2 is: baz
```

L'altro modo di iterare un elenco in base al valore dell'indice:

```
for i in range(0,len(my_list)):
    print(my_list[i])
#output:
>>>
foo
bar
baz
```

Tieni presente che la modifica di elementi in un elenco durante l'iterazione potrebbe avere risultati imprevisti:

```
for item in my_list:
    if item == 'foo':
        del my_list[0]
    print(item)

# Output: foo
# Output: baz
```

In questo ultimo esempio, abbiamo cancellato il primo elemento alla prima iterazione, ma quello ha causato l' `bar` della `bar` .

Verifica se un elemento è in una lista

Python rende molto semplice controllare se un elemento è in una lista. Basta usare l' `in` dell'operatore.

```
lst = ['test', 'twest', 'tweast', 'treast']

'test' in lst
# Out: True

'toast' in lst
# Out: False
```

> Nota: l' `in` operatore insiemi è asintoticamente più veloce che su liste. Se è necessario utilizzarlo più volte su elenchi potenzialmente di grandi dimensioni, è possibile

convertire l' `list` in un `set` e verificare la presenza di elementi sul `set` .

```
slst = set(lst)
'test' in slst
# Out: True
```

Invertire gli elementi dell'elenco

È possibile utilizzare la funzione `reversed` che restituisce un iteratore all'elenco invertito:

```
In [3]: rev = reversed(numbers)

In [4]: rev
Out[4]: [9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Si noti che l'elenco "numeri" rimane invariato per questa operazione e rimane nello stesso ordine in cui era originariamente.

Per invertire la posizione, puoi anche utilizzare il metodo `reverse` .

È anche possibile invertire una lista (ottenendo effettivamente una copia, l'elenco originale non viene modificato) utilizzando la sintassi slicing, impostando il terzo argomento (il passo) come -1:

```
In [1]: numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]

In [2]: numbers[::-1]
Out[2]: [9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Verifica se l'elenco è vuoto

Il vuoto di una lista è associato al `False` booleano, quindi non è necessario controllare `len(lst) == 0` , ma solo `lst` o `not lst`

```
lst = []
if not lst:
    print("list is empty")

# Output: list is empty
```

Concatena e Unisci elenchi

1. **Il modo più semplice per concatenare** `list1` e `list2` :

   ```
   merged = list1 + list2
   ```

2. **`zip` restituisce un elenco di tuple** , in cui l'i-esima tupla contiene l'elemento i-esimo da ciascuna delle sequenze di argomenti o iterabili:

   ```
   alist = ['a1', 'a2', 'a3']
   ```

```
blist = ['b1', 'b2', 'b3']

for a, b in zip(alist, blist):
    print(a, b)

# Output:
# a1 b1
# a2 b2
# a3 b3
```

Se gli elenchi hanno lunghezze diverse, il risultato includerà solo tanti elementi come il più breve:

```
alist = ['a1', 'a2', 'a3']
blist = ['b1', 'b2', 'b3', 'b4']
for a, b in zip(alist, blist):
    print(a, b)

# Output:
# a1 b1
# a2 b2
# a3 b3

alist = []
len(list(zip(alist, blist)))

# Output:
# 0
```

Per gli elenchi di riempimento di lunghezza non uguale a quella più lunga con `None` s usa `itertools.zip_longest` (`itertools.izip_longest` in Python 2)

```
alist = ['a1', 'a2', 'a3']
blist = ['b1']
clist = ['c1', 'c2', 'c3', 'c4']

for a,b,c in itertools.zip_longest(alist, blist, clist):
    print(a, b, c)

# Output:
# a1 b1 c1
# a2 None c2
# a3 None c3
# None None c4
```

3. **Inserisci a valori di indice specifici:**

```
 alist = [123, 'xyz', 'zara', 'abc']
 alist.insert(3, [2009])
 print("Final List :", alist)
```

Produzione:

```
 Final List : [123, 'xyz', 'zara', 2009, 'abc']
```

Ogni e qualsiasi

Puoi usare `all()` per determinare se tutti i valori in un iterabile valgono su True

```
nums = [1, 1, 0, 1]
all(nums)
# False
chars = ['a', 'b', 'c', 'd']
all(chars)
# True
```

Allo stesso modo, `any()` determina se uno o più valori in un iterabile valutano True

```
nums = [1, 1, 0, 1]
any(nums)
# True
vals = [None, None, None, False]
any(vals)
# False
```

Mentre questo esempio utilizza un elenco, è importante notare che questi lavori integrati funzionano con qualsiasi iterabile, inclusi i generatori.

```
vals = [1, 2, 3, 4]
any(val > 12 for val in vals)
# False
any((val * 2) > 6 for val in vals)
# True
```

Rimuovi i valori duplicati nella lista

La rimozione di valori duplicati in un elenco può essere effettuata convertendo l'elenco in un `set` (ovvero una raccolta non ordinata di oggetti distinti). Se è necessaria una struttura di dati `list` , il set può essere riconvertito in un elenco utilizzando la funzione `list()` :

```
names = ["aixk", "duke", "edik", "tofp", "duke"]
list(set(names))
# Out: ['duke', 'tofp', 'aixk', 'edik']
```

Si noti che convertendo una lista in un set si perde l'ordinamento originale.

Per preservare l'ordine della lista si può usare `OrderedDict`

```
import collections
>>> collections.OrderedDict.fromkeys(names).keys()
# Out: ['aixk', 'duke', 'edik', 'tofp']
```

Accesso ai valori nell'elenco nidificato

A partire da un elenco tridimensionale:

```
alist = [[[1,2],[3,4]], [[5,6,7],[8,9,10], [12, 13, 14]]]
```

Accedere agli elementi nell'elenco:

```
print(alist[0][0][1])
#2
#Accesses second element in the first list in the first list

print(alist[1][1][2])
#10
#Accesses the third element in the second list in the second list
```

Esecuzione delle operazioni di supporto:

```
alist[0][0].append(11)
print(alist[0][0][2])
#11
#Appends 11 to the end of the first list in the first list
```

Usando i cicli annidati per stampare l'elenco:

```
for row in alist: #One way to loop through nested lists
    for col in row:
        print(col)
#[1, 2, 11]
#[3, 4]
#[5, 6, 7]
#[8, 9, 10]
#[12, 13, 14]
```

Si noti che questa operazione può essere utilizzata in una comprensione di lista o anche come generatore per produrre efficienze, ad esempio:

```
[col for row in alist for col in row]
#[[1, 2, 11], [3, 4], [5, 6, 7], [8, 9, 10], [12, 13, 14]]
```

Non tutti gli elementi negli elenchi esterni devono essere elenchi stessi:

```
alist[1].insert(2, 15)
#Inserts 15 into the third position in the second list
```

Un altro modo di usare nested for loops. L'altro modo è migliore, ma ho avuto bisogno di usarlo occasionalmente:

```
for row in range(len(alist)): #A less Pythonic way to loop through lists
    for col in range(len(alist[row])):
       print(alist[row][col])

#[1, 2, 11]
#[3, 4]
#[5, 6, 7]
#[8, 9, 10]
#15
```

```
#[12, 13, 14]
```

Utilizzo delle sezioni nell'elenco nidificato:

```
print(alist[1][1:])
#[[8, 9, 10], 15, [12, 13, 14]]
#Slices still work
```

L'elenco finale:

```
print(alist)
#[[[1, 2, 11], [3, 4]], [[5, 6, 7], [8, 9, 10], 15, [12, 13, 14]]]
```

Confronto di liste

È possibile confrontare le liste e le altre sequenze lessicograficamente utilizzando operatori di confronto. Entrambi gli operandi devono essere dello stesso tipo.

```
[1, 10, 100] < [2, 10, 100]
# True, because 1 < 2
[1, 10, 100] < [1, 10, 100]
# False, because the lists are equal
[1, 10, 100] <= [1, 10, 100]
# True, because the lists are equal
[1, 10, 100] < [1, 10, 101]
# True, because 100 < 101
[1, 10, 100] < [0, 10, 100]
# False, because 0 < 1
```

Se una delle liste è contenuta all'inizio dell'altra, vince la lista più corta.

```
[1, 10] < [1, 10, 100]
# True
```

Inizializzazione di un elenco su un numero fisso di elementi

Per elementi **immutabili** (ad es. `None` , stringhe letterali ecc.):

```
my_list = [None] * 10
my_list = ['test'] * 10
```

Per gli elementi **mutabili** , lo stesso costrutto genererà tutti gli elementi della lista che si riferiscono allo stesso oggetto, ad esempio, per un insieme:

```
>>> my_list=[{1}] * 10
>>> print(my_list)
[{1}, {1}, {1}, {1}, {1}, {1}, {1}, {1}, {1}, {1}]
>>> my_list[0].add(2)
>>> print(my_list)
[{1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}]
```

Invece, per inizializzare la lista con un numero fisso di **diversi** oggetti **mutabili** , usa:

```
my_list=[{1} for _ in range(10)]
```

Leggi Elenco online: https://riptutorial.com/it/python/topic/209/elenco

Capitolo 54: Elenco delle comprensioni

introduzione

Una list comprehension è uno strumento sintattico per creare liste in modo naturale e conciso, come illustrato nel seguente codice per creare un elenco di quadrati dei numeri da 1 a 10: `[i ** 2 for i in range(1,11)]` Il dummy `i` di un `range` elenchi esistente viene utilizzato per creare un nuovo modello di elemento. È usato dove sarebbe necessario un ciclo for in lingue meno espressive.

Sintassi

- [i for i in range (10)] # comprensione di base dell'elenco
- [i for i in xrange (10)] # comprensione di base dell'elenco con l'oggetto generatore in python 2.x.
- [i for i in range (20) if i% 2 == 0] # con filtro
- [x + y per x in [1, 2, 3] per y in [3, 4, 5]] # cicli annidati
- [i if i> 6 else 0 for i in range (10)] # espressione ternaria
- [i if i> 4 else 0 per i in range (20) if i% 2 == 0] # con filtro e espressione ternaria
- [[x + y per x in [1, 2, 3]] per y in [3, 4, 5]] # nidificazione di elenchi nidificati

Osservazioni

La comprensione delle liste è stata delineata in PEP 202 e introdotta in Python 2.0.

Examples

Comprensioni di liste condizionali

Data una comprensione lista è possibile aggiungere una o più `if` le condizioni per filtrare i valori.

```
[<expression> for <element> in <iterable> if <condition>]
```

Per ogni `<element>` in `<iterable>` ; se `<condition>` restituisce `True` , aggiungere `<expression>` (di solito una funzione di `<element>`) all'elenco restituito.

Ad esempio, questo può essere usato per estrarre solo numeri pari da una sequenza di numeri interi:

```
[x for x in range(10) if x % 2 == 0]
# Out: [0, 2, 4, 6, 8]
```

Dimostrazione dal vivo

Il codice sopra è equivalente a:

```
even_numbers = []
for x in range(10):
    if x % 2 == 0:
        even_numbers.append(x)

print(even_numbers)
# Out: [0, 2, 4, 6, 8]
```

Inoltre, una comprensione dell'elenco condizionale della forma `[e for x in y if c]` (dove `e` `c` sono espressioni in termini di `x`) è equivalente alla `list(filter(lambda x: c, map(lambda x: e, y)))` .

Nonostante fornisca lo stesso risultato, presta attenzione al fatto che l'esempio precedente è quasi 2x più veloce di quest'ultimo. Per coloro che sono curiosi, questa è una bella spiegazione del motivo per cui.

Si noti che questo è molto diverso dall'espressione condizionale `... if ... else ...` (a volte nota come espressione ternaria) che è possibile utilizzare per la parte `<expression>` della list comprehension. Considera il seguente esempio:

```
[x if x % 2 == 0 else None for x in range(10)]
# Out: [0, None, 2, None, 4, None, 6, None, 8, None]
```

Dimostrazione dal vivo

Qui l'espressione condizionale non è un filtro, ma piuttosto un operatore che determina il valore da utilizzare per gli elementi dell'elenco:

```
<value-if-condition-is-true> if <condition> else <value-if-condition-is-false>
```

Questo diventa più ovvio se lo si combina con altri operatori:

```
[2 * (x if x % 2 == 0 else -1) + 1 for x in range(10)]
# Out: [1, -1, 5, -1, 9, -1, 13, -1, 17, -1]
```

Dimostrazione dal vivo

Se si utilizza Python 2.7, `xrange` può essere migliore `range` per diversi motivi, come descritto nella documentazione di `xrange` .

```
[2 * (x if x % 2 == 0 else -1) + 1 for x in xrange(10)]
# Out: [1, -1, 5, -1, 9, -1, 13, -1, 17, -1]
```

Il codice sopra è equivalente a:

```
numbers = []
for x in range(10):
    if x % 2 == 0:
```

```
        temp = x
    else:
        temp = -1
    numbers.append(2 * temp + 1)
print(numbers)
# Out: [1, -1, 5, -1, 9, -1, 13, -1, 17, -1]
```

Si può combinare espressioni ternari e `if` le condizioni. L'operatore ternario lavora sul risultato filtrato:

```
[x if x > 2 else '*' for x in range(10) if x % 2 == 0]
# Out: ['*', '*', 4, 6, 8]
```

Lo stesso non poteva essere raggiunto solo dall'operatore ternario:

```
[x if (x > 2 and x % 2 == 0) else '*' for x in range(10)]
# Out:['*', '*', '*', '*', 4, '*', 6, '*', 8, '*']
```

Vedi anche: Filtri , che spesso forniscono un'alternativa sufficiente alle comprensioni delle liste condizionali.

Elenco delle conclusioni con cicli annidati

Le comprensioni della lista possono usare annidati `for` cicli. È possibile codificare qualsiasi numero di cicli for innestati all'interno di una lista di comprensione, e ciascuno `for` ciclo può avere un optional associato `if` prova. Nel fare ciò, l'ordine del `for` costrutti è lo stesso ordine come quando si scrive una serie di nidificato `for` dichiarazioni. La struttura generale delle list comprehensions si presenta così:

```
[ expression for target1 in iterable1 [if condition1]
             for target2 in iterable2 [if condition2]...
             for targetN in iterableN [if conditionN] ]
```

Ad esempio, il seguente codice che appiattisce un elenco di elenchi utilizzando più istruzioni `for` :

```
data = [[1, 2], [3, 4], [5, 6]]
output = []
for each_list in data:
    for element in each_list:
        output.append(element)
print(output)
# Out: [1, 2, 3, 4, 5, 6]
```

può essere scritto in modo equivalente come una lista di comprensione con multipli `for` costrutti:

```
data = [[1, 2], [3, 4], [5, 6]]
output = [element for each_list in data for element in each_list]
print(output)
# Out: [1, 2, 3, 4, 5, 6]
```

Dimostrazione dal vivo

Sia nella forma espansa che nella comprensione della lista, il ciclo esterno (prima per la dichiarazione) viene prima di tutto.

Oltre ad essere più compatto, la comprensione annidata è anche significativamente più veloce.

```
In [1]: data = [[1,2],[3,4],[5,6]]
In [2]: def f():
   ...:     output=[]
   ...:     for each_list in data:
   ...:         for element in each_list:
   ...:             output.append(element)
   ...:     return output
In [3]: timeit f()
1000000 loops, best of 3: 1.37 µs per loop
In [4]: timeit [inner for outer in data for inner in outer]
1000000 loops, best of 3: 632 ns per loop
```

Il sovraccarico per la chiamata di funzione sopra è di circa *140ns* .

In linea `if` s sono nidificati in modo simile e possono verificarsi in qualsiasi posizione dopo il primo `for` :

```
data = [[1], [2, 3], [4, 5]]
output = [element for each_list in data
                if len(each_list) == 2
                for element in each_list
                if element != 5]
print(output)
# Out: [2, 3, 4]
```

Dimostrazione dal vivo

Per motivi di leggibilità, tuttavia, dovresti considerare l'utilizzo di *for-loops* tradizionali. Ciò è particolarmente vero quando la nidificazione è profonda più di 2 livelli e / o la logica della comprensione è troppo complessa. la comprensione di più elenchi di cicli annidati potrebbe essere soggetta a errori o fornire risultati imprevisti.

Refactoring filter e map to list comprehensions

Le funzioni di `filter` o `map` dovrebbero essere spesso sostituite dalla comprensione delle liste . Guido Van Rossum lo descrive bene in una lettera aperta nel 2005 :

> `filter(P, S)` è quasi sempre scritto più chiaro come `[x for x in S if P(x)]` , e questo ha l'enorme vantaggio che gli usi più comuni riguardano predicati che sono confronti, ad esempio `x==42` , e che definiscono un lambda per quello richiede solo molto più sforzo per il lettore (più il lambda è più lento della comprensione delle liste). Ancora di più per la `map(F, S)` che diventa `[F(x) for x in S]` . Ovviamente, in molti casi è possibile utilizzare le espressioni del generatore.

Le seguenti righe di codice sono considerate " *non pitoni* " e solleveranno errori in molti lint di pitone.

```
filter(lambda x: x % 2 == 0, range(10)) # even numbers < 10
map(lambda x: 2*x, range(10)) # multiply each number by two
reduce(lambda x,y: x+y, range(10)) # sum of all elements in list
```

Prendendo ciò che abbiamo imparato dalla citazione precedente, possiamo suddividere queste espressioni di `filter` e `map` nelle loro equivalenti *liste di comprensione* ; rimuovendo anche le funzioni *lambda* da ognuna - rendendo il codice più leggibile nel processo.

```
# Filter:
# P(x) = x % 2 == 0
# S = range(10)
[x for x in range(10) if x % 2 == 0]

# Map
# F(x) = 2*x
# S = range(10)
[2*x for x in range(10)]
```

La leggibilità diventa ancora più evidente quando si ha a che fare con le funzioni di concatenamento. Dove a causa della leggibilità, i risultati di una funzione mappa o filtro dovrebbero essere passati come risultato al successivo; con casi semplici, questi possono essere sostituiti con una sola lista di comprensione. Inoltre, possiamo facilmente capire dalla comprensione della lista quale sia il risultato del nostro processo, dove c'è un carico cognitivo maggiore nel ragionare sul processo di mappatura e filtro concatenato.

```
# Map & Filter
filtered = filter(lambda x: x % 2 == 0, range(10))
results = map(lambda x: 2*x, filtered)

# List comprehension
results = [2*x for x in range(10) if x % 2 == 0]
```

Refactoring - Riferimento rapido

- **Carta geografica**

  ```
  map(F, S) == [F(x) for x in S]
  ```

- **Filtro**

  ```
  filter(P, S) == [x for x in S if P(x)]
  ```

dove `F` e `P` sono funzioni che trasformano rispettivamente i valori di input e restituiscono un `bool`

Comprensioni di liste annidate

Le comprensioni delle liste annidate, a differenza delle comprensioni degli elenchi con cicli annidati, sono le Comprensioni delle liste all'interno di una comprensione di lista. L'espressione iniziale può essere qualsiasi espressione arbitraria, inclusa un'altra comprensione di lista.

```
#List Comprehension with nested loop
[x + y for x in [1, 2, 3] for y in [3, 4, 5]]
#Out: [4, 5, 6, 5, 6, 7, 6, 7, 8]

#Nested List Comprehension
[[x + y for x in [1, 2, 3]] for y in [3, 4, 5]]
#Out: [[4, 5, 6], [5, 6, 7], [6, 7, 8]]
```

L'esempio annidato è equivalente a

```
l = []
for y in [3, 4, 5]:
    temp = []
    for x in [1, 2, 3]:
        temp.append(x + y)
    l.append(temp)
```

Un esempio in cui una comprensione annidata può essere utilizzata per trasporre una matrice.

```
matrix = [[1,2,3],
          [4,5,6],
          [7,8,9]]

[[row[i] for row in matrix] for i in range(len(matrix))]
# [[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

Come nidificati `for` cicli, non c'è limite a come le intese profonde possono essere annidate.

```
[[[i + j + k for k in 'cd'] for j in 'ab'] for i in '12']
# Out: [[['1ac', '1ad'], ['1bc', '1bd']], [['2ac', '2ad'], ['2bc', '2bd']]]
```

Iterate due o più liste contemporaneamente all'interno della comprensione delle liste

Per iterare più di due elenchi contemporaneamente nella *comprensione di liste* , si può usare `zip()` come:

```
>>> list_1 = [1, 2, 3 , 4]
>>> list_2 = ['a', 'b', 'c', 'd']
>>> list_3 = ['6', '7', '8', '9']

# Two lists
>>> [(i, j) for i, j in zip(list_1, list_2)]
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]

# Three lists
>>> [(i, j, k) for i, j, k in zip(list_1, list_2, list_3)]
[(1, 'a', '6'), (2, 'b', '7'), (3, 'c', '8'), (4, 'd', '9')]
```

```
# so on ...
```

Leggi Elenco delle comprensioni online: https://riptutorial.com/it/python/topic/5265/elenco-delle-comprensioni

Capitolo 55: Elenco delle sezioni (selezione di parti di elenchi)

Sintassi

- a [inizio: fine] # articoli iniziano con end-1
- a [inizio:] # elementi iniziano attraverso il resto dell'array
- a [: fine] # elementi dall'inizio alla fine-1
- a [inizio: fine: passaggio] # dall'inizio non passato, per passaggio
- a [:] # una copia dell'intero array
- fonte

Osservazioni

- `lst[::-1]` ti dà una copia invertita della lista
- `start` o `end` può essere un numero negativo, il che significa che conta dalla fine dell'array anziché dall'inizio. Così:

```
a[-1]    # last item in the array
a[-2:]   # last two items in the array
a[:-2]   # everything except the last two items
```

(fonte)

Examples

Utilizzando il terzo argomento "step"

```
lst = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']

lst[::2]
# Output: ['a', 'c', 'e', 'g']

lst[::3]
# Output: ['a', 'd', 'g']
```

Selezione di una sottolista da una lista

```
lst = ['a', 'b', 'c', 'd', 'e']

lst[2:4]
# Output: ['c', 'd']

lst[2:]
# Output: ['c', 'd', 'e']
```

```
lst[:4]
# Output: ['a', 'b', 'c', 'd']
```

Inversione di una lista con affettatura

```
a = [1, 2, 3, 4, 5]

# steps through the list backwards (step=-1)
b = a[::-1]

# built-in list method to reverse 'a'
a.reverse()

if a = b:
    print(True)

print(b)

# Output:
# True
# [5, 4, 3, 2, 1]
```

Spostamento di un elenco utilizzando l'affettatura

```
def shift_list(array, s):
    """Shifts the elements of a list to the left or right.

    Args:
        array - the list to shift
        s - the amount to shift the list ('+': right-shift, '-': left-shift)

    Returns:
        shifted_array - the shifted list
    """
    # calculate actual shift amount (e.g., 11 --> 1 if length of the array is 5)
    s %= len(array)

    # reverse the shift direction to be more intuitive
    s *= -1

    # shift array with list slicing
    shifted_array = array[s:] + array[:s]

    return shifted_array

my_array = [1, 2, 3, 4, 5]

# negative numbers
shift_list(my_array, -7)
>>> [3, 4, 5, 1, 2]

# no shift on numbers equal to the size of the array
shift_list(my_array, 5)
>>> [1, 2, 3, 4, 5]

# works on positive numbers
shift_list(my_array, 3)
```

```
>>> [3, 4, 5, 1, 2]
```

Leggi Elenco delle sezioni (selezione di parti di elenchi) online:
https://riptutorial.com/it/python/topic/1494/elenco-delle-sezioni--selezione-di-parti-di-elenchi-

Capitolo 56: Elenco di destrutturazione (ovvero imballaggio e disimballaggio)

Examples

Incarico distruttivo

Nei compiti, puoi dividere un Iterable in valori usando la sintassi "decompressione":

Distruzione come valori

```
a, b = (1, 2)
print(a)
# Prints: 1
print(b)
# Prints: 2
```

Se tenti di decomprimere più della lunghezza del iterabile, riceverai un errore:

```
a, b, c = [1]
# Raises: ValueError: not enough values to unpack (expected 3, got 1)
```

Python 3.x 3.0

Distruzione come lista

È possibile decomprimere un elenco di lunghezza sconosciuta utilizzando la seguente sintassi:

```
head, *tail = [1, 2, 3, 4, 5]
```

Qui, estraiamo il primo valore come scalare e gli altri valori come un elenco:

```
print(head)
# Prints: 1
print(tail)
# Prints: [2, 3, 4, 5]
```

Che è equivalente a:

```
l = [1, 2, 3, 4, 5]
head = l[0]
tail = l[1:]
```

Funziona anche con più elementi o elementi alla fine dell'elenco:

```
a, b, *other, z = [1, 2, 3, 4, 5]
print(a, b, z, other)
# Prints: 1 2 5 [3, 4]
```

Ignorare i valori nei compiti distruttivi

Se ti interessa solo un determinato valore, puoi usare _ per indicare che non sei interessato. Nota: questo sarà ancora impostato su _ , solo la maggior parte delle persone non lo usa come variabile.

```
a, _ = [1, 2]
print(a)
# Prints: 1
a, _, c = (1, 2, 3)
print(a)
# Prints: 1
print(c)
# Prints: 3
```

Python 3.x 3.0

Ignorare le liste in compiti distruttivi

Infine, puoi ignorare molti valori usando la sintassi `*_` nell'assegnazione:

```
a, *_ = [1, 2, 3, 4, 5]
print(a)
# Prints: 1
```

che non è molto interessante, in quanto è possibile utilizzare l'indicizzazione nell'elenco. Dove è bello mantenere il primo e l'ultimo valore in un compito:

```
a, *_, b = [1, 2, 3, 4, 5]
print(a, b)
# Prints: 1 5
```

o estrai più valori contemporaneamente:

```
a, _, b, _, c, *_ = [1, 2, 3, 4, 5, 6]
print(a, b, c)
# Prints: 1 3 5
```

Argomenti della funzione di imballaggio

Nelle funzioni, è possibile definire un numero di argomenti obbligatori:

```
def fun1(arg1, arg2, arg3):
    return (arg1,arg2,arg3)
```

che renderà la funzione chiamabile solo quando vengono dati i tre argomenti:

```
fun1(1, 2, 3)
```

ed è possibile definire gli argomenti come facoltativi, utilizzando i valori predefiniti:

```
def fun2(arg1='a', arg2='b', arg3='c'):
    return (arg1,arg2,arg3)
```

quindi puoi chiamare la funzione in molti modi diversi, come:

```
fun2(1)              → (1,b,c)
fun2(1, 2)           → (1,2,c)
fun2(arg2=2, arg3=3) → (a,2,3)
...
```

Ma puoi anche usare la sintassi destructuring per *impacchettare gli* argomenti, così puoi assegnare variabili usando una `list` o un `dict` .

Imballaggio di un elenco di argomenti

Considera di avere una lista di valori

```
l = [1,2,3]
```

Puoi chiamare la funzione con l'elenco di valori come argomento usando la `*` sintassi:

```
fun1(*l)
# Returns: (1,2,3)
fun1(*['w', 't', 'f'])
# Returns: ('w','t','f')
```

Ma se non si fornisce un elenco con la lunghezza corrispondente al numero di argomenti:

```
fun1(*['oops'])
# Raises: TypeError: fun1() missing 2 required positional arguments: 'arg2' and 'arg3'
```

Imballaggio degli argomenti delle parole chiave

Ora puoi anche comprimere argomenti usando un dizionario. Puoi usare l'operatore `**` per dire a Python di decomprimere il `dict` come valori dei parametri:

```
d = {
  'arg1': 1,
  'arg2': 2,
  'arg3': 3
}
fun1(**d)
# Returns: (1, 2, 3)
```

quando la funzione ha solo argomenti posizionali (quelli senza valori predefiniti) è necessario che

il dizionario contenga tutti i parametri previsti e non disponga di parametri aggiuntivi, altrimenti si otterrà un errore:

```
fun1(**{'arg1':1, 'arg2':2})
# Raises: TypeError: fun1() missing 1 required positional argument: 'arg3'
fun1(**{'arg1':1, 'arg2':2, 'arg3':3, 'arg4':4})
# Raises: TypeError: fun1() got an unexpected keyword argument 'arg4'
```

Per le funzioni che dispongono di argomenti facoltativi, è possibile comprimere gli argomenti come dizionario nello stesso modo:

```
fun2(**d)
# Returns: (1, 2, 3)
```

Ma qui puoi omettere i valori, poiché verranno sostituiti con i valori predefiniti:

```
fun2(**{'arg2': 2})
# Returns: ('a', 2, 'c')
```

E lo stesso di prima, non puoi dare valori extra che non siano parametri esistenti:

```
fun2(**{'arg1':1, 'arg2':2, 'arg3':3, 'arg4':4})
# Raises: TypeError: fun2() got an unexpected keyword argument 'arg4'
```

Nell'uso del mondo reale, le funzioni possono avere sia argomenti posizionali che opzionali, e funziona allo stesso modo:

```
def fun3(arg1, arg2='b', arg3='c')
    return (arg1, arg2, arg3)
```

puoi chiamare la funzione solo con un iterable:

```
fun3(*[1])
# Returns: (1, 'b', 'c')
fun3(*[1,2,3])
# Returns: (1, 2, 3)
```

o con solo un dizionario:

```
fun3(**{'arg1':1})
# Returns: (1, 'b', 'c')
fun3(**{'arg1':1, 'arg2':2, 'arg3':3})
# Returns: (1, 2, 3)
```

oppure puoi utilizzare entrambi nella stessa chiamata:

```
fun3(*[1,2], **{'arg3':3})
# Returns: (1,2,3)
```

Attenzione però che non puoi fornire più valori per lo stesso argomento:

```
fun3(*[1,2], **{'arg2':42, 'arg3':3})
# Raises: TypeError: fun3() got multiple values for argument 'arg2'
```

Spacchettare gli argomenti della funzione

Quando si desidera creare una funzione che può accettare un numero qualsiasi di argomenti e non imporre la posizione o il nome dell'argomento in fase di "compilazione", è possibile ed ecco come:

```
def fun1(*args, **kwargs):
    print(args, kwargs)
```

I parametri `*args` e `**kwargs` sono parametri speciali che sono impostati rispettivamente su una `tuple` e un `dict` :

```
fun1(1,2,3)
# Prints: (1, 2, 3) {}
fun1(a=1, b=2, c=3)
# Prints: () {'a': 1, 'b': 2, 'c': 3}
fun1('x', 'y', 'z', a=1, b=2, c=3)
# Prints: ('x', 'y', 'z') {'a': 1, 'b': 2, 'c': 3}
```

Se si guarda abbastanza codice Python, si scoprirà rapidamente che è ampiamente utilizzato quando si passano gli argomenti a un'altra funzione. Ad esempio se si desidera estendere la classe string:

```
class MyString(str):
    def __init__(self, *args, **kwarg):
        print('Constructing MyString')
        super(MyString, self).__init__(*args, **kwarg)
```

Leggi Elenco di destrutturazione (ovvero imballaggio e disimballaggio) online: https://riptutorial.com/it/python/topic/4282/elenco-di-destrutturazione--ovvero-imballaggio-e-disimballaggio-

Capitolo 57: elevamento a potenza

Sintassi

- valore1 ** valore2
- pow (valore1, valore2 [, valore3])
- valore1 .__ pow __ (valore2 [, valore3])
- valore2 .__ rpow __ (valore1)
- operator.pow (valore1, valore2)
- operatore .__ pow __ (valore1, valore2)
- math.pow (valore1, valore2)
- Math.sqrt (valore1)
- math.exp (valore1)
- cmath.exp (valore1)
- math.expm1 (valore1)

Examples

Radice quadrata: math.sqrt () e cmath.sqrt

Il modulo `math` contiene la funzione `math.sqrt()` che può calcolare la radice quadrata di qualsiasi numero (che può essere convertito in un `float`) e il risultato sarà sempre un `float` :

```
import math

math.sqrt(9)                 # 3.0
math.sqrt(11.11)             # 3.3331666624997918
math.sqrt(Decimal('6.25'))   # 2.5
```

La funzione `math.sqrt()` solleva `math.sqrt() ValueError` se il risultato sarebbe `complex` :

```
math.sqrt(-10)
```

ValueError: errore del dominio matematico

`math.sqrt(x)` è *più veloce* di `math.pow(x, 0.5)` o `x ** 0.5` ma la precisione dei risultati è la stessa. Il modulo `cmath` è estremamente simile al modulo `math` , tranne per il fatto che può calcolare numeri complessi e tutti i suoi risultati sono nella forma di un + bi. Può anche usare `.sqrt()` :

```
import cmath

cmath.sqrt(4)  # 2+0j
cmath.sqrt(-4) # 2j
```

Cosa c'è nella `j` ? `j` è l'equivalente della radice quadrata di -1. Tutti i numeri possono essere messi in forma a + bi, o in questo caso, a + bj. `a` è la parte reale del numero come il 2 in `2+0j` . Poiché

non ha una parte immaginaria, `b` è 0. `b` rappresenta parte della parte immaginaria del numero come il 2 in `2j` . Poiché non c'è una parte reale in questo, `2j` può anche essere scritto come `0 + 2j` .

Esponenziazione usando i builtin: ** e pow ()

L'esponenziazione può essere utilizzata utilizzando la funzione integrata `pow` o l'operatore `**` :

```
2 ** 3    # 8
pow(2, 3) # 8
```

Per la maggior parte delle operazioni aritmetiche (tutte in Python 2.x) il tipo di risultato sarà quello dell'operando più ampio. Questo non è vero per `**` ; i seguenti casi sono eccezioni a questa regola:

- Base: `int` , esponente: `int < 0` :

  ```
  2 ** -3
  # Out: 0.125 (result is a float)
  ```

- Questo è valido anche per Python 3.x.
- Prima di Python 2.2.0, questo ha generato un `ValueError` .
- Base: `int < 0` o `float < 0` , esponente: `float != int`

  ```
  (-2) ** (0.5)  # also (-2.) ** (0.5)
  # Out: (8.659560562354934e-17+1.4142135623730951j) (result is complex)
  ```

- Prima di python 3.0.0, questo ha generato un `ValueError` .

Il modulo `operator` contiene due funzioni equivalenti al `**` -operator:

```
import operator
operator.pow(4, 2)       # 16
operator.__pow__(4, 3)   # 64
```

oppure si potrebbe chiamare direttamente il metodo `__pow__` :

```
val1, val2 = 4, 2
val1.__pow__(val2)       # 16
val2.__rpow__(val1)      # 16
# in-place power operation isn't supported by immutable classes like int, float, complex:
# val1.__ipow__(val2)
```

Esponenziazione usando il modulo matematico: math.pow ()

Il modulo `math` contiene un'altra funzione `math.pow()` . La differenza con la funzione built-in `pow()` o `**` è che il risultato è sempre un `float` :

```
import math
math.pow(2, 2)    # 4.0
math.pow(-2., 2)  # 4.0
```

Che esclude i calcoli con input complessi:

```
math.pow(2, 2+0j)
```

TypeError: impossibile convertire il complesso in float

e calcoli che porterebbero a risultati complessi:

```
math.pow(-2, 0.5)
```

ValueError: errore del dominio matematico

Funzione esponenziale: math.exp () e cmath.exp ()

Sia il modulo `math` che il modulo `cmath` contengono il numero di Eulero: e e lo utilizza con la funzione built-in `pow()` o `**` -operator funziona principalmente come `math.exp()` :

```
import math

math.e ** 2  # 7.3890560989306495
math.exp(2)  # 7.38905609893065

import cmath
cmath.e ** 2 # 7.3890560989306495
cmath.exp(2) # (7.38905609893065+0j)
```

Tuttavia, il risultato è diverso e l'utilizzo della funzione esponenziale direttamente è più affidabile rispetto alla funzione di esponenziazione integrata con base `math.e` :

```
print(math.e ** 10)       # 22026.465794806703
print(math.exp(10))       # 22026.465794806718
print(cmath.exp(10).real) # 22026.465794806718
#     difference starts here ---------------^
```

Funzione esponenziale meno 1: math.expm1 ()

Il modulo `math` contiene la funzione `expm1()` che può calcolare l'espressione `math.e ** x - 1` per `x` molto piccolo con maggiore precisione rispetto a `math.exp(x)` o `cmath.exp(x)` consentirebbe:

```
import math

print(math.e ** 1e-3 - 1)  # 0.0010005001667083846
print(math.exp(1e-3) - 1)  # 0.0010005001667083846
print(math.expm1(1e-3))    # 0.0010005001667083417
#                            ------------------^
```

Per piccolissime x la differenza aumenta:

```
print(math.e ** 1e-15 - 1) # 1.1102230246251565e-15
print(math.exp(1e-15) - 1) # 1.1102230246251565e-15
print(math.expm1(1e-15))   # 1.0000000000000007e-15
#                            ^-------------------
```

Il miglioramento è significativo nel calcolo scientifico. Ad esempio la legge di Planck contiene una funzione esponenziale meno 1:

```
def planks_law(lambda_, T):
    from scipy.constants import h, k, c  # If no scipy installed hardcode these!
    return 2 * h * c ** 2 / (lambda_ ** 5 * math.expm1(h * c / (lambda_ * k * T)))

def planks_law_naive(lambda_, T):
    from scipy.constants import h, k, c  # If no scipy installed hardcode these!
    return 2 * h * c ** 2 / (lambda_ ** 5 * (math.e ** (h * c / (lambda_ * k * T)) - 1))

planks_law(100, 5000)        # 4.139080074896474e-19
planks_law_naive(100, 5000)  # 4.139080073488451e-19
#                                      ^----------

planks_law(1000, 5000)       # 4.139080128493406e-23
planks_law_naive(1000, 5000) # 4.139080233183142e-23
#                                    ^------------
```

Metodi magici ed esponenziazione: builtin, matematica e cmath

Supponendo che tu abbia una classe che memorizza valori puramente interi:

```
class Integer(object):
    def __init__(self, value):
        self.value = int(value) # Cast to an integer

    def __repr__(self):
        return '{cls}({val})'.format(cls=self.__class__.__name__,
                                     val=self.value)

    def __pow__(self, other, modulo=None):
        if modulo is None:
            print('Using __pow__')
            return self.__class__(self.value ** other)
        else:
            print('Using __pow__ with modulo')
            return self.__class__(pow(self.value, other, modulo))

    def __float__(self):
        print('Using __float__')
        return float(self.value)

    def __complex__(self):
        print('Using __complex__')
        return complex(self.value, 0)
```

Usando la funzione built-in `pow` o `**` operatore chiama sempre `__pow__` :

```
Integer(2) ** 2                 # Integer(4)
# Prints: Using __pow__
```

```
Integer(2) ** 2.5               # Integer(5)
# Prints: Using __pow__
pow(Integer(2), 0.5)            # Integer(1)
# Prints: Using __pow__
operator.pow(Integer(2), 3)     # Integer(8)
# Prints: Using __pow__
operator.__pow__(Integer(3), 3) # Integer(27)
# Prints: Using __pow__
```

Il secondo argomento del metodo `__pow__()` può essere fornito solo usando builtin- `pow()` o chiamando direttamente il metodo:

```
pow(Integer(2), 3, 4)           # Integer(0)
# Prints: Using __pow__ with modulo
Integer(2).__pow__(3, 4)        # Integer(0)
# Prints: Using __pow__ with modulo
```

Mentre le funzioni `math` lo convertono sempre in `float` e usano il calcolo a virgola mobile:

```
import math

math.pow(Integer(2), 0.5) # 1.4142135623730951
# Prints: Using __float__
```

`cmath` provano a convertirlo in un insieme `complex` ma possono anche eseguire il fallback in modo che `float` se non vi è alcuna conversione esplicita a `complex` :

```
import cmath

cmath.exp(Integer(2))     # (7.38905609893065+0j)
# Prints: Using __complex__

del Integer.__complex__   # Deleting __complex__ method - instances cannot be cast to complex

cmath.exp(Integer(2))     # (7.38905609893065+0j)
# Prints: Using __float__
```

Né `math` né `cmath` funzioneranno se `__float__()` anche il `__float__()` :

```
del Integer.__float__  # Deleting __complex__ method

math.sqrt(Integer(2))  # also cmath.exp(Integer(2))
```

TypeError: è richiesto un float

Esponenziazione modulare: pow () con 3 argomenti

Fornire `pow()` con 3 argomenti `pow(a, b, c)` valuta l' esponenziazione modulare $a^b \bmod c$:

```
pow(3, 4, 17)   # 13

# equivalent unoptimized expression:
3 ** 4 % 17     # 13
```

```
# steps:
3 ** 4          # 81
81 % 17         # 13
```

Per i tipi built-in che utilizzano l'esponenziazione modulare è possibile solo se:

- Il primo argomento è un `int`
- Il secondo argomento è un `int >= 0`
- Il terzo argomento è un `int != 0`

Queste restrizioni sono presenti anche in python 3.x

Ad esempio si può usare la forma a 3 argomenti di `pow` per definire una funzione inversa modulare :

```
def modular_inverse(x, p):
    """Find a such as  a·x ≡ 1 (mod p), assuming p is prime."""
    return pow(x, p-2, p)

[modular_inverse(x, 13) for x in range(1,13)]
# Out: [1, 7, 9, 10, 8, 11, 2, 5, 3, 4, 6, 12]
```

Radici: nth-root con esponenti frazionali

Mentre la funzione `math.sqrt` è fornita per il caso specifico di radici quadrate, è spesso conveniente usare l'operatore di `math.sqrt` (`**`) con esponenti frazionari per eseguire operazioni di nth-root, come le radici di cubi.

L'inverso di un esponenziazione è esponenziale dal reciproco dell'esponente. Quindi, se puoi cubare un numero mettendolo all'esponente di 3, puoi trovare la radice cubica di un numero mettendolo all'esponente di 1/3.

```
>>> x = 3
>>> y = x ** 3
>>> y
27
>>> z = y ** (1.0 / 3)
>>> z
3.0
>>> z == x
True
```

Calcolo di radici intere di grandi dimensioni

Anche se Python supporta in modo nativo i grandi numeri interi, l'uso dell'ennesima radice di numeri molto grandi può fallire in Python.

```
x = 2 ** 100
cube = x ** 3
root = cube ** (1.0 / 3)
```

OverflowError: long int troppo grande per essere convertito in float

Quando si ha a che fare con numeri interi così grandi, è necessario utilizzare una funzione personalizzata per calcolare l'ennesima radice di un numero.

```
def nth_root(x, n):
    # Start with some reasonable bounds around the nth root.
    upper_bound = 1
    while upper_bound ** n <= x:
        upper_bound *= 2
    lower_bound = upper_bound // 2
    # Keep searching for a better result as long as the bounds make sense.
    while lower_bound < upper_bound:
        mid = (lower_bound + upper_bound) // 2
        mid_nth = mid ** n
        if lower_bound < mid and mid_nth < x:
            lower_bound = mid
        elif upper_bound > mid and mid_nth > x:
            upper_bound = mid
        else:
            # Found perfect nth root.
            return mid
    return mid + 1

x = 2 ** 100
cube = x ** 3
root = nth_root(cube, 3)
x == root
# True
```

Leggi elevamento a potenza online: https://riptutorial.com/it/python/topic/347/elevamento-a-potenza

Capitolo 58: enum

Osservazioni

Enum furono aggiunti a Python nella versione 3.4 di PEP 435 .

Examples

Creazione di un enum (Python 2.4 tramite 3.3)

Le enumerazioni sono state trasferite da Python 3.4 a Python 2.4 tramite Python 3.3. È possibile ottenere questo l'enum34 backport dal PyPI.

```
pip install enum34
```

La creazione di un enum è identica a come funziona in Python 3.4+

```
from enum import Enum

class Color(Enum):
    red = 1
    green = 2
    blue = 3

print(Color.red)  # Color.red
print(Color(1))  # Color.red
print(Color['red'])  # Color.red
```

Iterazione

Le enumerazioni sono iterabili:

```
class Color(Enum):
    red = 1
    green = 2
    blue = 3

[c for c in Color]  # [<Color.red: 1>, <Color.green: 2>, <Color.blue: 3>]
```

Leggi enum online: https://riptutorial.com/it/python/topic/947/enum

Capitolo 59: Esecuzione di codice dinamico con `exec` e` eval`

Sintassi

- eval (espressione [, globals = None [, locals = None]])
- exec (oggetto)
- exec (oggetto, globals)
- exec (oggetto, globals, gente del posto)

Parametri

Discussione	Dettagli
`expression`	Il codice dell'espressione come una stringa o un oggetto `code`
`object`	Il codice dell'istruzione come una stringa o un oggetto `code`
`globals`	Il dizionario da usare per le variabili globali. Se la gente del posto non è specificata, questo è anche usato per i locali. Se omesso, vengono utilizzati i `globals()` dell'ambito di chiamata.
`locals`	Un oggetto di *mappatura* che viene utilizzato per le variabili locali. Se omesso, viene utilizzato invece quello per le `globals` . Se entrambi vengono omessi, i `globals()` e i `locals()` dell'ambito chiamante vengono utilizzati rispettivamente per `globals` e `locals` .

Osservazioni

In `exec` , se i `globals` sono `locals` (cioè si riferiscono allo stesso oggetto), il codice viene eseguito come se fosse a livello di modulo. Se `globals` e `locals` sono oggetti distinti, il codice viene eseguito come se fosse in un *corpo di classe* .

Se l'oggetto `globals` viene passato, ma non specifica la chiave `__builtins__` , le funzioni e i nomi incorporati di Python vengono automaticamente aggiunti all'ambito globale. Per sopprimere la disponibilità di funzioni come `print` o `isinstance` nell'ambito eseguito, lasciare che le `globals` abbiano la chiave `__builtins__` mappata sul valore `None` . Tuttavia, questa non è una funzione di sicurezza.

La sintassi specifica di Python 2 non dovrebbe essere utilizzata; la sintassi Python 3 funzionerà in Python 2. Quindi i seguenti moduli sono deprecati: <s>

- `exec object`
- `exec object in globals`

- `exec object in globals, locals`

Examples

Valutazione di dichiarazioni con exec

```
>>> code = """for i in range(5):\n    print('Hello world!')"""
>>> exec(code)
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

Valutare un'espressione con eval

```
>>> expression = '5 + 3 * a'
>>> a = 5
>>> result = eval(expression)
>>> result
20
```

Precompilare un'espressione per valutarla più volte

`compile` funzione incorporata può essere utilizzata per precompilare un'espressione a un oggetto codice; questo oggetto codice può quindi essere passato a valutazione. Ciò velocizzerà le ripetute esecuzioni del codice valutato. Il terzo parametro da `compile` deve essere la stringa `'eval'` .

```
>>> code = compile('a * b + c', '<string>', 'eval')
>>> code
<code object <module> at 0x7f0e51a58830, file "<string>", line 1>
>>> a, b, c = 1, 2, 3
>>> eval(code)
5
```

Valutare un'espressione con eval utilizzando globals personalizzati

```
>>> variables = {'a': 6, 'b': 7}
>>> eval('a * b', globals=variables)
42
```

Come un vantaggio, con questo il codice non può accidentalmente fare riferimento ai nomi definiti al di fuori:

```
>>> eval('variables')
{'a': 6, 'b': 7}
>>> eval('variables', globals=variables)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1, in <module>
```

```
NameError: name 'variables' is not defined
```

L'uso di `defaultdict` consente ad esempio di avere variabili non definite impostate su zero:

```
>>> from collections import defaultdict
>>> variables = defaultdict(int, {'a': 42})
>>> eval('a * c', globals=variables)  # note that 'c' is not explicitly defined
0
```

Valutare una stringa contenente un letterale Python con ast.literal_eval

Se hai una stringa che contiene letterali Python, come stringhe, float ecc., Puoi usare `ast.literal_eval` per valutare il suo valore invece di `eval` . Questo ha la caratteristica aggiunta di consentire solo una certa sintassi.

```
>>> import ast
>>> code = """(1, 2, {'foo': 'bar'})"""
>>> object = ast.literal_eval(code)
>>> object
(1, 2, {'foo': 'bar'})
>>> type(object)
<class 'tuple'>
```

Tuttavia, questo non è sicuro per l'esecuzione del codice fornito da un utente non fidato, ed è banale arrestare un interprete con un input accuratamente predisposto

```
>>> import ast
>>> ast.literal_eval('()' * 1000000)
[5]    21358 segmentation fault (core dumped)  python3
```

Qui, l'input è una stringa di `()` ripetuta un milione di volte, che provoca un crash nel parser CPython. Gli sviluppatori CPython non considerano i bug nel parser come problemi di sicurezza.

Esecuzione del codice fornito dall'utente non fidato mediante exec, eval o ast.literal_eval

Non è possibile utilizzare `eval` o `exec` per eseguire il codice dell'utente inaffidabile. Anche `ast.literal_eval` è soggetto a crash nel parser. A volte è possibile evitare l'esecuzione di codice dannoso, ma non esclude la possibilità di arresto anomalo nel parser o nel tokenizer.

Per valutare il codice da un utente non fidato è necessario passare a qualche modulo di terze parti, o magari scrivere il proprio parser e la propria macchina virtuale in Python.

Leggi Esecuzione di codice dinamico con `exec` e` eval` online:
https://riptutorial.com/it/python/topic/2251/esecuzione-di-codice-dinamico-con--exec--e--eval-

Capitolo 60: Espressioni regolari (Regex)

introduzione

Python rende disponibili le espressioni regolari attraverso il modulo `re` .

Le espressioni regolari sono combinazioni di caratteri che vengono interpretate come regole per la corrispondenza delle sottostringhe. Ad esempio, l'espressione `'amount\D+\d+'` corrisponderà qualsiasi stringa composta dalla parola `amount` più un numero intero, separati da uno o più non cifre, ad esempio: `amount=100` , `amount is 3` , `amount is equal to: 33` , ecc.

Sintassi

- **Espressioni regolari dirette**
- re.match (pattern, stringa, flag = 0) # Out: combina pattern all'inizio della stringa o None
- re.search (pattern, string, flag = 0) # Out: combina pattern all'interno di string o None
- re.findall (pattern, stringa, flag = 0) # Out: lista di tutte le corrispondenze di pattern in stringa o []
- re.finditer (pattern, string, flag = 0) # Out: uguale a re.findall, ma restituisce l'oggetto iteratore
- re.sub (modello, sostituzione, stringa, flag = 0) # Out: stringa con sostituzione (stringa o funzione) al posto del modello
- **Espressioni regolari precompilate**
- precompiled_pattern = re.compile (pattern, flag = 0)
- precompiled_pattern.match (stringa) # Out: corrisponde all'inizio della stringa o None
- precompiled_pattern.search (stringa) # Out: corrisponde ovunque in stringa o None
- precompiled_pattern.findall (stringa) # Out: elenco di tutte le sottostringhe corrispondenti
- precompiled_pattern.sub (stringa / modello / funzione, stringa) # Out: stringa sostituita

Examples

Corrisponde all'inizio di una stringa

Il primo argomento di `re.match()` è l'espressione regolare, il secondo è la stringa da abbinare:

```
import re
```

```
pattern = r"123"
string = "123zzb"

re.match(pattern, string)
# Out: <_sre.SRE_Match object; span=(0, 3), match='123'>

match = re.match(pattern, string)

match.group()
# Out: '123'
```

Si può notare che la variabile pattern è una stringa preceduta da `r` , che indica che la stringa è un *letterale stringa raw* .

Un letterale stringa raw ha una sintassi leggermente diversa rispetto a un letterale stringa, ovvero una barra rovesciata `\` in una stringa raw significa letteralmente "solo un backslash" e non è necessario raddoppiare i backlashes per evitare "sequenze di escape" come newlines (`\n`) , schede (`\t`), backspaces (`\`), form-feeds (`\r`) e così via. In normali stringhe di stringa, ogni backslash deve essere raddoppiata per evitare di essere presa come l'inizio di una sequenza di escape.

Quindi, `r"\n"` è una stringa di 2 caratteri: `\` e `n` . Anche i pattern Regex usano i backslash, ad esempio `\d` riferisce a qualsiasi carattere numerico. È possibile evitare di dover eseguire il doppio escape delle stringhe (`"\\d"`) utilizzando stringhe non elaborate (`r"\d"`).

Per esempio:

```
string = "\\t123zzb" # here the backslash is escaped, so there's no tab, just '\' and 't'
pattern = "\\t123"   # this will match \t (escaping the backslash) followed by 123
re.match(pattern, string).group()   # no match
re.match(pattern, "\t123zzb").group()  # matches '\t123'

pattern = r"\\t123"
re.match(pattern, string).group()   # matches '\\t123'
```

La corrispondenza viene eseguita solo dall'inizio della stringa. Se vuoi abbinare ovunque usa `re.search` :

```
match = re.match(r"(123)", "a123zzb")

match is None
# Out: True

match = re.search(r"(123)", "a123zzb")

match.group()
# Out: '123'
```

Ricerca

```
pattern = r"(your base)"
sentence = "All your base are belong to us."
```

```
match = re.search(pattern, sentence)
match.group(1)
# Out: 'your base'

match = re.search(r"(belong.*)", sentence)
match.group(1)
# Out: 'belong to us.'
```

La ricerca viene eseguita ovunque nella stringa, diversamente da `re.match` . Puoi anche usare `re.findall` .

Puoi anche cercare all'inizio della stringa (usa `^`),

```
match = re.search(r"^123", "123zzb")
match.group(0)
# Out: '123'

match = re.search(r"^123", "a123zzb")
match is None
# Out: True
```

alla fine della stringa (usa `$`),

```
match = re.search(r"123$", "zzb123")
match.group(0)
# Out: '123'

match = re.search(r"123$", "123zzb")
match is None
# Out: True
```

o entrambi (usa sia `^` che `$`):

```
match = re.search(r"^123$", "123")
match.group(0)
# Out: '123'
```

Raggruppamento

Il raggruppamento è fatto con parentesi. Calling `group()` restituisce una stringa formata dai sottogruppi parentesi corrispondenti.

```
match.group() # Group without argument returns the entire match found
# Out: '123'
match.group(0) # Specifying 0 gives the same result as specifying no argument
# Out: '123'
```

Gli argomenti possono anche essere forniti a `group()` per recuperare un determinato sottogruppo.

Dai documenti :

> Se c'è un singolo argomento, il risultato è una singola stringa; se ci sono più

argomenti, il risultato è una tupla con un elemento per argomento.

Calling `groups()` d'altra parte, restituisce una lista di tuple contenenti i sottogruppi.

```
sentence = "This is a phone number 672-123-456-9910"
pattern = r".*(phone).*?([\d-]+)"

match = re.match(pattern, sentence)

match.groups()   # The entire match as a list of tuples of the paranthesized subgroups
# Out: ('phone', '672-123-456-9910')

m.group()        # The entire match as a string
# Out: 'This is a phone number 672-123-456-9910'

m.group(0)       # The entire match as a string
# Out: 'This is a phone number 672-123-456-9910'

m.group(1)       # The first parenthesized subgroup.
# Out: 'phone'

m.group(2)       # The second parenthesized subgroup.
# Out: '672-123-456-9910'

m.group(1, 2)    # Multiple arguments give us a tuple.
# Out: ('phone', '672-123-456-9910')
```

Gruppi con nome

```
match = re.search(r'My name is (?P<name>[A-Za-z ]+)', 'My name is John Smith')
match.group('name')
# Out: 'John Smith'

match.group(1)
# Out: 'John Smith'
```

Crea un gruppo di cattura che può essere referenziato per nome e per indice.

Gruppi non catturanti

Usando `(?:)` crea un gruppo, ma il gruppo non viene catturato. Ciò significa che puoi usarlo come gruppo, ma non inquinerà il tuo "spazio di gruppo".

```
re.match(r'(\d+)(\+(\d+))?', '11+22').groups()
# Out: ('11', '+22', '22')

re.match(r'(\d+)(?:\+(\d+))?', '11+22').groups()
# Out: ('11', '22')
```

Questo esempio corrisponde a `11+22` o `11` , ma non `11+` . Questo perché il segno `+` e il secondo termine sono raggruppati. D'altra parte, il segno `+` non viene catturato.

Escaping Special Characters

I caratteri speciali (come le parentesi `[` e `]` sotto) della classe di caratteri non sono letteralmente abbinati:

```
match = re.search(r'[b]', 'a[b]c')
match.group()
# Out: 'b'
```

Sfuggendo ai caratteri speciali, possono essere abbinati letteralmente:

```
match = re.search(r'\[b\]', 'a[b]c')
match.group()
# Out: '[b]'
```

La funzione `re.escape()` può essere utilizzata per fare questo per te:

```
re.escape('a[b]c')
# Out: 'a\\[b\\]c'
match = re.search(re.escape('a[b]c'), 'a[b]c')
match.group()
# Out: 'a[b]c'
```

La funzione `re.escape()` sfugge a tutti i caratteri speciali, quindi è utile se si sta scrivendo un'espressione regolare basata sull'input dell'utente:

```
username = 'A.C.'  # suppose this came from the user
re.findall(r'Hi {}!'.format(username), 'Hi A.C.! Hi ABCD!')
# Out: ['Hi A.C.!', 'Hi ABCD!']
re.findall(r'Hi {}!'.format(re.escape(username)), 'Hi A.C.! Hi ABCD!')
# Out: ['Hi A.C.!']
```

Sostituzione

Le sostituzioni possono essere fatte su stringhe usando `re.sub` .

Sostituire le stringhe

```
re.sub(r"t[0-9][0-9]", "foo", "my name t13 is t44 what t99 ever t44")
# Out: 'my name foo is foo what foo ever foo'
```

Utilizzando riferimenti di gruppo

Sostituzioni con un numero limitato di gruppi possono essere effettuate come segue:

```
re.sub(r"t([0-9])([0-9])", r"t\2\1", "t13 t19 t81 t25")
# Out: 't31 t91 t18 t52'
```

Tuttavia, se si crea un ID di gruppo come "10", questo non funziona : `\10` viene letto come 'ID numero 1 seguito da 0'. Quindi devi essere più specifico e usare la notazione `\g<i>` :

```
re.sub(r"t([0-9])([0-9])", r"t\g<2>\g<1>", "t13 t19 t81 t25")
# Out: 't31 t91 t18 t52'
```

Utilizzando una funzione di sostituzione

```
items = ["zero", "one", "two"]
re.sub(r"a\[([0-3])\]", lambda match: items[int(match.group(1))], "Items: a[0], a[1],
something, a[2]")
# Out: 'Items: zero, one, something, two'
```

Trova tutte le corrispondenze non sovrapposte

```
re.findall(r"[0-9]{2,3}", "some 1 text 12 is 945 here 4445588899")
# Out: ['12', '945', '444', '558', '889']
```

Notare che il `r` prima di `"[0-9]{2,3}"` dice a python di interpretare la stringa as-is; come una stringa "grezza".

Puoi anche usare `re.finditer()` che funziona allo stesso modo di `re.findall()` ma restituisce un iteratore con oggetti `SRE_Match` invece di un elenco di stringhe:

```
results = re.finditer(r"([0-9]{2,3})", "some 1 text 12 is 945 here 4445588899")
print(results)
# Out: <callable-iterator object at 0x105245890>
for result in results:
     print(result.group(0))
''' Out:
12
945
444
558
889
'''
```

Modelli precompilati

```
import re

precompiled_pattern = re.compile(r"(\d+)")
matches = precompiled_pattern.search("The answer is 41!")
matches.group(1)
# Out: 41

matches = precompiled_pattern.search("Or was it 42?")
matches.group(1)
# Out: 42
```

La compilazione di un modello consente di riutilizzarlo in seguito in un programma. Tuttavia, si noti che Python memorizza nella cache le espressioni utilizzate di recente (documenti , risposta SO), quindi *"i programmi che utilizzano solo alcune espressioni regolari alla volta non devono preoccuparsi di compilare le espressioni regolari"* .

```
import re

precompiled_pattern = re.compile(r"(.*\d+)")
matches = precompiled_pattern.match("The answer is 41!")
print(matches.group(1))
# Out: The answer is 41

matches = precompiled_pattern.match("Or was it 42?")
print(matches.group(1))
# Out: Or was it 42
```

Può essere usato con re.match ().

Verifica dei caratteri consentiti

Se vuoi verificare che una stringa contenga solo un determinato set di caratteri, in questo caso az, AZ e 0-9, puoi farlo in questo modo,

```
import re

def is_allowed(string):
    characherRegex = re.compile(r'[^a-zA-Z0-9.]')
    string = characherRegex.search(string)
    return not bool(string)

print (is_allowed("abyzABYZ0099"))
# Out: 'True'

print (is_allowed("#*@#$%^"))
# Out: 'False'
```

È anche possibile adattare la linea di espressione da `[^a-zA-Z0-9.] [^a-z0-9.]` , Per non consentire le lettere maiuscole, ad esempio.

Credito parziale: http://stackoverflow.com/a/1325265/2697955

Dividere una stringa usando le espressioni regolari

Puoi anche usare espressioni regolari per dividere una stringa. Per esempio,

```
import re
data = re.split(r'\s+', 'James 94 Samantha 417 Scarlett 74')
print( data )
# Output: ['James', '94', 'Samantha', '417', 'Scarlett', '74']
```

bandiere

Per alcuni casi speciali abbiamo bisogno di cambiare il comportamento dell'Espressione Regolare, questo viene fatto usando le bandiere. Le bandiere possono essere impostate in due modi, attraverso la parola chiave `flags` o direttamente nell'espressione.

Parola chiave delle bandiere

Di seguito un esempio per `re.search` ma funziona per la maggior parte delle funzioni nel modulo `re` .

```
m = re.search("b", "ABC")
m is None
# Out: True

m = re.search("b", "ABC", flags=re.IGNORECASE)
m.group()
# Out: 'B'

m = re.search("a.b", "A\nBC", flags=re.IGNORECASE)
m is None
# Out: True

m = re.search("a.b", "A\nBC", flags=re.IGNORECASE|re.DOTALL)
m.group()
# Out: 'A\nB'
```

Bandiere comuni

Bandiera	breve descrizione
`re.IGNORECASE` , `re.I`	Fa in modo che il modello ignori il caso
`re.DOTALL` , `re.S`	Rende `.` abbinare tutto compreso a capo
`re.MULTILINE` , `re.M`	Rende `^` corrisponde all'inizio di una linea e `$` alla fine di una linea
`re.DEBUG`	Attiva le informazioni di debug

Per l'elenco completo di tutte le flag disponibili, controlla i documenti

Bandiere in linea

Dai documenti :

> `(?iLmsux)` (Una o più lettere dal set 'i', 'L', 'm', 's', 'u', 'x'.)
>
> Il gruppo corrisponde alla stringa vuota; le lettere impostano i flag corrispondenti: re.I (ignora il caso), re.L (dipendente dalla locale), re.M (multi-line), re.S (punto corrisponde a tutti), re.U (dipendente da Unicode), e re.X (dettagliato), per l'intera espressione regolare. Ciò è utile se si desidera includere i flag come parte dell'espressione regolare, invece di passare un argomento flag alla funzione re.compile ().

Si noti che il flag (? X) cambia il modo in cui l'espressione viene analizzata. Dovrebbe essere usato prima nella stringa di espressione o dopo uno o più caratteri di spaziatura. Se ci sono caratteri non spazi bianchi prima del flag, i risultati non sono definiti.

Iterare le partite usando `re.finditer`

È possibile utilizzare `re.finditer` per `re.finditer` tutte le corrispondenze in una stringa. Questo ti dà (in confronto a `re.findall` informazioni extra, come ad esempio informazioni sulla posizione della partita nella stringa (indici):

```
import re
text = 'You can try to find an ant in this string'
pattern = 'an?\w' # find 'an' either with or without a following word character

for match in re.finditer(pattern, text):
    # Start index of match (integer)
    sStart = match.start()

    # Final index of match (integer)
    sEnd = match.end()

    # Complete match (string)
    sGroup = match.group()

    # Print match
    print('Match "{}" found at: [{},{}]'.format(sGroup, sStart,sEnd))
```

Risultato:

```
Match "an" found at: [5,7]
Match "an" found at: [20,22]
Match "ant" found at: [23,26]
```

Abbina un'espressione solo in posizioni specifiche

Spesso vuoi abbinare un'espressione solo in luoghi *specifici* (lasciandoli intatti in altri, cioè). Considera la seguente frase:

```
An apple a day keeps the doctor away (I eat an apple everyday).
```

Qui la "mela" si verifica due volte che può essere risolta con i cosiddetti *verbi di controllo backtracking* che sono supportati dal modulo **`regex`** più recente. L'idea è:

```
forget_this | or this | and this as well | (but keep this)
```

Con il nostro esempio di mela, questo sarebbe:

```
import regex as re
string = "An apple a day keeps the doctor away (I eat an apple everyday)."
rx = re.compile(r'''
```

```
    \([^()]*\) (*SKIP)(*FAIL)  # match anything in parentheses and "throw it away"
    |                          # or
    apple                      # match an apple
    ''', re.VERBOSE)
apples = rx.findall(string)
print(apples)
# only one
```

Questo corrisponde a "mela" solo quando può essere trovato al di fuori delle parentesi.

Ecco come funziona:

- Mentre guarda da **sinistra a destra** , il motore regex consuma tutto a sinistra, il `(*SKIP)` agisce come una "affermazione sempre vera". Successivamente, fallisce correttamente `(*FAIL)` e backtrack.
- Ora arriva al punto di `(*SKIP)` **da destra a sinistra** (aka mentre backtracking) dove è vietato andare ulteriormente a sinistra. Invece, al motore viene detto di gettare via qualsiasi cosa a sinistra e saltare al punto in cui è stato invocato il `(*SKIP)` .

Leggi Espressioni regolari (Regex) online: https://riptutorial.com/it/python/topic/632/espressioni-regolari--regex-

Capitolo 61: Eventi inviati dal server Python

introduzione

Server Sent Events (SSE) è una connessione unidirezionale tra un server e un client (di solito un browser Web) che consente al server di "spingere" le informazioni al client. È molto simile alle web socket e ai lunghi sondaggi. La principale differenza tra SSE e websockets è che SSE è unidirezionale, solo il server può inviare informazioni al client, dove come con i websocket, entrambi possono inviare informazioni a vicenda. Solitamente SSE è considerato molto più semplice da usare / implementare rispetto ai websocket.

Examples

Flacone SSE

```
@route("/stream")
def stream():
    def event_stream():
        while True:
            if message_to_send:
                yield "data:
                    {}\n\n".format(message_to_send)"

    return Response(event_stream(), mimetype="text/event-stream")
```

Asyncio SSE

Questo esempio utilizza la libreria SSE asyncio: https://github.com/brutasse/asyncio-sse

```
import asyncio
import sse

class Handler(sse.Handler):
    @asyncio.coroutine
    def handle_request(self):
        yield from asyncio.sleep(2)
        self.send('foo')
        yield from asyncio.sleep(2)
        self.send('bar', event='wakeup')

start_server = sse.serve(Handler, 'localhost', 8888)
asyncio.get_event_loop().run_until_complete(start_server)
asyncio.get_event_loop().run_forever()
```

Leggi Eventi inviati dal server Python online: https://riptutorial.com/it/python/topic/9100/eventi-inviati-dal-server-python

Capitolo 62: File e cartelle I / O

introduzione

Quando si tratta di memorizzare, leggere o comunicare dati, lavorare con i file di un sistema operativo è sia necessario che facile con Python. A differenza di altri linguaggi in cui l'input e l'output dei file richiedono oggetti di lettura e scrittura complessi, Python semplifica il processo richiedendo solo i comandi per aprire, leggere / scrivere e chiudere il file. Questo argomento spiega come Python può interfacciarsi con i file sul sistema operativo.

Sintassi

- file_object = open (filename [, access_mode] [, buffering])

Parametri

Parametro	Dettagli
nome del file	il percorso del tuo file o, se il file è nella directory di lavoro, il nome del file
access_mode	un valore stringa che determina come viene aperto il file
il buffering	un valore intero utilizzato per il buffering di riga opzionale

Osservazioni

Evitare l'Encoding Hell multipiattaforma

Quando si utilizza `open()` Python, è consigliabile passare sempre l'argomento di `encoding` , se si intende eseguire il codice su più piattaforme. Il motivo di ciò è che la codifica predefinita di un sistema differisce da piattaforma a piattaforma.

Mentre i sistemi `linux` usano effettivamente `utf-8` come default, questo ***non*** è necessariamente vero per MAC e Windows.

Per verificare la codifica predefinita di un sistema, prova questo:

```
import sys
sys.getdefaultencoding()
```

da qualsiasi interprete python.

Pertanto, è consigliabile separare sempre una codifica, per assicurarsi che le stringhe con cui si

sta lavorando siano codificate come ciò che si pensa di essere, garantendo la compatibilità tra piattaforme diverse.

```
with open('somefile.txt', 'r', encoding='UTF-8') as f:
    for line in f:
        print(line)
```

Examples

Modalità file

Esistono diverse modalità con cui è possibile aprire un file, specificato dal parametro `mode` . Questi includono:

- `'r'` - modalità di lettura. Il predefinito. Ti permette solo di leggere il file, non di modificarlo. Quando si utilizza questa modalità, il file deve esistere.
- `'w'` - modalità di scrittura. Creerà un nuovo file se non esiste, altrimenti cancellerà il file e ti consentirà di scrivere su di esso.
- `'a'` - modalità append. Scriverà i dati alla fine del file. Non cancella il file e il file deve esistere per questa modalità.
- `'rb'` - modalità di lettura in binario. Questo è simile a `r` tranne che la lettura è forzata in modalità binaria. Anche questa è una scelta predefinita.
- `'r+'` - modalità di lettura più modalità di scrittura allo stesso tempo. Ciò consente di leggere e scrivere in file contemporaneamente senza dover utilizzare `r` e `w` .
- `'rb+'` - modalità di lettura e scrittura in binario. Lo stesso di `r+` tranne che i dati sono in binario
- `'wb'` - modalità di scrittura in binario. Lo stesso di `w` eccetto che i dati sono in binario.
- `'w+'` - modalità di scrittura e lettura. Esattamente come `r+` ma se il file non esiste, ne viene creato uno nuovo. Altrimenti, il file viene sovrascritto.
- `'wb+'` - modalità di scrittura e lettura in modalità binaria. Lo stesso di `w+` ma i dati sono in binario.
- `'ab'` - l'aggiunta in modalità binaria. Simile ad `a` tranne che i dati sono in binario.
- `'a+'` - modalità di aggiunta e lettura. Simile a `w+` in quanto creerà un nuovo file se il file non esiste. Altrimenti, il puntatore del file si trova alla fine del file, se esiste.
- `'ab+'` - modalità di aggiunta e lettura in binario. Uguale `a+` tranne che i dati sono in binario.

```
with open(filename, 'r') as f:
    f.read()
```

```
with open(filename, 'w') as f:
    f.write(filedata)
with open(filename, 'a') as f:
    f.write('\n' + newdata)
```

	r	r +	w	w +	un	un +
Leggere	✔	✔	✘	✔	✘	✔
Scrivi	✘	✔	✔	✔	✔	✔
Crea un file	✘	✘	✔	✔	✔	✔
Cancella il file	✘	✘	✔	✔	✘	✘
Posizione iniziale	Inizio	Inizio	Inizio	Inizio	Fine	Fine

Python 3 ha aggiunto una nuova modalità per la `exclusive creation` modo da non troncare o sovrascrivere accidentalmente e file esistenti.

- `'x'` : aperto per la creazione esclusiva, genererà `FileExistsError` se il file esiste già
- `'xb'` : aperto per la modalità di scrittura di creazione esclusiva in binario. Lo stesso di `x` eccetto che i dati sono in binario.
- `'x+'` - modalità di lettura e scrittura. Simile a `w+` in quanto creerà un nuovo file se il file non esiste. Altrimenti, genererà `FileExistsError` .
- `'xb+'` - modalità di scrittura e lettura. Esattamente come `x+` ma i dati sono binari

	X	x +
Leggere	✘	✔
Scrivi	✔	✔
Crea un file	✔	✔
Cancella il file	✘	✘
Posizione iniziale	Inizio	Inizio

Consentire a uno di scrivere il codice aperto del file in modo più pignolo:

Python 3.x 3.3

```
try:
    with open("fname", "r") as fout:
        # Work with your open file
except FileExistsError:
    # Your error handling goes here
```

In Python 2 avresti fatto qualcosa di simile

Python 2.x 2.0

```
import os.path
if os.path.isfile(fname):
    with open("fname", "w") as fout:
        # Work with your open file
else:
    # Your error handling goes here
```

Leggere un file riga per riga

Il modo più semplice per scorrere su un file riga per riga:

```
with open('myfile.txt', 'r') as fp:
    for line in fp:
        print(line)
```

`readline()` consente un controllo più granulare su iterazione linea per riga. L'esempio sotto è equivalente a quello sopra:

```
with open('myfile.txt', 'r') as fp:
    while True:
        cur_line = fp.readline()
        # If the result is an empty string
        if cur_line == '':
            # We have reached the end of the file
            break
        print(cur_line)
```

Usare l'iteratore loop for e readline () insieme è considerato una cattiva pratica.

Più comunemente, il metodo `readlines()` viene utilizzato per memorizzare una raccolta iterabile delle righe del file:

```
with open("myfile.txt", "r") as fp:
    lines = fp.readlines()
for i in range(len(lines)):
    print("Line " + str(i) + ": " + line)
```

Questo dovrebbe stampare quanto segue:

Linea 0: ciao

Linea 1: mondo

Ottenere l'intero contenuto di un file

Il metodo preferito di file i / o è quello di utilizzare la parola chiave `with` . Ciò garantirà che l'handle del file venga chiuso una volta completata la lettura o la scrittura.

```
with open('myfile.txt') as in_file:
    content = in_file.read()

print(content)
```

o, per gestire la chiusura del file manualmente, si può rinunciare `with` e semplicemente chiamare `close` te stesso:

```
in_file = open('myfile.txt', 'r')
content = in_file.read()
print(content)
in_file.close()
```

Tieni presente che senza utilizzare un'istruzione `with` , potresti accidentalmente mantenere il file aperto nel caso si verificasse un'eccezione imprevista in questo modo:

```
in_file = open('myfile.txt', 'r')
raise Exception("oops")
in_file.close()  # This will never be called
```

Scrivere su un file

```
with open('myfile.txt', 'w') as f:
    f.write("Line 1")
    f.write("Line 2")
    f.write("Line 3")
    f.write("Line 4")
```

Se apri `myfile.txt` , vedrai che il suo contenuto è:

> Linea 1 Linea 2 Linea 3 Linea 4

Python non aggiunge automaticamente interruzioni di riga, è necessario farlo manualmente:

```
with open('myfile.txt', 'w') as f:
    f.write("Line 1\n")
    f.write("Line 2\n")
    f.write("Line 3\n")
    f.write("Line 4\n")
```

> Linea 1
> Linea 2
> Linea 3
> Linea 4

Non utilizzare `os.linesep` come terminatore di riga durante la scrittura di file aperti in modalità testo (impostazione predefinita); usa invece `\n` .

Se si desidera specificare una codifica, è sufficiente aggiungere il parametro di `encoding` alla funzione di `open` :

```
with open('my_file.txt', 'w', encoding='utf-8') as f:
    f.write('utf-8 text')
```

È anche possibile utilizzare l'istruzione print per scrivere su un file. Le meccaniche sono diverse in Python 2 vs Python 3, ma il concetto è lo stesso in quanto puoi prendere l'output che sarebbe andato sullo schermo e invece inviarlo a un file.

Python 3.x 3.0

```
with open('fred.txt', 'w') as outfile:
    s = "I'm Not Dead Yet!"
    print(s) # writes to stdout
    print(s, file = outfile) # writes to outfile

    #Note: it is possible to specify the file parameter AND write to the screen
    #by making sure file ends up with a None value either directly or via a variable
    myfile = None
    print(s, file = myfile) # writes to stdout
    print(s, file = None)   # writes to stdout
```

In Python 2 avresti fatto qualcosa di simile

Python 2.x 2.0

```
outfile = open('fred.txt', 'w')
s = "I'm Not Dead Yet!"
print s   # writes to stdout
print >> outfile, s   # writes to outfile
```

A differenza dell'utilizzo della funzione di scrittura, la funzione di stampa aggiunge automaticamente interruzioni di riga.

Copia del contenuto di un file in un altro file

```
with open(input_file, 'r') as in_file, open(output_file, 'w') as out_file:
    for line in in_file:
        out_file.write(line)
```

- Usando il modulo `shutil` :

```
import shutil
shutil.copyfile(src, dst)
```

Controlla se esiste un file o percorso

Impiega lo stile di codifica EAFP e `try` ad aprirlo.

```
import errno

try:
    with open(path) as f:
        # File exists
```

```
except IOError as e:
    # Raise the exception if it is not ENOENT (No such file or directory)
    if e.errno != errno.ENOENT:
        raise
    # No such file or directory
```

Ciò eviterà anche le condizioni di gara se un altro processo ha cancellato il file tra il controllo e quando viene utilizzato. Questa condizione di competizione potrebbe verificarsi nei seguenti casi:

- Usando il modulo `os` :

```
import os
os.path.isfile('/path/to/some/file.txt')
```

Python 3.x 3.4

- Utilizzando `pathlib` :

```
import pathlib
path = pathlib.Path('/path/to/some/file.txt')
if path.is_file():
    ...
```

Per verificare se esiste un determinato percorso o meno, è possibile seguire la procedura EAFP sopra indicata oppure controllare esplicitamente il percorso:

```
import os
path = "/home/myFiles/directory1"

if os.path.exists(path):
    ## Do stuff
```

Copia un albero di directory

```
import shutil
source='//192.168.1.2/Daily Reports'
destination='D:\\Reports\\Today'
shutil.copytree(source, destination)
```

La directory di destinazione **non deve esistere** già.

Iterare i file (in modo ricorsivo)

Per ripetere tutti i file, incluso nelle sottodirectory, utilizzare os.walk:

```
import os
for root, folders, files in os.walk(root_dir):
    for filename in files:
        print root, filename
```

root_dir può essere "." per iniziare dalla directory corrente o da qualsiasi altro percorso da cui partire.

Python 3.x 3.5

Se desideri anche ottenere informazioni sul file, puoi utilizzare il metodo più efficiente os.scandir in questo modo:

```
for entry in os.scandir(path):
   if not entry.name.startswith('.') and entry.is_file():
       print(entry.name)
```

Leggi un file tra un intervallo di linee

Quindi supponiamo di voler ripetere solo tra alcune linee specifiche di un file

Puoi fare uso di `itertools` per quello

```
import itertools

with open('myfile.txt', 'r') as f:
    for line in itertools.islice(f, 12, 30):
        # do something here
```

Questo leggerà le righe da 13 a 20 come nell'indicizzazione python inizia da 0. Quindi la riga numero 1 è indicizzata come 0

Come può anche leggere alcune righe aggiuntive facendo uso della parola chiave `next()` qui.

E quando usi l'oggetto file come un iterabile, per favore non usare l'istruzione `readline()` qui perché le due tecniche di attraversamento di un file non devono essere mescolate insieme

Accesso casuale ai file tramite mmap

L'uso del modulo `mmap` consente all'utente di accedere in modo casuale a posizioni in un file mappando il file in memoria. Questa è un'alternativa all'uso delle normali operazioni sui file.

```
import mmap

with open('filename.ext', 'r') as fd:
    # 0: map the whole file
    mm = mmap.mmap(fd.fileno(), 0)

    # print characters at indices 5 through 10
    print mm[5:10]

    # print the line starting from mm's current position
    print mm.readline()

    # write a character to the 5th index
    mm[5] = 'a'

    # return mm's position to the beginning of the file
```

```
    mm.seek(0)

    # close the mmap object
    mm.close()
```

Sostituzione del testo in un file

```
import fileinput

replacements = {'Search1': 'Replace1',
                'Search2': 'Replace2'}

for line in fileinput.input('filename.txt', inplace=True):
    for search_for in replacements:
        replace_with = replacements[search_for]
        line = line.replace(search_for, replace_with)
    print(line, end='')
```

Verifica se un file è vuoto

```
>>> import os
>>> os.stat(path_to_file).st_size == 0
```

o

```
>>> import os
>>> os.path.getsize(path_to_file) > 0
```

Tuttavia, entrambi genereranno un'eccezione se il file non esiste. Per evitare di dover rilevare un errore simile, procedere come segue:

```
import os
def is_empty_file(fpath):
    return os.path.isfile(fpath) and os.path.getsize(fpath) > 0
```

che restituirà un valore `bool` .

Leggi File e cartelle I / O online: https://riptutorial.com/it/python/topic/267/file-e-cartelle-i---o

Capitolo 63: Filtro

Sintassi

- filtro (funzione, iterabile)
- itertools.ifilter (function, iterable)
- future_builtins.filter (funzione, iterabile)
- itertools.ifilterfalse (funzione, iterabile)
- itertools.filterfalse (function, iterable)

Parametri

Parametro	Dettagli
funzione	*callable* che determina la condizione o `None` quindi usa la funzione identity per il filtraggio (*solo posizionale*)
iterabile	iterabile che verrà filtrato (*solo posizionale*)

Osservazioni

Nella maggior parte dei casi , l'espressione di comprensione o generatore è più leggibile, più potente e più efficiente di `filter()` o `ifilter()` .

Examples

Uso di base del filtro

Per `filter` elementi degli scarti di una sequenza in base ad alcuni criteri:

```
names = ['Fred', 'Wilma', 'Barney']

def long_name(name):
    return len(name) > 5
```

Python 2.x 2.0

```
filter(long_name, names)
# Out: ['Barney']

[name for name in names if len(name) > 5] # equivalent list comprehension
# Out: ['Barney']


from itertools import ifilter
ifilter(long_name, names)       # as generator (similar to python 3.x filter builtin)
```

```
# Out: <itertools.ifilter at 0x4197e10>
list(ifilter(long_name, names)) # equivalent to filter with lists
# Out: ['Barney']

(name for name in names if len(name) > 5) # equivalent generator expression
# Out: <generator object <genexpr> at 0x0000000003FD5D38>
```

Python 2.x 2.6

```
# Besides the options for older python 2.x versions there is a future_builtin function:
from future_builtins import filter
filter(long_name, names)       # identical to itertools.ifilter
# Out: <itertools.ifilter at 0x3eb0ba8>
```

Python 3.x 3.0

```
filter(long_name, names)        # returns a generator
# Out: <filter at 0x1fc6e443470>
list(filter(long_name, names))  # cast to list
# Out: ['Barney']

(name for name in names if len(name) > 5) # equivalent generator expression
# Out: <generator object <genexpr> at 0x000001C6F49BF4C0>
```

Filtro senza funzione

Se il parametro function è `None` , verrà utilizzata la funzione identity:

```
list(filter(None, [1, 0, 2, [], '', 'a']))  # discards 0, [] and ''
# Out: [1, 2, 'a']
```

Python 2.x 2.0.1

```
[i for i in [1, 0, 2, [], '', 'a'] if i] # equivalent list comprehension
```

Python 3.x 3.0.0

```
(i for i in [1, 0, 2, [], '', 'a'] if i) # equivalent generator expression
```

Filtro come controllo di cortocircuito

`filter` (python 3.x) e `ifilter` (python 2.x) restituiscono un generatore in modo che possano essere molto utili quando si crea un test di cortocircuito come `or` o `and` :

Python 2.x 2.0.1

```
 # not recommended in real use but keeps the example short:
from itertools import ifilter as filter
```

Python 2.x 2.6.1

```
 from future_builtins import filter
```

Per trovare il primo elemento che è inferiore a 100:

```
car_shop = [('Toyota', 1000), ('rectangular tire', 80), ('Porsche', 5000)]
def find_something_smaller_than(name_value_tuple):
    print('Check {0}, {1}$'.format(*name_value_tuple)
    return name_value_tuple[1] < 100
next(filter(find_something_smaller_than, car_shop))
# Print: Check Toyota, 1000$
#        Check rectangular tire, 80$
# Out: ('rectangular tire', 80)
```

La funzione `next` dà il prossimo (in questo caso il primo) elemento ed è quindi il motivo per cui è un cortocircuito.

Funzione complementare: filterfalse, ifilterfalse

C'è una funzione complementare per il `filter` nel modulo- `itertools` :

Python 2.x 2.0.1

```
 # not recommended in real use but keeps the example valid for python 2.x and python 3.x
from itertools import ifilterfalse as filterfalse
```

Python 3.x 3.0.0

```
from itertools import filterfalse
```

che funziona esattamente come il `filter` *generatore* ma mantiene solo gli elementi che sono `False` :

```
# Usage without function (None):
list(filterfalse(None, [1, 0, 2, [], '', 'a']))  # discards 1, 2, 'a'
# Out: [0, [], '']
```

```
# Usage with function
names = ['Fred', 'Wilma', 'Barney']

def long_name(name):
    return len(name) > 5

list(filterfalse(long_name, names))
# Out: ['Fred', 'Wilma']
```

```
# Short-circuit useage with next:
car_shop = [('Toyota', 1000), ('rectangular tire', 80), ('Porsche', 5000)]
def find_something_smaller_than(name_value_tuple):
    print('Check {0}, {1}$'.format(*name_value_tuple)
    return name_value_tuple[1] < 100
next(filterfalse(find_something_smaller_than, car_shop))
# Print: Check Toyota, 1000$
# Out: ('Toyota', 1000)
```

```
# Using an equivalent generator:
car_shop = [('Toyota', 1000), ('rectangular tire', 80), ('Porsche', 5000)]
generator = (car for car in car_shop if not car[1] < 100)
next(generator)
```

Leggi Filtro online: https://riptutorial.com/it/python/topic/201/filtro

Capitolo 64: Formattazione della data

Examples

Tempo tra due date

```
from datetime import datetime

a = datetime(2016,10,06,0,0,0)
b = datetime(2016,10,01,23,59,59)

a-b
# datetime.timedelta(4, 1)

(a-b).days
# 4
(a-b).total_seconds()
# 518399.0
```

Analisi della stringa sull'oggetto datetime

Utilizza i codici di formato standard C.

```
from datetime import datetime
datetime_string = 'Oct 1 2016, 00:00:00'
datetime_string_format = '%b %d %Y, %H:%M:%S'
datetime.strptime(datetime_string, datetime_string_format)
# datetime.datetime(2016, 10, 1, 0, 0)
```

Uscita dell'oggetto datetime alla stringa

Utilizza i codici di formato standard C.

```
from datetime import datetime
datetime_for_string = datetime(2016,10,1,0,0)
datetime_string_format = '%b %d %Y, %H:%M:%S'
datetime.strftime(datetime_for_string,datetime_string_format)
# Oct 01 2016, 00:00:00
```

Leggi Formattazione della data online: https://riptutorial.com/it/python/topic/7284/formattazione-della-data

Capitolo 65: Formattazione di stringhe

introduzione

Quando si memorizzano e si trasformano i dati per gli utenti, la formattazione delle stringhe può diventare molto importante. Python offre una vasta gamma di metodi di formattazione delle stringhe che sono delineati in questo argomento.

Sintassi

- "{}". format (42) ==> "42"
- "{0}". Format (42) ==> "42"
- "{0: .2f}". Format (42) ==> "42.00"
- "{0: .0f}". Format (42.1234) ==> "42"
- "{answer}". format (no_answer = 41, answer = 42) ==> "42"
- "{answer: .2f}". format (no_answer = 41, answer = 42) ==> "42.00"
- "{[chiave]}". format ({'chiave': 'valore'}) ==> "valore"
- "{[1]}". Format (['zero', 'one', 'two']) ==> "one"
- "{answer} = {answer}". format (answer = 42) ==> "42 = 42"
- " .join (['stack', 'overflow']) ==> "stack overflow"

Osservazioni

- Dovresti controllare PyFormat.info per un'introduzione / spiegazione molto approfondita e gentile di come funziona.

Examples

Nozioni di base sulla formattazione delle stringhe

```
foo = 1
bar = 'bar'
baz = 3.14
```

È possibile utilizzare `str.format` per formattare l'output. Le coppie di parentesi vengono sostituite con argomenti nell'ordine in cui vengono passati gli argomenti:

```
print('{}, {} and {}'.format(foo, bar, baz))
# Out: "1, bar and 3.14"
```

Gli indici possono anche essere specificati all'interno delle parentesi. I numeri corrispondono agli indici degli argomenti passati alla funzione `str.format` (basata su 0).

```
print('{0}, {1}, {2}, and {1}'.format(foo, bar, baz))
```

```
# Out: "1, bar, 3.14, and bar"
print('{0}, {1}, {2}, and {3}'.format(foo, bar, baz))
# Out: index out of range error
```

Gli argomenti con nome possono essere utilizzati anche:

```
print("X value is: {x_val}. Y value is: {y_val}.".format(x_val=2, y_val=3))
# Out: "X value is: 2. Y value is: 3."
```

Gli attributi degli oggetti possono essere referenziati quando vengono passati in `str.format` :

```
class AssignValue(object):
    def __init__(self, value):
        self.value = value
my_value = AssignValue(6)
print('My value is: {0.value}'.format(my_value))  # "0" is optional
# Out: "My value is: 6"
```

Le chiavi del dizionario possono essere utilizzate anche:

```
my_dict = {'key': 6, 'other_key': 7}
print("My other key is: {0[other_key]}".format(my_dict))  # "0" is optional
# Out: "My other key is: 7"
```

Lo stesso vale per gli indici list e tuple:

```
my_list = ['zero', 'one', 'two']
print("2nd element is: {0[2]}".format(my_list))  # "0" is optional
# Out: "2nd element is: two"
```

> Nota: oltre a `str.format` , Python fornisce anche l'operatore modulo `%` - noto anche come *operatore di formattazione* o *interpolazione* della *stringa* (consultare PEP 3101) - per la formattazione delle stringhe. `str.format` è un successore di `%` e offre una maggiore flessibilità, ad esempio facilitando l'esecuzione di più sostituzioni.

Oltre agli indici argomento, puoi anche includere una *specifica di formato* all'interno delle parentesi graffe. Questa è un'espressione che segue regole particolari e deve essere preceduto da due punti (`:`). Vedi i documenti per una descrizione completa delle specifiche di formato. Un esempio di specifica di formato è la direttiva di allineamento `:~^20` (`^` indica allineamento centrale, larghezza totale 20, riempimento con carattere ~):

```
'{:~^20}'.format('centered')
# Out: '~~~~~~centered~~~~~~'
```

`format` consente un comportamento non possibile con `%` , ad esempio la ripetizione di argomenti:

```
t = (12, 45, 22222, 103, 6)
print '{0} {2} {1} {2} {3} {2} {4} {2}'.format(*t)
# Out: 12 22222 45 22222 103 22222 6 22222
```

Poiché il `format` è una funzione, può essere utilizzato come argomento in altre funzioni:

```
number_list = [12,45,78]
print map('the number is {}'.format, number_list)
# Out: ['the number is 12', 'the number is 45', 'the number is 78']


from datetime import datetime,timedelta

once_upon_a_time = datetime(2010, 7, 1, 12, 0, 0)
delta = timedelta(days=13, hours=8,  minutes=20)

gen = (once_upon_a_time + x * delta for x in xrange(5))

print '\n'.join(map('{:%Y-%m-%d %H:%M:%S}'.format, gen))
#Out: 2010-07-01 12:00:00
#     2010-07-14 20:20:00
#     2010-07-28 04:40:00
#     2010-08-10 13:00:00
#     2010-08-23 21:20:00
```

Allineamento e imbottitura

Python 2.x 2.6

Il metodo `format()` può essere usato per cambiare l'allineamento della stringa. Devi farlo con un'espressione di formato del modulo `:[fill_char][align_operator][width]` dove `align_operator` è uno di:

- `<` forza il campo a essere allineato a sinistra all'interno della `width` .
- `>` forza il campo a essere allineato a destra all'interno della `width` .
- `^` forza il campo a essere centrato nella `width` .
- `=` forza il padding da posizionare dopo il segno (solo tipi numerici).

`fill_char` (se omesso default è spazio bianco) è il carattere usato per il padding.

```
'{:~<9s}, World'.format('Hello')
# 'Hello~~~~, World'

'{:~>9s}, World'.format('Hello')
# '~~~~Hello, World'

'{:~^9s}'.format('Hello')
# '~~Hello~~'

'{:0=6d}'.format(-123)
# '-00123'
```

Nota: è possibile ottenere gli stessi risultati usando le funzioni stringa `ljust()` , `rjust()` , `center()` , `zfill()` , tuttavia queste funzioni sono deprecate dalla versione 2.5.

Formato letterale (stringa-f)

Le stringhe in formato letterale sono state introdotte in PEP 498 (Python3.6 e versioni successive),

consentendo di anteporre `f` all'inizio di una stringa letterale per applicare efficacemente `.format` su di esso con tutte le variabili nell'ambito corrente.

```
>>> foo = 'bar'
>>> f'Foo is {foo}'
'Foo is bar'
```

Funziona anche con stringhe di formato più avanzate, tra cui l'allineamento e la notazione dei punti.

```
>>> f'{foo:^7s}'
'  bar  '
```

Nota: il `f''` non denota un tipo particolare come `b''` per `bytes` o `u''` per `unicode` in python2. La formatura viene immediatamente applicata, risultante in un normale stirng.

Le stringhe di formato possono anche essere *annidate* :

```
>>> price = 478.23
>>> f"{f'${price:0.2f}':*>20s}"
'*************$478.23'
```

Le espressioni in una stringa di stringa vengono valutate nell'ordine da sinistra a destra. Questo è rilevabile solo se le espressioni hanno effetti collaterali:

```
>>> def fn(l, incr):
...     result = l[0]
...     l[0] += incr
...     return result
...
>>> lst = [0]
>>> f'{fn(lst,2)} {fn(lst,3)}'
'0 2'
>>> f'{fn(lst,2)} {fn(lst,3)}'
'5 7'
>>> lst
[10]
```

Formattazione delle stringhe con datetime

Qualsiasi classe può configurare la propria sintassi di formattazione delle stringhe tramite il metodo `__format__` . Un tipo nella libreria Python standard che ne fa un pratico uso è il tipo `datetime` , dove si possono usare i codici di formattazione `strftime` direttamente all'interno di `str.format` :

```
>>> from datetime import datetime
>>> 'North America: {dt:%m/%d/%Y}.  ISO: {dt:%Y-%m-%d}.'.format(dt=datetime.now())
'North America: 07/21/2016.  ISO: 2016-07-21.'
```

Un elenco completo di elenchi di formattatori datetime è disponibile nella documentazione ufficiale .

Formato utilizzando Getitem e Getattr

Qualsiasi struttura dati che supporta `__getitem__` può avere la sua struttura nidificata formattata:

```
person = {'first': 'Arthur', 'last': 'Dent'}
'{p[first]} {p[last]}'.format(p=person)
# 'Arthur Dent'
```

È possibile accedere agli attributi degli oggetti utilizzando `getattr()` :

```
class Person(object):
    first = 'Zaphod'
    last = 'Beeblebrox'

'{p.first} {p.last}'.format(p=Person())
# 'Zaphod Beeblebrox'
```

Formattazione mobile

```
>>> '{0:.0f}'.format(42.12345)
'42'

>>> '{0:.1f}'.format(42.12345)
'42.1'

>>> '{0:.3f}'.format(42.12345)
'42.123'

>>> '{0:.5f}'.format(42.12345)
'42.12345'

>>> '{0:.7f}'.format(42.12345)
'42.1234500'
```

Stessa attesa per un altro modo di referenziare:

```
>>> '{:.3f}'.format(42.12345)
'42.123'

>>> '{answer:.3f}'.format(answer=42.12345)
'42.123'
```

I numeri in virgola mobile possono anche essere formattati in notazione scientifica o in percentuali:

```
>>> '{0:.3e}'.format(42.12345)
'4.212e+01'

>>> '{0:.0%}'.format(42.12345)
'4212%'
```

Puoi anche combinare le notazioni `{0}` e `{name}` . Ciò è particolarmente utile quando si desidera arrotondare tutte le variabili a un numero predefinito di decimali *con 1 dichiarazione* :

```
>>> s = 'Hello'
>>> a, b, c = 1.12345, 2.34567, 34.5678
>>> digits = 2

>>> '{0}! {1:.{n}f}, {2:.{n}f}, {3:.{n}f}'.format(s, a, b, c, n=digits)
'Hello! 1.12, 2.35, 34.57'
```

Formattazione dei valori numerici

Il metodo `.format()` può interpretare un numero in diversi formati, come ad esempio:

```
>>> '{:c}'.format(65)    # Unicode character
'A'

>>> '{:d}'.format(0x0a)  # base 10
'10'

>>> '{:n}'.format(0x0a)  # base 10 using current locale for separators
'10'
```

Formatta interi in basi diverse (esadecimale, ottale, binario)

```
>>> '{0:x}'.format(10) # base 16, lowercase - Hexadecimal
'a'

>>> '{0:X}'.format(10) # base 16, uppercase - Hexadecimal
'A'

>>> '{:o}'.format(10) # base 8 - Octal
'12'

>>> '{:b}'.format(10) # base 2 - Binary
'1010'

>>> '{0:#b}, {0:#o}, {0:#x}'.format(42) # With prefix
'0b101010, 0o52, 0x2a'

>>> '8 bit: {0:08b}; Three bytes: {0:06x}'.format(42) # Add zero padding
'8 bit: 00101010; Three bytes: 00002a'
```

Usa la formattazione per convertire una tupla fluttuante RGB in una stringa esadecimale a colori:

```
>>> r, g, b = (1.0, 0.4, 0.0)
>>> '#{:02X}{:02X}{:02X}'.format(int(255 * r), int(255 * g), int(255 * b))
'#FF6600'
```

Solo i numeri interi possono essere convertiti:

```
>>> '{:x}'.format(42.0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Unknown format code 'x' for object of type 'float'
```

Formattazione personalizzata per una classe

Nota:

> Tutto quanto segue si applica al metodo `str.format` e alla funzione di `format` . Nel testo qui sotto, i due sono intercambiabili.

Per ogni valore passato alla funzione `format` , Python cerca un metodo `__format__` per quell'argomento. La tua classe personalizzata può quindi avere il proprio metodo `__format__` per determinare in che modo la funzione di `format` visualizzerà e formatterà la tua classe e i suoi attributi.

Questo è diverso dal metodo `__str__` , poiché nel metodo `__format__` è possibile tenere conto del linguaggio di formattazione, inclusi allineamento, larghezza del campo ecc. E anche (se lo si desidera) implementare i propri specificatori di formato e le proprie estensioni della lingua di formattazione. 1

```
object.__format__(self, format_spec)
```

Per esempio :

```
# Example in Python 2 - but can be easily applied to Python 3

class Example(object):
    def __init__(self,a,b,c):
        self.a, self.b, self.c = a,b,c

    def __format__(self, format_spec):
        """ Implement special semantics for the 's' format specifier """
        # Reject anything that isn't an s
        if format_spec[-1] != 's':
            raise ValueError('{} format specifier not understood for this object',
format_spec[:-1])

        # Output in this example will be (<a>,<b>,<c>)
        raw = "(" + ",".join([str(self.a), str(self.b), str(self.c)]) + ")"
        # Honor the format language by using the inbuilt string format
        # Since we know the original format_spec ends in an 's'
        # we can take advantage of the str.format method with a
        # string argument we constructed above
        return "{r:{f}}".format( r=raw, f=format_spec )

inst = Example(1,2,3)
print "{0:>20s}".format( inst )
# out :              (1,2,3)
# Note how the right align and field width of 20 has been honored.
```

Nota:

> Se la tua classe personalizzata non ha un metodo `__format__` personalizzato e un'istanza della classe viene passata alla funzione di `format` , **Python2** utilizzerà sempre il valore di ritorno del metodo `__str__` o il metodo `__repr__` per determinare cosa stampare (e se nessuno dei due esiste allora il verrà usato default `repr`) e sarà necessario utilizzare l' `s` formato `s` per formattarlo. Con **Python3** , per passare la tua classe personalizzata alla funzione di `format` , dovrai definire il metodo `__format__` sulla

tua classe personalizzata.

Formattazione nidificata

Alcuni formati possono assumere parametri aggiuntivi, come la larghezza della stringa formattata o l'allineamento:

```
>>> '{:.>10}'.format('foo')
'.......foo'
```

Questi possono anche essere forniti come parametri da `format` nidificando di più `{}` all'interno di `{}` :

```
>>> '{:.>{}}'.format('foo', 10)
'.......foo'
'{:{}{}{}}'.format('foo', '*', '^', 15)
'******foo******'
```

Nell'ultimo esempio, la stringa di formato `'{:{}{}{}}'` viene modificata in `'{:*^15}'` (cioè "centro e pad con * fino a una lunghezza totale di 15") prima di applicarlo al stringa vera `'foo'` da formattare in questo modo.

Questo può essere utile nei casi in cui i parametri non sono noti in anticipo, per esempio quando si allineano i dati tabulari:

```
>>> data = ["a", "bbbbbbb", "ccc"]
>>> m = max(map(len, data))
>>> for d in data:
...     print('{:>{}}'.format(d, m))
      a
bbbbbbb
    ccc
```

Stringhe di imbottitura e troncatura, combinate

Supponi di voler stampare le variabili in una colonna di 3 caratteri.

Nota: il raddoppio `{` e `}` li sfugge.

```
s = """

pad
{{:3}}             :{a:3}:

truncate
{{:.3}}            :{e:.3}:

combined
{{:>3.3}}          :{a:>3.3}:
{{:3.3}}           :{a:3.3}:
{{:3.3}}           :{c:3.3}:
{{:3.3}}           :{e:3.3}:
```

```
"""

print (s.format(a="1"*1, c="3"*3, e="5"*5))
```

Produzione:

```
pad
{:3}              :1  :

truncate
{:.3}             :555:

combined
{:>3.3}           :  1:
{:3.3}            :1  :
{:3.3}            :333:
{:3.3}            :555:
```

Segnaposto nominati

Le stringhe di formato possono contenere segnaposti denominati che vengono interpolati utilizzando gli argomenti delle parole chiave per `format` .

Usare un dizionario (Python 2.x)

```
>>> data = {'first': 'Hodor', 'last': 'Hodor!'}
>>> '{first} {last}'.format(**data)
'Hodor Hodor!'
```

Usare un dizionario (Python 3.2+)

```
>>> '{first} {last}'.format_map(data)
'Hodor Hodor!'
```

`str.format_map` consente di utilizzare i dizionari senza doverli prima decomprimere. Anche la classe di `data` (che potrebbe essere un tipo personalizzato) viene utilizzata al posto di un `dict` appena riempito.

Senza un dizionario:

```
>>> '{first} {last}'.format(first='Hodor', last='Hodor!')
'Hodor Hodor!'
```

Leggi Formattazione di stringhe online: https://riptutorial.com/it/python/topic/1019/formattazione-di-stringhe

Capitolo 66: Funzione mappa

Sintassi

- map (function, iterable [, * additional_iterables])
- future_builtins.map (funzione, iterabile [, * additional_iterables])
- itertools.imap (function, iterable [, * additional_iterables])

Parametri

Parametro	Dettagli
funzione	funzione per la mappatura (deve prendere tutti i parametri quanti sono i iterabili) (*solo posizionali*)
iterabile	la funzione è applicata a ciascun elemento del iterabile (*solo posizionale*)
* additional_iterables	vedi iterabile, ma quanti ne vuoi (*opzionale* , *solo posizionale*)

Osservazioni

Tutto ciò che può essere fatto con la `map` può anche essere fatto con le `comprehensions` :

```
list(map(abs, [-1,-2,-3]))     # [1, 2, 3]
[abs(i) for i in [-1,-2,-3]]   # [1, 2, 3]
```

Anche se avresti bisogno di `zip` se hai più iterabili:

```
import operator
alist = [1,2,3]
list(map(operator.add, alist, alist))  # [2, 4, 6]
[i + j for i, j in zip(alist, alist)]  # [2, 4, 6]
```

La comprensione delle liste è efficiente e può essere più veloce della `map` in molti casi, quindi prova i tempi di entrambi gli approcci se la velocità è importante per te.

Examples

Uso di base della mappa, itertools.imap e future_builtins.map

La funzione map è la più semplice tra i built-in Python utilizzati per la programmazione funzionale. `map()` applica una funzione specificata a ciascun elemento in un iterabile:

```
names = ['Fred', 'Wilma', 'Barney']
```

Python 3.x 3.0

```
map(len, names)  # map in Python 3.x is a class; its instances are iterable
# Out: <map object at 0x00000198B32E2CF8>
```

Una `map` compatibile con Python 3 è inclusa nel modulo `future_builtins` :

Python 2.x 2.6

```
from future_builtins import map  # contains a Python 3.x compatible map()
map(len, names)                  # see below
# Out: <itertools.imap instance at 0x3eb0a20>
```

In alternativa, in Python 2 si può usare `imap` da `itertools` per ottenere un generatore

Python 2.x 2.3

```
map(len, names)   # map() returns a list
# Out: [4, 5, 6]

from itertools import imap
imap(len, names)  # itertools.imap() returns a generator
# Out: <itertools.imap at 0x405ea20>
```

Il risultato può essere convertito esplicitamente in una `list` per rimuovere le differenze tra Python 2 e 3:

```
list(map(len, names))
# Out: [4, 5, 6]
```

`map()` può essere sostituito da una *comprensione di lista* equivalente o *un'espressione di generatore* :

```
[len(item) for item in names] # equivalent to Python 2.x map()
# Out: [4, 5, 6]

(len(item) for item in names) # equivalent to Python 3.x map()
# Out: <generator object <genexpr> at 0x00000195888D5FC0>
```

Mappare ciascun valore in un iterabile

Ad esempio, puoi prendere il valore assoluto di ogni elemento:

```
list(map(abs, (1, -1, 2, -2, 3, -3))) # the call to `list` is unnecessary in 2.x
# Out: [1, 1, 2, 2, 3, 3]
```

La funzione anonima supporta anche la mappatura di un elenco:

```
map(lambda x:x*2, [1, 2, 3, 4, 5])
```

```
# Out: [2, 4, 6, 8, 10]
```

o conversione dei valori decimali in percentuali:

```
def to_percent(num):
    return num * 100

list(map(to_percent, [0.95, 0.75, 1.01, 0.1]))
# Out: [95.0, 75.0, 101.0, 10.0]
```

o convertire dollari in euro (dato un tasso di cambio):

```
from functools import partial
from operator import mul

rate = 0.9  # fictitious exchange rate, 1 dollar = 0.9 euros
dollars = {'under_my_bed': 1000,
           'jeans': 45,
           'bank': 5000}

sum(map(partial(mul, rate), dollars.values()))
# Out: 5440.5
```

`functools.partial` è un modo conveniente per fissare i parametri delle funzioni in modo che possano essere utilizzate con la `map` anziché utilizzare `lambda` o creare funzioni personalizzate.

Mappatura dei valori di diversi iterabili

Ad esempio, calcolando la media di ogni elemento `i` -esimo di più iterabili:

```
def average(*args):
    return float(sum(args)) / len(args)  # cast to float - only mandatory for python 2.x

measurement1 = [100, 111, 99, 97]
measurement2 = [102, 117, 91, 102]
measurement3 = [104, 102, 95, 101]

list(map(average, measurement1, measurement2, measurement3))
# Out: [102.0, 110.0, 95.0, 100.0]
```

Ci sono diversi requisiti se più di un iterabile viene passato alla `map` seconda della versione di python:

- La funzione deve prendere tutti i parametri quanti sono i iterabili:

  ```
  def median_of_three(a, b, c):
      return sorted((a, b, c))[1]

  list(map(median_of_three, measurement1, measurement2))
  ```

  TypeError: median_of_three () mancante 1 argomento posizionale richiesto: 'c'

```
list(map(median_of_three, measurement1, measurement2, measurement3, measurement3))
```

TypeError: median_of_three () accetta 3 argomenti posizionali ma ne sono stati assegnati 4

Python 2.x 2.0.1

- `map` : la mappatura itera finché un iterable non è ancora completamente consumato, ma assume `None` dai iterables completamente consumati:

```
import operator

measurement1 = [100, 111, 99, 97]
measurement2 = [102, 117]

# Calculate difference between elements
list(map(operator.sub, measurement1, measurement2))
```

TypeError: tipi di operando non supportati per -: 'int' e 'NoneType'

- `itertools.imap` e `future_builtins.map` : la mappatura si interrompe non appena si interrompe un iterable:

```
import operator
from itertools import imap

measurement1 = [100, 111, 99, 97]
measurement2 = [102, 117]

# Calculate difference between elements
list(imap(operator.sub, measurement1, measurement2))
# Out: [-2, -6]
list(imap(operator.sub, measurement2, measurement1))
# Out: [2, 6]
```

Python 3.x 3.0.0

- La mappatura si interrompe non appena si interrompe un iterable:

```
import operator

measurement1 = [100, 111, 99, 97]
measurement2 = [102, 117]

# Calculate difference between elements
list(map(operator.sub, measurement1, measurement2))
# Out: [-2, -6]
list(map(operator.sub, measurement2, measurement1))
# Out: [2, 6]
```

Trasposizione con mappa: utilizzo di "Nessuno" come argomento di funzione (solo python 2.x)

```
from itertools import imap
from future_builtins import map as fmap # Different name to highlight differences

image = [[1, 2, 3],
         [4, 5, 6],
         [7, 8, 9]]

list(map(None, *image))
# Out: [(1, 4, 7), (2, 5, 8), (3, 6, 9)]
list(fmap(None, *image))
# Out: [(1, 4, 7), (2, 5, 8), (3, 6, 9)]
list(imap(None, *image))
# Out: [(1, 4, 7), (2, 5, 8), (3, 6, 9)]

image2 = [[1, 2, 3],
          [4, 5],
          [7, 8, 9]]
list(map(None, *image2))
# Out: [(1, 4, 7), (2, 5, 8), (3, None, 9)]  # Fill missing values with None
list(fmap(None, *image2))
# Out: [(1, 4, 7), (2, 5, 8)]                # ignore columns with missing values
list(imap(None, *image2))
# Out: [(1, 4, 7), (2, 5, 8)]                # dito
```

Python 3.x 3.0.0

```
list(map(None, *image))
```

TypeError: l'oggetto 'NoneType' non è richiamabile

Ma c'è una soluzione alternativa per ottenere risultati simili:

```
def conv_to_list(*args):
    return list(args)

list(map(conv_to_list, *image))
# Out: [[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

Serie e mappatura parallela

map () è una funzione built-in, il che significa che è disponibile ovunque senza la necessità di utilizzare un'istruzione 'import'. È disponibile ovunque, proprio come print (). Se guardi l'esempio 5, vedrai che ho dovuto usare una dichiarazione di importazione prima di poter usare la stampa carina (import pprint). Quindi pprint non è una funzione incorporata

Mappatura delle serie

In questo caso, ogni argomento dell'iterazione viene fornito come argomento della funzione di mappatura in ordine ascendente. Ciò si verifica quando abbiamo un solo iterabile da mappare e la funzione di mappatura richiede un singolo argomento.

Esempio 1

```
insects = ['fly', 'ant', 'beetle', 'cankerworm']
```

```
f = lambda x: x + ' is an insect'
print(list(map(f, insects))) # the function defined by f is executed on each item of the
iterable insects
```

risultati in

```
['fly is an insect', 'ant is an insect', 'beetle is an insect', 'cankerworm is an insect']
```

Esempio 2

```
print(list(map(len, insects))) # the len function is executed each item in the insect list
```

risultati in

```
[3, 3, 6, 10]
```

Mappatura parallela

In questo caso ogni argomento della funzione di mappatura viene estratto da tutti i iterabili (uno da ciascun iterabile) in parallelo. Pertanto il numero di iterabili forniti deve corrispondere al numero di argomenti richiesti dalla funzione.

```
carnivores = ['lion', 'tiger', 'leopard', 'arctic fox']
herbivores = ['african buffalo', 'moose', 'okapi', 'parakeet']
omnivores = ['chicken', 'dove', 'mouse', 'pig']

def animals(w, x, y, z):
    return '{0}, {1}, {2}, and {3} ARE ALL ANIMALS'.format(w.title(), x, y, z)
```

Esempio 3

```
# Too many arguments
# observe here that map is trying to pass one item each from each of the four iterables to
len. This leads len to complain that
# it is being fed too many arguments
print(list(map(len, insects, carnivores, herbivores, omnivores)))
```

risultati in

```
TypeError: len() takes exactly one argument (4 given)
```

Esempio 4

```
# Too few arguments
# observe here that map is suppose to execute animal on individual elements of insects one-by-
one. But animals complain when
# it only gets one argument, whereas it was expecting four.
print(list(map(animals, insects)))
```

risultati in

```
TypeError: animals() missing 3 required positional arguments: 'x', 'y', and 'z'
```

Esempio 5

```
# here map supplies w, x, y, z with one value from across the list
import pprint
pprint.pprint(list(map(animals, insects, carnivores, herbivores, omnivores)))
```

risultati in

```
 ['Fly, lion, african buffalo, and chicken ARE ALL ANIMALS',
 'Ant, tiger, moose, and dove ARE ALL ANIMALS',
 'Beetle, leopard, okapi, and mouse ARE ALL ANIMALS',
 'Cankerworm, arctic fox, parakeet, and pig ARE ALL ANIMALS']
```

Leggi Funzione mappa online: https://riptutorial.com/it/python/topic/333/funzione-mappa

Capitolo 67: funzioni

introduzione

Le funzioni in Python forniscono codice organizzato, riutilizzabile e modulare per eseguire una serie di azioni specifiche. Le funzioni semplificano il processo di codifica, impediscono la logica ridondante e rendono il codice più facile da seguire. Questo argomento descrive la dichiarazione e l'utilizzo delle funzioni in Python.

Python ha molte *funzioni integrate* come `print()` , `input()` , `len()` . Oltre ai built-in è possibile anche creare le proprie funzioni per svolgere lavori più specifici, queste sono chiamate *funzioni definite dall'utente* .

Sintassi

- def *function_name* (*arg1, ... argN, * args, kw1, kw2 = default, ..., ** kwargs*): *istruzioni*
- lambda *arg1, ... argN, * args, kw1, kw2 = default, ..., ** kwargs* : *espressione*

Parametri

Parametro	Dettagli
arg1 , ..., *argN*	Argomenti regolari
* *args*	Argomenti posizionali senza nome
kw1 , ..., *kwN*	Argomenti basati su parole chiave
** *kwargs*	Il resto degli argomenti delle parole chiave

Osservazioni

5 cose basilari che puoi fare con le funzioni:

- Assegna funzioni alle variabili

```
def f():
  print(20)
y = f
y()
# Output: 20
```

- Definire funzioni all'interno di altre funzioni (funzioni annidate)

```
def f(a, b, y):
```

```
    def inner_add(a, b):      # inner_add is hidden from outer code
        return a + b
    return inner_add(a, b)**y
```

- Le funzioni possono restituire altre funzioni

```
def f(y):
    def nth_power(x):
        return x ** y
    return nth_power    # returns a function

squareOf = f(2)         # function that returns the square of a number
cubeOf = f(3)           # function that returns the cube of a number
squareOf(3)             # Output: 9
cubeOf(2)               # Output: 8
```

- Le funzioni possono essere passate come parametri ad altre funzioni

```
def a(x, y):
    print(x, y)
def b(fun, str):        # b has two arguments: a function and a string
    fun('Hello', str)
b(a, 'Sophia')           # Output: Hello Sophia
```

- Le funzioni interne hanno accesso all'ambito di chiusura (chiusura)

```
def outer_fun(name):
    def inner_fun():     # the variable name is available to the inner function
        return "Hello "+ name + "!"
    return inner_fun
greet = outer_fun("Sophia")
print(greet())            # Output: Hello Sophia!
```

Risorse addizionali

- Altro su funzioni e decoratori: https://www.thecodeship.com/patterns/guide-to-python-function-decorators/

Examples

Definire e chiamare semplici funzioni

Usare l'istruzione `def` è il modo più comune per definire una funzione in python. Questa affermazione è una cosiddetta *dichiarazione composta* con *clausola singola* con la seguente sintassi:

```
def function_name(parameters):
    statement(s)
```

`function_name` è noto come *identificativo* della funzione. Poiché una definizione di funzione è

un'istruzione eseguibile, la sua esecuzione *associa* il nome della funzione all'oggetto funzione che può essere chiamato in seguito utilizzando l'identificatore.

`parameters` è un elenco opzionale di identificatori che vengono associati ai valori forniti come argomenti quando viene chiamata la funzione. Una funzione può avere un numero arbitrario di argomenti separati da virgole.

`statement(s)` - noto anche come il *corpo* della *funzione* - sono una sequenza non vuota di istruzioni eseguite ogni volta che viene chiamata la funzione. Ciò significa che un corpo della funzione non può essere vuoto, proprio come qualsiasi *blocco rientrato* .

Ecco un esempio di una definizione di una funzione semplice che ha lo scopo di stampare `Hello` ogni volta che viene chiamato:

```
def greet():
    print("Hello")
```

Chiamiamo ora la funzione `greet()` definita:

```
greet()
# Out: Hello
```

Questo è un altro esempio di una definizione di funzione che prende un singolo argomento e visualizza il valore passato ogni volta che viene chiamata la funzione:

```
def greet_two(greeting):
    print(greeting)
```

Dopo che la funzione `greet_two()` deve essere chiamata con un argomento:

```
greet_two("Howdy")
# Out: Howdy
```

Inoltre puoi dare un valore predefinito a quell'argomento di funzione:

```
def greet_two(greeting="Howdy"):
    print(greeting)
```

Ora puoi chiamare la funzione senza dare un valore:

```
greet_two()
# Out: Howdy
```

Noterai che a differenza di molti altri linguaggi, non è necessario dichiarare esplicitamente un tipo di ritorno della funzione. Le funzioni Python possono restituire valori di qualsiasi tipo tramite la parola chiave `return` . Una funzione può restituire qualsiasi numero di tipi diversi!

```
def many_types(x):
    if x < 0:
```

```
        return "Hello!"
    else:
        return 0

print(many_types(1))
print(many_types(-1))

# Output:
0
Hello!
```

Finché questo è gestito correttamente dal chiamante, questo è un codice Python perfettamente valido.

Una funzione che raggiunge la fine dell'esecuzione senza un'istruzione return restituirà sempre `None` :

```
def do_nothing():
    pass

print(do_nothing())
# Out: None
```

Come accennato in precedenza, una definizione di funzione deve avere un corpo di funzione, una sequenza di dichiarazioni non vuote. Pertanto l'istruzione `pass` è usata come corpo della funzione, che è un'operazione nulla, quando viene eseguita, non accade nulla. Fa quello che significa, salta. È utile come segnaposto quando un'istruzione è richiesta sintatticamente, ma non è necessario eseguire alcun codice.

Valori di ritorno da funzioni

Le funzioni possono `return` un valore che è possibile utilizzare direttamente:

```
def give_me_five():
    return 5

print(give_me_five())  # Print the returned value
# Out: 5
```

o salvare il valore per un uso successivo:

```
num = give_me_five()
print(num)             # Print the saved returned value
# Out: 5
```

oppure utilizzare il valore per qualsiasi operazione:

```
print(give_me_five() + 10)
# Out: 15
```

Se si verifica un `return` nella funzione, la funzione verrà immediatamente chiusa e le successive

operazioni non saranno valutate:

```
def give_me_another_five():
    return 5
    print('This statement will not be printed. Ever.')

print(give_me_another_five())
# Out: 5
```

Puoi anche `return` più valori (sotto forma di tupla):

```
def give_me_two_fives():
    return 5, 5  # Returns two 5

first, second = give_me_two_fives()
print(first)
# Out: 5
print(second)
# Out: 5
```

Una funzione *senza* dichiarazione di `return` restituisce implicitamente `None` . Allo stesso modo una funzione con un'istruzione `return` , ma nessun valore restituito o variabile restituisce `None` .

Definire una funzione con argomenti

Gli argomenti sono definiti tra parentesi dopo il nome della funzione:

```
def divide(dividend, divisor):  # The names of the function and its arguments
    # The arguments are available by name in the body of the function
    print(dividend / divisor)
```

Il nome della funzione e il suo elenco di argomenti sono chiamati la *firma* della funzione. Ogni argomento con nome è effettivamente una variabile locale della funzione.

Quando si chiama la funzione, fornire i valori per gli argomenti elencandoli nell'ordine

```
divide(10, 2)
# output: 5
```

o specificarli in qualsiasi ordine utilizzando i nomi dalla definizione della funzione:

```
divide(divisor=2, dividend=10)
# output: 5
```

Definizione di una funzione con argomenti opzionali

Gli argomenti facoltativi possono essere definiti assegnando (usando `=`) un valore predefinito all'argomento-nome:

```
def make(action='nothing'):
    return action
```

La chiamata a questa funzione è possibile in 3 modi diversi:

```
make("fun")
# Out: fun

make(action="sleep")
# Out: sleep

# The argument is optional so the function will use the default value if the argument is
# not passed in.
make()
# Out: nothing
```

avvertimento

I tipi mutabili (`list` , `dict` , `set` , ecc.) Devono essere trattati con attenzione quando vengono forniti come attributo **predefinito** . Qualsiasi mutazione dell'argomento predefinito lo cambierà in modo permanente. Vedi Definire una funzione con argomenti mutabili opzionali .

Definizione di una funzione con più argomenti

Si può dare una funzione al numero di argomenti che si desidera, le uniche regole fisse sono che ogni nome di argomento deve essere univoco e che gli argomenti facoltativi devono essere successivi a quelli non facoltativi:

```
def func(value1, value2, optionalvalue=10):
    return '{0} {1} {2}'.format(value1, value2, optionalvalue1)
```

Quando si chiama la funzione, è possibile assegnare a ciascuna parola chiave senza il nome, ma l'ordine conta:

```
print(func(1, 'a', 100))
# Out: 1 a 100

print(func('abc', 14))
# abc 14 10
```

O combinare dando gli argomenti con il nome e senza. Quindi quelli con nome devono seguire quelli senza ma l'ordine di quelli con nome non ha importanza:

```
print(func('This', optionalvalue='StackOverflow Documentation', value2='is'))
# Out: This is StackOverflow Documentation
```

Definire una funzione con un numero arbitrario di argomenti

Numero arbitrario di argomenti posizionali:

La definizione di una funzione in grado di prendere un numero arbitrario di argomenti può essere effettuata con il prefisso di uno degli argomenti con un `*`

```
def func(*args):
    # args will be a tuple containing all values that are passed in
    for i in args:
        print(i)

func(1, 2, 3)  # Calling it with 3 arguments
# Out: 1
#      2
#      3

list_of_arg_values = [1, 2, 3]
func(*list_of_arg_values)  # Calling it with list of values, * expands the list
# Out: 1
#      2
#      3

func()  # Calling it without arguments
# No Output
```

Non è **possibile** fornire un valore predefinito per `args` , ad esempio `func(*args=[1, 2, 3])` genererà un errore di sintassi (non verrà nemmeno compilato).

Non è **possibile** fornire questi per nome quando si chiama la funzione, ad esempio `func(*args=[1, 2, 3])` genererà un `TypeError` .

Ma se hai già i tuoi argomenti in un array (o qualsiasi altro `Iterable`), **puoi** invocare la tua funzione in questo modo: `func(*my_stuff)` .

Questi argomenti (`*args`) sono accessibili dall'indice, ad esempio `args[0]` restituirà il primo argomento

Numero arbitrario di argomenti di parole chiave

Puoi prendere un numero arbitrario di argomenti con un nome definendo un argomento nella definizione con **due** `*` di fronte ad esso:

```
def func(**kwargs):
    # kwargs will be a dictionary containing the names as keys and the values as values
    for name, value in kwargs.items():
        print(name, value)

func(value1=1, value2=2, value3=3)   # Calling it with 3 arguments
# Out: value1 1
#      value2 2
#      value3 3

func()                               # Calling it without arguments
# No Out put
```

```
my_dict = {'foo': 1, 'bar': 2}
func(**my_dict)                      # Calling it with a dictionary
# Out: foo 1
#      bar 2
```

Non è **possibile** fornire questi **senza** nomi, ad esempio `func(1, 2, 3)` genererà un `TypeError` .

`kwargs` è un semplice dizionario Python nativo. Ad esempio, `args['value1']` fornirà il valore per argomento `value1` . Assicurati di controllare in anticipo che c'è un tale argomento o che verrà sollevato un `KeyError` .

avvertimento

È possibile combinarli con altri argomenti opzionali e obbligatori ma l'ordine all'interno della definizione è importante.

Gli argomenti **posizionale / parola chiave** vengono prima. (Argomenti richiesti).
Poi vengono gli argomenti **arbitrari** `*arg` . (Opzionale).
Quindi gli argomenti con **parole chiave** vengono dopo. (Necessario).
Infine viene la **parola chiave arbitraria** `**kwargs` . (Opzionale).

```
#       |-positional-|-optional-|---keyword-only--|-optional-|
def func(arg1, arg2=10 , *args, kwarg1, kwarg2=2, **kwargs):
     pass
```

- deve essere fornito `arg1` , altrimenti viene sollevato un `TypeError` . Può essere dato come argomento posizionale (`func(10)`) o parola chiave (`func(arg1=10)`).
- `kwarg1` deve essere fornito, ma può essere fornito solo come argomento-chiave: `func(kwarg1=10)` .
- `arg2` e `kwarg2` sono opzionali. Se il valore deve essere modificato, si applicano le stesse regole di `arg1` (posizionale o parola chiave) e `kwarg1` (solo parola chiave).
- `*args` acquisisce parametri posizionali aggiuntivi. Ma nota, che `arg1` e `arg2` devono essere forniti come argomenti posizionali per passare argomenti a `*args` : `func(1, 1, 1, 1)` .
- `**kwargs` tutti i parametri aggiuntivi delle parole chiave. In questo caso qualsiasi parametro che non sia `arg1` , `arg2` , `kwarg1` o `kwarg2` . Ad esempio: `func(kwarg3=10)` .
- In Python 3, puoi usare `*` da solo per indicare che tutti gli argomenti successivi devono essere specificati come parole chiave. Ad esempio la funzione `math.isclose` in Python 3.5 e successive viene definita usando `def math.isclose (a, b, *, rel_tol=1e-09, abs_tol=0.0)` , il che significa che i primi due argomenti possono essere forniti in modo posizionale ma facoltativo il terzo e il quarto parametro possono essere forniti solo come argomenti di parole chiave.

Python 2.x non supporta i parametri solo per le parole chiave. Questo comportamento può essere emulato con `kwargs` :

```
def func(arg1, arg2=10, **kwargs):
    try:
```

```
        kwarg1 = kwargs.pop("kwarg1")
    except KeyError:
        raise TypeError("missing required keyword-only argument: 'kwarg1'")

    kwarg2 = kwargs.pop("kwarg2", 2)
    # function body ...
```

Nota sulla denominazione

La convenzione di denominazione opzionali argomenti posizionali `args` e opzionali argomenti chiave `kwargs` è solo una convenzione è **possibile** utilizzare qualsiasi nome che ti **piace, ma** è utile per seguire la convenzione in modo che gli altri sanno cosa si sta facendo, *o anche te più tardi* quindi per favore fate.

Nota sull'unicità

Qualsiasi funzione può essere definita con **none o un** `*args` e **nessuno o uno** `**kwargs` ma non con più di uno di ciascuno. Anche `*args` **deve** essere l'ultimo argomento posizionale e `**kwargs` deve essere l'ultimo parametro. Il tentativo di utilizzare più di uno dei due **genererà** un'eccezione di Errore di sintassi.

Nota sulle funzioni di annidamento con argomenti opzionali

È possibile annidare tali funzioni e la consueta convenzione è rimuovere gli elementi che il codice ha già gestito, **ma** se si stanno passando i parametri è necessario passare arg facoltativi di posizione con un prefisso `*` e arg parole facoltative con un prefisso `**` , altrimenti gli argomenti vengono passati come elenco o tupla e kwargs come un singolo dizionario. per esempio:

```
def fn(**kwargs):
    print(kwargs)
    f1(**kwargs)

def f1(**kwargs):
    print(len(kwargs))

fn(a=1, b=2)
# Out:
# {'a': 1, 'b': 2}
# 2
```

Definizione di una funzione con argomenti mutabili opzionali

C'è un problema quando si usano **argomenti opzionali** con un **tipo predefinito mutabile** (descritto in Definire una funzione con argomenti facoltativi), che può potenzialmente portare a comportamenti imprevisti.

Spiegazione

Questo problema si verifica perché gli argomenti predefiniti di una funzione vengono inizializzati **una volta** , nel momento in cui viene *definita* la funzione, e **non** (come molti altri linguaggi) quando viene *chiamata* la funzione. I valori predefiniti sono memorizzati nella variabile membro `__defaults__` dell'oggetto della `__defaults__` .

```
def f(a, b=42, c=[]):
    pass

print(f.__defaults__)
# Out: (42, [])
```

Per i tipi **immutabili** (vedere passaggio degli argomenti e mutabilità) questo non è un problema perché non c'è modo di mutare la variabile; può essere riassegnato solo sempre, lasciando invariato il valore originale. Quindi, successivamente è garantito avere lo stesso valore di default. Tuttavia, per un tipo **mutabile** , il valore originale può essere mutato, effettuando chiamate alle sue varie funzioni membro. Di conseguenza, non è garantito che le chiamate successive alla funzione abbiano il valore predefinito iniziale.

```
def append(elem, to=[]):
    to.append(elem)      # This call to append() mutates the default variable "to"
    return to

append(1)
# Out: [1]

append(2)  # Appends it to the internally stored list
# Out: [1, 2]

append(3, [])  # Using a new created list gives the expected result
# Out: [3]

# Calling it again without argument will append to the internally stored list again
append(4)
# Out: [1, 2, 4]
```

> **Nota:** alcuni IDE come PyCharm emetteranno un avviso quando un tipo mutabile viene specificato come attributo predefinito.

Soluzione

Se si desidera assicurarsi che l'argomento predefinito sia sempre quello specificato nella definizione della funzione, la soluzione deve **sempre** utilizzare un tipo immutabile come argomento predefinito.

Un idioma comune per ottenere ciò quando è necessario un tipo mutabile come predefinito è utilizzare `None` (immutable) come argomento predefinito e quindi assegnare il valore predefinito effettivo alla variabile argomento se è uguale a `None` .

```
def append(elem, to=None):
    if to is None:
        to = []
```

```
    to.append(elem)
    return to
```

Funzioni Lambda (Inline / Anonimo)

La parola chiave `lambda` crea una funzione inline che contiene una singola espressione. Il valore di questa espressione è ciò che la funzione restituisce quando viene richiamata.

Considera la funzione:

```
def greeting():
    return "Hello"
```

che, quando chiamato come:

```
print(greeting())
```

stampe:

```
Hello
```

Questo può essere scritto come una funzione lambda come segue:

```
greet_me = lambda: "Hello"
```

> Vedere la nota in fondo a questa sezione riguardante l'assegnazione di lambda alle variabili. Generalmente, non farlo.

Questo crea una funzione inline con il nome `greet_me` che restituisce `Hello` . Si noti che non si scrive `return` quando si crea una funzione con lambda. Il valore dopo `:` viene automaticamente restituito.

Una volta assegnato a una variabile, può essere utilizzato proprio come una funzione regolare:

```
print(greet_me())
```

stampe:

```
Hello
```

`lambda` s può prendere argomenti anche:

```
strip_and_upper_case = lambda s: s.strip().upper()

strip_and_upper_case("  Hello   ")
```

restituisce la stringa:

```
HELLO
```

Possono anche prendere un numero arbitrario di argomenti / argomenti di parole chiave, come le normali funzioni.

```
greeting = lambda x, *args, **kwargs: print(x, args, kwargs)
greeting('hello', 'world', world='world')
```

stampe:

```
hello ('world',) {'world': 'world'}
```

`lambda` vengono comunemente utilizzate per le funzioni brevi che sono convenienti da definire nel punto in cui vengono chiamate (in genere con `sorted` , `filter` e `map`).

Ad esempio, questa riga ordina un elenco di stringhe che ignorano il loro caso e ignorano gli spazi bianchi all'inizio e alla fine:

```
sorted( [" foo ", "    bAR", "BaZ    "], key=lambda s: s.strip().upper())
# Out:
# ['    bAR', 'BaZ    ', ' foo ']
```

Ordina la lista semplicemente ignorando gli spazi bianchi:

```
sorted( [" foo ", "    bAR", "BaZ    "], key=lambda s: s.strip())
# Out:
# ['BaZ    ', '    bAR', ' foo ']
```

Esempi con la `map` :

```
sorted( map( lambda s: s.strip().upper(), [" foo ", "    bAR", "BaZ    "]))
# Out:
# ['BAR', 'BAZ', 'FOO']

sorted( map( lambda s: s.strip(), [" foo ", "    bAR", "BaZ    "]))
# Out:
# ['BaZ', 'bAR', 'foo']
```

Esempi con liste numeriche:

```
my_list = [3, -4, -2, 5, 1, 7]
sorted( my_list, key=lambda x: abs(x))
# Out:
# [1, -2, 3, -4, 5, 7]

list( filter( lambda x: x>0, my_list))
# Out:
# [3, 5, 1, 7]

list( map( lambda x: abs(x), my_list))
# Out:
[3, 4, 2, 5, 1, 7]
```

Si possono chiamare altre funzioni (con / senza argomenti) dall'interno di una funzione lambda.

```
def foo(msg):
    print(msg)

greet = lambda x = "hello world": foo(x)
greet()
```

stampe:

```
hello world
```

Ciò è utile perché `lambda` può contenere solo un'espressione e utilizzando una funzione sussidiaria è possibile eseguire più istruzioni.

NOTA

Ricordare che PEP-8 (la guida di stile Python ufficiale) non consiglia l'assegnazione di lambdas alle variabili (come abbiamo fatto nei primi due esempi):

> Usa sempre un'istruzione def invece di un'istruzione di assegnazione che associa un'espressione lambda direttamente a un identificatore.
>
> Sì:
>
> ```
> def f(x): return 2*x
> ```
>
> No:
>
> ```
> f = lambda x: 2*x
> ```
>
> Il primo modulo indica che il nome dell'oggetto funzionale risultante è specificamente `f` invece del generico `<lambda>` . Questo è più utile per traceback e rappresentazioni di stringhe in generale. L'uso della dichiarazione di assegnazione elimina l'unico vantaggio che un'espressione lambda può offrire rispetto a una dichiarazione di `def` esplicita (ossia che può essere incorporata in un'espressione più grande).

Passaggio di argomenti e mutabilità

Innanzitutto, alcuni termini:

- **argomento (parametro *attuale*):** la variabile attuale passata a una funzione;
- **parametro (parametro *formale*):** la variabile ricevente che viene utilizzata in una funzione.

In Python, gli argomenti vengono passati per *assegnazione* (al contrario di altri linguaggi, dove gli argomenti possono essere passati per valore / riferimento / puntatore).

- La modifica di un parametro muterà l'argomento (se il tipo dell'argomento è modificabile).

```
def foo(x):        # here x is the parameter
    x[0] = 9       # This mutates the list labelled by both x and y
    print(x)

y = [4, 5, 6]
foo(y)             # call foo with y as argument
# Out: [9, 5, 6]   # list labelled by x has been mutated
print(y)
# Out: [9, 5, 6]   # list labelled by y has been mutated too
```

- La riassegnazione del parametro non riassegnerà l'argomento.

```
def foo(x):        # here x is the parameter, when we call foo(y) we assign y to x
    x[0] = 9       # This mutates the list labelled by both x and y
    x = [1, 2, 3]  # x is now labeling a different list (y is unaffected)
    x[2] = 8       # This mutates x's list, not y's list

y = [4, 5, 6]      # y is the argument, x is the parameter
foo(y)             # Pretend that we wrote "x = y", then go to line 1
y
# Out: [9, 5, 6]
```

In Python, non assegniamo realmente valori alle variabili, ma *leghiamo* (cioè assegniamo, attribuiamo) variabili (considerate come *nomi*) agli oggetti.

- **Immutabile:** numeri interi, archi, tuple e così via. Tutte le operazioni fanno copie.
- **Mutabile:** elenchi, dizionari, set e così via. Le operazioni possono o non possono mutare.

```
x = [3, 1, 9]
y = x
x.append(5)    # Mutates the list labelled by x and y, both x and y are bound to [3, 1, 9]
x.sort()       # Mutates the list labelled by x and y (in-place sorting)
x = x + [4]    # Does not mutate the list (makes a copy for x only, not y)
z = x          # z is x ([1, 3, 9, 4])
x += [6]       # Mutates the list labelled by both x and z (uses the extend function).
x = sorted(x)  # Does not mutate the list (makes a copy for x only).
x
# Out: [1, 3, 4, 5, 6, 9]
y
# Out: [1, 3, 5, 9]
z
# Out: [1, 3, 5, 9, 4, 6]
```

Chiusura

Le chiusure in Python sono create da chiamate di funzione. Qui, la chiamata a `makeInc` crea un binding per `x` cui si fa riferimento all'interno della funzione `inc` . Ogni chiamata a `makeInc` crea una nuova istanza di questa funzione, ma ogni istanza ha un collegamento a un diverso bind di `x` .

```
def makeInc(x):
  def inc(y):
     # x is "attached" in the definition of inc
     return y + x
```

```
  return inc

incOne = makeInc(1)
incFive = makeInc(5)

incOne(5) # returns 6
incFive(5) # returns 10
```

Si noti che mentre in una chiusura regolare la funzione racchiusa eredita completamente tutte le variabili dal suo ambiente di chiusura, in questo costrutto la funzione chiusa ha solo accesso in lettura alle variabili ereditate ma non può assegnarle

```
def makeInc(x):
  def inc(y):
     # incrementing x is not allowed
     x += y
     return x

  return inc

incOne = makeInc(1)
incOne(5) # UnboundLocalError: local variable 'x' referenced before assignment
```

Python 3 offre l'istruzione `nonlocal` (Variabili `nonlocal`) per realizzare una chiusura completa con funzioni annidate.

Python 3.x 3.0

```
def makeInc(x):
  def inc(y):
     nonlocal x
     # now assigning a value to x is allowed
     x += y
     return x

  return inc

incOne = makeInc(1)
incOne(5) # returns 6
```

Funzioni ricorsive

Una funzione ricorsiva è una funzione che si chiama nella sua definizione. Ad esempio la funzione matematica, fattoriale, definita da `factorial(n) = n*(n-1)*(n-2)*...*3*2*1` . può essere programmato come

```
def factorial(n):
    #n here should be an integer
    if n == 0:
        return 1
    else:
        return n*factorial(n-1)
```

le uscite qui sono:

```
factorial(0)
#out 1
factorial(1)
#out 1
factorial(2)
#out 2
factorial(3)
#out 6
```

come previsto. Si noti che questa funzione è ricorsiva perché il secondo `return factorial(n-1)` , in cui la funzione si chiama nella sua definizione.

Alcune funzioni ricorsive possono essere implementate usando lambda , la funzione fattoriale che usa lambda sarebbe qualcosa del genere:

```
factorial = lambda n: 1 if n == 0 else n*factorial(n-1)
```

La funzione restituisce lo stesso come sopra.

Limite di ricorsione

C'è un limite alla profondità della possibile ricorsione, che dipende dall'implementazione di Python. Quando viene raggiunto il limite, viene sollevata un'eccezione RuntimeError:

```
def cursing(depth):
  try:
    cursing(depth + 1) # actually, re-cursing
  except RuntimeError as RE:
    print('I recursed {} times!'.format(depth))

cursing(0)
# Out: I recursed 1083 times!
```

È possibile modificare il limite di profondità della ricorsione utilizzando `sys.setrecursionlimit(limit)` e controllare questo limite con `sys.getrecursionlimit()` .

```
sys.setrecursionlimit(2000)
cursing(0)
# Out: I recursed 1997 times!
```

Da Python 3.5, l'eccezione è `RecursionError` , che deriva da `RuntimeError` .

Funzioni annidate

Le funzioni in python sono oggetti di prima classe. Possono essere definiti in qualsiasi ambito

```
def fibonacci(n):
    def step(a,b):
        return b, a+b
    a, b = 0, 1
    for i in range(n):
        a, b = step(a, b)
```

```
    return a
```

Le funzioni acquisiscono il loro ambito di inclusione possono essere passate come qualsiasi altro tipo di oggetto

```
def make_adder(n):
    def adder(x):
        return n + x
    return adder
add5 = make_adder(5)
add6 = make_adder(6)
add5(10)
#Out: 15
add6(10)
#Out: 16

def repeatedly_apply(func, n, x):
    for i in range(n):
        x = func(x)
    return x

repeatedly_apply(add5, 5, 1)
#Out: 26
```

Iterable e dizionario disimballaggio

Le funzioni consentono di specificare questi tipi di parametri: posizionali, denominati, variabili posizionali, argomenti di parole chiave (kwargs). Ecco un uso chiaro e conciso di ogni tipo.

```
def unpacking(a, b, c=45, d=60, *args, **kwargs):
    print(a, b, c, d, args, kwargs)

>>> unpacking(1, 2)
1 2 45 60 () {}
>>> unpacking(1, 2, 3, 4)
1 2 3 4 () {}
>>> unpacking(1, 2, c=3, d=4)
1 2 3 4 () {}
>>> unpacking(1, 2, d=4, c=3)
1 2 3 4 () {}


>>> pair = (3,)
>>> unpacking(1, 2, *pair, d=4)
1 2 3 4 () {}
>>> unpacking(1, 2, d=4, *pair)
1 2 3 4 () {}
>>> unpacking(1, 2, *pair, c=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'c'
>>> unpacking(1, 2, c=3, *pair)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'c'

>>> args_list = [3]
```

```
>>> unpacking(1, 2, *args_list, d=4)
1 2 3 4 () {}
>>> unpacking(1, 2, d=4, *args_list)
1 2 3 4 () {}
>>> unpacking(1, 2, c=3, *args_list)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'c'
>>> unpacking(1, 2, *args_list, c=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'c'


>>> pair = (3, 4)
>>> unpacking(1, 2, *pair)
1 2 3 4 () {}
>>> unpacking(1, 2, 3, 4, *pair)
1 2 3 4 (3, 4) {}
>>> unpacking(1, 2, d=4, *pair)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'd'
>>> unpacking(1, 2, *pair, d=4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'd'



>>> args_list = [3, 4]
>>> unpacking(1, 2, *args_list)
1 2 3 4 () {}
>>> unpacking(1, 2, 3, 4, *args_list)
1 2 3 4 (3, 4) {}
>>> unpacking(1, 2, d=4, *args_list)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'd'
>>> unpacking(1, 2, *args_list, d=4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'd'


>>> arg_dict = {'c':3, 'd':4}
>>> unpacking(1, 2, **arg_dict)
1 2 3 4 () {}
>>> arg_dict = {'d':4, 'c':3}
>>> unpacking(1, 2, **arg_dict)
1 2 3 4 () {}
>>> arg_dict = {'c':3, 'd':4, 'not_a_parameter': 75}
>>> unpacking(1, 2, **arg_dict)
1 2 3 4 () {'not_a_parameter': 75}


>>> unpacking(1, 2, *pair, **arg_dict)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'd'
>>> unpacking(1, 2, 3, 4, **arg_dict)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'd'

# Positional arguments take priority over any other form of argument passing
>>> unpacking(1, 2, **arg_dict, c=3)
1 2 3 4 () {'not_a_parameter': 75}
>>> unpacking(1, 2, 3, **arg_dict, c=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'c'
```

Forzare l'uso dei parametri denominati

Tutti i parametri specificati dopo il primo asterisco nella firma della funzione sono solo parole chiave.

```
def f(*a, b):
    pass

f(1, 2, 3)
# TypeError: f() missing 1 required keyword-only argument: 'b'
```

In Python 3 è possibile inserire un singolo asterisco nella firma della funzione per garantire che gli argomenti rimanenti possano essere passati solo utilizzando gli argomenti delle parole chiave.

```
def f(a, b, *, c):
    pass

f(1, 2, 3)
# TypeError: f() takes 2 positional arguments but 3 were given
f(1, 2, c=3)
# No error
```

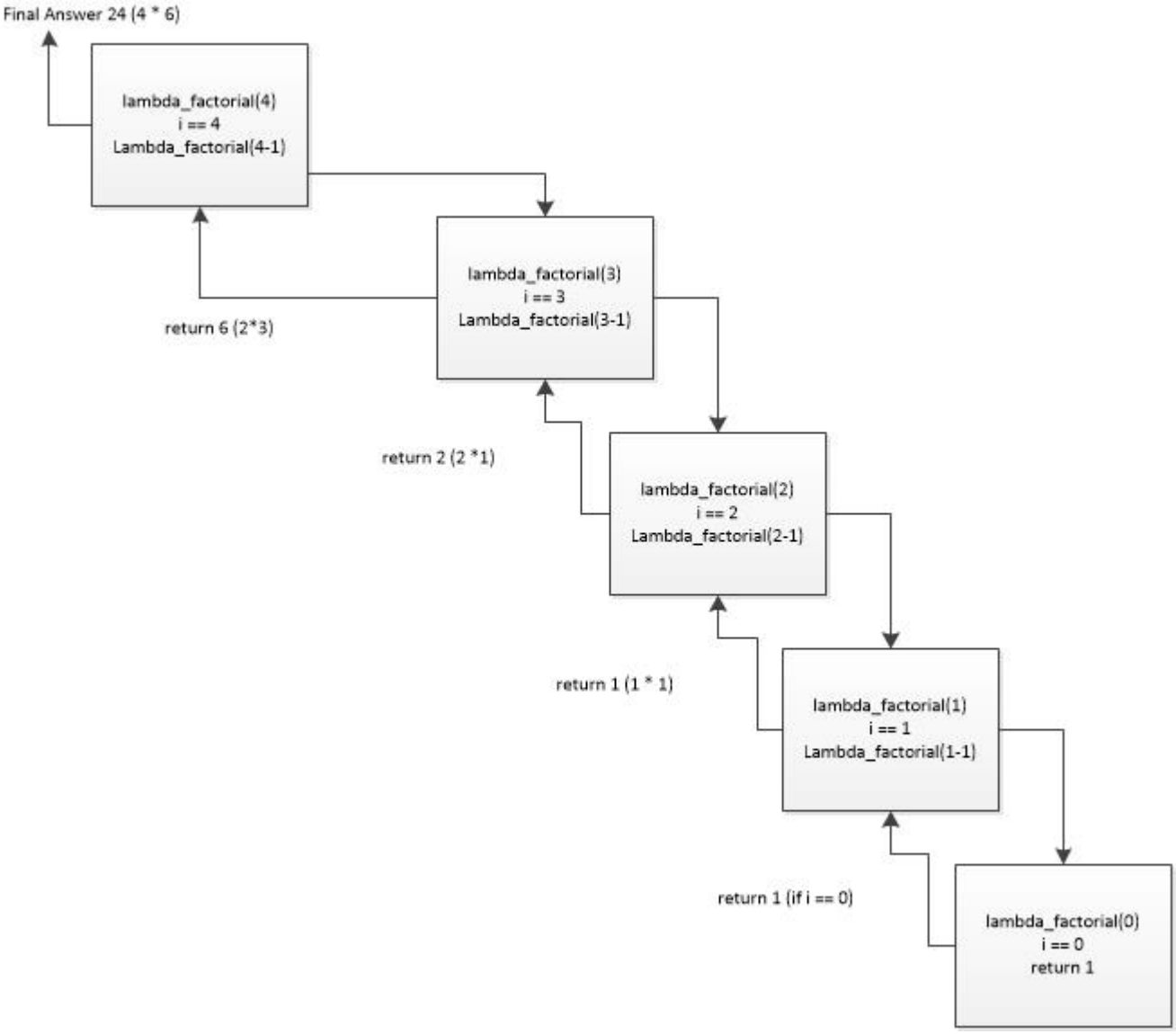
Lambda ricorsivo con variabile assegnata

Un metodo per la creazione di funzioni lambda ricorsive implica l'assegnazione della funzione a una variabile e quindi il riferimento a tale variabile all'interno della funzione stessa. Un esempio comune di questo è il calcolo ricorsivo del fattoriale di un numero, come mostrato nel seguente codice:

```
lambda_factorial = lambda i:1 if i==0 else i*lambda_factorial(i-1)
print(lambda_factorial(4)) # 4 * 3 * 2 * 1 = 12 * 2 = 24
```

Descrizione del codice

La funzione lambda, tramite la sua assegnazione di variabili, riceve un valore (4) che valuta e restituisce 1 se è 0 oppure restituisce il valore corrente (`i`) * un altro calcolo dalla funzione lambda del valore - 1 (`i-1`). Questo continua fino a quando il valore passato viene decrementato a 0 (`return 1`). Un processo che può essere visualizzato come:



Leggi funzioni online: https://riptutorial.com/it/python/topic/228/funzioni

Capitolo 68: Funzioni parziali

introduzione

Come probabilmente saprai, se venivi dalla scuola OOP, specializzarti in una classe astratta e usarla è una pratica da tenere a mente quando scrivi il tuo codice.

E se potessi definire una funzione astratta e specializzarla per crearne diverse versioni? Lo considera come una sorta di *funzione di ereditarietà in* cui si associano parametri specifici per renderli affidabili per uno scenario specifico.

Sintassi

- parziale (funzione, ** params_you_want_fix)

Parametri

Param	dettagli
X	il numero da sollevare
y	l'esponente
aumentare	la funzione di essere specializzata

Osservazioni

Come dichiarato in Python doc il *functools.partial* :

> Restituisce un nuovo oggetto parziale che, quando chiamato, si comporterà come func chiamato con gli argomenti posizionali args e keyword keyword arguments. Se vengono forniti più argomenti alla chiamata, vengono aggiunti agli argomenti. Se vengono forniti ulteriori argomenti per le parole chiave, essi estendono e sostituiscono le parole chiave.

Controlla questo link per vedere come può essere implementato *parziale* .

Examples

Alza il potere

Supponiamo di voler aumentare *x* a un numero *y* .

Scriveresti questo come:

```
def raise_power(x, y):
    return x**y
```

Cosa succede se il tuo valore *y* può assumere un insieme finito di valori?

Supponiamo *y* può essere uno dei *[3,4,5]* e diciamo che non si desidera che l'utente finale offerta la possibilità di utilizzare tale funzione in quanto è molto computazionalmente intensive. Infatti verificherai se fornito *y* assume un valore valido e riscrivi la tua funzione come:

```
def raise(x, y):
    if y in (3,4,5):
        return x**y
    raise NumberNotInRangeException("You should provide a valid exponent")
```

Disordinato? Usiamo la forma astratta e la specializziamo in tutti e tre i casi: implementiamoli **parzialmente** .

```
from functors import partial
raise_to_three = partial(raise, y=3)
raise_to_four = partial(raise, y=4)
raise_to_five = partial(raise, y=5)
```

Che succede qui? Abbiamo corretto i parametri *y* e abbiamo definito tre diverse funzioni.

Non è necessario utilizzare la funzione astratta definita in precedenza (è possibile renderla *privata*) ma è possibile utilizzare **le** funzioni **applicate parziali** per gestire l'aumento di un numero a un valore fisso.

Leggi Funzioni parziali online: https://riptutorial.com/it/python/topic/9383/funzioni-parziali

Capitolo 69: generatori

introduzione

I generatori sono gli iteratori pigri creati dalle funzioni del generatore (usando `yield`) o dalle espressioni del generatore (usando `(an_expression for x in an_iterator)`).

Sintassi

- resa `<expr>`
- rendimento da `<expr>`
- `<var>` = rendimento `<expr>`
- next (`<iter>`)

Examples

Iterazione

Un oggetto generatore supporta il *protocollo iteratore* . Vale a dire, fornisce un metodo `next()` (`__next__()` in Python 3.x), che viene usato per passare attraverso la sua esecuzione, e il suo metodo `__iter__` restituisce sé stesso. Ciò significa che un generatore può essere utilizzato in qualsiasi costrutto linguistico che supporti oggetti iterabili generici.

```
# naive partial implementation of the Python 2.x xrange()
def xrange(n):
    i = 0
    while i < n:
        yield i
        i += 1

# looping
for i in xrange(10):
    print(i)  # prints the values 0, 1, ..., 9

# unpacking
a, b, c = xrange(3)  # 0, 1, 2

# building a list
l = list(xrange(10))  # [0, 1, ..., 9]
```

La funzione next ()

Il `next()` built-in è un comodo wrapper che può essere usato per ricevere un valore da qualsiasi iteratore (incluso un iteratore di generatore) e per fornire un valore predefinito nel caso in cui l'iteratore sia esaurito.

```
def nums():
```

```
    yield 1
    yield 2
    yield 3
generator = nums()

next(generator, None)  # 1
next(generator, None)  # 2
next(generator, None)  # 3
next(generator, None)  # None
next(generator, None)  # None
# ...
```

La sintassi è `next(iterator[, default])` . Se l'iteratore termina e viene passato un valore predefinito, viene restituito. Se non è stato fornito alcun valore predefinito, `StopIteration` viene generato.

Invio di oggetti a un generatore

Oltre a ricevere i valori da un generatore, è possibile *inviare* un oggetto a un generatore usando il metodo `send()` .

```
def accumulator():
    total = 0
    value = None
    while True:
        # receive sent value
        value = yield total
        if value is None: break
        # aggregate values
        total += value

generator = accumulator()

# advance until the first "yield"
next(generator)      # 0

# from this point on, the generator aggregates values
generator.send(1)    # 1
generator.send(10)   # 11
generator.send(100)  # 111
# ...

# Calling next(generator) is equivalent to calling generator.send(None)
next(generator)      # StopIteration
```

Quello che succede qui è il seguente:

- Alla prima chiamata `next(generator)` , il programma avanza alla prima dichiarazione di `yield` e restituisce il valore `total` a quel punto, che è 0. L'esecuzione del generatore viene sospesa a questo punto.
- Quando si chiama `generator.send(x)` , l'interprete prende l'argomento `x` e lo rende il valore di ritorno dell'ultima dichiarazione di `yield` , che viene assegnata al `value` . Il generatore procede come al solito, finché non produce il valore successivo.
- Quando finalmente si chiama `next(generator)` , il programma lo tratta come se invii `None` al

generatore. Non c'è nulla di speciale in `None` , tuttavia, questo esempio usa `None` come valore speciale per chiedere al generatore di fermarsi.

Espressioni del generatore

È possibile creare iteratori di generatore utilizzando una sintassi simile alla comprensione.

```
generator = (i * 2 for i in range(3))

next(generator)  # 0
next(generator)  # 2
next(generator)  # 4
next(generator)  # raises StopIteration
```

Se non è necessario che una funzione passi una lista, è possibile salvare sui caratteri (e migliorare la leggibilità) inserendo un'espressione di generatore all'interno di una chiamata di funzione. La parentesi della chiamata di funzione rende implicitamente la tua espressione un'espressione di generatore.

```
sum(i ** 2 for i in range(4))  # 0^2 + 1^2 + 2^2 + 3^2 = 0 + 1 + 4 + 9 = 14
```

Inoltre, si risparmia sulla memoria perché invece di caricare l'intero elenco su cui si sta iterando (`[0, 1, 2, 3]` nell'esempio precedente), il generatore consente a Python di utilizzare i valori secondo necessità.

introduzione

Le espressioni del generatore sono simili a list, dictionary e set comprehensions, ma sono racchiuse tra parentesi. Le parentesi non devono essere presenti quando vengono utilizzate come unico argomento per una chiamata di funzione.

```
expression = (x**2 for x in range(10))
```

Questo esempio genera i primi 10 quadrati perfetti, incluso 0 (in cui x = 0).

Le funzioni del generatore sono simili alle funzioni regolari, tranne che hanno una o più dichiarazioni di `yield` nel loro corpo. Tali funzioni non possono `return` alcun valore (tuttavia è possibile `return` vuoti se si desidera arrestare anticipatamente il generatore).

```
def function():
    for x in range(10):
        yield x**2
```

Questa funzione del generatore è equivalente alla precedente espressione del generatore, emette lo stesso risultato.

Nota : tutte le espressioni del generatore hanno le loro funzioni *equivalenti* , ma non viceversa.

Un'espressione di generatore può essere utilizzata senza parentesi se entrambe le parentesi si ripetessero diversamente:

```
sum(i for i in range(10) if i % 2 == 0)   #Output: 20
any(x = 0 for x in foo)                   #Output: True or False depending on foo
type(a > b for a in foo if a % 2 == 1)    #Output: <class 'generator'>
```

Invece di:

```
sum((i for i in range(10) if i % 2 == 0))
any((x = 0 for x in foo))
type((a > b for a in foo if a % 2 == 1))
```

Ma no:

```
fooFunction(i for i in range(10) if i % 2 == 0,foo,bar)
return x = 0 for x in foo
barFunction(baz, a > b for a in foo if a % 2 == 1)
```

Chiamando una funzione generatore si genera un **oggetto generatore** , che può essere successivamente ripetuto. A differenza di altri tipi di iteratori, gli oggetti generatore possono essere attraversati solo una volta.

```
g1 = function()
print(g1)  # Out: <generator object function at 0x1012e1888>
```

Si noti che il corpo di un generatore **non** viene immediatamente eseguito: quando si chiama `function()` nell'esempio precedente, viene immediatamente restituito un oggetto generatore, senza eseguire nemmeno la prima istruzione print. Ciò consente ai generatori di consumare meno memoria rispetto alle funzioni che restituiscono una lista e consente la creazione di generatori che producono sequenze infinitamente lunghe.

Per questo motivo, i generatori vengono spesso utilizzati nella scienza dei dati e in altri contesti che coinvolgono grandi quantità di dati. Un altro vantaggio è che l'altro codice può utilizzare immediatamente i valori prodotti da un generatore, senza attendere che venga prodotta la sequenza completa.

Tuttavia, se è necessario utilizzare i valori prodotti da un generatore più di una volta, e se generarli costa più della memorizzazione, potrebbe essere preferibile memorizzare i valori ottenuti come `list` piuttosto che rigenerare la sequenza. Vedi 'Ripristino di un generatore' sotto per maggiori dettagli.

In genere un oggetto generatore viene utilizzato in un ciclo o in qualsiasi funzione che richiede un iterable:

```
for x in g1:
    print("Received", x)

# Output:
```

```
# Received 0
# Received 1
# Received 4
# Received 9
# Received 16
# Received 25
# Received 36
# Received 49
# Received 64
# Received 81

arr1 = list(g1)
# arr1 = [], because the loop above already consumed all the values.
g2 = function()
arr2 = list(g2)  # arr2 = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Poiché gli oggetti generatore sono iteratori, uno può scorrere su di essi manualmente usando la funzione `next()` . Fare così restituirà i valori ottenuti uno per uno su ogni chiamata successiva.

Sotto il cofano, ogni volta che chiamate `next()` su un generatore, Python esegue istruzioni nel corpo della funzione generatore fino a quando non raggiunge la successiva dichiarazione di `yield` . A questo punto restituisce l'argomento del comando `yield` e ricorda il punto in cui è successo. Chiamando `next()` ancora una volta riprenderà l'esecuzione da quel punto e continuerà fino alla prossima dichiarazione di `yield` .

Se Python raggiunge la fine della funzione del generatore senza incontrare altri `yield` s, viene sollevata un'eccezione `StopIteration` (questo è normale, tutti gli iteratori si comportano allo stesso modo).

```
g3 = function()
a = next(g3)  # a becomes 0
b = next(g3)  # b becomes 1
c = next(g3)  # c becomes 2
...
j = next(g3)  # Raises StopIteration, j remains undefined
```

Si noti che in Python 2 gli oggetti del generatore avevano metodi `.next()` che potevano essere utilizzati per scorrere manualmente i valori ottenuti. In Python 3 questo metodo è stato sostituito con lo standard `.__next__()` per tutti gli iteratori.

Reimpostazione di un generatore

Ricorda che puoi solo scorrere gli oggetti generati da un generatore *una sola volta* . Se avete già iterato attraverso gli oggetti in uno script, ogni ulteriore tentativo di farlo produrrà `None` .

Se è necessario utilizzare gli oggetti generati da un generatore più di una volta, è possibile definire nuovamente la funzione generatore e utilizzarla una seconda volta oppure, in alternativa, è possibile memorizzare l'uscita della funzione generatore in un elenco al primo utilizzo. La ridefinizione della funzione del generatore sarà una buona opzione se si hanno a che fare con grandi volumi di dati e la memorizzazione di un elenco di tutti gli elementi di dati richiederebbe molto spazio sul disco. Viceversa, se inizialmente è costoso generare gli articoli, è preferibile memorizzare gli articoli generati in un elenco in modo da poterli riutilizzare.

Usare un generatore per trovare i numeri di Fibonacci

Un caso pratico d'uso di un generatore è quello di scorrere i valori di una serie infinita. Ecco un esempio di come trovare i primi dieci termini della sequenza di Fibonacci .

```
def fib(a=0, b=1):
    """Generator that yields Fibonacci numbers. `a` and `b` are the seed values"""
    while True:
        yield a
        a, b = b, a + b

f = fib()
print(', '.join(str(next(f)) for _ in range(10)))
```

0, 1, 1, 2, 3, 5, 8, 13, 21, 34

Sequenze infinite

I generatori possono essere utilizzati per rappresentare sequenze infinite:

```
def integers_starting_from(n):
    while True:
        yield n
        n += 1

natural_numbers = integers_starting_from(1)
```

La sequenza infinita di numeri come sopra può anche essere generata con l'aiuto di `itertools.count` . Il codice sopra potrebbe essere scritto come sotto

```
natural_numbers = itertools.count(1)
```

È possibile utilizzare la comprensione del generatore su generatori infiniti per produrre nuovi generatori:

```
multiples_of_two = (x * 2 for x in natural_numbers)
multiples_of_three = (x for x in natural_numbers if x % 3 == 0)
```

Siate consapevoli del fatto che un generatore infinito non ha una fine, quindi passarlo a qualsiasi funzione che tenterà di consumare del tutto il generatore avrà **conseguenze disastrose** :

```
list(multiples_of_two)  # will never terminate, or raise an OS-specific error
```

Invece, usa list / set comprehensions con `range` (o `xrange` per python <3.0):

```
first_five_multiples_of_three = [next(multiples_of_three) for _ in range(5)]
# [3, 6, 9, 12, 15]
```

oppure usa `itertools.islice()` per `itertools.islice()` l'iteratore in un sottoinsieme:

```
from itertools import islice
multiples_of_four = (x * 4 for x in integers_starting_from(1))
first_five_multiples_of_four = list(islice(multiples_of_four, 5))
# [4, 8, 12, 16, 20]
```

Nota che anche il generatore originale viene aggiornato, proprio come tutti gli altri generatori che provengono dalla stessa "root":

```
next(natural_numbers)    # yields 16
next(multiples_of_two)   # yields 34
next(multiples_of_four)  # yields 24
```

Una sequenza infinita può anche essere iterata con un `for` -loop . Assicurati di includere un'istruzione di `break` condizionale in modo che il ciclo possa terminare alla fine:

```
for idx, number in enumerate(multiplies_of_two):
    print(number)
    if idx == 9:
        break  # stop after taking the first 10 multiplies of two
```

Esempio classico: numeri di Fibonacci

```
import itertools

def fibonacci():
    a, b = 1, 1
    while True:
        yield a
        a, b = b, a + b

first_ten_fibs = list(itertools.islice(fibonacci(), 10))
# [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]

def nth_fib(n):
    return next(itertools.islice(fibonacci(), n - 1, n))

ninety_nineth_fib = nth_fib(99)  # 354224848179261915075
```

Cedendo tutti i valori da un altro iterabile

Python 3.x 3.3

Usa `yield from` se vuoi cedere tutti i valori da un altro iterabile:

```
def foob(x):
    yield from range(x * 2)
    yield from range(2)

list(foob(5))  # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1]
```

Funziona anche con i generatori.

```
def fibto(n):
    a, b = 1, 1
    while True:
        if a >= n: break
        yield a
        a, b = b, a + b

def usefib():
    yield from fibto(10)
    yield from fibto(20)

list(usefib())  # [1, 1, 2, 3, 5, 8, 1, 1, 2, 3, 5, 8, 13]
```

coroutine

I generatori possono essere utilizzati per implementare le coroutine:

```
# create and advance generator to the first yield
def coroutine(func):
    def start(*args,**kwargs):
        cr = func(*args,**kwargs)
        next(cr)
        return cr
    return start

# example coroutine
@coroutine
def adder(sum = 0):
    while True:
        x = yield sum
        sum += x

# example use
s = adder()
s.send(1) # 1
s.send(2) # 3
```

Coroutine sono comunemente utilizzate per implementare macchine a stati, in quanto sono principalmente utili per la creazione di procedure a metodo singolo che richiedono uno stato per funzionare correttamente. Operano su uno stato esistente e restituiscono il valore ottenuto al termine dell'operazione.

Resa con ricorsione: elenca in modo ricorsivo tutti i file in una directory

Innanzitutto, importa le librerie che funzionano con i file:

```
from os import listdir
from os.path import isfile, join, exists
```

Una funzione di supporto per leggere solo i file da una directory:

```
def get_files(path):
    for file in listdir(path):
        full_path = join(path, file)
```

```
        if isfile(full_path):
            if exists(full_path):
                yield full_path
```

Un'altra funzione di supporto per ottenere solo le sottodirectory:

```
def get_directories(path):
    for directory in listdir(path):
        full_path = join(path, directory)
        if not isfile(full_path):
            if exists(full_path):
                yield full_path
```

Ora usa queste funzioni per ottenere ricorsivamente tutti i file all'interno di una directory e tutte le sue sottodirectory (usando i generatori):

```
def get_files_recursive(directory):
    for file in get_files(directory):
        yield file
    for subdirectory in get_directories(directory):
        for file in get_files_recursive(subdirectory): # here the recursive call
            yield file
```

Questa funzione può essere semplificata utilizzando il `yield from` :

```
def get_files_recursive(directory):
    yield from get_files(directory)
    for subdirectory in get_directories(directory):
        yield from get_files_recursive(subdirectory)
```

Iterazione su generatori in parallelo

Per eseguire un'iterazione su più generatori in parallelo, utilizzare lo `zip` incorporato:

```
for x, y in zip(a,b):
    print(x,y)
```

Risultati in:

```
1 x
2 y
3 z
```

In python 2 dovresti invece usare `itertools.izip` . Qui possiamo anche vedere che tutte le funzioni di `zip` producono tuple.

Nota che zip interromperà l'iterazione non appena uno degli iterabili finirà gli elementi. Se desideri ripetere l'iterazione fino a quando è il più lungo iterabile, usa `itertools.zip_longest()` .

Refactoring code-list code

Supponiamo che tu abbia un codice complesso che crea e restituisce un elenco iniziando da un elenco vuoto e aggiungendolo ripetutamente ad esso:

```
def create():
    result = []
    # logic here...
    result.append(value) # possibly in several places
    # more logic...
    return result # possibly in several places

values = create()
```

Quando non è pratico sostituire la logica interna con una comprensione di lista, puoi trasformare l'intera funzione in un generatore sul posto e quindi raccogliere i risultati:

```
def create_gen():
    # logic...
    yield value
    # more logic
    return # not needed if at the end of the function, of course

values = list(create_gen())
```

Se la logica è ricorsiva, usa `yield from` per includere tutti i valori della chiamata ricorsiva in un risultato "appiattito":

```
def preorder_traversal(node):
    yield node.value
    for child in node.children:
        yield from preorder_traversal(child)
```

Ricerca

La `next` funzione è utile anche senza iterare. Passare un'espressione di generatore al `next` è un modo rapido per cercare la prima occorrenza di un elemento che corrisponde ad un predicato. Codice procedurale come

```
def find_and_transform(sequence, predicate, func):
    for element in sequence:
        if predicate(element):
            return func(element)
    raise ValueError

item = find_and_transform(my_sequence, my_predicate, my_func)
```

può essere sostituito con:

```
item = next(my_func(x) for x in my_sequence if my_predicate(x))
# StopIteration will be raised if there are no matches; this exception can
# be caught and transformed, if desired.
```

A tale scopo, potrebbe essere opportuno creare un alias, come `first = next` , o una funzione

wrapper per convertire l'eccezione:

```
def first(generator):
    try:
        return next(generator)
    except StopIteration:
        raise ValueError
```

Leggi generatori online: https://riptutorial.com/it/python/topic/292/generatori

Capitolo 70: Gestori di contesto ("con" istruzione)

introduzione

Sebbene i gestori di contesto di Python siano ampiamente utilizzati, pochi capiscono lo scopo dietro il loro uso. Queste istruzioni, comunemente utilizzate con i file di lettura e scrittura, aiutano l'applicazione a conservare la memoria di sistema e a migliorare la gestione delle risorse assicurando che risorse specifiche siano utilizzate solo per determinati processi. Questo argomento spiega e dimostra l'uso dei gestori di contesto di Python.

Sintassi

- con "context_manager" (come "alias") (, "context_manager" (come "alias")?) *:

Osservazioni

I gestori di contesto sono definiti in PEP 343 . Sono destinati a essere usati come meccanismo più succinto per la gestione delle risorse rispetto a `try ... finally` costrutti. La definizione formale è la seguente.

> In questo PEP, i gestori di contesto forniscono i `__enter__()` e `__exit__()` che vengono richiamati all'ingresso e all'uscita dal corpo `__enter__()` with.

Quindi continua a definire l'istruzione `with` come segue.

```
with EXPR as VAR:
    BLOCK
```

> La traduzione della dichiarazione di cui sopra è:

```
   mgr = (EXPR)
   exit = type(mgr).__exit__  # Not calling it yet
   value = type(mgr).__enter__(mgr)
   exc = True
   try:
       try:
           VAR = value  # Only if "as VAR" is present
           BLOCK
       except:
           # The exceptional case is handled here
           exc = False
           if not exit(mgr, *sys.exc_info()):
               raise
           # The exception is swallowed if exit() returns true
   finally:
       # The normal and non-local-goto cases are handled here
```

```
        if exc:
            exit(mgr, None, None, None)
```

Examples

Introduzione ai gestori di contesto e alla dichiarazione con

Un gestore di contesto è un oggetto che viene avvisato quando un contesto (un blocco di codice) *inizia* e *finisce* . Di solito ne usi uno con l'istruzione `with` . Si prende cura della notifica.

Ad esempio, gli oggetti file sono gestori contesto. Al termine di un contesto, l'oggetto file viene chiuso automaticamente:

```
open_file = open(filename)
with open_file:
    file_contents = open_file.read()

# the open_file object has automatically been closed.
```

L'esempio sopra è di solito semplificato usando la parola chiave `as` :

```
with open(filename) as open_file:
    file_contents = open_file.read()

# the open_file object has automatically been closed.
```

Tutto ciò che termina l'esecuzione del blocco causa il richiamo del metodo di uscita del gestore contesto. Ciò include le eccezioni e può essere utile quando un errore causa l'uscita prematura da un file o una connessione aperti. Uscire da uno script senza chiudere correttamente i file / le connessioni è una cattiva idea, che potrebbe causare la perdita di dati o altri problemi. Utilizzando un gestore di contesto è possibile garantire che vengano sempre adottate precauzioni per prevenire danni o perdite in questo modo. Questa funzione è stata aggiunta in Python 2.5.

Assegnare a un bersaglio

Molti gestori di contesto restituiscono un oggetto una volta inseriti. È possibile assegnare quell'oggetto a un nuovo nome nell'istruzione `with` .

Ad esempio, l'utilizzo di una connessione al database in un'istruzione `with` potrebbe fornire un oggetto cursore:

```
with database_connection as cursor:
    cursor.execute(sql_query)
```

Gli oggetti File si restituiscono da soli, ciò rende possibile sia aprire l'oggetto file che utilizzarlo come gestore di contesto in un'unica espressione:

```
with open(filename) as open_file:
```

```
    file_contents = open_file.read()
```

Scrivi il tuo gestore di contesto

Un gestore di contesto è un qualsiasi oggetto che implementa due metodi magici `__enter__()` e `__exit__()` (sebbene possa implementare anche altri metodi):

```
class AContextManager():

    def __enter__(self):
        print("Entered")
        # optionally return an object
        return "A-instance"

    def __exit__(self, exc_type, exc_value, traceback):
        print("Exited" + (" (with an exception)" if exc_type else ""))
        # return True if you want to suppress the exception
```

Se il contesto esce con un'eccezione, le informazioni su `exc_type` verranno passate come triplo `exc_type` , `exc_value` , `traceback` (queste sono le stesse variabili restituite dalla funzione `sys.exc_info()`). Se il contesto esce normalmente, tutti e tre questi argomenti saranno `None` .

Se si verifica un'eccezione e viene passata al metodo `__exit__` , il metodo può restituire `True` per sopprimere l'eccezione, altrimenti l'eccezione verrà `__exit__` alla fine della funzione `__exit__` .

```
with AContextManager() as a:
    print("a is %r" % a)
# Entered
# a is 'A-instance'
# Exited

with AContextManager() as a:
    print("a is %d" % a)
# Entered
# Exited (with an exception)
# Traceback (most recent call last):
#   File "<stdin>", line 2, in <module>
# TypeError: %d format: a number is required, not str
```

Si noti che nel secondo esempio anche se si verifica un'eccezione nel mezzo del corpo `__exit__` with, il gestore `__exit__` viene ancora eseguito, prima che l'eccezione si propaga all'ambito esterno.

Se hai solo bisogno di un metodo `__exit__` , puoi restituire l'istanza del gestore di contesto:

```
class MyContextManager:
    def __enter__(self):
        return self

    def __exit__(self):
        print('something')
```

Scrivere il proprio contextmanager usando la sintassi del generatore

È anche possibile scrivere un gestore di contesto usando la sintassi del generatore grazie al decoratore `contextlib.contextmanager` :

```
import contextlib

@contextlib.contextmanager
def context_manager(num):
    print('Enter')
    yield num + 1
    print('Exit')

with context_manager(2) as cm:
    # the following instructions are run when the 'yield' point of the context
    # manager is reached.
    # 'cm' will have the value that was yielded
    print('Right in the middle with cm = {}'.format(cm))
```

produce:

```
Enter
Right in the middle with cm = 3
Exit
```

Il decoratore semplifica il compito di scrivere un gestore di contesto convertendo un generatore in uno. Tutto prima che l'espressione yield diventi il metodo `__enter__` , il valore restituito diventa il valore restituito dal generatore (che può essere associato a una variabile `__exit__` with) e tutto ciò che `__exit__` l'espressione yield diventa il metodo `__exit__` .

Se un'eccezione deve essere gestita dal contesto manager, un `try..except..finally` -block può essere scritto nel generatore e qualsiasi eccezione sollevata nel `with` -block saranno trattati da questo blocco eccezione.

```
@contextlib.contextmanager
def error_handling_context_manager(num):
    print("Enter")
    try:
        yield num + 1
    except ZeroDivisionError:
        print("Caught error")
    finally:
        print("Cleaning up")
    print("Exit")

with error_handling_context_manager(-1) as cm:
    print("Dividing by cm = {}".format(cm))
    print(2 / cm)
```

Questo produce:

```
Enter
Dividing by cm = 0
Caught error
Cleaning up
Exit
```

Più gestori di contesto

Puoi aprire diversi gestori di contenuti allo stesso tempo:

```
with open(input_path) as input_file, open(output_path, 'w') as output_file:

    # do something with both files.

    # e.g. copy the contents of input_file into output_file
    for line in input_file:
        output_file.write(line + '\n')
```

Ha lo stesso effetto dei gestori di contesto di nesting:

```
with open(input_path) as input_file:
    with open(output_path, 'w') as output_file:
        for line in input_file:
            output_file.write(line + '\n')
```

Gestisci risorse

```
class File():
    def __init__(self, filename, mode):
        self.filename = filename
        self.mode = mode

    def __enter__(self):
        self.open_file = open(self.filename, self.mode)
        return self.open_file

    def __exit__(self, *args):
        self.open_file.close()
```

`__init__()` metodo `__init__()` imposta l'oggetto, in questo caso settando il nome del file e la modalità per aprire il file. `__enter__()` apre e restituisce il file e `__exit__()` lo chiude.

L'utilizzo di questi metodi magici (`__enter__` , `__exit__`) consente di implementare oggetti che possono essere utilizzati facilmente `with` l'istruzione with.

Usa la classe File:

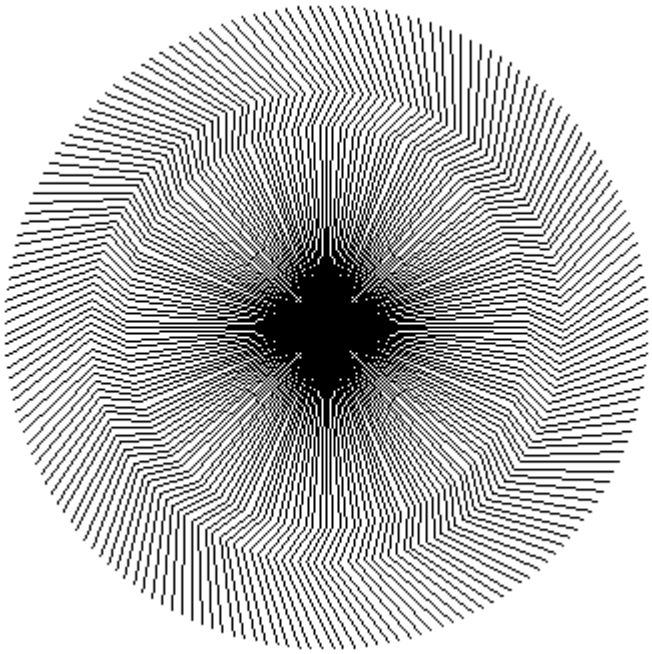
```
for _ in range(10000):
    with File('foo.txt', 'w') as f:
        f.write('foo')
```

Leggi Gestori di contesto ("con" istruzione) online:
https://riptutorial.com/it/python/topic/928/gestori-di-contesto---con--istruzione-

Capitolo 71: Grafica tartaruga

Examples

Ninja Twist (Turtle Graphics)



Qui una tartaruga Ninja Twist:

```
import turtle

ninja = turtle.Turtle()

ninja.speed(10)

for i in range(180):
    ninja.forward(100)
    ninja.right(30)
    ninja.forward(20)
    ninja.left(60)
    ninja.forward(50)
    ninja.right(30)

    ninja.penup()
    ninja.setposition(0, 0)
    ninja.pendown()

    ninja.right(2)

turtle.done()
```

Leggi Grafica tartaruga online: https://riptutorial.com/it/python/topic/7915/grafica-tartaruga

Capitolo 72: grafico-utensile

introduzione

Gli strumenti Python possono essere utilizzati per generare grafici

Examples

PyDotPlus

PyDotPlus è una versione migliorata del vecchio progetto pydot che fornisce un'interfaccia Python al linguaggio Dot di Graphviz.

Installazione

Per l'ultima versione stabile:

```
pip install pydotplus
```

Per la versione di sviluppo:

```
pip install https://github.com/carlos-jenkins/pydotplus/archive/master.zip
```

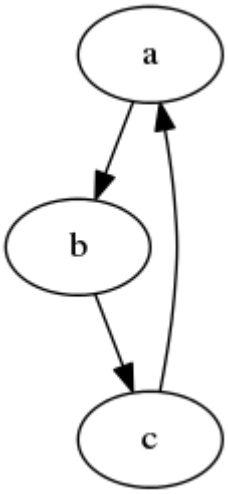
Carica il grafico come definito da un file DOT

- Si presume che il file sia nel formato DOT. Verrà caricato, analizzato e verrà restituita una classe Dot, che rappresenta il grafico. Ad esempio, un semplice demo.dot:

  digraph demo1 {a -> b -> c; c -> a; }

```
import pydotplus
graph_a = pydotplus.graph_from_dot_file('demo.dot')
graph_a.write_svg('test.svg') # generate graph in svg.
```

Otterrai una svg (Scalable Vector Graphics) come questa:



PyGraphviz

Ottieni PyGraphviz dall'Indice dei pacchetti Python su http://pypi.python.org/pypi/pygraphviz

o installarlo con:

```
pip install pygraphviz
```

e si tenterà di trovare e installare una versione appropriata che corrisponda al sistema operativo e alla versione di Python.

Puoi installare la versione di sviluppo (su github.com) con:

```
pip install git://github.com/pygraphviz/pygraphviz.git#egg=pygraphviz
```

Ottieni PyGraphviz dall'Indice dei pacchetti Python su http://pypi.python.org/pypi/pygraphviz

o installarlo con:

```
easy_install pygraphviz
```

e si tenterà di trovare e installare una versione appropriata che corrisponda al sistema operativo e alla versione di Python.

Carica il grafico come definito da un file DOT

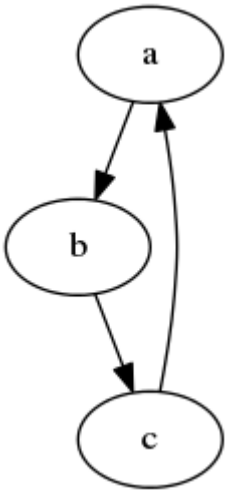
- Si presume che il file sia nel formato DOT. Verrà caricato, analizzato e verrà restituita una classe Dot, che rappresenta il grafico. Ad esempio, un semplice demo.dot:

  digraph demo1 {a -> b -> c; c -> a; }

- Caricalo e disegnalo.

```
import pygraphviz as pgv
G = pgv.AGraph("demo.dot")
G.draw('test', format='svg', prog='dot')
```

Otterrai una svg (Scalable Vector Graphics) come questa:



Leggi grafico-utensile online: https://riptutorial.com/it/python/topic/9483/grafico-utensile

Capitolo 73: hashlib

introduzione

hashlib implementa un'interfaccia comune a molti diversi algoritmi di hash sicuro e di messaggistica dei messaggi. Sono inclusi gli algoritmi di hash sicuro FIPS SHA1, SHA224, SHA256, SHA384 e SHA512.

Examples

Hash MD5 di una stringa

Questo modulo implementa un'interfaccia comune a molti diversi algoritmi di hash sicuro e di messaggistica dei messaggi. Sono inclusi gli algoritmi di hash sicuro FAS SHA1, SHA224, SHA256, SHA384 e SHA512 (definiti in FIPS 180-2) e l'algoritmo MD5 di RSA (definito in Internet RFC 1321).

Esiste un metodo di costruzione denominato per ciascun tipo di hash. Tutti restituiscono un oggetto hash con la stessa semplice interfaccia. Ad esempio: utilizzare `sha1()` per creare un oggetto hash SHA1.

```
hash.sha1()
```

I costruttori per gli algoritmi di hash che sono sempre presenti in questo modulo sono `md5()` , `sha1()` , `sha224()` , `sha256()` , `sha384()` e `sha512()` .

Ora puoi alimentare questo oggetto con stringhe arbitrarie usando il metodo `update()` . In qualsiasi momento puoi chiedergli il riassunto della concatenazione delle stringhe ad esso alimentate fino ad ora usando i metodi `digest()` o `hexdigest()` .

```
hash.update(arg)
```

> Aggiorna l'oggetto hash con la stringa arg. Le chiamate ripetute sono equivalenti a una singola chiamata con la concatenazione di tutti gli argomenti: m.update (a); m.update (b) è equivalente a m.update (a + b).

```
hash.digest()
```

> Restituisce il riassunto delle stringhe passato al metodo update () finora. Questa è una stringa di byte digest_size che può contenere caratteri non ASCII, inclusi byte null.

```
hash.hexdigest()
```

> Come digest () eccetto che il digest viene restituito come una stringa di doppia lunghezza, contenente solo cifre esadecimali. Questo può essere usato per scambiare

il valore in sicurezza in e-mail o in altri ambienti non binari.

Ecco un esempio:

```
>>> import hashlib
>>> m = hashlib.md5()
>>> m.update("Nobody inspects")
>>> m.update(" the spammish repetition")
>>> m.digest()
'\xbbd\x9c\x83\xdd\x1e\xa5\xc9\xd9\xde\xc9\xa1\x8d\xf0\xff\xe9'
>>> m.hexdigest()
'bb649c83dd1ea5c9d9dec9a18df0ffe9'
>>> m.digest_size
16
>>> m.block_size
64
```

o:

```
hashlib.md5("Nobody inspects the spammish repetition").hexdigest()
    'bb649c83dd1ea5c9d9dec9a18df0ffe9'
```

algoritmo fornito da OpenSSL

Esiste anche un costruttore `new()` generico che accetta il nome stringa dell'algoritmo desiderato come primo parametro per consentire l'accesso agli hash elencati sopra così come a qualsiasi altro algoritmo che la libreria OpenSSL può offrire. I costruttori denominati sono molto più veloci di `new()` e dovrebbero essere preferiti.

Usando `new()` con un algoritmo fornito da OpenSSL:

```
>>> h = hashlib.new('ripemd160')
>>> h.update("Nobody inspects the spammish repetition")
>>> h.hexdigest()
'cc4a5ce1b3df48aec5d22d1f16b894a0b894eccc'
```

Leggi hashlib online: https://riptutorial.com/it/python/topic/8980/hashlib

Capitolo 74: Heapq

Examples

Articoli più grandi e più piccoli in una collezione

Per trovare gli elementi più grandi in una raccolta, il modulo `heapq` ha una funzione chiamata `nlargest` , gli passiamo due argomenti, il primo è il numero di elementi che vogliamo recuperare, il secondo è il nome della raccolta:

```
import heapq


numbers = [1, 4, 2, 100, 20, 50, 32, 200, 150, 8]
print(heapq.nlargest(4, numbers))  # [200, 150, 100, 50]
```

Allo stesso modo, per trovare gli elementi più piccoli in una raccolta, usiamo la funzione `nsmallest` :

```
print(heapq.nsmallest(4, numbers))  # [1, 2, 4, 8]
```

Entrambe le funzioni `nlargest` e `nsmallest` accettano un argomento opzionale (parametro chiave) per strutture dati complicate. Nell'esempio seguente viene illustrato l'utilizzo della proprietà `age` per recuperare il dizionario più vecchio e più recente del `people` :

```
people = [
    {'firstname': 'John', 'lastname': 'Doe', 'age': 30},
    {'firstname': 'Jane', 'lastname': 'Doe', 'age': 25},
    {'firstname': 'Janie', 'lastname': 'Doe', 'age': 10},
    {'firstname': 'Jane', 'lastname': 'Roe', 'age': 22},
    {'firstname': 'Johnny', 'lastname': 'Doe', 'age': 12},
    {'firstname': 'John', 'lastname': 'Roe', 'age': 45}
]

oldest = heapq.nlargest(2, people, key=lambda s: s['age'])
print(oldest)
# Output: [{'firstname': 'John', 'age': 45, 'lastname': 'Roe'}, {'firstname': 'John', 'age':
30, 'lastname': 'Doe'}]

youngest = heapq.nsmallest(2, people, key=lambda s: s['age'])
print(youngest)
# Output: [{'firstname': 'Janie', 'age': 10, 'lastname': 'Doe'}, {'firstname': 'Johnny',
'age': 12, 'lastname': 'Doe'}]
```

Il più piccolo oggetto di una collezione

La proprietà più interessante di un `heap` è che il suo elemento più piccolo è sempre il primo elemento: `heap[0]`

```
import heapq
```

```
numbers = [10, 4, 2, 100, 20, 50, 32, 200, 150, 8]

heapq.heapify(numbers)
print(numbers)
# Output: [2, 4, 10, 100, 8, 50, 32, 200, 150, 20]

heapq.heappop(numbers)  # 2
print(numbers)
# Output: [4, 8, 10, 100, 20, 50, 32, 200, 150]

heapq.heappop(numbers)  # 4
print(numbers)
# Output:  [8, 20, 10, 100, 150, 50, 32, 200]
```

Leggi Heapq online: https://riptutorial.com/it/python/topic/7489/heapq

Capitolo 75: idiomi

Examples

Inizializzazione chiavi del dizionario

Preferisci il metodo `dict.get` se non sei sicuro che la chiave sia presente. Ti consente di restituire un valore predefinito se la chiave non viene trovata. Il metodo tradizionale `dict[key]` solleva un'eccezione `KeyError` .

Piuttosto che farlo

```
def add_student():
    try:
        students['count'] += 1
    except KeyError:
        students['count'] = 1
```

Fare

```
def add_student():
        students['count'] = students.get('count', 0) + 1
```

Commutazione delle variabili

Per cambiare il valore di due variabili puoi usare tuple unpacking.

```
x = True
y = False
x, y = y, x
x
# False
y
# True
```

Utilizzare il test del valore di verità

Python convertirà implicitamente qualsiasi oggetto in un valore booleano per il test, quindi usalo dove possibile.

```
# Good examples, using implicit truth testing
if attr:
    # do something

if not attr:
    # do something

# Bad examples, using specific types
if attr == 1:
```

```
    # do something

if attr == True:
    # do something

if attr != '':
    # do something

# If you are looking to specifically check for None, use 'is' or 'is not'
if attr is None:
    # do something
```

Generalmente questo produce un codice più leggibile e di solito è molto più sicuro quando si tratta di tipi imprevisti.

Fare clic qui per un elenco di ciò che verrà valutato su `False` .

Prova per "__main__" per evitare l'esecuzione inaspettata del codice

È buona norma testare la variabile `__name__` del programma chiamante prima di eseguire il codice.

```
import sys

def main():
    # Your code starts here

    # Don't forget to provide a return code
    return 0

if __name__ == "__main__":
    sys.exit(main())
```

L'utilizzo di questo modello garantisce che il codice venga eseguito solo quando ci si aspetta che lo sia; ad esempio, quando esegui il tuo file esplicitamente:

```
python my_program.py
```

Il vantaggio, tuttavia, arriva se si decide di `import` il file in un altro programma (ad esempio se lo si scrive come parte di una libreria). È quindi possibile `import` il file e il trap `__main__` garantirà che nessun codice venga eseguito in modo imprevisto:

```
# A new program file
import my_program        # main() is not run

# But you can run main() explicitly if you really want it to run:
my_program.main()
```

Leggi idiomi online: https://riptutorial.com/it/python/topic/3070/idiomi

Capitolo 76: ijson

introduzione

ijson è una grande libreria per lavorare con i file JSON in Python. Sfortunatamente, per impostazione predefinita usa un parser Python JSON puro come backend. È possibile ottenere prestazioni molto più elevate utilizzando un backend C.

Examples

Semplice esempio

Esempio di esempio Tratto da un benchmarking

```
import ijson

def load_json(filename):
    with open(filename, 'r') as fd:
        parser = ijson.parse(fd)
        ret = {'builders': {}}
        for prefix, event, value in parser:
            if (prefix, event) == ('builders', 'map_key'):
                buildername = value
                ret['builders'][buildername] = {}
            elif prefix.endswith('.shortname'):
                ret['builders'][buildername]['shortname'] = value

        return ret

if __name__ == "__main__":
    load_json('allthethings.json')
```

JSON FILE LINK

Leggi ijson online: https://riptutorial.com/it/python/topic/8342/ijson

Capitolo 77: Il modulo base64

introduzione

La codifica Base 64 rappresenta uno schema comune per la codifica binaria in formato di stringa ASCII usando radix 64. Il modulo base64 fa parte della libreria standard, il che significa che si installa insieme a Python. La comprensione di byte e stringhe è fondamentale per questo argomento e può essere esaminata qui . Questo argomento spiega come utilizzare le varie funzioni e basi numeriche del modulo base64.

Sintassi

- base64.b64encode (s, altchars = Nessuno)
- base64.b64decode (s, altchars = None, validate = False)
- base64.standard_b64encode (s)
- base64.standard_b64decode (s)
- base64.urlsafe_b64encode (s)
- base64.urlsafe_b64decode (s)
- base64.b32encode (s)
- base64.b32decode (s)
- base64.b16encode (s)
- base64.b16decode (s)
- base64.a85encode (b, foldspaces = False, wrapcol = 0, pad = False, adobe = False)
- base64.a85decode (b, foldpaces = False, adobe = False, ignorechars = b '\ t \ n \ r \ v')
- base64.b85encode (b, pad = False)
- base64.b85decode (b)

Parametri

Parametro	Descrizione
`base64.b64encode(s, altchars=None)`	
S	Un oggetto simile a un byte
altchars	Un oggetto simile a byte di lunghezza 2+ di caratteri per sostituire i caratteri '+' e '=' durante la creazione dell'alfabeto Base64. I personaggi extra vengono ignorati.
`base64.b64decode(s, altchars=None, validate=False)`	
S	Un oggetto simile a un byte
altchars	Un oggetto simile a byte di lunghezza 2+ di

Parametro	Descrizione
	caratteri per sostituire i caratteri '+' e '=' durante la creazione dell'alfabeto Base64. I personaggi extra vengono ignorati.
convalidare	Se valida è True, i caratteri non nel normale alfabeto Base64 o nell'alfabeto alternativo **non** vengono scartati prima del controllo del riempimento
`base64.standard_b64encode(s)`	
S	Un oggetto simile a un byte
`base64.standard_b64decode(s)`	
S	Un oggetto simile a un byte
`base64.urlsafe_b64encode(s)`	
S	Un oggetto simile a un byte
`base64.urlsafe_b64decode(s)`	
S	Un oggetto simile a un byte
`b32encode(s)`	
S	Un oggetto simile a un byte
`b32decode(s)`	
S	Un oggetto simile a un byte
`base64.b16encode(s)`	
S	Un oggetto simile a un byte
`base64.b16decode(s)`	
S	Un oggetto simile a un byte
`base64.a85encode(b, foldspaces=False, wrapcol=0, pad=False, adobe=False)`	
B	Un oggetto simile a un byte
foldspaces	Se i foldspaces sono True, verrà utilizzato il carattere 'y' invece di 4 spazi consecutivi.
wrapcol	Il numero di caratteri prima di una nuova riga (0 non implica nessuna nuova riga)
tampone	Se pad è True, i byte vengono riempiti su un

Parametro	Descrizione
	multiplo di 4 prima della codifica
adobe	Se Adobe è True, le sequenze codificate vengono incorniciate con '<~' e '' ~> 'come usate con i prodotti Adobe
`base64.a85decode(b, foldspaces=False, adobe=False, ignorechars=b'\t\n\r\v')`	
B	Un oggetto simile a un byte
foldspaces	Se i foldspaces sono True, verrà utilizzato il carattere 'y' invece di 4 spazi consecutivi.
adobe	Se Adobe è True, le sequenze codificate vengono incorniciate con '<~' e '' ~> 'come usate con i prodotti Adobe
ignorechars	Un oggetto simile a un byte di caratteri da ignorare nel processo di codifica
`base64.b85encode(b, pad=False)`	
B	Un oggetto simile a un byte
tampone	Se pad è True, i byte vengono riempiti su un multiplo di 4 prima della codifica
`base64.b85decode(b)`	
B	Un oggetto simile a un byte

Osservazioni

Fino a quando non uscì Python 3.4, le funzioni di codifica e decodifica base64 funzionavano solo con i tipi di `bytes` o di `bytearray` . Ora queste funzioni accettano qualsiasi oggetto simile a un byte .

Examples

Codifica e decodifica Base64

Per includere il modulo base64 nello script, devi prima importarlo:

```
import base64
```

Le funzioni di codifica e decodifica base64 richiedono entrambi un oggetto simile a un byte . Per ottenere la nostra stringa in byte, dobbiamo codificarla usando la funzione di codifica incorporata

di Python. Più comunemente, viene utilizzata la codifica `UTF-8` , tuttavia un elenco completo di queste codifiche standard (incluse le lingue con caratteri diversi) può essere trovato qui nella documentazione ufficiale di Python. Di seguito è riportato un esempio di codifica di una stringa in byte:

```
s = "Hello World!"
b = s.encode("UTF-8")
```

L'output dell'ultima riga sarà:

```
b'Hello World!'
```

Il prefisso `b` viene utilizzato per indicare che il valore è un oggetto byte.

Per Base64 codificare questi byte, usiamo la funzione `base64.b64encode()` :

```
import base64
s = "Hello World!"
b = s.encode("UTF-8")
e = base64.b64encode(b)
print(e)
```

Quel codice produrrebbe quanto segue:

```
b'SGVsbG8gV29ybGQh'
```

che è ancora nell'oggetto bytes. Per ottenere una stringa da questi byte, possiamo usare il metodo `decode()` di Python con la codifica `UTF-8` :

```
import base64
s = "Hello World!"
b = s.encode("UTF-8")
e = base64.b64encode(b)
s1 = e.decode("UTF-8")
print(s1)
```

L'output sarebbe quindi:

```
SGVsbG8gV29ybGQh
```

Se volessimo codificare la stringa e decodificare, potremmo usare il metodo `base64.b64decode()` :

```
import base64
# Creating a string
s = "Hello World!"
# Encoding the string into bytes
b = s.encode("UTF-8")
# Base64 Encode the bytes
e = base64.b64encode(b)
# Decoding the Base64 bytes to string
s1 = e.decode("UTF-8")
# Printing Base64 encoded string
print("Base64 Encoded:", s1)
# Encoding the Base64 encoded string into bytes
```

```
b1 = s1.encode("UTF-8")
# Decoding the Base64 bytes
d = base64.b64decode(b1)
# Decoding the bytes to string
s2 = d.decode("UTF-8")
print(s2)
```

Come ci si potrebbe aspettare, l'output sarebbe la stringa originale:

```
Base64 Encoded: SGVsbG8gV29ybGQh
Hello World!
```

Codifica e decodifica di Base32

Il modulo base64 include anche funzioni di codifica e decodifica per Base32. Queste funzioni sono molto simili alle funzioni Base64:

```
import base64
# Creating a string
s = "Hello World!"
# Encoding the string into bytes
b = s.encode("UTF-8")
# Base32 Encode the bytes
e = base64.b32encode(b)
# Decoding the Base32 bytes to string
s1 = e.decode("UTF-8")
# Printing Base32 encoded string
print("Base32 Encoded:", s1)
# Encoding the Base32 encoded string into bytes
b1 = s1.encode("UTF-8")
# Decoding the Base32 bytes
d = base64.b32decode(b1)
# Decoding the bytes to string
s2 = d.decode("UTF-8")
print(s2)
```

Questo produrrebbe il seguente risultato:

```
Base32 Encoded: JBSWY3DPEBLW64TMMQQQ====
Hello World!
```

Codifica e decodifica Base16

Il modulo base64 include anche funzioni di codifica e decodifica per Base16. La base 16 viene comunemente indicata come **esadecimale** . Queste funzioni sono molto simili alle funzioni Base64 e Base32:

```
import base64
# Creating a string
s = "Hello World!"
# Encoding the string into bytes
b = s.encode("UTF-8")
# Base16 Encode the bytes
```

```
e = base64.b16encode(b)
# Decoding the Base16 bytes to string
s1 = e.decode("UTF-8")
# Printing Base16 encoded string
print("Base16 Encoded:", s1)
# Encoding the Base16 encoded string into bytes
b1 = s1.encode("UTF-8")
# Decoding the Base16 bytes
d = base64.b16decode(b1)
# Decoding the bytes to string
s2 = d.decode("UTF-8")
print(s2)
```

Questo produrrebbe il seguente risultato:

```
Base16 Encoded: 48656C6C6F20576F726C6421
Hello World!
```

Codifica e decodifica ASCII85

Adobe ha creato la propria codifica **ASCII85** che è simile a Base85, ma ha le sue differenze. Questa codifica viene utilizzata frequentemente nei file Adobe PDF. Queste funzioni sono state rilasciate in Python versione 3.4. Altrimenti, le funzioni `base64.a85encode()` e `base64.a85encode()` sono simili alle precedenti:

```
import base64
# Creating a string
s = "Hello World!"
# Encoding the string into bytes
b = s.encode("UTF-8")
# ASCII85 Encode the bytes
e = base64.a85encode(b)
# Decoding the ASCII85 bytes to string
s1 = e.decode("UTF-8")
# Printing ASCII85 encoded string
print("ASCII85 Encoded:", s1)
# Encoding the ASCII85 encoded string into bytes
b1 = s1.encode("UTF-8")
# Decoding the ASCII85 bytes
d = base64.a85decode(b1)
# Decoding the bytes to string
s2 = d.decode("UTF-8")
print(s2)
```

Questo produce quanto segue:

```
ASCII85 Encoded: 87cURD]i,"Ebo80
Hello World!
```

Codifica e decodifica Base85

Proprio come le funzioni Base64, Base32 e Base16, le funzioni di codifica e decodifica `base64.b85encode()` sono `base64.b85encode()` e `base64.b85decode()` :

```
import base64
# Creating a string
s = "Hello World!"
# Encoding the string into bytes
b = s.encode("UTF-8")
# Base85 Encode the bytes
e = base64.b85encode(b)
# Decoding the Base85 bytes to string
s1 = e.decode("UTF-8")
# Printing Base85 encoded string
print("Base85 Encoded:", s1)
# Encoding the Base85 encoded string into bytes
b1 = s1.encode("UTF-8")
# Decoding the Base85 bytes
d = base64.b85decode(b1)
# Decoding the bytes to string
s2 = d.decode("UTF-8")
print(s2)
```

che emette quanto segue:

```
Base85 Encoded: NM&qnZy;B1a%^NF
Hello World!
```

Leggi Il modulo base64 online: https://riptutorial.com/it/python/topic/8678/il-modulo-base64

Capitolo 78: Il modulo dis

Examples

Costanti nel modulo DIS

```
EXTENDED_ARG = 145 # All opcodes greater than this have 2 operands
HAVE_ARGUMENT = 90 # All opcodes greater than this have at least 1 operands

cmp_op = ('<', '<=', '==', '!=', '>', '>=', 'in', 'not in', 'is', 'is ...
       # A list of comparator id's. The indecies are used as operands in some opcodes

# All opcodes in these lists have the respective types as there operands
hascompare = [107]
hasconst = [100]
hasfree = [135, 136, 137]
hasjabs = [111, 112, 113, 114, 115, 119]
hasjrel = [93, 110, 120, 121, 122, 143]
haslocal = [124, 125, 126]
hasname = [90, 91, 95, 96, 97, 98, 101, 106, 108, 109, 116]

# A map of opcodes to ids
opmap = {'BINARY_ADD': 23, 'BINARY_AND': 64, 'BINARY_DIVIDE': 21, 'BIN...
# A map of ids to opcodes
opname = ['STOP_CODE', 'POP_TOP', 'ROT_TWO', 'ROT_THREE', 'DUP_TOP', '...
```

Cos'è il bytecode Python?

Python è un interprete ibrido. Quando si esegue un programma, prima lo assembla in un *bytecode* che può quindi essere eseguito nell'interprete Python (chiamato anche una *macchina virtuale Python*). Il modulo `dis` nella libreria standard può essere utilizzato per rendere leggibile il bytecode Python smontando classi, metodi, funzioni e oggetti codice.

```
>>> def hello():
...     print "Hello, World"
...
>>> dis.dis(hello)
  2           0 LOAD_CONST               1 ('Hello, World')
              3 PRINT_ITEM
              4 PRINT_NEWLINE
              5 LOAD_CONST               0 (None)
              8 RETURN_VALUE
```

L'interprete Python è basato sullo stack e utilizza un primo sistema di ultimo esaurimento.

Ogni codice operazione (opcode) nel linguaggio assembly Python (il bytecode) prende un numero fisso di elementi dallo stack e restituisce un numero fisso di elementi allo stack. Se non ci sono abbastanza elementi in pila per un opcode, l'interprete Python si bloccherà, probabilmente senza un messaggio di errore.

Disassemblare i moduli

Per disassemblare un modulo Python, prima questo deve essere trasformato in un file `.pyc` (compilato con Python). Per fare questo, corri

```
python -m compileall <file>.py
```

Poi in un interprete, corri

```
import dis
import marshal
with open("<file>.pyc", "rb") as code_f:
    code_f.read(8) # Magic number and modification time
    code = marshal.load(code_f) # Returns a code object which can be disassembled
    dis.dis(code) # Output the disassembly
```

Questo compilerà un modulo Python e produrrà le istruzioni bytecode con `dis` . Il modulo non viene mai importato, quindi è sicuro da utilizzare con codice non affidabile.

Leggi Il modulo dis online: https://riptutorial.com/it/python/topic/1763/il-modulo-dis

Capitolo 79: Il modulo locale

Osservazioni

Python 2 Doc: [https://docs.python.org/2/library/locale.html#locale.currency][1]

Examples

Valuta Formattazione dei dollari USA utilizzando il modulo locale

```
import locale

locale.setlocale(locale.LC_ALL, '')
Out[2]: 'English_United States.1252'

locale.currency(762559748.49)
Out[3]: '$762559748.49'

locale.currency(762559748.49, grouping=True)
Out[4]: '$762,559,748.49'
```

Leggi Il modulo locale online: https://riptutorial.com/it/python/topic/1783/il-modulo-locale

Capitolo 80: Il modulo os

introduzione

Questo modulo fornisce un modo portatile per utilizzare la funzionalità dipendente dal sistema operativo.

Sintassi

- importazione os

Parametri

Parametro	Dettagli
Sentiero	Un percorso per un file. Il separatore del percorso può essere determinato da `os.path.sep` .
Modalità	Il permesso desiderato, in ottale (es. `0700`)

Examples

Crea una directory

```
os.mkdir('newdir')
```

Se è necessario specificare le autorizzazioni, è possibile utilizzare l'argomento della `mode` facoltativo:

```
os.mkdir('newdir', mode=0700)
```

Ottieni la directory corrente

Usa la funzione `os.getcwd()` :

```
print(os.getcwd())
```

Determina il nome del sistema operativo

Il modulo `os` fornisce un'interfaccia per determinare il tipo di sistema operativo su cui è in esecuzione il codice.

```
os.name
```

Questo può restituire uno dei seguenti in Python 3:

- `posix`
- `nt`
- `ce`
- `java`

Informazioni più dettagliate possono essere recuperate da `sys.platform`

Rimuovi una directory

Rimuovi la directory sul `path` :

```
os.rmdir(path)
```

Non dovresti usare `os.remove()` per rimuovere una directory. Quella funzione è per i *file* e usarla nelle directory risulterà in un `OSError`

Segui un link simbolico (POSIX)

A volte è necessario determinare l'obiettivo di un collegamento simbolico. `os.readlink` farà questo:

```
print(os.readlink(path_to_symlink))
```

Modifica le autorizzazioni su un file

```
os.chmod(path, mode)
```

dove la `mode` è il permesso desiderato, in ottale.

makedir - creazione di directory ricorsiva

Data una directory locale con i seguenti contenuti:

```
└── dir1
    ├── subdir1
    └── subdir2
```

Vogliamo creare lo stesso subdir1, subdir2 sotto una nuova directory dir2, che non esiste ancora.

```
import os

os.makedirs("./dir2/subdir1")
os.makedirs("./dir2/subdir2")
```

Esecuzione di questo risultato

```
├── dir1
│   ├── subdir1
│   └── subdir2
└── dir2
    ├── subdir1
    └── subdir2
```

dir2 viene creato solo la prima volta che è necessario, per la creazione di subdir1.

Se avessimo usato **os.mkdir** , avremmo avuto un'eccezione perché dir2 non sarebbe esistito ancora.

```
    os.mkdir("./dir2/subdir1")
OSError: [Errno 2] No such file or directory: './dir2/subdir1'
```

a os.makedirs non piacerà se la directory di destinazione esiste già. Se lo ripetiamo di nuovo:

```
OSError: [Errno 17] File exists: './dir2/subdir1'
```

Tuttavia, questo potrebbe essere risolto facilmente rilevando l'eccezione e controllando che la directory sia stata creata.

```
try:
    os.makedirs("./dir2/subdir1")
except OSError:
    if not os.path.isdir("./dir2/subdir1"):
        raise

try:
    os.makedirs("./dir2/subdir2")
except OSError:
    if not os.path.isdir("./dir2/subdir2"):
        raise
```

Leggi Il modulo os online: https://riptutorial.com/it/python/topic/4127/il-modulo-os

Capitolo 81: Implementazioni Python non ufficiali

Examples

IronPython

Implementazione open source per .NET e Mono scritto in C #, concesso in licenza con Apache License 2.0. Si basa su DLR (Dynamic Language Runtime). Supporta solo la versione 2.7, la versione 3 è attualmente in fase di sviluppo.

Differenze con CPython:

- Stretta integrazione con .NET Framework.
- Le stringhe sono Unicode per impostazione predefinita.
- Non supporta le estensioni per CPython scritto in C.
- Non soffre di Global Interpreter Lock.
- Le prestazioni sono generalmente inferiori, anche se dipendono dai test.

Ciao mondo

```
print "Hello World!"
```

Puoi anche utilizzare le funzioni .NET:

```
import clr
from System import Console
Console.WriteLine("Hello World!")
```

link esterno

- Sito ufficiale
- Repository GitHub

Jython

Implementazione open source per JVM scritta in Java, concessa in licenza con Python Software Foundation License. Supporta solo la versione 2.7, la versione 3 è attualmente in fase di sviluppo.

Differenze con CPython:

- Stretta integrazione con JVM.

- Le stringhe sono Unicode.
- Non supporta le estensioni per CPython scritto in C.
- Non soffre di Global Interpreter Lock.
- Le prestazioni sono generalmente inferiori, anche se dipendono dai test.

Ciao mondo

```
print "Hello World!"
```

Puoi anche utilizzare le funzioni Java:

```
from java.lang import System
System.out.println("Hello World!")
```

link esterno

- Sito ufficiale
- Repository Mercurial

Transcrypt

Transcrypt è uno strumento per precompilare un sottoinsieme abbastanza ampio di Python in Javascript compatto e leggibile. Ha le seguenti caratteristiche:

- Permette la programmazione OO classica con ereditarietà multipla usando la pura sintassi Python, analizzata dal parser nativo di CPython
- Perfetta integrazione con l'universo di librerie JavaScript orientate al web di alta qualità, piuttosto che con quelle di Python orientate al desktop
- Sistema di moduli gerarchico basato su URL che consente la distribuzione dei moduli tramite PyPi
- Semplice relazione tra l'origine Python e il codice JavaScript generato per un facile debug
- Sorgenti di codice multi-livello e annotazione facoltativa del codice di destinazione con riferimenti di origine
- Download compatti, kB anziché MB
- Codice JavaScript ottimizzato, utilizzando la memoizzazione (caching delle chiamate) per aggirare facoltativamente la catena di ricerca del prototipo
- Il sovraccarico dell'operatore può essere attivato e disattivato localmente per facilitare la matematica numerica leggibile

Dimensioni e velocità del codice

L'esperienza ha dimostrato che 650 kB di codice sorgente Python si traducono approssimativamente nella stessa quantità di codice sorgente JavaScript. La velocità corrisponde

alla velocità del JavaScript scritto a mano e può superarla se viene attivata la funzione Memo.

Integrazione con HTML

```
<script src="__javascript__/hello.js"></script>
<h2>Hello demo</h2>

<p>
<div id = "greet">...</div>
<button onclick="hello.solarSystem.greet ()">Click me repeatedly!</button>

<p>
<div id = "explain">...</div>
<button onclick="hello.solarSystem.explain ()">And click me repeatedly too!</button>
```

Integrazione con JavaScript e DOM

```
from itertools import chain

class SolarSystem:
    planets = [list (chain (planet, (index + 1,))) for index, planet in enumerate ((
        ('Mercury', 'hot', 2240),
        ('Venus', 'sulphurous', 6052),
        ('Earth', 'fertile', 6378),
        ('Mars', 'reddish', 3397),
        ('Jupiter', 'stormy', 71492),
        ('Saturn', 'ringed', 60268),
        ('Uranus', 'cold', 25559),
        ('Neptune', 'very cold', 24766)
    ))]

    lines = (
        '{} is a {} planet',
        'The radius of {} is {} km',
        '{} is planet nr. {} counting from the sun'
    )

    def __init__ (self):
        self.lineIndex = 0

    def greet (self):
        self.planet = self.planets [int (Math.random () * len (self.planets))]
        document.getElementById ('greet') .innerHTML = 'Hello {}'.format (self.planet [0])
        self.explain ()

    def explain (self):
        document.getElementById ('explain').innerHTML = (
            self.lines [self.lineIndex] .format (self.planet [0], self.planet [self.lineIndex
+ 1])
        )
        self.lineIndex = (self.lineIndex + 1) % 3
         solarSystem = SolarSystem ()
```

Integrazione con altre librerie JavaScript

Transcrypt può essere utilizzato in combinazione con qualsiasi libreria JavaScript senza misure speciali o sintassi. Nella documentazione sono forniti esempi per ao react.js, riot.js, fabric.js e node.js.

Relazione tra Python e codice JavaScript

Pitone

```
class A:
    def __init__ (self, x):
        self.x = x

    def show (self, label):
        print ('A.show', label, self.x)

class B:
    def __init__ (self, y):
        alert ('In B constructor')
        self.y = y

    def show (self, label):
        print ('B.show', label, self.y)

class C (A, B):
    def __init__ (self, x, y):
        alert ('In C constructor')
        A.__init__ (self, x)
        B.__init__ (self, y)
        self.show ('constructor')

    def show (self, label):
        B.show (self, label)
        print ('C.show', label, self.x, self.y)

a = A (1001)
a.show ('america')

b = B (2002)
b.show ('russia')

c = C (3003, 4004)
c.show ('netherlands')

show2 = c.show
show2 ('copy')
```

JavaScript

```
var A = __class__ ('A', [object], {
    get __init__ () {return __get__ (this, function (self, x) {
        self.x = x;
    });},
```

```
    get show () {return __get__ (this, function (self, label) {
        print ('A.show', label, self.x);
    });}
});
var B = __class__ ('B', [object], {
    get __init__ () {return __get__ (this, function (self, y) {
        alert ('In B constructor');
        self.y = y;
    });},
    get show () {return __get__ (this, function (self, label) {
        print ('B.show', label, self.y);
    });}
});
var C = __class__ ('C', [A, B], {
    get __init__ () {return __get__ (this, function (self, x, y) {
        alert ('In C constructor');
        A.__init__ (self, x);
        B.__init__ (self, y);
        self.show ('constructor');
    });},
    get show () {return __get__ (this, function (self, label) {
        B.show (self, label);
        print ('C.show', label, self.x, self.y);
    });}
});
var a = A (1001);
a.show ('america');
var b = B (2002);
b.show ('russia');
var c = C (3003, 4004);
c.show ('netherlands');
var show2 = c.show;
show2 ('copy');
```

link esterno

- Sito web ufficiale: http://www.transcrypt.org/
- Repository: https://github.com/JdeH/Transcrypt

Leggi Implementazioni Python non ufficiali online:
https://riptutorial.com/it/python/topic/5225/implementazioni-python-non-ufficiali

Capitolo 82: Importazione di moduli

Sintassi

- importa *module_name*
- import *module_name.submodule_name*
- da *module_name* import *
- da *module_name* import *submodule_name* [, *class_name* , *function_name* , ... ecc.]
- da *module_name some_name* importazione come *NEW_NAME*
- da *module_name.submodule_name* importazione *class_name* [, *function_name,* ... etc]

Osservazioni

L'importazione di un modulo farà in modo che Python valuti tutto il codice di primo livello in questo modulo in modo che *impari* tutte le funzioni, le classi e le variabili contenute nel modulo. Quando vuoi che un tuo modulo venga importato da qualche altra parte, fai attenzione con il tuo codice di primo livello e `if __name__ == '__main__':` in `if __name__ == '__main__':` se non vuoi che venga eseguito quando il modulo viene importato.

Examples

Importare un modulo

Usa la dichiarazione di `import` :

```
>>> import random
>>> print(random.randint(1, 10))
4
```

`import module` importa un modulo e quindi ti permette di fare riferimento ai suoi oggetti - valori, funzioni e classi, per esempio - usando la sintassi `module.name` . Nell'esempio precedente, viene importato il modulo `random` , che contiene la funzione `randint` . Quindi importando `random` puoi chiamare `randint` con `random.randint` .

Puoi importare un modulo e assegnarlo a un nome diverso:

```
>>> import random as rn
>>> print(rn.randint(1, 10))
4
```

Se il file python `main.py` trova nella stessa cartella di `custom.py` . Puoi importarlo in questo modo:

```
import custom
```

È anche possibile importare una funzione da un modulo:

```
>>> from math import sin
>>> sin(1)
0.8414709848078965
```

Per importare funzioni specifiche più in profondità in un modulo, l'operatore punto può essere utilizzato **solo** sul lato sinistro della parola chiave di `import` :

```
from urllib.request import urlopen
```

In Python, abbiamo due modi per chiamare la funzione dal livello superiore. Uno è `import` e un altro è `from` . Dovremmo usare l' `import` quando abbiamo la possibilità di una collisione di nomi. Supponiamo di avere file `hello.py` e file `world.py` aventi la stessa funzione chiamata `function` . Quindi la dichiarazione di `import` funzionerà correttamente.

```
from hello import function
from world import function

function() #world's function will be invoked. Not hello's
```

In generale l' `import` ti fornirà uno spazio dei nomi.

```
import hello
import world

hello.function() # exclusively hello's function will be invoked
world.function() # exclusively world's function will be invoked
```

Ma se siete abbastanza sicuri, in tutto il progetto non c'è alcun modo avere lo stesso nome di funzione è necessario utilizzare `from` dichiarazione

Più importazioni possono essere effettuate sulla stessa linea:

```
>>> # Multiple modules
>>> import time, sockets, random
>>> # Multiple functions
>>> from math import sin, cos, tan
>>> # Multiple constants
>>> from math import pi, e

>>> print(pi)
3.141592653589793
>>> print(cos(45))
0.5253219888177297
>>> print(time.time())
1482807222.7240417
```

Le parole chiave e la sintassi mostrate sopra possono essere utilizzate anche in combinazioni:

```
>>> from urllib.request import urlopen as geturl, pathname2url as path2url, getproxies
>>> from math import factorial as fact, gamma, atan as arctan
>>> import random.randint, time, sys

>>> print(time.time())
```

```
1482807222.7240417
>>> print(arctan(60))
1.554131203080956
>>> filepath = "/dogs/jumping poodle (december).png"
>>> print(path2url(filepath))
/dogs/jumping%20poodle%20%28december%29.png
```

Importazione di nomi specifici da un modulo

Invece di importare il modulo completo puoi importare solo i nomi specificati:

```
from random import randint # Syntax "from MODULENAME import NAME1[, NAME2[, ...]]"
print(randint(1, 10))      # Out: 5
```

`from random` è necessario, perché l'interprete python deve sapere da quale risorsa deve importare una funzione o classe e `import randint` specifica la funzione o la classe stessa.

Un altro esempio qui sotto (simile a quello sopra):

```
from math import pi
print(pi)                  # Out: 3.14159265359
```

Il seguente esempio genererà un errore, perché non abbiamo importato un modulo:

```
random.randrange(1, 10)    # works only if "import random" has been run before
```

Uscite:

```
NameError: name 'random' is not defined
```

L'interprete python non capisce cosa intendi con `random` . Deve essere dichiarato aggiungendo l' `import random` all'esempio:

```
import random
random.randrange(1, 10)
```

Importare tutti i nomi da un modulo

```
from module_name import *
```

per esempio:

```
from math import *
sqrt(2)    # instead of math.sqrt(2)
ceil(2.7)  # instead of math.ceil(2.7)
```

Ciò importerà tutti i nomi definiti nel modulo `math` nello spazio dei nomi globale, diversi dai nomi che iniziano con un carattere di sottolineatura (che indica che lo scrittore ritiene che sia solo per

uso interno).

Avvertenza : se una funzione con lo stesso nome è già stata definita o importata, verrà **sovrascritta** . Quasi sempre importando solo nomi specifici `from math import sqrt, ceil` è il **modo consigliato** :

```
def sqrt(num):
    print("I don't know what's the square root of {}.".format(num))

sqrt(4)
# Output: I don't know what's the square root of 4.

from math import *
sqrt(4)
# Output: 2.0
```

Le importazioni speciali sono consentite solo a livello di modulo. I tentativi di eseguirli nelle definizioni di classe o funzione producono un `SyntaxError` .

```
def f():
    from math import *
```

e

```
class A:
    from math import *
```

entrambi falliscono con:

```
SyntaxError: import * only allowed at module level
```

La variabile speciale __all__

I moduli possono avere una variabile speciale chiamata `__all__` per limitare quali variabili vengono importate quando si utilizza `from mymodule import *` .

Dato il seguente modulo:

```
# mymodule.py

__all__ = ['imported_by_star']

imported_by_star = 42
not_imported_by_star = 21
```

Solo `imported_by_star` viene importata quando si utilizza `from mymodule import *` :

```
>>> from mymodule import *
>>> imported_by_star
42
>>> not_imported_by_star
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'not_imported_by_star' is not defined
```

Tuttavia, `not_imported_by_star` può essere importato esplicitamente:

```
>>> from mymodule import not_imported_by_star
>>> not_imported_by_star
21
```

Importazione programmatica

Python 2.x 2.7

Per importare un modulo tramite una chiamata di funzione, usa il modulo `importlib` (incluso in Python che inizia nella versione 2.7):

```
import importlib
random = importlib.import_module("random")
```

La funzione `importlib.import_module()` importerà anche il sottomodulo di un pacchetto direttamente:

```
collections_abc = importlib.import_module("collections.abc")
```

Per le versioni precedenti di Python, usa il modulo `imp` .

Python 2.x 2.7

Utilizzare le funzioni `imp.find_module` e `imp.load_module` per eseguire un'importazione programmatica.

Tratto dalla documentazione della libreria standard

```
import imp, sys
def import_module(name):
    fp, pathname, description = imp.find_module(name)
    try:
        return imp.load_module(name, fp, pathname, description)
    finally:
        if fp:
            fp.close()
```

NON usare `__import__()` per importare moduli in modo programmatico! Ci sono dettagli sottili che coinvolgono `sys.modules` , l'argomento `fromlist` , ecc. Che è facile ignorare quale `importlib.import_module()` gestisce per te.

Importa moduli da una posizione arbitraria del filesystem

Se si desidera importare un modulo che non esiste già come modulo integrato nella libreria

standard Python né come pacchetto laterale, è possibile farlo aggiungendo il percorso alla directory in cui si trova il modulo su `sys.path` . Questo può essere utile laddove esistono più ambienti python su un host.

```
import sys
sys.path.append("/path/to/directory/containing/your/module")
import mymodule
```

È importante aggiungere il percorso alla *directory* in cui è stato trovato `mymodule` , non il percorso del modulo stesso.

Regole PEP8 per le importazioni

Alcune linee guida di stile PEP8 raccomandate per le importazioni:

1. Le importazioni dovrebbero essere su linee separate:

   ```
   from math import sqrt, ceil      # Not recommended
   from math import sqrt            # Recommended
   from math import ceil
   ```

2. Ordinare le importazioni come segue nella parte superiore del modulo:

   - Importazioni di librerie standard
   - Importazioni di terze parti correlate
   - Importazioni specifiche di applicazioni / librerie locali

3. Le importazioni con caratteri jolly devono essere evitate poiché causano confusione nei nomi nello spazio dei nomi corrente. Se si esegue l' `from module import *` , non è chiaro se un nome specifico nel codice proviene dal `module` o meno. Questo è doppiamente vero se si dispone di più istruzioni `from module import *` -type `from module import *` .

4. Evitare l'uso di importazioni relative; utilizzare invece le importazioni esplicite.

Importazione di sottomoduli

```
from module.submodule import function
```

Questa `function` importa da `module.submodule` .

__import __ () funzione

La funzione `__import__()` può essere utilizzata per importare moduli in cui il nome è noto solo in fase di esecuzione

```
if user_input == "os":
    os = __import__("os")

# equivalent to import os
```

Questa funzione può anche essere utilizzata per specificare il percorso del file per un modulo

```
mod = __import__(r"C:/path/to/file/anywhere/on/computer/module.py")
```

Reimportazione di un modulo

Quando si utilizza l'interprete interattivo, è possibile ricaricare un modulo. Questo può essere utile se stai modificando un modulo e vuoi importare la versione più recente, o se hai patch-patch un elemento di un modulo esistente e vuoi annullare le modifiche.

Nota che non **puoi** semplicemente `import` nuovo il modulo per annullare:

```
import math
math.pi = 3
print(math.pi)    # 3
import math
print(math.pi)    # 3
```

Questo perché l'interprete registra ogni modulo che importi. E quando provi a reimportare un modulo, l'interprete lo vede nel registro e non fa nulla. Quindi il modo più difficile per reimportare è utilizzare l' `import` dopo aver rimosso l'elemento corrispondente dal registro:

```
print(math.pi)    # 3
import sys
if 'math' in sys.modules:  # Is the ``math`` module in the register?
    del sys.modules['math']  # If so, remove it.
import math
print(math.pi)    # 3.141592653589793
```

Ma c'è di più un modo semplice e diretto.

Python 2

Usa la funzione di `reload` :

Python 2.x 2.3

```
import math
math.pi = 3
print(math.pi)    # 3
reload(math)
print(math.pi)    # 3.141592653589793
```

Python 3

La funzione di `reload` è stata spostata in `importlib` :

Python 3.x 3.0

```
import math
math.pi = 3
print(math.pi)    # 3
from importlib import reload
reload(math)
print(math.pi)    # 3.141592653589793
```

Leggi Importazione di moduli online: https://riptutorial.com/it/python/topic/249/importazione-di-moduli

Capitolo 83: Impostato

Sintassi

- empty_set = set () # inizializza un set vuoto
- literal_set = {'foo', 'bar', 'baz'} # costruisce un set con 3 stringhe al suo interno
- set_from_list = set (['foo', 'bar', 'baz']) # chiama la funzione set per un nuovo set
- set_from_iter = set (x per x in range (30)) # usa iterables arbitrari per creare un set
- set_from_iter = {x per x in [random.randint (0,10) per i in range (10)]} # notazione alternativa

Osservazioni

Gli insiemi sono *non ordinati* e hanno *un tempo di ricerca molto veloce* (ammortizzato O (1) se si vuole ottenere tecnico). È fantastico da usare quando si ha una collezione di cose, l'ordine non ha importanza, e cercherete articoli per nome molto. Se ha più senso cercare gli articoli per un numero di indice, prendere in considerazione l'uso di un elenco. Se l'ordine è importante, considera anche un elenco.

Gli insiemi sono *mutabili* e quindi non possono essere sottoposti a hash, quindi non puoi usarli come chiavi del dizionario o metterli in altri set, o in qualsiasi altro posto che richiede tipi lavabili. In questi casi, puoi usare un `frozenset` immutabile.

Gli elementi di un set devono essere *lavabili* . Ciò significa che hanno un metodo `__hash__` corretto, che è coerente con `__eq__` . In generale, i tipi mutabili come la `list` o il `set` non sono lavabili e non possono essere messi in un set. Se si verifica questo problema, considerare l'utilizzo di chiavi `dict` e immutabili.

Examples

Ottieni gli elementi unici di una lista

Supponiamo che tu abbia un elenco di ristoranti, forse lo leggi da un file. Ti interessano i ristoranti *unici* nella lista. Il modo migliore per ottenere gli elementi unici da un elenco è trasformarlo in un set:

```
restaurants = ["McDonald's", "Burger King", "McDonald's", "Chicken Chicken"]
unique_restaurants = set(restaurants)
print(unique_restaurants)
# prints {'Chicken Chicken', "McDonald's", 'Burger King'}
```

Si noti che il set non è nello stesso ordine della lista originale; questo perché gli insiemi sono *non ordinati* , proprio come `dict` s.

Questo può essere facilmente trasformato in una `List` con la funzione di `list` incorporata di Python, dando un altro elenco che è lo stesso elenco dell'originale ma senza duplicati:

```
list(unique_restaurants)
# ['Chicken Chicken', "McDonald's", 'Burger King']
```

È anche comune vederlo come una riga:

```
# Removes all duplicates and returns another list
list(set(restaurants))
```

Ora qualsiasi operazione che potrebbe essere eseguita nell'elenco originale può essere eseguita di nuovo.

Operazioni sui set

con altri set

```
# Intersection
{1, 2, 3, 4, 5}.intersection({3, 4, 5, 6})  # {3, 4, 5}
{1, 2, 3, 4, 5} & {3, 4, 5, 6}              # {3, 4, 5}

# Union
{1, 2, 3, 4, 5}.union({3, 4, 5, 6})  # {1, 2, 3, 4, 5, 6}
{1, 2, 3, 4, 5} | {3, 4, 5, 6}       # {1, 2, 3, 4, 5, 6}

# Difference
{1, 2, 3, 4}.difference({2, 3, 5})  # {1, 4}
{1, 2, 3, 4} - {2, 3, 5}            # {1, 4}

# Symmetric difference with
{1, 2, 3, 4}.symmetric_difference({2, 3, 5})  # {1, 4, 5}
{1, 2, 3, 4} ^ {2, 3, 5}                      # {1, 4, 5}

# Superset check
{1, 2}.issuperset({1, 2, 3})  # False
{1, 2} >= {1, 2, 3}           # False

# Subset check
{1, 2}.issubset({1, 2, 3})  # True
{1, 2} <= {1, 2, 3}         # True

# Disjoint check
{1, 2}.isdisjoint({3, 4})  # True
{1, 2}.isdisjoint({1, 4})  # False
```

con singoli elementi

```
# Existence check
2 in {1,2,3}      # True
4 in {1,2,3}      # False
4 not in {1,2,3}  # True

# Add and Remove
s = {1,2,3}
s.add(4)        # s == {1,2,3,4}

s.discard(3)    # s == {1,2,4}
s.discard(5)    # s == {1,2,4}
```

```
s.remove(2)     # s == {1,4}
s.remove(2)     # KeyError!
```

Imposta le operazioni restituiscono nuovi set, ma hanno le corrispondenti versioni sul posto:

metodo	operazione sul posto	metodo sul posto
unione	s | = t	aggiornare
intersezione	s & = t	intersection_update
differenza	s = = t	difference_update
symmetric_difference	s ^ = t	symmetric_difference_update

Per esempio:

```
s = {1, 2}
s.update({3, 4})   # s == {1, 2, 3, 4}
```

Imposta contro multiset

Gli insiemi sono raccolte non ordinate di elementi distinti. Ma a volte vogliamo lavorare con collezioni non ordinate di elementi che non sono necessariamente distinti e tenere traccia delle molteplicità degli elementi.

Considera questo esempio:

```
>>> setA = {'a','b','b','c'}
>>> setA
set(['a', 'c', 'b'])
```

Salvando le stringhe `'a'` , `'b'` , `'b'` , `'c'` in una struttura dati impostata abbiamo perso le informazioni sul fatto che `'b'` verifica due volte. Naturalmente il salvataggio degli elementi in una lista manterrebbe queste informazioni

```
>>> listA = ['a','b','b','c']
>>> listA
['a', 'b', 'b', 'c']
```

ma una struttura di dati di lista introduce un ordine extra non necessario che rallenterà i nostri calcoli.

Per implementare i multiset Python fornisce la classe `Counter` dal modulo `collections` (a partire dalla versione 2.7):

Python 2.x 2.7

```
>>> from collections import Counter
>>> counterA = Counter(['a','b','b','c'])
>>> counterA
Counter({'b': 2, 'a': 1, 'c': 1})
```

`Counter` è un dizionario in cui gli elementi vengono memorizzati come chiavi del dizionario e i loro conteggi sono memorizzati come valori del dizionario. E come tutti i dizionari, è una collezione non ordinata.

Imposta le operazioni usando Methods e Builtins

Definiamo due serie `a` e `b`

```
>>> a = {1, 2, 2, 3, 4}
>>> b = {3, 3, 4, 4, 5}
```

> NOTA: `{1}` crea un set di un elemento, ma `{}` crea un `dict` vuoto. Il modo corretto per creare un set vuoto è `set()` .

Intersezione

`a.intersection(b)` restituisce un nuovo set con elementi presenti sia in `a` che in `b`

```
>>> a.intersection(b)
{3, 4}
```

Unione

`a.union(b)` restituisce un nuovo set con elementi presenti in `a` e `b`

```
>>> a.union(b)
{1, 2, 3, 4, 5}
```

Differenza

`a.difference(b)` restituisce un nuovo set con elementi presenti in `a` ma non in `b`

```
>>> a.difference(b)
{1, 2}
>>> b.difference(a)
{5}
```

Differenza simmetrica

`a.symmetric_difference(b)` restituisce un nuovo set con elementi presenti in `a` o `b` ma non in entrambi

```
>>> a.symmetric_difference(b)
{1, 2, 5}
>>> b.symmetric_difference(a)
{1, 2, 5}
```

NOTA : `a.symmetric_difference(b) == b.symmetric_difference(a)`

Sottoinsieme e superset

`c.issubset(a)` verifica se ogni elemento di `c` è in `a` .

`a.issuperset(c)` verifica se ogni elemento di `c` è in `a` .

```
>>> c = {1, 2}
>>> c.issubset(a)
True
>>> a.issuperset(c)
True
```

Queste ultime operazioni hanno operatori equivalenti come mostrato di seguito:

Metodo	Operatore
`a.intersection(b)`	`a & b`
`a.union(b)`	`a | b`
`a.difference(b)`	`a - b`
`a.symmetric_difference(b)`	`a ^ b`
`a.issubset(b)`	`a <= b`
`a.issuperset(b)`	`a >= b`

Set disgiunti

Imposta `a` e `d` sono disgiunti se nessun elemento in `a` è anche in `d` e viceversa.

```
>>> d = {5, 6}
>>> a.isdisjoint(b) # {2, 3, 4} are in both sets
False
>>> a.isdisjoint(d)
True

# This is an equivalent check, but less efficient
```

```
>>> len(a & d) == 0
True

# This is even less efficient
>>> a & d == set()
True
```

Test dell'appartenenza

Il built-in `in` ricerche di parole chiave per occorrenze

```
>>> 1 in a
True
>>> 6 in a
False
```

Lunghezza

La funzione built-in `len()` restituisce il numero di elementi nel set

```
>>> len(a)
4
>>> len(b)
3
```

Set di Set

```
{{1,2}, {3,4}}
```

porta a:

```
TypeError: unhashable type: 'set'
```

Invece, usa `frozenset` :

```
{frozenset({1, 2}), frozenset({3, 4})}
```

Leggi Impostato online: https://riptutorial.com/it/python/topic/497/impostato

Capitolo 84: Incompatibilità che si spostano da Python 2 a Python 3

introduzione

A differenza della maggior parte delle lingue, Python supporta due versioni principali. Dal 2008, quando è stato rilasciato Python 3, molti hanno fatto la transizione, mentre molti non l'hanno fatto. Per comprendere entrambi, questa sezione copre le importanti differenze tra Python 2 e Python 3.

Osservazioni

Ci sono attualmente due versioni supportate di Python: 2.7 (Python 2) e 3.6 (Python 3). Inoltre, le versioni 3.3 e 3.4 ricevono aggiornamenti di sicurezza in formato sorgente.

Python 2.7 è retrocompatibile con la maggior parte delle versioni precedenti di Python e può eseguire il codice Python dalla maggior parte delle versioni 1.xe 2.x di Python invariato. È ampiamente disponibile, con una vasta collezione di pacchetti. Viene anche considerato deprecato dagli sviluppatori CPython e riceve solo sicurezza e sviluppo di correzioni di errori. Gli sviluppatori CPython intendono abbandonare questa versione della lingua nel 2020 .

Secondo Python Enhancement Proposal 373 non sono previste versioni future di Python 2 dopo il 25 giugno 2016, ma correzioni di bug e aggiornamenti di sicurezza saranno supportati fino al 2020. (Non specifica quale data esatta nel 2020 sarà la data di tramonto di Python 2.)

Python 3 ha intenzionalmente rotto la compatibilità all'indietro, per affrontare le preoccupazioni che gli sviluppatori linguistici avevano del nucleo della lingua. Python 3 riceve nuovo sviluppo e nuove funzionalità. È la versione della lingua che gli sviluppatori di lingue intendono portare avanti.

Nel tempo intercorso tra la versione iniziale di Python 3.0 e la versione corrente, alcune funzionalità di Python 3 sono state sottoposte a back-port in Python 2.6 e altre parti di Python 3 sono state estese per avere la sintassi compatibile con Python 2. Pertanto è possibile scrivere Python che funzionerà sia su Python 2 che su Python 3, usando future importazioni e moduli speciali (come **sei**).

Le importazioni future devono essere all'inizio del modulo:

```
from __future__ import print_function
# other imports and instructions go after __future__
print('Hello world')
```

Per ulteriori informazioni sul modulo `__future__` , consultare la pagina pertinente nella documentazione di Python .

Lo strumento 2to3 è un programma Python che converte il codice Python 2.x in codice Python 3.x, vedi anche la documentazione di Python .

Il pacchetto six fornisce utility per la compatibilità con Python 2/3:

- accesso unificato alle librerie rinominate
- variabili per tipi string / unicode
- funzioni per il metodo rimosso o rinominato

Un riferimento per le differenze tra Python 2 e Python 3 può essere trovato qui .

Examples

Stampa dichiarazione vs. funzione di stampa

In Python 2, la `print` è una dichiarazione:

Python 2.x 2.7

```
print "Hello World"
print                           # print a newline
print "No newline",             # add trailing comma to remove newline
print >>sys.stderr, "Error"     # print to stderr
print("hello")                  # print "hello", since ("hello") == "hello"
print()                         # print an empty tuple "()"
print 1, 2, 3                   # print space-separated arguments: "1 2 3"
print(1, 2, 3)                  # print tuple "(1, 2, 3)"
```

In Python 3, `print()` è una funzione, con argomenti parola chiave per usi comuni:

Python 3.x 3.0

```
print "Hello World"              # SyntaxError
print("Hello World")
print()                         # print a newline (must use parentheses)
print("No newline", end="")      # end specifies what to append (defaults to newline)
print("Error", file=sys.stderr)  # file specifies the output buffer
print("Comma", "separated", "output", sep=",")  # sep specifies the separator
print("A", "B", "C", sep="")     # null string for sep: prints as ABC
print("Flush this", flush=True)  # flush the output buffer, added in Python 3.3
print(1, 2, 3)                   # print space-separated arguments: "1 2 3"
print((1, 2, 3))                 # print tuple "(1, 2, 3)"
```

La funzione di stampa ha i seguenti parametri:

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

`sep` è ciò che separa gli oggetti che passi per stampare. Per esempio:

```
print('foo', 'bar', sep='~') # out: foo~bar
print('foo', 'bar', sep='.') # out: foo.bar
```

`end` è ciò a cui è seguita la fine della dichiarazione di stampa. Per esempio:

```
print('foo', 'bar', end='!') # out: foo bar!
```

La stampa successiva a una stampa di fine non nuova riga *verrà* stampata sulla stessa riga:

```
print('foo', end='~')
print('bar')
# out: foo~bar
```

Nota: per compatibilità futura, la *funzione di* `print` è disponibile anche in Python 2.6; tuttavia non può essere utilizzato a meno che l'analisi dell'istruzione di `print` *non* sia disabilitata con

```
from __future__ import print_function
```

Questa funzione ha esattamente lo stesso formato di Python 3, tranne per il fatto che manca il parametro `flush` .

Vedere PEP 3105 per la logica.

Stringhe: byte contro Unicode

Python 2.x 2.7

In Python 2 ci sono due varianti di stringa: quelle fatte di byte con tipo (`str`) e quelle fatte di testo con tipo (`unicode`).

In Python 2, un oggetto di tipo `str` è sempre una sequenza di byte, ma è comunemente usato sia per i dati di testo che per quelli binari.

Una stringa letterale viene interpretata come una stringa di byte.

```
s = 'Cafe'    # type(s) == str
```

Ci sono due eccezioni: puoi definire un *letterale Unicode (testo)* esplicitamente prefiggendo il letterale con `u` :

```
s = u'Café'   # type(s) == unicode
b = 'Lorem ipsum'  # type(b) == str
```

In alternativa, è possibile specificare che i valori letterali stringa di un intero modulo devono creare valori letterali Unicode (testo):

```
from __future__ import unicode_literals

s = 'Café'   # type(s) == unicode
b = 'Lorem ipsum'  # type(b) == unicode
```

Per verificare se la variabile è una stringa (Unicode o una stringa di byte), puoi utilizzare:

```
isinstance(s, basestring)
```

Python 3.x 3.0

In Python 3, il tipo `str` è un tipo di testo Unicode.

```
s = 'Cafe'           # type(s) == str
s = 'Café'           # type(s) == str (note the accented trailing e)
```

Inoltre, Python 3 ha aggiunto un oggetto `bytes` , adatto per "blob" binari o per scrivere in file indipendenti dalla codifica. Per creare un oggetto bytes, puoi anteporre `b` a un letterale stringa o chiamare il metodo `encode` della stringa:

```
# Or, if you really need a byte string:
s = b'Cafe'          # type(s) == bytes
s = 'Café'.encode()  # type(s) == bytes
```

Per verificare se un valore è una stringa, utilizzare:

```
isinstance(s, str)
```

Python 3.x 3.3

È anche possibile prefissare i letterali stringa con un prefisso `u` per facilitare la compatibilità tra le basi di codice Python 2 e Python 3. Poiché, in Python 3, tutte le stringhe sono Unicode per impostazione predefinita, la preposizione di una stringa letterale con `u` non ha alcun effetto:

```
u'Cafe' == 'Cafe'
```

Il prefisso di stringa Unicode non `ur` Python 2 non è supportato, tuttavia:

```
>>> ur'Café'
  File "<stdin>", line 1
    ur'Café'
           ^
SyntaxError: invalid syntax
```

Nota che devi `encode` un oggetto testo (`str`) Python 3 per convertirlo in una rappresentazione in `bytes` di quel testo. La codifica predefinita di questo metodo è UTF-8 .

Puoi utilizzare la `decode` per chiedere a un oggetto `bytes` quale testo Unicode rappresenta:

```
>>> b.decode()
'Café'
```

Python 2.x 2.6

Mentre il tipo di `bytes` esiste sia in Python 2 che in 3, il tipo `unicode` esiste solo in Python 2. Per usare le stringhe Unicode implicite di Python 3 in Python 2, aggiungi quanto segue all'inizio del tuo file di codice:

```
from __future__ import unicode_literals
print(repr("hi"))
```

```
# u'hi'
```

Python 3.x 3.0

Un'altra importante differenza è che l'indicizzazione dei byte in Python 3 ha come risultato un output `int` simile:

```
b"abc"[0] == 97
```

Mentre l'affettatura in una dimensione di un risultato in un oggetto di lunghezza 1 byte:

```
b"abc"[0:1] == b"a"
```

Inoltre, Python 3 corregge alcuni comportamenti insoliti con unicode, ovvero invertendo le stringhe di byte in Python 2. Ad esempio, il seguente problema è stato risolto:

```
# -*- coding: utf8 -*-
print("Hi, my name is Łukasz Langa.")
print(u"Hi, my name is Łukasz Langa."[::-1])
print("Hi, my name is Łukasz Langa."[::-1])

# Output in Python 2
# Hi, my name is Łukasz Langa.
# .agnaL zsakuŁ si eman ym ,iH
# .agnaL zsaku◆◆ si eman ym ,iH

# Output in Python 3
# Hi, my name is Łukasz Langa.
# .agnaL zsakuŁ si eman ym ,iH
# .agnaL zsakuŁ si eman ym ,iH
```

Divisione intera

Il **simbolo di divisione** standard (`/`) opera in modo diverso in Python 3 e Python 2 quando applicato a numeri interi.

Quando si divide un intero di un altro intero in Python 3, l'operazione di divisione `x / y` rappresenta una **divisione vera** (utilizza il metodo `__truediv__`) e produce un risultato in virgola mobile. Nel frattempo, la stessa operazione in Python 2 rappresenta una **divisione classica** che arrotonda il risultato verso l'infinito negativo (noto anche come prendere la *parola*).

Per esempio:

Codice	Uscita Python 2	Uscita Python 3
`3 / 2`	1	1.5
`2 / 3`	0	,6666666666666666
`-3 / 2`	-2	-1.5

Il comportamento di arrotondamento verso lo zero era deprecato in Python 2.2 , ma rimane in Python 2.7 per motivi di compatibilità con le versioni precedenti ed è stato rimosso in Python 3.

Nota: per ottenere un risultato *float* in Python 2 (senza arrotondamento del pavimento) possiamo specificare uno degli operandi con il punto decimale. L'esempio precedente di `2/3` che dà `0` in Python 2 deve essere usato come `2 / 3.0` o `2.0 / 3` o `2.0/3.0` per ottenere `0.6666666666666666`

Codice	Uscita Python 2	Uscita Python 3
`3.0 / 2.0`	1.5	1.5
`2 / 3.0`	,6666666666666666	,6666666666666666
`-3.0 / 2`	-1.5	-1.5

Esiste anche l' **operatore della divisione floor** (`//`), che funziona allo stesso modo in entrambe le versioni: si arrotonda al numero intero più vicino. (anche se un float viene restituito quando viene utilizzato con float) In entrambe le versioni l'operatore `//` associa a `__floordiv__` .

Codice	Uscita Python 2	Uscita Python 3
`3 // 2`	1	1
`2 // 3`	0	0
`-3 // 2`	-2	-2
`3.0 // 2.0`	1.0	1.0
`2.0 // 3`	0.0	0.0
`-3 // 2.0`	-2.0	-2.0

Si può forzare esplicitamente la vera divisione o divisione del piano usando le funzioni native nel modulo `operator` :

```
from operator import truediv, floordiv
assert truediv(10, 8) == 1.25            # equivalent to `/` in Python 3
assert floordiv(10, 8) == 1              # equivalent to `//`
```

Mentre è chiaro ed esplicito, usare le funzioni dell'operatore per ogni divisione può essere noioso. Spesso si preferisce modificare il comportamento dell'operatore `/` . Una pratica comune consiste nell'eliminare il tipico comportamento di divisione aggiungendo `from __future__ import division` come prima affermazione in ogni modulo:

```
# needs to be the first statement in a module
from __future__ import division
```

Codice	Uscita Python 2	Uscita Python 3
`3 / 2`	1.5	1.5
`2 / 3`	,6666666666666666	,6666666666666666
`-3 / 2`	-1.5	-1.5

`from __future__ import division` garantisce che l'operatore `/` rappresenta la vera divisione e solo all'interno dei moduli che contengono l'importazione `__future__` , quindi non ci sono motivi validi per non abilitarlo in tutti i nuovi moduli.

Nota : alcuni altri linguaggi di programmazione utilizzano l' *arrotondamento verso lo zero* (troncamento) anziché l' *arrotondamento verso il basso negativo* come fa Python (cioè in quei linguaggi `-3 / 2 == -1`). Questo comportamento può creare confusione durante il porting o il confronto del codice.

Nota sugli operandi float : in alternativa alla `from __future__ import division` , si potrebbe usare il consueto simbolo di divisione `/` e assicurarsi che almeno uno degli operandi sia float: `3 / 2.0 == 1.5` . Tuttavia, questo può essere considerato una cattiva pratica. È troppo facile scrivere `average = sum(items) / len(items)` e dimenticare di lanciare uno degli argomenti per renderlo mobile. Inoltre, tali casi possono spesso eludere la notifica durante il test, ad esempio, se si prova su un array contenente `float` s ma si riceve una matrice di `int` s in produzione. Inoltre, se in Python 3 viene utilizzato lo stesso codice, i programmi che prevedono che 3/2 `3 / 2 == 1` siano True non funzioneranno correttamente.

Vedi PEP 238 per ragioni più dettagliate sul perché l'operatore di divisione è stato modificato in Python 3 e perché la divisione vecchio stile dovrebbe essere evitata.

Vedi l' argomento *Simple Math* per ulteriori informazioni sulla divisione.

Ridurre non è più un built-in

In Python 2, `reduce` è disponibile sia come funzione integrata o dal pacchetto `functools` (versione 2.6 in poi), mentre in Python 3 la `reduce` è disponibile solo da `functools` . Tuttavia la sintassi per la `reduce` sia in Python2 che in Python3 è la stessa ed è `reduce(function_to_reduce, list_to_reduce)` .

Ad esempio, consideriamo la possibilità di ridurre un elenco a un singolo valore dividendo ciascuno dei numeri adiacenti. Qui usiamo la funzione `truediv` dalla libreria `operator` .

In Python 2.x è semplice come:

Python 2.x 2.3

```
>>> my_list = [1, 2, 3, 4, 5]
>>> import operator
>>> reduce(operator.truediv, my_list)
0.008333333333333333
```

In Python 3.x l'esempio diventa un po 'più complicato:

Python 3.x 3.0

```
>>> my_list = [1, 2, 3, 4, 5]
>>> import operator, functools
>>> functools.reduce(operator.truediv, my_list)
0.008333333333333333
```

Possiamo anche usare `from functools import reduce` per evitare di chiamare `reduce` con il nome del namespace.

Differenze tra le funzioni range e xrange

In Python 2, la funzione `range` restituisce un elenco mentre `xrange` crea un oggetto `xrange` speciale, che è una sequenza immutabile, che a differenza di altri tipi di sequenza incorporati, non supporta l'slicing e non ha né `index` né metodi di `count` :

Python 2.x 2.3

```
print(range(1, 10))
# Out: [1, 2, 3, 4, 5, 6, 7, 8, 9]

print(isinstance(range(1, 10), list))
# Out: True

print(xrange(1, 10))
# Out: xrange(1, 10)

print(isinstance(xrange(1, 10), xrange))
# Out: True
```

In Python 3, `xrange` stato espanso alla sequenza di `range` , che ora crea un oggetto `range` . Non esiste un tipo `xrange` :

Python 3.x 3.0

```
print(range(1, 10))
# Out: range(1, 10)

print(isinstance(range(1, 10), range))
# Out: True

# print(xrange(1, 10))
# The output will be:
#Traceback (most recent call last):
#  File "<stdin>", line 1, in <module>
#NameError: name 'xrange' is not defined
```

Inoltre, dal momento che Python 3.2, `range` supporta anche affettare, `index` e `count` :

```
print(range(1, 10)[3:7])
# Out: range(3, 7)
print(range(1, 10).count(5))
```

```
# Out: 1
print(range(1, 10).index(7))
# Out: 6
```

Il vantaggio di usare un tipo di sequenza speciale invece di una lista è che l'interprete non deve allocare memoria per un elenco e popolarlo:

Python 2.x 2.3

```
# range(10000000000000000)
# The output would be:
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# MemoryError

print(xrange(10000000000000000))
# Out: xrange(10000000000000000)
```

Poiché il secondo comportamento è generalmente desiderato, il primo è stato rimosso in Python 3. Se vuoi comunque avere un elenco in Python 3, puoi semplicemente usare il costruttore di `list()` su un oggetto `range` :

Python 3.x 3.0

```
print(list(range(1, 10)))
# Out: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Compatibilità

Per mantenere la compatibilità tra le versioni di Python 2.xe di Python 3.x, è possibile utilizzare il modulo `builtins` dal pacchetto esterno `future` per ottenere sia *compatibilità* `builtins` *compatibilità all'indietro* :

Python 2.x 2.0

```
#forward-compatible
from builtins import range

for i in range(10**8):
    pass
```

Python 3.x 3.0

```
#backward-compatible
from past.builtins import xrange

for i in xrange(10**8):
    pass
```

L' `range` nella `future` libreria supporta slicing, `index` e `count` in tutte le versioni di Python, proprio

come il metodo built-in su Python 3.2+.

Disimballare Iterables

Python 3.x 3.0

In Python 3, puoi decomprimere un iterabile senza conoscere il numero esatto di elementi in esso contenuti e persino avere una variabile che tenga la fine del iterabile. Per questo, si fornisce una variabile che può raccogliere un elenco di valori. Questo viene fatto mettendo un asterisco prima del nome. Ad esempio, decomprimere un `list` :

```
first, second, *tail, last = [1, 2, 3, 4, 5]
print(first)
# Out: 1
print(second)
# Out: 2
print(tail)
# Out: [3, 4]
print(last)
# Out: 5
```

Nota : quando si utilizza la sintassi `*variable` , la `variable` sarà sempre un elenco, anche se il tipo originale non era un elenco. Può contenere zero o più elementi a seconda del numero di elementi nell'elenco originale.

```
first, second, *tail, last = [1, 2, 3, 4]
print(tail)
# Out: [3]

first, second, *tail, last = [1, 2, 3]
print(tail)
# Out: []
print(last)
# Out: 3
```

Allo stesso modo, scompattando un `str` :

```
begin, *tail = "Hello"
print(begin)
# Out: 'H'
print(tail)
# Out: ['e', 'l', 'l', 'o']
```

Esempio di disimballaggio di una `date` ; `_` è usato in questo esempio come variabile throwaway (ci interessa solo il valore `year`):

```
person = ('John', 'Doe', (10, 16, 2016))
*_, (*_, year_of_birth) = person
print(year_of_birth)
# Out: 2016
```

Vale la pena ricordare che, dal momento che `*` mangia un numero variabile di elementi, non è

possibile avere due `*` s *per lo stesso iterabile* in un compito - non saprebbe quanti elementi vanno nel primo disimballaggio, e quanti nel secondo :

```
*head, *tail = [1, 2]
# Out: SyntaxError: two starred expressions in assignment
```

Python 3.x 3.5

Finora abbiamo discusso di disfare i compiti. `*` e `**` sono stati estesi in Python 3.5 . Ora è possibile avere diverse operazioni di spacchettamento in un'unica espressione:

```
{*range(4), 4, *(5, 6, 7)}
# Out: {0, 1, 2, 3, 4, 5, 6, 7}
```

Python 2.x 2.0

È anche possibile decomprimere un argomento iterabile in funzione:

```
iterable = [1, 2, 3, 4, 5]
print(iterable)
# Out: [1, 2, 3, 4, 5]
print(*iterable)
# Out: 1 2 3 4 5
```

Python 3.x 3.5

La decompressione di un dizionario utilizza due stelle adiacenti `**` (PEP 448):

```
tail = {'y': 2, 'z': 3}
{'x': 1, **tail}
 # Out: {'x': 1, 'y': 2, 'z': 3}
```

Ciò consente sia la sovrascrittura dei vecchi valori sia la fusione dei dizionari.

```
dict1 = {'x': 1, 'y': 1}
dict2 = {'y': 2, 'z': 3}
{**dict1, **dict2}
# Out: {'x': 1, 'y': 2, 'z': 3}
```

Python 3.x 3.0

Python 3 ha rimosso la disimballaggio tuple nelle funzioni. Quindi quanto segue non funziona in Python 3

```
# Works in Python 2, but syntax error in Python 3:
map(lambda (x, y): x + y, zip(range(5), range(5)))
# Same is true for non-lambdas:
def example((x, y)):
    pass

# Works in both Python 2 and Python 3:
map(lambda x: x[0] + x[1], zip(range(5), range(5)))
# And non-lambdas, too:
```

```
def working_example(x_y):
    x, y = x_y
    pass
```

Vedi PEP 3113 per informazioni dettagliate.

Sollevamento e gestione delle eccezioni

Questa è la sintassi di Python 2, nota le virgole `,` `raise` e `except` linee:

Python 2.x 2.3

```
try:
    raise IOError, "input/output error"
except IOError, exc:
    print exc
```

In Python 3, la sintassi `,` viene eliminata e sostituita da parentesi e la parola chiave `as` :

```
try:
    raise IOError("input/output error")
except IOError as exc:
    print(exc)
```

Per compatibilità con le versioni precedenti, la sintassi di Python 3 è disponibile anche in Python 2.6, quindi dovrebbe essere utilizzata per tutti i nuovi codici che non devono essere compatibili con le versioni precedenti.

Python 3.x 3.0

Python 3 aggiunge anche il concatenamento delle eccezioni , in cui è possibile segnalare che qualche altra eccezione è stata la *causa* di questa eccezione. Per esempio

```
try:
    file = open('database.db')
except FileNotFoundError as e:
    raise DatabaseError('Cannot open {}') from e
```

L'eccezione sollevata `__cause__` `except` è di tipo `DatabaseError` , ma l'eccezione originale è contrassegnata come l'attributo `__cause__` di tale eccezione. Quando viene visualizzato il traceback, l'eccezione originale verrà visualizzata anche nel traceback:

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
FileNotFoundError

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
DatabaseError('Cannot open database.db')
```

Se lanci un blocco `except` *senza* concatenamento esplicito:

```
try:
    file = open('database.db')
except FileNotFoundError as e:
    raise DatabaseError('Cannot open {}')
```

Il traceback è

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
FileNotFoundError

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
DatabaseError('Cannot open database.db')
```

Python 2.x 2.0

Nessuno dei due è supportato in Python 2.x; l'eccezione originale e il suo traceback saranno persi se viene sollevata un'altra eccezione nel blocco eccetto. Il seguente codice può essere utilizzato per la compatibilità:

```
import sys
import traceback

try:
    funcWithError()
except:
    sys_vers = getattr(sys, 'version_info', (0,))
    if sys_vers < (3, 0):
        traceback.print_exc()
    raise Exception("new exception")
```

Python 3.x 3.3

Per "dimenticare" l'eccezione precedentemente generata, usa `raise from None`

```
try:
    file = open('database.db')
except FileNotFoundError as e:
    raise DatabaseError('Cannot open {}') from None
```

Ora il traceback sarebbe semplicemente

```
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
DatabaseError('Cannot open database.db')
```

Oppure per renderlo compatibile con entrambi i Python 2 e 3 è possibile utilizzare il pacchetto sei in questo modo:

```
import six
try:
    file = open('database.db')
except FileNotFoundError as e:
    six.raise_from(DatabaseError('Cannot open {}'), None)
```

.next () metodo sugli iteratori rinominato

In Python 2, un iteratore può essere attraversato usando un metodo chiamato `next` sull'iteratore stesso:

Python 2.x 2.3

```
g = (i for i in range(0, 3))
g.next()  # Yields 0
g.next()  # Yields 1
g.next()  # Yields 2
```

In Python 3 il metodo `.next` è stato rinominato in `.__next__` , riconoscendo il suo ruolo "magico", quindi chiamando `.next` genererà un `AttributeError` . Il modo corretto per accedere a questa funzionalità sia in Python 2 che in Python 3 è chiamare la *funzione* `next` con l'iteratore come argomento.

Python 3.x 3.0

```
g = (i for i in range(0, 3))
next(g)  # Yields 0
next(g)  # Yields 1
next(g)  # Yields 2
```

Questo codice è portatile tra le versioni dalla 2.6 alla versione corrente.

Confronto tra diversi tipi

Python 2.x 2.3

Oggetti di diverso tipo possono essere confrontati. I risultati sono arbitrari, ma coerenti. Sono ordinati in modo tale che `None` sia inferiore a qualsiasi altra cosa, i tipi numerici sono più piccoli dei tipi non numerici e tutto il resto è ordinato lessicograficamente per tipo. Quindi, un `int` è minore di un `str` e una `tuple` è maggiore di una `list` :

```
[1, 2] > 'foo'
# Out: False
(1, 2) > 'foo'
# Out: True
[1, 2] > (1, 2)
# Out: False
100 < [1, 'x'] < 'xyz' < (1, 'x')
# Out: True
```

Inizialmente era stato fatto in modo da poter ordinare un elenco di tipi misti e raggruppare gli

oggetti per tipo:

```
l = [7, 'x', (1, 2), [5, 6], 5, 8.0, 'y', 1.2, [7, 8], 'z']
sorted(l)
# Out: [1.2, 5, 7, 8.0, [5, 6], [7, 8], 'x', 'y', 'z', (1, 2)]
```

Python 3.x 3.0

Viene sollevata un'eccezione quando si confrontano diversi tipi (non numerici):

```
1 < 1.5
# Out: True

[1, 2] > 'foo'
# TypeError: unorderable types: list() > str()
(1, 2) > 'foo'
# TypeError: unorderable types: tuple() > str()
[1, 2] > (1, 2)
# TypeError: unorderable types: list() > tuple()
```

Per ordinare gli elenchi misti in Python 3 in base ai tipi e per ottenere la compatibilità tra le versioni, è necessario fornire una chiave per la funzione ordinata:

```
>>> list = [1, 'hello', [3, 4], {'python': 2}, 'stackoverflow', 8, {'python': 3}, [5, 6]]
>>> sorted(list, key=str)
# Out: [1, 8, [3, 4], [5, 6], 'hello', 'stackoverflow', {'python': 2}, {'python': 3}]
```

L'utilizzo di `str` come funzione `key` converte temporaneamente ciascun elemento in una stringa solo ai fini del confronto. Quindi vede la rappresentazione della stringa che inizia con `[` , `'` , `{` o `0-9` ed è in grado di ordinare quelli (e tutti i seguenti caratteri).

Input dell'utente

In Python 2, l'input dell'utente è accettato usando la funzione `raw_input` ,

Python 2.x 2.3

```
user_input = raw_input()
```

Mentre in Python 3 l'input dell'utente è accettato usando la funzione di `input` .

Python 3.x 3.0

```
user_input = input()
```

In Python 2, l' `input` funzione accetta input e *interpretarlo.* Anche se questo può essere utile, ha diverse considerazioni sulla sicurezza ed è stato rimosso in Python 3. Per accedere alla stessa funzionalità, è possibile utilizzare `eval(input())` .

Per mantenere uno script portatile tra le due versioni, puoi inserire il codice sotto la parte superiore del tuo script Python:

```
try:
    input = raw_input
except NameError:
    pass
```

Modifiche al metodo dei dizionari

In Python 3, molti dei metodi del dizionario sono molto diversi nel comportamento di Python 2, e molti sono stati rimossi: `has_key` , `iter*` e `view*` sono spariti. Invece di `d.has_key(key)` , che era stato a lungo deprecato, è necessario utilizzare la `key in d` .

In Python 2, le `keys` metodi del dizionario, i `values` e gli `items` restituiscono gli elenchi. In Python 3 restituiscono invece gli oggetti *vista* ; gli oggetti vista non sono iteratori e differiscono da essi in due modi:

- hanno dimensioni (si può usare la funzione `len` su di loro)
- possono essere ripetuti più volte

Inoltre, come con gli iteratori, le modifiche nel dizionario si riflettono negli oggetti vista.

Python 2.7 ha eseguito il backport di questi metodi da Python 3; sono disponibili come `viewkeys` , `viewvalues` e `viewitems` . Per trasformare il codice Python 2 in codice Python 3, i moduli corrispondenti sono:

- `d.keys()` , `d.values()` e `d.items()` di Python 2 dovrebbero essere cambiati in `list(d.keys())` , `list(d.values())` ed `list(d.items())`
- `d.iterkeys()` , `d.itervalues()` e `d.iteritems()` dovrebbero essere modificati in `iter(d.keys())` , o anche meglio `iter(d)` ; `iter(d.values())` e `iter(d.items())` rispettivamente
- e infine il metodo Python 2.7 chiama `d.viewkeys()` , `d.viewvalues()` e `d.viewitems()` può essere sostituito con `d.keys()` , `d.values()` e `d.items()` .

Porting Python 2 codice che *itera* su chiavi, valori o voci del dizionario, mentre la sua mutazione è a volte difficile. Tenere conto:

```
d = {'a': 0, 'b': 1, 'c': 2, '!': 3}
for key in d.keys():
    if key.isalpha():
        del d[key]
```

Il codice sembra funzionare in modo simile in Python 3, ma il metodo `keys` restituisce un oggetto vista, non una lista, e se il dizionario cambia dimensione mentre viene iterato, il codice Python 3 si bloccherà con `RuntimeError: dictionary changed size during iteration` . La soluzione è ovviamente quella di scrivere correttamente `for key in list(d)` .

Allo stesso modo, gli oggetti vista si comportano diversamente dagli iteratori: non si può usare `next()` su di essi, e non si può *riprendere l'* iterazione; sarebbe invece ricominciare; se il codice Python 2 supera il valore restituito da `d.iterkeys()` , `d.itervalues()` o `d.iteritems()` a un metodo che si aspetta un iteratore invece di un *iterabile* , allora dovrebbe essere `iter(d)` , `iter(d.values())`

o `iter(d.items())` in Python 3.

la dichiarazione exec è una funzione in Python 3

In Python 2, `exec` è un'istruzione, con una sintassi speciale: `exec code [in globals[, locals]]`. In Python 3 `exec` è ora una funzione: `exec(code, [, globals[, locals]])` , e la sintassi Python 2 solleverà un `SyntaxError` .

Poiché la `print` stata modificata da una funzione in una funzione, è stata aggiunta un'importazione `__future__` . Tuttavia, non esiste `from __future__ import exec_function` , poiché non è necessario: l'istruzione exec in Python 2 può anche essere utilizzata con una sintassi che assomiglia esattamente alla `exec` funzione `exec` in Python 3. In questo modo è possibile modificare le istruzioni

Python 2.x 2.3

```
exec 'code'
exec 'code' in global_vars
exec 'code' in global_vars, local_vars
```

alle forme

Python 3.x 3.0

```
exec('code')
exec('code', global_vars)
exec('code', global_vars, local_vars)
```

e queste ultime sono garantite per funzionare in modo identico sia in Python 2 che in Python 3.

hasattr bug di funzionalità in Python 2

In Python 2, quando una proprietà `hasattr` un errore, `hasattr` ignorerà questa proprietà, restituendo `False` .

```
class A(object):
    @property
    def get(self):
        raise IOError


class B(object):
    @property
    def get(self):
        return 'get in b'

a = A()
b = B()

print 'a hasattr get: ', hasattr(a, 'get')
# output False in Python 2 (fixed, True in Python 3)
print 'b hasattr get', hasattr(b, 'get')
# output True in Python 2 and Python 3
```

Questo bug è stato risolto in Python3. Quindi se usi Python 2, usa

```
try:
    a.get
except AttributeError:
    print("no get property!")
```

o usa invece `getattr`

```
p = getattr(a, "get", None)
if p is not None:
    print(p)
else:
    print("no get property!")
```

Moduli rinominati

Alcuni moduli nella libreria standard sono stati rinominati:

Vecchio nome	Nuovo nome
_winreg	winreg
ConfigParser	ConfigParser
copy_reg	copyreg
Coda	coda
SocketServer	SocketServer
_markupbase	markupbase
repr	reprlib
test.test_support	test.support
Tkinter	Tkinter
tkFileDialog	tkinter.filedialog
urllib / urllib2	urllib, urllib.parse, urllib.error, urllib.response, urllib.request, urllib.robotparser

Alcuni moduli sono stati addirittura convertiti da file in librerie. Prendi tkinter e urllib dall'alto come esempio.

Compatibilità

Quando si mantiene la compatibilità tra entrambe le versioni di Python 2.xe 3.x, è possibile utilizzare il `future` pacchetto esterno per abilitare l'importazione di pacchetti di libreria standard di livello superiore con nomi Python 3.x nelle versioni di Python 2.x.

Costanti ottali

In Python 2, un letterale ottale potrebbe essere definito come

```
>>> 0755  # only Python 2
```

Per garantire la compatibilità incrociata, utilizzare

```
0o755  # both Python 2 and Python 3
```

Tutte le classi sono "classi di nuovo stile" in Python 3.

In Python `3.x` tutte le classi sono classi di *nuovo stile* ; quando la definizione di una nuova classe python la rende implicitamente ereditata `object` . In quanto tale, specificare l' `object` in una definizione di `class` è completamente facoltativo:

Python 3.x 3.0

```
class X: pass
class Y(object): pass
```

Entrambe queste classi ora contengono `object` nel loro `mro` (ordine di risoluzione dei metodi):

Python 3.x 3.0

```
>>> X.__mro__
(__main__.X, object)

>>> Y.__mro__
(__main__.Y, object)
```

In Python `2.x` classi sono, di default, classi vecchio stile; non ereditano implicitamente `object` . Questo fa sì che la semantica delle classi differisca a seconda se aggiungiamo esplicitamente l' `object` come una `class` base:

Python 2.x 2.3

```
class X: pass
class Y(object): pass
```

In questo caso, se proviamo a stampare il `__mro__` di `Y` , `__mro__` un output simile a quello nel caso Python `3.x` :

Python 2.x 2.3

```
>>> Y.__mro__
(<class '__main__.Y'>, <type 'object'>)
```

Questo accade perché abbiamo esplicitamente fatto ereditare `Y` dall'oggetto quando lo definiamo: `class Y(object): pass` . Per la classe `X` che *non* eredita da un oggetto, l'attributo `__mro__` non esiste, il tentativo di accedervi produce un `AttributeError` .

Per **garantire la compatibilità** tra entrambe le versioni di Python, le classi possono essere definite con `object` come classe base:

```
class mycls(object):
    """I am fully compatible with Python 2/3"""
```

In alternativa, se la variabile `__metaclass__` è impostata su `type` at global scope, tutte le classi definite successivamente in un dato modulo sono implicitamente di nuovo stile senza bisogno di ereditare esplicitamente da `object` :

```
__metaclass__ = type

class mycls:
    """I am also fully compatible with Python 2/3"""
```

Operatori rimossi <> e ``, sinonimi di! = E repr ()

In Python 2, `<>` è un sinonimo di `!=` ; allo stesso modo, `` `foo` `` è un sinonimo di `repr(foo)` .

Python 2.x 2.7

```
>>> 1 <> 2
True
>>> 1 <> 1
False
>>> foo = 'hello world'
>>> repr(foo)
"'hello world'"
>>> `foo`
"'hello world'"
```

Python 3.x 3.0

```
>>> 1 <> 2
  File "<stdin>", line 1
    1 <> 2
       ^
SyntaxError: invalid syntax
>>> `foo`
  File "<stdin>", line 1
    `foo`
    ^
SyntaxError: invalid syntax
```

codifica / decodifica in esadecimale non più disponibile

Python 2.x 2.7

```
"1deadbeef3".decode('hex')
# Out: '\x1d\xea\xdb\xee\xf3'
'\x1d\xea\xdb\xee\xf3'.encode('hex')
# Out: 1deadbeef3
```

Python 3.x 3.0

```
"1deadbeef3".decode('hex')
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# AttributeError: 'str' object has no attribute 'decode'

b"1deadbeef3".decode('hex')
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# LookupError: 'hex' is not a text encoding; use codecs.decode() to handle arbitrary codecs

'\x1d\xea\xdb\xee\xf3'.encode('hex')
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# LookupError: 'hex' is not a text encoding; use codecs.encode() to handle arbitrary codecs

b'\x1d\xea\xdb\xee\xf3'.encode('hex')
# Traceback (most recent call last):
#  File "<stdin>", line 1, in <module>
# AttributeError: 'bytes' object has no attribute 'encode'
```

Tuttavia, come suggerito dal messaggio di errore, è possibile utilizzare il modulo `codecs` per ottenere lo stesso risultato:

```
import codecs
codecs.decode('1deadbeef4', 'hex')
# Out: b'\x1d\xea\xdb\xee\xf4'
codecs.encode(b'\x1d\xea\xdb\xee\xf4', 'hex')
# Out: b'1deadbeef4'
```

Si noti che `codecs.encode` restituisce un oggetto `bytes` . Per ottenere un oggetto `str` basta `decode` in ASCII:

```
codecs.encode(b'\x1d\xea\xdb\xee\xff', 'hex').decode('ascii')
# Out: '1deadbeeff'
```

funzione cmp rimossa in Python 3

In Python 3 la funzione integrata di `cmp` stata rimossa, insieme al metodo speciale `__cmp__` .

Dalla documentazione:

> La funzione `cmp()` dovrebbe essere considerata come andata, e il metodo speciale `__cmp__()` non è più supportato. Utilizzare `__lt__()` per l'ordinamento, `__eq__()` con `__hash__()` e altri confronti ricchi secondo necessità. (Se hai davvero bisogno della funzionalità `cmp()` , potresti usare l'espressione `(a > b) - (a < b)` come equivalente per

`cmp(a, b)` .)

Inoltre tutte le funzioni built-in che hanno accettato il parametro `cmp` ora accettano solo il parametro `key` keyword only.

Nel modulo `functools` è anche utile la funzione `cmp_to_key(func)` che consente di convertire da una funzione stile `cmp` a una funzione stile- `key` :

> Trasforma una funzione di confronto vecchio stile in una funzione chiave. Utilizzato con strumenti che accettano funzioni chiave (come `sorted()` , `min()` , `max()` , `heapq.nlargest()` , `heapq.nsmallest()` , `itertools.groupby()`). Questa funzione viene principalmente utilizzata come strumento di transizione per i programmi in fase di conversione da Python 2 che supporta l'utilizzo di funzioni di confronto.

Variabili trapelate nella comprensione delle liste

Python 2.x 2.3

```
x = 'hello world!'
vowels = [x for x in 'AEIOU']

print (vowels)
# Out: ['A', 'E', 'I', 'O', 'U']
print(x)
# Out: 'U'
```

Python 3.x 3.0

```
x = 'hello world!'
vowels = [x for x in 'AEIOU']

print (vowels)
# Out: ['A', 'E', 'I', 'O', 'U']
print(x)
# Out: 'hello world!'
```

Come si può vedere dall'esempio, in Python 2 è stato trapelato il valore di `x` : è mascherato `hello world!` e stampato `U` , poiché questo era l'ultimo valore di `x` quando il ciclo terminava.

Tuttavia, in Python 3 `x` stampato il `hello world!` originariamente definito `hello world!` , poiché la variabile locale dalla comprensione della lista non maschera le variabili dall'ambito circostante.

Inoltre, né le espressioni del generatore (disponibili in Python dal 2.5) né le definizioni del dizionario o dell'insieme (che sono state trasferite a Python 2.7 da Python 3) perdono variabili in Python 2.

Nota che sia in Python 2 che in Python 3, le variabili penetreranno nell'ambiente circostante quando si utilizza un ciclo for:

```
x = 'hello world!'
vowels = []
for x in 'AEIOU':
```

```
    vowels.append(x)
print(x)
# Out: 'U'
```

carta geografica()

`map()` è un builtin utile per applicare una funzione agli elementi di un iterabile. In Python 2, la `map` restituisce una lista. In Python 3, `map` restituisce un *oggetto map* , che è un generatore.

```
# Python 2.X
>>> map(str, [1, 2, 3, 4, 5])
['1', '2', '3', '4', '5']
>>> type(_)
>>> <class 'list'>

# Python 3.X
>>> map(str, [1, 2, 3, 4, 5])
<map object at 0x*>
>>> type(_)
<class 'map'>

# We need to apply map again because we "consumed" the previous map....
>>> map(str, [1, 2, 3, 4, 5])
>>> list(_)
['1', '2', '3', '4', '5']
```

In Python 2, è possibile passare `None` per fungere da funzione di identità. Questo non funziona più in Python 3.

Python 2.x 2.3

```
>>> map(None, [0, 1, 2, 3, 0, 4])
[0, 1, 2, 3, 0, 4]
```

Python 3.x 3.0

```
>>> list(map(None, [0, 1, 2, 3, 0, 5]))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'NoneType' object is not callable
```

Inoltre, quando si passa più di un iterable come argomento in Python 2, `map` esegue il pad dei file `itertools.izip_longest` più brevi con `None` (simile a `itertools.izip_longest`). In Python 3, l'iterazione si interrompe dopo il più breve iterabile.

In Python 2:

Python 2.x 2.3

```
>>> map(None, [1, 2, 3], [1, 2], [1, 2, 3, 4, 5])
[(1, 1, 1), (2, 2, 2), (3, None, 3), (None, None, 4), (None, None, 5)]
```

In Python 3:

Python 3.x 3.0

```
>>> list(map(lambda x, y, z: (x, y, z), [1, 2, 3], [1, 2], [1, 2, 3, 4, 5]))
[(1, 1, 1), (2, 2, 2)]

# to obtain the same padding as in Python 2 use zip_longest from itertools
>>> import itertools
>>> list(itertools.zip_longest([1, 2, 3], [1, 2], [1, 2, 3, 4, 5]))
[(1, 1, 1), (2, 2, 2), (3, None, 3), (None, None, 4), (None, None, 5)]
```

Nota : al posto della `map` considera l'utilizzo delle list comprehensions, che sono compatibili con Python 2/3. Sostituzione `map(str, [1, 2, 3, 4, 5])` :

```
>>> [str(i) for i in [1, 2, 3, 4, 5]]
['1', '2', '3', '4', '5']
```

filter (), map () e zip () restituiscono gli iteratori anziché le sequenze

Python 2.x 2.7

Nel `filter` Python 2, le funzioni incorporate di `map` e `zip` restituiscono una sequenza. `map` e `zip` restituiscono sempre una lista mentre con `filter` il tipo restituito dipende dal tipo di parametro dato:

```
>>> s = filter(lambda x: x.isalpha(), 'a1b2c3')
>>> s
'abc'
>>> s = map(lambda x: x * x, [0, 1, 2])
>>> s
[0, 1, 4]
>>> s = zip([0, 1, 2], [3, 4, 5])
>>> s
[(0, 3), (1, 4), (2, 5)]
```

Python 3.x 3.0

Nel `filter` Python 3, `map` e `zip` restituiscono invece iterator:

```
>>> it = filter(lambda x: x.isalpha(), 'a1b2c3')
>>> it
<filter object at 0x00000098A55C2518>
>>> ''.join(it)
'abc'
>>> it = map(lambda x: x * x, [0, 1, 2])
>>> it
<map object at 0x000000E0763C2D30>
>>> list(it)
[0, 1, 4]
>>> it = zip([0, 1, 2], [3, 4, 5])
>>> it
<zip object at 0x000000E0763C52C8>
>>> list(it)
[(0, 3), (1, 4), (2, 5)]
```

Poiché Python 2 `itertools.izip` è equivalente a Python 3 `zip` `izip` è stato rimosso su Python 3.

Importazioni assolute / relative

In Python 3, PEP 404 cambia il modo in cui le importazioni funzionano da Python 2. Le importazioni *relative implicite* non sono più consentite nei pacchetti e `from ... import *` import sono consentite solo nel codice a livello di modulo.

Per ottenere il comportamento di Python 3 in Python 2:

- la funzione di importazione assoluta può essere abilitata con `from __future__ import absolute_import`
- *le* importazioni *relative esplicite* sono incoraggiate al posto delle importazioni *relative implicite*

Per chiarimenti, in Python 2, un modulo può importare il contenuto di un altro modulo situato nella stessa directory come segue:

```
import foo
```

Si noti che la posizione di `foo` è ambigua dalla sola dichiarazione di importazione. Questo tipo di importazione relativa implicita è quindi scoraggiato a favore delle importazioni relative esplicite , che assomigliano alle seguenti:

```
from .moduleY import spam
from .moduleY import spam as ham
from . import moduleY
from ..subpackage1 import moduleY
from ..subpackage2.moduleZ import eggs
from ..moduleA import foo
from ...package import bar
from ...sys import path
```

Il punto `.` consente una dichiarazione esplicita della posizione del modulo all'interno dell'albero delle directory.

Maggiori informazioni sulle importazioni relative

Prendi in considerazione un pacchetto definito dall'utente chiamato `shapes` . La struttura della directory è la seguente:

```
shapes
├── __init__.py
|
├── circle.py
|
├── square.py
|
└── triangle.py
```

`circle.py` , `square.py` e `triangle.py` importano `util.py` come modulo. Come faranno riferimento a un

modulo nello stesso livello?

```
from . import util # use util.PI, util.sq(x), etc
```

O

```
from .util import * #use PI, sq(x), etc to call functions
```

Il `.` è usato per importazioni relative allo stesso livello.

Ora, considera un layout alternativo del modulo `shapes` :

```
shapes
├── __init__.py
|
├── circle
|   ├── __init__.py
|   └── circle.py
|
├── square
|   ├── __init__.py
|   └── square.py
|
├── triangle
|   ├── __init__.py
|   ├── triangle.py
|
└── util.py
```

Ora, in che modo queste 3 classi si riferiscono a util.py?

```
from .. import util # use util.PI, util.sq(x), etc
```

O

```
from ..util import * # use PI, sq(x), etc to call functions
```

Il `..` è usato per importazioni relative a livello di genitore. Aggiungi altro `.` s con il numero di livelli tra genitore e figlio.

File I / O

`file` non è più un nome incorporato in 3.x (`open` funziona ancora).

I dettagli interni del file I / O sono stati spostati nel modulo `io` libreria standard, che è anche la nuova casa di `StringIO` :

```
import io
assert io.open is open # the builtin is an alias
buffer = io.StringIO()
buffer.write('hello, ') # returns number of characters written
```

```
buffer.write('world!\n')
buffer.getvalue() # 'hello, world!\n'
```

La modalità file (testo vs binario) ora determina il tipo di dati prodotti dalla lettura di un file (e il tipo richiesto per la scrittura):

```
with open('data.txt') as f:
    first_line = next(f)
    assert type(first_line) is str
with open('data.bin', 'rb') as f:
    first_kb = f.read(1024)
    assert type(first_kb) is bytes
```

La codifica per i file di testo ha come valore predefinito quello che viene restituito da `locale.getpreferredencoding(False)` . Per specificare esplicitamente una codifica, utilizzare il parametro della parola chiave `encoding` :

```
with open('old_japanese_poetry.txt', 'shift_jis') as text:
    haiku = text.read()
```

La funzione round () tie-break e return

round () tie break

In Python 2, l'uso di `round()` su un numero uguale a due numeri interi restituirà quello più lontano da 0. Ad esempio:

Python 2.x 2.7

```
round(1.5)  # Out: 2.0
round(0.5)  # Out: 1.0
round(-0.5)  # Out: -1.0
round(-1.5)  # Out: -2.0
```

In Python 3 tuttavia, `round()` restituirà l'intero pari (ovvero *l'arrotondamento dei banchieri*). Per esempio:

Python 3.x 3.0

```
round(1.5)  # Out: 2
round(0.5)  # Out: 0
round(-0.5)  # Out: 0
round(-1.5)  # Out: -2
```

La funzione round () segue la strategia di *arrotondamento da mezzo a arrotondamento* che arrotonderà i numeri a metà fino al numero intero pari più vicino (ad esempio, `round(2.5)` restituisce 2 anziché 3.0).

Come riferimento in Wikipedia , questo è anche conosciuto come *arrotondamento imparziale* , *arrotondamento convergente* , *arrotondamento di uno statistico, arrotondamento olandese* ,

arrotondamento gaussiano o *arrotondamento dispari-pari* .

La metà dell'arrotondamento è parte dello standard IEEE 754 ed è anche la modalità di arrotondamento predefinita in .NET di Microsoft.

Questa strategia di arrotondamento tende a ridurre l'errore di arrotondamento totale. Poiché in media la quantità di numeri arrotondati è uguale alla quantità di numeri arrotondati, gli errori di arrotondamento vengono annullati. Altri metodi di arrotondamento tendono invece ad avere una tendenza verso l'alto o verso il basso nell'errore medio.

---

round () tipo di ritorno

La funzione `round()` restituisce un tipo `float` in Python 2.7

Python 2.x 2.7

```
round(4.8)
# 5.0
```

A partire da Python 3.0, se il secondo argomento (numero di cifre) viene omesso, restituisce un `int` .

Python 3.x 3.0

```
round(4.8)
# 5
```

Vero, Falso e Nessuno

In Python 2, `True` , `False` e `None` sono costanti incorporate. Il che significa che è possibile riassegnarli.

Python 2.x 2.0

```
True, False = False, True
True   # False
False  # True
```

Non puoi farlo con `None` da Python 2.4.

Python 2.x 2.4

```
None = None  # SyntaxError: cannot assign to None
```

In Python 3, `True` , `False` e `None` ora sono le parole chiave.

Python 3.x 3.0

```
True, False = False, True  # SyntaxError: can't assign to keyword
```

```
None = None  # SyntaxError: can't assign to keyword
```

Restituisce valore quando si scrive su un oggetto file

In Python 2, scrivere direttamente su un handle di file restituisce `None` :

Python 2.x 2.3

```
hi = sys.stdout.write('hello world\n')
# Out: hello world
type(hi)
# Out: <type 'NoneType'>
```

In Python 3, la scrittura su un handle restituirà il numero di caratteri scritti durante la scrittura del testo e il numero di byte scritti durante la scrittura di byte:

Python 3.x 3.0

```
import sys

char_count = sys.stdout.write('hello world \n')
# Out: hello world 
char_count
# Out: 14

byte_count = sys.stdout.buffer.write(b'hello world \xf0\x9f\x90\x8d\n')
# Out: hello world 
byte_count
# Out: 17
```

long vs. int

In Python 2, qualsiasi intero più grande di un C `ssize_t` verrebbe convertito nel tipo di dati `long` , indicato da un suffisso `L` sul letterale. Ad esempio, su una build di Python a 32 bit:

Python 2.x 2.7

```
>>> 2**31
2147483648L
>>> type(2**31)
<type 'long'>
>>> 2**30
1073741824
>>> type(2**30)
<type 'int'>
>>> 2**31 - 1  # 2**31 is long and long - int is long
2147483647L
```

Tuttavia, in Python 3, il `long` tipo di dati è stato rimosso; non importa quanto sia grande il numero intero, sarà un `int` .

Python 3.x 3.0

```
2**1024
# Output:
179769313486231590772930519078902473361797697894230657273430081157732675805500963132708477322407536021

print(-(2**1024))
# Output: -
179769313486231590772930519078902473361797697894230657273430081157732675805500963132708477322407536021

type(2**1024)
# Output: <class 'int'>
```

Classe Valore booleano

Python 2.x 2.7

In Python 2, se si desidera definire un valore booleano della classe da soli, è necessario implementare il metodo `__nonzero__` sulla classe. Il valore è True per impostazione predefinita.

```
class MyClass:
    def __nonzero__(self):
        return False

my_instance = MyClass()
print bool(MyClass)        # True
print bool(my_instance)    # False
```

Python 3.x 3.0

In Python 3, `__bool__` è usato al posto di `__nonzero__`

```
class MyClass:
    def __bool__(self):
        return False

my_instance = MyClass()
print(bool(MyClass))        # True
print(bool(my_instance))    # False
```

Leggi Incompatibilità che si spostano da Python 2 a Python 3 online:
https://riptutorial.com/it/python/topic/809/incompatibilita-che-si-spostano-da-python-2-a-python-3

Capitolo 85: Indicizzazione e affettatura

Sintassi

- obj [start: fermata: passo]
- slice (stop)
- slice (start, stop [, step])

Parametri

Paramer	Descrizione
`obj`	L'oggetto da cui estrarre un "oggetto secondario"
`start`	L'indice di `obj` cui si desidera che l'oggetto secondario inizi (si tenga presente che Python è a indice zero, ovvero che il primo elemento di `obj` ha un indice di `0`). Se omesso, il valore predefinito è `0` .
`stop`	L'indice (non inclusivo) di `obj` cui si desidera terminare l'oggetto secondario. Se omesso, il valore predefinito è `len(obj)` .
`step`	Consente di selezionare solo ogni elemento del `step` . Se omesso, il valore predefinito è `1` .

Osservazioni

È possibile unificare il concetto di affettare le stringhe con quello di affettare le altre sequenze visualizzando le stringhe come una collezione immutabile di caratteri, con l'avvertenza che un carattere unicode è rappresentato da una stringa di lunghezza 1.

Nella notazione matematica è possibile considerare l'affettamento per utilizzare un intervallo semiaperto di `[start, end)` , vale a dire che l'inizio è incluso ma la fine non lo è. La natura semiaperta dell'intervallo ha il vantaggio che `len(x[:n])` = `n` dove `len(x)` > = `n` , mentre l'intervallo che viene chiuso all'inizio ha il vantaggio che `x[n:n+1]` = `[x[n]]` dove `x` è una lista con `len(x) >= n` , mantenendo così la coerenza tra l'indicizzazione e la notazione slicing.

Examples

Slicing di base

Per ogni iterabile (per esempio una stringa, una lista, ecc.), Python consente di suddividere e restituire una sottostringa o una sottolista dei suoi dati.

Formato per affettare:

```
iterable_name[start:stop:step]
```

dove,

- `start` è il primo indice della sezione. Il valore predefinito è 0 (l'indice del primo elemento)
- `stop` uno dopo l'ultimo indice della fetta. Il valore predefinito è len (iterabile)
- `step` è la dimensione del passo (meglio spiegato dagli esempi di seguito)

Esempi:

```
a = "abcdef"
a             # "abcdef"
              # Same as a[:] or a[::] since it uses the defaults for all three indices
a[-1]         # "f"
a[:]          # "abcdef"
a[::]         # "abcdef"
a[3:]         # "def" (from index 3, to end(defaults to size of iterable))
a[:4]         # "abcd" (from beginning(default 0) to position 4 (excluded))
a[2:4]        # "cd" (from position 2, to position 4 (excluded))
```

Inoltre, uno dei precedenti può essere utilizzato con le dimensioni del passo definite:

```
a[::2]        # "ace" (every 2nd element)
a[1:4:2]      # "bd" (from index 1, to index 4 (excluded), every 2nd element)
```

Gli indici possono essere negativi, nel qual caso vengono calcolati dalla fine della sequenza

```
a[:-1]      # "abcde" (from index 0 (default), to the second last element (last element - 1))
a[:-2]      # "abcd" (from index 0 (default), to the third last element (last element -2))
a[-1:]      # "f" (from the last element to the end (default len())
```

Anche le dimensioni degli step possono essere negative, nel qual caso la slice itererà attraverso la lista in ordine inverso:

```
a[3:1:-1]   # "dc" (from index 2 to None (default), in reverse order)
```

Questo costrutto è utile per invertire un iterabile

```
a[::-1]     # "fedcba" (from last element (default len()-1), to first, in reverse order(-1))
```

Nota che per i passaggi negativi l' `end_index` predefinito è `None` (vedi http://stackoverflow.com/a/12521981)

```
a[5:None:-1] # "fedcba" (this is equivalent to a[::-1])
a[5:0:-1]    # "fedcb" (from the last element (index 5) to second element (index 1)
```

Fare una copia superficiale di un array

Un modo rapido per fare una copia di un array (al contrario di assegnare una variabile con un altro

riferimento alla matrice originale) è:

```
arr[:]
```

Esaminiamo la sintassi. `[:]` significa che `start` , `end` e `slice` sono tutti omessi. Hanno come valore predefinito `0` , `len(arr)` e `1` , rispettivamente, il che significa che il subarray che stiamo richiedendo avrà tutti gli elementi di `arr` dall'inizio fino alla fine.

In pratica, questo assomiglia a qualcosa:

```
arr = ['a', 'b', 'c']
copy = arr[:]
arr.append('d')
print(arr)    # ['a', 'b', 'c', 'd']
print(copy)   # ['a', 'b', 'c']
```

Come puoi vedere, `arr.append('d')` aggiunto `d` a `arr` , ma la `copy` rimasta invariata!

Si noti che questo fa una copia *superficiale* , ed è identico a `arr.copy()` .

Inversione di un oggetto

È possibile utilizzare le sezioni per invertire molto facilmente una `str` , `list` o `tuple` (o sostanzialmente qualsiasi oggetto di raccolta che implementa l'slicing con il parametro step). Ecco un esempio di inversione di una stringa, sebbene ciò si applichi anche agli altri tipi sopra elencati:

```
s = 'reverse me!'
s[::-1]    # '!em esrever'
```

Diamo un'occhiata veloce alla sintassi. `[::-1]` significa che la sezione dovrebbe essere dall'inizio fino alla fine della stringa (poiché l' `start` e la `end` sono omessi) e un passo di `-1` significa che dovrebbe spostarsi attraverso la stringa al contrario.

Indicizzazione delle classi personalizzate: __getitem__, __setitem__ e __delitem__

```
class MultiIndexingList:
    def __init__(self, value):
        self.value = value

    def __repr__(self):
        return repr(self.value)

    def __getitem__(self, item):
        if isinstance(item, (int, slice)):
            return self.__class__(self.value[item])
        return [self.value[i] for i in item]

    def __setitem__(self, item, value):
        if isinstance(item, int):
            self.value[item] = value
```

```
        elif isinstance(item, slice):
            raise ValueError('Cannot interpret slice with multiindexing')
        else:
            for i in item:
                if isinstance(i, slice):
                    raise ValueError('Cannot interpret slice with multiindexing')
                self.value[i] = value

    def __delitem__(self, item):
        if isinstance(item, int):
            del self.value[item]
        elif isinstance(item, slice):
            del self.value[item]
        else:
            if any(isinstance(elem, slice) for elem in item):
                raise ValueError('Cannot interpret slice with multiindexing')
            item = sorted(item, reverse=True)
            for elem in item:
                del self.value[elem]
```

Ciò consente l'affettatura e l'indicizzazione per l'accesso agli elementi:

```
a = MultiIndexingList([1,2,3,4,5,6,7,8])
a
# Out: [1, 2, 3, 4, 5, 6, 7, 8]
a[1,5,2,6,1]
# Out: [2, 6, 3, 7, 2]
a[4, 1, 5:, 2, ::2]
# Out: [5, 2, [6, 7, 8], 3, [1, 3, 5, 7]]
#       4|1-|----50:---|2-|-----::2-----   <-- indicated which element came from which index
```

Mentre l'impostazione e l'eliminazione di elementi consente solo l'indicizzazione di numeri interi *separati da virgole* (nessuna suddivisione):

```
a[4] = 1000
a
# Out: [1, 2, 3, 4, 1000, 6, 7, 8]
a[2,6,1] = 100
a
# Out: [1, 100, 100, 4, 1000, 6, 100, 8]
del a[5]
a
# Out: [1, 100, 100, 4, 1000, 100, 8]
del a[4,2,5]
a
# Out: [1, 100, 4, 8]
```

Assegnazione delle fette

Un'altra caratteristica accurata che utilizza le slice è l'assegnazione delle slice. Python consente di assegnare nuove sezioni per sostituire le vecchie sezioni di un elenco in una singola operazione.

Ciò significa che se hai una lista, puoi sostituire più membri in un singolo incarico:

```
lst = [1, 2, 3]
```

```
lst[1:3] = [4, 5]
print(lst) # Out: [1, 4, 5]
```

Il compito non deve corrispondere anche alle dimensioni, quindi se si volesse sostituire una vecchia sezione con una nuova sezione di dimensioni diverse, è possibile:

```
lst = [1, 2, 3, 4, 5]
lst[1:4] = [6]
print(lst) # Out: [1, 6, 5]
```

È anche possibile utilizzare la sintassi di slicing nota per eseguire operazioni come la sostituzione dell'intero elenco:

```
lst = [1, 2, 3]
lst[:] = [4, 5, 6]
print(lst) # Out: [4, 5, 6]
```

O solo gli ultimi due membri:

```
lst = [1, 2, 3]
lst[-2:] = [4, 5, 6]
print(lst) # Out: [1, 4, 5, 6]
```

Affetta oggetti

Le `slice()` sono oggetti in sé e possono essere archiviate in variabili con la funzione `slice()` integrata. Le variabili di sezione possono essere utilizzate per rendere il codice più leggibile e per promuovere il riutilizzo.

```
>>> programmer_1 = [ 1956, 'Guido', 'van Rossum', 'Python', 'Netherlands']
>>> programmer_2 = [ 1815, 'Ada', 'Lovelace', 'Analytical Engine', 'England']
>>> name_columns = slice(1, 3)
>>> programmer_1[name_columns]
['Guido', 'van Rossum']
>>> programmer_2[name_columns]
['Ada', 'Lovelace']
```

Indicizzazione di base

Gli elenchi Python sono basati su 0, *ovvero* il primo elemento nell'elenco può essere consultato dall'indice `0`

```
arr = ['a', 'b', 'c', 'd']
print(arr[0])
>> 'a'
```

È possibile accedere al secondo elemento nell'elenco per indice `1` , terzo elemento per indice `2` e così via:

```
print(arr[1])
```

```
>> 'b'
print(arr[2])
>> 'c'
```

È inoltre possibile utilizzare indici negativi per accedere agli elementi dalla fine dell'elenco. per esempio. index `-1` ti darà l'ultimo elemento della lista e index `-2` ti darà il penultimo elemento della lista:

```
print(arr[-1])
>> 'd'
print(arr[-2])
>> 'c'
```

Se si tenta di accedere a un indice che non è presente nell'elenco, verrà sollevato un `IndexError` :

```
print arr[6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Leggi Indicizzazione e affettatura online: https://riptutorial.com/it/python/topic/289/indicizzazione-e-affettatura

Capitolo 86: iniziare con GZip

introduzione

Questo modulo fornisce una semplice interfaccia per comprimere e decomprimere i file proprio come i programmi GNU sarebbero gzip e gunzip.

La compressione dei dati è fornita dal modulo zlib.

Il modulo gzip fornisce la classe GzipFile che è modellata su Python's File Object. La classe GzipFile legge e scrive file in formato gzip, comprimendo o decomprimendo automaticamente i dati in modo che assomiglino a un normale oggetto file.

Examples

Leggi e scrivi i file zip GNU

```
import gzip
import os

outfilename = 'example.txt.gz'
output = gzip.open(outfilename, 'wb')
try:
    output.write('Contents of the example file go here.\n')
finally:
    output.close()

print outfilename, 'contains', os.stat(outfilename).st_size, 'bytes of compressed data'
os.system('file -b --mime %s' % outfilename)
```

Salvalo come 1gzip_write.py1.Eseguilo attraverso il terminale.

```
$ python gzip_write.py

application/x-gzip; charset=binary
example.txt.gz contains 68 bytes of compressed data
```

Leggi iniziare con GZip online: https://riptutorial.com/it/python/topic/8993/iniziare-con-gzip

Capitolo 87: Input e output di base

Examples

Utilizzo di input () e raw_input ()

Python 2.x 2.3

`raw_input` attenderà che l'utente inserisca del testo e restituisca il risultato come una stringa.

```
foo = raw_input("Put a message here that asks the user for input")
```

Nell'esempio precedente `foo` memorizzerà qualsiasi input fornito dall'utente.

Python 3.x 3.0

`input` attenderà che l'utente inserisca il testo e quindi restituisca il risultato come una stringa.

```
foo = input("Put a message here that asks the user for input")
```

Nell'esempio precedente `foo` memorizzerà qualsiasi input fornito dall'utente.

Utilizzando la funzione di stampa

Python 3.x 3.0

In Python 3, la funzionalità di stampa ha la forma di una funzione:

```
print("This string will be displayed in the output")
# This string will be displayed in the output

print("You can print \n escape characters too.")
# You can print escape characters too.
```

Python 2.x 2.3

In Python 2, la stampa era in origine una dichiarazione, come mostrato di seguito.

```
print "This string will be displayed in the output"
# This string will be displayed in the output

print "You can print \n escape characters too."
# You can print escape characters too.
```

Nota: l'utilizzo `from __future__ import print_function` in Python 2 consentirà agli utenti di utilizzare la funzione `print()` come nel codice Python 3. Questo è disponibile solo in Python 2.6 e versioni successive.

Funzione per richiedere all'utente un numero

```
def input_number(msg, err_msg=None):
    while True:
        try:
            return float(raw_input(msg))
        except ValueError:
            if err_msg is not None:
                print(err_msg)


def input_number(msg, err_msg=None):
    while True:
        try:
            return float(input(msg))
        except ValueError:
            if err_msg is not None:
                print(err_msg)
```

E per usarlo:

```
user_number = input_number("input a number: ", "that's not a number!")
```

Oppure, se non vuoi un "messaggio di errore":

```
user_number = input_number("input a number: ")
```

Stampa una stringa senza una nuova riga alla fine

Python 2.x 2.3

In Python 2.x, per continuare una riga con la `print` , terminare l'istruzione di `print` con una virgola. Aggiungerà automaticamente uno spazio.

```
print "Hello,",
print "World!"
# Hello, World!
```

Python 3.x 3.0

In Python 3.x, la funzione di `print` ha un parametro `end` opzionale che è ciò che stampa alla fine della stringa specificata. Di default è un carattere di nuova riga, quindi equivalente a questo:

```
print("Hello, ", end="\n")
print("World!")
# Hello,
# World!
```

Ma potresti passare in altre stringhe

```
print("Hello, ", end="")
```

```
print("World!")
# Hello, World!

print("Hello, ", end="<br>")
print("World!")
# Hello, <br>World!

print("Hello, ", end="BREAK")
print("World!")
# Hello, BREAKWorld!
```

Se vuoi un maggiore controllo sull'output, puoi usare `sys.stdout.write` :

```
import sys

sys.stdout.write("Hello, ")
sys.stdout.write("World!")
# Hello, World!
```

Leggi da stdin

I programmi Python possono leggere da pipeline unix . Ecco un semplice esempio su come leggere da `stdin` :

```
import sys

for line in sys.stdin:
    print(line)
```

Essere consapevoli del fatto che `sys.stdin` è un flusso. Significa che il ciclo for terminerà solo al termine dello stream.

Ora puoi reindirizzare l'output di un altro programma nel tuo programma python come segue:

```
$ cat myfile | python myprogram.py
```

In questo esempio `cat myfile` può essere un qualsiasi comando unix che esce `stdout` .

In alternativa, l'uso del modulo fileinput può tornare utile:

```
import fileinput
for line in fileinput.input():
    process(line)
```

Input da un file

L'input può anche essere letto dai file. I file possono essere aperti utilizzando la funzione integrata `open` . L'utilizzo di un `with <command> as <name>` sintassi `with <command> as <name>` (chiamato "Gestore del contesto") rende l'uso di `open` e ottenere un handle per il file super facile:

```
with open('somefile.txt', 'r') as fileobj:
```

```
    # write code here using fileobj
```

Ciò garantisce che quando l'esecuzione del codice lascia il blocco, il file viene automaticamente chiuso.

I file possono essere aperti in diverse modalità. Nell'esempio sopra il file è aperto in sola lettura. Per aprire un file esistente per la sola lettura usa `r` . Se vuoi leggere quel file come byte usa `rb` . Per aggiungere dati a un file esistente, utilizzare `a` . Usa `w` per creare un file o sovrascrivere qualsiasi file esistente con lo stesso nome. Puoi usare `r+` per aprire un file sia per la lettura che per la scrittura. Il primo argomento di `open()` è il nome del file, il secondo è la modalità. Se la modalità viene lasciata vuota, verrà impostata su `r` .

```
# let's create an example file:
with open('shoppinglist.txt', 'w') as fileobj:
    fileobj.write('tomato\npasta\ngarlic')

with open('shoppinglist.txt', 'r') as fileobj:
    # this method makes a list where each line
    # of the file is an element in the list
    lines = fileobj.readlines()

print(lines)
# ['tomato\n', 'pasta\n', 'garlic']

with open('shoppinglist.txt', 'r') as fileobj:
    # here we read the whole content into one string:
    content = fileobj.read()
    # get a list of lines, just like int the previous example:
    lines = content.split('\n')

print(lines)
# ['tomato', 'pasta', 'garlic']
```

Se la dimensione del file è minuscola, è sicuro leggere l'intero contenuto del file in memoria. Se il file è molto grande, spesso è meglio leggere riga per riga o blocchi e elaborare l'input nello stesso ciclo. Fare quello:

```
with open('shoppinglist.txt', 'r') as fileobj:
    # this method reads line by line:
    lines = []
    for line in fileobj:
        lines.append(line.strip())
```

Durante la lettura dei file, prestare attenzione ai caratteri di interruzione di riga specifici del sistema operativo. Anche se `for line in fileobj` le `for line in fileobj` automaticamente, è sempre possibile chiamare `strip()` sulle righe lette, come mostrato sopra.

I file aperti (`fileobj` negli esempi precedenti) indicano sempre una posizione specifica nel file. Quando vengono aperti per la prima volta, il file handle punta all'inizio del file, che è la posizione `0` . L'handle del file può visualizzare la posizione corrente con `tell` :

```
fileobj = open('shoppinglist.txt', 'r')
pos = fileobj.tell()
```

```
print('We are at %u.' % pos) # We are at 0.
```

Dopo aver letto tutto il contenuto, la posizione del gestore file sarà indicata alla fine del file:

```
content = fileobj.read()
end = fileobj.tell()
print('This file was %u characters long.' % end)
# This file was 22 characters long.
fileobj.close()
```

La posizione del gestore file può essere impostata su qualsiasi cosa sia necessaria:

```
fileobj = open('shoppinglist.txt', 'r')
fileobj.seek(7)
pos = fileobj.tell()
print('We are at character #%u.' % pos)
```

Puoi anche leggere qualsiasi lunghezza dal contenuto del file durante una determinata chiamata. Per fare questo passare un argomento per `read()` . Quando `read()` viene chiamato senza argomento, leggerà fino alla fine del file. Se passi un argomento leggerà quel numero di byte o caratteri, a seconda della modalità (rispettivamente `rb` e `r`):

```
# reads the next 4 characters
# starting at the current position
next4 = fileobj.read(4)
# what we got?
print(next4) # 'cucu'
# where we are now?
pos = fileobj.tell()
print('We are at %u.' % pos) # We are at 11, as we was at 7, and read 4 chars.

fileobj.close()
```

Per dimostrare la differenza tra caratteri e byte:

```
with open('shoppinglist.txt', 'r') as fileobj:
    print(type(fileobj.read())) # <class 'str'>

with open('shoppinglist.txt', 'rb') as fileobj:
    print(type(fileobj.read())) # <class 'bytes'>
```

Leggi Input e output di base online: https://riptutorial.com/it/python/topic/266/input-e-output-di-base

Capitolo 88: Input, Subset e Output File di dati esterni che utilizzano Panda

introduzione

Questa sezione mostra il codice di base per la lettura, la sotto-impostazione e la scrittura di file di dati esterni usando i panda.

Examples

Codice di base per importare, sottoinsi e scrivere file di dati esterni usando panda

```
# Print the working directory
import os
print os.getcwd()
# C:\Python27\Scripts

# Set the working directory
os.chdir('C:/Users/general1/Documents/simple Python files')
print os.getcwd()
# C:\Users\general1\Documents\simple Python files

# load pandas
import pandas as pd

# read a csv data file named 'small_dataset.csv' containing 4 lines and 3 variables
my_data = pd.read_csv("small_dataset.csv")
my_data
#     x   y   z
# 0   1   2   3
# 1   4   5   6
# 2   7   8   9
# 3  10  11  12

my_data.shape       # number of rows and columns in data set
# (4, 3)

my_data.shape[0]    # number of rows in data set
# 4

my_data.shape[1]    # number of columns in data set
# 3

# Python uses 0-based indexing.  The first row or column in a data set is located
# at position 0.  In R the first row or column in a data set is located
# at position 1.

# Select the first two rows
my_data[0:2]
#    x   y   z
#0   1   2   3
```

```
#1   4   5   6

# Select the second and third rows
my_data[1:3]
#    x  y  z
# 1  4  5  6
# 2  7  8  9

# Select the third row
my_data[2:3]
#    x   y   z
#2   7   8   9

# Select the first two elements of the first column
my_data.iloc[0:2, 0:1]
#    x
# 0  1
# 1  4

# Select the first element of the variables y and z
my_data.loc[0, ['y', 'z']]
# y    2
# z    3

# Select the first three elements of the variables y and z
my_data.loc[0:2, ['y', 'z']]
#    y  z
# 0  2  3
# 1  5  6
# 2  8  9

# Write the first three elements of the variables y and z
# to an external file.  Here index = 0 means do not write row names.

my_data2 = my_data.loc[0:2, ['y', 'z']]

my_data2.to_csv('my.output.csv', index = 0)
```

Leggi Input, Subset e Output File di dati esterni che utilizzano Panda online: https://riptutorial.com/it/python/topic/8854/input--subset-e-output-file-di-dati-esterni-che-utilizzano-panda

Capitolo 89: Insidie comuni

introduzione

Python è un linguaggio pensato per essere chiaro e leggibile senza ambiguità e comportamenti imprevisti. Sfortunatamente, questi obiettivi non sono raggiungibili in tutti i casi, ed è per questo che Python ha alcuni casi d'angolo in cui potrebbe fare qualcosa di diverso da quello che ti aspettavi.

Questa sezione ti mostrerà alcuni problemi che potresti incontrare durante la scrittura del codice Python.

Examples

Modifica della sequenza su cui stai iterando

Un ciclo `for` itera su una sequenza, quindi **alterare questa sequenza all'interno del ciclo potrebbe portare a risultati imprevisti** (specialmente quando si aggiungono o rimuovono elementi):

```
alist = [0, 1, 2]
for index, value in enumerate(alist):
    alist.pop(index)
print(alist)
# Out: [1]
```

Nota: `list.pop()` viene utilizzato per rimuovere elementi dall'elenco.

Il secondo elemento non è stato cancellato perché l'iterazione attraversa gli indici in ordine. Il ciclo sopra riportato itera due volte, con i seguenti risultati:

```
# Iteration #1
index = 0
alist = [0, 1, 2]
alist.pop(0) # removes '0'

# Iteration #2
index = 1
alist = [1, 2]
alist.pop(1) # removes '2'

# loop terminates, but alist is not empty:
alist = [1]
```

Questo problema sorge perché gli indici stanno cambiando mentre si sta ripetendo nella direzione di aumentare l'indice. Per evitare questo problema, puoi **scorrere il ciclo all'indietro** :

```
alist = [1,2,3,4,5,6,7]
```

```
for index, item in reversed(list(enumerate(alist))):
    # delete all even items
    if item % 2 == 0:
        alist.pop(index)
print(alist)
# Out: [1, 3, 5, 7]
```

Eseguendo il ciclo dall'inizio alla fine, quando gli elementi vengono rimossi (o aggiunti), non influisce sugli indici delle voci precedenti nell'elenco. Quindi questo esempio rimuoverà correttamente tutti gli elementi che sono anche da `alist` .

Un problema simile si presenta quando si **inseriscono o si aggiungono elementi a un elenco su cui si sta iterando** , il che può generare un loop infinito:

```
alist = [0, 1, 2]
for index, value in enumerate(alist):
    # break to avoid infinite loop:
    if index == 20:
        break
    alist.insert(index, 'a')
print(alist)
# Out (abbreviated): ['a', 'a', ..., 'a', 'a',  0,   1,   2]
```

Senza la condizione di `break` , il ciclo inserirà `'a'` fintanto che il computer non esaurirà la memoria e il programma potrà continuare. In una situazione come questa, in genere è preferibile creare un nuovo elenco e aggiungere elementi al nuovo elenco mentre si scorre l'elenco originale.

Quando si utilizza un ciclo `for` , **non è possibile modificare gli elementi dell'elenco con la variabile placeholder** :

```
alist = [1,2,3,4]
for item in alist:
    if item % 2 == 0:
        item = 'even'
print(alist)
# Out: [1,2,3,4]
```

Nell'esempio sopra, la **modifica `item` non modifica nulla nella lista originale** . È necessario utilizzare l'indice di lista (`alist[2]`) e `enumerate()` funziona bene per questo:

```
alist = [1,2,3,4]
for index, item in enumerate(alist):
    if item % 2 == 0:
        alist[index] = 'even'
print(alist)
# Out: [1, 'even', 3, 'even']
```

Un **ciclo `while`** potrebbe essere una scelta migliore in alcuni casi:

Se stai per **eliminare tutti gli elementi** nell'elenco:

```
zlist = [0, 1, 2]
while zlist:
    print(zlist[0])
    zlist.pop(0)
print('After: zlist =', zlist)

# Out: 0
#      1
#      2
# After: zlist = []
```

Sebbene la semplice reimpostazione di `zlist` lo stesso risultato;

```
zlist = []
```

L'esempio sopra può anche essere combinato con `len()` per fermarsi dopo un certo punto, o per eliminare tutti gli elementi tranne `x` nella lista:

```
zlist = [0, 1, 2]
x = 1
while len(zlist) > x:
    print(zlist[0])
    zlist.pop(0)
print('After: zlist =', zlist)

# Out: 0
#      1
# After: zlist = [2]
```

Oppure per **scorrere un elenco durante l'eliminazione di elementi che soddisfano una determinata condizione** (in questo caso l'eliminazione di tutti gli elementi pari):

```
zlist = [1,2,3,4,5]
i = 0
while i < len(zlist):
    if zlist[i] % 2 == 0:
        zlist.pop(i)
    else:
        i += 1
print(zlist)
# Out: [1, 3, 5]
```

Si noti che non si incrementa `i` dopo aver eliminato un elemento. `zlist[i]` l'elemento su `zlist[i]` , l'indice dell'elemento successivo è diminuito di uno, quindi controllando `zlist[i]` con lo stesso valore per `i` alla successiva iterazione, si verificherà correttamente l'elemento successivo nell'elenco .

Un modo contrario di pensare a rimuovere elementi indesiderati da una lista è **aggiungere elementi desiderati a una nuova lista** . L'esempio seguente è un'alternativa al secondo esempio di ciclo `while` :

```
zlist = [1,2,3,4,5]
```

```
z_temp = []
for item in zlist:
    if item % 2 != 0:
        z_temp.append(item)
zlist = z_temp
print(zlist)
# Out: [1, 3, 5]
```

Qui stiamo incanalando i risultati desiderati in una nuova lista. Possiamo quindi riassegnare facoltativamente l'elenco temporaneo alla variabile originale.

Con questa tendenza del pensiero, puoi invocare una delle più eleganti e potenti funzionalità di Python, la **list comprehensions** , che elimina le liste temporanee e le divergenze dall'ideologia sul listino sul posto / sull'indice degli indici.

```
zlist = [1,2,3,4,5]
[item for item in zlist if item % 2 != 0]
# Out: [1, 3, 5]
```

Argomento predefinito mutabile

```
def foo(li=[]):
    li.append(1)
    print(li)

foo([2])
# Out: [2, 1]
foo([3])
# Out: [3, 1]
```

Questo codice si comporta come previsto, ma cosa succede se non passiamo un argomento?

```
foo()
# Out: [1] As expected...

foo()
# Out: [1, 1]  Not as expected...
```

Questo perché gli argomenti predefiniti di funzioni e metodi vengono valutati in fase di **definizione** piuttosto che in fase di esecuzione. Quindi abbiamo sempre una sola istanza della lista `li` .

Il modo per aggirarlo è usare solo tipi immutabili per gli argomenti predefiniti:

```
def foo(li=None):
    if not li:
        li = []
    li.append(1)
    print(li)

foo()
# Out: [1]
```

```
foo()
# Out: [1]
```

Sebbene sia un miglioramento e sebbene `if not li` correttamente per `False` , anche molti altri oggetti funzionano come sequenze di lunghezza zero. I seguenti argomenti di esempio possono causare risultati indesiderati:

```
x = []
foo(li=x)
# Out: [1]

foo(li="")
# Out: [1]

foo(li=0)
# Out: [1]
```

L'approccio idiomatico consiste nel controllare direttamente l'argomento rispetto all'oggetto `None` :

```
def foo(li=None):
    if li is None:
        li = []
    li.append(1)
    print(li)

foo()
# Out: [1]
```

Elenco di moltiplicazione e riferimenti comuni

Si consideri il caso di creare una struttura ad elenco nidificata moltiplicando:

```
li = [[]] * 3
print(li)
# Out: [[], [], []]
```

A prima vista, penseremmo di avere un elenco contenente 3 diverse liste annidate. Proviamo ad aggiungere `1` al primo:

```
li[0].append(1)
print(li)
# Out: [[1], [1], [1]]
```

`1` stato aggiunto a tutte le liste in `li` .

Il motivo è che `[[]] * 3` non crea una `list` di 3 `list` diverse. Piuttosto, crea una `list` contenente 3 riferimenti allo stesso oggetto `list` . Come tale, quando aggiungiamo a `li[0]` il cambiamento è visibile in tutti i sottoelementi di `li` . Questo è l'equivalente di:

```
li = []
element = [[]]
li = element + element + element
```

```
print(li)
# Out: [[], [], []]
element.append(1)
print(li)
# Out: [[1], [1], [1]]
```

Questo può essere ulteriormente confermato se stampiamo gli indirizzi di memoria della `list` contenuta usando `id` :

```
li = [[]] * 3
print([id(inner_list) for inner_list in li])
# Out: [6830760, 6830760, 6830760]
```

La soluzione è creare le liste interne con un ciclo:

```
li = [[] for _ in range(3)]
```

Invece di creare una singola `list` e quindi fare 3 riferimenti ad essa, ora creiamo 3 diversi elenchi distinti. Questo, ancora, può essere verificato usando la funzione `id` :

```
print([id(inner_list) for inner_list in li])
# Out: [6331048, 6331528, 6331488]
```

Puoi anche farlo. Fa sì che venga creata una nuova lista vuota in ogni chiamata `append` .

```
>>> li = []
>>> li.append([])
>>> li.append([])
>>> li.append([])
>>> for k in li: print(id(k))
...
4315469256
4315564552
4315564808
```

Non utilizzare l'indice per eseguire il loop su una sequenza.

Non:

```
for i in range(len(tab)):
    print(tab[i])
```

Fai :

```
for elem in tab:
    print(elem)
```

`for` automatizzare la maggior parte delle operazioni di iterazione per te.

Usa enumerare se hai davvero bisogno sia dell'indice che dell'elemento .

```
for i, elem in enumerate(tab):
     print((i, elem))
```

Fai attenzione quando usi "==" per controllare True o False

```
if (var == True):
    # this will execute if var is True or 1, 1.0, 1L

if (var != True):
    # this will execute if var is neither True nor 1

if (var == False):
    # this will execute if var is False or 0 (or 0.0, 0L, 0j)

if (var == None):
    # only execute if var is None

if var:
    # execute if var is a non-empty string/list/dictionary/tuple, non-0, etc

if not var:
    # execute if var is "", {}, [], (), 0, None, etc.

if var is True:
    # only execute if var is boolean True, not 1

if var is False:
    # only execute if var is boolean False, not 0

if var is None:
    # same as var == None
```

Non controllare se puoi, fallo e gestisci l'errore

Pythonistas di solito dice "È più facile chiedere perdono che il permesso".

Non:

```
if os.path.isfile(file_path):
    file = open(file_path)
else:
    # do something
```

Fare:

```
try:
    file = open(file_path)
except OSError as e:
    # do something
```

O ancora meglio con `Python 2.6+` :

```
with open(file_path) as file:
```

È molto meglio perché è molto più generico. Puoi applicare `try/except` a quasi tutto. Non è

necessario preoccuparsi di cosa fare per prevenirlo, basta preoccuparsi dell'errore che si sta rischiando.

Non controllare contro il tipo

Python è digitato in modo dinamico, quindi il controllo per il tipo ti fa perdere la flessibilità. Invece, usa la digitazione anatra controllando il comportamento. Se ti aspetti una stringa in una funzione, usa `str()` per convertire qualsiasi oggetto in una stringa. Se ti aspetti una lista, usa `list()` per convertire qualsiasi iterabile in una lista.

Non:

```
def foo(name):
    if isinstance(name, str):
        print(name.lower())

def bar(listing):
    if isinstance(listing, list):
        listing.extend((1, 2, 3))
        return ", ".join(listing)
```

Fare:

```
def foo(name) :
    print(str(name).lower())

def bar(listing) :
    l = list(listing)
    l.extend((1, 2, 3))
    return ", ".join(l)
```

Usando l'ultima modalità, `foo` accetterà qualsiasi oggetto. `bar` accetterà corde, tuple, set, elenchi e molto altro. A buon mercato ASCIUTTO.

Non mescolare spazi e tabulazioni

Usa *oggetto* come primo genitore

È complicato, ma ti morderà man mano che il tuo programma cresce. Esistono classi vecchie e nuove in `Python 2.x` I vecchi sono, bene, vecchi. Mancano di alcune funzionalità e possono avere un comportamento scomodo con l'ereditarietà. Per essere utilizzabile, qualsiasi classe deve essere del "nuovo stile". Per fare ciò, ereditarlo `object` .

Non:

```
class Father:
    pass

class Child(Father):
    pass
```

Fare:

```
class Father(object):
    pass


class Child(Father):
    pass
```

In `Python 3.x` tutte le classi sono di nuovo stile, quindi non è necessario farlo.

Non inizializzare gli attributi della classe al di fuori del metodo init

Persone provenienti da altre lingue trovano allettante perché è ciò che fai in Java o in PHP. Scrivi il nome della classe, quindi elenca i tuoi attributi e assegna loro un valore predefinito. Sembra funzionare in Python, tuttavia, questo non funziona come pensi. In questo modo verranno impostati gli attributi di classe (attributi statici), quindi quando tenterai di ottenere l'attributo dell'oggetto, ti darà il suo valore a meno che non sia vuoto. In tal caso restituirà gli attributi della classe. Implica due grandi rischi:

- Se l'attributo della classe è cambiato, allora il valore iniziale è cambiato.
- Se imposti un oggetto mutabile come valore predefinito, otterrai lo stesso oggetto condiviso tra le istanze.

Non (a meno che tu non voglia statico):

```
class Car(object):
    color = "red"
    wheels = [Wheel(), Wheel(), Wheel(), Wheel()]
```

Fare :

```
class Car(object):
    def __init__(self):
        self.color = "red"
        self.wheels = [Wheel(), Wheel(), Wheel(), Wheel()]
```

Intero e identità di stringa

Python utilizza la memorizzazione nella cache interna per un intervallo di numeri interi per ridurre il sovraccarico non necessario dalla loro ripetuta creazione.

In effetti, questo può portare a comportamenti confusi quando si confrontano le identità dei numeri interi:

```
>>> -8 is (-7 - 1)
False
>>> -3 is (-2 - 1)
True
```

e, usando un altro esempio:

```
>>> (255 + 1) is (255 + 1)
True
>>> (256 + 1) is (256 + 1)
False
```

Aspetta cosa?

Possiamo vedere che l'operazione di identità `is` resa `True` per alcuni interi (`-3` , `256`) ma non per altri (`-8` , `257`).

Per essere più specifici, gli interi nell'intervallo `[-5, 256]` sono memorizzati internamente nella cache durante l'avvio dell'interprete e vengono creati una sola volta. Come tali, essi sono **identici** e confrontando le loro identità con `is` rendimenti `True` ; gli interi al di fuori di questo intervallo sono (di solito) creati al volo e le loro identità si confrontano con `False` .

Questo è un errore comune poiché questo è un intervallo comune per i test, ma spesso abbastanza, il codice fallisce nel processo di gestione temporanea successivo (o peggio - produzione) senza una ragione apparente dopo aver lavorato perfettamente nello sviluppo.

La soluzione è **confrontare sempre i valori usando l'** operatore di **uguaglianza** (`==`) e **non** l'operatore di identità (`is`).

Python mantiene anche i riferimenti alle stringhe di uso comune e può comportare un comportamento simile e confuso quando si confrontano le identità (cioè l'utilizzo di `is`) delle stringhe.

```
>>> 'python' is 'py' + 'thon'
True
```

La stringa `'python'` è comunemente usata, quindi Python ha un oggetto che usa tutti i riferimenti alla stringa `'python'` .

Per stringhe non comuni, il confronto dell'identità non riesce anche se le stringhe sono uguali.

```
>>> 'this is not a common string' is 'this is not' + ' a common string'
False
>>> 'this is not a common string' == 'this is not' + ' a common string'
True
```

Quindi, proprio come la regola per Interi, **confronta sempre i valori stringa usando l'** operatore di **uguaglianza** (`==`) e **non** l'operatore di identità (`is`).

Accesso agli attributi letterali di int

Potresti aver sentito che tutto in Python è un oggetto, anche letterale. Ciò significa che, ad esempio, `7` è anche un oggetto, il che significa che ha degli attributi. Ad esempio, uno di questi attributi è il `bit_length` . Restituisce la quantità di bit necessari per rappresentare il valore su cui è chiamato.

```
x = 7
x.bit_length()
# Out: 3
```

Vedendo che il codice sopra funziona, potresti pensare intuitivamente che anche `7.bit_length()` funzionerebbe, solo per scoprire che genera un `SyntaxError` . Perché? perché l'interprete deve distinguere tra un accesso di attributo e un numero mobile (ad esempio `7.2` o `7.bit_length()`). Non può, ed è per questo che viene sollevata un'eccezione.

Esistono alcuni modi per accedere agli attributi di un letterale `int` :

```
# parenthesis
(7).bit_length()
# a space
7 .bit_length()
```

L'uso di due punti (come questo `7..bit_length()`) non funziona in questo caso, poiché ciò crea un valore letterale `float` e i float non hanno il metodo `bit_length()` .

Questo problema non esiste quando si accede agli attributi letterali `float` poiché l'interperter è abbastanza "intelligente" da sapere che un letterale `float` non può contenere due `.` , per esempio:

```
7.2.as_integer_ratio()
# Out: (8106479329266893, 1125899906842624)
```

Concatenamento di o operatore

Durante il test per uno dei numerosi confronti di uguaglianza:

```
if a == 3 or b == 3 or c == 3:
```

si è tentati di abbreviare questo a

```
if a or b or c == 3: # Wrong
```

Questo è sbagliato; l'operatore `or` ha una precedenza inferiore a `==` , quindi l'espressione sarà valutata come `if (a) or (b) or (c == 3):` Il modo corretto è il controllo esplicito di tutte le condizioni:

```
if a == 3 or b == 3 or c == 3:  # Right Way
```

In alternativa, è possibile utilizzare la funzione `any()` integrata al posto di concatenati `or` operatori:

```
if any([a == 3, b == 3, c == 3]): # Right
```

Oppure, per renderlo più efficiente:

```
if any(x == 3 for x in (a, b, c)): # Right
```

Oppure, per renderlo più breve:

```
if 3 in (a, b, c): # Right
```

Qui, usiamo l' `in` all'operatore di verificare se il valore è presente in una tupla contenente i valori che vogliamo confrontare con.

Allo stesso modo, non è corretto scrivere

```
if a == 1 or 2 or 3:
```

che dovrebbe essere scritto come

```
if a in (1, 2, 3):
```

sys.argv [0] è il nome del file in esecuzione

Il primo elemento di `sys.argv[0]` è il nome del file python in esecuzione. Gli elementi rimanenti sono gli argomenti dello script.

```
# script.py
import sys

print(sys.argv[0])
print(sys.argv)
```

```
$ python script.py
=> script.py
=> ['script.py']

$ python script.py fizz
=> script.py
=> ['script.py', 'fizz']

$ python script.py fizz buzz
=> script.py
=> ['script.py', 'fizz', 'buzz']
```

I dizionari non sono ordinati

Ci si potrebbe aspettare che un dizionario Python sia ordinato per chiavi come, per esempio, una `std::map` C ++, ma questo non è il caso:

```
myDict = {'first': 1, 'second': 2, 'third': 3}
print(myDict)
# Out: {'first': 1, 'second': 2, 'third': 3}

print([k for k in myDict])
# Out: ['second', 'third', 'first']
```

Python non ha alcuna classe built-in che ordina automaticamente i suoi elementi per chiave.

Tuttavia, se l'ordinamento non è obbligatorio e vuoi che il tuo dizionario ricordi l'ordine di inserimento delle sue coppie chiave / valore, puoi utilizzare `collections.OrderedDict` :

```
from collections import OrderedDict

oDict = OrderedDict([('first', 1), ('second', 2), ('third', 3)])

print([k for k in oDict])
# Out: ['first', 'second', 'third']
```

Tieni presente che l'inizializzazione di `OrderedDict` con un dizionario standard non `OrderedDict` in alcun modo il dizionario per te. Tutto ciò che questa struttura fa è *preservare* l'ordine di inserimento della chiave.

L'implementazione dei dizionari è stata modificata in Python 3.6 per migliorare il consumo di memoria. Un effetto collaterale di questa nuova implementazione è che mantiene anche l'ordine degli argomenti delle parole chiave passati a una funzione:

Python 3.x 3.6

```
def func(**kw): print(kw.keys())

func(a=1, b=2, c=3, d=4, e=5)
dict_keys(['a', 'b', 'c', 'd', 'e']) # expected order
```

> **Avvertenza** : **fai** attenzione che " *l'aspetto di conservazione di questa nuova implementazione è considerato un dettaglio di implementazione e non dovrebbe essere invocato* " , poiché potrebbe cambiare in futuro.

Global Interpreter Lock (GIL) e thread di blocco

Molto è stato scritto su GIL di Python . A volte può causare confusione quando si tratta di applicazioni multi-thread (da non confondere con multiprocesso).

Ecco un esempio:

```
import math
from threading import Thread

def calc_fact(num):
    math.factorial(num)

num = 600000
t = Thread(target=calc_fact, daemon=True, args=[num])
print("About to calculate: {}!".format(num))
t.start()
print("Calculating...")
t.join()
print("Calculated")
```

Ci si aspetterebbe di vedere `Calculating...` stampato immediatamente dopo l'avvio del thread, volevamo che il calcolo si verificasse in un nuovo thread dopo tutto! Ma in realtà, vedi che viene stampato dopo il calcolo è completo. Questo perché il nuovo thread si basa su una funzione C (`math.factorial`) che bloccherà GIL durante l'esecuzione.

Ci sono un paio di modi per aggirare questo. Il primo è implementare la funzione fattoriale in Python nativo. Ciò consentirà al thread principale di prendere il controllo mentre sei all'interno del tuo loop. Il rovescio della medaglia è che questa soluzione sarà **molto** più lenta, dal momento che non usiamo più la funzione C.

```
def calc_fact(num):
    """ A slow version of factorial in native Python """
    res = 1
    while num >= 1:
        res = res * num
        num -= 1
    return res
```

Puoi anche `sleep` per un periodo di tempo prima di iniziare l'esecuzione. Nota: questo non permetterà al tuo programma di interrompere il calcolo che avviene all'interno della funzione C, ma permetterà al tuo thread principale di continuare dopo lo spawn, che è ciò che potresti aspettarti.

```
def calc_fact(num):
    sleep(0.001)
    math.factorial(num)
```

Perdita variabile nella comprensione delle liste e nei cicli

Considera la seguente lista di comprensione

Python 2.x 2.7

```
i = 0
a = [i for i in range(3)]
print(i) # Outputs 2
```

Ciò si verifica solo in Python 2 a causa del fatto che la comprensione della lista "perde" la variabile di controllo del ciclo nello scope circostante (source). Questo comportamento può portare a bug difficili da trovare ed ***è stato corretto in Python 3*** .

Python 3.x 3.0

```
i = 0
a = [i for i in range(3)]
print(i) # Outputs 0
```

Allo stesso modo, i cicli for non hanno scope privato per la loro variabile di iterazione

```
i = 0
```

```
for i in range(3):
    pass
print(i) # Outputs 2
```

Questo tipo di comportamento si verifica sia in Python 2 che in Python 3.

Per evitare problemi con variabili che perdono, utilizzare nuove variabili nelle list comprehensions e nei loop appropriati.

Ritorno multiplo

La funzione xyz restituisce due valori aeb:

```
def xyz():
  return a, b
```

Il codice che chiama xyz memorizza il risultato in una variabile assumendo che xyz restituisca un solo valore:

```
t = xyz()
```

Il valore di `t` è in realtà una tupla (a, b) quindi qualsiasi azione su `t` assumendo che non si tratti di una tupla potrebbe fallire nel **profondo** del codice con un **errore** imprevisto sulle tuple.

> TypeError: type tuple does not define ... metodo

La soluzione sarebbe quella di fare:

```
a, b = xyz()
```

I principianti avranno difficoltà a trovare la ragione di questo messaggio leggendo solo il messaggio di errore tupla!

Chiavi Python JSON

```
my_var = 'bla';
api_key = 'key';
...lots of code here...
params = {"language": "en", my_var: api_key}
```

Se sei abituato a JavaScript, la valutazione variabile nei dizionari Python non sarà quella che ti aspetti che sia. Questa dichiarazione in JavaScript risulterebbe nell'oggetto `params` come segue:

```
{
    "language": "en",
    "my_var": "key"
}
```

In Python, tuttavia, risulterebbe il seguente dizionario:

```
{
    "language": "en",
    "bla": "key"
}
```

`my_var` viene valutato e il suo valore è usato come chiave.

Leggi Insidie comuni online: https://riptutorial.com/it/python/topic/3553/insidie---comuni

Capitolo 90: Interfaccia Web Server Gateway (WSGI)

Parametri

Parametro	Dettagli
start_response	Una funzione utilizzata per elaborare l'inizio

Examples

Oggetto server (metodo)

Al nostro oggetto server viene assegnato un parametro 'applicazione' che può essere qualsiasi oggetto di applicazione richiamabile (vedere altri esempi). Scrive prima le intestazioni, quindi il corpo dei dati restituiti dalla nostra applicazione all'output standard del sistema.

```
import os, sys

def run(application):
    environ['wsgi.input']        = sys.stdin
    environ['wsgi.errors']       = sys.stderr

    headers_set = []
    headers_sent = []

    def write (data):
        """
        Writes header data from 'start_response()' as well as body data from 'response'
        to system standard output.
        """
        if not headers_set:
            raise AssertionError("write() before start_response()")

        elif not headers_sent:
            status, response_headers = headers_sent[:] = headers_set
            sys.stdout.write('Status: %s\r\n' % status)
            for header in response_headers:
                sys.stdout.write('%s: %s\r\n' % header)
            sys.stdout.write('\r\n')

        sys.stdout.write(data)
        sys.stdout.flush()

    def start_response(status, response_headers):
        """ Sets headers for the response returned by this server."""
        if headers_set:
            raise AssertionError("Headers already set!")

        headers_set[:] = [status, response_headers]
        return write
```

```
    # This is the most important piece of the 'server object'
    # Our result will be generated by the 'application' given to this method as a parameter
    result = application(environ, start_response)
    try:
        for data in result:
            if data:
                write(data)          # Body isn't empty send its data to 'write()'
        if not headers_sent:
            write('')                # Body is empty, send empty string to 'write()'
```

Leggi Interfaccia Web Server Gateway (WSGI) online:
https://riptutorial.com/it/python/topic/5315/interfaccia-web-server-gateway--wsgi-

Capitolo 91: Introduzione a RabbitMQ usando AMQPStorm

Osservazioni

L'ultima versione di AMQPStorm è disponibile su pypi oppure è possibile installarla tramite pip

```
pip install amqpstorm
```

Examples

Come consumare messaggi da RabbitMQ

Inizia con l'importazione della libreria.

```
from amqpstorm import Connection
```

Quando si consumano messaggi, dobbiamo prima definire una funzione per gestire i messaggi in arrivo. Questa può essere una funzione chiamabile e deve prendere un oggetto messaggio o una tupla di messaggi (in base al parametro `to_tuple` definito in `start_consuming`).

Oltre all'elaborazione dei dati dal messaggio in arrivo, dovremo anche riconoscere o rifiutare il messaggio. Questo è importante, poiché è necessario far sapere a RabbitMQ che abbiamo ricevuto e elaborato correttamente il messaggio.

```
def on_message(message):
    """This function is called on message received.

    :param message: Delivered message.
    :return:
    """
    print("Message:", message.body)

    # Acknowledge that we handled the message without any issues.
    message.ack()

    # Reject the message.
    # message.reject()

    # Reject the message, and put it back in the queue.
    # message.reject(requeue=True)
```

Quindi dobbiamo impostare la connessione al server RabbitMQ.

```
connection = Connection('127.0.0.1', 'guest', 'guest')
```

Dopodiché abbiamo bisogno di creare un canale. Ogni connessione può avere più canali e, in

generale, quando si eseguono attività multi-thread, è consigliabile (ma non obbligatorio) avere uno per thread.

```
channel = connection.channel()
```

Una volta configurato il nostro canale, dobbiamo far sapere a RabbitMQ che vogliamo iniziare a consumare messaggi. In questo caso useremo la nostra funzione `on_message` precedentemente definita per gestire tutti i nostri messaggi consumati.

La coda che ascolteremo sul server RabbitMQ sarà `simple_queue` , e stiamo anche dicendo a RabbitMQ che avremo riconosciuto tutti i messaggi in arrivo una volta che avremo finito con loro.

```
channel.basic.consume(callback=on_message, queue='simple_queue', no_ack=False)
```

Infine, è necessario avviare il loop IO per avviare l'elaborazione dei messaggi consegnati dal server RabbitMQ.

```
channel.start_consuming(to_tuple=False)
```

Come pubblicare messaggi su RabbitMQ

Inizia con l'importazione della libreria.

```
from amqpstorm import Connection
from amqpstorm import Message
```

Quindi è necessario aprire una connessione al server RabbitMQ.

```
connection = Connection('127.0.0.1', 'guest', 'guest')
```

Dopodiché abbiamo bisogno di creare un canale. Ogni connessione può avere più canali e, in generale, quando si eseguono attività multi-thread, è consigliabile (ma non obbligatorio) avere uno per thread.

```
channel = connection.channel()
```

Una volta impostato il nostro canale, possiamo iniziare a preparare il nostro messaggio.

```
# Message Properties.
properties = {
    'content_type': 'text/plain',
    'headers': {'key': 'value'}
}

# Create the message.
message = Message.create(channel=channel, body='Hello World!', properties=properties)
```

Ora possiamo pubblicare il messaggio semplicemente chiamando la `publish` e fornendo un `routing_key`

. In questo caso, invieremo il messaggio a una coda chiamata `simple_queue` .

```
message.publish(routing_key='simple_queue')
```

Come creare una coda in ritardo in RabbitMQ

Per prima cosa dobbiamo impostare due canali di base, uno per la coda principale e uno per la coda di ritardo. Nel mio esempio alla fine, includo un paio di flag aggiuntivi che non sono richiesti, ma rendono il codice più affidabile; come `confirm delivery` , `delivery_mode` e `durable` . Puoi trovare maggiori informazioni su questi nel manuale RabbitMQ.

Dopo aver impostato i canali, aggiungiamo un'associazione al canale principale che possiamo usare per inviare messaggi dal canale di ritardo alla nostra coda principale.

```
channel.queue.bind(exchange='amq.direct', routing_key='hello', queue='hello')
```

Successivamente dobbiamo configurare il nostro canale di ritardo per inoltrare i messaggi alla coda principale una volta scaduti.

```
delay_channel.queue.declare(queue='hello_delay', durable=True, arguments={
    'x-message-ttl': 5000,
    'x-dead-letter-exchange': 'amq.direct',
    'x-dead-letter-routing-key': 'hello'
})
```

- x-message-ttl *(Message - Time To Live)*

  Questo è normalmente usato per rimuovere automaticamente i vecchi messaggi nella coda dopo una durata specifica, ma aggiungendo due argomenti opzionali possiamo cambiare questo comportamento, e invece questo parametro determina in millisecondi per quanto tempo i messaggi rimarranno nella coda di delay.

- x--alternativo-chiave lettera morta

  Questa variabile ci consente di trasferire il messaggio in una coda diversa una volta scaduti, invece del comportamento predefinito di rimuoverlo completamente.

- x-dead-letter-scambio

  Questa variabile determina quale Exchange ha usato per trasferire il messaggio da hello_delay a hello queue.

Pubblicazione sulla coda di ritardo

Quando abbiamo finito di impostare tutti i parametri di base di Pika, semplicemente invii un messaggio alla coda di ritardo usando la pubblicazione di base.

```
delay_channel.basic.publish(exchange='',
                            routing_key='hello_delay',
```

```
                           body='test',
                           properties={'delivery_mod': 2})
```

Una volta eseguito lo script, dovresti vedere le seguenti code create nel tuo modulo di gestione RabbitMQ.

Overview					Messages				
Name	**Exclusive**	Parameters	**Policy**	**Status**	**Ready**	**Unacked**	**Total**	**incoming**	**deliv**
hello		D		Idle	1	0	1		
hello_delay		TTL DLX DLK D		Idle	0	0	0	0.00/s	

Esempio.

```
from amqpstorm import Connection

connection = Connection('127.0.0.1', 'guest', 'guest')

# Create normal 'Hello World' type channel.
channel = connection.channel()
channel.confirm_deliveries()
channel.queue.declare(queue='hello', durable=True)

# We need to bind this channel to an exchange, that will be used to transfer
# messages from our delay queue.
channel.queue.bind(exchange='amq.direct', routing_key='hello', queue='hello')

# Create our delay channel.
delay_channel = connection.channel()
delay_channel.confirm_deliveries()

# This is where we declare the delay, and routing for our delay channel.
delay_channel.queue.declare(queue='hello_delay', durable=True, arguments={
    'x-message-ttl': 5000, # Delay until the message is transferred in milliseconds.
    'x-dead-letter-exchange': 'amq.direct', # Exchange used to transfer the message from A to
B.
    'x-dead-letter-routing-key': 'hello' # Name of the queue we want the message transferred
to.
})

delay_channel.basic.publish(exchange='',
                            routing_key='hello_delay',
                            body='test',
                            properties={'delivery_mode': 2})

print("[x] Sent")
```

Leggi Introduzione a RabbitMQ usando AMQPStorm online:
https://riptutorial.com/it/python/topic/3373/introduzione-a-rabbitmq-usando-amqpstorm

Capitolo 92: Iterables e Iterators

Examples

Iterator vs Iterable vs Generator

Un **iterabile** è un oggetto che può restituire un **iteratore** . Qualsiasi oggetto con stato che ha un metodo `__iter__` e restituisce un iteratore è un iterabile. Potrebbe anche essere un oggetto *senza* stato che implementa un metodo `__getitem__` . - Il metodo può prendere indici (partendo da zero) e generare un `IndexError` quando gli indici non sono più validi.

La classe `str` di Python è un esempio di iterable `__getitem__` .

Un **Iterator** è un oggetto che produce il valore successivo in una sequenza quando si chiama `next(*object*)` su qualche oggetto. Inoltre, qualsiasi oggetto con un metodo `__next__` è un iteratore. Un iteratore solleva `StopIteration` dopo aver esaurito l'iteratore e *non può* essere riutilizzato a questo punto.

Classi Iterable:

Le classi Iterable definiscono un `__iter__` e `__next__` . Esempio di una classe iterabile:

```
class MyIterable:

    def __iter__(self):

         return self

    def __next__(self):
         #code

#Classic iterable object in older versions of python, __getitem__ is still supported...
class MySequence:

    def __getitem__(self, index):

         if (condition):
             raise IndexError
         return (item)

 #Can produce a plain `iterator` instance by using iter(MySequence())
```

Cercando di creare un'istanza della classe astratta dal modulo delle `collections` per vederlo meglio.

Esempio:

Python 2.x 2.3

```
import collections
>>> collections.Iterator()
```

```
>>> TypeError: Cant instantiate abstract class Iterator with abstract methods next
```

Python 3.x 3.0

```
>>> TypeError: Cant instantiate abstract class Iterator with abstract methods __next__
```

Gestisci la compatibilità con Python 3 per le classi iterabili in Python 2 effettuando le seguenti operazioni:

Python 2.x 2.3

```
class MyIterable(object): #or collections.Iterator, which I'd recommend....

     ....

     def __iter__(self):

          return self

     def next(self): #code

     __next__ = next
```

Entrambi sono ora iteratori e possono essere collegati in loop:

```
ex1 = MyIterableClass()
ex2 = MySequence()

for (item) in (ex1): #code
for (item) in (ex2): #code
```

I generatori sono semplici modi per creare iteratori. Un generatore *è* un iteratore e un iteratore è un iterabile.

Cosa può essere iterabile

Iterable può essere qualsiasi cosa per cui gli articoli vengono ricevuti *uno per uno, solo in avanti* . Le collezioni Python integrate sono iterabili:

```
[1, 2, 3]     # list, iterate over items
(1, 2, 3)     # tuple
{1, 2, 3}     # set
{1: 2, 3: 4}  # dict, iterate over keys
```

I generatori restituiscono iterabili:

```
def foo():  # foo isn't iterable yet...
    yield 1

res = foo()  # ...but res already is
```

Iterazione su intero iterabile

```
s = {1, 2, 3}

# get every element in s
for a in s:
    print a  # prints 1, then 2, then 3

# copy into list
l1 = list(s)  # l1 = [1, 2, 3]

# use list comprehension
l2 = [a * 2 for a in s if a > 2]  # l2 = [6]
```

Verifica solo un elemento in iterabile

Usa decompressione per estrarre il primo elemento e assicurarti che sia l'unico:

```
a, = iterable

def foo():
    yield 1

a, = foo()  # a = 1

nums = [1, 2, 3]
a, = nums  # ValueError: too many values to unpack
```

Estrarre i valori uno per uno

Inizia con `iter()` integrato per ottenere **iteratore** su iterabile e usa `next()` per ottenere gli elementi uno alla volta fino a quando `StopIteration` viene `StopIteration` significare la fine:

```
s = {1, 2}   # or list or generator or even iterator
i = iter(s)  # get iterator
a = next(i)  # a = 1
b = next(i)  # b = 2
c = next(i)  # raises StopIteration
```

Iterator non è rientranti!

```
def gen():
    yield 1

iterable = gen()
for a in iterable:
    print a

# What was the first item of iterable? No way to get it now.
# Only to get a new iterator
gen()
```

Leggi Iterables e Iterators online: https://riptutorial.com/it/python/topic/2343/iterables-e-iterators

Capitolo 93: kivy - Framework Python multipiattaforma per lo sviluppo NUI

introduzione

NUI: un'interfaccia utente naturale (NUI) è un sistema per l'interazione uomo-computer che l'utente opera attraverso azioni intuitive relative al comportamento umano naturale e quotidiano.

Kivy è una libreria Python per lo sviluppo di applicazioni multimediali multimediali multi-touch che possono essere installate su diversi dispositivi. Il multi-touch si riferisce alla capacità di una superficie sensibile al tocco (di solito un touch screen o un trackpad) per rilevare o rilevare l'input da due o più punti di contatto contemporaneamente.

Examples

Prima app

Per creare un'applicazione kivy

1. sottoclasse la classe **dell'app**
2. Implementa il metodo di **build** , che restituirà il widget.
3. Crea un'istanza della classe e invoca la **corsa** .

```
from kivy.app import App
from kivy.uix.label import Label

class Test(App):
    def build(self):
        return Label(text='Hello world')

if __name__ == '__main__':
    Test().run()
```

Spiegazione

```
from kivy.app import App
```

L'istruzione precedente importerà l' **app della** classe genitore. Questo sarà presente nella directory di installazione your_installtion_directory / kivy / app.py

```
from kivy.uix.label import Label
```

L'affermazione precedente importerà l' **etichetta** dell'elemento ux. Tutti gli elementi ux sono presenti nella directory di installazione your_installation_directory / kivy / uix /.

```
class Test(App):
```

L'affermazione sopra è per creare la tua app e il nome della classe sarà il nome della tua app. Questa classe viene ereditata dalla classe dell'app principale.

```
def build(self):
```

L'istruzione precedente sostituisce il metodo di build della classe app. Che restituirà il widget che deve essere mostrato quando avvierai l'app.

```
return Label(text='Hello world')
```

L'affermazione precedente è il corpo del metodo di compilazione. Sta restituendo l'etichetta con il suo testo **Hello world** .

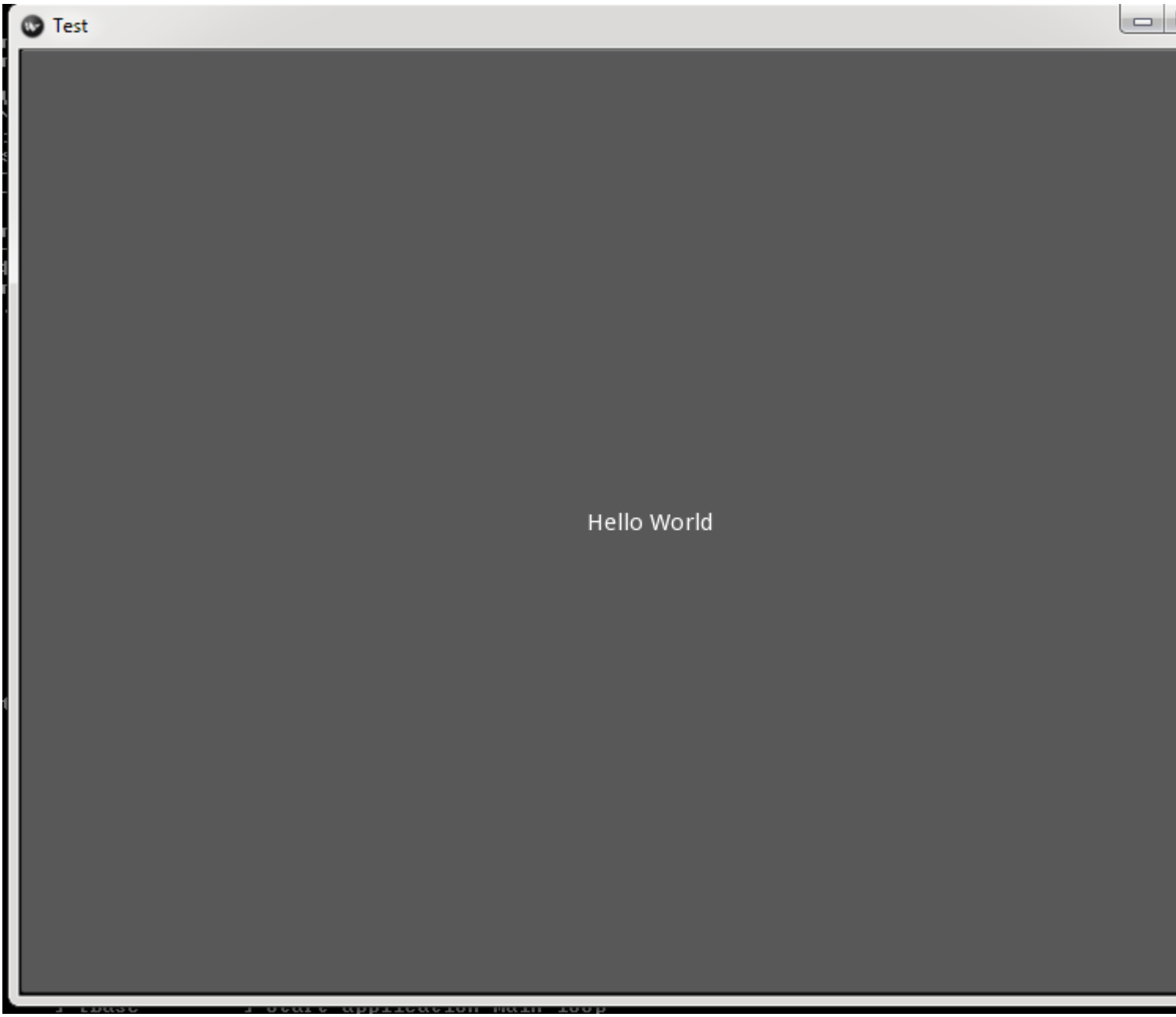
```
if __name__ == '__main__':
```

L'affermazione precedente è il punto di ingresso da cui l'interprete python avvia l'esecuzione della tua app.

```
Test().run()
```

L'affermazione precedente Inizializza la classe Test creando la sua istanza. E invoca la funzione della classe dell'app run ().

La tua app sarà simile alla seguente immagine.



Leggi kivy - Framework Python multipiattaforma per lo sviluppo NUI online: https://riptutorial.com/it/python/topic/10743/kivy---framework-python-multipiattaforma-per-lo-sviluppo-nui

Capitolo 94: La dichiarazione del passaggio

Sintassi

- passaggio

Osservazioni

Perché vorresti mai dire all'interprete di fare esplicitamente nulla? Python ha il requisito sintattico che i blocchi di codice (dopo `if` , `except` , `def` , `class` ecc.) Non possono essere vuoti.

Ma a volte un blocco di codice vuoto è utile di per sé. Un blocco di `class` vuoto può definire una nuova classe diversa, ad esempio un'eccezione che può essere rilevata. Un blocco vuoto `except` può essere il modo più semplice per esprimere "chiedere perdono dopo" se non c'era nulla da chiedere perdono. Se un iteratore fa tutto il sollevamento pesante, può essere utile un loop vuoto `for` eseguire semplicemente l'iteratore.

Pertanto, se non si suppone che accada in un blocco di codice, è necessario un `pass` per tale blocco per non produrre un `IndentationError` . In alternativa, qualsiasi affermazione (incluso solo un termine da valutare, come il letterale `Ellipsis` `...` o una stringa, più spesso una docstring) può essere usata, ma il `pass` chiarisce che effettivamente non dovrebbe accadere nulla, e non ha bisogno per essere effettivamente valutato e (almeno temporaneamente) memorizzato. Ecco una piccola raccolta annotata degli usi più frequenti del `pass` che incrociava la mia strada - insieme ad alcuni commenti sulla buona e cattiva pratica.

- Ignorando (tutto o) un certo tipo di `Exception` (esempio da `xml`):

```
try:
    self.version = "Expat %d.%d.%d" % expat.version_info
except AttributeError:
    pass # unknown
```

Nota: ignorare tutti i tipi di rilanci, come nel seguente esempio dei `pandas` , è generalmente considerato una cattiva pratica, perché `SystemExit` anche eccezioni che dovrebbero probabilmente essere passate al chiamante, ad esempio `KeyboardInterrupt` o `SystemExit` (o anche `HardwareIsOnFireError` - How do you know non si sta eseguendo su una casella personalizzata con errori specifici definiti, che alcune applicazioni chiamanti vorrebbero conoscere?).

```
try:
    os.unlink(filename_larry)
except:
    pass
```

Invece di usare almeno l' `except Error:` o in questo caso preferibilmente `except OSError:` è considerata una pratica molto migliore. Una rapida analisi di tutti i moduli python che ho

installato mi ha dato che oltre il 10% di tutti `except ...: pass` istruzioni di `except ...: pass` catturano tutte le eccezioni, quindi è ancora un pattern frequente nella programmazione Python.

- Derivazione di una classe di eccezioni che non aggiunge nuovi comportamenti (ad esempio in `scipy`):

```
class CompileError(Exception):
    pass
```

Allo stesso modo, le classi intese come classe base astratta hanno spesso un `__init__` vuoto esplicito o altri metodi che le sottoclassi dovrebbero derivare. (es. `pebl`)

```
class _BaseSubmittingController(_BaseController):
    def submit(self, tasks): pass
    def retrieve(self, deferred_results): pass
```

- Verificare che il codice funzioni correttamente per alcuni valori di test, senza preoccuparsi dei risultati (da `mpmath`):

```
for x, error in MDNewton(mp, f, (1,-2), verbose=0,
                         norm=lambda x: norm(x, inf)):
    pass
```

- Nelle definizioni di classi o funzioni, spesso una docstring è già presente come *istruzione obbligatoria* da eseguire come unica cosa nel blocco. In questi casi, il blocco può contenere un `pass` *in aggiunta* alla docstring per poter dire "Questo è in effetti inteso a non fare nulla.", Ad esempio in `pebl` :

```
class ParsingError(Exception):
    """Error encountered while parsing an ill-formed datafile."""
    pass
```

- In alcuni casi, `pass` è usato come segnaposto per dire "Questo metodo / classe / if-block / ... non è ancora stato implementato, ma questo sarà il posto giusto per farlo", anche se personalmente preferisco il letterale `Ellipsis ...` (NOTA: solo python-3) per distinguere strettamente tra questo e il "no-op" intenzionale nell'esempio precedente. Ad esempio, se scrivo un modello a grandi linee, potrei scrivere

```
def update_agent(agent):
    ...
```

dove altri potrebbero avere

```
def update_agent(agent):
    pass
```

prima

```
def time_step(agents):
    for agent in agents:
        update_agent(agent)
```

come promemoria per `update_agent` funzione `update_agent` in un secondo momento, ma esegui alcuni test già per vedere se il resto del codice si comporta come previsto. (Una terza opzione per questo caso è il `raise NotImplementedError` . Ciò è utile in particolare per due casi: o *"Questo metodo astratto dovrebbe essere implementato da ogni sottoclasse, non esiste un modo generico per definirlo in questa classe base"* o *"Questa funzione , con questo nome, non è ancora stato implementato in questa versione, ma questo è come sarà la sua firma "*)

Examples

Ignora un'eccezione

```
try:
    metadata = metadata['properties']
except KeyError:
    pass
```

Crea una nuova eccezione che può essere catturata

```
class CompileError(Exception):
    pass
```

Leggi La dichiarazione del passaggio online: https://riptutorial.com/it/python/topic/6891/la-dichiarazione-del-passaggio

Capitolo 95: La funzione di stampa

Examples

Informazioni di base sulla stampa

In Python 3 e versioni successive, la `print` è una funzione piuttosto che una parola chiave.

```
print('hello world!')
# out: hello world!

foo = 1
bar = 'bar'
baz = 3.14

print(foo)
# out: 1
print(bar)
# out: bar
print(baz)
# out: 3.14
```

È anche possibile passare un numero di parametri per `print` :

```
print(foo, bar, baz)
# out: 1 bar 3.14
```

Un altro modo per `print` più parametri è usare un `+`

```
print(str(foo) + " " + bar + " " + str(baz))
# out: 1 bar 3.14
```

Quello che dovresti fare attenzione quando usi `+` per stampare più parametri, è che il tipo dei parametri dovrebbe essere lo stesso. Cercando di stampare l'esempio sopra senza il cast to `string` primo si darebbe un errore, perché proverebbe ad aggiungere il numero `1` alla stringa `"bar"` e aggiungerlo al numero `3.14` .

```
# Wrong:
# type:int  str  float
print(foo + bar + baz)
# will result in an error
```

Questo perché il contenuto della `print` verrà valutato per primo:

```
print(4 + 5)
# out: 9
print("4" + "5")
# out: 45
print([4] + [5])
# out: [4, 5]
```

Altrimenti, usare un `+` può essere molto utile per un utente per leggere l'output delle variabili Nell'esempio seguente l'output è molto facile da leggere!

Lo script qui sotto lo dimostra

```
import random
#telling python to include a function to create random numbers
randnum = random.randint(0, 12)
#make a random number between 0 and 12 and assign it to a variable
print("The randomly generated number was - " + str(randnum))
```

È possibile impedire la `print` funzione dalla stampa automaticamente una nuova riga utilizzando la `end` del parametro:

```
print("this has no newline at the end of it... ", end="")
print("see?")
# out: this has no newline at the end of it... see?
```

Se vuoi scrivere su un file, puoi passarlo come `file` parametri:

```
with open('my_file.txt', 'w+') as my_file:
    print("this goes to the file!", file=my_file)
```

questo va al file!

Stampa i parametri

Puoi fare di più che stampare semplicemente testo. `print` ha anche diversi parametri per aiutarti.

Argomento `sep` : posiziona una stringa tra gli argomenti.

Hai bisogno di stampare un elenco di parole separate da una virgola o da qualche altra stringa?

```
>>> print('apples','bannas', 'cherries', sep=', ')
apple, bannas, cherries
>>> print('apple','banna', 'cherries', sep=', ')
apple, banna, cherries
>>>
```

`end` argomento: utilizzare qualcosa di diverso da una nuova riga alla fine

Senza l'argomento `end` , tutte `print()` funzioni `print()` scrivono una riga e poi si spostano all'inizio della riga successiva. Puoi cambiarlo per non fare nulla (usa una stringa vuota di ""), o una doppia spaziatura tra i paragrafi usando due newline.

```
>>> print("<a", end=''); print(" class='jidn'" if 1 else "", end=''); print("/>")
<a class='jidn'/>
>>> print("paragraph1", end="\n\n"); print("paragraph2")
paragraph1

paragraph2
```

```
>>>
```

`file` argomenti: invia l'output a un posto diverso da sys.stdout.

Ora puoi inviare il tuo testo a stdout, a un file o a StringIO e non ti interessa quale ti viene dato. Se cova come un file, funziona come un file.

```
>>> def sendit(out, *values, sep=' ', end='\n'):
...     print(*values, sep=sep, end=end, file=out)
...
>>> sendit(sys.stdout, 'apples', 'bannas', 'cherries', sep='\t')
apples    bannas    cherries
>>> with open("delete-me.txt", "w+") as f:
...    sendit(f, 'apples', 'bannas', 'cherries', sep=' ', end='\n')
...
>>> with open("delete-me.txt", "rt") as f:
...     print(f.read())
...
apples bannas cherries

>>>
```

C'è un quarto parametro `flush` che forzatamente sciacquare il flusso.

Leggi La funzione di stampa online: https://riptutorial.com/it/python/topic/1360/la-funzione-di-stampa

Capitolo 96: La variabile speciale __name__

introduzione

La variabile speciale `__name__` viene utilizzata per verificare se un file è stato importato come modulo o meno e per identificare una funzione, classe, oggetto modulo con il loro attributo `__name__` .

Osservazioni

La variabile speciale Python `__name__` è impostata sul nome del modulo contenitore. Al livello più alto (come nell'interprete interattivo o nel file principale) è impostato su `'__main__'` . Questo può essere usato per eseguire un blocco di istruzioni se un modulo viene eseguito direttamente anziché importato.

L'attributo speciale correlato `obj.__name__` si trova su classi, moduli e funzioni importati *(compresi i metodi)* e fornisce il nome dell'oggetto quando definito.

Examples

__name__ == '__main__'

La variabile speciale `__name__` non è impostata dall'utente. Viene in gran parte utilizzato per verificare se il modulo viene eseguito da solo o eseguito perché è stata eseguita `import` . Per evitare che il tuo modulo esegua determinate parti del suo codice quando viene importato, controlla `if __name__ == '__main__'` .

Lascia che **module_1.py** sia lungo una sola riga:

```
import module2.py
```

E vediamo cosa succede, a seconda del **modulo2.py**

Situazione 1

module2.py

```
print('hello')
```

L'esecuzione di **module1.py** stamperà `hello`
Eseguendo **module2.py** verrà stampato `hello`

Situazione 2

module2.py

```
if __name__ == '__main__':
    print('hello')
```

L'esecuzione di **module1.py** non stampa nulla
Eseguendo **module2.py** verrà stampato `hello`

function_class_or_module .__ name__

L'attributo speciale `__name__` di una funzione, classe o modulo è una stringa che contiene il suo nome.

```
import os

class C:
    pass

def f(x):
    x += 2
    return x


print(f)
# <function f at 0x029976B0>
print(f.__name__)
# f

print(C)
# <class '__main__.C'>
print(C.__name__)
# C

print(os)
# <module 'os' from '/spam/eggs/'>
print(os.__name__)
# os
```

L'attributo `__name__` non è, tuttavia, il nome della variabile che fa riferimento alla classe, al metodo o alla funzione, piuttosto è il nome che gli viene dato quando definito.

```
def f():
    pass

print(f.__name__)
# f - as expected

g = f
print(g.__name__)
# f - even though the variable is named g, the function is still named f
```

Questo può essere usato, tra gli altri, per il debug:

```
def enter_exit_info(func):
```

```
    def wrapper(*arg, **kw):
        print '-- entering', func.__name__
        res = func(*arg, **kw)
        print '-- exiting', func.__name__
        return res
    return wrapper

@enter_exit_info
def f(x):
    print 'In:', x
    res = x + 2
    print 'Out:', res
    return res

a = f(2)

# Outputs:
#     -- entering f
#     In: 2
#     Out: 4
#     -- exiting f
```

Utilizzare nella registrazione

Quando si configura la funzionalità di `logging` incorporata, uno schema comune è creare un logger con `__name__` del modulo corrente:

```
logger = logging.getLogger(__name__)
```

Ciò significa che il nome completo del modulo apparirà nei log, rendendo più facile vedere da dove vengono i messaggi.

Leggi La variabile speciale __name__ online: https://riptutorial.com/it/python/topic/1223/la-variabile-speciale---name--

Capitolo 97: Lavorare attorno al Global Interpreter Lock (GIL)

Osservazioni

Perché c'è un GIL?

La GIL è stata utilizzata in CPython sin dall'inizio dei thread Python, nel 1992. È progettata per garantire la sicurezza dei thread nell'esecuzione del codice Python. Gli interpreti Python scritti con un GIL impediscono a più thread nativi di eseguire contemporaneamente bytecode Python. Ciò semplifica i plug-in per garantire che il loro codice sia sicuro per i thread: basta bloccare GIL e solo il thread attivo può essere eseguito, quindi il codice è automaticamente thread-safe.

Versione breve: GIL garantisce che, indipendentemente dal numero di processori e thread, *verrà eseguito solo un thread di un interprete Python alla volta.*

Questo ha molti benefici di facilità d'uso, ma ha anche molti benefici negativi.

Nota che un GIL non è un requirment del linguaggio Python. Di conseguenza, non puoi accedere a GIL direttamente dal codice Python standard. Non tutte le implementazioni di Python usano un GIL.

Interpreti che hanno un GIL: CPython, PyPy, Cython (ma puoi disabilitare GIL con `nogil`)

Interpreti che non hanno un GIL: Jython, IronPython

Dettagli su come funziona GIL:

Quando un thread è in esecuzione, blocca GIL. Quando un thread vuole essere eseguito, richiede GIL e attende finché non è disponibile. In CPython, prima della versione 3.2, il thread in esecuzione controllava dopo un certo numero di istruzioni python per vedere se altro codice voleva il lock (cioè, rilasciava il lock e poi lo richiedeva nuovamente). Questo metodo tendeva a causare l'inattività del thread, in gran parte perché il thread che rilasciava il blocco lo acquisiva nuovamente prima che i thread in attesa avessero la possibilità di riattivarsi. Dalla versione 3.2, i thread che vogliono che il GIL attenda il blocco per un po 'di tempo e, dopo questo periodo, impostano una variabile condivisa che impone il rendimento del thread in esecuzione. Ciò può comunque comportare tempi di esecuzione drasticamente più lunghi. Vedi i collegamenti sottostanti da dabeaz.com (nella sezione dei riferimenti) per maggiori dettagli.

CPython rilascia automaticamente GIL quando un thread esegue un'operazione di I / O. Le librerie di elaborazione delle immagini e le operazioni di crunch numerico rilasciano GIL prima di elaborarle.

Benefici della GIL

Per gli interpreti che usano GIL, GIL è sistemico. È usato per preservare lo stato dell'applicazione. I vantaggi includono:

- Garbage Collection: i conteggi dei riferimenti a livello di thread devono essere modificati mentre GIL è bloccato. *In CPython, tutta la collezione di garbarge è legata a GIL.* Questo è un grande; vedi l'articolo wiki di python.org sul GIL (elencato in Riferimenti, sotto) per i dettagli su cosa deve ancora essere funzionale se si volesse rimuovere GIL.
- Facilità per i programmatori che si occupano di GIL - il blocco di tutto è semplicistico, ma facile da codificare
- Facilita l'importazione di moduli da altre lingue

Conseguenze del GIL

GIL consente solo a un thread di eseguire codice Python alla volta all'interno dell'interprete python. Ciò significa che il multithreading di processi che eseguono rigorosi codici Python semplicemente non funziona. Quando si utilizzano i thread contro GIL, è probabile che si ottengano prestazioni peggiori con i thread rispetto a quando si esegue una singola discussione.

Riferimenti:

https://wiki.python.org/moin/GlobalInterpreterLock - riepilogo rapido di ciò che fa, dettagli precisi su tutti i vantaggi

http://programmers.stackexchange.com/questions/186889/why-was-python-written-with-the-gil - riassunto scritto in modo chiaro

http://www.dabeaz.com/python/UnderstandingGIL.pdf - come funziona GIL e perché rallenta su più core

http://www.dabeaz.com/GIL/gilvis/index.html - visualizzazione dei dati che mostrano come GIL blocca i thread

http://jeffknupp.com/blog/2012/03/31/pythons-hardest-problem/ - storia semplice da capire del problema GIL

https://jeffknupp.com/blog/2013/06/30/pythons-hardest-problem-revisited/ - dettagli sui modi per aggirare i limiti del GIL

Examples

Multiprocessing.Pool

La semplice risposta, quando si chiede come utilizzare i thread in Python è: "Non usare i processi, invece". Il modulo multiprocessing consente di creare processi con sintassi simile alla creazione di thread, ma preferisco utilizzare il loro conveniente oggetto Pool.

Usando il codice che David Beazley ha usato per mostrare i pericoli dei thread contro il GIL , lo riscriviamo usando multiprocessing.Pool :

Il codice di David Beazley che mostrava problemi di threading GIL

```
from threading import Thread
import time
def countdown(n):
    while n > 0:
        n -= 1

COUNT = 10000000

t1 = Thread(target=countdown,args=(COUNT/2,))
t2 = Thread(target=countdown,args=(COUNT/2,))
start = time.time()
t1.start();t2.start()
t1.join();t2.join()
end = time.time()
print end-start
```

Riscritto usando multiprocessing.Pool:

```
import multiprocessing
import time
def countdown(n):
    while n > 0:
        n -= 1

COUNT = 10000000

start = time.time()
with multiprocessing.Pool as pool:
    pool.map(countdown, [COUNT/2, COUNT/2])

    pool.close()
    pool.join()

end = time.time()
print(end-start)
```

Invece di creare thread, questo crea nuovi processi. Poiché ogni processo è il suo interprete, non ci sono collisioni GIL. multiprocessing.Pool aprirà tanti processi quanti sono i core sulla macchina, sebbene nell'esempio sopra, ne avrebbero solo bisogno due. In uno scenario del mondo reale, si desidera progettare il proprio elenco in modo che abbia una lunghezza pari a quella dei processori sulla macchina. Il Pool eseguirà la funzione che gli dici di eseguire con ogni argomento, fino al numero di processi che crea. Al termine della funzione, tutte le restanti funzioni nell'elenco

verranno eseguite su tale processo.

Ho scoperto che, anche usando l'istruzione `with` , se non si chiude e si aggiunge al pool, i processi continuano ad esistere. Per ripulire le risorse, chiudo sempre e mi unisco ai miei pool.

Cython nogil:

Cython è un interprete python alternativo. Usa GIL, ma ti consente di disabilitarlo. Vedi la loro documentazione

Ad esempio, usando il codice che David Beazley ha usato per mostrare i pericoli dei thread contro il GIL , lo riscriviamo usando nogil:

Il codice di David Beazley che mostrava problemi di threading GIL

```
from threading import Thread
import time
def countdown(n):
    while n > 0:
        n -= 1

COUNT = 10000000

t1 = Thread(target=countdown,args=(COUNT/2,))
t2 = Thread(target=countdown,args=(COUNT/2,))
start = time.time()
t1.start();t2.start()
t1.join();t2.join()
end = time.time()
print end-start
```

Riscritto usando nogil (SOLO FUNZIONA IN CYTHON):

```
from threading import Thread
import time
def countdown(n):
    while n > 0:
        n -= 1

COUNT = 10000000

with nogil:
    t1 = Thread(target=countdown,args=(COUNT/2,))
    t2 = Thread(target=countdown,args=(COUNT/2,))
    start = time.time()
    t1.start();t2.start()
    t1.join();t2.join()
```

```
end = time.time()
print end-start
```

È così semplice, finché usi cython. Nota che la documentazione dice che devi assicurarti di non cambiare alcun oggetto python:

> Il codice nel corpo dell'istruzione non deve manipolare oggetti Python in alcun modo e non deve chiamare nulla che manipoli oggetti Python senza prima re-acquisire il GIL. Attualmente Cython non controlla questo.

Leggi Lavorare attorno al Global Interpreter Lock (GIL) online: https://riptutorial.com/it/python/topic/4061/lavorare-attorno-al-global-interpreter-lock--gil-

Capitolo 98: Lavorare con gli archivi ZIP

Sintassi

- zipfile di importazione
- zipfile di *classe* . **ZipFile** (*file, modalità = 'r', compressione = ZIP_STORED, allowZip64 = True*)

Osservazioni

Se si tenta di aprire un file che non è un file ZIP, viene sollevata l'eccezione `zipfile.BadZipFile` .

In Python 2.7, questo era scritto `zipfile.BadZipfile` , e questo vecchio nome è mantenuto insieme a quello nuovo in Python 3.2+

Examples

Aprire i file zip

Per iniziare, importa il modulo `zipfile` e imposta il nome file.

```
import zipfile
filename = 'zipfile.zip'
```

Lavorare con gli archivi zip è molto simile a lavorare con i file , si crea l'oggetto aprendo il file zip, che consente di lavorarci sopra prima di chiudere nuovamente il file.

```
zip = zipfile.ZipFile(filename)
print(zip)
# <zipfile.ZipFile object at 0x0000000002E51A90>
zip.close()
```

In Python 2.7 e Python in 3 versioni superiori a 3.2, possiamo usare la `with` contesto manager. Apriamo il file in modalità "lettura", quindi stampiamo un elenco di nomi di file:

```
with zipfile.ZipFile(filename, 'r') as z:
    print(zip)
    # <zipfile.ZipFile object at 0x0000000002E51A90>
```

Esaminando i contenuti di Zipfile

Ci sono alcuni modi per ispezionare il contenuto di un file zip. È possibile utilizzare `printdir` per ottenere solo una serie di informazioni inviate allo `stdout`

```
with zipfile.ZipFile(filename) as zip:
```

```
    zip.printdir()

    # Out:
    # File Name                                             Modified             Size
    # pyexpat.pyd                                    2016-06-25 22:13:34       157336
    # python.exe                                     2016-06-25 22:13:34        39576
    # python3.dll                                    2016-06-25 22:13:34        51864
    # python35.dll                                   2016-06-25 22:13:34      3127960
    # etc.
```

Possiamo anche ottenere una lista di nomi di file con il `namelist` metodo. Qui, stampiamo semplicemente la lista:

```
with zipfile.ZipFile(filename) as zip:
    print(zip.namelist())

# Out: ['pyexpat.pyd', 'python.exe', 'python3.dll', 'python35.dll', ... etc. ...]
```

Invece `namelist` , possiamo chiamare il metodo `infolist` , che restituisce un elenco di oggetti `ZipInfo` , che contengono informazioni aggiuntive su ciascun file, ad esempio un timestamp e una dimensione del file:

```
with zipfile.ZipFile(filename) as zip:
    info = zip.infolist()
    print(zip[0].filename)
    print(zip[0].date_time)
    print(info[0].file_size)

# Out: pyexpat.pyd
# Out: (2016, 6, 25, 22, 13, 34)
# Out: 157336
```

Estrazione del contenuto del file zip in una directory

Estrai tutti i contenuti dei file di un file zip

```
import zipfile
with zipfile.ZipFile('zipfile.zip','r') as zfile:
    zfile.extractall('path')
```

Se vuoi estrarre singoli file usa il metodo di estrazione, prende nome elenco e percorso come parametro di input

```
import zipfile
f=open('zipfile.zip','rb')
zfile=zipfile.ZipFile(f)
for cont in zfile.namelist():
    zfile.extract(cont,path)
```

Creare nuovi archivi

Per creare un nuovo archivio aperto zipfile con modalità di scrittura.

```
import zipfile
new_arch=zipfile.ZipFile("filename.zip",mode="w")
```

Per aggiungere file a questo archivio usa il metodo write ().

```
new_arch.write('filename.txt','filename_in_archive.txt') #first parameter is filename and
second parameter is filename in archive by default filename will taken if not provided
new_arch.close()
```

Se vuoi scrivere una stringa di byte nell'archivio, puoi usare il metodo writstr ().

```
str_bytes="string buffer"
new_arch.writestr('filename_string_in_archive.txt',str_bytes)
new_arch.close()
```

Leggi Lavorare con gli archivi ZIP online: https://riptutorial.com/it/python/topic/3728/lavorare-con-gli-archivi-zip

Capitolo 99: Lettura e scrittura CSV

Examples

Scrivere un file TSV

Pitone

```
import csv

with open('/tmp/output.tsv', 'wt') as out_file:
    tsv_writer = csv.writer(out_file, delimiter='\t')
    tsv_writer.writerow(['name', 'field'])
    tsv_writer.writerow(['Dijkstra', 'Computer Science'])
    tsv_writer.writerow(['Shelah', 'Math'])
    tsv_writer.writerow(['Aumann', 'Economic Sciences'])
```

File di uscita

```
$ cat /tmp/output.tsv

name    field
Dijkstra    Computer Science
Shelah    Math
Aumann    Economic Sciences
```

Usando i panda

Scrivi un file CSV da un `dict` o da un `DataFrame` .

```
import pandas as pd

d = {'a': (1, 101), 'b': (2, 202), 'c': (3, 303)}
pd.DataFrame.from_dict(d, orient="index")
df.to_csv("data.csv")
```

Leggi un file CSV come `DataFrame` e convertilo in un `dict` :

```
df = pd.read_csv("data.csv")
d = df.to_dict()
```

Leggi Lettura e scrittura CSV online: https://riptutorial.com/it/python/topic/2116/lettura-e-scrittura-csv

Capitolo 100: Libreria di sottoprocesso

Sintassi

- subprocess.call (args, *, stdin = None, stdout = None, stderr = None, shell = False, timeout = None)
- subprocess.Popen (args, bufsize = -1, eseguibile = Nessuno, stdin = Nessuno, stdout = Nessuno, stderr = Nessuno, preexec_fn = Nessuno, close_fds = Vero, shell = False, cwd = Nessuno, env = Nessuno, universal_newlines = False , startupinfo = Nessuno, creationflags = 0, restore_signals = True, start_new_session = False, pass_fds = ())

Parametri

Parametro	Dettagli
`args`	Un singolo eseguibile, o una sequenza di file eseguibili e argomenti - `'ls'` , `['ls', '-la']`
`shell`	Corri sotto un guscio? La shell predefinita su `/bin/sh` su POSIX.
`cwd`	Directory di lavoro del processo figlio.

Examples

Chiamare i comandi esterni

Il caso d'uso più semplice sta usando la funzione `subprocess.call` . Accetta un elenco come primo argomento. Il primo elemento nell'elenco dovrebbe essere l'applicazione esterna che si desidera chiamare. Gli altri elementi nell'elenco sono argomenti che verranno passati a tale applicazione.

```
subprocess.call([r'C:\path\to\app.exe', 'arg1', '--flag', 'arg'])
```

Per i comandi della shell, imposta `shell=True` e fornisci il comando come una stringa invece di una lista.

```
subprocess.call('echo "Hello, world"', shell=True)
```

Si noti che i due comandi precedenti restituiscono solo lo `exit status` di `exit status` del sottoprocesso. Inoltre, fai attenzione quando usi `shell=True` dato che fornisce problemi di sicurezza (vedi qui).

Se si desidera essere in grado di ottenere l'output standard del sottoprocesso, quindi sostituire il `subprocess.call` con `subprocess.check_output` . Per un uso più avanzato, fare riferimento a questo .

Maggiore flessibilità con Popen

Utilizzo di `subprocess.Popen` offre un controllo più dettagliato sui processi avviati rispetto a `subprocess.call` .

Avvio di un sottoprocesso

```
process = subprocess.Popen([r'C:\path\to\app.exe', 'arg1', '--flag', 'arg'])
```

La firma per `Popen` è molto simile alla funzione di `call` ; tuttavia, `Popen` ritornerà immediatamente invece di attendere che il sottoprocesso completi come fa la `call` .

In attesa di un sottoprocesso da completare

```
process = subprocess.Popen([r'C:\path\to\app.exe', 'arg1', '--flag', 'arg'])
process.wait()
```

Lettura dell'output da un sottoprocesso

```
process = subprocess.Popen([r'C:\path\to\app.exe'], stdout=subprocess.PIPE,
stderr=subprocess.PIPE)

# This will block until process completes
stdout, stderr = process.communicate()
print stdout
print stderr
```

Accesso interattivo ai sottoprocessi in esecuzione

È possibile leggere e scrivere su `stdin` e `stdout` anche se il sottoprocesso non è stato completato. Questo potrebbe essere utile quando si automatizzano le funzionalità in un altro programma.

Scrivere su un sottoprocesso

```
process = subprocess.Popen([r'C:\path\to\app.exe'], stdout = subprocess.PIPE, stdin =
subprocess.PIPE)


process.stdin.write('line of input\n') # Write input

line  = process.stdout.readline() # Read a line from stdout
```

```
# Do logic on line read.
```

Tuttavia, se hai solo bisogno di un set di input e output, piuttosto che di interazione dinamica, dovresti usare `communicate()` piuttosto che accedere direttamente a `stdin` e `stdout` .

Leggere un flusso da un sottoprocesso

Nel caso in cui si desideri visualizzare l'output di un sottoprocesso riga per riga, è possibile utilizzare il seguente frammento:

```
process = subprocess.Popen(<your_command>, stdout=subprocess.PIPE)
while process.poll() is None:
    output_line = process.stdout.readline()
```

nel caso in cui l'output del sottocomando non abbia carattere EOL, lo snippet sopra riportato non funziona. È quindi possibile leggere l'output carattere per carattere come segue:

```
process = subprocess.Popen(<your_command>, stdout=subprocess.PIPE)
while process.poll() is None:
    output_line = process.stdout.read(1)
```

Il `1` specificata come argomento al `read` metodo indica Leggi per leggere 1 carattere alla volta. Puoi specificare di leggere tutti i caratteri che vuoi usando un numero diverso. Il numero negativo o 0 indica di `read` come una singola stringa finché non viene rilevato EOF (vedere qui).

In entrambi i frammenti di cui sopra, il file `process.poll()` è `None` fino al termine del sottoprocesso. Questo è usato per uscire dal ciclo una volta che non c'è più uscita da leggere.

La stessa procedura potrebbe essere applicata allo `stderr` del sottoprocesso.

Come creare l'argomento della lista comandi

Il metodo di sottoprocesso che consente di eseguire i comandi richiede il comando sotto forma di elenco (almeno utilizzando `shell_mode=True`).

Le regole per creare l'elenco non sono sempre semplici da seguire, specialmente con comandi complessi. Fortunatamente, c'è uno strumento molto utile che consente di farlo: `shlex` . Il modo più semplice per creare l'elenco da utilizzare come comando è il seguente:

```
import shlex
cmd_to_subprocess = shlex.split(command_used_in_the_shell)
```

Un semplice esempio:

```
import shlex
shlex.split('ls --color -l -t -r')

out: ['ls', '--color', '-l', '-t', '-r']
```

Leggi Libreria di sottoprocesso online: https://riptutorial.com/it/python/topic/1393/libreria-di-sottoprocesso

Capitolo 101: Liste collegate

introduzione

Un elenco collegato è una raccolta di nodi, ciascuno costituito da un riferimento e un valore. I nodi sono messi insieme in una sequenza usando i loro riferimenti. Le liste collegate possono essere utilizzate per implementare strutture di dati più complesse come liste, stack, code e array associativi.

Examples

Esempio di elenco singolo collegato

Questo esempio implementa un elenco collegato con molti degli stessi metodi dell'oggetto di elenco incorporato.

```
class Node:
    def __init__(self, val):
        self.data = val
        self.next = None

    def getData(self):
        return self.data

    def getNext(self):
        return self.next

    def setData(self, val):
        self.data = val

    def setNext(self, val):
        self.next = val

class LinkedList:
    def __init__(self):
        self.head = None

    def isEmpty(self):
        """Check if the list is empty"""
        return self.head is None

    def add(self, item):
        """Add the item to the list"""
        new_node = Node(item)
        new_node.setNext(self.head)
        self.head = new_node

    def size(self):
        """Return the length/size of the list"""
        count = 0
        current = self.head
        while current is not None:
            count += 1
```

```
            current = current.getNext()
        return count

    def search(self, item):
        """Search for item in list. If found, return True. If not found, return False"""
        current = self.head
        found = False
        while current is not None and not found:
            if current.getData() is item:
                found = True
            else:
                current = current.getNext()
        return found

    def remove(self, item):
        """Remove item from list. If item is not found in list, raise ValueError"""
        current = self.head
        previous = None
        found = False
        while current is not None and not found:
            if current.getData() is item:
                found = True
            else:
                previous = current
                current = current.getNext()
        if found:
            if previous is None:
                self.head = current.getNext()
            else:
                previous.setNext(current.getNext())
        else:
            raise ValueError
            print 'Value not found.'

    def insert(self, position, item):
        """
        Insert item at position specified. If position specified is
        out of bounds, raise IndexError
        """
        if position > self.size() - 1:
            raise IndexError
            print "Index out of bounds."
        current = self.head
        previous = None
        pos = 0
        if position is 0:
            self.add(item)
        else:
            new_node = Node(item)
            while pos < position:
                pos += 1
                previous = current
                current = current.getNext()
            previous.setNext(new_node)
            new_node.setNext(current)

    def index(self, item):
        """
        Return the index where item is found.
        If item is not found, return None.
        """
```

```
        current = self.head
        pos = 0
        found = False
        while current is not None and not found:
            if current.getData() is item:
                found = True
            else:
                current = current.getNext()
                pos += 1
        if found:
            pass
        else:
            pos = None
        return pos

    def pop(self, position = None):
        """
        If no argument is provided, return and remove the item at the head.
        If position is provided, return and remove the item at that position.
        If index is out of bounds, raise IndexError
        """
        if position > self.size():
            print 'Index out of bounds'
            raise IndexError

        current = self.head
        if position is None:
            ret = current.getData()
            self.head = current.getNext()
        else:
            pos = 0
            previous = None
            while pos < position:
                previous = current
                current = current.getNext()
                pos += 1
                ret = current.getData()
            previous.setNext(current.getNext())
        print ret
        return ret

    def append(self, item):
        """Append item to the end of the list"""
        current = self.head
        previous = None
        pos = 0
        length = self.size()
        while pos < length:
            previous = current
            current = current.getNext()
            pos += 1
        new_node = Node(item)
        if previous is None:
            new_node.setNext(current)
            self.head = new_node
        else:
            previous.setNext(new_node)

    def printList(self):
        """Print the list"""
        current = self.head
```

```
        while current is not None:
            print current.getData()
            current = current.getNext()
```

Le funzioni di utilizzo sono molto simili a quelle dell'elenco incorporato.

```
ll = LinkedList()
ll.add('l')
ll.add('H')
ll.insert(1,'e')
ll.append('l')
ll.append('o')
ll.printList()

H
e
l
l
o
```

Leggi Liste collegate online: https://riptutorial.com/it/python/topic/9299/liste-collegate

Capitolo 102: Loops

introduzione

Essendo una delle funzioni di base della programmazione, i loop sono un elemento importante in quasi tutti i linguaggi di programmazione. I loop consentono agli sviluppatori di impostare alcune parti del loro codice per ripetere attraverso un numero di cicli che vengono definiti iterazioni. Questo argomento tratta l'utilizzo di più tipi di loop e applicazioni di loop in Python.

Sintassi

- while <espressione booleana>:
- per <variabile> in <iterabile>:
- per <variabile> nell'intervallo (<numero>):
- per <variabile> nell'intervallo (<numero_iniziale>, <numero_infinizione>):
- per <variabile> nell'intervallo (<numero_avvio>, <numero_endenza>, <formato_passo>):
- per i, <variabile> in enumerate (<iterable>): # con indice i
- per <variable1>, <variable2> in zip (<iterable1>, <iterable2>):

Parametri

Parametro	Dettagli
espressione booleana	espressione che può essere valutata in un contesto booleano, ad esempio `x < 10`
variabile	nome della variabile per l'elemento corrente dal `iterable`
iterabile	tutto ciò che implementa le iterazioni

Examples

Iterare sulle liste

Per scorrere un elenco che puoi utilizzare `for` :

```
for x in ['one', 'two', 'three', 'four']:
    print(x)
```

Questo stamperà gli elementi della lista:

```
one
two
three
```

```
four
```

La funzione `range` genera numeri che vengono spesso utilizzati anche in un ciclo for.

```
for x in range(1, 6):
    print(x)
```

Il risultato sarà un tipo di sequenza intervallo speciale in python> = 3 e una lista in python <= 2. Entrambi possono essere collegati tramite il ciclo for.

```
1
2
3
4
5
```

Se si desidera eseguire il loop di entrambi gli elementi di una lista *e* avere anche un indice per gli elementi, è possibile utilizzare la funzione di `enumerate` di Python:

```
for index, item in enumerate(['one', 'two', 'three', 'four']):
    print(index, '::', item)
```

`enumerate` genererà tuple, che vengono decompresse in `index` (un intero) e `item` (il valore effettivo dall'elenco). Il ciclo sopra verrà stampato

```
(0, '::', 'one')
(1, '::', 'two')
(2, '::', 'three')
(3, '::', 'four')
```

Iterare su un elenco con manipolazione dei valori utilizzando `map` e `lambda` , ovvero applica la funzione lambda su ogni elemento nell'elenco:

```
x = map(lambda e :  e.upper(), ['one', 'two', 'three', 'four'])
print(x)
```

Produzione:

```
['ONE', 'TWO', 'THREE', 'FOUR'] # Python 2.x
```

NB: in Python 3.x la `map` restituisce un iteratore invece di una lista in modo che nel caso abbiate bisogno di una lista dovete lanciare il risultato `print(list(x))` (vedere http://www.Scriptutorial.com/python/ esempio / 8186 / map-- in http://www.Scriptutorial.com/python/topic/809/incompatibilities-moving-from-python-2-to-python-3).

Per i loop

`for` cicli itera su una raccolta di elementi, come `list` o `dict` , ed esegui un blocco di codice con

ogni elemento della raccolta.

```
for i in [0, 1, 2, 3, 4]:
    print(i)
```

Il ciclo sopra `for` itera su un elenco di numeri.

Ogni iterazione imposta il valore di `i` sull'elemento successivo dell'elenco. Quindi prima sarà `0` , poi `1` , poi `2` , ecc. L'output sarà il seguente:

```
0
1
2
3
4
```

`range` è una funzione che restituisce una serie di numeri sotto una forma iterabile, quindi può essere utilizzata in cicli `for` :

```
for i in range(5):
    print(i)
```

dà lo stesso risultato esatto come la prima `for` ciclo. Notare che `5` non è stampato poiché l'intervallo qui è i primi cinque numeri che contano da `0` .

Oggetti e iteratori iterabili

`for` loop può iterare su qualsiasi oggetto iterabile che è un oggetto che definisce una funzione `__getitem__` o `__iter__` . La funzione `__iter__` restituisce un iteratore, che è un oggetto con una funzione `next` che viene utilizzata per accedere all'elemento successivo del iterabile.

Rompi e continua in loop

dichiarazione di `break`

Quando un'istruzione `break` viene eseguita all'interno di un ciclo, il flusso di controllo "si interrompe" immediatamente dal ciclo:

```
i = 0
while i < 7:
    print(i)
    if i == 4:
        print("Breaking from loop")
        break
    i += 1
```

Il ciclo condizionale non verrà valutato dopo l'esecuzione dell'istruzione `break` . Nota che le frasi `break`

sono permesse solo *all'interno di loop* , sintatticamente. Un'istruzione `break` all'interno di una funzione non può essere utilizzata per terminare i loop che hanno chiamato quella funzione.

Eseguendo le seguenti stampe ogni cifra fino al numero `4` quando l'istruzione `break` è soddisfatta e il ciclo si ferma:

```
0
1
2
3
4
Breaking from loop
```

`break` istruzioni `break` possono anche essere utilizzate all'interno `for` loop, l'altro costrutto di loop fornito da Python:

```
for i in (0, 1, 2, 3, 4):
    print(i)
    if i == 2:
        break
```

L'esecuzione di questo ciclo ora stampa:

```
0
1
2
```

Si noti che 3 e 4 non vengono stampati poiché il ciclo è terminato.

Se un ciclo ha una clausola `else` , non viene eseguito quando il ciclo termina con un'istruzione `break` .

`continue` dichiarazione

Un'istruzione `continue` salterà alla successiva iterazione del ciclo ignorando il resto del blocco corrente ma continuando il ciclo. Come con la `break` , `continue` può apparire solo nei loop interni:

```
for i in (0, 1, 2, 3, 4, 5):
    if i == 2 or i == 4:
        continue
    print(i)

0
1
3
5
```

Nota che `2` e `4` non sono stampati, questo perché `continue` passa alla successiva iterazione invece di continuare a `print(i)` quando `i == 2` o `i == 4` .

Cicli annidati

`break` e `continue` funzionano solo su un singolo livello di loop. L'esempio seguente si interromperà solo dal ciclo `for` interno, non dal ciclo `while` esterno:

```
while True:
    for i in range(1,5):
        if i == 2:
            break    # Will only break out of the inner loop!
```

Python non ha la capacità di uscire da più livelli di loop contemporaneamente - se questo comportamento è desiderato, refactoring uno o più loop in una funzione e la sostituzione di `break` with `return` potrebbero essere la strada da percorrere.

Utilizzare il `return` da una funzione come una `break`

L' istruzione `return` esce da una funzione, senza eseguire il codice che viene dopo.

Se si dispone di un ciclo all'interno di una funzione, l'utilizzo del `return` dall'interno di quel ciclo equivale a `break` poiché il resto del codice del ciclo non viene eseguito (si *noti che qualsiasi codice dopo il ciclo non viene eseguito neanche*):

```
def break_loop():
    for i in range(1, 5):
        if (i == 2):
            return(i)
        print(i)
    return(5)
```

Se hai cicli annidati, l'istruzione `return` interromperà tutti i loop:

```
def break_all():
    for j in range(1, 5):
        for i in range(1,4):
            if i*j == 6:
                return(i)
            print(i*j)
```

produrrà:

```
1 # 1*1
2 # 1*2
3 # 1*3
4 # 1*4
2 # 2*1
4 # 2*2
# return because 2*3 = 6, the remaining iterations of both loops are not executed
```

Loop con una clausola "else"

Le dichiarazioni `for` e `while` (loop) possono facoltativamente avere una clausola `else` (in pratica, questo utilizzo è piuttosto raro).

La clausola `else` viene eseguita solo dopo che un ciclo `for` termina eseguendo il iterazione fino al completamento o dopo che un ciclo `while` termina con la sua espressione condizionale che diventa false.

```
for i in range(3):
    print(i)
else:
    print('done')

i = 0
while i < 3:
    print(i)
    i += 1
else:
    print('done')
```

produzione:

```
 0
 1
 2
 done
```

La clausola `else` non *viene* eseguita se il ciclo termina in un altro modo (attraverso un'istruzione `break` o sollevando un'eccezione):

```
for i in range(2):
    print(i)
    if i == 1:
        break
else:
    print('done')
```

produzione:

```
0
1
```

La maggior parte degli altri linguaggi di programmazione mancano di questa facoltativa `else` clausola di loop. L'utilizzo della parola chiave `else` , in particolare, è spesso considerata fonte di confusione.

Il concetto originale di tale clausola risale a Donald Knuth e il significato della parola chiave `else` diventa chiaro se riscriviamo un ciclo in termini di istruzioni `if` e istruzioni `goto` dei giorni precedenti alla programmazione strutturata o da un linguaggio assembly di livello inferiore.

Per esempio:

```
while loop_condition():
```

```
    ...
    if break_condition():
        break
    ...
```

è equivalente a:

```
# pseudocode

<<start>>:
if loop_condition():
    ...
    if break_condition():
        goto <<end>>
    ...
    goto <<start>>

<<end>>:
```

Rimangono equivalenti se alleghiamo una clausola `else` .

Per esempio:

```
while loop_condition():
    ...
    if break_condition():
        break
    ...
else:
    print('done')
```

è equivalente a:

```
# pseudocode

<<start>>:
if loop_condition():
    ...
    if break_condition():
        goto <<end>>
    ...
    goto <<start>>
else:
    print('done')

<<end>>:
```

Un ciclo `for` con una clausola `else` può essere compreso allo stesso modo. Concettualmente, esiste una condizione di loop che rimane True fintanto che l'oggetto o la sequenza iterabili ha ancora alcuni elementi rimanenti.

Perché dovremmo usare questo strano costrutto?

Il caso d'uso principale per il costrutto `for...else` è un'implementazione concisa della ricerca, ad

esempio:

```
a = [1, 2, 3, 4]
for i in a:
    if type(i) is not int:
        print(i)
        break
else:
    print("no exception")
```

Per rendere il `else` in questo costrutto meno confuso, si può pensare ad esso come " *se non si rompe* " o " *se non si trova* ".

Alcune discussioni su questo argomento possono essere trovate in [Python-ideas] Riepilogo di ... altri thread , perché python usa 'else' after for e while loops? e Else Clauses su Loop Statements

Iterare sui dizionari

Considerando il seguente dizionario:

```
 d = {"a": 1, "b": 2, "c": 3}
```

Per scorrere le sue chiavi, puoi usare:

```
for key in d:
    print(key)
```

Produzione:

```
"a"
"b"
"c"
```

Questo è equivalente a:

```
for key in d.keys():
    print(key)
```

o in Python 2:

```
for key in d.iterkeys():
    print(key)
```

Per scorrere i suoi valori, utilizzare:

```
for value in d.values():
    print(value)
```

Produzione:

```
1
2
3
```

Per scorrere le sue chiavi e i suoi valori, utilizzare:

```
for key, value in d.items():
    print(key, "::", value)
```

Produzione:

```
a :: 1
b :: 2
c :: 3
```

Si noti che in Python 2, `.keys()` , `.values()` e `.items()` restituiscono un oggetto `list` . Se hai semplicemente bisogno di iterare attraverso il risultato, puoi usare l'equivalente `.iterkeys()` , `.itervalues()` e `.iteritems()` .

La differenza tra `.keys()` e `.iterkeys()` , `.values()` e `.itervalues()` , `.items()` e `.iteritems()` è che i metodi `iter*` sono generatori. Pertanto, gli elementi all'interno del dizionario vengono restituiti uno per uno mentre vengono valutati. Quando viene restituito un oggetto `list` , tutti gli elementi vengono inseriti in un elenco e quindi restituiti per un'ulteriore valutazione.

> Si noti inoltre che in Python 3, l'ordine degli articoli stampati nel modo sopraindicato non segue alcun ordine.

Mentre Loop

Un ciclo `while` farà sì che le istruzioni del ciclo siano eseguite fino a quando la condizione del loop non è falsa . Il seguente codice eseguirà le istruzioni del ciclo per un totale di 4 volte.

```
i = 0
while i < 4:
    #loop statements
    i = i + 1
```

Mentre il ciclo sopra può essere facilmente tradotto in un ciclo `for` più elegante, `while` loop sono utili per verificare se alcune condizioni sono state soddisfatte. Il seguente ciclo continuerà ad essere eseguito fino a quando `myObject` sarà pronto.

```
myObject = anObject()
while myObject.isNotReady():
    myObject.tryToGetReady()
```

`while` loop possono anche essere eseguiti senza una condizione utilizzando numeri (complessi o

reali) o `True` :

```
import cmath

complex_num = cmath.sqrt(-1)
while complex_num:      # You can also replace complex_num with any number, True or a value of
any type
    print(complex_num)   # Prints 1j forever
```

Se la condizione è sempre true, il ciclo while verrà eseguito per sempre (ciclo infinito) se non viene terminato da un'istruzione break o return o un'eccezione.

```
while True:
    print "Infinite loop"
# Infinite loop
# Infinite loop
# Infinite loop
# ...
```

La dichiarazione del passaggio

`pass` è un'istruzione nulla per quando un'istruzione è richiesta dalla sintassi di Python (come all'interno del corpo di un ciclo `for` o `while`), ma nessuna azione è richiesta o desiderata dal programmatore. Questo può essere utile come segnaposto per il codice che deve ancora essere scritto.

```
for x in range(10):
    pass #we don't want to do anything, or are not ready to do anything here, so we'll pass
```

In questo esempio, non accadrà nulla. Il ciclo `for` si completerà senza errori, ma non verranno eseguiti comandi o codice. `pass` ci consente di eseguire il nostro codice con successo senza che tutti i comandi e le azioni siano pienamente implementati.

Allo stesso modo, il `pass` può essere utilizzato in cicli `while` , così come in selezioni e definizioni di funzioni, ecc.

```
while x == y:
    pass
```

Iterazione di parti diverse di un elenco con diverse dimensioni del passo

Immagina di avere una lunga lista di elementi e sei interessato solo ad ogni altro elemento della lista. Forse vuoi solo esaminare il primo o l'ultimo elemento o un intervallo specifico di voci nel tuo elenco. Python ha potenti funzionalità integrate di indicizzazione. Ecco alcuni esempi su come raggiungere questi scenari.

Ecco una semplice lista che verrà utilizzata negli esempi:

```
lst = ['alpha', 'bravo', 'charlie', 'delta', 'echo']
```

Iterazione sull'intera lista

Per scorrere su ciascun elemento nell'elenco, è possibile utilizzare un ciclo `for` come di seguito:

```
for s in lst:
    print s[:1] # print the first letter
```

Il ciclo `for` assegna s per ogni elemento di `lst` . Questo stamperà:

```
a
b
c
d
e
```

Spesso hai bisogno sia dell'elemento che dell'indice di quell'elemento. La parola chiave `enumerate` esegue questa attività.

```
for idx, s in enumerate(lst):
    print("%s has an index of %d" % (s, idx))
```

L'indice `idx` inizierà con zero e incrementerà per ogni iterazione, mentre i `s` conterranno l'elemento in elaborazione. Lo snippet precedente verrà visualizzato:

```
alpha has an index of 0
bravo has an index of 1
charlie has an index of 2
delta has an index of 3
echo has an index of 4
```

Iterare su sotto-lista

Se vogliamo iterare su un intervallo (ricordando che Python usa l'indicizzazione basata su zero), usa la parola chiave `range` .

```
for i in range(2,4):
    print("lst at %d contains %s" % (i, lst[i]))
```

Ciò produrrebbe:

```
lst at 2 contains charlie
lst at 3 contains delta
```

L'elenco può anche essere affettato. La seguente notazione di sezione passa dall'elemento all'indice 1 alla fine con un passo di 2. I due cicli `for` danno lo stesso risultato.

```
for s in lst[1::2]:
```

```
    print(s)

for i in range(1, len(lst), 2):
    print(lst[i])
```

I risultati del frammento di cui sopra:

```
bravo
delta
```

L'indicizzazione e l'affettamento sono un argomento a parte.

Il "mezzo giro" do-while

A differenza di altri linguaggi, Python non ha un costrutto do-until o do-while (questo permetterà al codice di essere eseguito una volta prima che la condizione venga testata). Tuttavia, è possibile combinare un `while True` con `break` per ottenere lo stesso scopo.

```
a = 10
while True:
    a = a-1
    print(a)
    if a<7:
        break
print('Done.')
```

Questo stamperà:

```
9
8
7
6
Done.
```

Looping e Disimballaggio

Ad esempio, se desideri scorrere un elenco di tuple:

```
collection = [('a', 'b', 'c'), ('x', 'y', 'z'), ('1', '2', '3')]
```

invece di fare qualcosa del genere:

```
for item in collection:
    i1 = item[0]
    i2 = item[1]
    i3 = item[2]
    # logic
```

o qualcosa del genere:

```
for item in collection:
    i1, i2, i3 = item
    # logic
```

Puoi semplicemente fare questo:

```
for i1, i2, i3 in collection:
    # logic
```

Questo funzionerà anche per la *maggior parte dei* tipi di iterabili, non solo per le tuple.

Leggi Loops online: https://riptutorial.com/it/python/topic/237/loops

Capitolo 103: Maledizioni di base con Python

Osservazioni

Maledizioni è un modulo di gestione terminale (o di visualizzazione caratteri) di base di Python. Questo può essere usato per creare interfacce utente o TUI basate su terminale.

Questa è una porta Python di una libreria C più popolare 'ncurses'

Examples

Esempio di chiamata di base

```
import curses
import traceback

try:
    # -- Initialize --
    stdscr = curses.initscr()    # initialize curses screen
    curses.noecho()              # turn off auto echoing of keypress on to screen
    curses.cbreak()              # enter break mode where pressing Enter key
                                 #  after keystroke is not required for it to register
    stdscr.keypad(1)             # enable special Key values such as curses.KEY_LEFT etc

    # -- Perform an action with Screen --
    stdscr.border(0)
    stdscr.addstr(5, 5, 'Hello from Curses!', curses.A_BOLD)
    stdscr.addstr(6, 5, 'Press q to close this screen', curses.A_NORMAL)

    while True:
        # stay in this loop till the user presses 'q'
        ch = stdscr.getch()
        if ch == ord('q'):
            break

    # -- End of user code --

except:
    traceback.print_exc()     # print trace back log of the error

finally:
    # --- Cleanup on exit ---
    stdscr.keypad(0)
    curses.echo()
    curses.nocbreak()
    curses.endwin()
```

La funzione helper wrapper ().

Mentre l'invocazione base sopra è abbastanza semplice, il pacchetto curses fornisce la `wrapper(func, ...)` helper `wrapper(func, ...)` . L'esempio seguente contiene l'equivalente di sopra:

```
main(scr, *args):
    # -- Perform an action with Screen --
    scr.border(0)
    scr.addstr(5, 5, 'Hello from Curses!', curses.A_BOLD)
    scr.addstr(6, 5, 'Press q to close this screen', curses.A_NORMAL)

    while True:
        # stay in this loop till the user presses 'q'
        ch = scr.getch()
        if ch == ord('q'):

curses.wrapper(main)
```

Qui, wrapper inizializzerà curses, creerà `stdscr` , un WindowObject e passerà sia a stdscr che a qualsiasi ulteriore argomento per `func` . Quando `func` ritorna, `wrapper` ripristinerà il terminale prima che il programma esca.

Leggi Maledizioni di base con Python online:
https://riptutorial.com/it/python/topic/5851/maledizioni-di-base-con-python

Capitolo 104: Manipolazione di XML

Osservazioni

Non tutti gli elementi dell'ingresso XML finiranno come elementi dell'albero analizzato. Attualmente, questo modulo ignora tutti i commenti XML, le istruzioni di elaborazione e le dichiarazioni del tipo di documento nell'input. Tuttavia, gli alberi creati usando l'API di questo modulo piuttosto che l'analisi dal testo XML possono contenere commenti e istruzioni di elaborazione; saranno inclusi durante la generazione dell'output XML.

Examples

Aprire e leggere usando un ElementTree

Importa l'oggetto ElementTree, apri il relativo file .xml e ottieni il tag radice:

```
import xml.etree.ElementTree as ET
tree = ET.parse("yourXMLfile.xml")
root = tree.getroot()
```

Ci sono alcuni modi per cercare attraverso l'albero. Il primo è per iterazione:

```
for child in root:
    print(child.tag, child.attrib)
```

Altrimenti puoi fare riferimento a posizioni specifiche come una lista:

```
print(root[0][1].text)
```

Per cercare tag specifici per nome, usa il `.find` o `.findall` :

```
print(root.findall("myTag"))
print(root[0].find("myOtherTag"))
```

Modifica di un file XML

Importa il modulo Albero degli elementi e apri il file xml, ottieni un elemento xml

```
import xml.etree.ElementTree as ET
tree = ET.parse('sample.xml')
root=tree.getroot()
element = root[0] #get first child of root element
```

L'oggetto Element può essere manipolato cambiando i suoi campi, aggiungendo e modificando attributi, aggiungendo e rimuovendo i bambini

```
element.set('attribute_name', 'attribute_value') #set the attribute to xml element
element.text="string_text"
```

Se si desidera rimuovere un elemento, utilizzare il metodo Element.remove ()

```
root.remove(element)
```

Metodo ElementTree.write () utilizzato per l'output di oggetti xml in file xml.

```
tree.write('output.xml')
```

Crea e crea documenti XML

Importa il modulo Albero degli elementi

```
import xml.etree.ElementTree as ET
```

La funzione Element () viene utilizzata per creare elementi XML

```
p=ET.Element('parent')
```

Funzione SubElement () utilizzata per creare elementi secondari in un elemento give

```
c = ET.SubElement(p, 'child1')
```

la funzione dump () viene utilizzata per scaricare elementi xml.

```
ET.dump(p)
# Output will be like this
#<parent><child1 /></parent>
```

Se vuoi salvare in un file creare un albero xml con la funzione ElementTree () e salvare in un file usa il metodo write ()

```
tree = ET.ElementTree(p)
tree.write("output.xml")
```

La funzione Comment () è utilizzata per inserire commenti nel file xml.

```
comment = ET.Comment('user comment')
p.append(comment) #this comment will be appended to parent element
```

Apertura e lettura di file XML di grandi dimensioni mediante iterparse (analisi incrementale)

A volte non vogliamo caricare l'intero file XML per ottenere le informazioni di cui abbiamo bisogno. In questi casi, è utile poter caricare in modo incrementale le sezioni pertinenti e quindi cancellarle

quando abbiamo finito. Con la funzione iterparse puoi modificare l'albero degli elementi che viene memorizzato mentre analizzi l'XML.

Importa l'oggetto ElementTree:

```
import xml.etree.ElementTree as ET
```

Aprire il file .xml e scorrere su tutti gli elementi:

```
for event, elem in ET.iterparse("yourXMLfile.xml"):
    ... do something ...
```

In alternativa, possiamo cercare solo eventi specifici, come tag inizio / fine o spazi dei nomi. Se questa opzione è omessa (come sopra), vengono restituiti solo gli eventi "di fine":

```
events=("start", "end", "start-ns", "end-ns")
for event, elem in ET.iterparse("yourXMLfile.xml", events=events):
    ... do something ...
```

Ecco l'esempio completo che mostra come cancellare elementi dalla struttura in memoria quando abbiamo finito con loro:

```
for event, elem in ET.iterparse("yourXMLfile.xml", events=("start","end")):
    if elem.tag == "record_tag" and event == "end":
        print elem.text
        elem.clear()
    ... do something else ...
```

Ricerca nell'XML con XPath

A partire dalla versione 2.7 `ElementTree` ha un supporto migliore per le query XPath. XPath è una sintassi che ti consente di navigare attraverso un xml come SQL è usato per cercare attraverso un database. Entrambe le funzioni `find` e `findall` supportano XPath. L'xml qui sotto sarà usato per questo esempio

```
 <Catalog>
    <Books>
        <Book id="1" price="7.95">
            <Title>Do Androids Dream of Electric Sheep?</Title>
            <Author>Philip K. Dick</Author>
        </Book>
        <Book id="5" price="5.95">
            <Title>The Colour of Magic</Title>
            <Author>Terry Pratchett</Author>
        </Book>
        <Book id="7" price="6.95">
            <Title>The Eye of The World</Title>
            <Author>Robert Jordan</Author>
        </Book>
    </Books>
</Catalog>
```

Ricerca di tutti i libri:

```
import xml.etree.cElementTree as ET
tree = ET.parse('sample.xml')
tree.findall('Books/Book')
```

Alla ricerca del libro con titolo = 'The Colour of Magic':

```
tree.find("Books/Book[Title='The Colour of Magic']")
# always use '' in the right side of the comparison
```

Cercando il libro con id = 5:

```
tree.find("Books/Book[@id='5']")
# searches with xml attributes must have '@' before the name
```

Cerca il secondo libro:

```
tree.find("Books/Book[2]")
# indexes starts at 1, not 0
```

Cerca l'ultimo libro:

```
tree.find("Books/Book[last()]")
# 'last' is the only xpath function allowed in ElementTree
```

Cerca tutti gli autori:

```
tree.findall(".//Author")
#searches with // must use a relative path
```

Leggi Manipolazione di XML online: https://riptutorial.com/it/python/topic/479/manipolazione-di-xml

Capitolo 105: Matematica complessa

Sintassi

- cmath.rect (AbsoluteValue, Phase)

Examples

Aritmetica complessa avanzata

Il modulo `cmath` include funzioni aggiuntive per utilizzare numeri complessi.

```
import cmath
```

Questo modulo può calcolare la fase di un numero complesso, in radianti:

```
z = 2+3j # A complex number
cmath.phase(z) # 0.982793723247329
```

Permette la conversione tra le rappresentazioni cartesiane (rettangolari) e polari di numeri complessi:

```
cmath.polar(z) # (3.605551275463989, 0.982793723247329)
cmath.rect(2, cmath.pi/2) # (0+2j)
```

Il modulo contiene la versione complessa di

- Funzioni esponenziali e logaritmiche (come al solito, `log` è il logaritmo naturale e `log10` il logaritmo decimale):

  ```
  cmath.exp(z) # (-7.315110094901103+1.0427436562359045j)
  cmath.log(z) # (1.2824746787307684+0.982793723247329j)
  cmath.log10(-100) # (2+1.3643763538418412j)
  ```

- Radici quadrate:

  ```
  cmath.sqrt(z) # (1.6741492280355401+0.8959774761298381j)
  ```

- Funzioni trigonometriche e loro inverse:

  ```
  cmath.sin(z)  # (9.15449914691143-4.168906959966565j)
  cmath.cos(z)  # (-4.189625690968807-9.109227893755337j)
  cmath.tan(z)  # (-0.003764025641504249+1.003238627353 61j)
  cmath.asin(z) # (0.5706527843210994+1.9833870299165355j)
  cmath.acos(z) # (1.0001435424737972-1.9833870299165355j)
  cmath.atan(z) # (1.4099210495965755+0.229072682968538 78j)
  cmath.sin(z)**2 + cmath.cos(z)**2 # (1+0j)
  ```

- Funzioni iperboliche e loro invers:

```
cmath.sinh(z)  # (-3.59056458998578+0.5309210862485197j)
cmath.cosh(z)  # (-3.7245455049153224+0.5118225699873846j)
cmath.tanh(z)  # (0.965385879022133-0.009884375038322495j)
cmath.asinh(z) # (0.5706527843210994+1.9833870299165355j)
cmath.acosh(z) # (1.9833870299165355+1.0001435424737972j)
cmath.atanh(z) # (0.14694666622552977+1.3389725222944935j)
cmath.cosh(z)**2 - cmath.sin(z)**2  # (1+0j)
cmath.cosh((0+1j)*z) - cmath.cos(z) # 0j
```

Aritmetica complessa di base

Python ha il supporto integrato per l'aritmetica complessa. L'unità immaginaria è denotata da `j` :

```
z = 2+3j # A complex number
w = 1-7j # Another complex number
```

I numeri complessi possono essere sommati, sottratti, moltiplicati, divisi ed esponenziati:

```
z + w # (3-4j)
z - w # (1+10j)
z * w # (23-11j)
z / w # (-0.38+0.34j)
z**3  # (-46+9j)
```

Python può anche estrarre le parti reali e immaginarie di numeri complessi e calcolare il loro valore assoluto e coniugare:

```
z.real # 2.0
z.imag # 3.0
abs(z) # 3.605551275463989
z.conjugate() # (2-3j)
```

Leggi Matematica complessa online: https://riptutorial.com/it/python/topic/1142/matematica-complessa

Capitolo 106: Matrici multidimensionali

Examples

Elenchi in elenchi

Un buon modo per visualizzare un array 2d è come un elenco di elenchi. Qualcosa come questo:

```
lst=[[1,2,3],[4,5,6],[7,8,9]]
```

qui la lista esterna `lst` ha tre cose in esso. ognuna di queste cose è un'altra lista: la prima è: `[1,2,3]` , la seconda è: `[4,5,6]` e la terza è: `[7,8,9]` . Puoi accedere a questi elenchi nello stesso modo in cui accederai a un altro elemento di un elenco, come questo:

```
print (lst[0])
#output: [1, 2, 3]

print (lst[1])
#output: [4, 5, 6]

print (lst[2])
#output: [7, 8, 9]
```

È quindi possibile accedere ai diversi elementi in ciascuno di questi elenchi allo stesso modo:

```
print (lst[0][0])
#output: 1

print (lst[0][1])
#output: 2
```

Qui il primo numero all'interno delle parentesi `[]` significa ottenere l'elenco in quella posizione. Nell'esempio sopra abbiamo usato il numero `0` per ottenere la lista nella posizione 0 che è `[1,2,3]` . Il secondo set di parentesi `[]` significa ottenere l'oggetto in quella posizione dall'elenco interno. In questo caso abbiamo usato sia `0` che `1` la 0a posizione nella lista che abbiamo ottenuto è il numero `1` e nella 1a posizione è `2`

Puoi anche impostare valori all'interno di questi elenchi allo stesso modo:

```
lst[0]=[10,11,12]
```

Ora la lista è `[[10,11,12],[4,5,6],[7,8,9]]` . In questo esempio abbiamo cambiato l'intero primo elenco per essere un elenco completamente nuovo.

```
lst[1][2]=15
```

Ora la lista è `[[10,11,12],[4,5,15],[7,8,9]]` . In questo esempio abbiamo modificato un singolo

elemento all'interno di una delle liste interne. Per prima cosa siamo entrati nella lista in posizione 1 e abbiamo cambiato l'elemento al suo interno in posizione 2, che ora era 6 ora è 15.

Elenchi in elenchi in elenchi in ...

Questo comportamento può essere esteso. Ecco una matrice tridimensionale:

```
[[[111,112,113],[121,122,123],[131,132,133]],[[211,212,213],[221,222,223],[231,232,233]],[[311,312,313]
```

Come è probabilmente ovvio, questo diventa un po 'difficile da leggere. Utilizza le barre rovesciate per suddividere le diverse dimensioni:

```
[[[111,112,113],[121,122,123],[131,132,133]],\
 [[211,212,213],[221,222,223],[231,232,233]],\
 [[311,312,313],[321,322,323],[331,332,333]]]
```

Nidificando gli elenchi come questo, puoi estendere a dimensioni arbitrariamente elevate.

L'accesso è simile agli array 2D:

```
print(myarray)
print(myarray[1])
print(myarray[2][1])
print(myarray[1][0][2])
etc.
```

E il montaggio è anche simile:

```
myarray[1]=new_n-1_d_list
myarray[2][1]=new_n-2_d_list
myarray[1][0][2]=new_n-3_d_list #or a single number if you're dealing with 3D arrays
etc.
```

Leggi Matrici multidimensionali online: https://riptutorial.com/it/python/topic/8186/matrici-multidimensionali

Capitolo 107: metaclassi

introduzione

Le metaclassi ti permettono di modificare profondamente il comportamento delle classi Python (in termini di come sono definite, istanziate, accessibili e altro) sostituendo la metaclasse di `type` che le nuove classi usano di default.

Osservazioni

Quando progettate la vostra architettura, considerate che molte cose che possono essere compiute con le metaclassi possono anche essere realizzate usando semantica più semplice:

- L'ereditarietà tradizionale è spesso più che sufficiente.
- I decoratori di classe possono integrare le funzionalità in lezioni con un approccio ad hoc.
- Python 3.6 introduce `__init_subclass__()` che consente a una classe di partecipare alla creazione della sua sottoclasse.

Examples

Metaclasses di base

Quando il `type` viene chiamato con tre argomenti, si comporta come la (meta) classe che è, e crea una nuova istanza, es. produce una nuova classe / tipo.

```
Dummy = type('OtherDummy', (), dict(x=1))
Dummy.__class__              # <type 'type'>
Dummy().__class__.__class__  # <type 'type'>
```

È possibile sottoclasse il `type` per creare un metaclass personalizzato.

```
class mytype(type):
    def __init__(cls, name, bases, dict):
        # call the base initializer
        type.__init__(cls, name, bases, dict)

        # perform custom initialization...
        cls.__custom_attribute__ = 2
```

Ora abbiamo un nuovo metaclasse `mytype` personalizzato che può essere utilizzato per creare classi nello stesso modo del `type` .

```
MyDummy = mytype('MyDummy', (), dict(x=2))
MyDummy.__class__              # <class '__main__.mytype'>
MyDummy().__class__.__class__  # <class '__main__.mytype'>
MyDummy.__custom_attribute__   # 2
```

Quando creiamo una nuova classe usando la parola chiave della `class` il metaclasse viene scelto di default in base alle baseclass.

```
>>> class Foo(object):
...     pass

>>> type(Foo)
type
```

Nell'esempio precedente l'unica classe di base è l' `object` quindi il nostro metaclasse sarà il tipo di `object` , che è il `type` . È possibile sovrascrivere il valore predefinito, tuttavia dipende se utilizziamo Python 2 o Python 3:

Python 2.x 2.7

È possibile utilizzare un attributo speciale di livello di classe `__metaclass__` per specificare il metaclass.

```
class MyDummy(object):
    __metaclass__ = mytype
type(MyDummy)  # <class '__main__.mytype'>
```

Python 3.x 3.0

Un argomento specifico per la parola chiave `metaclass` specifica il metaclass.

```
class MyDummy(metaclass=mytype):
    pass
type(MyDummy)  # <class '__main__.mytype'>
```

Tutti gli argomenti delle parole chiave (tranne il `metaclass`) nella dichiarazione della classe verranno passati al metaclasse. Quindi la `class MyDummy(metaclass=mytype, x=2)` passerà `x=2` come argomento della parola chiave al costruttore `mytype` .

Leggi questa descrizione approfondita delle meta-classi Python per maggiori dettagli.

Singleton usando metaclassi

Un singleton è un pattern che limita l'istanza di una classe a un'istanza / oggetto. Per maggiori informazioni sui modelli di design singleton Python, vedere qui .

```
class SingletonType(type):
    def __call__(cls, *args, **kwargs):
        try:
            return cls.__instance
        except AttributeError:
            cls.__instance = super(SingletonType, cls).__call__(*args, **kwargs)
            return cls.__instance
```

Python 2.x 2.7

```
class MySingleton(object):
    __metaclass__ = SingletonType
```

Python 3.x 3.0

```
class MySingleton(metaclass=SingletonType):
    pass
```

```
MySingleton() is MySingleton()  # True, only one instantiation occurs
```

Utilizzando un metaclass

Sintassi del metaclass

Python 2.x 2.7

```
class MyClass(object):
    __metaclass__ = SomeMetaclass
```

Python 3.x 3.0

```
class MyClass(metaclass=SomeMetaclass):
    pass
```

Compatibilità con Python 2 e 3 con six

```
import six

class MyClass(six.with_metaclass(SomeMetaclass)):
    pass
```

Funzionalità personalizzate con metaclassi

Le funzionalità delle metaclassi possono essere modificate in modo che ogni volta che viene creata una classe, una stringa venga stampata sullo standard output o venga generata un'eccezione. Questo metaclasse stamperà il nome della classe in costruzione.

```
class VerboseMetaclass(type):

    def __new__(cls, class_name, class_parents, class_dict):
        print("Creating class ", class_name)
        new_class = super().__new__(cls, class_name, class_parents, class_dict)
        return new_class
```

Puoi usare la metaclatura in questo modo:

```
class Spam(metaclass=VerboseMetaclass):
```

```
    def eggs(self):
        print("[insert example string here]")
s = Spam()
s.eggs()
```

Lo standard output sarà:

```
Creating class Spam
[insert example string here]
```

Introduzione ai metaclassi

Cos'è un metaclass?

In Python, tutto è un oggetto: interi, stringhe, liste, persino funzioni e classi stesse sono oggetti. E ogni oggetto è un'istanza di una classe.

Per verificare la classe di un oggetto x, si può chiamare `type(x)` , quindi:

```
>>> type(5)
<type 'int'>
>>> type(str)
<type 'type'>
>>> type([1, 2, 3])
<type 'list'>

>>> class C(object):
...     pass
...
>>> type(C)
<type 'type'>
```

La maggior parte delle classi in python sono istanze di `type` . `type` stesso è anche una classe. Tali classi le cui istanze sono anche classi sono chiamate metaclassi.

Il Metaclass più semplice

OK, quindi c'è già un metaclasse in Python: `type` . Possiamo crearne un altro?

```
class SimplestMetaclass(type):
    pass

class MyClass(object):
    __metaclass__ = SimplestMetaclass
```

Questo non aggiunge alcuna funzionalità, ma è un nuovo metaclasse, vedi che MyClass è ora un'istanza di SimplestMetaclass:

```
>>> type(MyClass)
<class '__main__.SimplestMetaclass'>
```

Un Metaclass che fa qualcosa

Una metaclasse che fa qualcosa di solito sovrascrive il `type` `__new__` , per modificare alcune proprietà della classe da creare, prima di chiamare l' `__new__` originale che crea la classe:

```
class AnotherMetaclass(type):
    def __new__(cls, name, parents, dct):
        # cls is this class
        # name is the name of the class to be created
        # parents is the list of the class's parent classes
        # dct is the list of class's attributes (methods, static variables)

        # here all of the attributes can be modified before creating the class, e.g.

        dct['x'] = 8  # now the class will have a static variable x = 8

        # return value is the new class. super will take care of that
        return super(AnotherMetaclass, cls).__new__(cls, name, parents, dct)
```

Il metaclass predefinito

Potresti aver sentito che tutto in Python è un oggetto. È vero, e tutti gli oggetti hanno una classe:

```
>>> type(1)
int
```

Il letterale 1 è un'istanza di `int` . Dichiariamo una classe:

```
>>> class Foo(object):
...    pass
...
```

Ora consente di istanziarlo:

```
>>> bar = Foo()
```

Qual è la classe di `bar` ?

```
>>> type(bar)
Foo
```

Bello, il `bar` è un'istanza di `Foo` . Ma qual è la classe di `Foo` stessa?

```
>>> type(Foo)
type
```

Ok, `Foo` stesso è un'istanza di `type` . Che ne dici del `type` stesso?

```
>>> type(type)
type
```

Allora, cos'è un metaclass? Per ora, facciamo finta che sia solo un nome di fantasia per la classe di una classe. Takeaways:

- Tutto è un oggetto in Python, quindi tutto ha una classe
- La classe di una classe è chiamata metaclasse
- Il metaclass predefinito è `type` e di gran lunga è il metaclass più comune

Ma perché dovresti sapere delle metaclassi? Bene, Python stesso è abbastanza "hackerabile", e il concetto di metaclasse è importante se stai facendo cose avanzate come la meta-programmazione o se vuoi controllare come vengono inizializzate le tue classi.

Leggi metaclassi online: https://riptutorial.com/it/python/topic/286/metaclassi

Capitolo 108: Metodi definiti dall'utente

Examples

Creazione di oggetti metodo definiti dall'utente

Gli oggetti metodo definiti dall'utente possono essere creati quando si ottiene un attributo di una classe (magari tramite un'istanza di tale classe), se tale attributo è un oggetto funzione definito dall'utente, un oggetto metodo definito non associato o un oggetto metodo classe.

```
class A(object):
    # func: A user-defined function object
    #
    # Note that func is a function object when it's defined,
    # and an unbound method object when it's retrieved.
    def func(self):
        pass

    # classMethod: A class method
    @classmethod
    def classMethod(self):
        pass

class B(object):
    # unboundMeth: A unbound user-defined method object
    #
    # Parent.func is an unbound user-defined method object here,
    # because it's retrieved.
    unboundMeth = A.func

a = A()
b = B()

print A.func
# output: <unbound method A.func>
print a.func
# output: <bound method A.func of <__main__.A object at 0x10e9ab910>>
print B.unboundMeth
# output: <unbound method A.func>
print b.unboundMeth
# output: <unbound method A.func>
print A.classMethod
# output: <bound method type.classMethod of <class '__main__.A'>>
print a.classMethod
# output: <bound method type.classMethod of <class '__main__.A'>>
```

Quando l'attributo è un oggetto metodo definito dall'utente, un nuovo oggetto metodo viene creato solo se la classe da cui viene richiamato è uguale o una classe derivata della classe memorizzata nell'oggetto metodo originale; in caso contrario, l'oggetto metodo originale viene utilizzato così com'è.

```
# Parent: The class stored in the original method object
class Parent(object):
```

```
    # func: The underlying function of original method object
    def func(self):
        pass
    func2 = func

# Child: A derived class of Parent
class Child(Parent):
    func = Parent.func

# AnotherClass: A different class, neither subclasses nor subclassed
class AnotherClass(object):
    func = Parent.func

print Parent.func is Parent.func                # False, new object created
print Parent.func2 is Parent.func2              # False, new object created
print Child.func is Child.func                  # False, new object created
print AnotherClass.func is AnotherClass.func    # True, original object used
```

Esempio di tartaruga

Di seguito è riportato un esempio di utilizzo di una funzione definita dall'utente da chiamare più volte (∞) in uno script con facilità.

```
import turtle, time, random #tell python we need 3 different modules
turtle.speed(0) #set draw speed to the fastest
turtle.colormode(255) #special colormode
turtle.pensize(4) #size of the lines that will be drawn
def triangle(size): #This is our own function, in the parenthesis is a variable we have
defined that will be used in THIS FUNCTION ONLY. This fucntion creates a right triangle
    turtle.forward(size) #to begin this function we go forward, the amount to go forward by is
the variable size
    turtle.right(90) #turn right by 90 degree
    turtle.forward(size) #go forward, again with variable
    turtle.right(135) #turn right again
    turtle.forward(size * 1.5) #close the triangle. thanks to the Pythagorean theorem we know
that this line must be 1.5 times longer than the other two(if they are equal)
while(1): #INFINITE LOOP
    turtle.setpos(random.randint(-200, 200), random.randint(-200, 200)) #set the draw point to
a random (x,y) position
    turtle.pencolor(random.randint(1, 255), random.randint(1, 255), random.randint(1, 255))
#randomize the RGB color
    triangle(random.randint(5, 55)) #use our function, because it has only one variable we can
simply put a value in the parenthesis. The value that will be sent will be random between 5 -
55, end the end it really just changes ow big the triangle is.
    turtle.pencolor(random.randint(1, 255), random.randint(1, 255), random.randint(1, 255))
#randomize color again
```

Leggi Metodi definiti dall'utente online: https://riptutorial.com/it/python/topic/3965/metodi-definiti-dall-utente

Capitolo 109: Metodi di stringa

Sintassi

- str.capitalize () -> str
- str.casefold () -> str [solo per Python> 3.3]
- str.center (width [, fillchar]) -> str
- str.count (sub [, start [, end]]) -> int
- str.decode (encoding = "utf-8" [, errors]) -> unicode [solo in Python 2.x]
- str.encode (encoding = "utf-8", errors = "strict") -> byte
- str.endswith (suffisso [, inizio [, fine]]) -> bool
- str.expandtabs (tabsize = 8) -> str
- str.find (sub [, start [, end]]) -> int
- str.format (* args, ** kwargs) -> str
- str.format_map (mapping) -> str
- str.index (sub [, start [, end]]) -> int
- str.isalnum () -> bool
- str.isalpha () -> bool
- str.isdecimal () -> bool
- str.isdigit () -> bool
- str.isidentifier () -> bool
- str.islower () -> bool
- str.isnumeric () -> bool
- str.isprintable () -> bool
- str.isspace () -> bool
- str.istitle () -> bool
- str.isupper () -> bool
- str.join (iterable) -> str
- str.ljust (width [, fillchar]) -> str
- str.lower () -> str
- str.lstrip ([chars]) -> str
- statico str.maketrans (x [, y [, z]])
- str.partition (sep) -> (testa, sep, coda)
- str.replace (old, new [, count]) -> str
- str.rfind (sub [, start [, end]]) -> int
- str.rindex (sub [, start [, end]]) -> int
- str.rjust (width [, fillchar]) -> str
- str.rpartition (sep) -> (testa, sep, coda)
- str.rsplit (sep = None, maxsplit = -1) -> elenco di stringhe
- str.rstrip ([chars]) -> str
- str.split (sep = None, maxsplit = -1) -> elenco di stringhe
- str.splitlines ([keepends]) -> lista di stringhe
- str.startswith (prefisso [, inizio [, fine]]) -> libro
- str.strip ([chars]) -> str

- str.swapcase () -> str
- str.title () -> str
- str.translate (table) -> str
- str.upper () -> str
- str.zfill (larghezza) -> str

Osservazioni

Gli oggetti stringa sono immutabili, il che significa che non possono essere modificati sul posto come può fare una lista. Per questo motivo, i metodi sul tipo `str` restituiscono sempre un **nuovo** oggetto `str` , che contiene il risultato della chiamata al metodo.

Examples

Modifica della maiuscola di una stringa

Il tipo di stringa di Python fornisce molte funzioni che agiscono sulla maiuscola di una stringa. Questi includono :

- `str.casefold`
- `str.upper`
- `str.lower`
- `str.capitalize`
- `str.title`
- `str.swapcase`

Con le stringhe unicode (l'impostazione predefinita in Python 3), queste operazioni **non** sono mappature 1: 1 o reversibili. La maggior parte di queste operazioni è destinata alla visualizzazione, piuttosto che alla normalizzazione.

Python 3.x 3.3

`str.casefold()`

`str.casefold` crea una stringa minuscola adatta per confronti senza distinzione tra maiuscole e minuscole. Questo è più aggressivo di `str.lower` e può modificare stringhe che sono già in minuscolo o che fanno crescere le stringhe di lunghezza, e non è inteso per scopi di visualizzazione.

```
"XßΣ".casefold()
# 'xssσ'

"XßΣ".lower()
# 'xßς'
```

Le trasformazioni che avvengono sotto la distinzione tra maiuscole e minuscole sono definite dal Consorzio Unicode nel file CaseFolding.txt sul loro sito web.

---

str.upper()

`str.upper` prende ogni carattere in una stringa e lo converte in equivalente in maiuscolo, ad esempio:

```
"This is a 'string'.".upper()
# "THIS IS A 'STRING'."
```

---

str.lower()

`str.lower` fa il contrario; prende ogni carattere in una stringa e lo converte in equivalente in lettere minuscole:

```
"This IS a 'string'.".lower()
# "this is a 'string'."
```

---

str.capitalize()

`str.capitalize` restituisce una versione in maiuscolo della stringa, ovvero rende il primo carattere con maiuscole e il resto inferiore:

```
"this Is A 'String'.".capitalize() # Capitalizes the first character and lowercases all others
# "This is a 'string'."
```

---

str.title()

`str.title` restituisce la versione con `str.title` del titolo della stringa, cioè ogni lettera all'inizio di una parola è fatta in maiuscolo e tutte le altre sono scritte in minuscolo:

```
"this Is a 'String'".title()
# "This Is A 'String'"
```

---

str.swapcase()

`str.swapcase` restituisce un nuovo oggetto stringa in cui tutti i caratteri minuscoli vengono scambiati in maiuscolo e tutti i caratteri maiuscoli in basso:

```
"this iS A STRiNG".swapcase() #Swaps case of each character
# "THIS Is a strIng"
```

Utilizzo come metodi di classe `str`

Vale la pena notare che questi metodi possono essere chiamati sia su oggetti stringa (come mostrato sopra) o come metodo di classe della classe `str` (con una chiamata esplicita a `str.upper` , ecc.)

```
str.upper("This is a 'string'")
# "THIS IS A 'STRING'"
```

Questo è molto utile quando si applica uno di questi metodi a più stringhe contemporaneamente, ad esempio, una funzione *map* .

```
map(str.upper,["These","are","some","'strings'"])
# ['THESE', 'ARE', 'SOME', "'STRINGS'"]
```

Dividere una stringa in base a un delimitatore in un elenco di stringhe

str.split(sep=None, maxsplit=-1)

`str.split` prende una stringa e restituisce una lista di sottostringhe della stringa originale. Il comportamento varia a seconda che l'argomento `sep` sia fornito o omesso.

Se `sep` non è fornito, o è `None` , la suddivisione avviene ovunque ci sia spazio bianco. Tuttavia, gli spazi bianchi iniziali e finali vengono ignorati e più caratteri spazi bianchi consecutivi vengono trattati allo stesso modo di un singolo carattere di spaziatura:

```
>>> "This is a sentence.".split()
['This', 'is', 'a', 'sentence.']

>>> " This is    a sentence.  ".split()
['This', 'is', 'a', 'sentence.']

>>> "            ".split()
[]
```

Il parametro `sep` può essere utilizzato per definire una stringa delimitatore. La stringa originale è divisa in cui si verifica la stringa del delimitatore e il delimitatore stesso viene scartato. Più delimitatori consecutivi *non* vengono trattati allo stesso modo di una singola occorrenza, ma piuttosto creano stringhe vuote da creare.

```
>>> "This is a sentence.".split(' ')
['This', 'is', 'a', 'sentence.']

>>> "Earth,Stars,Sun,Moon".split(',')
['Earth', 'Stars', 'Sun', 'Moon']

>>> " This is    a sentence.  ".split(' ')
['', 'This', 'is', '', '', '', 'a', 'sentence.', '', '']

>>> "This is a sentence.".split('e')
['This is a s', 'nt', 'nc', '.']

>>> "This is a sentence.".split('en')
['This is a s', 't', 'ce.']
```

L'impostazione predefinita è dividere *ogni* occorrenza del delimitatore, tuttavia il parametro `maxsplit` limita il numero di errori che si verificano. Il valore predefinito di `-1` indica nessun limite:

```
>>> "This is a sentence.".split('e', maxsplit=0)
['This is a sentence.']

>>> "This is a sentence.".split('e', maxsplit=1)
['This is a s', 'ntence.']

>>> "This is a sentence.".split('e', maxsplit=2)
['This is a s', 'nt', 'nce.']

>>> "This is a sentence.".split('e', maxsplit=-1)
['This is a s', 'nt', 'nc', '.']
```

`str.rsplit(sep=None, maxsplit=-1)`

`str.rsplit` ("right split") si differenzia da `str.split` ("left split") quando viene specificato `maxsplit` . La divisione inizia alla fine della stringa anziché all'inizio:

```
>>> "This is a sentence.".rsplit('e', maxsplit=1)
['This is a sentenc', '.']

>>> "This is a sentence.".rsplit('e', maxsplit=2)
['This is a sent', 'nc', '.']
```

Nota : Python specifica il numero massimo di *suddivisioni* eseguite, mentre la maggior parte degli altri linguaggi di programmazione specifica il numero massimo di *sottostringhe* create. Ciò potrebbe creare confusione durante il porting o il confronto del codice.

Sostituisci tutte le occorrenze di una sottostringa con un'altra sottostringa

Il tipo `str` di Python ha anche un metodo per sostituire le occorrenze di una sottostringa con un'altra sottostringa in una determinata stringa. Per i casi più impegnativi, si può usare `re.sub` .

`str.replace(old, new[, count])` :

`str.replace` accetta due argomenti `old` e `new` contenenti la `old` sottostringa che deve essere sostituita dalla `new` sottostringa. Il `count` argomento facoltativo specifica il numero di sostituzioni da apportare:

Ad esempio, per sostituire `'foo'` con `'spam'` nella stringa seguente, possiamo chiamare `str.replace` con `old = 'foo'` e `new = 'spam'` :

```
>>> "Make sure to foo your sentence.".replace('foo', 'spam')
"Make sure to spam your sentence."
```

Se la stringa data contiene più esempi che corrispondono al `old` argomento, **tutte le** occorrenze vengono sostituite con il valore fornito in `new` :

```
>>> "It can foo multiple examples of foo if you want.".replace('foo', 'spam')
"It can spam multiple examples of spam if you want."
```

a meno che, ovviamente, forniamo un valore per il `count` . In questo caso, le occorrenze di `count` verranno sostituite:

```
>>> """It can foo multiple examples of foo if you want, \
... or you can limit the foo with the third argument.""".replace('foo', 'spam', 1)
'It can spam multiple examples of foo if you want, or you can limit the foo with the third
argument.'
```

str.format e f-strings: formatta i valori in una stringa

Python fornisce funzionalità di interpolazione e formattazione delle `str.format` attraverso la funzione `str.format` , introdotta nella versione 2.6 e nella sezione f introdotta nella versione 3.6.

Date le seguenti variabili:

```
i = 10
f = 1.5
s = "foo"
l = ['a', 1, 2]
d = {'a': 1, 2: 'foo'}
```

Le seguenti affermazioni sono tutte equivalenti

```
"10 1.5 foo ['a', 1, 2] {'a': 1, 2: 'foo'}"
```

```
>>> "{} {} {} {} {}".format(i, f, s, l, d)

>>> str.format("{} {} {} {} {}", i, f, s, l, d)

>>> "{0} {1} {2} {3} {4}".format(i, f, s, l, d)

>>> "{0:d} {1:0.1f} {2} {3!r} {4!r}".format(i, f, s, l, d)

>>> "{i:d} {f:0.1f} {s} {l!r} {d!r}".format(i=i, f=f, s=s, l=l, d=d)
```

```
>>> f"{i} {f} {s} {l} {d}"

>>> f"{i:d} {f:0.1f} {s} {l!r} {d!r}"
```

Per riferimento, Python supporta anche qualificatori in stile C per la formattazione delle stringhe. Gli esempi di seguito sono equivalenti a quelli precedenti, ma le versioni `str.format` sono preferite a causa dei vantaggi in termini di flessibilità, coerenza della notazione ed estensibilità:

```
"%d %0.1f %s %r %r" % (i, f, s, l, d)

"%(i)d %(f)0.1f %(s)s %(l)r %(d)r" % dict(i=i, f=f, s=s, l=l, d=d)
```

Le graffe utilizzate per l'interpolazione in `str.format` possono anche essere numerate per ridurre la duplicazione durante la formattazione delle stringhe. Ad esempio, i seguenti sono equivalenti:

```
"I am from Australia. I love cupcakes from Australia!"
```

```
>>> "I am from {}. I love cupcakes from {}!".format("Australia", "Australia")

>>> "I am from {0}. I love cupcakes from {0}!".format("Australia")
```

Mentre la documentazione ufficiale di Python è, come al solito, abbastanza completa, pyformat.info ha una grande serie di esempi con spiegazioni dettagliate.

Inoltre, i caratteri `{` e `}` possono essere sfuggiti utilizzando le parentesi doppie:

```
"{'a': 5, 'b': 6}"
```

```
>>> "{{'{}': {}, '{}': {}}}".format("a", 5, "b", 6)

>>> f"{{'{'a'}': {5}, '{'b'}': {6}}"
```

Vedere Formattazione di stringhe per ulteriori informazioni. `str.format()` stato proposto in PEP 3101 e f-string in PEP 498 .

Numero di volte in cui una sottostringa viene visualizzata in una stringa

Un metodo è disponibile per il conteggio del numero di occorrenze di una sottostringa in un'altra stringa, `str.count` .

`str.count(sub[, start[, end]])`

`str.count` restituisce un `int` che indica il numero di occorrenze non sovrapposte alla sub-stringa `sub` in un'altra stringa. Gli argomenti opzionali `start` e `end` indicano l'inizio e la fine in cui avverrà la ricerca. Di default `start = 0` e `end = len(str)` significa che verrà cercata l'intera stringa:

```
>>> s = "She sells seashells by the seashore."
>>> s.count("sh")
2
>>> s.count("se")
3
>>> s.count("sea")
2
>>> s.count("seashells")
1
```

Specificando un valore diverso per `start` , `end` possiamo ottenere una ricerca più localizzata e contare, ad esempio, se `start` è uguale a `13` la chiamata a:

```
>>> s.count("sea", start)
1
```

è equivalente a:

```
>>> t = s[start:]
>>> t.count("sea")
1
```

Prova i caratteri iniziali e finali di una stringa

Per testare l'inizio e la fine di una data stringa in Python, si possono usare i metodi `str.startswith()` e `str.endswith()` .

`str.startswith(prefix[, start[, end]])`

Come suggerisce il nome, `str.startswith` viene utilizzato per verificare se una determinata stringa inizia con i caratteri specificati nel `prefix` .

```
>>> s = "This is a test string"
>>> s.startswith("T")
True
>>> s.startswith("Thi")
True
>>> s.startswith("thi")
False
```

Gli argomenti facoltativi di `start` e `end` specificano i punti di inizio e di fine da cui inizierà e terminerà il test. Nell'esempio seguente, specificando un valore iniziale di `2` nostra stringa verrà ricercata dalla posizione `2` e in seguito:

```
>>> s.startswith("is", 2)
True
```

Questo produce `True` poiché `s[2] == 'i'` and `s[3] == 's'` .

Puoi anche usare una `tuple` per verificare se inizia con una serie di stringhe

```
>>> s.startswith(('This', 'That'))
True
>>> s.startswith(('ab', 'bc'))
False
```

`str.endswith(prefix[, start[, end]])`

`str.endswith` è esattamente simile a `str.startswith` con la sola differenza che cerca caratteri finali e non caratteri iniziali. Ad esempio, per verificare se una stringa termina con un punto, si potrebbe scrivere:

```
>>> s = "this ends in a full stop."
>>> s.endswith('.')
True
>>> s.endswith('!')
False
```

come con `startswith` più di un carattere può essere usato come sequenza finale:

```
>>> s.endswith('stop.')
```

```
True
>>> s.endswith('Stop.')
False
```

Puoi anche usare una `tuple` per verificare se termina con una serie di stringhe

```
>>> s.endswith(('.', 'something'))
True
>>> s.endswith(('ab', 'bc'))
False
```

Test di cosa è composta una stringa

Il tipo `str` di Python include anche numerosi metodi che possono essere utilizzati per valutare il contenuto di una stringa. Questi sono `str.isalpha` , `str.isdigit` , `str.isalnum` , `str.isspace` . Le maiuscole possono essere testate con `str.isupper` , `str.islower` e `str.istitle` .

`str.isalpha`

`str.isalpha` non accetta argomenti e restituisce `True` se tutti i caratteri in una determinata stringa sono alfabetici, ad esempio:

```
>>> "Hello World".isalpha()  # contains a space
False
>>> "Hello2World".isalpha()  # contains a number
False
>>> "HelloWorld!".isalpha()  # contains punctuation
False
>>> "HelloWorld".isalpha()
True
```

Come caso limite, la stringa vuota `"".isalpha() False` se utilizzata con `"".isalpha()` .

`str.isupper` , `str.islower` , `str.istitle`

Questi metodi verificano la maiuscola in una determinata stringa.

`str.isupper` è un metodo che restituisce `True` se tutti i caratteri di una data stringa sono maiuscoli e `False` altrimenti.

```
>>> "HeLLO WORLD".isupper()
False
>>> "HELLO WORLD".isupper()
True
>>> "".isupper()
False
```

Al contrario, `str.islower` è un metodo che restituisce `True` se tutti i caratteri in una data stringa sono in minuscolo e `False` altrimenti.

```
>>> "Hello world".islower()
False
>>> "hello world".islower()
True
>>> "".islower()
False
```

`str.istitle` restituisce `True` se la stringa data è title cased; cioè, ogni parola inizia con un carattere maiuscolo seguito da caratteri minuscoli.

```
>>> "hello world".istitle()
False
>>> "Hello world".istitle()
False
>>> "Hello World".istitle()
True
>>> "".istitle()
False
```

`str.isdecimal` , `str.isdigit` , `str.isnumeric`

`str.isdecimal` restituisce se la stringa è una sequenza di cifre decimali, adatta a rappresentare un numero decimale.

`str.isdigit` include cifre non in una forma adatta a rappresentare un numero decimale, come le cifre in apice.

`str.isnumeric` include tutti i valori numerici, anche se non le cifre, come valori al di fuori dell'intervallo 0-9.

```
            isdecimal    isdigit   isnumeric

12345        True         True       True
□2□□5        True         True       True
①²³□₅       False        True       True
⑩□           False        False      True
Five         False        False      False
```

Bytestrings (`bytes` in Python 3, `str` in Python 2), supportano solo `isdigit` , che controlla solo le cifre ASCII di base.

Come con `str.isalpha` , la stringa vuota `str.isalpha` `False` .

`str.isalnum`

Questa è una combinazione di `str.isalpha` e `str.isnumeric` , in particolare valuta su `True` se tutti i caratteri nella stringa data sono **alfanumerici** , ovvero sono costituiti da caratteri alfabetici *o* numerici:

```
>>> "Hello2World".isalnum()
```

```
True
>>> "HelloWorld".isalnum()
True
>>> "2016".isalnum()
True
>>> "Hello World".isalnum()  # contains whitespace
False
```

---

str.isspace

Valuta `True` se la stringa contiene solo caratteri di spaziatura.

```
>>> "\t\r\n".isspace()
True
>>> " ".isspace()
True
```

A volte una stringa sembra "vuota" ma non sappiamo se è perché contiene solo spazi bianchi o nessun carattere

```
>>> "".isspace()
False
```

Per coprire questo caso abbiamo bisogno di un test aggiuntivo

```
>>> my_str = ''
>>> my_str.isspace()
False
>>> my_str.isspace() or not my_str
True
```

Ma il modo più breve per verificare se una stringa è vuota o contiene solo caratteri di spaziatura è usare la `strip` (senza argomenti rimuove tutti i caratteri di spaziatura iniziali e finali)

```
>>> not my_str.strip()
True
```

str.translate: traduzione di caratteri in una stringa

Python supporta un metodo di `translate` sul tipo `str` che consente di specificare la tabella di conversione (utilizzata per le sostituzioni) e qualsiasi carattere che dovrebbe essere eliminato nel processo.

```
str.translate(table[, deletechars])
```

Parametro	Descrizione
`table`	È una tabella di ricerca che definisce la mappatura da un personaggio all'altro.
`deletechars`	Un elenco di caratteri che devono essere rimossi dalla stringa.

Il metodo `maketrans` (`str.maketrans` in Python 3 e `string.maketrans` in Python 2) consente di generare una tabella di traduzione.

```
>>> translation_table = str.maketrans("aeiou", "12345")
>>> my_string = "This is a string!"
>>> translated = my_string.translate(translation_table)
'Th3s 3s 1 str3ng!'
```

Il metodo `translate` restituisce una stringa che è una copia tradotta della stringa originale.

È possibile impostare l'argomento `table` su `None` se è necessario solo cancellare caratteri.

```
>>> 'this syntax is very useful'.translate(None, 'aeiou')
'ths syntx s vry sfl'
```

Eliminazione di caratteri iniziali / finali indesiderati da una stringa

Vengono forniti tre metodi che offrono la possibilità di `str.strip` `str.rstrip` e finali da una stringa: `str.strip` , `str.rstrip` e `str.lstrip` . Tutti e tre i metodi hanno la stessa firma e tutti e tre restituiscono un nuovo oggetto stringa con caratteri indesiderati rimossi.

`str.strip([chars])`

`str.strip` agisce su una determinata stringa e rimuove (strisce) qualsiasi carattere iniziale o finale contenuto nei `chars` argomento; se i `chars` non vengono forniti o è `None` , tutti i caratteri dello spazio bianco vengono rimossi per impostazione predefinita. Per esempio:

```
>>> "    a line with leading and trailing space     ".strip()
'a line with leading and trailing space'
```

Se vengono forniti `chars` , tutti i caratteri in esso contenuti vengono rimossi dalla stringa, che viene restituita. Per esempio:

```
>>> ">>> a Python prompt".strip('> ')  # strips '>' character and space character
'a Python prompt'
```

`str.rstrip([chars])` e `str.lstrip([chars])`

Questi metodi hanno semantica e argomenti simili con `str.strip()` , la loro differenza sta nella direzione da cui partono. `str.rstrip()` inizia dalla fine della stringa mentre `str.lstrip()` divide dall'inizio della stringa.

Ad esempio, utilizzando `str.rstrip` :

```
>>> "     spacious string      ".rstrip()
'     spacious string'
```

Mentre, usando `str.lstrip` :

```
>>> "     spacious string      ".rstrip()
'spacious string      '
```

Confronto tra stringhe senza confronti tra maiuscole e minuscole

Confrontando la stringa in un modo insensibile al caso sembra qualcosa di banale, ma non lo è. Questa sezione considera solo stringhe unicode (il default in Python 3). Si noti che Python 2 può avere debolezze sottili rispetto a Python 3 - la successiva gestione unicode è molto più completa.

La prima cosa da notare è che le conversioni di rimozione del caso in Unicode non sono banali. C'è del testo per il quale `text.lower() != text.upper().lower()` , come `"ß"` :

```
>>> "ß".lower()
'ß'

>>> "ß".upper().lower()
'ss'
```

Ma diciamo che si voleva maiuscole e minuscole confrontare `"BUSSE"` e `"Buße"` . Diamine, probabilmente vuoi anche confrontare `"BUSSE"` e `"BU□E"` uguale - questa è la nuova forma maiuscola. Il modo consigliato è utilizzare la `casefold` :

Python 3.x 3.3

```
>>> help(str.casefold)
"""
Help on method_descriptor:

casefold(...)
      S.casefold() -> str

     Return a version of S suitable for caseless comparisons.
"""
```

Non usare solo `lower` . Se la `casefold` e `casefold` non è disponibile, facendo `.upper().lower()` aiuta (ma solo un po ').

Quindi dovresti considerare gli accenti. Se il tuo renderer dei font è buono, probabilmente pensi `"ê" == "ê "` - ma non lo fa:

```
>>> "ê" == "ê "
False
```

Questo perché sono in realtà

```
>>> import unicodedata

>>> [unicodedata.name(char) for char in "ê"]
['LATIN SMALL LETTER E WITH CIRCUMFLEX']
```

```
>>> [unicodedata.name(char) for char in "ê "]
['LATIN SMALL LETTER E', 'COMBINING CIRCUMFLEX ACCENT']
```

Il modo più semplice per `unicodedata.normalize` è `unicodedata.normalize` . Probabilmente vuoi usare la normalizzazione **NFKD** , ma sentiti libero di controllare la documentazione. Quindi si

```
>>> unicodedata.normalize("NFKD", "ê") == unicodedata.normalize("NFKD", "ê ")
True
```

Per finire, qui questo è espresso in funzioni:

```
import unicodedata

def normalize_caseless(text):
    return unicodedata.normalize("NFKD", text.casefold())

def caseless_equal(left, right):
    return normalize_caseless(left) == normalize_caseless(right)
```

Unisci un elenco di stringhe in un'unica stringa

Una stringa può essere utilizzata come separatore per unire un elenco di stringhe insieme in una singola stringa utilizzando il metodo `join()` . Ad esempio è possibile creare una stringa in cui ogni elemento di una lista è separato da uno spazio.

```
>>> " ".join(["once","upon","a","time"])
"once upon a time"
```

L'esempio seguente separa gli elementi stringa con tre trattini.

```
>>> "---".join(["once", "upon", "a", "time"])
"once---upon---a---time"
```

Costanti utili del modulo stringa

Il modulo di `string` di Python fornisce costanti per le operazioni legate alle stringhe. Per usarli, importa il modulo `string` :

```
>>> import string
```

`string.ascii_letters` :

Concatenazione di `ascii_lowercase` e `ascii_uppercase` :

```
>>> string.ascii_letters
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

---

string.ascii_lowercase :

Contiene tutti i caratteri ASCII in minuscolo:

```
>>> string.ascii_lowercase
'abcdefghijklmnopqrstuvwxyz'
```

---

string.ascii_uppercase :

Contiene tutti i caratteri ASCII maiuscoli:

```
>>> string.ascii_uppercase
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

---

string.digits :

Contiene tutti i caratteri decimali:

```
>>> string.digits
'0123456789'
```

---

string.hexdigits :

Contiene tutti i caratteri a cifre esadecimali:

```
>>> string.hexdigits
'0123456789abcdefABCDEF'
```

---

string.octaldigits :

Contiene tutti i caratteri ottali delle cifre:

```
>>> string.octaldigits
'01234567'
```

---

string.punctuation :

Contiene tutti i caratteri che sono considerati segni di punteggiatura nella locale `C` :

```
>>> string.punctuation
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

---

string.whitespace

:

Contiene tutti i caratteri ASCII considerati spazi bianchi:

```
>>> string.whitespace
' \t\n\r\x0b\x0c'
```

In modalità script, `print(string.whitespace)` stamperà i caratteri reali, usa `str` per ottenere la stringa restituita sopra.

`string.printable` :

Contiene tutti i caratteri che sono considerati stampabili; una combinazione di `string.digits` , `string.ascii_letters` , `string.punctuation` e `string.whitespace` .

```
>>> string.printable
'0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&\'()*+,-
./:;<=>?@[\\]^_`{|}~ \t\n\r\x0b\x0c'
```

Inversione di una stringa

Una stringa può essere invertita usando la funzione built-in reverse `reversed()` , che prende una stringa e restituisce un iteratore in ordine inverso.

```
>>> reversed('hello')
<reversed object at 0x0000000000000000>
>>> [char for char in reversed('hello')]
['o', 'l', 'l', 'e', 'h']
```

`reversed()` può essere racchiuso in una chiamata a `''.join()` per creare una stringa dall'iteratore.

```
>>> ''.join(reversed('hello'))
'olleh'
```

Mentre l'uso di `reversed()` potrebbe essere più leggibile agli utenti non esperti di Python, l'uso di slicing esteso con un passo di `-1` è più veloce e più conciso. Qui, prova ad implementarlo come funzione:

```
>>> def reversed_string(main_string):
...     return main_string[::-1]
...
>>> reversed_string('hello')
'olleh'
```

Giustificare le corde

Python fornisce funzioni per giustificare le stringhe, consentendo al padding del testo di rendere molto più semplice l'allineamento di varie stringhe.

Di seguito è riportato un esempio di `str.ljust` e `str.rjust` :

```
interstates_lengths = {
    5: (1381, 2222),
    19: (63, 102),
    40: (2555, 4112),
    93: (189,305),
}
for road, length in interstates_lengths.items():
    miles,kms = length
    print('{} -> {} mi. ({} km.)'.format(str(road).rjust(4), str(miles).ljust(4),
str(kms).ljust(4)))
```

```
  40 -> 2555 mi. (4112 km.)
  19 -> 63   mi. (102  km.)
   5 -> 1381 mi. (2222 km.)
  93 -> 189  mi. (305  km.)
```

`ljust` e le `rjust` sono molto simili. Entrambi hanno un parametro `width` e un parametro `fillchar` opzionale. Qualsiasi stringa creata da queste funzioni è lunga almeno quanto il parametro `width` passato nella funzione. Se la stringa è più lunga della `width` , non viene troncata. L'argomento `fillchar` , che ha come valore predefinito il carattere spazio `' '` deve essere un singolo carattere, non una stringa multicharacter.

La `ljust` funzione pads alla fine della stringa è chiamato con il `fillchar` finché è `width` lungo caratteri. La funzione `rjust` l'inizio della stringa in modo simile. Pertanto, la `l` e `r` nei nomi di queste funzioni si riferiscono al lato che la stringa originale, *non il* `fillchar` , è posizionata nella stringa di output.

Conversione tra str o byte di dati e caratteri Unicode

Il contenuto di file e messaggi di rete può rappresentare caratteri codificati. Spesso devono essere convertiti in unicode per una visualizzazione corretta.

In Python 2, potrebbe essere necessario convertire i dati str in caratteri Unicode. Il valore predefinito (`''` , `""` , ecc.) È una stringa ASCII, con qualsiasi valore esterno all'intervallo ASCII visualizzato come valori di escape. Le stringhe Unicode sono `u''` (o `u""` , ecc.).

Python 2.x 2.3

```
# You get "© abc" encoded in UTF-8 from a file, network, or other data source

s = '\xc2\xa9 abc'  # s is a byte array, not a string of characters
                    # Doesn't know the original was UTF-8
                    # Default form of string literals in Python 2
s[0]                # '\xc2' - meaningless byte (without context such as an encoding)
type(s)             # str - even though it's not a useful one w/o having a known encoding

u = s.decode('utf-8')  # u'\xa9 abc'
                       # Now we have a Unicode string, which can be read as UTF-8 and printed
properly
                       # In Python 2, Unicode string literals need a leading u
                       # str.decode converts a string which may contain escaped bytes to a
```

```
Unicode string
u[0]                # u'\xa9' - Unicode Character 'COPYRIGHT SIGN' (U+00A9) '©'
type(u)             # unicode

u.encode('utf-8')   # '\xc2\xa9 abc'
                    # unicode.encode produces a string with escaped bytes for non-ASCII
characters
```

In Python 3 potrebbe essere necessario convertire array di byte (indicati come "byte letterali") in stringhe di caratteri Unicode. L'impostazione predefinita è ora una stringa Unicode e ora i valori letterali devono essere immessi come `b''` , `b""` , ecc. Un byte letterale restituirà `True` a `isinstance(some_val, byte)` , assumendo `some_val` come stringa che potrebbe essere codificata come byte.

Python 3.x 3.0

```
# You get from file or network "© abc" encoded in UTF-8

s = b'\xc2\xa9 abc' # s is a byte array, not characters
                    # In Python 3, the default string literal is Unicode; byte array literals
need a leading b
s[0]                # b'\xc2' - meaningless byte (without context such as an encoding)
type(s)             # bytes - now that byte arrays are explicit, Python can show that.

u = s.decode('utf-8')  # '© abc' on a Unicode terminal
                       # bytes.decode converts a byte array to a string (which will, in Python
3, be Unicode)
u[0]                # '\u00a9' - Unicode Character 'COPYRIGHT SIGN' (U+00A9) '©'
type(u)             # str
                    # The default string literal in Python 3 is UTF-8 Unicode

u.encode('utf-8')   # b'\xc2\xa9 abc'
                    # str.encode produces a byte array, showing ASCII-range bytes as unescaped
characters.
```

La stringa contiene

Python rende estremamente intuitivo verificare se una stringa contiene una sottostringa data. Basta usare il `in` dell'operatore:

```
>>> "foo" in "foo.baz.bar"
True
```

Nota: testare una stringa vuota risulterà sempre in `True` :

```
>>> "" in "test"
True
```

Leggi Metodi di stringa online: https://riptutorial.com/it/python/topic/278/metodi-di-stringa

Capitolo 110: Metodo Overriding

Examples

Metodo di base prioritario

Ecco un esempio di override di base in Python (per chiarezza e compatibilità con Python 2 e 3, usando la nuova classe di stile e `print` with `()`):

```
class Parent(object):
    def introduce(self):
        print("Hello!")

    def print_name(self):
        print("Parent")


class Child(Parent):
    def print_name(self):
        print("Child")


p = Parent()
c = Child()

p.introduce()
p.print_name()

c.introduce()
c.print_name()

$ python basic_override.py
Hello!
Parent
Hello!
Child
```

Quando viene creata la classe `Child` , eredita i metodi della classe `Parent` . Ciò significa che qualsiasi metodo che ha la classe genitore avrà anche la classe figlia. Nell'esempio, l' `introduce` è definita per la classe `Child` perché è definita per `Parent` , nonostante non sia stata definita esplicitamente nella definizione di classe di `Child` .

In questo esempio, l'override si verifica quando `Child` definisce il proprio metodo `print_name` . Se questo metodo non è stato dichiarato, allora `c.print_name()` avrebbe stampato `"Parent"` . Tuttavia, `Child` ha sovrascritto la `Parent` definizione di 's di `print_name` , e così ora su chiamando `c.print_name()` , la parola `"Child"` viene stampato.

Leggi Metodo Overriding online: https://riptutorial.com/it/python/topic/3131/metodo-overriding

Capitolo 111: mixins

Sintassi

- class *ClassName* (*MainClass* , *Mixin1* , *Mixin2* , ...): # Usato per dichiarare una classe con il nome *ClassName* , main (first) class *MainClass* e *mixins Mixin1* , *Mixin2* , ecc.
- class *ClassName* (*Mixin1* , *MainClass* , *Mixin2* , ...): # La classe 'main' non deve essere la prima classe; non c'è davvero differenza tra esso e il mixin

Osservazioni

Aggiungere un mixin a una classe assomiglia molto all'aggiunta di una superclasse, perché praticamente è solo questo. Un oggetto di una classe con il mixin *Foo* sarà anche un'istanza di *Foo* , e `isinstance(instance, Foo)` restituirà true

Examples

mixin

Un **Mixin** è un insieme di proprietà e metodi che possono essere utilizzati in classi diverse, che *non* provengono da una classe base. Nei linguaggi di programmazione orientata agli oggetti, in genere si utilizza l' *ereditarietà* per assegnare agli oggetti di classi diverse la stessa funzionalità; se un insieme di oggetti ha delle capacità, metti quell'abilità in una classe base da cui entrambi gli oggetti *ereditano* .

> Ad esempio, supponi di avere le classi `Car` , `Boat` e `Plane` . Gli oggetti di tutte queste classi hanno la possibilità di viaggiare, quindi ottengono la funzione di `travel` . In questo scenario, viaggiano tutti allo stesso modo di base; ottenendo un percorso e muovendolo lungo Per implementare questa funzione, è possibile derivare tutte le classi da `Vehicle` e inserire la funzione in tale classe condivisa:

```
class Vehicle(object):
   """A generic vehicle class."""

   def __init__(self, position):
       self.position = position

   def travel(self, destination):
       route = calculate_route(from=self.position, to=destination)
       self.move_along(route)

class Car(Vehicle):
   ...

class Boat(Vehicle):
   ...

class Plane(Vehicle):
```

```
    ...
```

Con questo codice è possibile chiamare il `travel` su un'auto (`car.travel("Montana")`), barca (`boat.travel("Hawaii")`), e plane (`plane.travel("France")`)

Tuttavia, cosa succede se si dispone di funzionalità che non sono disponibili per una classe base? Per esempio, vuoi dare a `Car` una radio e la possibilità di usarla per suonare una canzone su una stazione radio, con `play_song_on_station` , ma hai anche un `Clock` che può usare anche una radio. `Car` e `Clock` potrebbero condividere una classe base (`Machine`). Tuttavia, non tutte le macchine possono riprodurre canzoni; `Boat` e `Plane` non possono (almeno in questo esempio). Quindi come realizzi senza duplicare il codice? Puoi usare un mixin. In Python, dare una lezione a un mixin è semplice come aggiungerlo alla lista delle sottoclassi, come questa

```
class Foo(main_super, mixin): ...
```

`Foo` erediterà tutte le proprietà e i metodi di `main_super` , ma anche quelli di `mixin` .

Quindi, per dare alle classi `Car` e orologio la possibilità di usare una radio, potresti sovrascrivere `Car` dall'ultimo esempio e scrivere questo:

```
class RadioUserMixin(object):
    def __init__(self):
        self.radio = Radio()

    def play_song_on_station(self, station):
        self.radio.set_station(station)
        self.radio.play_song()

class Car(Vehicle, RadioUserMixin):
    ...

class Clock(Vehicle, RadioUserMixin):
    ...
```

Ora puoi chiamare `car.play_song_on_station(98.7)` e `clock.play_song_on_station(101.3)` , ma non qualcosa come `boat.play_song_on_station(100.5)`

La cosa importante con i mixin è che ti permettono di aggiungere funzionalità a oggetti molto diversi, che non condividono una sottoclasse "principale" con questa funzionalità, ma comunque ne condividono comunque il codice. Senza mixin, fare qualcosa come l'esempio sopra sarebbe molto più difficile e / o potrebbe richiedere alcune ripetizioni.

Metodi di sovrascrittura in Mixins

Le mixine sono una sorta di classe che viene utilizzata per "mescolare" proprietà e metodi extra in una classe. Questo di solito va bene perché molte volte le classi di mixin non si sovrappongono l'una all'altra, o i metodi della classe base. Ma se si sostituiscono metodi o proprietà nei propri mix, ciò può portare a risultati imprevisti perché in Python la gerarchia delle classi è definita da destra a sinistra.

Ad esempio, prendi le seguenti classi

```
class Mixin1(object):
    def test(self):
        print "Mixin1"

class Mixin2(object):
    def test(self):
        print "Mixin2"

class BaseClass(object):
    def test(self):
        print "Base"

class MyClass(BaseClass, Mixin1, Mixin2):
    pass
```

In questo caso la classe Mixin2 è la classe base, estesa da Mixin1 e infine da BaseClass. Pertanto, se eseguiamo il seguente frammento di codice:

```
>>> x = MyClass()
>>> x.test()
Base
```

Vediamo che il risultato restituito è dalla classe Base. Ciò può portare a errori imprevisti nella logica del codice e deve essere tenuto in considerazione e tenuto presente

Leggi mixins online: https://riptutorial.com/it/python/topic/4359/mixins

Capitolo 112: Modelli di progettazione

introduzione

Un modello di progettazione è una soluzione generale a un problema comune nello sviluppo del software. Questo argomento di documentazione è specificamente volto a fornire esempi di modelli di progettazione comuni in Python.

Examples

Modello di strategia

Questo modello di design si chiama Strategy Pattern. È usato per definire una famiglia di algoritmi, incapsula ognuno e li rende intercambiabili. Il modello di progettazione strategica consente ad un algoritmo di variare in modo indipendente dai client che lo utilizzano.

Ad esempio, gli animali possono "camminare" in molti modi diversi. Camminare potrebbe essere considerato una strategia implementata da diversi tipi di animali:

```
from types import MethodType


class Animal(object):

    def __init__(self, *args, **kwargs):
        self.name = kwargs.pop('name', None) or 'Animal'
        if kwargs.get('walk', None):
            self.walk = MethodType(kwargs.pop('walk'), self)

    def walk(self):
        """
        Cause animal instance to walk

        Walking funcionallity is a strategy, and is intended to
        be implemented separately by different types of animals.
        """
        message = '{} should implement a walk method'.format(
            self.__class__.__name__)
        raise NotImplementedError(message)


# Here are some different walking algorithms that can be used with Animal
def snake_walk(self):
    print('I am slithering side to side because I am a {}.'.format(self.name))

def four_legged_animal_walk(self):
    print('I am using all four of my legs to walk because I am a(n) {}.'.format(
        self.name))

def two_legged_animal_walk(self):
    print('I am standing up on my two legs to walk because I am a {}.'.format(
        self.name))
```

L'esecuzione di questo esempio produrrebbe il seguente risultato:

```
generic_animal = Animal()
king_cobra = Animal(name='King Cobra', walk=snake_walk)
elephant = Animal(name='Elephant', walk=four_legged_animal_walk)
kangaroo = Animal(name='Kangaroo', walk=two_legged_animal_walk)

kangaroo.walk()
elephant.walk()
king_cobra.walk()
# This one will Raise a NotImplementedError to let the programmer
# know that the walk method is intended to be used as a strategy.
generic_animal.walk()

    # OUTPUT:
    #
    # I am standing up on my two legs to walk because I am a Kangaroo.
    # I am using all four of my legs to walk because I am a(n) Elephant.
    # I am slithering side to side because I am a King Cobra.
    # Traceback (most recent call last):
    #   File "./strategy.py", line 56, in <module>
    #     generic_animal.walk()
    #   File "./strategy.py", line 30, in walk
    #     raise NotImplementedError(message)
    # NotImplementedError: Animal should implement a walk method
```

Nota che in linguaggi come C ++ o Java, questo modello è implementato usando una classe astratta o un'interfaccia per definire una strategia. In Python ha più senso definire solo esternamente alcune funzioni che possono essere aggiunte dinamicamente ad una classe usando `types.MethodType` .

Introduzione ai modelli di design e Singleton Pattern

I modelli di progettazione forniscono soluzioni ai `commonly occurring problems` nella progettazione del software. I modelli di progettazione sono stati introdotti per la prima volta da `GoF(Gang of Four)` dove hanno descritto gli schemi comuni come problemi che si verificano ripetutamente e soluzioni a tali problemi.

I modelli di progettazione hanno quattro elementi essenziali:

1. `The pattern name` è una maniglia che possiamo usare per descrivere un problema di progettazione, le sue soluzioni e le conseguenze in una parola o due.
2. `The problem` descrive quando applicare il modello.
3. `The solution` descrive gli elementi che compongono il design, le loro relazioni, responsabilità e collaborazioni.
4. `The consequences` sono i risultati e i trade-off dell'applicazione del modello.

Vantaggi dei modelli di progettazione:

1. Sono riutilizzabili in più progetti.
2. Il livello architettonico dei problemi può essere risolto
3. Sono testati nel tempo e ben collaudati, che è l'esperienza di sviluppatori e architetti
4. Hanno affidabilità e dipendenza

Gli schemi di progettazione possono essere classificati in tre categorie:

1. Modello creativo
2. Modello strutturale
3. Modello comportamentale

`Creational Pattern` creativo: si preoccupano di come l'oggetto può essere creato e isolano i dettagli della creazione dell'oggetto.

`Structural Pattern` : progettano la struttura di classi e oggetti in modo che possano comporsi per ottenere risultati più ampi.

`Behavioral Pattern` - Sono interessati all'interazione tra oggetti e alla responsabilità degli oggetti.

Singleton Pattern :

È un tipo di `creational pattern` che fornisce un meccanismo per avere solo uno e un oggetto di un determinato tipo e fornisce un punto di accesso globale.

Ad esempio Singleton può essere utilizzato nelle operazioni di database, dove vogliamo che l'oggetto del database mantenga la coerenza dei dati.

Implementazione

Possiamo implementare Singleton Pattern in Python creando solo un'istanza della classe Singleton e servendo di nuovo lo stesso oggetto.

```
class Singleton(object):
    def __new__(cls):
        # hasattr method checks if the class object an instance property or not.
        if not hasattr(cls, 'instance'):
            cls.instance = super(Singleton, cls).__new__(cls)
        return cls.instance

s = Singleton()
print ("Object created", s)

s1 = Singleton()
print ("Object2 created", s1)
```

Produzione:

```
('Object created', <__main__.Singleton object at 0x10a7cc310>)
('Object2 created', <__main__.Singleton object at 0x10a7cc310>)
```

Si noti che in linguaggi come C ++ o Java, questo modello viene implementato rendendo privato il costruttore e creando un metodo statico che esegue l'inizializzazione dell'oggetto. In questo modo, un oggetto viene creato alla prima chiamata e la classe restituisce lo stesso oggetto in seguito. Ma in Python, non abbiamo alcun modo per creare costruttori privati.

Modello di fabbrica

Il modello di fabbrica è anche un `Creational pattern` . Il termine `factory` indica che una classe è responsabile della creazione di oggetti di altri tipi. Esiste una classe che agisce come una fabbrica che ha oggetti e metodi ad essa associati. Il client crea un oggetto chiamando i metodi con determinati parametri e factory crea l'oggetto del tipo desiderato e lo restituisce al client.

```
from abc import ABCMeta, abstractmethod

class Music():
    __metaclass__ = ABCMeta
    @abstractmethod
    def do_play(self):
        pass

class Mp3(Music):
    def do_play(self):
        print ("Playing .mp3 music!")

class Ogg(Music):
    def do_play(self):
        print ("Playing .ogg music!")

class MusicFactory(object):
    def play_sound(self, object_type):
        return eval(object_type)().do_play()

if __name__ == "__main__":
    mf = MusicFactory()
    music = input("Which music you want to play Mp3 or Ogg")
    mf.play_sound(music)
```

Produzione:

```
Which music you want to play Mp3 or Ogg"Ogg"
Playing .ogg music!
```

`MusicFactory` è la classe di fabbrica qui che crea un oggetto di tipo `Mp3` o `Ogg` seconda della scelta dell'utente.

delega

L'oggetto proxy viene spesso utilizzato per garantire l'accesso protetto a un altro oggetto, che la logica aziendale interna non vogliamo inquinare con i requisiti di sicurezza.

Supponiamo di voler garantire che solo l'utente con autorizzazioni specifiche possa accedere alla risorsa.

Definizione del proxy: (garantisce che solo gli utenti che effettivamente possono vedere le prenotazioni saranno in grado di prenotare il servizio di prenotazione)

```
from datetime import date
from operator import attrgetter

class Proxy:
    def __init__(self, current_user, reservation_service):
```

```
        self.current_user = current_user
        self.reservation_service = reservation_service

    def highest_total_price_reservations(self, date_from, date_to, reservations_count):
        if self.current_user.can_see_reservations:
            return self.reservation_service.highest_total_price_reservations(
                date_from,
                date_to,
                reservations_count
              )
        else:
            return []

#Models and ReservationService:

class Reservation:
    def __init__(self, date, total_price):
        self.date = date
        self.total_price = total_price

class ReservationService:
    def highest_total_price_reservations(self, date_from, date_to, reservations_count):
        # normally it would be read from database/external service
        reservations = [
            Reservation(date(2014, 5, 15), 100),
            Reservation(date(2017, 5, 15), 10),
            Reservation(date(2017, 1, 15), 50)
        ]

        filtered_reservations = [r for r in reservations if (date_from <= r.date <= date_to)]

        sorted_reservations = sorted(filtered_reservations, key=attrgetter('total_price'),
reverse=True)

        return sorted_reservations[0:reservations_count]


class User:
    def __init__(self, can_see_reservations, name):
        self.can_see_reservations = can_see_reservations
        self.name = name

#Consumer service:

class StatsService:
    def __init__(self, reservation_service):
        self.reservation_service = reservation_service

    def year_top_100_reservations_average_total_price(self, year):
        reservations = self.reservation_service.highest_total_price_reservations(
            date(year, 1, 1),
            date(year, 12, 31),
            1
        )

        if len(reservations) > 0:
            total = sum(r.total_price for r in reservations)

            return total / len(reservations)
        else:
            return 0
```

```
#Test:
def test(user, year):
    reservations_service = Proxy(user, ReservationService())
    stats_service = StatsService(reservations_service)
    average_price = stats_service.year_top_100_reservations_average_total_price(year)
    print("{0} will see: {1}".format(user.name, average_price))

test(User(True, "John the Admin"), 2017)
test(User(False, "Guest"),         2017)
```

BENEFICI

- **stiamo evitando qualsiasi modifica in `ReservationService` quando vengono modificate le restrizioni di accesso.**
- **non stiamo mescolando i dati relativi al business (`date_from` , `date_to` , `reservations_count`) con concetti non collegati al dominio (permessi dell'utente) in servizio.**
- **Anche Consumer (`StatsService`) è libero dalla logica relativa alle autorizzazioni**

CAVEATS

- L'interfaccia proxy è sempre esattamente uguale all'oggetto che nasconde, quindi l'utente che utilizza il servizio fornito dal proxy non era nemmeno a conoscenza della presenza del proxy.

Leggi Modelli di progettazione online: https://riptutorial.com/it/python/topic/8056/modelli-di-progettazione

Capitolo 113: Modelli in python

Examples

Semplice programma di output dei dati utilizzando il modello

```
from string import Template

data = dict(item = "candy", price = 8, qty = 2)

# define the template
t = Template("Simon bought $qty $item for $price dollar")
print(t.substitute(data))
```

Produzione:

```
Simon bought 2 candy for 8 dollar
```

I modelli supportano sostituzioni basate su $ invece di sostituzione basata su%. **Sostituisci** (mappatura, parole chiave) esegue la sostituzione del modello, restituendo una nuova stringa.

La mappatura è qualsiasi oggetto simile a un dizionario con chiavi che corrispondono ai segnaposti del modello. In questo esempio, prezzo e quantità sono segnaposto. Gli argomenti delle parole chiave possono anche essere utilizzati come segnaposti. I segnaposto delle parole chiave hanno la precedenza se entrambi sono presenti.

Modifica del delimitatore

È possibile modificare il delimitatore "$" su qualsiasi altro. Il seguente esempio:

```
from string import Template

class MyOtherTemplate(Template):
    delimiter = "#"


data = dict(id = 1, name = "Ricardo")
t = MyOtherTemplate("My name is #name and I have the id: #id")
print(t.substitute(data))
```

Puoi leggere de documenti qui

Leggi Modelli in python online: https://riptutorial.com/it/python/topic/6029/modelli-in-python

Capitolo 114: Modulo Asyncio

Examples

Sintassi di Coroutine e Delegazione

Prima che Python 3.5+ venisse rilasciato, il modulo `asyncio` usava i generatori per imitare le chiamate asincrone e quindi aveva una sintassi diversa rispetto all'attuale versione di Python 3.5.

Python 3.x 3.5

Python 3.5 ha introdotto le parole chiave `async` e `await` . Notare la mancanza di parentesi attorno alla chiamata `await func()` .

```
import asyncio

async def main():
    print(await func())

async def func():
    # Do time intensive stuff...
    return "Hello, world!"

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main())
```

Python 3.x 3.3 3.5

Prima di Python 3.5, il decoratore `@asyncio.coroutine` stato utilizzato per definire una coroutine. La resa da espressione è stata utilizzata per la delega del generatore. Notare le parentesi attorno al `yield from func()` .

```
import asyncio

@asyncio.coroutine
def main():
    print((yield from func()))

@asyncio.coroutine
def func():
    # Do time intensive stuff..
    return "Hello, world!"

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main())
```

Python 3.x 3.5

Ecco un esempio che mostra come due funzioni possono essere eseguite in modo asincrono:

```
import asyncio

async def cor1():
    print("cor1 start")
    for i in range(10):
        await asyncio.sleep(1.5)
        print("cor1", i)

async def cor2():
    print("cor2 start")
    for i in range(15):
        await asyncio.sleep(1)
        print("cor2", i)

loop = asyncio.get_event_loop()
cors = asyncio.wait([cor1(), cor2()])
loop.run_until_complete(cors)
```

Esecutori asincroni

Nota: utilizza Python 3.5+ async / attende la sintassi

`asyncio` supporta l'uso di oggetti `Executor` trovati in `concurrent.futures` per la pianificazione delle attività in modo asincrono. I loop di eventi hanno la funzione `run_in_executor()` che accetta un oggetto `Executor` , un `Callable` e i parametri di Callable.

Pianificazione di un'attività per un `Executor`

```
import asyncio
from concurrent.futures import ThreadPoolExecutor

def func(a, b):
    # Do time intensive stuff...
    return a + b

async def main(loop):
    executor = ThreadPoolExecutor()
    result = await loop.run_in_executor(executor, func, "Hello,", " world!")
    print(result)

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main(loop))
```

Ogni loop di eventi ha anche uno slot `Executor` "predefinito" che può essere assegnato a un `Executor` . Per assegnare un `Executor` e pianificare le attività dal ciclo, utilizzare il metodo `set_default_executor()` .

```
import asyncio
from concurrent.futures import ThreadPoolExecutor

def func(a, b):
    # Do time intensive stuff...
    return a + b
```

```
async def main(loop):
    # NOTE: Using `None` as the first parameter designates the `default` Executor.
    result = await loop.run_in_executor(None, func, "Hello,", " world!")
    print(result)

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.set_default_executor(ThreadPoolExecutor())
    loop.run_until_complete(main(loop))
```

Esistono due tipi principali di `Executor` in `concurrent.futures` , `ThreadPoolExecutor` e `ProcessPoolExecutor` . `ThreadPoolExecutor` contiene un pool di thread che può essere impostato manualmente su un numero specifico di thread tramite il costruttore o predefinito sul numero di core sulla macchina time 5. `ThreadPoolExecutor` utilizza il pool di thread per eseguire attività assegnate ad esso ed è generalmente meglio alle operazioni associate alla CPU piuttosto che alle operazioni associate all'I / O. Contrasto a `ProcessPoolExecutor` che genera un nuovo processo per ogni attività assegnata ad esso. `ProcessPoolExecutor` può solo eseguire attività e parametri che sono selezionabili. I compiti non selezionabili più comuni sono i metodi degli oggetti. Se è necessario pianificare il metodo di un oggetto come attività in un `Executor` è necessario utilizzare `ThreadPoolExecutor` .

Utilizzando UVLoop

`uvloop` è un'implementazione per `asyncio.AbstractEventLoop` basato su libuv (usato da nodejs). È conforme al 99% delle funzionalità di `asyncio` ed è molto più veloce rispetto al tradizionale `asyncio.EventLoop` . `uvloop` è attualmente disponibile su Windows, installalo con `pip install uvloop` .

```
import asyncio
import uvloop

if __name__ == "__main__":
    asyncio.set_event_loop(uvloop.new_event_loop())
    # Do your stuff here ...
```

Si può anche cambiare il factory del loop degli eventi impostando `EventLoopPolicy` su quello in `uvloop` .

```
import asyncio
import uvloop

if __name__ == "__main__":
    asyncio.set_event_loop_policy(uvloop.EventLoopPolicy())
    loop = asyncio.new_event_loop()
```

Sincronizzazione primitiva: evento

Concetto

Utilizzare un `Event` per **sincronizzare la pianificazione di più coroutine** .

In parole povere, un evento è come la pistola sparata a una corsa in corsa: lascia i corridori fuori dai blocchi di partenza.

Esempio

```
import asyncio

# event trigger function
def trigger(event):
    print('EVENT SET')
    event.set()  # wake up coroutines waiting

# event consumers
async def consumer_a(event):
    consumer_name = 'Consumer A'
    print('{} waiting'.format(consumer_name))
    await event.wait()
    print('{} triggered'.format(consumer_name))

async def consumer_b(event):
    consumer_name = 'Consumer B'
    print('{} waiting'.format(consumer_name))
    await event.wait()
    print('{} triggered'.format(consumer_name))

# event
event = asyncio.Event()

# wrap coroutines in one future
main_future = asyncio.wait([consumer_a(event),
                            consumer_b(event)])

# event loop
event_loop = asyncio.get_event_loop()
event_loop.call_later(0.1, functools.partial(trigger, event))  # trigger event in 0.1 sec

# complete main_future
done, pending = event_loop.run_until_complete(main_future)
```

Produzione:

Consumatore B in attesa
Consumatore A in attesa
SET EVENTO
Consumer B attivato
Consumatore A attivato

Un Websocket semplice

Qui creiamo un semplice web echo usando `asyncio` . Definiamo le coroutine per la connessione a un server e l'invio / ricezione di messaggi. Le comunicazioni del websocket vengono eseguite in una coroutine `main` , che viene eseguita da un ciclo di eventi. Questo esempio è stato modificato da un post precedente .

```
import asyncio
import aiohttp

session = aiohttp.ClientSession()                              # handles the context manager
class EchoWebsocket:

    async def connect(self):
        self.websocket = await session.ws_connect("wss://echo.websocket.org")

    async def send(self, message):
        self.websocket.send_str(message)

    async def receive(self):
        result = (await self.websocket.receive())
        return result.data

async def main():
    echo = EchoWebsocket()
    await echo.connect()
    await echo.send("Hello World!")
    print(await echo.receive())                                # "Hello World!"


if __name__ == '__main__':
    # The main loop
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main())
```

Idee sbagliate comuni sull'asyncio

probabilmente *il* più comune malinteso su `asnycio` è che ti permette di eseguire qualsiasi attività in parallelo - `asnycio` il GIL (global interpreter lock) e quindi eseguire processi di blocco in parallelo (su thread separati). **non** è così!

`asyncio` (e le librerie che sono costruite per collaborare con `asyncio`) si basano su coroutine: funzioni che (in modo collaborativo) restituiscono il flusso di controllo alla funzione chiamante. nota `asyncio.sleep` negli esempi sopra. questo è un esempio di una coroutine non bloccante che attende "in background" e restituisce il flusso di controllo alla funzione chiamante (quando chiamata con `await`). `time.sleep` è un esempio di una funzione di blocco. il flusso di esecuzione del programma si fermerà qui e ritorna solo dopo che il `time.sleep` è `time.sleep` .

un esempio reale è la libreria delle `requests` che consiste (per il momento) solo sulle funzioni di blocco. non c'è concorrenza se si chiama una delle sue funzioni all'interno di `asyncio` . `aiohttp` d'altra parte, è stato costruito `asyncio` ad `asyncio` . le sue coroutine verranno eseguite contemporaneamente.

- se hai attività con CPU a lungo termine che ti piacerebbe eseguire in parallelo, `asyncio` **non fa** per te. per questo hai bisogno di `threads` o `multiprocessing` .
- se sono in esecuzione lavori con `asyncio` IO, è *possibile* eseguirli contemporaneamente utilizzando `asyncio` .

Leggi Modulo Asyncio online: https://riptutorial.com/it/python/topic/1319/modulo-asyncio

Capitolo 115: Modulo casuale

Sintassi

- random.seed (a = None, version = 2) (la versione è disponibile solo per python 3.x)
- random.getstate ()
- random.setstate (stato)
- random.randint (a, b)
- random.randrange (stop)
- random.randrange (start, stop, step = 1)
- random.choice (ss)
- random.shuffle (x, random = random.random)
- random.sample (popolazione, k)

Examples

Casuale e sequenze: shuffle, scelta e campione

```
import random
```

Shuffle ()

Puoi usare `random.shuffle()` per mescolare / randomizzare gli elementi in una sequenza **mutabile e indicizzabile** . Ad esempio un `list` :

```
laughs = ["Hi", "Ho", "He"]

random.shuffle(laughs)     # Shuffles in-place! Don't do: laughs = random.shuffle(laughs)

print(laughs)
# Out: ["He", "Hi", "Ho"]  # Output may vary!
```

scelta()

Prende un elemento casuale da una **sequenza** arbitrale:

```
print(random.choice(laughs))
# Out: He                  # Output may vary!
```

campione()

Come `choice` prende elementi casuali da una **sequenza** arbitrale ma puoi specificare quanti:

```
#                   |--sequence--|--number--|
print(random.sample(    laughs    ,     1    ))  # Take one element
# Out: ['Ho']                     # Output may vary!
```

non prenderà lo stesso elemento due volte:

```
print(random.sample(laughs, 3))  # Take 3 random element from the sequence.
# Out: ['Ho', 'He', 'Hi']        # Output may vary!

print(random.sample(laughs, 4))  # Take 4 random element from the 3-item sequence.
```

ValueError: campione più grande della popolazione

Creazione di numeri interi e float casuali: randint, randrange, random e uniform

```
import random
```

randInt ()

Restituisce un intero casuale tra `x` ed `y` (compreso):

```
random.randint(x, y)
```

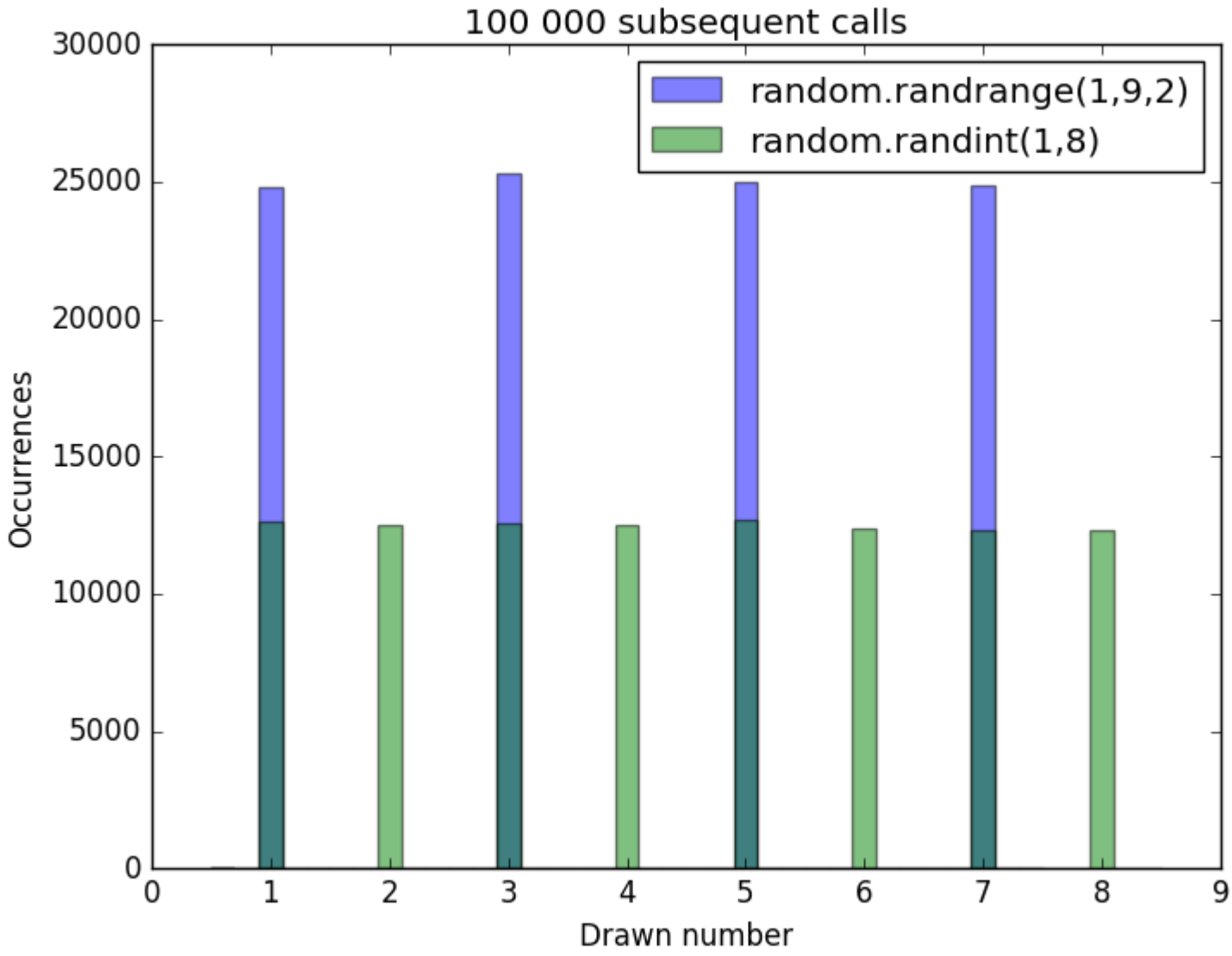
Ad esempio ottenendo un numero casuale compreso tra `1` e `8` :

```
random.randint(1, 8) # Out: 8
```

randrange ()

`random.randrange` ha la stessa sintassi `range` e diversamente da `random.randint` , l'ultimo valore **non** è compreso:

```
random.randrange(100)       # Random integer between 0 and 99
random.randrange(20, 50)    # Random integer between 20 and 49
random.rangrange(10, 20, 3) # Random integer between 10 and 19 with step 3 (10, 13, 16 and 19)
```



casuale

Restituisce un numero a virgola mobile casuale compreso tra 0 e 1:

```
random.random() # Out: 0.6648609321530631 7
```

uniforme

Restituisce un numero decimale casuale tra `x` ed `y` (compreso):

```
random.uniform(1, 8) # Out: 3.726062641730108
```

Numeri casuali riproducibili: seme e stato

L'impostazione di un seme specifico creerà una serie di numeri casuali fissi:

```
random.seed(5)                  # Create a fixed state
print(random.randrange(0, 10))  # Get a random integer between 0 and 9
# Out: 9
print(random.randrange(0, 10))
# Out: 4
```

La reimpostazione del seme creerà nuovamente la stessa sequenza "casuale":

```
random.seed(5)                  # Reset the random module to the same fixed state.
print(random.randrange(0, 10))
# Out: 9
print(random.randrange(0, 10))
# Out: 4
```

Poiché il seme è fisso, questi risultati sono sempre `9` e `4` . Se non è necessario avere numeri specifici, solo che i valori saranno gli stessi, è possibile utilizzare semplicemente `getstate` e `setstate` per ripristinare uno stato precedente:

```
save_state = random.getstate()  # Get the current state
print(random.randrange(0, 10))
# Out: 5
print(random.randrange(0, 10))
# Out: 8

random.setstate(save_state)     # Reset to saved state
print(random.randrange(0, 10))
# Out: 5
print(random.randrange(0, 10))
# Out: 8
```

Per pseudo-randomizzare nuovamente la sequenza, fai il `seed` con `None` :

```
random.seed(None)
```

Oppure chiama il metodo `seed` senza argomenti:

```
random.seed()
```

Crea numeri casuali crittograficamente sicuri

Di default il modulo random Python usa il MNSEN Twister `PRNG` per generare numeri casuali, che, sebbene adatti in domini come le simulazioni, non riescono a soddisfare i requisiti di sicurezza in ambienti più esigenti.

Per creare un numero pseudocasuale crittograficamente sicuro, si può usare `SystemRandom` che, usando `os.urandom` , è in grado di agire come un generatore di numeri pseudocasuali crittograficamente sicuro, CPRNG .

Il modo più semplice per usarlo consiste semplicemente `SystemRandom` classe `SystemRandom` . I metodi forniti sono simili a quelli esportati dal modulo casuale.

```
from random import SystemRandom
secure_rand_gen = SystemRandom()
```

Per creare una sequenza casuale di 10 `int` nell'intervallo `[0, 20]` , è sufficiente chiamare `randrange()` :

```
print([secure_rand_gen.randrange(10) for i in range(10)])
# [9, 6, 9, 2, 2, 3, 8, 0, 9, 9]
```

Per creare un numero intero casuale in un determinato intervallo, è possibile utilizzare `randint` :

```
print(secure_rand_gen.randint(0, 20))
# 5
```

e, di conseguenza, per tutti gli altri metodi. L'interfaccia è esattamente la stessa, l'unica modifica è il generatore di numeri sottostante.

Puoi anche utilizzare `os.urandom` direttamente per ottenere byte casuali crittograficamente sicuri.

Creazione di una password utente casuale

Per creare una password utente casuale, possiamo usare i simboli forniti nel modulo `string` . Precisamente `punctuation` per i simboli di punteggiatura, `ascii_letters` per lettere e `digits` per cifre:

```
from string import punctuation, ascii_letters, digits
```

Possiamo quindi combinare tutti questi simboli in un nome chiamato `symbols` :

```
symbols = ascii_letters + digits + punctuation
```

Rimuovere uno di questi per creare un pool di simboli con meno elementi.

Dopo questo, possiamo usare `random.SystemRandom` per generare una password. Per una password di 10 lunghezze:

```
secure_random = random.SystemRandom()
password = "".join(secure_random.choice(symbols) for i in range(10))
print(password)  # '^@g;J?]M6e'
```

Nota che altre routine rese immediatamente disponibili dal modulo `random` - come `random.choice` , `random.randint` , ecc. - *non* sono *adatte* per scopi crittografici.

Dietro le quinte, queste routine utilizzano il Mersenne Twister PRNG , che non soddisfa i requisiti di un CSPRNG . Quindi, in particolare, non dovresti usarne nessuna per generare password che intendi utilizzare. Usa sempre un'istanza di `SystemRandom` come mostrato sopra.

Python 3.x 3.6

A partire da Python 3.6, è disponibile il modulo dei `secrets` , che espone funzionalità

crittograficamente sicure.

Citando la documentazione ufficiale , per generare *"una password alfanumerica di dieci caratteri con almeno un carattere minuscolo, almeno un carattere maiuscolo e almeno tre cifre",* potresti:

```
import string
alphabet = string.ascii_letters + string.digits
while True:
    password = ''.join(choice(alphabet) for i in range(10))
    if (any(c.islower() for c in password)
            and any(c.isupper() for c in password)
            and sum(c.isdigit() for c in password) >= 3):
        break
```

Decisione binaria casuale

```
import random

probability = 0.3

if random.random() < probability:
    print("Decision with probability 0.3")
else:
    print("Decision with probability 0.7")
```

Leggi Modulo casuale online: https://riptutorial.com/it/python/topic/239/modulo-casuale

Capitolo 116: Modulo collezioni

introduzione

Il pacchetto di `collections` integrato offre diversi tipi di raccolta flessibili e specializzati che sono sia ad alte prestazioni che forniscono alternative ai tipi di raccolta generale di `dict` , `list` , `tuple` e `set` . Il modulo definisce anche classi di base astratte che descrivono diversi tipi di funzionalità di raccolta (come `MutableSet` e `ItemsView`).

Osservazioni

Ci sono altri tre tipi disponibili nel modulo **raccolte** , vale a dire:

1. UserDict
2. Lista degli utenti
3. UserString

Ognuno di essi funge da involucro attorno all'oggetto legato, ad esempio *UserDict* funge da involucro attorno a un oggetto *dict* . In ogni caso, la classe simula il suo tipo con nome. I contenuti dell'istanza sono conservati in un oggetto di tipo regolare, accessibile tramite l'attributo data dell'istanza wrapper. In ciascuno di questi tre casi, la necessità di questi tipi è stata parzialmente soppiantata dalla possibilità di sottoclasse direttamente dal tipo di base; tuttavia, la classe wrapper può essere più semplice da utilizzare perché il tipo sottostante è accessibile come attributo.

Examples

collections.Counter

Counter è una sottoclasse di dict che consente di contare facilmente gli oggetti. Ha metodi di utilità per lavorare con le frequenze degli oggetti che si stanno contando.

```
import collections
counts = collections.Counter([1,2,3])
```

il codice precedente crea un oggetto, conteggi, che ha le frequenze di tutti gli elementi passati al costruttore. Questo esempio ha il valore `Counter({1: 1, 2: 1, 3: 1})`

Esempi di costruttore

Contatore di lettere

```
>>> collections.Counter('Happy Birthday')
Counter({'a': 2, 'p': 2, 'y': 2, 'i': 1, 'r': 1, 'B': 1, ' ': 1, 'H': 1, 'd': 1, 'h': 1, 't':
1})
```

Contatore di parole

```
>>> collections.Counter('I am Sam Sam I am That Sam-I-am That Sam-I-am! I do not like that
Sam-I-am'.split())
Counter({'I': 3, 'Sam': 2, 'Sam-I-am': 2, 'That': 2, 'am': 2, 'do': 1, 'Sam-I-am!': 1, 'that':
1, 'not': 1, 'like': 1})
```

ricette

```
>>> c = collections.Counter({'a': 4, 'b': 2, 'c': -2, 'd': 0})
```

Ottieni il conteggio dei singoli elementi

```
>>> c['a']
4
```

Imposta il conteggio del singolo elemento

```
>>> c['c'] = -3
>>> c
Counter({'a': 4, 'b': 2, 'd': 0, 'c': -3})
```

Ottieni il numero totale di elementi nel contatore (4 + 2 + 0 - 3)

```
>>> sum(c.itervalues())  # negative numbers are counted!
3
```

Ottieni elementi (vengono mantenuti solo quelli con contatore positivo)

```
>>> list(c.elements())
['a', 'a', 'a', 'a', 'b', 'b']
```

Rimuovi le chiavi con 0 o valore negativo

```
>>> c - collections.Counter()
Counter({'a': 4, 'b': 2})
```

Rimuovi tutto

```
>>> c.clear()
>>> c
Counter()
```

Aggiungi rimuovi singoli elementi

```
>>> c.update({'a': 3, 'b':3})
>>> c.update({'a': 2, 'c':2})  # adds to existing, sets if they don't exist
>>> c
Counter({'a': 5, 'b': 3, 'c': 2})
>>> c.subtract({'a': 3, 'b': 3, 'c': 3})  # subtracts (negative values are allowed)
```

```
>>> c
Counter({'a': 2, 'b': 0, 'c': -1})
```

collections.defaultdict

collections.defaultdict (default_factory) restituisce una sottoclasse di `dict` che ha un valore predefinito per le chiavi mancanti. L'argomento dovrebbe essere una funzione che restituisce il valore predefinito quando viene chiamato senza argomenti. Se non viene passato nulla, il valore predefinito è `None` .

```
>>> state_capitals = collections.defaultdict(str)
>>> state_capitals
defaultdict(<class 'str'>, {})
```

restituisce un riferimento ad un defaultdict che creerà un oggetto stringa con il suo metodo default_factory.

Un tipico uso di `defaultdict` è di usare uno dei tipi builtin come `str` , `int` , `list` o `dict` come default_factory, poiché restituiscono tipi vuoti quando vengono chiamati senza argomenti:

```
>>> str()
''
>>> int()
0
>>> list
[]
```

Chiamare il defaultdict con una chiave che non esiste non produce un errore come farebbe in un normale dizionario.

```
>>> state_capitals['Alaska']
''
>>> state_capitals
defaultdict(<class 'str'>, {'Alaska': ''})
```

Un altro esempio con `int` :

```
>>> fruit_counts = defaultdict(int)
>>> fruit_counts['apple'] += 2  # No errors should occur
>>> fruit_counts
default_dict(int, {'apple': 2})
>>> fruit_counts['banana']  # No errors should occur
0
>>> fruit_counts  # A new key is created
default_dict(int, {'apple': 2, 'banana': 0})
```

I metodi di dizionario normali funzionano con il dizionario predefinito

```
>>> state_capitals['Alabama'] = 'Montgomery'
>>> state_capitals
defaultdict(<class 'str'>, {'Alabama': 'Montgomery', 'Alaska': ''})
```

Usando `list` come default_factory creerai un elenco per ogni nuova chiave.

```
>>> s = [('NC', 'Raleigh'), ('VA', 'Richmond'), ('WA', 'Seattle'), ('NC', 'Asheville')]
>>> dd = collections.defaultdict(list)
>>> for k, v in s:
...     dd[k].append(v)
>>> dd
defaultdict(<class 'list'>,
    {'VA': ['Richmond'],
     'NC': ['Raleigh', 'Asheville'],
     'WA': ['Seattle']})
```

collections.OrderedDict

L'ordine delle chiavi nei dizionari Python è arbitrario: non sono regolati dall'ordine in cui vengono aggiunti.

Per esempio:

```
>>> d = {'foo': 5, 'bar': 6}
>>> print(d)
{'foo': 5, 'bar': 6}
>>> d['baz'] = 7
>>> print(a)
{'baz': 7, 'foo': 5, 'bar': 6}
>>> d['foobar'] = 8
>>> print(a)
{'baz': 7, 'foo': 5, 'bar': 6, 'foobar': 8}
```

(L'ordinamento arbitrario implicito sopra significa che potresti ottenere risultati diversi con il codice sopra a quello mostrato qui.)

L'ordine in cui appaiono le chiavi è l'ordine su cui verrebbero ripetuti, ad esempio utilizzando un ciclo `for` .

La classe `collections.OrderedDict` fornisce oggetti dizionario che mantengono l'ordine delle chiavi. `OrderedDict` s può essere creato come mostrato di seguito con una serie di elementi ordinati (qui, un elenco di coppie chiave-valore di tuple):

```
>>> from collections import OrderedDict
>>> d = OrderedDict([('foo', 5), ('bar', 6)])
>>> print(d)
OrderedDict([('foo', 5), ('bar', 6)])
>>> d['baz'] = 7
>>> print(d)
OrderedDict([('foo', 5), ('bar', 6), ('baz', 7)])
>>> d['foobar'] = 8
>>> print(d)
OrderedDict([('foo', 5), ('bar', 6), ('baz', 7), ('foobar', 8)])
```

Oppure possiamo creare un `OrderedDict` vuoto e quindi aggiungere elementi:

```
>>> o = OrderedDict()
>>> o['key1'] = "value1"
>>> o['key2'] = "value2"
>>> print(o)
OrderedDict([('key1', 'value1'), ('key2', 'value2')])
```

L'iterazione di un `OrderedDict` consente l'accesso tramite chiave nell'ordine in cui sono stati aggiunti.

Cosa succede se assegniamo un nuovo valore a una chiave esistente?

```
>>> d['foo'] = 4
>>> print(d)
OrderedDict([('foo', 4), ('bar', 6), ('baz', 7), ('foobar', 8)])
```

La chiave mantiene il suo posto originale in `OrderedDict` .

collections.namedtuple

Definisci un nuovo tipo `Person` usa `namedtuple` questo modo:

```
Person = namedtuple('Person', ['age', 'height', 'name'])
```

Il secondo argomento è l'elenco degli attributi che avrà la tupla. È possibile elencare questi attributi anche come stringa separata o separata da virgole:

```
Person = namedtuple('Person', 'age, height, name')
```

o

```
Person = namedtuple('Person', 'age height name')
```

Una volta definita, una tupla con nome può essere istanziata chiamando l'oggetto con i parametri necessari, ad esempio:

```
dave = Person(30, 178, 'Dave')
```

Gli argomenti con nome possono anche essere usati:

```
jack = Person(age=30, height=178, name='Jack S.')
```

Ora puoi accedere agli attributi di namedtuple:

```
print(jack.age)  # 30
print(jack.name)  # 'Jack S.'
```

Il primo argomento del costruttore namedtuple (nel nostro esempio `'Person'`) è il `typename` . È tipico usare la stessa parola per il costruttore e il typename, ma possono essere diversi:

```
Human = namedtuple('Person',  'age, height, name')
dave = Human(30, 178, 'Dave')
print(dave)  # yields: Person(age=30, height=178, name='Dave')
```

collections.deque

Restituisce un nuovo oggetto `deque` inizializzato da sinistra a destra (usando append ()) con i dati da iterable. Se iterable non è specificato, il nuovo `deque` è vuoto.

Deques sono una generalizzazione di stack e code (il nome è pronunciato "deck" ed è l'abbreviazione di "double-ended queue"). I Deques supportano gli accessi e gli schiocchi a prova di thread ed efficienza da entrambi i lati della `deque` con approssimativamente le stesse prestazioni O (1) in entrambe le direzioni.

Sebbene gli oggetti elenco supportino operazioni simili, sono ottimizzati per operazioni veloci a lunghezza fissa e sostengono costi di spostamento della memoria O (n) per operazioni pop (0) e insert (0, v) che cambiano sia la dimensione che la posizione della rappresentazione sottostante dei dati .

Novità nella versione 2.4.

Se `maxlen` non è specificato o è `None` , i deques possono raggiungere una lunghezza arbitraria. Altrimenti, la `deque` è limitata alla lunghezza massima specificata. Quando una `deque` lunghezza limitata è piena, quando vengono aggiunti nuovi elementi, un numero corrispondente di elementi viene scartato dall'estremità opposta. Gli accessori di lunghezza limitata offrono funzionalità simili al filtro di coda in Unix. Sono anche utili per tracciare le transazioni e altri pool di dati in cui interessa solo l'attività più recente.

Modificato nella versione 2.6: Aggiunto il parametro maxlen.

```
>>> from collections import deque
>>> d = deque('ghi')                 # make a new deque with three items
>>> for elem in d:                   # iterate over the deque's elements
...     print elem.upper()
G
H
I

>>> d.append('j')                    # add a new entry to the right side
>>> d.appendleft('f')                # add a new entry to the left side
>>> d                                # show the representation of the deque
deque(['f', 'g', 'h', 'i', 'j'])

>>> d.pop()                          # return and remove the rightmost item
'j'
>>> d.popleft()                      # return and remove the leftmost item
'f'
>>> list(d)                          # list the contents of the deque
['g', 'h', 'i']
>>> d[0]                             # peek at leftmost item
'g'
>>> d[-1]                            # peek at rightmost item
'i'
```

```
>>> list(reversed(d))               # list the contents of a deque in reverse
['i', 'h', 'g']
>>> 'h' in d                        # search the deque
True
>>> d.extend('jkl')                 # add multiple elements at once
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])
>>> d.rotate(1)                     # right rotation
>>> d
deque(['l', 'g', 'h', 'i', 'j', 'k'])
>>> d.rotate(-1)                    # left rotation
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])

>>> deque(reversed(d))              # make a new deque in reverse order
deque(['l', 'k', 'j', 'i', 'h', 'g'])
>>> d.clear()                       # empty the deque
>>> d.pop()                         # cannot pop from an empty deque
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in -toplevel-
    d.pop()
IndexError: pop from an empty deque

>>> d.extendleft('abc')             # extendleft() reverses the input order
>>> d
deque(['c', 'b', 'a'])
```

Fonte: https://docs.python.org/2/library/collections.html

collections.ChainMap

`ChainMap` è nuovo nella **versione 3.3**

Restituisce un nuovo oggetto `ChainMap` dato un numero di `maps` . Questo oggetto raggruppa più dict o altri mapping per creare un'unica vista aggiornabile.

`ChainMap` sono utili per gestire i contesti nidificati e gli overlay. Un esempio nel mondo Python si trova nell'implementazione della classe `Context` nel motore di template di Django. È utile per collegare rapidamente un numero di mappature in modo che il risultato possa essere trattato come una singola unità. Spesso è molto più veloce della creazione di un nuovo dizionario e dell'esecuzione di più chiamate `update()` .

Ogni volta che uno ha una catena di valori di ricerca, può esserci un caso per `ChainMap` . Un esempio include avere entrambi i valori specificati dall'utente e un dizionario di valori predefiniti. Un altro esempio sono le mappe dei parametri `POST` e `GET` trovate nell'uso del web, ad esempio Django o Flask. Attraverso l'uso di `ChainMap` si restituisce una vista combinata di due dizionari distinti.

L'elenco dei parametri delle `maps` è ordinato dalla ricerca per la prima all'ultima ricerca. Le ricerche eseguono la ricerca dei mapping sottostanti fino a quando non viene trovata una chiave. Al contrario, le scritture, gli aggiornamenti e le eliminazioni funzionano solo sulla prima mappatura.

```
import collections
```

```
# define two dictionaries with at least some keys overlapping.
dict1 = {'apple': 1, 'banana': 2}
dict2 = {'coconut': 1, 'date': 1, 'apple': 3}

# create two ChainMaps with different ordering of those dicts.
combined_dict = collections.ChainMap(dict1, dict2)
reverse_ordered_dict = collections.ChainMap(dict2, dict1)
```

Nota l'impatto dell'ordine su quale valore viene trovato per primo nella ricerca successiva

```
for k, v in combined_dict.items():
    print(k, v)

date 1
apple 1
banana 2
coconut 1

for k, v in reverse_ordered_dict.items():
    print(k, v)

date 1
apple 3
banana 2
coconut 1
```

Leggi Modulo collezioni online: https://riptutorial.com/it/python/topic/498/modulo-collezioni

Capitolo 117: Modulo Deque

Sintassi

- dq = deque () # Crea un deque vuoto
- dq = deque (iterable) # Crea un deque con alcuni elementi
- dq.append (oggetto) # Aggiunge l'oggetto alla destra del deque
- dq.appendleft (oggetto) # Aggiunge l'oggetto alla sinistra del deque
- dq.pop () -> oggetto # Rimuove e restituisce l'oggetto più giusto
- dq.popleft () -> oggetto # Rimuove e restituisce l'oggetto più a sinistra
- dq.extend (iterable) # Aggiunge alcuni elementi alla destra del deque
- dq.extendleft (iterable) # Aggiunge alcuni elementi a sinistra del deque

Parametri

Parametro	Dettagli
`iterable`	Crea il deque con gli elementi iniziali copiati da un altro iterabile.
`maxlen`	Limita la dimensione della deque, eliminando i vecchi elementi quando vengono aggiunti nuovi elementi.

Osservazioni

Questa classe è utile quando è necessario un oggetto simile a un elenco che consente operazioni rapide di accodamento e pop da entrambi i lati (il nome `deque` sta per " *coda a doppio attacco* ").

I metodi forniti sono davvero molto simili, tranne per il fatto che alcuni come `pop` , `append` o `extend` possono essere suffissi con `left` . La struttura dei dati di `deque` dovrebbe essere preferibile a un elenco se è necessario inserire ed eliminare frequentemente elementi a entrambe le estremità poiché consente di farlo in tempo costante O (1).

Examples

Deque di base usando

I metodi principali che sono utili con questa classe sono `popleft` e `appendleft`

```
from collections import deque

d = deque([1, 2, 3])
p = d.popleft()         # p = 1, d = deque([2, 3])
d.appendleft(5)         # d = deque([5, 2, 3])
```

limite dimensioni deque

Utilizzare il parametro `maxlen` durante la creazione di una deque per limitare le dimensioni della deque:

```
from collections import deque
d = deque(maxlen=3)  # only holds 3 items
d.append(1)  # deque([1])
d.append(2)  # deque([1, 2])
d.append(3)  # deque([1, 2, 3])
d.append(4)  # deque([2, 3, 4]) (1 is removed because its maxlen is 3)
```

Metodi disponibili in deque

Creazione deque vuota:

```
dl = deque()  # deque([]) creating empty deque
```

Creare deque con alcuni elementi:

```
dl = deque([1, 2, 3, 4])  # deque([1, 2, 3, 4])
```

Aggiungere un elemento a deque:

```
dl.append(5)  # deque([1, 2, 3, 4, 5])
```

Aggiunta elemento lato sinistro del deque:

```
dl.appendleft(0)  # deque([0, 1, 2, 3, 4, 5])
```

Aggiunta di un elenco di elementi da deque:

```
dl.extend([6, 7])  # deque([0, 1, 2, 3, 4, 5, 6, 7])
```

Aggiunta di un elenco di elementi dal lato sinistro:

```
dl.extendleft([-2, -1])  # deque([-1, -2, 0, 1, 2, 3, 4, 5, 6, 7])
```

L'utilizzo `.pop()` rimuoverà naturalmente un oggetto dal lato destro:

```
dl.pop()  # 7 => deque([-1, -2, 0, 1, 2, 3, 4, 5, 6])
```

Utilizzando l'elemento `.popleft()` per rimuovere un elemento dal lato sinistro:

```
dl.popleft()  # -1 deque([-2, 0, 1, 2, 3, 4, 5, 6])
```

Rimuovi elemento dal suo valore:

```
dl.remove(1)  # deque([-2, 0, 2, 3, 4, 5, 6])
```

Invertire l'ordine degli elementi nella nota:

```
dl.reverse()  # deque([6, 5, 4, 3, 2, 0, -2])
```

Larghezza Prima ricerca

Il Deque è l'unica struttura dati Python con operazioni di coda veloci. (Nota `queue.Queue` non è normalmente adatto, poiché è inteso per la comunicazione tra thread). Un caso d'uso di base di una coda è la prima ricerca in ampiezza .

```
from collections import deque

def bfs(graph, root):
    distances = {}
    distances[root] = 0
    q = deque([root])
    while q:
        # The oldest seen (but not yet visited) node will be the left most one.
        current = q.popleft()
        for neighbor in graph[current]:
            if neighbor not in distances:
                distances[neighbor] = distances[current] + 1
                # When we see a new node, we add it to the right side of the queue.
                q.append(neighbor)
    return distances
```

Supponiamo di avere un semplice grafico diretto:

```
graph = {1:[2,3], 2:[4], 3:[4,5], 4:[3,5], 5:[]}
```

Ora possiamo trovare le distanze da una posizione iniziale:

```
>>> bfs(graph, 1)
{1: 0, 2: 1, 3: 1, 4: 2, 5: 2}

>>> bfs(graph, 3)
{3: 0, 4: 1, 5: 1}
```

Leggi Modulo Deque online: https://riptutorial.com/it/python/topic/1976/modulo-deque

Capitolo 118: Modulo di coda

introduzione

Il modulo Queue implementa code multiproduttore e multi-consumer. È particolarmente utile nella programmazione con thread quando le informazioni devono essere scambiate in modo sicuro tra più thread. Esistono tre tipi di code forniti dal modulo code, che sono i seguenti: 1. Coda 2. LifoQueue 3. PriorityQueue Eccezione che potrebbe essere richiesta: 1. Completo (overflow della coda) 2. Vuoto (underflow della coda)

Examples

Semplice esempio

```
from Queue import Queue

question_queue = Queue()

for x in range(1,10):
    temp_dict = ('key', x)
    question_queue.put(temp_dict)

while(not question_queue.empty()):
    item = question_queue.get()
    print(str(item))
```

Produzione:

```
('key', 1)
('key', 2)
('key', 3)
('key', 4)
('key', 5)
('key', 6)
('key', 7)
('key', 8)
('key', 9)
```

Leggi Modulo di coda online: https://riptutorial.com/it/python/topic/8339/modulo-di-coda

Capitolo 119: Modulo Functools

Examples

parziale

La funzione `partial` crea un'applicazione di funzione parziale da un'altra funzione. Viene utilizzato per *associare* valori ad alcuni argomenti della funzione (o argomenti della parola chiave) e produrre un *callable* senza gli argomenti già definiti.

```
>>> from functools import partial
>>> unhex = partial(int, base=16)
>>> unhex.__doc__ = 'Convert base16 string to int'
>>> unhex('ca11ab1e')
3390155550
```

`partial()` , come suggerisce il nome, consente una valutazione parziale di una funzione. Diamo un'occhiata al seguente esempio:

```
In [2]: from functools import partial

In [3]: def f(a, b, c, x):
   ...:     return 1000*a + 100*b + 10*c + x
   ...:

In [4]: g = partial(f, 1, 1, 1)

In [5]: print g(2)
1112
```

Quando `g` viene creato, `f` , che prende quattro argomenti (`a, b, c, x`), viene anche parzialmente valutato per i primi tre argomenti, `a, b, c,` La valutazione di `f` è completata quando `g` è chiamato, `g(2)` , che passa il quarto argomento a `f` .

Un modo di pensare `partial` è un registro a scorrimento; spingendo in un argomento alla volta in qualche funzione. `partial` è utile per i casi in cui i dati arrivano come stream e non possiamo passare più di un argomento.

total_ordering

Quando vogliamo creare una classe ordinabile, normalmente dobbiamo definire i metodi `__eq()__` , `__lt__()` , `__le__()` , `__gt__()` e `__ge__()` .

Il decoratore `total_ordering` , applicato a una classe, consente la definizione di `__eq__()` e solo uno tra `__lt__()` , `__le__()` , `__gt__()` e `__ge__()` , e consente comunque tutte le operazioni di ordinamento sulla classe.

```
@total_ordering
```

```
class Employee:

    ...

    def __eq__(self, other):
        return ((self.surname, self.name) == (other.surname, other.name))

    def __lt__(self, other):
        return ((self.surname, self.name) < (other.surname, other.name))
```

Il decoratore usa una composizione dei metodi forniti e delle operazioni algebriche per ricavare gli altri metodi di confronto. Ad esempio, se abbiamo definito `__lt__()` e `__eq()__` e vogliamo derivare `__gt__()` , possiamo semplicemente controllare `not __lt__() and not __eq()__` .

Nota : la funzione `total_ordering` è disponibile solo da Python 2.7.

ridurre

In Python 3.x, la funzione di `reduce` già illustrata qui è stata rimossa dai built-in e deve ora essere importata da `functools` .

```
from functools import reduce
def factorial(n):
    return reduce(lambda a, b: (a*b), range(1, n+1))
```

lru_cache

Il decoratore `@lru_cache` può essere utilizzato per avvolgere una funzione costosa e ad alta intensità di calcolo con una cache meno recente . Ciò consente di memorizzare le chiamate di funzione, in modo che le chiamate future con gli stessi parametri possano tornare istantaneamente invece di dover essere ricalcolate.

```
@lru_cache(maxsize=None)  # Boundless cache
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

>>> fibonacci(15)
```

Nell'esempio sopra, il valore di `fibonacci(3)` viene calcolato solo una volta, mentre se `fibonacci` non ha una cache LRU, `fibonacci(3)` sarebbe stato calcolato verso l'alto di 230 volte. Quindi `@lru_cache` è particolarmente adatto per funzioni ricorsive o programmazione dinamica, in cui una funzione costosa può essere chiamata più volte con gli stessi parametri esatti.

`@lru_cache` ha due argomenti

- `maxsize` : numero di chiamate da salvare. Quando il numero di chiamate univoche supera il numero `maxsize` , la cache della LRU rimuoverà le chiamate utilizzate meno di recente.
- `typed` (aggiunto in 3.3): Flag per determinare se argomenti equivalenti di tipi diversi appartengono a diversi record della cache (cioè se `3.0` e `3` contano come argomenti diversi)

Possiamo anche vedere le statistiche della cache:

```
>>> fib.cache_info()
CacheInfo(hits=13, misses=16, maxsize=None, currsize=16)
```

NOTA : Poiché `@lru_cache` utilizza i dizionari per memorizzare i risultati nella cache, tutti i parametri per la funzione devono essere lavabili affinché la cache funzioni.

Documenti ufficiali di Python per `@lru_cache` . `@lru_cache` stato aggiunto in 3.2.

cmp_to_key

Python ha cambiato i suoi metodi di ordinamento per accettare una funzione chiave. Queste funzioni prendono un valore e restituiscono una chiave che viene utilizzata per ordinare gli array.

Vecchie funzioni di confronto usate per prendere due valori e restituiscono -1, 0 o +1 se il primo argomento è piccolo, uguale o maggiore del secondo argomento rispettivamente. Questo è incompatibile con la nuova funzione chiave.

Ecco dove arriva `functools.cmp_to_key` :

```
>>> import functools
>>> import locale
>>> sorted(["A", "S", "F", "D"], key=functools.cmp_to_key(locale.strcoll))
['A', 'D', 'F', 'S']
```

Esempio preso e adattato dalla documentazione della libreria standard Python .

Leggi Modulo Functools online: https://riptutorial.com/it/python/topic/2492/modulo-functools

Capitolo 120: Modulo Itertools

Sintassi

- `import itertools`

Examples

Raggruppamento di elementi da un oggetto iterabile mediante una funzione

Inizia con un iterabile che deve essere raggruppato

```
lst = [("a", 5, 6), ("b", 2, 4), ("a", 2, 5), ("c", 2, 6)]
```

Genera il generatore raggruppato, raggruppando il secondo elemento in ciascuna tupla:

```
def testGroupBy(lst):
    groups = itertools.groupby(lst, key=lambda x: x[1])
    for key, group in groups:
        print(key, list(group))

testGroupBy(lst)

# 5 [('a', 5, 6)]
# 2 [('b', 2, 4), ('a', 2, 5), ('c', 2, 6)]
```

Solo i gruppi di elementi consecutivi sono raggruppati. Potrebbe essere necessario ordinare la stessa chiave prima di chiamare groupby Per Eg, (L'ultimo elemento è cambiato)

```
lst = [("a", 5, 6), ("b", 2, 4), ("a", 2, 5), ("c", 5, 6)]
testGroupBy(lst)

# 5 [('a', 5, 6)]
# 2 [('b', 2, 4), ('a', 2, 5)]
# 5 [('c', 5, 6)]
```

Il gruppo restituito da groupby è un iteratore che non sarà valido prima della prossima iterazione. Ad esempio il seguente non funzionerà se si desidera che i gruppi siano ordinati per chiave. Il gruppo 5 è vuoto sotto perché quando viene recuperato il gruppo 2 invalida 5

```
lst = [("a", 5, 6), ("b", 2, 4), ("a", 2, 5), ("c", 2, 6)]
groups = itertools.groupby(lst, key=lambda x: x[1])
for key, group in sorted(groups):
    print(key, list(group))

# 2 [('c', 2, 6)]
# 5 []
```

Per eseguire correttamente l'ordinamento, creare un elenco dall'iteratore prima di ordinare

```
groups = itertools.groupby(lst, key=lambda x: x[1])
for key, group in sorted((key, list(group)) for key, group in groups):
    print(key, list(group))

# 2 [('b', 2, 4), ('a', 2, 5), ('c', 2, 6)]
# 5 [('a', 5, 6)]
```

Prendi una fetta di un generatore

Itertools "islice" ti permette di tagliare un generatore:

```
results = fetch_paged_results()  # returns a generator
limit = 20  # Only want the first 20 results
for data in itertools.islice(results, limit):
    print(data)
```

Normalmente non puoi tagliare un generatore:

```
def gen():
    n = 0
    while n < 20:
        n += 1
        yield n

for part in gen()[:3]:
    print(part)
```

Darà

```
Traceback (most recent call last):
  File "gen.py", line 6, in <module>
    for part in gen()[:3]:
TypeError: 'generator' object is not subscriptable
```

Tuttavia, questo funziona:

```
import itertools

def gen():
    n = 0
    while n < 20:
        n += 1
        yield n

for part in itertools.islice(gen(), 3):
    print(part)
```

Tieni presente che, come una normale slice, puoi anche utilizzare gli argomenti `start` , `stop` e `step` :

```
itertools.islice(iterable, 1, 30, 3)
```

itertools.product

Questa funzione consente di scorrere il prodotto cartesiano di un elenco di iterabili.

Per esempio,

```
for x, y in itertools.product(xrange(10), xrange(10)):
    print x, y
```

è equivalente a

```
for x in xrange(10):
    for y in xrange(10):
        print x, y
```

Come tutte le funzioni python che accettano un numero variabile di argomenti, possiamo passare una lista a itertools.product per decomprimere, con l'operatore *.

Così,

```
its = [xrange(10)] * 2
for x,y in itertools.product(*its):
    print x, y
```

produce gli stessi risultati di entrambi gli esempi precedenti.

```
>>> from itertools import product
>>> a=[1,2,3,4]
>>> b=['a','b','c']
>>> product(a,b)
<itertools.product object at 0x0000000002712F78>
>>> for i in product(a,b):
...     print i
...
(1, 'a')
(1, 'b')
(1, 'c')
(2, 'a')
(2, 'b')
(2, 'c')
(3, 'a')
(3, 'b')
(3, 'c')
(4, 'a')
(4, 'b')
(4, 'c')
```

itertools.count

Introduzione:

Questa semplice funzione genera serie infinite di numeri. Per esempio...

```
for number in itertools.count():
    if number > 20:
        break
    print(number)
```

Nota che dobbiamo rompere o stampare per sempre!

Produzione:

```
0
1
2
3
4
5
6
7
8
9
10
```

Argomenti:

`count()` accetta due argomenti, `start` e `step` :

```
for number in itertools.count(start=10, step=4):
    print(number)
    if number > 20:
        break
```

Produzione:

```
10
14
18
22
```

itertools.takewhile

itertools.takewhile consente di prendere elementi da una sequenza finché una condizione non diventa `False` .

```
def is_even(x):
    return x % 2 == 0


lst = [0, 2, 4, 12, 18, 13, 14, 22, 23, 44]
result = list(itertools.takewhile(is_even, lst))

print(result)
```

Questo produce `[0, 2, 4, 12, 18]` .

Nota che, il primo numero che viola il predicato (es: la funzione che restituisce un valore booleano) `is_even` è, `13` . Una volta che `takewhile` incontra un valore che produce `False` per il predicato specificato, esso scoppia.

L' **output prodotto** da `takewhile` è simile all'output generato dal codice sottostante.

```
def takewhile(predicate, iterable):
    for x in iterable:
        if predicate(x):
            yield x
        else:
            break
```

Nota: la concatenazione dei risultati prodotti da `takewhile` e `dropwhile` produce l'originale iterabile.

```
result = list(itertools.takewhile(is_even, lst)) + list(itertools.dropwhile(is_even, lst))
```

itertools.dropwhile

itertools.dropwhile consente di prelevare elementi da una sequenza dopo che una condizione diventa `False` .

```
def is_even(x):
    return x % 2 == 0


lst = [0, 2, 4, 12, 18, 13, 14, 22, 23, 44]
result = list(itertools.dropwhile(is_even, lst))

print(result)
```

Questo produce `[13, 14, 22, 23, 44]` .

(*Questo esempio è lo stesso dell'esempio per* `takewhile` *ma usando* `dropwhile` *.*)

Nota che, il primo numero che viola il predicato (es: la funzione che restituisce un valore booleano) `is_even` è, `13` . Tutti gli elementi prima di questo vengono scartati.

L' **output prodotto** da `dropwhile` è simile all'output generato dal codice seguente.

```
def dropwhile(predicate, iterable):
    iterable = iter(iterable)
    for x in iterable:
        if not predicate(x):
            yield x
            break
    for x in iterable:
        yield x
```

La concatenazione dei risultati prodotti da `takewhile` e `dropwhile` produce l'originale iterabile.

```
result = list(itertools.takewhile(is_even, lst)) + list(itertools.dropwhile(is_even, lst))
```

Zippare due iteratori finché non sono entrambi esauriti

Simile alla funzione built-in `zip()` , `itertools.zip_longest` continuerà a `itertools.zip_longest` oltre la fine del più breve di due iterabili.

```
from itertools import zip_longest
a = [i for i in range(5)] # Length is 5
b = ['a', 'b', 'c', 'd', 'e', 'f', 'g'] # Length is 7
for i in zip_longest(a, b):
    x, y = i  # Note that zip longest returns the values as a tuple
    print(x, y)
```

È possibile `fillvalue` argomento `fillvalue` facoltativo (predefinito su `''`) in questo modo:

```
for i in zip_longest(a, b, fillvalue='Hogwash!'):
    x, y = i  # Note that zip longest returns the values as a tuple
    print(x, y)
```

In Python 2.6 e 2.7, questa funzione è chiamata `itertools.izip_longest` .

Metodo di combinazioni nel modulo Itertools

`itertools.combinations` restituirà un generatore della sequenza *k* -combination di una lista.

In altre parole: restituirà un generatore di tuple di tutte le possibili combinazioni di k dell'elenco di input.

Per esempio:

Se hai una lista:

```
a = [1,2,3,4,5]
b = list(itertools.combinations(a, 2))
print b
```

Produzione:

```
[(1, 2), (1, 3), (1, 4), (1, 5), (2, 3), (2, 4), (2, 5), (3, 4), (3, 5), (4, 5)]
```

L'uscita di cui sopra è un generatore convertito in una lista di tuple tutte le possibili combinazioni di *coppia* -wise lista di input `a`

Puoi anche trovare tutte le 3 combinazioni:

```
a = [1,2,3,4,5]
b = list(itertools.combinations(a, 3))
print b
```

Produzione:

```
[(1, 2, 3), (1, 2, 4), (1, 2, 5), (1, 3, 4),
```

```
 (1, 3, 5), (1, 4, 5), (2, 3, 4), (2, 3, 5),
 (2, 4, 5), (3, 4, 5)]
```

Concatenare più iteratori insieme

Utilizzare `itertools.chain` per creare un singolo generatore che fornirà in sequenza i valori di diversi generatori.

```
from itertools import chain
a = (x for x in ['1', '2', '3', '4'])
b = (x for x in ['x', 'y', 'z'])
' '.join(chain(a, b))
```

Risultati in:

```
'1 2 3 4 x y z'
```

Come costruttore alternativo, puoi usare il metodo `chain.from_iterable` che prende come parametro singolo un iterabile di iterabili. Per ottenere lo stesso risultato di cui sopra:

```
' '.join(chain.from_iterable([a,b])
```

Mentre `chain` può assumere un numero arbitrario di argomenti, `chain.from_iterable` è l'unico modo per concatenare un numero *infinito* di iterabili.

itertools.repeat

Ripeti qualcosa n volte:

```
>>> import itertools
>>> for i in itertools.repeat('over-and-over', 3):
...     print(i)
over-and-over
over-and-over
over-and-over
```

Ottieni una somma cumulativa di numeri in un iterabile

Python 3.x 3.2

`accumulate` produce una somma cumulativa (o un prodotto) di numeri.

```
>>> import itertools as it
>>> import operator

>>> list(it.accumulate([1,2,3,4,5]))
[1, 3, 6, 10, 15]

>>> list(it.accumulate([1,2,3,4,5], func=operator.mul))
[1, 2, 6, 24, 120]
```

Passa attraverso gli elementi in un iteratore

`cycle` è un iteratore infinito.

```
>>> import itertools as it
>>> it.cycle('ABCD')
A B C D A B C D A B C D ...
```

Pertanto, fai attenzione a dare dei limiti quando usi questo per evitare un ciclo infinito. Esempio:

```
>>> # Iterate over each element in cycle for a fixed range
>>> cycle_iterator = it.cycle('abc123')
>>> [next(cycle_iterator) for i in range(0, 10)]
['a', 'b', 'c', '1', '2', '3', 'a', 'b', 'c', '1']
```

itertools.permutations

`itertools.permutations` restituisce un generatore con permutazioni di lunghezza-lunghezza successive di elementi nel iterabile.

```
a = [1,2,3]
list(itertools.permutations(a))
# [(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)]

list(itertools.permutations(a, 2))
[(1, 2), (1, 3), (2, 1), (2, 3), (3, 1), (3, 2)]
```

se l'elenco `a` ha elementi duplicati, le permutazioni risultanti avranno elementi duplicati, è possibile utilizzare `set` per ottenere permute uniche:

```
a = [1,2,1]
list(itertools.permutations(a))
# [(1, 2, 1), (1, 1, 2), (2, 1, 1), (2, 1, 1), (1, 1, 2), (1, 2, 1)]

set(itertools.permutations(a))
# {(1, 1, 2), (1, 2, 1), (2, 1, 1)}
```

Leggi Modulo Itertools online: https://riptutorial.com/it/python/topic/1564/modulo-itertools

Capitolo 121: Modulo JSON

Osservazioni

Per la documentazione completa inclusa la funzionalità specifica della versione, consultare la documentazione ufficiale .

tipi

Defaults

il modulo `json` gestirà la codifica e la decodifica dei seguenti tipi di default:

Tipi di serializzazione:

JSON	Pitone
oggetto	dict
schieramento	elenco
stringa	str
numero (int)	int
numero (reale)	galleggiante
vero falso	Vero falso
nullo	Nessuna

Il modulo `json` comprende anche `NaN` , `Infinity` e `-Infinity` come valori float corrispondenti, che è al di fuori della specifica JSON.

Tipi di serializzazione:

Pitone	JSON
dict	oggetto
lista, tupla	schieramento
str	stringa
int, float, (int / float) -derived Enums	numero

Pitone	JSON
Vero	vero
falso	falso
Nessuna	nullo

Per disabilitare la codifica di `NaN` , `Infinity` e `-Infinity` devi codificare con `allow_nan=False` . Ciò solleverà `ValueError` se `ValueError` di codificare questi valori.

Serializzazione personalizzata (de-)

Esistono vari hook che consentono di gestire i dati che devono essere rappresentati in modo diverso. L'uso di `functools.partial` consente di applicare parzialmente i parametri rilevanti a queste funzioni per comodità.

serializzazione:

È possibile fornire una funzione che opera sugli oggetti prima di essere serializzati in questo modo:

```
# my_json module

import json
from functools import partial

def serialise_object(obj):
    # Do something to produce json-serialisable data
    return dict_obj

dump = partial(json.dump, default=serialise_object)
dumps = partial(json.dumps, default=serialise_object)
```

De-serializzazione:

Ci sono vari hook che sono gestiti dalle funzioni JSON, come object_hook e parse_float. Per un elenco completo per la tua versione di Python, vedi qui .

```
# my_json module

import json
from functools import partial

def deserialise_object(dict_obj):
    # Do something custom
    return obj

def deserialise_float(str_obj):
    # Do something custom
    return obj
```

```
load = partial(json.load, object_hook=deserialise_object, parse_float=deserialise_float)
loads = partial(json.loads, object_hook=deserialise_object, parse_float=deserialise_float)
```

Ulteriori personalizzazioni (de-) serializzazione:

Il modulo `json` consente anche l'estensione / sostituzione di `json.JSONEncoder` e `json.JSONDecoder` per gestire vari tipi. Gli hook documentati sopra possono essere aggiunti come predefiniti creando un metodo con un nome equivalente. Per utilizzarli basta passare la classe come parametro `cls` alla funzione rilevante. L'uso di `functools.partial` consente di applicare parzialmente il parametro cls a queste funzioni per comodità, ad es

```
# my_json module

import json
from functools import partial

class MyEncoder(json.JSONEncoder):
    # Do something custom

class MyDecoder(json.JSONDecoder):
    # Do something custom

dump = partial(json.dump, cls=MyEncoder)
dumps = partial(json.dumps, cls=MyEncoder)
load = partial(json.load, cls=MyDecoder)
loads = partial(json.loads, cls=MyDecoder)
```

Examples

Creazione di JSON da Python dict

```
import json
d = {
    'foo': 'bar',
    'alice': 1,
    'wonderland': [1, 2, 3]
}
json.dumps(d)
```

Lo snippet riportato sopra restituirà quanto segue:

```
'{"wonderland": [1, 2, 3], "foo": "bar", "alice": 1}'
```

Creazione di Dict Python da JSON

```
import json
s = '{"wonderland": [1, 2, 3], "foo": "bar", "alice": 1}'
json.loads(s)
```

Lo snippet riportato sopra restituirà quanto segue:

```
{u'alice': 1, u'foo': u'bar', u'wonderland': [1, 2, 3]}
```

Memorizzazione di dati in un file

Il seguente frammento codifica i dati memorizzati in `d` in JSON e li archivia in un file (sostituisce il `filename` con il nome effettivo del file).

```
import json

d = {
    'foo': 'bar',
    'alice': 1,
    'wonderland': [1, 2, 3]
}

with open(filename, 'w') as f:
    json.dump(d, f)
```

Recupero di dati da un file

Lo snippet seguente apre un file con codifica JSON (sostituisce il `filename` con il nome effettivo del file) e restituisce l'oggetto che è archiviato nel file.

```
import json

with open(filename, 'r') as f:
    d = json.load(f)
```

`load` vs` loads`, `dump` vs` dumps`

Il modulo `json` contiene funzioni sia per la lettura e la scrittura da e verso le stringhe Unicode, sia per la lettura e la scrittura da e verso i file. Questi sono differenziati da un trailing `s` nel nome della funzione. In questi esempi utilizziamo un oggetto StringIO, ma le stesse funzioni si applicherebbero a qualsiasi oggetto simile a un file.

Qui usiamo le funzioni basate su stringhe:

```
import json

data = {u"foo": u"bar", u"baz": []}
json_string = json.dumps(data)
# u'{"foo": "bar", "baz": []}'
json.loads(json_string)
# {u"foo": u"bar", u"baz": []}
```

E qui usiamo le funzioni basate su file:

```
import json

from io import StringIO
```

```
json_file = StringIO()
data = {u"foo": u"bar", u"baz": []}
json.dump(data, json_file)
json_file.seek(0)  # Seek back to the start of the file before reading
json_file_content = json_file.read()
# u'{"foo": "bar", "baz": []}'
json_file.seek(0)  # Seek back to the start of the file before reading
json.load(json_file)
# {u"foo": u"bar", u"baz": []}
```

Come puoi vedere, la differenza principale è che quando si scaricano dati json è necessario passare l'handle del file alla funzione, invece di acquisire il valore restituito. Vale anche la pena notare che è necessario cercare all'inizio del file prima di leggere o scrivere, al fine di evitare la corruzione dei dati. Quando si apre un file, il cursore si trova nella posizione `0` , quindi anche il seguente funzionerà:

```
import json

json_file_path = './data.json'
data = {u"foo": u"bar", u"baz": []}

with open(json_file_path, 'w') as json_file:
    json.dump(data, json_file)

with open(json_file_path) as json_file:
    json_file_content = json_file.read()
    # u'{"foo": "bar", "baz": []}'

with open(json_file_path) as json_file:
    json.load(json_file)
    # {u"foo": u"bar", u"baz": []}
```

Avere entrambi i modi di gestire i dati json consente di lavorare in modo idiomatico ed efficiente con i formati basati su json, come ad esempio json-per-line di `pyspark` :

```
# loading from a file
data = [json.loads(line) for line in open(file_path).splitlines()]

# dumping to a file
with open(file_path, 'w') as json_file:
    for item in data:
        json.dump(item, json_file)
        json_file.write('\n')
```

Chiamando `json.tool` dalla riga di comando all'output JSON pretty-print

Dato un file JSON "foo.json" come:

```
{"foo": {"bar": {"baz": 1}}}
```

possiamo chiamare il modulo direttamente dalla riga di comando (passando il nome del file come argomento) per stamparlo in modo carino:

```
$ python -m json.tool foo.json
{
    "foo": {
        "bar": {
            "baz": 1
        }
    }
}
```

Il modulo prenderà anche input da STDOUT, quindi (in Bash) potremmo fare altrettanto:

```
$ cat foo.json | python -m json.tool
```

Formattazione dell'output JSON

Diciamo che abbiamo i seguenti dati:

```
>>> data = {"cats": [{"name": "Tubbs", "color": "white"}, {"name": "Pepper", "color":
"black"}]}
```

Basta scaricare questo come JSON non fa nulla di speciale qui:

```
>>> print(json.dumps(data))
{"cats": [{"name": "Tubbs", "color": "white"}, {"name": "Pepper", "color": "black"}]}
```

Impostazione dell'indentazione per ottenere risultati migliori

Se vogliamo una bella stampa, possiamo impostare una dimensione del `indent` :

```
>>> print(json.dumps(data, indent=2))
{
  "cats": [
    {
      "name": "Tubbs",
      "color": "white"
    },
    {
      "name": "Pepper",
      "color": "black"
    }
  ]
}
```

Ordinamento dei tasti alfabeticamente per ottenere risultati coerenti

Per impostazione predefinita, l'ordine delle chiavi nell'output non è definito. Possiamo ottenerli in ordine alfabetico per assicurarci di ottenere sempre lo stesso risultato:

```
>>> print(json.dumps(data, sort_keys=True))
{"cats": [{"color": "white", "name": "Tubbs"}, {"color": "black", "name": "Pepper"}]}
```

Liberarsi degli spazi bianchi per ottenere risultati compatti

Potremmo desiderare di eliminare gli spazi non necessari, operazione che viene eseguita impostando le stringhe di separazione diverse da `', '` e `': '` predefiniti:

```
>>>print(json.dumps(data, separators=(',', ':')))
{"cats":[{"name":"Tubbs","color":"white"},{"name":"Pepper","color":"black"}]}
```

JSON che codifica oggetti personalizzati

Se solo proviamo il seguente:

```
import json
from datetime import datetime
data = {'datetime': datetime(2016, 9, 26, 4, 44, 0)}
print(json.dumps(data))
```

viene visualizzato un errore che dice `TypeError: datetime.datetime(2016, 9, 26, 4, 44) is not JSON serializable` .

Per poter serializzare correttamente l'oggetto datetime, è necessario scrivere codice personalizzato per come convertirlo:

```
class DatetimeJSONEncoder(json.JSONEncoder):
    def default(self, obj):
        try:
            return obj.isoformat()
        except AttributeError:
            # obj has no isoformat method; let the builtin JSON encoder handle it
            return super(DatetimeJSONEncoder, self).default(obj)
```

e quindi utilizzare questa classe di encoder invece di `json.dumps` :

```
encoder = DatetimeJSONEncoder()
print(encoder.encode(data))
# prints {"datetime": "2016-09-26T04:44:00"}
```

Leggi Modulo JSON online: https://riptutorial.com/it/python/topic/272/modulo-json

Capitolo 122: Modulo matematico

Examples

Arrotondamento: rotondo, pavimento, ceil, trunc

Oltre alla funzione `round` incorporata, il modulo `math` fornisce le funzioni `floor` , `ceil` e `trunc` .

```
x = 1.55
y = -1.55

# round to the nearest integer
round(x)       #  2
round(y)       # -2

# the second argument gives how many decimal places to round to (defaults to 0)
round(x, 1)    #  1.6
round(y, 1)    # -1.6

# math is a module so import it first, then use it.
import math

# get the largest integer less than x
math.floor(x)  #  1
math.floor(y)  # -2

# get the smallest integer greater than x
math.ceil(x)   #  2
math.ceil(y)   # -1

# drop fractional part of x
math.trunc(x)  #  1, equivalent to math.floor for positive numbers
math.trunc(y)  # -1, equivalent to math.ceil for negative numbers
```

Python 2.x 2.7

`floor` , `ceil` , `trunc` e `round` restituiscono sempre un `float` .

```
round(1.3)  # 1.0
```

`round` rompe sempre i legami dallo zero.

```
round(0.5)  # 1.0
round(1.5)  # 2.0
```

Python 3.x 3.0

`floor` , `ceil` e `trunc` restituiscono sempre un valore `Integral` , mentre `round` restituisce un valore `Integral` se chiamato con un argomento.

```
round(1.3)      # 1
round(1.33, 1)  # 1.3
```

`round` rompe i legami verso il numero pari più vicino. Questo corregge la tendenza verso numeri più grandi quando si esegue un numero elevato di calcoli.

```
round(0.5)  # 0
round(1.5)  # 2
```

Avvertimento!

Come con qualsiasi rappresentazione a virgola mobile, alcune frazioni *non possono essere rappresentate esattamente* . Ciò può comportare un comportamento di arrotondamento imprevisto.

```
round(2.675, 2)  # 2.67, not 2.68!
```

Avviso sul piano, trunc e divisione intera dei numeri negativi

Python (e C ++ e Java) arrotondati da zero per i numeri negativi. Tenere conto:

```
>>> math.floor(-1.7)
-2.0
>>> -5 // 2
-3
```

logaritmi

`math.log(x)` fornisce il logaritmo naturale (base `e`) di `x` .

```
math.log(math.e)  # 1.0
math.log(1)       # 0.0
math.log(100)     # 4.605170185988092
```

`math.log` può perdere precisione con numeri vicini a 1, a causa delle limitazioni dei numeri in virgola mobile. Per calcolare con precisione i registri vicini a 1, utilizzare `math.log1p` , che valuta il logaritmo naturale di 1 più l'argomento:

```
math.log(1 + 1e-20)  # 0.0
math.log1p(1e-20)    # 1e-20
```

`math.log10` può essere utilizzato per i registri di base 10:

```
math.log10(10)  # 1.0
```

Python 2.x 2.3.0

Quando viene utilizzato con due argomenti, `math.log(x, base)` fornisce il logaritmo di `x` nella `base` fornita (ad esempio, `log(x) / log(base)` .

```
math.log(100, 10) # 2.0
```

```
math.log(27, 3)   # 3.0
math.log(1, 10)   # 0.0
```

Copia di segni

In Python 2.6 e versioni successive, `math.copysign(x, y)` restituisce `x` con il segno di `y` . Il valore restituito è sempre un `float` .

Python 2.x 2.6

```
math.copysign(-2, 3)    # 2.0
math.copysign(3, -3)    # -3.0
math.copysign(4, 14.2)  # 4.0
math.copysign(1, -0.0)  # -1.0, on a platform which supports signed zero
```

Trigonometria

Calcolo della lunghezza dell'ipotenusa

```
math.hypot(2, 4) # Just a shorthand for SquareRoot(2**2 + 4**2)
# Out: 4.47213595499958
```

Conversione dei gradi in / da radianti

Tutte le funzioni `math` prevedono i **radianti** quindi è necessario convertire i gradi in radianti:

```
math.radians(45)              # Convert 45 degrees to radians
# Out: 0.7853981633974483
```

Tutti i risultati delle funzioni trigonometriche inverse restituiscono il risultato in radianti, quindi potrebbe essere necessario convertirlo in gradi:

```
math.degrees(math.asin(1))    # Convert the result of asin to degrees
# Out: 90.0
```

Funzioni seno, coseno, tangente e inverso

```
# Sine and arc sine
math.sin(math.pi / 2)
# Out: 1.0
math.sin(math.radians(90))   # Sine of 90 degrees
# Out: 1.0

math.asin(1)
# Out: 1.5707963267948966    # "= pi / 2"
math.asin(1) / math.pi
# Out: 0.5

# Cosine and arc cosine:
math.cos(math.pi / 2)
```

```
# Out: 6.123233995736766e-17
# Almost zero but not exactly because "pi" is a float with limited precision!

math.acos(1)
# Out: 0.0

# Tangent and arc tangent:
math.tan(math.pi/2)
# Out: 1.633123935319537e+16
# Very large but not exactly "Inf" because "pi" is a float with limited precision
```

Python 3.x 3.5

```
math.atan(math.inf)
# Out: 1.5707963267948966 # This is just "pi / 2"
```

```
math.atan(float('inf'))
# Out: 1.5707963267948966 # This is just "pi / 2"
```

Oltre a `math.atan` esiste anche una funzione `math.atan2` due argomenti, che calcola il quadrante corretto ed evita le insidie della divisione per zero:

```
math.atan2(1, 2)   # Equivalent to "math.atan(1/2)"
# Out: 0.4636476090008061 # ≈ 26.57 degrees, 1st quadrant

math.atan2(-1, -2) # Not equal to "math.atan(-1/-2)" == "math.atan(1/2)"
# Out: -2.677945044588987 # ≈ -153.43 degrees (or 206.57 degrees), 3rd quadrant

math.atan2(1, 0)   # math.atan(1/0) would raise ZeroDivisionError
# Out: 1.5707963267948966 # This is just "pi / 2"
```

Seno iperbolico, coseno e tangente

```
# Hyperbolic sine function
math.sinh(math.pi) # = 11.548739357257746
math.asinh(1)      # = 0.8813735870195429

# Hyperbolic cosine function
math.cosh(math.pi) # = 11.591953275521519
math.acosh(1)      # = 0.0

# Hyperbolic tangent function
math.tanh(math.pi) # = 0.99627207622075
math.atanh(0.5)    # = 0.5493061443340549
```

costanti

`math` moduli `math` includono due costanti matematiche comunemente usate.

- `math.pi` - La costante matematica pi
- `math.e` - La costante matematica *e* (base del logaritmo naturale)

```
>>> from math import pi, e
```

```
>>> pi
3.141592653589793
>>> e
2.718281828459045
>>>
```

Python 3.5 e successive hanno costanti per infinito e NaN ("non un numero"). La più vecchia sintassi del passaggio di una stringa a `float()` funziona ancora.

Python 3.x 3.5

```
math.inf == float('inf')
# Out: True

-math.inf == float('-inf')
# Out: True

# NaN never compares equal to anything, even itself
math.nan == float('nan')
# Out: False
```

Numeri immaginari

I numeri immaginari in Python sono rappresentati da una "j" o "J" che trascina il numero di destinazione.

```
1j         # Equivalent to the square root of -1.
1j * 1j    # = (-1+0j)
```

Infinity e NaN ("non un numero")

In tutte le versioni di Python, possiamo rappresentare infinito e NaN ("non un numero") come segue:

```
pos_inf = float('inf')     # positive infinity
neg_inf = float('-inf')    # negative infinity
not_a_num = float('nan')   # NaN ("not a number")
```

In Python 3.5 e versioni successive, possiamo anche usare le costanti definite `math.inf` e `math.nan` :

Python 3.x 3.5

```
pos_inf = math.inf
neg_inf = -math.inf
not_a_num = math.nan
```

Le rappresentazioni di stringa vengono visualizzate come `inf` e `-inf` e `nan` :

```
pos_inf, neg_inf, not_a_num
# Out: (inf, -inf, nan)
```

Possiamo testare l'infinito positivo o negativo con il metodo `isinf` :

```
math.isinf(pos_inf)
# Out: True

math.isinf(neg_inf)
# Out: True
```

Possiamo testare specificamente per infinito positivo o infinito negativo per confronto diretto:

```
pos_inf == float('inf')    # or  == math.inf in Python 3.5+
# Out: True

neg_inf == float('-inf')   # or  == -math.inf in Python 3.5+
# Out: True

neg_inf == pos_inf
# Out: False
```

Python 3.2 e versioni successive consentono anche di controllare la finezza:

Python 3.x 3.2

```
math.isfinite(pos_inf)
# Out: False

math.isfinite(0.0)
# Out: True
```

Gli operatori di confronto funzionano come previsto per l'infinito positivo e negativo:

```
import sys

sys.float_info.max
# Out: 1.7976931348623157e+308  (this is system-dependent)

pos_inf > sys.float_info.max
# Out: True

neg_inf < -sys.float_info.max
# Out: True
```

Ma se un'espressione aritmetica produce un valore maggiore del massimo che può essere rappresentato come un `float` , diventerà infinito:

```
pos_inf == sys.float_info.max * 1.0000001
# Out: True

neg_inf == -sys.float_info.max * 1.0000001
# Out: True
```

Comunque la divisione per zero non dà un risultato di infinito (o infinito negativo se appropriato), piuttosto solleva un'eccezione `ZeroDivisionError` .

```
try:
    x = 1.0 / 0.0
    print(x)
except ZeroDivisionError:
    print("Division by zero")

# Out: Division by zero
```

Le operazioni aritmetiche sull'infinito danno solo risultati infiniti o talvolta NaN:

```
-5.0 * pos_inf == neg_inf
# Out: True

-5.0 * neg_inf == pos_inf
# Out: True

pos_inf * neg_inf == neg_inf
# Out: True

0.0 * pos_inf
# Out: nan

0.0 * neg_inf
# Out: nan

pos_inf / pos_inf
# Out: nan
```

NaN non è mai uguale a niente, nemmeno a se stesso. Possiamo testare perché è con il metodo `isnan` :

```
not_a_num == not_a_num
# Out: False

math.isnan(not_a_num)
Out: True
```

NaN si confronta sempre come "non uguale", ma mai inferiore o superiore a:

```
not_a_num != 5.0   # or any random value
# Out: True

not_a_num > 5.0   or   not_a_num < 5.0   or   not_a_num == 5.0
# Out: False
```

Le operazioni aritmetiche su NaN danno sempre NaN. Ciò include la moltiplicazione per -1: non c'è "NaN negativo".

```
5.0 * not_a_num
# Out: nan

float('-nan')
# Out: nan
```

Python 3.x 3.5

```
-math.nan
# Out: nan
```

C'è una sottile differenza tra le vecchie versioni `float` di NaN e infinity e le costanti della libreria `math` Python 3.5+:

Python 3.x 3.5

```
math.inf is math.inf, math.nan is math.nan
# Out: (True, True)

float('inf') is float('inf'), float('nan') is float('nan')
# Out: (False, False)
```

Pow per un'esponenziazione più rapida

Utilizzando il modulo timeit dalla riga di comando:

```
> python -m timeit 'for x in xrange(50000): b = x**3'
10 loops, best of 3: 51.2 msec per loop
> python -m timeit 'from math import pow' 'for x in xrange(50000): b = pow(x,3)'
100 loops, best of 3: 9.15 msec per loop
```

L'operatore integrato `**` spesso è utile, ma se le prestazioni sono essenziali, usa math.pow. Assicurati di notare, tuttavia, che pow restituisce float, anche se gli argomenti sono interi:

```
> from math import pow
> pow(5,5)
3125.0
```

Numeri complessi e il modulo cmath

Il modulo `cmath` è simile al modulo `math` , ma definisce le funzioni in modo appropriato per il piano complesso.

Prima di tutto, i numeri complessi sono un tipo numerico che fa parte del linguaggio Python stesso anziché essere fornito da una classe di libreria. Pertanto non è necessario `import cmath` per le normali espressioni aritmetiche.

Nota che usiamo `j` (o `J`) e non `i` .

```
z = 1 + 3j
```

Dobbiamo usare `1j` poiché `j` sarebbe il nome di una variabile piuttosto che un valore letterale numerico.

```
1j * 1j
Out: (-1+0j)

1j ** 1j
```

```
# Out: (0.20787957635076193+0j)     # "i to the i"  ==  math.e ** -(math.pi/2)
```

Abbiamo la parte `real` e la parte `imag` (immaginaria), così come il complesso `conjugate` :

```
# real part and imaginary part are both float type
z.real, z.imag
# Out: (1.0, 3.0)

z.conjugate()
# Out: (1-3j)    # z.conjugate() == z.real - z.imag * 1j
```

Le funzioni integrate `abs` e `complex` sono anche parte del linguaggio stesso e non richiedono alcuna importazione:

```
abs(1 + 1j)
# Out: 1.4142135623730951     # square root of 2

complex(1)
# Out: (1+0j)

complex(imag=1)
# Out: (1j)

complex(1, 1)
# Out: (1+1j)
```

La funzione `complex` può prendere una stringa, ma non può avere spazi:

```
complex('1+1j')
# Out: (1+1j)

complex('1 + 1j')
# Exception: ValueError: complex() arg is a malformed string
```

Ma per la maggior parte delle funzioni abbiamo bisogno del modulo, ad esempio `sqrt` :

```
import cmath

cmath.sqrt(-1)
# Out: 1j
```

Naturalmente il comportamento di `sqrt` è diverso per numeri complessi e numeri reali. Nella `math` non complessa la radice quadrata di un numero negativo solleva un'eccezione:

```
import math

math.sqrt(-1)
# Exception: ValueError: math domain error
```

Le funzioni sono fornite per convertire da e verso le coordinate polari:

```
cmath.polar(1 + 1j)
# Out: (1.4142135623730951, 0.7853981633974483)    # == (sqrt(1 + 1), atan2(1, 1))
```

```
abs(1 + 1j), cmath.phase(1 + 1j)
# Out: (1.4142135623730951, 0.7853981633974483)    # same as previous calculation

cmath.rect(math.sqrt(2), math.atan(1))
# Out: (1.0000000000000002+1.0000000000000002j)
```

Il campo matematico dell'analisi complessa va oltre lo scopo di questo esempio, ma molte funzioni nel piano complesso hanno un "taglio di ramo", solitamente lungo l'asse reale o l'asse immaginario. La maggior parte delle piattaforme moderne supporta lo "zero firmato" come specificato in IEEE 754, che fornisce la continuità di tali funzioni su entrambi i lati del taglio del ramo. Il seguente esempio è tratto dalla documentazione di Python:

```
cmath.phase(complex(-1.0, 0.0))
# Out: 3.141592653589793

cmath.phase(complex(-1.0, -0.0))
# Out: -3.141592653589793
```

Il modulo `cmath` fornisce anche molte funzioni con controparti dirette dal modulo `math` .

Oltre a `sqrt` , esistono versioni complesse di `exp` , `log` , `log10` , le funzioni trigonometriche e le loro inverse (`sin` , `cos` , `tan` , `asin` , `acos` , `atan`) e le funzioni iperboliche e le loro inverse (`sinh` , `cosh` , `tanh` , `asinh` , `acosh` , `atanh`). Si noti tuttavia che non esiste una controparte complessa di `math.atan2` , la forma a due argomenti di arcotangente.

```
cmath.log(1+1j)
# Out: (0.3465735902799726+0.7853981633974483j)

cmath.exp(1j * cmath.pi)
# Out: (-1+1.2246467991473532e-16j)   # e to the i pi == -1, within rounding error
```

Le costanti `pi` ed `e` sono fornite. Nota che questi sono `float` e non `complex` .

```
type(cmath.pi)
# Out: <class 'float'>
```

Il modulo `cmath` fornisce anche versioni complesse di `isinf` , e (per Python 3.2+) `isfinite` . Vedi " Infinito e NaN ". Un numero complesso è considerato infinito se la sua parte reale o la sua parte immaginaria è infinita.

```
cmath.isinf(complex(float('inf'), 0.0))
# Out: True
```

Allo stesso modo, il modulo `cmath` fornisce una versione complessa di `isnan` . Vedi " Infinito e NaN ". Un numero complesso è considerato "non un numero" se la sua parte reale o la sua parte immaginaria è "non un numero".

```
cmath.isnan(0.0, float('nan'))
# Out: True
```

Nota: non esiste una controparte di `cmath` delle costanti `math.inf` e `math.nan` (da Python 3.5 e versioni successive)

Python 3.x 3.5

```
cmath.isinf(complex(0.0, math.inf))
# Out: True

cmath.isnan(complex(math.nan, 0.0))
# Out: True

cmath.inf
# Exception: AttributeError: module 'cmath' has no attribute 'inf'
```

In Python 3.5 e superiore, v'è un `isclose` metodo sia `cmath` e `math` moduli.

Python 3.x 3.5

```
z = cmath.rect(*cmath.polar(1+1j))

z
# Out: (1.0000000000000002+1.0000000000000002j)

cmath.isclose(z, 1+1j)
# True
```

Leggi Modulo matematico online: https://riptutorial.com/it/python/topic/230/modulo-matematico

Capitolo 123: Modulo operatore

Examples

Operatori in alternativa a un operatore infisso

Per ogni operatore infisso, ad es. `+` C'è una funzione- `operator` (`operator.add` per `+`):

```
1 + 1
# Output: 2
from operator import add
add(1, 1)
# Output: 2
```

anche se la documentazione principale afferma che per gli operatori aritmetici è consentito solo l'input numerico è possibile:

```
from operator import mul
mul('a', 10)
# Output: 'aaaaaaaaaa'
mul([3], 3)
# Output: [3, 3, 3]
```

Vedi anche: mappatura da operazione a funzione operatore nella documentazione ufficiale di Python .

Methodcaller

Invece di questa funzione `lambda` che chiama esplicitamente il metodo:

```
alist = ['wolf', 'sheep', 'duck']
list(filter(lambda x: x.startswith('d'), alist))     # Keep only elements that start with 'd'
# Output: ['duck']
```

si potrebbe usare una funzione dell'operatore che faccia lo stesso:

```
from operator import methodcaller
list(filter(methodcaller('startswith', 'd'), alist)) # Does the same but is faster.
# Output: ['duck']
```

Itemgetter

Raggruppamento delle coppie chiave-valore di un dizionario in base al valore con `itemgetter` :

```
from itertools import groupby
from operator import itemgetter
adict = {'a': 1, 'b': 5, 'c': 1}
```

```
dict((i, dict(v)) for i, v in groupby(adict.items(), itemgetter(1)))
# Output: {1: {'a': 1, 'c': 1}, 5: {'b': 5}}
```

che è equivalente (ma più veloce) a una funzione `lambda` come questa:

```
dict((i, dict(v)) for i, v in groupby(adict.items(), lambda x: x[1]))
```

O ordinando una lista di tuple dal secondo elemento prima il primo elemento come secondario:

```
alist_of_tuples = [(5,2), (1,3), (2,2)]
sorted(alist_of_tuples, key=itemgetter(1,0))
# Output: [(2, 2), (5, 2), (1, 3)]
```

Leggi Modulo operatore online: https://riptutorial.com/it/python/topic/257/modulo-operatore

Capitolo 124: modulo pyautogui

introduzione

pyautogui è un modulo usato per controllare mouse e tastiera. Questo modulo è fondamentalmente utilizzato per automatizzare le attività di clic del mouse e della tastiera. Per il mouse, le coordinate dello schermo (0,0) iniziano dall'angolo in alto a sinistra. Se sei fuori controllo, sposta rapidamente il cursore del mouse in alto a sinistra, prenderà il controllo del mouse e della tastiera da Python e te lo restituirà.

Examples

Funzioni del mouse

Queste sono alcune delle utili funzioni del mouse per controllare il mouse.

```
size()           #gave you the size of the screen
position()     #return current position of mouse
moveTo(200,0,duration=1.5)     #move the cursor  to (200,0) position  with 1.5 second delay

moveRel()          #move the cursor relative to your current position.
click(337,46)           #it will click on the position mention there
dragRel()              #it will drag the mouse relative to position
pyautogui.displayMousePosition()     #gave you the current mouse position but should be done
on terminal.
```

Funzioni della tastiera

Queste sono alcune delle utili funzioni della tastiera per automatizzare la pressione dei tasti.

```
typewrite('')    #this will type the string on the screen where current window has focused.
typewrite(['a','b','left','left','X','Y'])
pyautogui.KEYBOARD_KEYS    #get the list of all the keyboard_keys.
pyautogui.hotkey('ctrl','o')    #for the combination of keys to enter.
```

ScreenShot e riconoscimento dell'immagine

Queste funzioni ti aiuteranno a prendere lo screenshot e ad abbinare anche l'immagine con la parte dello schermo.

```
.screenshot('c:\\path')        #get the screenshot.
.locateOnScreen('c:\\path')    #search that image on screen and get the coordinates for you.
locateCenterOnScreen('c:\\path')       #get the coordinate for the image on screen.
```

Leggi modulo pyautogui online: https://riptutorial.com/it/python/topic/9432/modulo-pyautogui

Capitolo 125: Modulo Sqlite3

Examples

Sqlite3: non richiede un processo server separato.

Il modulo sqlite3 è stato scritto da Gerhard Häring. Per utilizzare il modulo, è necessario innanzitutto creare un oggetto Connection che rappresenti il database. Qui i dati saranno memorizzati nel file example.db:

```
import sqlite3
conn = sqlite3.connect('example.db')
```

È inoltre possibile fornire il nome speciale: memoria: per creare un database nella RAM. Una volta che hai una connessione, puoi creare un oggetto cursore e chiamare il suo metodo execute () per eseguire comandi SQL:

```
c = conn.cursor()

# Create table
c.execute('''CREATE TABLE stocks
         (date text, trans text, symbol text, qty real, price real)''')

# Insert a row of data
c.execute("INSERT INTO stocks VALUES ('2006-01-05','BUY','RHAT',100,35.14)")

# Save (commit) the changes
conn.commit()

# We can also close the connection if we are done with it.
# Just be sure any changes have been committed or they will be lost.
conn.close()
```

Ottenere i valori dal database e la gestione degli errori

Recupero dei valori dal database SQLite3.

Stampa i valori delle righe restituite dalla query selezionata

```
import sqlite3
conn = sqlite3.connect('example.db')
c = conn.cursor()
c.execute("SELECT * from table_name where id=cust_id")
for row in c:
    print row # will be a list
```

Per recuperare un singolo metodo fetchone ()

```
print c.fetchone()
```

Per più righe utilizzare il metodo fetchall ()

```
a=c.fetchall() #which is similar to list(cursor) method used previously
for row in a:
    print row
```

La gestione degli errori può essere eseguita utilizzando sqlite3.Error built in function

```
try:
    #SQL Code
except sqlite3.Error as e:
    print "An error occurred:", e.args[0]
```

Leggi Modulo Sqlite3 online: https://riptutorial.com/it/python/topic/7754/modulo-sqlite3

Capitolo 126: Modulo Webbrowser

introduzione

Secondo la documentazione standard di Python, il modulo browser fornisce un'interfaccia di alto livello per consentire la visualizzazione di documenti basati sul Web agli utenti. Questo argomento spiega e dimostra l'uso corretto del modulo browser web.

Sintassi

- `webbrowser.open(url, new=0, autoraise=False)`
- `webbrowser.open_new(url)`
- `webbrowser.open_new_tab(url)`
- `webbrowser.get(usage=None)`
- `webbrowser.register(name, constructor, instance=None)`

Parametri

Parametro	Dettagli
`webbrowser.open()`	
url	l'URL da aprire nel browser web
nuovo	0 apre l'URL nella scheda esistente, 1 si apre in una nuova finestra, 2 si apre in una nuova scheda
autoraise	se impostato su True, la finestra verrà spostata in cima alle altre finestre
`webbrowser.open_new()`	
url	l'URL da aprire nel browser web
`webbrowser.open_new_tab()`	
url	l'URL da aprire nel browser web
`webbrowser.get()`	
utilizzando	il browser da usare
`webbrowser.register()`	
url	nome del browser
costruttore	percorso per il browser eseguibile (aiuto)
esempio	Un'istanza di un browser Web restituita dal metodo

Parametro	Dettagli
	`webbrowser.get()`

Osservazioni

La seguente tabella elenca i tipi di browser predefiniti. La colonna di sinistra sono nomi che possono essere passati al metodo `webbrowser.get()` e la colonna di destra elenca i nomi di classe per ciascun tipo di browser.

Digita il nome	Nome della classe
`'mozilla'`	`Mozilla('mozilla')`
`'firefox'`	`Mozilla('mozilla')`
`'netscape'`	`Mozilla('netscape')`
`'galeon'`	`Galeon('galeon')`
`'epiphany'`	`Galeon('epiphany')`
`'skipstone'`	`BackgroundBrowser('skipstone')`
`'kfmclient'`	`Konqueror()`
`'konqueror'`	`Konqueror()`
`'kfm'`	`Konqueror()`
`'mosaic'`	`BackgroundBrowser('mosaic')`
`'opera'`	`Opera()`
`'grail'`	`Grail()`
`'links'`	`GenericBrowser('links')`
`'elinks'`	`Elinks('elinks')`
`'lynx'`	`GenericBrowser('lynx')`
`'w3m'`	`GenericBrowser('w3m')`
`'windows-default'`	`WindowsDefault`
`'macosx'`	`MacOSX('default')`
`'safari'`	`MacOSX('safari')`
`'google-chrome'`	`Chrome('google-chrome')`
`'chrome'`	`Chrome('chrome')`
`'chromium'`	`Chromium('chromium')`
`'chromium-browser'`	`Chromium('chromium-browser')`

Examples

Apertura di un URL con Browser predefinito

Per aprire semplicemente un URL, usa il metodo `webbrowser.open()` :

```
import webbrowser
webbrowser.open("http://stackoverflow.com")
```

Se una finestra del browser è attualmente aperta, il metodo aprirà una nuova scheda nell'URL specificato. Se nessuna finestra è aperta, il metodo aprirà il browser predefinito del sistema operativo e passerà all'URL nel parametro. Il metodo aperto supporta i seguenti parametri:

- `url` : l'URL da aprire nel browser Web (stringa) **[richiesto]**
- `new` - 0 si apre nella scheda esistente, 1 apre una nuova finestra, 2 apre una nuova scheda (numero intero) **[default 0]**
- `autoraise` - se impostato su True, la finestra verrà spostata sopra le finestre di altre applicazioni (Boolean) **[default False]**

Nota: gli argomenti `new` e `autoraise` raramente funzionano poiché la maggior parte dei browser moderni rifiuta questi comandi.

Webbrowser può anche provare ad aprire gli URL in nuove finestre con il metodo `open_new` :

```
import webbrowser
webbrowser.open_new("http://stackoverflow.com")
```

Questo metodo è comunemente ignorato dai browser moderni e l'URL viene solitamente aperto in una nuova scheda. L'apertura di una nuova scheda può essere provata dal modulo usando il metodo `open_new_tab` :

```
import webbrowser
webbrowser.open_new_tab("http://stackoverflow.com")
```

Aprire un URL con diversi browser

Il modulo browser Web supporta anche diversi browser che utilizzano i metodi `register()` e `get()` . Il metodo get viene utilizzato per creare un controller browser utilizzando il percorso di un eseguibile specifico e il metodo di registrazione viene utilizzato per collegare questi eseguibili ai tipi di browser preimpostati per l'utilizzo futuro, in genere quando vengono utilizzati più tipi di browser.

```
import webbrowser
ff_path = webbrowser.get("C:/Program Files/Mozilla Firefox/firefox.exe")
ff = webbrowser.get(ff_path)
ff.open("http://stackoverflow.com/")
```

Registrazione di un tipo di browser:

```
import webbrowser
ff_path = webbrowser.get("C:/Program Files/Mozilla Firefox/firefox.exe")
ff = webbrowser.get(ff_path)
webbrowser.register('firefox', None, ff)
# Now to refer to use Firefox in the future you can use this
webbrowser.get('firefox').open("https://stackoverflow.com/")
```

Leggi Modulo Webbrowser online: https://riptutorial.com/it/python/topic/8676/modulo-webbrowser

Capitolo 127: multiprocessing

Examples

Esecuzione di due processi semplici

Un semplice esempio di utilizzo di più processi sarebbe costituito da due processi (operatori) eseguiti separatamente. Nell'esempio seguente, vengono avviati due processi:

- `countUp()` conta 1 su, ogni secondo.
- `countDown()` conta 1 down, ogni secondo.

```
import multiprocessing
import time
from random import randint

def countUp():
    i = 0
    while i <= 3:
        print('Up:\t{}'.format(i))
        time.sleep(randint(1, 3)) # sleep 1, 2 or 3 seconds
        i += 1

def countDown():
    i = 3
    while i >= 0:
        print('Down:\t{}'.format(i))
        time.sleep(randint(1, 3)) # sleep 1, 2 or 3 seconds
        i -= 1

if __name__ == '__main__':
    # Initiate the workers.
    workerUp = multiprocessing.Process(target=countUp)
    workerDown = multiprocessing.Process(target=countDown)

    # Start the workers.
    workerUp.start()
    workerDown.start()

    # Join the workers. This will block in the main (parent) process
    # until the workers are complete.
    workerUp.join()
    workerDown.join()
```

L'output è il seguente:

```
Up:    0
Down:    3
Up:    1
Up:    2
Down:    2
Up:    3
Down:    1
Down:    0
```

Utilizzando Pool e Mappa

```
from multiprocessing import Pool

def cube(x):
    return x ** 3

if __name__ == "__main__":
    pool = Pool(5)
    result = pool.map(cube, [0, 1, 2, 3])
```

`Pool` è una classe che gestisce più `Workers` (processi) dietro le quinte e consente a te, il programmatore, di usarli.

`Pool(5)` crea un nuovo pool con 5 processi e `pool.map` funziona come la mappa ma utilizza più processi (la quantità definita durante la creazione del pool).

Risultati simili possono essere ottenuti usando `map_async` , `apply` e `apply_async` che possono essere trovati nella documentazione .

Leggi multiprocessing online: https://riptutorial.com/it/python/topic/3601/multiprocessing

Capitolo 128: multithreading

introduzione

I thread consentono ai programmi Python di gestire più funzioni contemporaneamente anziché eseguire una sequenza di comandi individualmente. Questo argomento spiega i principi alla base del threading e ne dimostra l'utilizzo.

Examples

Nozioni di base sul multithreading

Utilizzando il modulo di `threading` , è possibile avviare un nuovo thread di esecuzione creando un nuovo `threading.Thread` e assegnargli una funzione da eseguire:

```
import threading

def foo():
  print "Hello threading!"

my_thread = threading.Thread(target=foo)
```

Il parametro `target` riferimento alla funzione (o oggetto callable) da eseguire. Il thread non inizierà l'esecuzione finché non viene chiamato `start` sull'oggetto `Thread` .

Iniziare una discussione

```
my_thread.start() # prints 'Hello threading!'
```

Ora che `my_thread` è stato eseguito e terminato, richiamando `start` verrà `RuntimeError` un `RuntimeError` . Se si desidera eseguire il thread come daemon, passando il `daemon=True` kwarg o impostando `my_thread.daemon` su `True` prima di chiamare `start()` , il `Thread` avvia silenziosamente in background come daemon.

Partecipare a una discussione

Nei casi in cui dividi un grosso lavoro in più di uno piccolo e desideri eseguirli contemporaneamente, ma `Thread.join()` aspettare che finiscano tutti prima di continuare, `Thread.join()` è il metodo che stai cercando.

Ad esempio, supponiamo di voler scaricare diverse pagine di un sito Web e di compilarle in un'unica pagina. Faresti questo:

```
import requests
from threading import Thread
from queue import Queue
```

```
q = Queue(maxsize=20)
def put_page_to_q(page_num):
    q.put(requests.get('http://some-website.com/page_%s.html' % page_num)

def compile(q):
    # magic function that needs all pages before being able to be executed
    if not q.full():
        raise ValueError
    else:
        print("Done compiling!")

threads = []
for page_num in range(20):
     t = Thread(target=requests.get, args=(page_num,))
     t.start()
     threads.append(t)

# Next, join all threads to make sure all threads are done running before
# we continue. join() is a blocking call (unless specified otherwise using
# the kwarg blocking=False when calling join)
for t in threads:
    t.join()

# Call compile() now, since all threads have completed
compile(q)
```

Uno sguardo più ravvicinato a come funziona `join()` può essere trovato qui .

Creare una classe thread personalizzata

Usando la classe `threading.Thread` possiamo sottoclasse la nuova classe Thread personalizzata. dobbiamo sovrascrivere il metodo `run` in una sottoclasse.

```
from threading import Thread
import time

class Sleepy(Thread):

    def run(self):
        time.sleep(5)
        print("Hello form Thread")

if __name__ == "__main__":
    t = Sleepy()
    t.start()      # start method automatic call Thread class run method.
    # print 'The main program continues to run in foreground.'
    t.join()
    print("The main program continues to run in the foreground.")
```

Comunicare tra i thread

Ci sono più thread nel codice e devi comunicare in sicurezza tra di loro.

È possibile utilizzare una `Queue` dalla libreria della `queue` .

```
from queue import Queue
```

```
from threading import Thread

# create a data producer
def producer(output_queue):
    while True:
        data = data_computation()

        output_queue.put(data)

# create a consumer
def consumer(input_queue):
    while True:
        # retrieve data (blocking)
        data = input_queue.get()

        # do something with the data

        # indicate data has been consumed
        input_queue.task_done()
```

Creazione di thread di produzione e di consumo con una coda condivisa

```
q = Queue()
t1 = Thread(target=consumer, args=(q,))
t2 = Thread(target=producer, args=(q,))
t1.start()
t2.start()
```

Creazione di un pool di worker

Utilizzo di `threading` e `queue` :

```
from socket import socket, AF_INET, SOCK_STREAM
from threading import Thread
from queue import Queue

def echo_server(addr, nworkers):
    print('Echo server running at', addr)
    # Launch the client workers
    q = Queue()
    for n in range(nworkers):
        t = Thread(target=echo_client, args=(q,))
        t.daemon = True
        t.start()

    # Run the server
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(addr)
    sock.listen(5)
    while True:
        client_sock, client_addr = sock.accept()
        q.put((client_sock, client_addr))

echo_server(('',15000), 128)
```

Utilizzando `concurrent.futures.Threadpoolexecutor` :

```
from socket import AF_INET, SOCK_STREAM, socket
from concurrent.futures import ThreadPoolExecutor

def echo_server(addr):
    print('Echo server running at', addr)
    pool = ThreadPoolExecutor(128)
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(addr)
    sock.listen(5)
    while True:
        client_sock, client_addr = sock.accept()
        pool.submit(echo_client, client_sock, client_addr)

echo_server(('',15000))
```

Python Cookbook, 3rd edition, di David Beazley e Brian K. Jones (O'Reilly). Copyright 2013 David Beazley e Brian Jones, 978-1-449-34037-7.

Uso avanzato di multithread

Questa sezione conterrà alcuni degli esempi più avanzati realizzati utilizzando il multithreading.

Stampante avanzata (logger)

Un thread che stampa tutto viene ricevuto e modifica l'output in base alla larghezza del terminale. La parte interessante è che anche l'output "già scritto" viene modificato quando cambia la larghezza del terminale.

```
#!/usr/bin/env python2

import threading
import Queue
import time
import sys
import subprocess
from backports.shutil_get_terminal_size import get_terminal_size

printq = Queue.Queue()
interrupt = False
lines = []

def main():

    ptt = threading.Thread(target=printer) # Turn the printer on
    ptt.daemon = True
    ptt.start()

    # Stupid example of stuff to print
    for i in xrange(1,100):
        printq.put(' '.join([str(x) for x in range(1,i)]))           # The actual way to send
stuff to the printer
        time.sleep(.5)

def split_line(line, cols):
    if len(line) > cols:
        new_line = ''
```

```
        ww = line.split()
        i = 0
        while len(new_line) <= (cols - len(ww[i]) - 1):
            new_line += ww[i] + ' '
            i += 1
            print len(new_line)
        if new_line == '':
            return (line, '')

        return (new_line, ' '.join(ww[i:]))
    else:
        return (line, '')


def printer():

    while True:
        cols, rows = get_terminal_size() # Get the terminal dimensions
        msg = '#' + '-' * (cols - 2) + '#\n' # Create the
        try:
            new_line = str(printq.get_nowait())
            if new_line != '!@#EXIT#@!': # A nice way to turn the printer
                                         # thread out gracefully
                lines.append(new_line)
                printq.task_done()
            else:
                printq.task_done()
                sys.exit()
        except Queue.Empty:
            pass

        # Build the new message to show and split too long lines
        for line in lines:
            res = line          # The following is to split lines which are
                                # longer than cols.
            while len(res) !=0:
                toprint, res = split_line(res, cols)
                msg += '\n' + toprint

        # Clear the shell and print the new output
        subprocess.check_call('clear') # Keep the shell clean
        sys.stdout.write(msg)
        sys.stdout.flush()
        time.sleep(.5)
```

Filo bloccabile con un ciclo while

```
import threading
import time

class StoppableThread(threading.Thread):
    """Thread class with a stop() method. The thread itself has to check
    regularly for the stopped() condition."""

    def __init__(self):
        super(StoppableThread, self).__init__()
        self._stop_event = threading.Event()

    def stop(self):
```

```
        self._stop_event.set()

    def join(self, *args, **kwargs):
        self.stop()
        super(StoppableThread,self).join(*args, **kwargs)

    def run()
        while not self._stop_event.is_set():
            print("Still running!")
            time.sleep(2)
        print("stopped!"
```

Basato su questa domanda .

Leggi multithreading online: https://riptutorial.com/it/python/topic/544/multithreading

Capitolo 129: Mutevole vs Immutabile (e Lavabile) in Python

Examples

Mutevole vs Immutabile

Ci sono due tipi di tipi in Python. Tipi immutabili e tipi mutevoli.

immutabili

Un oggetto di un tipo immutabile non può essere cambiato. Qualsiasi tentativo di modificare l'oggetto comporterà la creazione di una copia.

Questa categoria include: interi, float, complessi, stringhe, byte, tuple, intervalli e frozenset.

Per evidenziare questa proprietà, giochiamo con l' `id` integrato. Questa funzione restituisce l'identificativo univoco dell'oggetto passato come parametro. Se l'id è lo stesso, questo è lo stesso oggetto. Se cambia, allora questo è un altro oggetto. *(Alcuni dicono che questo è in realtà l'indirizzo di memoria dell'oggetto, ma attenzione, sono dal lato oscuro della forza ...)*

```
>>> a = 1
>>> id(a)
140128142243264
>>> a += 2
>>> a
3
>>> id(a)
140128142243328
```

Ok, 1 non è 3 ... Ultime notizie ... Forse no. Tuttavia, questo comportamento viene spesso dimenticato quando si tratta di tipi più complessi, in particolare stringhe.

```
>>> stack = "Overflow"
>>> stack
'Overflow'
>>> id(stack)
140128123955504
>>> stack += " rocks!"
>>> stack
'Overflow rocks!'
```

Aha! Vedere? Possiamo modificarlo!

```
>>> id(stack)
140128123911472
```

No. Mentre sembra che possiamo cambiare la stringa nominata dallo `stack` variabile, ciò che effettivamente facciamo è creare un nuovo oggetto per contenere il risultato della concatenazione. Siamo ingannati perché nel processo, il vecchio oggetto non va da nessuna parte, quindi viene distrutto. In un'altra situazione, sarebbe stato più ovvio:

```
>>> stack = "Stack"
>>> stackoverflow = stack + "Overflow"
>>> id(stack)
140128069348184
>>> id(stackoverflow)
140128123911480
```

In questo caso è chiaro che se vogliamo mantenere la prima stringa, abbiamo bisogno di una copia. Ma è così ovvio per altri tipi?

Esercizio

Ora, sapendo come funzionano i tipi immutabili, cosa diresti con il codice sottostante? È saggio?

```
s = ""
for i in range(1, 1000):
    s += str(i)
    s += ","
```

Mutables

Un oggetto di un tipo mutevole può essere cambiato, ed è cambiato *in-situ* . Non vengono eseguite copie implicite.

Questa categoria include: elenchi, dizionari, filtri e set.

Continuiamo a giocare con la nostra piccola funzione di `id` .

```
>>> b = bytearray(b'Stack')
>>> b
bytearray(b'Stack')
>>> b = bytearray(b'Stack')
>>> id(b)
140128030688288
>>> b += b'Overflow'
>>> b
bytearray(b'StackOverflow')
>>> id(b)
140128030688288
```

(Come nota a margine, uso i byte che contengono i dati ascii per chiarire il mio punto, ma ricorda che i byte non sono progettati per contenere dati testuali.

Cosa abbiamo? Creiamo un bytearray, lo modifichiamo e usando l' `id` , possiamo assicurarci che questo sia lo stesso oggetto, modificato. Non una copia di esso.

Ovviamente, se un oggetto viene modificato spesso, un tipo mutevole svolge un lavoro molto migliore di un tipo immutabile. Sfortunatamente, la realtà di questa proprietà è spesso dimenticata quando fa più male.

```
>>> c = b
>>> c += b' rocks!'
>>> c
bytearray(b'StackOverflow rocks!')
```

Va bene...

```
>>> b
bytearray(b'StackOverflow rocks!')
```

Waiiit un secondo ...

```
>>> id(c) == id(b)
True
```

Infatti. `c` non è una copia di `b` . `c` è `b` .

Esercizio

Ora è meglio capire quale effetto collaterale è implicito da un tipo mutevole, puoi spiegare cosa sta andando storto in questo esempio?

```
>>> ll = [ [] ]*4 # Create a list of 4 lists to contain our results
>>> ll
[[], [], [], []]
>>> ll[0].append(23) # Add result 23 to first list
>>> ll
[[23], [23], [23], [23]]
>>> # Oops...
```

Mutevole e immutabile come argomenti

Uno dei principali casi d'uso in cui uno sviluppatore deve prendere in considerazione la mutabilità è quando si passano gli argomenti a una funzione. Questo è molto importante, perché questo determinerà la possibilità per la funzione di modificare oggetti che non appartengono al suo scopo, o in altre parole se la funzione ha effetti collaterali. Questo è anche importante per capire dove deve essere reso disponibile il risultato di una funzione.

```
>>> def list_add3(lin):
    lin += [3]
    return lin

>>> a = [1, 2, 3]
>>> b = list_add3(a)
>>> b
[1, 2, 3, 3]
```

```
>>> a
[1, 2, 3, 3]
```

Qui, l'errore è pensare che `lin` , come parametro per la funzione, possa essere modificato localmente. Invece, `lin` e `a` riferimento lo stesso oggetto. Poiché questo oggetto è mutabile, la modifica viene eseguita sul posto, il che significa che l'oggetto a cui fa riferimento sia `lin` che `a` viene modificato. `lin` non ha davvero bisogno di essere restituito, perché abbiamo già un riferimento a questo oggetto sotto forma di `a` . `a` e `b` terminano con riferimento allo stesso oggetto.

Questo non è lo stesso per le tuple.

```
>>> def tuple_add3(tin):
    tin += (3,)
    return tin

>>> a = (1, 2, 3)
>>> b = tuple_add3(a)
>>> b
(1, 2, 3, 3)
>>> a
(1, 2, 3)
```

All'inizio della funzione, `tin` e `a` riferimento allo stesso oggetto. Ma questo è un oggetto immutabile. Così, quando la funzione tenta di modificarlo, `tin` ricevere un nuovo oggetto con la modifica, mentre `a` mantiene un riferimento all'oggetto originale. In questo caso, restituire `tin` è obbligatorio, altrimenti il nuovo oggetto andrebbe perso.

Esercizio

```
>>> def yoda(prologue, sentence):
    sentence.reverse()
    prologue += " ".join(sentence)
    return prologue

>>> focused = ["You must", "stay focused"]
>>> saying = "Yoda said: "
>>> yoda_sentence = yoda(saying, focused)
```

Nota: il `reverse` funziona sul posto.

Cosa ne pensi di questa funzione? Ha effetti collaterali? Il reso è necessario? Dopo la chiamata, qual è il valore di `saying` ? Di `focused` ? Cosa succede se la funzione viene richiamata con gli stessi parametri?

Leggi Mutevole vs Immutabile (e Lavabile) in Python online:
https://riptutorial.com/it/python/topic/9182/mutevole-vs-immutabile--e-lavabile--in-python

Capitolo 130: Neo4j e Cypher utilizzano Py2Neo

Examples

Importare e autenticare

```
from py2neo import authenticate, Graph, Node, Relationship
authenticate("localhost:7474", "neo4j", "<pass>")
graph = Graph()
```

Devi assicurarti che il tuo database Neo4j esista su localhost: 7474 con le credenziali appropriate.

l'oggetto `graph` è la tua interfaccia con l'istanza neo4j nel resto del tuo codice Python. Ringraziamo piuttosto rendendo questa variabile globale, dovresti tenerla nel metodo `__init__` una classe.

Aggiunta di nodi al grafico Neo4j

```
results = News.objects.todays_news()
for r in results:
    article = graph.merge_one("NewsArticle", "news_id", r)
    article.properties["title"] = results[r]['news_title']
    article.properties["timestamp"] = results[r]['news_timestamp']
    article.push()
    [...]
```

L'aggiunta di nodi al grafico è piuttosto semplice, `graph.merge_one` è importante in quanto impedisce gli elementi duplicati. (Se si esegue lo script due volte, la seconda volta aggiorna il titolo e non crea nuovi nodi per gli stessi articoli)

`timestamp` dovrebbe essere un numero intero e non una stringa di date dato che neo4j non ha realmente un datatype di data. Ciò causa problemi di ordinamento quando si memorizza la data come '05 -06-1989 '

`article.push()` è una chiamata che in realtà commette l'operazione in neo4j. Non dimenticare questo passaggio.

Aggiunta di relazioni al grafico Neo4j

```
results = News.objects.todays_news()
for r in results:
    article = graph.merge_one("NewsArticle", "news_id", r)
    if 'LOCATION' in results[r].keys():
        for loc in results[r]['LOCATION']:
            loc = graph.merge_one("Location", "name", loc)
            try:
                rel = graph.create_unique(Relationship(article, "about_place", loc))
            except Exception, e:
```

```
                print e
```

`create_unique` è importante per evitare duplicati. Ma per il resto è un'operazione piuttosto semplice. Anche il nome della relazione è importante poiché lo si utilizzerà in casi avanzati.

Query 1: completamento automatico sui titoli di notizie

```
def get_autocomplete(text):
    query = """
    start n = node(*) where n.name =~ '(?i)%s.*' return n.name,labels(n) limit 10;
    """
    query = query % (text)
    obj = []
    for res in graph.cypher.execute(query):
        # print res[0],res[1]
        obj.append({'name':res[0],'entity_type':res[1]})
    return res
```

Questa è una query di esempio per ottenere tutti i nodi con il `name` della proprietà che inizia con il `text` dell'argomento.

Query 2: ottieni articoli di notizie per posizione in una data specifica

```
def search_news_by_entity(location,timestamp):
    query = """
    MATCH (n)-[]->(l)
    where l.name='%s' and n.timestamp='%s'
    RETURN n.news_id limit 10
    """

    query = query % (location,timestamp)

    news_ids = []
    for res in graph.cypher.execute(query):
        news_ids.append(str(res[0]))

    return news_ids
```

È possibile utilizzare questa query per trovare tutti gli articoli di notizie `(n)` connessi a una posizione `(l)` da una relazione.

Cypher Query Samples

Conta gli articoli collegati a una determinata persona nel tempo

```
MATCH (n)-[]->(l)
where l.name='Donald Trump'
RETURN n.date,count(*) order by n.date
```

Cerca altre persone / posizioni collegate agli stessi articoli di notizie di Trump con almeno 5 nodi di relazione totali.

```
MATCH (n:NewsArticle)-[]->(l)
where l.name='Donald Trump'
MATCH (n:NewsArticle)-[]->(m)
with m,count(n) as num where num>5
return labels(m)[0],(m.name), num order by num desc limit 10
```

Leggi Neo4j e Cypher utilizzano Py2Neo online: https://riptutorial.com/it/python/topic/5841/neo4j-e-cypher-utilizzano-py2neo

Capitolo 131: Nodo di elenco collegato

Examples

Scrivi un semplice nodo Elenco collegato in python

Un elenco collegato è:

- la lista vuota, rappresentata da Nessuno, o
- un nodo che contiene un oggetto cargo e un riferimento a un elenco collegato.

```
#! /usr/bin/env python

class Node:
      def __init__(self, cargo=None, next=None):
          self.car = cargo
          self.cdr = next
      def __str__(self):
          return str(self.car)


      def display(lst):
          if lst:
             w("%s " % lst)
             display(lst.cdr)
          else:
             w("nil\n")
```

Leggi Nodo di elenco collegato online: https://riptutorial.com/it/python/topic/6916/nodo-di-elenco-collegato

Capitolo 132: Oggetti di proprietà

Osservazioni

Nota : in Python 2, assicurarsi che la classe erediti dall'oggetto (rendendolo una classe di stile nuovo) in modo che tutte le caratteristiche delle proprietà siano disponibili.

Examples

Usando il decoratore @property

Il decoratore `@property` può essere utilizzato per definire metodi in una classe che si comportano come attributi. Un esempio in cui ciò può essere utile è quando si espongono informazioni che potrebbero richiedere una ricerca iniziale (costosa) e un semplice recupero in seguito.

Dato qualche modulo `foobar.py` :

```
class Foo(object):
    def __init__(self):
        self.__bar = None

    @property
    def bar(self):
        if self.__bar is None:
            self.__bar = some_expensive_lookup_operation()
        return self.__bar
```

Poi

```
>>> from foobar import Foo
>>> foo = Foo()
>>> print(foo.bar)  # This will take some time since bar is None after initialization
42
>>> print(foo.bar)  # This is much faster since bar has a value now
42
```

Utilizzo del decoratore @property per le proprietà di lettura-scrittura

Se si desidera utilizzare `@property` per implementare il comportamento personalizzato per l'impostazione e il `@property` , utilizzare questo modello:

```
class Cash(object):
    def __init__(self, value):
        self.value = value
    @property
    def formatted(self):
        return '${:.2f}'.format(self.value)
    @formatted.setter
```

```
    def formatted(self, new):
        self.value = float(new[1:])
```

Per usare questo:

```
>>> wallet = Cash(2.50)
>>> print(wallet.formatted)
$2.50
>>> print(wallet.value)
2.5
>>> wallet.formatted = '$123.45'
>>> print(wallet.formatted)
$123.45
>>> print(wallet.value)
123.45
```

Sovrascrivere solo un getter, setter o un deleter di un oggetto proprietà

Quando si eredita da una classe con una proprietà, è possibile fornire una nuova implementazione per una o più funzioni `getter` , `setter` o `deleter` della proprietà, facendo riferimento all'oggetto proprietà *nella classe padre* :

```
class BaseClass(object):
    @property
    def foo(self):
        return some_calculated_value()

    @foo.setter
    def foo(self, value):
        do_something_with_value(value)


class DerivedClass(BaseClass):
    @BaseClass.foo.setter
    def foo(self, value):
        do_something_different_with_value(value)
```

Puoi anche aggiungere un setter o deleter dove prima non ce n'era uno nella classe base.

Usando le proprietà senza decoratori

L'uso della sintassi del decoratore (con il simbolo @) è comodo, ma anche un po 'nascosto. Puoi usare le proprietà direttamente, senza decoratori. Il seguente esempio di Python 3.x mostra questo:

```
class A:
    p = 1234
    def getX (self):
        return self._x

    def setX (self, value):
        self._x = value

    def getY (self):
```

```
        return self._y

    def setY (self, value):
        self._y = 1000 + value    # Weird but possible

    def getY2 (self):
        return self._y

    def setY2 (self, value):
        self._y = value

    def getT    (self):
        return self._t

    def setT (self, value):
        self._t = value

    def getU (self):
        return self._u + 10000

    def setU (self, value):
        self._u = value - 5000

    x, y, y2 = property (getX, setX), property (getY, setY), property (getY2, setY2)
    t = property (getT, setT)
    u = property (getU, setU)

A.q = 5678

class B:
    def getZ (self):
        return self.z_

    def setZ (self, value):
        self.z_ = value

    z = property (getZ, setZ)

class C:
    def __init__ (self):
        self.offset = 1234

    def getW (self):
        return self.w_ + self.offset

    def setW (self, value):
        self.w_ = value - self.offset

    w = property (getW, setW)

a1 = A ()
a2 = A ()

a1.y2 = 1000
a2.y2 = 2000

a1.x = 5
a1.y = 6

a2.x = 7
a2.y = 8
```

```
a1.t = 77
a1.u = 88

print (a1.x, a1.y, a1.y2)
print (a2.x, a2.y, a2.y2)
print (a1.p, a2.p, a1.q, a2.q)

print (a1.t, a1.u)

b = B ()
c = C ()

b.z = 100100
c.z = 200200
c.w = 300300

print (a1.x, b.z, c.z, c.w)

c.w = 400400
c.z = 500500
b.z = 600600

print (a1.x, b.z, c.z, c.w)
```

Leggi Oggetti di proprietà online: https://riptutorial.com/it/python/topic/2050/oggetti-di-proprieta

Capitolo 133: Operatori bit a bit

introduzione

Le operazioni bit a bit alterano le stringhe binarie a livello di bit. Queste operazioni sono incredibilmente semplici e sono supportate direttamente dal processore. Queste poche operazioni sono necessarie per lavorare con driver di periferica, grafica di basso livello, crittografia e comunicazioni di rete. Questa sezione fornisce utili conoscenze ed esempi degli operatori bit a bit di Python.

Sintassi

- x << y # Bitwise Left Shift
- x >> y # Bitwise Right Shift
- x & y # Bitwise AND
- x | y # Bitwise OR
- ~ x # Bitwise NOT
- x ^ y # Bitwise XOR

Examples

Bitwise AND

L'operatore `&` eseguirà un binario **AND** , dove viene copiato un bit se esiste in **entrambi gli** operandi. Questo significa:

```
# 0 & 0 = 0
# 0 & 1 = 0
# 1 & 0 = 0
# 1 & 1 = 1

# 60 = 0b111100
# 30 = 0b011110
60 & 30
# Out: 28
# 28 = 0b11100

bin(60 & 30)
# Out: 0b11100
```

Bitwise OR

Il `|` l'operatore eseguirà un binario "o", in cui un bit viene copiato se esiste in entrambi gli operandi.

Questo significa:

```
# 0 | 0 = 0
# 0 | 1 = 1
# 1 | 0 = 1
# 1 | 1 = 1

# 60 = 0b111100
# 30 = 0b011110
60 | 30
# Out: 62
# 62 = 0b111110

bin(60 | 30)
# Out: 0b111110
```

XOR bit a bit (OR esclusivo)

L'operatore `^` eseguirà uno **XOR** binario in cui un binario `1` viene copiato se e solo se è il valore di esattamente **un** operando. Un altro modo per affermarlo è che il risultato è `1` solo se gli operandi sono diversi. Esempi inclusi:

```
# 0 ^ 0 = 0
# 0 ^ 1 = 1
# 1 ^ 0 = 1
# 1 ^ 1 = 0

# 60 = 0b111100
# 30 = 0b011110
60 ^ 30
# Out: 34
# 34 = 0b100010

bin(60 ^ 30)
# Out: 0b100010
```

Spostamento a sinistra bit a bit

L'operatore `<<` eseguirà un "spostamento a sinistra" bit a bit, in cui il valore dell'operando di sinistra viene spostato a sinistra dal numero di bit dati dall'operando di destra.

```
# 2 = 0b10
2 << 2
# Out: 8
# 8 = 0b1000

bin(2 << 2)
# Out: 0b1000
```

L'esecuzione di uno spostamento di bit sinistro di `1` equivale alla moltiplicazione per `2` :

```
7 << 1
# Out: 14
```

L'esecuzione di uno spostamento di bit sinistro di `n` equivale alla moltiplicazione per `2**n` :

```
3 << 4
# Out: 48
```

Spostamento a destra bit a bit

L'operatore `>>` eseguirà un "spostamento a destra" bit a bit, in cui il valore dell'operando di sinistra viene spostato a destra dal numero di bit forniti dall'operando di destra.

```
# 8 = 0b1000
8 >> 2
# Out: 2
# 2 = 0b10

bin(8 >> 2)
# Out: 0b10
```

L'esecuzione di uno spostamento di bit a destra di `1` equivale alla divisione di interi per `2` :

```
36 >> 1
# Out: 18

15 >> 1
# Out: 7
```

L'esecuzione di uno spostamento di bit a destra di `n` equivale alla divisione di interi per `2**n` :

```
48 >> 4
# Out: 3

59 >> 3
# Out: 7
```

Bitwise NOT

L'operatore `~` capovolgerà tutti i bit del numero. Poiché i computer usano le rappresentazioni numerate firmate , in particolare la notazione a complemento dei due per codificare i numeri binari negativi dove i numeri negativi sono scritti con uno iniziale (1) invece che uno zero iniziale (0).

Ciò significa che se si utilizzavano 8 bit per rappresentare i numeri del complemento a due, si trattano i pattern da `0000 0000` a `0111 1111` per rappresentare i numeri da 0 a 127 e riservare `1xxx xxxx` per rappresentare i numeri negativi.

Numeri a complemento a otto bit

bits	Valore senza segno	Valore a complemento a due
0000 0000	0	0
0000 0001	1	1

bits	Valore senza segno	Valore a complemento a due
0000 0010	2	2
0111 1110	126	126
0111 1111	127	127
1000 0000	128	-128
1000 0001	129	-127
1000 0010	130	-126
1111 1110	254	-2
1111 1111	255	-1

In sostanza, ciò significa che mentre `1010 0110` ha un valore senza segno di 166 (arrivato aggiungendo `(128 * 1) + (64 * 0) + (32 * 1) + (16 * 0) + (8 * 0) + (4 * 1) + (2 * 1) + (1 * 0)`), ha un valore di complemento a due di -90 (arrivato aggiungendo `(128 * 1) - (64 * 0) - (32 * 1) - (16 * 0) - (8 * 0) - (4 * 1) - (2 * 1) - (1 * 0)` e completano il valore).

In questo modo, i numeri negativi vanno da -128 (`1000 0000`). Zero (0) è rappresentato come `0000 0000` e meno uno (-1) come `1111 1111` .

In generale, però, questo significa `~n = -n - 1` .

```
# 0 = 0b0000 0000
~0
# Out: -1
# -1 = 0b1111 1111

# 1 = 0b0000 0001
~1
# Out: -2
# -2 = 1111 1110

# 2 = 0b0000 0010
~2
# Out: -3
# -3 = 0b1111 1101

# 123 = 0b0111 1011
~123
# Out: -124
# -124 = 0b1000 0100
```

Nota , l'effetto complessivo di questa operazione quando applicato a numeri positivi può essere riassunto:

```
~n -> -|n+1|
```

E poi, se applicato a numeri negativi, l'effetto corrispondente è:

```
    ~-n -> |n-1|
```

I seguenti esempi illustrano questa ultima regola ...

```
# -0 = 0b0000 0000
~-0
# Out: -1
# -1 = 0b1111 1111
# 0 is the obvious exception to this rule, as -0 == 0 always

# -1 = 0b1000 0001
~-1
# Out: 0
# 0 = 0b0000 0000

# -2 = 0b1111 1110
~-2
# Out: 1
# 1 = 0b0000 0001

# -123 = 0b1111 1011
~-123
# Out: 122
# 122 = 0b0111 1010
```

Operazioni interne

Tutti gli operatori Bitwise (eccetto `~`) hanno le proprie versioni sul posto

```
a = 0b001
a &= 0b010
# a = 0b000

a = 0b001
a |= 0b010
# a = 0b011

a = 0b001
a <<= 2
# a = 0b100

a = 0b100
a >>= 2
# a = 0b001

a = 0b101
a ^= 0b011
# a = 0b110
```

Leggi Operatori bit a bit online: https://riptutorial.com/it/python/topic/730/operatori-bit-a-bit

Capitolo 134: Operatori booleani

Examples

e

Valuta il secondo argomento se e solo se entrambi gli argomenti sono veri. Altrimenti valuta il primo argomento falso.

```
x = True
y = True
z = x and y # z = True

x = True
y = False
z = x and y # z = False

x = False
y = True
z = x and y # z = False

x = False
y = False
z = x and y # z = False

x = 1
y = 1
z = x and y # z = y, so z = 1, see `and` and `or` are not guaranteed to be a boolean

x = 0
y = 1
z = x and y # z = x, so z = 0 (see above)

x = 1
y = 0
z = x and y # z = y, so z = 0 (see above)

x = 0
y = 0
z = x and y # z = x, so z = 0 (see above)
```

Gli `1` nell'esempio sopra possono essere cambiati in qualsiasi valore di verità, e gli `0` possono essere cambiati in qualsiasi valore di falso.

o

Valuta il primo argomento di verità se uno degli argomenti è vero. Se entrambi gli argomenti sono falsi, valuta il secondo argomento.

```
x = True
y = True
z = x or y # z = True
```

```
x = True
y = False
z = x or y # z = True

x = False
y = True
z = x or y # z = True

x = False
y = False
z = x or y # z = False

x = 1
y = 1
z = x or y # z = x, so z = 1, see `and` and `or` are not guaranteed to be a boolean

x = 1
y = 0
z = x or y # z = x, so z = 1 (see above)

x = 0
y = 1
z = x or y # z = y, so z = 1 (see above)

x = 0
y = 0
z = x or y # z = y, so z = 0 (see above)
```

Gli `1` nell'esempio sopra possono essere cambiati in qualsiasi valore di verità, e gli `0` possono essere cambiati in qualsiasi valore di falso.

non

Restituisce il contrario della seguente affermazione:

```
x = True
y = not x # y = False

x = False
y = not x # y = True
```

Valutazione del cortocircuito

Python valuta minimamente le espressioni booleane.

```
>>> def true_func():
...     print("true_func()")
...     return True
...
>>> def false_func():
...     print("false_func()")
...     return False
...
>>> true_func() or false_func()
true_func()
True
```

```
>>> false_func() or true_func()
false_func()
true_func()
True
>>> true_func() and false_func()
true_func()
false_func()
False
>>> false_func() and false_func()
false_func()
False
```

`e` e` or` non sono garantiti per restituire un valore booleano

Quando usi `or` , restituirà il primo valore nell'espressione se è vero, altrimenti restituirà ciecamente il secondo valore. Vale a dire `or` è equivalente a:

```
def or_(a, b):
    if a:
        return a
    else:
        return b
```

Per `and` , restituirà il suo primo valore se è falso, altrimenti restituisce l'ultimo valore:

```
def and_(a, b):
    if not a:
        return a
    else:
        return b
```

Un semplice esempio

In Python puoi confrontare un singolo elemento usando due operatori binari: uno su entrambi i lati:

```
if 3.14 < x < 3.142:
    print("x is near pi")
```

In molti (più?) Linguaggi di programmazione, questo sarebbe valutato in modo contrario alla matematica normale: `(3.14 < x) < 3.142` , ma in Python è trattato come `3.14 < x and x < 3.142` , proprio come la maggior parte dei non programmatori mi aspetterei.

Leggi Operatori booleani online: https://riptutorial.com/it/python/topic/1731/operatori-booleani

Capitolo 135: Ordinamento, minimo e massimo

Examples

Ottenere il minimo o il massimo di più valori

```
min(7,2,1,5)
# Output: 1

max(7,2,1,5)
# Output: 7
```

Usando l'argomento chiave

È possibile trovare il minimo / massimo di una sequenza di sequenze:

```
list_of_tuples = [(0, 10), (1, 15), (2, 8)]
min(list_of_tuples)
# Output: (0, 10)
```

ma se vuoi ordinare da un elemento specifico in ogni sequenza usa il `key` -argomento:

```
min(list_of_tuples, key=lambda x: x[0])         # Sorting by first element
# Output: (0, 10)

min(list_of_tuples, key=lambda x: x[1])         # Sorting by second element
# Output: (2, 8)

sorted(list_of_tuples, key=lambda x: x[0])      # Sorting by first element (increasing)
# Output: [(0, 10), (1, 15), (2, 8)]

sorted(list_of_tuples, key=lambda x: x[1])      # Sorting by first element
# Output: [(2, 8), (0, 10), (1, 15)]

import operator
# The operator module contains efficient alternatives to the lambda function
max(list_of_tuples, key=operator.itemgetter(0)) # Sorting by first element
# Output: (2, 8)

max(list_of_tuples, key=operator.itemgetter(1)) # Sorting by second element
# Output: (1, 15)

sorted(list_of_tuples, key=operator.itemgetter(0), reverse=True) # Reversed (decreasing)
# Output: [(2, 8), (1, 15), (0, 10)]

sorted(list_of_tuples, key=operator.itemgetter(1), reverse=True) # Reversed(decreasing)
# Output: [(1, 15), (0, 10), (2, 8)]
```

Argomento predefinito su max, min

Non è possibile passare una sequenza vuota in `max` o `min` :

```
min([])
```

ValueError: min () arg è una sequenza vuota

Tuttavia, con Python 3, puoi passare l'argomento della parola chiave `default` con un valore che verrà restituito se la sequenza è vuota, invece di generare un'eccezione:

```
max([], default=42)
# Output: 42
max([], default=0)
# Output: 0
```

Caso speciale: dizionari

Ottenere il minimo o il massimo o utilizzare `sorted` dipende dalle iterazioni sull'oggetto. Nel caso di `dict` , l'iterazione è solo sopra i tasti:

```
adict = {'a': 3, 'b': 5, 'c': 1}
min(adict)
# Output: 'a'
max(adict)
# Output: 'c'
sorted(adict)
# Output: ['a', 'b', 'c']
```

Per mantenere la struttura del dizionario, è necessario eseguire iterazioni su `.items()` :

```
min(adict.items())
# Output: ('a', 3)
max(adict.items())
# Output: ('c', 1)
sorted(adict.items())
# Output: [('a', 3), ('b', 5), ('c', 1)]
```

Per `sorted` , è possibile creare un `OrderedDict` per mantenere l'ordinamento pur avendo una `dict` - come struttura:

```
from collections import OrderedDict
OrderedDict(sorted(adict.items()))
# Output: OrderedDict([('a', 3), ('b', 5), ('c', 1)])
res = OrderedDict(sorted(adict.items()))
res['a']
# Output: 3
```

In base al valore

Anche in questo caso è possibile utilizzare l'argomento `key` :

```
min(adict.items(), key=lambda x: x[1])
# Output: ('c', 1)
max(adict.items(), key=operator.itemgetter(1))
# Output: ('b', 5)
sorted(adict.items(), key=operator.itemgetter(1), reverse=True)
# Output: [('b', 5), ('a', 3), ('c', 1)]
```

Ottenere una sequenza ordinata

Utilizzando **una** sequenza:

```
sorted((7, 2, 1, 5))                 # tuple
# Output: [1, 2, 5, 7]

sorted(['c', 'A', 'b'])              # list
# Output: ['A', 'b', 'c']

sorted({11, 8, 1})                   # set
# Output: [1, 8, 11]

sorted({'11': 5, '3': 2, '10': 15})  # dict
# Output: ['10', '11', '3']          # only iterates over the keys

sorted('bdca')                       # string
# Output: ['a','b','c','d']
```

Il risultato è sempre una nuova `list` ; i dati originali rimangono invariati.

Minimo e Massimo di una sequenza

Ottenere il minimo di una sequenza (iterabile) equivale all'accesso al primo elemento di una sequenza `sorted` :

```
min([2, 7, 5])
# Output: 2
sorted([2, 7, 5])[0]
# Output: 2
```

Il massimo è un po 'più complicato, perché `sorted` mantiene l'ordine e `max` restituisce il primo valore rilevato. Nel caso in cui non ci siano duplicati il massimo è uguale all'ultimo elemento del reso ordinato:

```
max([2, 7, 5])
# Output: 7
sorted([2, 7, 5])[-1]
# Output: 7
```

Ma non se ci sono più elementi che vengono valutati come aventi il valore massimo:

```
class MyClass(object):
    def __init__(self, value, name):
        self.value = value
        self.name = name
```

```
    def __lt__(self, other):
        return self.value < other.value

    def __repr__(self):
        return str(self.name)

sorted([MyClass(4, 'first'), MyClass(1, 'second'), MyClass(4, 'third')])
# Output: [second, first, third]
max([MyClass(4, 'first'), MyClass(1, 'second'), MyClass(4, 'third')])
# Output: first
```

Sono permessi tutti gli elementi contenenti iterabili che supportano le operazioni `<` o `>` .

Rendi ordinabili le classi personalizzate

`min` , `max` e `sorted` hanno bisogno che gli oggetti siano `sorted` . Per essere correttamente ordinabile, la classe deve definire tutti i 6 metodi `__lt__` , `__gt__` , `__ge__` , `__le__` , `__ne__` e `__eq__` :

```
class IntegerContainer(object):
    def __init__(self, value):
        self.value = value

    def __repr__(self):
        return "{}({})".format(self.__class__.__name__, self.value)

    def __lt__(self, other):
        print('{!r} - Test less than {!r}'.format(self, other))
        return self.value < other.value

    def __le__(self, other):
        print('{!r} - Test less than or equal to {!r}'.format(self, other))
        return self.value <= other.value

    def __gt__(self, other):
        print('{!r} - Test greater than {!r}'.format(self, other))
        return self.value > other.value

    def __ge__(self, other):
        print('{!r} - Test greater than or equal to {!r}'.format(self, other))
        return self.value >= other.value

    def __eq__(self, other):
        print('{!r} - Test equal to {!r}'.format(self, other))
        return self.value == other.value

    def __ne__(self, other):
        print('{!r} - Test not equal to {!r}'.format(self, other))
        return self.value != other.value
```

Sebbene l'implementazione di tutti questi metodi non sia necessaria, l' omissione di alcuni di essi renderà il tuo codice soggetto a bug .

Esempi:

```
alist = [IntegerContainer(5), IntegerContainer(3),
```

```
         IntegerContainer(10), IntegerContainer(7)
        ]

res = max(alist)
# Out: IntegerContainer(3) - Test greater than IntegerContainer(5)
#      IntegerContainer(10) - Test greater than IntegerContainer(5)
#      IntegerContainer(7) - Test greater than IntegerContainer(10)
print(res)
# Out: IntegerContainer(10)

res = min(alist)
# Out: IntegerContainer(3) - Test less than IntegerContainer(5)
#      IntegerContainer(10) - Test less than IntegerContainer(3)
#      IntegerContainer(7) - Test less than IntegerContainer(3)
print(res)
# Out: IntegerContainer(3)

res = sorted(alist)
# Out: IntegerContainer(3) - Test less than IntegerContainer(5)
#      IntegerContainer(10) - Test less than IntegerContainer(3)
#      IntegerContainer(10) - Test less than IntegerContainer(5)
#      IntegerContainer(7) - Test less than IntegerContainer(5)
#      IntegerContainer(7) - Test less than IntegerContainer(10)
print(res)
# Out: [IntegerContainer(3), IntegerContainer(5), IntegerContainer(7), IntegerContainer(10)]
```

`sorted` con `reverse=True` usa anche `__lt__` :

```
res = sorted(alist, reverse=True)
# Out: IntegerContainer(10) - Test less than IntegerContainer(7)
#      IntegerContainer(3) - Test less than IntegerContainer(10)
#      IntegerContainer(3) - Test less than IntegerContainer(10)
#      IntegerContainer(3) - Test less than IntegerContainer(7)
#      IntegerContainer(5) - Test less than IntegerContainer(7)
#      IntegerContainer(5) - Test less than IntegerContainer(3)
print(res)
# Out: [IntegerContainer(10), IntegerContainer(7), IntegerContainer(5), IntegerContainer(3)]
```

Ma `sorted` può usare `__gt__` se il default non è implementato:

```
del IntegerContainer.__lt__   # The IntegerContainer no longer implements "less than"

res = min(alist)
# Out: IntegerContainer(5) - Test greater than IntegerContainer(3)
#      IntegerContainer(3) - Test greater than IntegerContainer(10)
#      IntegerContainer(3) - Test greater than IntegerContainer(7)
print(res)
# Out: IntegerContainer(3)
```

I metodi di ordinamento `__lt__` un `TypeError` se non vengono implementati `__lt__` né `__gt__` :

```
del IntegerContainer.__gt__   # The IntegerContainer no longer implements "greater then"

res = min(alist)
```

TypeError: tipi non ordinabili: IntegerContainer () <IntegerContainer ()

`functools.total_ordering` decorator può essere utilizzato semplificando lo sforzo di scrivere questi metodi di confronto ricchi. Se decori la tua classe con `total_ordering` , devi implementare `__eq__` , `__ne__` e solo uno tra `__lt__` , `__le__` , `__ge__` o `__gt__` , e il decoratore riempirà il resto:

```
import functools

@functools.total_ordering
class IntegerContainer(object):
    def __init__(self, value):
        self.value = value

    def __repr__(self):
        return "{}({})".format(self.__class__.__name__, self.value)

    def __lt__(self, other):
        print('{!r} - Test less than {!r}'.format(self, other))
        return self.value < other.value

    def __eq__(self, other):
        print('{!r} - Test equal to {!r}'.format(self, other))
        return self.value == other.value

    def __ne__(self, other):
        print('{!r} - Test not equal to {!r}'.format(self, other))
        return self.value != other.value


IntegerContainer(5) > IntegerContainer(6)
# Output: IntegerContainer(5) - Test less than IntegerContainer(6)
# Returns: False

IntegerContainer(6) > IntegerContainer(5)
# Output: IntegerContainer(6) - Test less than IntegerContainer(5)
# Output: IntegerContainer(6) - Test equal to IntegerContainer(5)
# Returns True
```

Si noti come `>` (*maggiore di*) ora finisce per chiamare il metodo *meno di* , e in alcuni casi anche il metodo `__eq__` . Ciò significa anche che se la velocità è di grande importanza, è necessario implementare personalmente ciascun metodo di confronto ricco.

Estraendo N il più grande o il N più piccolo da un iterabile

Per trovare un numero (più di uno) dei valori più grandi o più piccoli di un iterabile, è possibile utilizzare il `nlargest` e `nsmallest` del `heapq` modulo:

```
import heapq

# get 5 largest items from the range

heapq.nlargest(5, range(10))
# Output: [9, 8, 7, 6, 5]

heapq.nsmallest(5, range(10))
# Output: [0, 1, 2, 3, 4]
```

Questo è molto più efficiente dell'ordinamento dell'intero iterable e quindi dell'affinamento dalla

fine o dall'inizio. Internamente queste funzioni utilizzano la struttura dati della coda di priorità heap binario , che è molto efficiente per questo caso d'uso.

Come `min` , `max` e `sorted` , queste funzioni accettano l'argomento parola `key` opzionale, che deve essere una funzione che, dato un elemento, restituisce la sua chiave di ordinamento.

Ecco un programma che estrae 1000 righe più lunghe da un file:

```
import heapq
with open(filename) as f:
    longest_lines = heapq.nlargest(1000, f, key=len)
```

Qui apriamo il file e passiamo il file handle `f` a `nlargest` . L'iterazione del file produce ogni riga del file come una stringa separata; `nlargest` passa quindi ogni elemento (o linea) viene passato alla funzione `len` per determinare la sua chiave di ordinamento. `len` , data una stringa, restituisce la lunghezza della linea in caratteri.

Questo ha solo bisogno di spazio per un elenco di 1000 linee più grandi finora, che può essere contrastato con

```
longest_lines = sorted(f, key=len)[1000:]
```

che dovrà conservare *l'intero file in memoria* .

Leggi Ordinamento, minimo e massimo online:
https://riptutorial.com/it/python/topic/252/ordinamento--minimo-e-massimo

Capitolo 136: os.path

introduzione

Questo modulo implementa alcune utili funzioni sui percorsi. I parametri del percorso possono essere passati come stringhe o byte. Le applicazioni sono incoraggiate a rappresentare i nomi dei file come stringhe di caratteri (Unicode).

Sintassi

- os.path.join (a, * p)
- os.path.basename (p)
- os.path.dirname (p)
- os.path.split (p)
- os.path.splitext (p)

Examples

Join Paths

Per unire due o più componenti del percorso, prima importa il modulo os di python e poi usa:

```
import os
os.path.join('a', 'b', 'c')
```

Il vantaggio dell'utilizzo di os.path è che consente al codice di rimanere compatibile su tutti i sistemi operativi, in quanto utilizza il separatore appropriato per la piattaforma su cui è in esecuzione.

Ad esempio, il risultato di questo comando su Windows sarà:

```
>>> os.path.join('a', 'b', 'c')
'a\b\c'
```

In un sistema operativo Unix:

```
>>> os.path.join('a', 'b', 'c')
'a/b/c'
```

Percorso assoluto dal percorso relativo

Usa `os.path.abspath` :

```
>>> os.getcwd()
'/Users/csaftoiu/tmp'
```

```
>>> os.path.abspath('foo')
'/Users/csaftoiu/tmp/foo'
>>> os.path.abspath('../foo')
'/Users/csaftoiu/foo'
>>> os.path.abspath('/foo')
'/foo'
```

Manipolazione dei componenti del percorso

Per dividere un componente fuori dal percorso:

```
>>> p = os.path.join(os.getcwd(), 'foo.txt')
>>> p
'/Users/csaftoiu/tmp/foo.txt'
>>> os.path.dirname(p)
'/Users/csaftoiu/tmp'
>>> os.path.basename(p)
'foo.txt'
>>> os.path.split(os.getcwd())
('/Users/csaftoiu/tmp', 'foo.txt')
>>> os.path.splitext(os.path.basename(p))
('foo', '.txt')
```

Ottieni la directory principale

```
os.path.abspath(os.path.join(PATH_TO_GET_THE_PARENT, os.pardir))
```

Se il percorso specificato esiste.

per verificare se il percorso specificato esiste

```
path = '/home/john/temp'
os.path.exists(path)
#this returns false if path doesn't exist or if the path is a broken symbolic link
```

controlla se il percorso dato è una directory, un file, un link simbolico, un punto di mount ecc.

per verificare se il percorso indicato è una directory

```
dirname = '/home/john/python'
os.path.isdir(dirname)
```

per verificare se il percorso indicato è un file

```
filename = dirname + 'main.py'
os.path.isfile(filename)
```

per verificare se il percorso indicato è un collegamento simbolico

```
symlink = dirname + 'some_sym_link'
os.path.islink(symlink)
```

per verificare se il percorso indicato è un punto di montaggio

```
mount_path = '/home'
os.path.ismount(mount_path)
```

Leggi os.path online: https://riptutorial.com/it/python/topic/1380/os-path

Capitolo 137: Ottimizzazione delle prestazioni

Osservazioni

Quando si tenta di migliorare le prestazioni di uno script Python, prima di tutto si dovrebbe essere in grado di trovare il collo di bottiglia del proprio script e notare che nessuna ottimizzazione può compensare una scelta sbagliata nelle strutture dati o un difetto nella progettazione dell'algoritmo. Identificare i colli di bottiglia delle prestazioni può essere fatto profilando il tuo script. In secondo luogo, non cercare di ottimizzare troppo presto il processo di codifica a scapito della leggibilità / design / qualità. Donald Knuth ha rilasciato la seguente dichiarazione sull'ottimizzazione:

> "Dovremmo dimenticare le piccole efficienze, diciamo circa il 97% delle volte: l'ottimizzazione prematura è la radice di tutto il male. Tuttavia non dovremmo perdere le nostre opportunità in quel 3% critico ".

Examples

Codice di profilazione

Prima di tutto dovresti essere in grado di trovare il collo di bottiglia del tuo script e notare che nessuna ottimizzazione può compensare una scelta sbagliata nella struttura dei dati o un difetto nella progettazione dell'algoritmo. In secondo luogo, non cercare di ottimizzare troppo presto il processo di codifica a scapito della leggibilità / design / qualità. Donald Knuth ha rilasciato la seguente dichiarazione sull'ottimizzazione:

> "Dovremmo dimenticare le piccole efficienze, diciamo circa il 97% delle volte: l'ottimizzazione prematura è la radice di tutto il male, ma non dovremmo perdere le nostre opportunità in quel 3% critico"

Per profilare il tuo codice hai diversi strumenti: `cProfile` (o il `profile` più lento) dalla libreria standard, `line_profiler` e `timeit` . Ognuno di loro ha uno scopo diverso.

`cProfile` è un profiler deterministico: la funzione chiamata, la funzione di ritorno e gli eventi di eccezione sono monitorati e vengono eseguiti tempi precisi per gli intervalli tra questi eventi (fino a 0,001 s). La documentazione della biblioteca ([https://docs.python.org/2/library/profile.html][1]) ci fornisce un caso di utilizzo semplice

```
import cProfile
def f(x):
    return "42!"
cProfile.run('f(12)')
```

O se preferisci avvolgere parti del tuo codice esistente:

```
import cProfile, pstats, StringIO
pr = cProfile.Profile()
```

```
pr.enable()
# ... do something ...
# ... long ...
pr.disable()
sortby = 'cumulative'
ps = pstats.Stats(pr, stream=s).sort_stats(sortby)
ps.print_stats()
print s.getvalue()
```

Ciò creerà le uscite come nella tabella sottostante, in cui è possibile vedere rapidamente dove trascorre la maggior parte del tempo il programma e identificare le funzioni da ottimizzare.

```
         3 function calls in 0.000 seconds

Ordered by: standard name
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    0.000    0.000 <stdin>:1(f)
     1    0.000    0.000    0.000    0.000 <string>:1(<module>)
     1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Il modulo `line_profiler` ([https://github.com/rkern/line_profiler][1]) è utile per avere un'analisi linea per linea del tuo codice. Questo ovviamente non è gestibile per script lunghi ma è indirizzato a frammenti. Vedi la documentazione per maggiori dettagli. Il modo più semplice per iniziare è usare lo script kernprof come spiegato nella pagina del pacchetto, nota che dovrai specificare manualmente le funzioni del profilo.

```
$ kernprof -l script_to_profile.py
```

kernprof creerà un'istanza di LineProfiler e la inserirà nello spazio `__builtins__` nomi `__builtins__` con il nome del profilo. È stato scritto per essere usato come decoratore, quindi nella sceneggiatura, decori le funzioni che vuoi profilare con `@profile` .

```
@profile
def slow_function(a, b, c):
    ...
```

Il comportamento predefinito di kernprof è di mettere i risultati in un file binario `script_to_profile.py.lprof` . Puoi dire a kernprof di visualizzare immediatamente i risultati formattati sul terminale con l'opzione [-v / - visualizza]. Altrimenti, puoi visualizzare i risultati in un secondo momento in questo modo:

```
$ python -m line_profiler script_to_profile.py.lprof
```

Infine `timeit` fornisce un modo semplice per testare un liner o una piccola espressione sia dalla riga di comando che dalla shell python. Questo modulo risponderà a domande come, è più veloce fare una comprensione di lista o usare la `list()` built-in `list()` quando si trasforma un set in una lista. Cerca la parola chiave `setup` o `-s` per aggiungere il codice di configurazione.

```
>>> import timeit
>>> timeit.timeit('"-".join(str(n) for n in range(100))', number=10000)
```

```
0.8187260627746582
```

da un terminale

```
$ python -m timeit '"-".join(str(n) for n in range(100))'
10000 loops, best of 3: 40.3 usec per loop
```

Leggi Ottimizzazione delle prestazioni online:
https://riptutorial.com/it/python/topic/5889/ottimizzazione-delle-prestazioni

Capitolo 138: Pallone

introduzione

Flask è un framework per micro-web Python utilizzato per gestire i principali siti Web tra cui Pintrest, Twilio e Linkedin. Questo argomento spiega e dimostra la varietà di funzioni offerte da Flask per lo sviluppo web sia front-end che back-end.

Sintassi

- @ app.route ("/ urlpath", methods = ["GET", "POST", "DELETE", "PUTS", "HEAD", "OPTIONS"])
- @ app.route ("/ urlpath / <param>", methods = ["GET", "POST", "DELETE", "PUTS", "HEAD", "OPTIONS"])

Examples

Le basi

Il seguente esempio è un esempio di un server di base:

```
# Imports the Flask class
from flask import Flask
# Creates an app and checks if its the main or imported
app = Flask(__name__)

# Specifies what URL triggers hello_world()
@app.route('/')
# The function run on the index route
def hello_world():
    # Returns the text to be displayed
    return "Hello World!"

# If this script isn't an import
if __name__ == "__main__":
    # Run the app until stopped
    app.run()
```

L'esecuzione di questo script (con tutte le giuste dipendenze installate) dovrebbe avviare un server locale. L'host è `127.0.0.1` comunemente noto come **localhost** . Questo server funziona di default sulla porta **5000** . Per accedere al tuo webserver, apri un browser web e inserisci l'URL `localhost:5000` o `127.0.0.1:5000` (nessuna differenza). Attualmente, solo il tuo computer può accedere al webserver.

`app.run()` ha tre parametri, **host** , **porta** e **debug** . L'host è per impostazione predefinita `127.0.0.1` , ma impostandolo su `0.0.0.0` il server Web sarà accessibile da qualsiasi dispositivo sulla rete utilizzando l'indirizzo IP privato nell'URL. la porta è di default 5000 ma se il parametro è impostato sulla porta `80` , gli utenti non dovranno specificare un numero di porta in quanto i browser

utilizzano la porta 80 per impostazione predefinita. Per quanto riguarda l'opzione di debug, durante il processo di sviluppo (mai in produzione) aiuta a impostare questo parametro su True, poiché il server si riavvierà quando verranno apportate modifiche al progetto Flask.

```
if __name__ == "__main__":
    app.run(host="0.0.0.0", port=80, debug=True)
```

URL di instradamento

Con Flask, il routing degli URL viene tradizionalmente fatto utilizzando i decoratori. Questi decoratori possono essere utilizzati per il routing statico, oltre agli URL di routing con parametri. Per il seguente esempio, immagina che questo script di Flask stia eseguendo il sito `www.example.com` .

```
@app.route("/")
def index():
    return "You went to www.example.com"

@app.route("/about")
def about():
    return "You went to www.example.com/about"

@app.route("/users/guido-van-rossum")
    return "You went to www.example.com/guido-van-rossum"
```

Con quell'ultimo percorso, puoi vedere che dato un URL con / users / e il nome del profilo, potremmo restituire un profilo. Dal momento che sarebbe orribilmente inefficiente e disordinato includere un `@app.route()` per ogni utente, Flask offre di prendere parametri dall'URL:

```
@app.route("/users/<username>")
def profile(username):
    return "Welcome to the profile of " + username

cities = ["OMAHA", "MELBOURNE", "NEPAL", "STUTTGART", "LIMA", "CAIRO", "SHANGHAI"]

@app.route("/stores/locations/<city>")
def storefronts(city):
    if city in cities:
        return "Yes! We are located in " + city
    else:
        return "No. We are not located in " + city
```

Metodi HTTP

I due metodi HTTP più comuni sono **GET** e **POST** . Flask può eseguire codice diverso dallo stesso URL in base al metodo HTTP utilizzato. Ad esempio, in un servizio Web con account, è più conveniente instradare la pagina di accesso e la procedura di accesso attraverso lo stesso URL. Una richiesta GET, la stessa che viene eseguita quando si apre un URL nel browser deve mostrare il modulo di accesso, mentre una richiesta POST (che porta i dati di accesso) deve essere elaborata separatamente. Viene inoltre creata una route per gestire il metodo DELETE e PUT HTTP.

```
@app.route("/login", methods=["GET"])
def login_form():
    return "This is the login form"
@app.route("/login", methods=["POST"])
def login_auth():
    return "Processing your data"
@app.route("/login", methods=["DELETE", "PUT"])
def deny():
    return "This method is not allowed"
```

Per semplificare un po 'il codice, possiamo importare il pacchetto di `request` dal pallone.

```
from flask import request

@app.route("/login", methods=["GET", "POST", "DELETE", "PUT"])
def login():
    if request.method == "DELETE" or request.method == "PUT":
        return "This method is not allowed"
    elif request.method == "GET":
        return "This is the login forum"
    elif request.method == "POST":
        return "Processing your data"
```

Per recuperare i dati dalla richiesta POST, dobbiamo usare il pacchetto di `request` :

```
from flask import request
@app.route("/login", methods=["GET", "POST", "DELETE", "PUT"])
def login():
    if request.method == "DELETE" or request.method == "PUT":
        return "This method is not allowed"
    elif request.method == "GET":
        return "This is the login forum"
    elif request.method == "POST":
        return "Username was " + request.form["username"] + " and password was " +
request.form["password"]
```

File e modelli

Invece di digitare il nostro markup HTML nelle istruzioni return, possiamo usare la funzione `render_template()` :

```
from flask import Flask
from flask import render_template
app = Flask(__name__)

@app.route("/about")
def about():
    return render_template("about-us.html")

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=80, debug=True)
```

Questo userà il nostro file modello `about-us.html` . Per garantire che la nostra applicazione possa trovare questo file, dobbiamo organizzare la nostra directory nel seguente formato:

```
- application.py
/templates
    - about-us.html
    - login-form.html
/static
    /styles
        - about-style.css
        - login-style.css
    /scripts
        - about-script.js
        - login-script.js
```

Ancora più importante, i riferimenti a questi file nell'HTML devono essere come questi:

```
<link rel="stylesheet" type="text/css", href="{{url_for('static', filename='styles/about-
style.css')}}">
```

che dirigerà l'applicazione alla ricerca di `about-style.css` nella cartella styles nella cartella statica. Lo stesso formato di percorso si applica a tutti i riferimenti a immagini, stili, script o file.

Jinja Templating

Simile a Meteor.js, Flask si integra bene con i servizi di template front-end. Flask utilizza per impostazione predefinita Jinja Templating. I modelli consentono di utilizzare piccoli frammenti di codice nel file HTML come condizionali o loop.

Quando eseguiamo il rendering di un modello, tutti i parametri oltre il nome del file modello vengono passati al servizio di template HTML. Il seguente percorso passerà il nome utente e la data unita (da una funzione da qualche altra parte) nel codice HTML.

```
@app.route("/users/<username>)
def profile(username):
    joinedDate = get_joined_date(username) # This function's code is irrelevant
    awards = get_awards(username) # This function's code is irrelevant
    # The joinDate is a string and awards is an array of strings
    return render_template("profile.html", username=username, joinDate=joinDate,
awards=awards)
```

Quando questo template è reso, può usare le variabili passate dalla funzione `render_template()` . Ecco i contenuti di `profile.html` :

```
<!DOCTYPE html>
<html>
    <head>
        # if username
            <title>Profile of {{ username }}</title>
        # else
            <title>No User Found</title>
        # endif
    <head>
    <body>
        {% if username %}
            <h1>{{ username }} joined on the date {{ date }}</h1>
            {% if len(awards) > 0 %}
                <h3>{{ username }} has the following awards:</h3>
```

```
                <ul>
                {% for award in awards %}
                    <li>{{award}}</li>
                {% endfor %}
                </ul>
            {% else %}
                <h3>{{ username }} has no awards</h3>
            {% endif %}
        {% else %}
            <h1>No user was found under that username</h1>
        {% endif %}
        {# This is a comment and doesn't affect the output #}
    </body>
</html>
```

I seguenti delimitatori sono usati per diverse interpretazioni:

- `{% ... %}` indica una dichiarazione
- `{{ ... }}` denota un'espressione in cui viene fornito un modello
- `{# ... #}` denota un commento (non incluso nell'output del modello)
- `{# ... ##` implica che il resto della riga debba essere interpretato come una dichiarazione

L'oggetto della richiesta

L'oggetto `request` fornisce informazioni sulla richiesta che è stata fatta al percorso. Per utilizzare questo oggetto, deve essere importato dal modulo flask:

```
from flask import request
```

Parametri URL

Negli esempi precedenti sono stati utilizzati `request.method` e `request.form` , tuttavia è anche possibile utilizzare la proprietà `request.args` per recuperare un dizionario delle chiavi / valori nei parametri URL.

```
@app.route("/api/users/<username>")
def user_api(username):
    try:
        token = request.args.get("key")
        if key == "pA55w0Rd":
            if isUser(username): # The code of this method is irrelevant
                joined = joinDate(username) # The code of this method is irrelevant
                return "User " + username + " joined on " + joined
            else:
                return "User not found"
        else:
            return "Incorrect key"
    # If there is no key parameter
    except KeyError:
        return "No key provided"
```

Per autenticare correttamente in questo contesto, sarebbe necessario il seguente URL

(sostituendo il nome utente con qualsiasi nome utente:

`www.example.com/api/users/guido-van-rossum?key=pa55w0Rd`

Upload di file

Se un caricamento di file faceva parte del modulo inviato in una richiesta POST, i file possono essere gestiti utilizzando l'oggetto `request` :

```
@app.route("/upload", methods=["POST"])
def upload_file():
    f = request.files["wordlist-upload"]
    f.save("/var/www/uploads/" + f.filename) # Store with the original filename
```

Biscotti

La richiesta può anche includere cookie in un dizionario simile ai parametri URL.

```
@app.route("/home")
def home():
    try:
        username = request.cookies.get("username")
        return "Your stored username is " + username
    except KeyError:
        return "No username cookies was found")
```

Leggi Pallone online: https://riptutorial.com/it/python/topic/8682/pallone

Capitolo 139: Persistenza di Python

Sintassi

- pickle.dump (obj, file, protocol = None, *, fix_imports = True)
- pickle.load (file, *, fix_imports = True, encoding = "ASCII", errors = "strict")

Parametri

Parametro	Dettagli
obj	rappresentazione decapitata di obj nel file oggetto file aperto
protocollo	un intero, dice al pickler di usare il protocollo dato, `0` -ASCII, `1` - vecchio formato binario
file	L'argomento del file deve avere un metodo write () `wb` per il metodo *dump* e per caricare il metodo read () `rb`

Examples

Persistenza di Python

Oggetti come numeri, elenchi, dizionari, strutture nidificate e oggetti istanza di classe vivono nella memoria del computer e vengono persi non appena termina lo script.

pickle memorizza i dati in modo persistente in un file separato.

la rappresentazione decapitata di un oggetto è sempre un oggetto byte in tutti i casi, quindi è necessario aprire i file in `wb` per memorizzare i dati e `rb` per caricare i dati da pickle.

i dati potrebbero essere di qualsiasi tipo, ad esempio

```
data={'a':'some_value',
     'b':[9,4,7],
     'c':['some_str','another_str','spam','ham'],
     'd':{'key':'nested_dictionary'},
     }
```

Immagazzina dati

```
import pickle
file=open('filename','wb')  #file object in binary write mode
pickle.dump(data,file)      #dump the data in the file object
file.close()                #close the file to write into the file
```

Caricare dati

```
import pickle
file=open('filename','rb')  #file object in binary read mode
data=pickle.load(file)      #load the data back
file.close()

>>>data
{'b': [9, 4, 7], 'a': 'some_value', 'd': {'key': 'nested_dictionary'},
 'c': ['some_str', 'another_str', 'spam', 'ham']}
```

I seguenti tipi possono essere decapati

1. Nessuna, vera e falsa
2. numeri interi, numeri in virgola mobile, numeri complessi
3. stringhe, byte, bytearrays
4. tuple, elenchi, set e dizionari contenenti solo oggetti selezionabili
5. funzioni definite al livello più alto di un modulo (usando def, non lambda)
6. funzioni integrate definite al livello più alto di un modulo
7. classi che sono definite al livello più alto di un modulo
8. istanze di tali classi il cui **dict** o il risultato di chiamare **getstate** ()

Funzione di utilità per salvare e caricare

Salva dati da e verso file

```
import pickle
def save(filename,object):
    file=open(filename,'wb')
    pickle.dump(object,file)
    file.close()

def load(filename):
    file=open(filename,'rb')
    object=pickle.load(file)
    file.close()
    return object


>>>list_object=[1,1,2,3,5,8,'a','e','i','o','u']
>>>save(list_file,list_object)
>>>new_list=load(list_file)
>>>new_list
[1, 1, 2, 3, 5, 8, 'a', 'e', 'i', 'o', 'u'
```

Leggi Persistenza di Python online: https://riptutorial.com/it/python/topic/7810/persistenza-di-python

Capitolo 140: Pila

introduzione

Una pila è un contenitore di oggetti che vengono inseriti e rimossi secondo il principio LIFO (last-in first-out). Negli stack pushdown sono consentite solo due operazioni: **spingere l'elemento nella pila e far uscire l'elemento dalla pila** . Una pila è una struttura di dati ad accesso limitato: gli **elementi possono essere aggiunti e rimossi dallo stack solo nella parte superiore** . Ecco una definizione strutturale di una pila: una pila è vuota o è composta da una cima e il resto è una pila.

Sintassi

- stack = [] # Crea lo stack
- stack.append (oggetto) # Aggiungi oggetto all'inizio della pila
- stack.pop () -> oggetto # Restituisce la maggior parte degli oggetti in cima allo stack e la rimuove anche
- lista [-1] -> oggetto # Guarda l'oggetto più in alto senza rimuoverlo

Osservazioni

Da Wikipedia :

> In informatica, uno *stack* è un tipo di dati astratto che funge da raccolta di elementi, con due operazioni principali: *push* , che aggiunge un elemento alla raccolta e *pop* , che rimuove l'ultimo elemento aggiunto che non è stato ancora rimosso.

A causa del modo in cui i loro elementi sono accessibili, le pile sono noti anche come *Last-In, First-Out (LIFO) impila.*

In Python si possono usare gli elenchi come stack con `append()` come push e `pop()` come operazioni pop. Entrambe le operazioni vengono eseguite in tempo costante O (1).

La struttura dati `deque` di Python può anche essere usata come una pila. Rispetto agli elenchi, i `deque` s consentono operazioni push e pop con una complessità temporale costante da entrambe le estremità.

Examples

Creazione di una classe di stack con un oggetto di elenco

Utilizzando un oggetto `list` è possibile creare uno stack generico completamente funzionale con metodi di supporto come sbirciare e verificare se lo stack è vuoto. Controlla i documenti Python ufficiali per l'utilizzo `list` come `Stack` qui .

```
#define a stack class
class Stack:
    def __init__(self):
        self.items = []

    #method to check the stack is empty or not
    def isEmpty(self):
        return self.items == []

    #method for pushing an item
    def push(self, item):
        self.items.append(item)

    #method for popping an item
    def pop(self):
        return self.items.pop()

    #check what item is on top of the stack without removing it
    def peek(self):
        return self.items[-1]

    #method to get the size
    def size(self):
        return len(self.items)

    #to view the entire stack
    def fullStack(self):
        return self.items
```

Un esempio di esecuzione:

```
stack = Stack()
print('Current stack:', stack.fullStack())
print('Stack empty?:', stack.isEmpty())
print('Pushing integer 1')
stack.push(1)
print('Pushing string "Told you, I am generic stack!"')
stack.push('Told you, I am generic stack!')
print('Pushing integer 3')
stack.push(3)
print('Current stack:', stack.fullStack())
print('Popped item:', stack.pop())
print('Current stack:', stack.fullStack())
print('Stack empty?:', stack.isEmpty())
```

Produzione:

```
Current stack: []
Stack empty?: True
Pushing integer 1
Pushing string "Told you, I am generic stack!"
Pushing integer 3
Current stack: [1, 'Told you, I am generic stack!', 3]
Popped item: 3
Current stack: [1, 'Told you, I am generic stack!']
Stack empty?: False
```

Analisi delle parentesi

Le pile vengono spesso utilizzate per l'analisi. Un semplice compito di analisi consiste nel verificare se una stringa di parentesi corrisponde.

Ad esempio, la stringa `([])` corrisponde, poiché le parentesi esterne e interne formano coppie. `()<>)` non corrisponde, perché l'ultimo `)` non ha un partner. `([)]` non corrisponde, perché le coppie devono essere interamente all'interno o all'esterno di altre coppie.

```
def checkParenth(str):
    stack = Stack()
    pushChars, popChars = "<({[", ">)}]"
    for c in str:
        if c in pushChars:
            stack.push(c)
        elif c in popChars:
            if stack.isEmpty():
                return False
            else:
                stackTop = stack.pop()
                # Checks to see whether the opening bracket matches the closing one
                balancingBracket = pushChars[popChars.index(c)]
                if stackTop != balancingBracket:
                    return False
        else:
            return False

    return not stack.isEmpty()
```

Leggi Pila online: https://riptutorial.com/it/python/topic/3807/pila

Capitolo 141: pip: PyPI Package Manager

introduzione

pip è il gestore di pacchetti più utilizzato per Python Package Index, installato di default con le versioni recenti di Python.

Sintassi

- pip <comando> [opzioni] dove <comando> è uno di:
  - installare
    - Installa i pacchetti
  - disinstallazione
    - Disinstalla i pacchetti
  - congelare
    - Emetti i pacchetti installati nel formato dei requisiti
  - elenco
    - Elenca i pacchetti installati
  - mostrare
    - Mostra informazioni sui pacchetti installati
  - ricerca
    - Cerca PyPI per i pacchetti
  - ruota
    - Costruisci le ruote in base alle tue esigenze
  - cerniera lampo
    - Pacchetti singoli zip (deprecati)
  - Unzip
    - Scompatta i singoli pacchetti (deprecati)
  - fascio
    - Crea pioli (deprecato)
  - Aiuto
    - Mostra la guida per i comandi

Osservazioni

A volte, `pip` eseguirà una compilazione manuale di codice nativo. Su Linux python sceglierà automaticamente un compilatore C disponibile sul tuo sistema. Fare riferimento alla tabella seguente per la versione di Visual Studio / Visual C ++ richiesta su Windows (le versioni più recenti non funzioneranno.).

Versione Python	Versione di Visual Studio	Versione Visual C ++
2.6 - 3.2	Visual Studio 2008	Visual C ++ 9.0

Versione Python	Versione di Visual Studio	Versione Visual C ++
3.3 - 3.4	Visual Studio 2010	Visual C ++ 10.0
3.5	Visual Studio 2015	Visual C ++ 14.0

Fonte: wiki.python.org

Examples

Installa pacchetti

Per installare l'ultima versione di un pacchetto chiamato `SomePackage` :

```
$ pip install SomePackage
```

Per installare una versione specifica di un pacchetto:

```
$ pip install SomePackage==1.0.4
```

Per specificare una versione minima da installare per un pacchetto:

```
$ pip install SomePackage>=1.0.4
```

Se i comandi mostrano un errore di autorizzazione negato su Linux / Unix, utilizzare `sudo` con i comandi

---

Installa dai file dei requisiti

```
$ pip install -r requirements.txt
```

Ogni riga del file dei requisiti indica qualcosa da installare, e come gli argomenti per installare pip, i dettagli sul formato dei file sono qui: Requisiti Formato file .

Dopo aver installato il pacchetto puoi controllarlo usando il comando `freeze` :

```
$ pip freeze
```

Disinstalla pacchetti

Per disinstallare un pacchetto:

```
$ pip uninstall SomePackage
```

Per elencare tutti i pacchetti installati usando `pip`

Per elencare i pacchetti installati:

```
$ pip list
# example output
docutils (0.9.1)
Jinja2 (2.6)
Pygments (1.5)
Sphinx (1.1.2)
```

Per elencare i pacchetti obsoleti e mostrare l'ultima versione disponibile:

```
$ pip list --outdated
# example output
docutils (Current: 0.9.1 Latest: 0.10)
Sphinx (Current: 1.1.2 Latest: 1.1.3)
```

Aggiorna pacchetti

In esecuzione

```
$ pip install --upgrade SomePackage
```

aggiornerà il pacchetto `SomePackage` e tutte le sue dipendenze. Inoltre, pip rimuove automaticamente la versione precedente del pacchetto prima dell'aggiornamento.

Per aggiornare pip in sé, fallo

```
$ pip install --upgrade pip
```

su Unix o

```
$ python -m pip install --upgrade pip
```

su macchine Windows.

Aggiornamento di tutti i pacchetti obsoleti su Linux

`pip` non contiene un flag per consentire a un utente di aggiornare tutti i pacchetti obsoleti in un colpo solo. Tuttavia, questo può essere ottenuto collegando i comandi in un ambiente Linux:

```
pip list --outdated --local | grep -v '^\-e' | cut -d = -f 1  | xargs -n1 pip install -U
```

Questo comando prende tutti i pacchetti nel virtualenv locale e verifica se sono obsoleti. Da questo elenco, ottiene il nome del pacchetto e poi lo convoglia con un comando `pip install -U`. Alla fine di questo processo, tutti i pacchetti locali dovrebbero essere aggiornati.

Aggiornamento di tutti i pacchetti obsoleti su Windows

`pip` non contiene un flag per consentire a un utente di aggiornare tutti i pacchetti obsoleti in un colpo solo. Tuttavia, questo può essere ottenuto collegando i comandi in un ambiente Windows:

```
for /F "delims= " %i in ('pip list --outdated --local') do pip install -U %i
```

Questo comando prende tutti i pacchetti nel virtualenv locale e verifica se sono obsoleti. Da questo elenco, ottiene il nome del pacchetto e poi lo convoglia con un comando `pip install -U` . Alla fine di questo processo, tutti i pacchetti locali dovrebbero essere aggiornati.

Creare un file requirements.txt di tutti i pacchetti sul sistema

`pip` assiste nella creazione di file `requirements.txt` fornendo l'opzione `freeze` .

```
pip freeze > requirements.txt
```

Ciò salverà un elenco di tutti i pacchetti e la loro versione installata sul sistema in un file denominato `requirements.txt` nella cartella corrente.

Crea un file requirements.txt di pacchetti solo nella virtualenv corrente

`pip` assiste nella creazione di file `requirements.txt` fornendo l'opzione `freeze` .

```
pip freeze --local > requirements.txt
```

Il parametro `--local` restituirà solo un elenco di pacchetti e versioni installate localmente su virtualenv. I pacchetti globali non saranno elencati.

Usando una certa versione di Python con pip

Se hai installato sia Python 3 che Python 2, puoi specificare quale versione di Python vuoi usare con pip. Ciò è utile quando i pacchetti supportano solo Python 2 o 3 o quando si desidera testare con entrambi.

Se vuoi installare i pacchetti per Python 2, esegui entrambi:

```
pip install [package]
```

o:

```
pip2 install [package]
```

Se si desidera installare pacchetti per Python 3, fare:

```
pip3 install [package]
```

È anche possibile richiamare l'installazione di un pacchetto in una specifica installazione python con:

```
\path\to\that\python.exe -m pip install some_package # on Windows OR
/usr/bin/python25 -m pip install some_package # on OS-X/Linux
```

Su piattaforme OS-X / Linux / Unix è importante essere consapevoli della distinzione tra la versione di sistema di python, (che l'aggiornamento rende il rendering del sistema inoperabile) e le versioni utente di python. È **possibile** , a *seconda di quale si sta tentando di eseguire l'aggiornamento* , è necessario prefixare questi comandi con `sudo` e immettere una password.

Allo stesso modo in Windows alcune installazioni python, specialmente quelle che fanno parte di un altro pacchetto, possono finire installate nelle directory di sistema - quelle che dovrete aggiornare da una finestra di comando in esecuzione in modalità Admin - se vi sembra che sia necessario fare questo è una **buona** idea per verificare quali l'installazione di Python si sta cercando di aggiornare con un comando come `python -c"import sys;print(sys.path);"` o `py -3.5 -c"import sys;print(sys.path);"` puoi anche verificare quale pip stai cercando di eseguire con `pip --version`

Su Windows, se hai installato sia python 2 che python 3 e sul tuo percorso e il tuo python 3 è maggiore di 3.4, probabilmente avrai `py` python launcher sul tuo percorso di sistema. Puoi quindi fare trucchi come:

```
py -3 -m pip install -U some_package # Install/Upgrade some_package to the latest python 3
py -3.3 -m pip install -U some_package # Install/Upgrade some_package to python 3.3 if present
py -2 -m pip install -U some_package # Install/Upgrade some_package to the latest python 2 -
64 bit if present
py -2.7-32 -m pip install -U some_package # Install/Upgrade some_package to python 2.7 - 32
bit if present
```

Se si stanno eseguendo e mantenendo più versioni di python, si consiglia vivamente di leggere gli `venv` virtuali python `virtualenv` o `venv` che consentono di isolare sia la versione di python che i pacchetti presenti.

Installare pacchetti non ancora sul pip come ruote

Molti, puri python, i pacchetti non sono ancora disponibili sull'Indice dei Pacchetti Python come ruote ma continuano a essere installati bene. Tuttavia, alcuni pacchetti su Windows danno il temuto errore vcvarsall.bat non trovato.

Il problema è che il pacchetto che stai cercando di installare contiene un'estensione C o C ++ e al *momento* non è disponibile come una ruota *precostruita* dall'indice del pacchetto python, *pypi* , e su Windows non hai la catena di strumenti necessaria per costruire tali articoli.

La risposta più semplice è quella di visitare l' eccellente sito di Christoph Gohlke e individuare la versione **appropriata** delle librerie di cui hai bisogno. Se appropriato nel nome del pacchetto a **-cp *NN* -** deve corrispondere alla tua versione di python, cioè se stai usando windows 32 bit python *anche su win64* il nome deve includere **-win32-** e se si usa il python a 64 bit deve includere **-win_amd64 -** e quindi la versione python deve corrispondere, cioè per Python 34 il nome file **deve**

includere **-cp *34-*** , ecc. Questo è fondamentalmente la magia che pip fa per te sul sito pypi.

In alternativa, è necessario ottenere il kit di sviluppo di Windows appropriato per la versione di python che si sta utilizzando, le intestazioni per qualsiasi libreria che il pacchetto si sta tentando di creare interfacce, possibilmente le intestazioni python per la versione di python, ecc.

Python 2.7 utilizzava Visual Studio 2008, Python 3.3 e 3.4 utilizzava Visual Studio 2010 e Python 3.5+ utilizza Visual Studio 2015.

- Installare " Visual C ++ Compiler Package for Python 2.7 ", disponibile dal sito Web di Microsoft **o**
- Installare " Windows SDK per Windows 7 e .NET Framework 4 " (v7.1), disponibile dal sito Web di Microsoft **o**
- Installa Visual Studio 2015 Community Edition *(o qualsiasi versione successiva, quando vengono rilasciati)* , **assicurandoti di selezionare le opzioni per installare il supporto C & C ++** *non più quello predefinito - Mi viene detto che questo può richiedere fino a* ***8 ore*** per il download e l'installazione quindi **assicurati** che queste opzioni siano impostate al primo tentativo.

Quindi *potrebbe essere necessario* individuare i file di intestazione, *nella revisione corrispondente* per tutte le librerie a cui si collega il pacchetto desiderato e scaricarle in posizioni appropriate.

Infine puoi lasciare che pip faccia la tua build - ovviamente se il pacchetto ha dipendenze che non hai ancora potresti dover trovare anche i file header per loro.

Alternative: Vale anche la pena di guardare, *sia sul sito di pypi che su quello di Christop* , per qualsiasi versione leggermente precedente del pacchetto che si sta cercando sia in puro python *sia precostruito* per la propria piattaforma e la versione python e possibilmente usando quelli, se trovato, fino a quando il pacchetto non sarà disponibile. Allo stesso modo, se stai usando la versione più recente di python, potresti trovare che i manutentori del pacchetto impiegano un po 'di tempo per aggiornarsi, quindi per i progetti che hanno davvero **bisogno di** un pacchetto specifico potresti dover utilizzare per il momento un python leggermente più vecchio. Puoi anche controllare il sito sorgente dei pacchetti per vedere se esiste una versione biforcuta che è disponibile pre-costruita o come puro python e cercare pacchetti alternativi che forniscano le funzionalità di cui hai bisogno ma che siano disponibili - un esempio che ti viene in mente è il Cuscino , *mantenuto attivamente* , in sostituzione di PIL *attualmente non aggiornato in 6 anni e non disponibile per Python 3* .

Afterword , incoraggerei chiunque abbia questo problema ad andare sul bug tracker per il pacchetto e aggiungere, o rilanciare se non ce n'è già uno, un ticket che richiede **educatamente** che i manutentori del pacchetto forniscano una ruota su pypi per il tuo specifico combinazione di piattaforma e python, se questo è fatto, normalmente le cose andranno meglio con il tempo, alcuni manutentori di pacchetti non si rendono conto di aver perso una determinata combinazione che le persone potrebbero utilizzare.

Nota sull'installazione di pre-release

Pip segue le regole del versioning semantico e, per impostazione predefinita, preferisce i pacchetti

rilasciati rispetto ai pre-release. Quindi se un determinato pacchetto è stato rilasciato come `V0.98` e c'è anche un release candidate `V1.0-rc1` il comportamento predefinito di `pip install` di `pip install` sarà installare `V0.98` - se si desidera installare il candidato di rilascio, *si è avvisati per testare prima in un ambiente virtuale* , è possibile abilitarlo con `--pip install --pre` *nome-pacchetto* o `--pip install --pre --upgrade` *nome-pacchetto* . In molti casi, i pre-release o i release candidate potrebbero non avere ruote costruite per tutte le combinazioni di piattaforme e versioni, quindi è più probabile che si verifichino i problemi di cui sopra.

Nota sull'installazione di versioni di sviluppo

Puoi anche usare pip per installare le versioni di sviluppo dei pacchetti da github e altre posizioni, dal momento che tale codice è in flusso è molto improbabile che abbia le ruote create per questo, quindi qualsiasi pacchetto impuro richiederà la presenza degli strumenti di compilazione, e potrebbero essere interrotto in qualsiasi momento, quindi l'utente è **fortemente** incoraggiato a installare tali pacchetti solo in un ambiente virtuale.

Esistono tre opzioni per tali installazioni:

1. Scarica snapshot compresso, la maggior parte dei sistemi di controllo versione online ha la possibilità di scaricare un'istantanea compressa del codice. Questo può essere scaricato manualmente e quindi installato con *percorso di* `pip install` */ su / scaricato / file* nota che per la maggior parte dei formati di compressione pip gestirà il disimballaggio in un'area della cache, ecc.
2. Lascia che sia il pip a gestire il download e l'installazione per te con: *URL di* `pip install` */ di / package / repository* - potresti anche aver bisogno di usare i `--trusted-host` , `--client-cert` e / o `--proxy` per farlo funzionare correttamente, specialmente in un ambiente aziendale. per esempio:

```
    > py -3.5-32 -m venv demo-pip
    > demo-pip\Scripts\activate.bat
    > python -m pip install -U pip
    Collecting pip
      Using cached pip-9.0.1-py2.py3-none-any.whl
    Installing collected packages: pip
      Found existing installation: pip 8.1.1
        Uninstalling pip-8.1.1:
          Successfully uninstalled pip-8.1.1
    Successfully installed pip-9.0.1
    > pip install git+https://github.com/sphinx-doc/sphinx/
    Collecting git+https://github.com/sphinx-doc/sphinx/
      Cloning https://github.com/sphinx-doc/sphinx/ to c:\users\steve-
~1\appdata\local\temp\pip-04yn9hpp-build
    Collecting six>=1.5 (from Sphinx==1.7.dev20170506)
      Using cached six-1.10.0-py2.py3-none-any.whl
    Collecting Jinja2>=2.3 (from Sphinx==1.7.dev20170506)
      Using cached Jinja2-2.9.6-py2.py3-none-any.whl
    Collecting Pygments>=2.0 (from Sphinx==1.7.dev20170506)
      Using cached Pygments-2.2.0-py2.py3-none-any.whl
    Collecting docutils>=0.11 (from Sphinx==1.7.dev20170506)
      Using cached docutils-0.13.1-py3-none-any.whl
    Collecting snowballstemmer>=1.1 (from Sphinx==1.7.dev20170506)
      Using cached snowballstemmer-1.2.1-py2.py3-none-any.whl
```

```
    Collecting babel!=2.0,>=1.3 (from Sphinx==1.7.dev20170506)
      Using cached Babel-2.4.0-py2.py3-none-any.whl
    Collecting alabaster<0.8,>=0.7 (from Sphinx==1.7.dev20170506)
      Using cached alabaster-0.7.10-py2.py3-none-any.whl
    Collecting imagesize (from Sphinx==1.7.dev20170506)
      Using cached imagesize-0.7.1-py2.py3-none-any.whl
    Collecting requests>=2.0.0 (from Sphinx==1.7.dev20170506)
      Using cached requests-2.13.0-py2.py3-none-any.whl
    Collecting typing (from Sphinx==1.7.dev20170506)
      Using cached typing-3.6.1.tar.gz
    Requirement already satisfied: setuptools in f:\toolbuild\temp\demo-pip\lib\site-packages
(from Sphinx==1.7.dev20170506)
    Collecting sphinxcontrib-websupport (from Sphinx==1.7.dev20170506)
      Downloading sphinxcontrib_websupport-1.0.0-py2.py3-none-any.whl
    Collecting colorama>=0.3.5 (from Sphinx==1.7.dev20170506)
      Using cached colorama-0.3.9-py2.py3-none-any.whl
    Collecting MarkupSafe>=0.23 (from Jinja2>=2.3->Sphinx==1.7.dev20170506)
      Using cached MarkupSafe-1.0.tar.gz
    Collecting pytz>=0a (from babel!=2.0,>=1.3->Sphinx==1.7.dev20170506)
      Using cached pytz-2017.2-py2.py3-none-any.whl
    Collecting sqlalchemy>=0.9 (from sphinxcontrib-websupport->Sphinx==1.7.dev20170506)
      Downloading SQLAlchemy-1.1.9.tar.gz (5.2MB)
        100% |################################| 5.2MB 220kB/s
    Collecting whoosh>=2.0 (from sphinxcontrib-websupport->Sphinx==1.7.dev20170506)
      Downloading Whoosh-2.7.4-py2.py3-none-any.whl (468kB)
        100% |################################| 471kB 1.1MB/s
    Installing collected packages: six, MarkupSafe, Jinja2, Pygments, docutils,
snowballstemmer, pytz, babel, alabaster, imagesize, requests, typing, sqlalchemy, whoosh,
sphinxcontrib-websupport, colorama, Sphinx
      Running setup.py install for MarkupSafe ... done
      Running setup.py install for typing ... done
      Running setup.py install for sqlalchemy ... done
      Running setup.py install for Sphinx ... done
    Successfully installed Jinja2-2.9.6 MarkupSafe-1.0 Pygments-2.2.0 Sphinx-1.7.dev20170506
alabaster-0.7.10 babel-2.4.0 colorama-0.3.9 docutils-0.13.1 imagesize-0.7.1 pytz-2017.2
requests-2.13.0 six-1.10.0 snowballstemmer-1.2.1 sphinxcontrib-websupport-1.0.0 sqlalchemy-
1.1.9 typing-3.6.1 whoosh-2.7.4
```

Nota il prefisso `git+` per l'URL.

3. Clonare il repository usando `git` , `mercurial` o altro strumento accettabile, *preferibilmente uno strumento DVCS,* e usa il *percorso di* `pip install` */ to / cloned / repo* - questo elabora **entrambi** i file require.text ed esegue i passi di configurazione e configurazione, *puoi cambiare manualmente directory al tuo repository clonato ed esegui* `pip install -r requires.txt` *e poi* `python setup.py install` *per ottenere lo stesso effetto.* I grandi vantaggi di questo approccio sono che mentre l'operazione di clonazione iniziale può richiedere più tempo del download di snapshot è possibile eseguire l'aggiornamento al più recente, nel caso di git: `git pull origin master` e se la versione corrente contiene errori è possibile utilizzare `pip uninstall` *nome-pacchetto* quindi utilizzare i comandi `git checkout` per tornare indietro nella cronologia del repository alle versioni precedenti e riprovare.

Leggi pip: PyPI Package Manager online: https://riptutorial.com/it/python/topic/1781/pip--pypi-package-manager

Capitolo 142: Plugin ed Extension Classes

Examples

mixins

Nel linguaggio di programmazione orientato agli oggetti, un mixin è una classe che contiene metodi per l'uso da parte di altre classi senza dover essere la classe madre di quelle altre classi. Il modo in cui quelle altre classi ottengono l'accesso ai metodi del mixin dipende dalla lingua.

Fornisce un meccanismo per l'ereditarietà multipla consentendo a più classi di utilizzare la funzionalità comune, ma senza la semantica complessa dell'ereditarietà multipla. I mix sono utili quando un programmatore vuole condividere funzionalità tra classi diverse. Invece di ripetere lo stesso codice più e più volte, la funzionalità comune può essere semplicemente raggruppata in un mixin e quindi ereditata in ogni classe che lo richiede.

Quando usiamo più di un mix, l'ordine dei mixin è importante. Qui c'è un semplice esempio:

```
class Mixin1(object):
    def test(self):
        print "Mixin1"

class Mixin2(object):
    def test(self):
        print "Mixin2"

class MyClass(Mixin1, Mixin2):
    pass
```

In questo esempio chiamiamo `MyClass` e metodo di `test` ,

```
>>> obj = MyClass()
>>> obj.test()
Mixin1
```

Il risultato deve essere Mixin1 perché l'ordine è da sinistra a destra. Ciò potrebbe mostrare risultati imprevisti quando le super classi aggiungono con esso. Quindi l'ordine inverso è più buono proprio come questo:

```
class MyClass(Mixin2, Mixin1):
    pass
```

Il risultato sarà:

```
>>> obj = MyClass()
>>> obj.test()
Mixin2
```

I mixin possono essere usati per definire plugin personalizzati.

Python 3.x 3.0

```
class Base(object):
    def test(self):
        print("Base.")

class PluginA(object):
    def test(self):
        super().test()
        print("Plugin A.")

class PluginB(object):
    def test(self):
        super().test()
        print("Plugin B.")

plugins = PluginA, PluginB

class PluginSystemA(PluginA, Base):
    pass

class PluginSystemB(PluginB, Base):
    pass

PluginSystemA().test()
# Base.
# Plugin A.

 PluginSystemB().test()
# Base.
# Plugin B.
```

Plugin con classi personalizzate

In Python 3.6, PEP 487 ha aggiunto il metodo speciale `__init_subclass__` , che semplifica e estende la personalizzazione della classe senza utilizzare metaclassi . Di conseguenza, questa funzione consente di creare semplici plugin . Qui dimostriamo questa funzionalità modificando un esempio precedente :

Python 3.x 3.6

```
class Base:
    plugins = []

    def __init_subclass__(cls, **kwargs):
        super().__init_subclass__(**kwargs)
        cls.plugins.append(cls)

    def test(self):
        print("Base.")

class PluginA(Base):
    def test(self):
        super().test()
        print("Plugin A.")
```

```
class PluginB(Base):
    def test(self):
        super().test()
        print("Plugin B.")
```

risultati:

```
PluginA().test()
# Base.
# Plugin A.

PluginB().test()
# Base.
# Plugin B.

Base.plugins
# [__main__.PluginA, __main__.PluginB]
```

Leggi Plugin ed Extension Classes online: https://riptutorial.com/it/python/topic/4724/plugin-ed-extension-classes

Capitolo 143: Polimorfismo

Examples

Polimorfismo di base

Il polimorfismo è la capacità di eseguire un'azione su un oggetto indipendentemente dal suo tipo. Questo è generalmente implementato creando una classe base e avendo due o più sottoclassi che implementano tutti i metodi con la stessa firma. Qualsiasi altra funzione o metodo che manipoli questi oggetti può chiamare gli stessi metodi indipendentemente dal tipo di oggetto su cui sta operando, senza dover prima eseguire un controllo del tipo. Nella terminologia orientata agli oggetti quando la classe X estende la classe Y, allora Y è chiamato super classe o classe base e X è chiamato sottoclasse o classe derivata.

```
class Shape:
    """
    This is a parent class that is intended to be inherited by other classes
    """

    def calculate_area(self):
        """
        This method is intended to be overridden in subclasses.
        If a subclass doesn't implement it but it is called, NotImplemented will be raised.

        """
        raise NotImplemented

class Square(Shape):
    """
    This is a subclass of the Shape class, and represents a square
    """
    side_length = 2     # in this example, the sides are 2 units long

    def calculate_area(self):
        """
        This method overrides Shape.calculate_area(). When an object of type
        Square has its calculate_area() method called, this is the method that
        will be called, rather than the parent class' version.

        It performs the calculation necessary for this shape, a square, and
        returns the result.
        """
        return self.side_length * 2

class Triangle(Shape):
    """
    This is also a subclass of the Shape class, and it represents a triangle
    """
    base_length = 4
    height = 3

    def calculate_area(self):
        """
        This method also overrides Shape.calculate_area() and performs the area
```

```
        calculation for a triangle, returning the result.
        """

        return 0.5 * self.base_length * self.height

def get_area(input_obj):
    """
    This function accepts an input object, and will call that object's
    calculate_area() method. Note that the object type is not specified. It
    could be a Square, Triangle, or Shape object.
    """

    print(input_obj.calculate_area())

# Create one object of each class
shape_obj = Shape()
square_obj = Square()
triangle_obj = Triangle()

# Now pass each object, one at a time, to the get_area() function and see the
# result.
get_area(shape_obj)
get_area(square_obj)
get_area(triangle_obj)
```

Dovremmo vedere questo risultato:

Nessuna
4
6.0

Cosa succede senza polimorfismo?
Senza il polimorfismo, potrebbe essere necessario un controllo del tipo prima di eseguire un'azione su un oggetto per determinare il metodo corretto da chiamare. Il seguente **esempio di contatore** esegue lo stesso compito del codice precedente, ma senza l'uso del polimorfismo, la funzione `get_area()` deve svolgere più lavoro.

```
class Square:

    side_length = 2

    def calculate_square_area(self):
        return self.side_length ** 2

class Triangle:

    base_length = 4
    height = 3

    def calculate_triangle_area(self):
        return (0.5 * self.base_length) * self.height

def get_area(input_obj):

    # Notice the type checks that are now necessary here. These type checks
    # could get very complicated for a more complex example, resulting in
    # duplicate and difficult to maintain code.
```

```
    if type(input_obj).__name__ == "Square":
        area = input_obj.calculate_square_area()

    elif type(input_obj).__name__ == "Triangle":
        area = input_obj.calculate_triangle_area()

    print(area)

# Create one object of each class
square_obj = Square()
triangle_obj = Triangle()

# Now pass each object, one at a time, to the get_area() function and see the
# result.
get_area(square_obj)
get_area(triangle_obj)
```

Dovremmo vedere questo risultato:

4
6.0

Nota importante
Si noti che le classi utilizzate nell'esempio del contatore sono classi "new style" e ereditano implicitamente dalla classe dell'oggetto se si utilizza Python 3. Il polimorfismo funzionerà sia in Python 2.x che in 3.x, ma il codice del polimorfo del controesempio solleverà un'eccezione se eseguito in un interprete Python 2.x perché type (input_obj). **il nome** restituirà "istanza" al posto del nome della classe se non eredita esplicitamente dall'oggetto, risultando in un'area che non viene mai assegnata.

Duck Typing

Polymorphism senza ereditarietà sotto forma di anatra digitata come disponibile in Python grazie al suo sistema di battitura dinamico. Questo significa che finché le classi contengono gli stessi metodi l'interprete Python non fa distinzioni tra loro, poiché l'unico controllo delle chiamate avviene in fase di esecuzione.

```
class Duck:
    def quack(self):
        print("Quaaaaaack!")
    def feathers(self):
        print("The duck has white and gray feathers.")

class Person:
    def quack(self):
        print("The person imitates a duck.")
    def feathers(self):
        print("The person takes a feather from the ground and shows it.")
    def name(self):
        print("John Smith")

def in_the_forest(obj):
    obj.quack()
    obj.feathers()
```

```
donald = Duck()
john = Person()
in_the_forest(donald)
in_the_forest(john)
```

L'output è:

> Quaaaaaack!
> L'anatra ha piume bianche e grigie.
> La persona imita un'anatra.
> La persona prende una piuma da terra e la mostra.

Leggi Polimorfismo online: https://riptutorial.com/it/python/topic/5100/polimorfismo

Capitolo 144: PostgreSQL

Examples

Iniziare

PostgreSQL è un database open source attivamente sviluppato e maturo. Usando il modulo `psycopg2` , possiamo eseguire query sul database.

Installazione usando pip

```
pip install psycopg2
```

Utilizzo di base

Supponiamo di avere una tabella `my_table` nel database `my_database` definita come segue.

id	nome di battesimo	cognome
1	John	daino

Possiamo usare il modulo `psycopg2` per eseguire query sul database nel modo seguente.

```
import psycopg2

# Establish a connection to the existing database 'my_database' using
# the user 'my_user' with password 'my_password'
con = psycopg2.connect("host=localhost dbname=my_database user=my_user password=my_password")

# Create a cursor
cur = con.cursor()

# Insert a record into 'my_table'
cur.execute("INSERT INTO my_table(id, first_name, last_name) VALUES (2, 'Jane', 'Doe');")

# Commit the current transaction
con.commit()

# Retrieve all records from 'my_table'
cur.execute("SELECT * FROM my_table;")
results = cur.fetchall()

# Close the database connection
con.close()

# Print the results
print(results)

# OUTPUT: [(1, 'John', 'Doe'), (2, 'Jane', 'Doe')]
```

Leggi PostgreSQL online: https://riptutorial.com/it/python/topic/3374/postgresql

Capitolo 145: Precedenza dell'operatore

introduzione

Gli operatori Python hanno un **ordine di precedenza** impostato, che determina quali operatori vengono valutati per primi in un'espressione potenzialmente ambigua. Ad esempio, nell'espressione 3 * 2 + 7, il primo 3 viene moltiplicato per 2, e quindi il risultato viene aggiunto a 7, ottenendo 13. L'espressione non viene valutata al contrario, perché * ha una precedenza più alta di +.

Di seguito è riportato un elenco di operatori in base alla precedenza e una breve descrizione di ciò che fanno (di solito).

Osservazioni

Dalla documentazione di Python:

> La tabella seguente riepiloga le precedenze dell'operatore in Python, dalla precedenza più bassa (meno vincolante) alla precedenza più alta (la maggior parte del binding). Gli operatori nella stessa casella hanno la stessa precedenza. A meno che la sintassi non sia esplicitamente data, gli operatori sono binari. Gli operatori nello stesso gruppo si spostano da sinistra a destra (eccetto per i confronti, inclusi i test, che hanno tutti la stessa precedenza e catena da sinistra a destra e l'esponenziazione, quali gruppi da destra a sinistra).

Operatore	Descrizione
lambda	Espressione di Lambda
se altro	Espressione condizionale
o	Booleano OR
e	AND booleano
non x	NON booleano
in, non in, is, is not, <, <=,>,> =, <>,! =, ==	Confronti, inclusi test di appartenenza e test di identità
|	Bitwise OR
^	XOR bit a bit
&	Bitwise AND
<<, >>	Turni

Operatore	Descrizione
+, -	Addizione e sottrazione
*, /, //,%	Moltiplicazione, divisione, resto [8]
+ x, -x, ~ x	Positivo, negativo, bit negativo NON
**	Esponenziazione [9]
x [indice], x [indice: indice], x (argomenti ...), x.attributo	Sottoscrizione, slicing, chiamata, riferimento attributo
(espressioni ...), [espressioni ...], {chiave: valore ...}, espressioni ...	Visualizzazione di binding o tupla, visualizzazione elenco, visualizzazione dizionario, conversione stringhe

Examples

Esempi di precedenza di operatore semplice in python.

Python segue la regola PEMDAS. PEMDAS sta per Parentesi, Esponenti, Moltiplicazione e Divisione, Addizione e Sottrazione.

Esempio:

```
>>> a, b, c, d = 2, 3, 5, 7
>>> a ** (b + c)  # parentheses
256
>>> a * b ** c  # exponent: same as `a * (b ** c)`
7776
>>> a + b * c / d  # multiplication / division: same as `a + (b * c / d)`
4.142857142857142
```

Extra: le regole matematiche sono valide, ma non sempre :

```
>>> 300 / 300 * 200
200.0
>>> 300 * 200 / 300
200.0
>>> 1e300 / 1e300 * 1e200
1e+200
>>> 1e300 * 1e200 / 1e300
inf
```

Leggi Precedenza dell'operatore online: https://riptutorial.com/it/python/topic/5040/precedenza-dell-operatore

Capitolo 146: Processi e discussioni

introduzione

La maggior parte dei programmi viene eseguita riga per riga, eseguendo solo un singolo processo alla volta. I thread consentono a più processi di fluire indipendentemente l'uno dall'altro. Il threading con più processori consente ai programmi di eseguire più processi contemporaneamente. Questo argomento documenta l'implementazione e l'uso dei thread in Python.

Examples

Global Interpreter Lock

Le prestazioni di multithreading di Python possono spesso soffrire a causa del **Global Interpreter Lock** . In breve, anche se è possibile avere più thread in un programma Python, solo un'istruzione bytecode può essere eseguita in parallelo in qualsiasi momento, indipendentemente dal numero di CPU.

Di conseguenza, il multithreading nei casi in cui le operazioni sono bloccate da eventi esterni, come l'accesso alla rete, può essere piuttosto efficace:

```
import threading
import time


def process():
    time.sleep(2)


start = time.time()
process()
print("One run took %.2fs" % (time.time() - start))


start = time.time()
threads = [threading.Thread(target=process) for _ in range(4)]
for t in threads:
    t.start()
for t in threads:
    t.join()
print("Four runs took %.2fs" % (time.time() - start))

# Out: One run took 2.00s
# Out: Four runs took 2.00s
```

Si noti che anche se ogni `process` impiegava 2 secondi per essere eseguito, i quattro processi insieme erano in grado di funzionare in modo efficace in parallelo, prendendo 2 secondi in totale.

Tuttavia, il multithreading nei casi in cui i calcoli intensivi vengono eseguiti in codice Python, come

un sacco di calcoli, non comporta molti miglioramenti e può anche essere più lento rispetto all'esecuzione in parallelo:

```
import threading
import time


def somefunc(i):
    return i * i

def otherfunc(m, i):
    return m + i

def process():
    for j in range(100):
        result = 0
        for i in range(100000):
            result = otherfunc(result, somefunc(i))


start = time.time()
process()
print("One run took %.2fs" % (time.time() - start))


start = time.time()
threads = [threading.Thread(target=process) for _ in range(4)]
for t in threads:
    t.start()
for t in threads:
    t.join()
print("Four runs took %.2fs" % (time.time() - start))

# Out: One run took 2.05s
# Out: Four runs took 14.42s
```

In quest'ultimo caso, il multiprocessing può essere efficace in quanto più processi possono, ovviamente, eseguire più istruzioni contemporaneamente:

```
import multiprocessing
import time


def somefunc(i):
    return i * i

def otherfunc(m, i):
    return m + i

def process():
    for j in range(100):
        result = 0
        for i in range(100000):
            result = otherfunc(result, somefunc(i))


start = time.time()
process()
print("One run took %.2fs" % (time.time() - start))
```

```
start = time.time()
processes = [multiprocessing.Process(target=process) for _ in range(4)]
for p in processes:
    p.start()
for p in processes:
    p.join()
print("Four runs took %.2fs" % (time.time() - start))

# Out: One run took 2.07s
# Out: Four runs took 2.30s
```

Esecuzione in più thread

Usa `threading.Thread` per eseguire una funzione in un altro thread.

```
import threading
import os

def process():
    print("Pid is %s, thread id is %s" % (os.getpid(), threading.current_thread().name))

threads = [threading.Thread(target=process) for _ in range(4)]
for t in threads:
    t.start()
for t in threads:
    t.join()

# Out: Pid is 11240, thread id is Thread-1
# Out: Pid is 11240, thread id is Thread-2
# Out: Pid is 11240, thread id is Thread-3
# Out: Pid is 11240, thread id is Thread-4
```

Esecuzione in più processi

Usa `multiprocessing.Process` per eseguire una funzione in un altro processo. L'interfaccia è simile alla `threading.Thread` :

```
import multiprocessing
import os

def process():
    print("Pid is %s" % (os.getpid(),))

processes = [multiprocessing.Process(target=process) for _ in range(4)]
for p in processes:
    p.start()
for p in processes:
    p.join()

# Out: Pid is 11206
# Out: Pid is 11207
# Out: Pid is 11208
# Out: Pid is 11209
```

Condivisione dello stato tra thread

Poiché tutti i thread sono in esecuzione nello stesso processo, tutti i thread hanno accesso agli stessi dati.

Tuttavia, l'accesso concorrente ai dati condivisi deve essere protetto con un blocco per evitare problemi di sincronizzazione.

```
import threading

obj = {}
obj_lock = threading.Lock()

def objify(key, val):
    print("Obj has %d values" % len(obj))
    with obj_lock:
        obj[key] = val
    print("Obj now has %d values" % len(obj))

ts = [threading.Thread(target=objify, args=(str(n), n)) for n in range(4)]
for t in ts:
    t.start()
for t in ts:
    t.join()
print("Obj final result:")
import pprint; pprint.pprint(obj)

# Out: Obj has 0 values
# Out:  Obj has 0 values
# Out: Obj now has 1 values
# Out: Obj now has 2 valuesObj has 2 values
# Out: Obj now has 3 values
# Out:
# Out:  Obj has 3 values
# Out: Obj now has 4 values
# Out: Obj final result:
# Out: {'0': 0, '1': 1, '2': 2, '3': 3}
```

Condivisione dello stato tra processi

Il codice in esecuzione in processi diversi non condivide, per impostazione predefinita, gli stessi dati. Tuttavia, il modulo `multiprocessing` contiene le primitive per aiutare a condividere i valori tra più processi.

```
import multiprocessing

plain_num = 0
shared_num = multiprocessing.Value('d', 0)
lock = multiprocessing.Lock()

def increment():
    global plain_num
    with lock:
        # ordinary variable modifications are not visible across processes
        plain_num += 1
        # multiprocessing.Value modifications are
```

```
        shared_num.value += 1

ps = [multiprocessing.Process(target=increment) for n in range(4)]
for p in ps:
    p.start()
for p in ps:
    p.join()

print("plain_num is %d, shared_num is %d" % (plain_num, shared_num.value))

# Out: plain_num is 0, shared_num is 4
```

Leggi Processi e discussioni online: https://riptutorial.com/it/python/topic/4110/processi-e-discussioni

Capitolo 147: profiling

Examples

%% timeit e% timeit in IPython

Concatenamento di stringhe di profilazione:

```
In [1]: import string

In [2]: %%timeit s=""; long_list=list(string.ascii_letters)*50
  ....: for substring in long_list:
  ....:   s+=substring
  ....:
1000 loops, best of 3: 570 us per loop

In [3]: %%timeit long_list=list(string.ascii_letters)*50
  ....: s="".join(long_list)
  ....:
100000 loops, best of 3: 16.1 us per loop
```

Loop di profilatura su iterables ed elenchi:

```
In [4]: %timeit for i in range(100000):pass
100 loops, best of 3: 2.82 ms per loop

In [5]: %timeit for i in list(range(100000)):pass
100 loops, best of 3: 3.95 ms per loop
```

funzione timeit ()

Ripetizione del profilo degli elementi in un array

```
>>> import timeit
>>> timeit.timeit('list(itertools.repeat("a", 100))', 'import itertools', number = 10000000)
10.997665435877963
>>> timeit.timeit('["a"]*100', number = 10000000)
7.118789926862576
```

riga di comando timeit

Conciliazione dei numeri

```
python -m timeit "'-'.join(str(n) for n in range(100))"
10000 loops, best of 3: 29.2 usec per loop

python -m timeit "'-'.join(map(str,range(100)))"
100000 loops, best of 3: 19.4 usec per loop
```

line_profiler nella riga di comando

Il codice sorgente con la direttiva @profile prima della funzione che vogliamo profilare:

```
import requests

@profile
def slow_func():
    s = requests.session()
    html=s.get("https://en.wikipedia.org/").text
    sum([pow(ord(x),3.1) for x in list(html)])

for i in range(50):
    slow_func()
```

Usando il comando kernprof per calcolare il profilo riga per riga

```
$ kernprof -lv so6.py

Wrote profile results to so6.py.lprof
Timer unit: 4.27654e-07 s

Total time: 22.6427 s
File: so6.py
Function: slow_func at line 4

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     4                                           @profile
     5                                           def slow_func():
     6        50        20729    414.6      0.0      s = requests.session()
     7        50     47618627 952372.5     89.9
html=s.get("https://en.wikipedia.org/").text
     8        50      5306958 106139.2     10.0      sum([pow(ord(x),3.1) for x in
list(html)])
```

La richiesta di pagina è quasi sempre più lenta di qualsiasi calcolo basato sulle informazioni sulla pagina.

Utilizzo di cProfile (profilo preferito)

Python include un profiler chiamato cProfile. Questo è generalmente preferito rispetto al timeit.

Spezza l'intero script e per ogni metodo nel tuo script ti dice:

- `ncalls` : il numero di volte in cui è stato chiamato un metodo
- `tottime` : tempo totale trascorso nella funzione data (escluso il tempo trascorso nelle chiamate alle sotto-funzioni)
- `percall` : tempo trascorso per chiamata. O il quoziente di tempo diviso per ncalls
- `cumtime` : il tempo cumulativo trascorso in questa e tutte le sottofunzioni (dall'invocazione all'uscita). Questa cifra è accurata anche per le funzioni ricorsive.
- `percall` : è il quoziente di cumtime diviso per le chiamate primitive
- `filename:lineno(function)` : fornisce i rispettivi dati di ciascuna funzione

Il cProfiler può essere facilmente richiamato su Command Line usando:

```
$ python -m cProfile main.py
```

Per ordinare l'elenco restituito dei metodi profilati in base al tempo impiegato nel metodo:

```
$ python -m cProfile -s time main.py
```

Leggi profiling online: https://riptutorial.com/it/python/topic/3818/profiling

Capitolo 148: Programmazione funzionale in Python

introduzione

La programmazione funzionale decompone un problema in un insieme di funzioni. Idealmente, le funzioni prendono solo input e producono output e non hanno nessuno stato interno che influenzi l'output prodotto per un dato input. Di seguito sono riportate le tecniche funzionali comuni a molte lingue: come lambda, map, reduce.

Examples

Funzione Lambda

Una funzione anonima e in linea definita con lambda. I parametri del lambda sono definiti a sinistra del colon. Il corpo della funzione è definito a destra del colon. Il risultato dell'esecuzione del corpo della funzione è (implicitamente) restituito.

```
s=lambda x:x*x
s(2)     =>4
```

Funzione mappa

Mappa prende una funzione e una collezione di oggetti. Crea una nuova raccolta vuota, esegue la funzione su ciascun elemento della raccolta originale e inserisce ciascun valore restituito nella nuova raccolta. Restituisce la nuova collezione.

Questa è una mappa semplice che prende un elenco di nomi e restituisce un elenco delle lunghezze di quei nomi:

```
name_lengths = map(len, ["Mary", "Isla", "Sam"])
print(name_lengths)     =>[4, 4, 3]
```

Ridurre la funzione

Ridurre prende una funzione e una collezione di oggetti. Restituisce un valore che viene creato combinando gli elementi.

Questa è una semplice riduzione. Restituisce la somma di tutti gli articoli nella collezione.

```
total = reduce(lambda a, x: a + x, [0, 1, 2, 3, 4])
print(total)     =>10
```

Funzione filtro

Il filtro ha una funzione e una collezione. Restituisce una collezione di ogni oggetto per il quale la funzione ha restituito True.

```
arr=[1,2,3,4,5,6]
[i for i in filter(lambda x:x>4,arr)]    # outputs[5,6]
```

Leggi Programmazione funzionale in Python online:
https://riptutorial.com/it/python/topic/9552/programmazione-funzionale-in-python

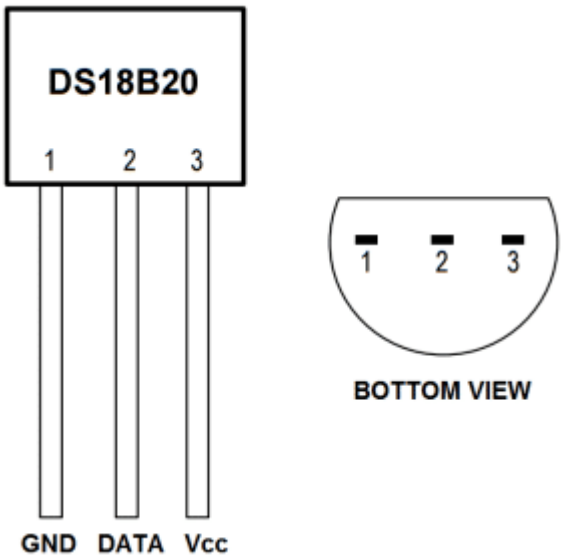
Capitolo 149: Programmazione IoT con Python e Raspberry PI

Examples

Esempio: sensore di temperatura

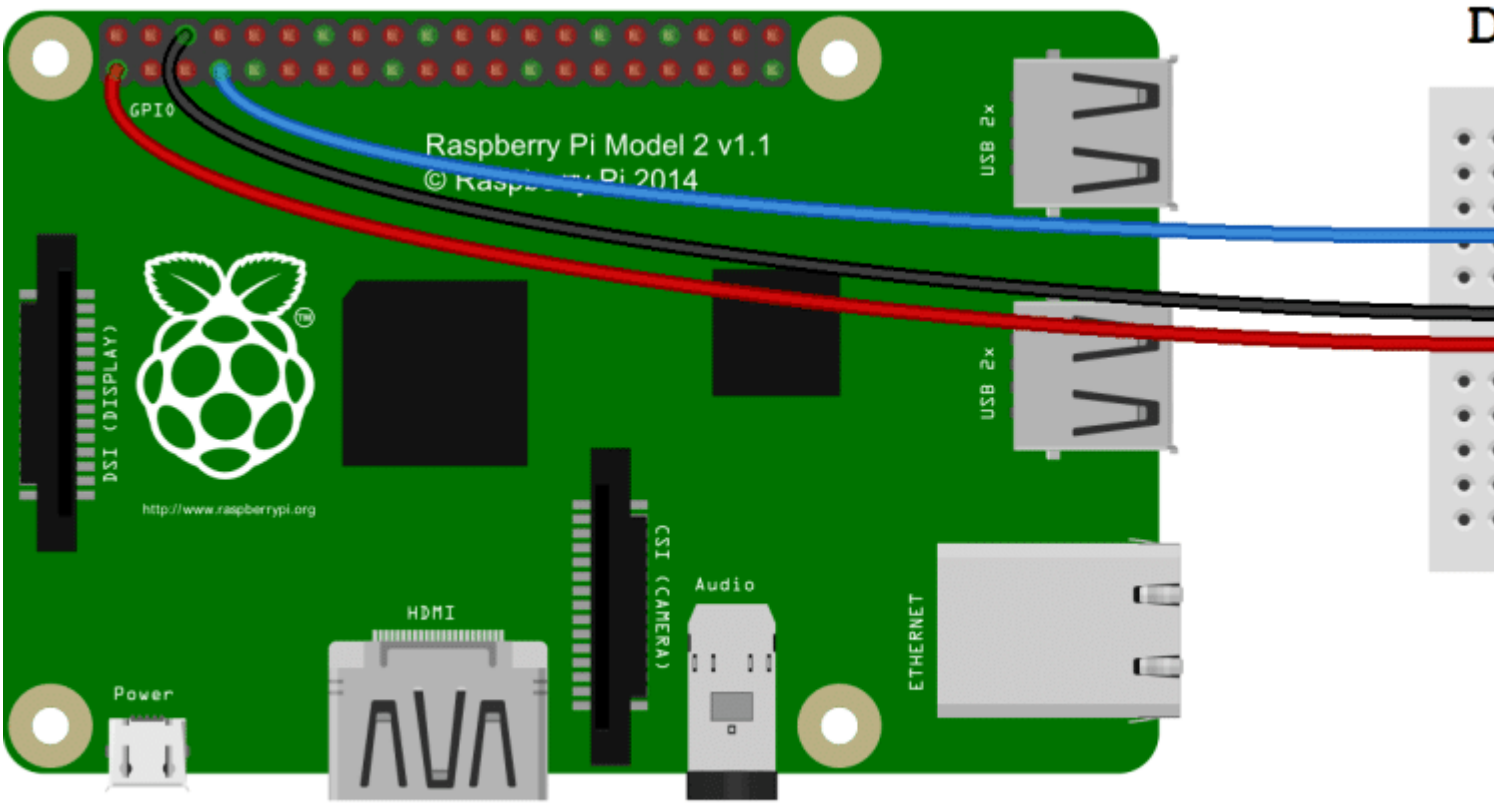
Interfacciamento di DS18B20 con Raspberry pi

Connessione di DS18B20 con Raspberry pi



Puoi vedere che ci sono tre terminali

1. Vcc
2. Gnd
3. Dati (protocollo a un filo)



R1 ha una resistenza di 4,7 k ohm per alzare il livello di tensione

1. **Vcc** dovrebbe essere collegato a uno dei pin 5v o 3.3v di Raspberry pi (PIN: 01, 02, 04, 17).
2. **Gnd** dovrebbe essere collegato a uno dei pin Gnd di Raspberry pi (PIN: 06, 09, 14, 20, 25).
3. **DATI** devono essere collegati a (PIN: 07)

Abilitazione dell'interfaccia a un filo dal lato RPi

4. Accedi a Raspberry pi usando stucco o qualsiasi altro terminale linux / unix.
5. Dopo l'accesso, apri il file /boot/config.txt nel tuo browser preferito.

   nano /boot/config.txt
6. Ora aggiungi la riga `dtoverlay=w1-gpio` alla fine del file.
7. Ora riavviare il riavvio di Raspberry pi `sudo reboot` .
8. Accedi a Raspberry pi ed esegui `sudo modprobe g1-gpio`
9. Quindi eseguire `sudo modprobe w1-therm`
10. Ora vai nella directory / sys / bus / w1 / devices `cd /sys/bus/w1/devices`
11. Ora scoprirai una directory virtuale creata dal tuo sensore di temperatura a partire da 28 - ********.

12. Vai a questa directory `cd 28-********`

13. Ora c'è un nome di file **w1-slave** , questo file contiene la temperatura e altre informazioni come CRC. `cat w1-slave` .

Ora scrivi un modulo in python per leggere la temperatura

```
import glob
import time

RATE = 30
sensor_dirs = glob.glob("/sys/bus/w1/devices/28*")

if len(sensor_dirs) != 0:
    while True:
        time.sleep(RATE)
        for directories in sensor_dirs:
            temperature_file = open(directories + "/w1_slave")
            # Reading the files
            text = temperature_file.read()
            temperature_file.close()
            # Split the text with new lines (\n) and select the second line.
            second_line = text.split("\n")[1]
            # Split the line into words, and select the 10th word
            temperature_data = second_line.split(" ")[9]
            # We will read after ignoring first two character.
            temperature = float(temperature_data[2:])
            # Now normalise the temperature by dividing 1000.
            temperature = temperature / 1000
            print 'Address : '+str(directories.split('/')[-1])+', Temperature :
'+str(temperature)
```

Sopra il modulo python stamperà la temperatura rispetto all'indirizzo per un tempo infinito. Il parametro RATE è definito per modificare o regolare la frequenza della richiesta di temperatura dal sensore.

Diagramma pin GPIO

1. [https://www.element14.com/community/servlet/JiveServlet/previewBody/73950-102-11-339300/pi3_gpio.png][3]

Leggi Programmazione IoT con Python e Raspberry PI online:
https://riptutorial.com/it/python/topic/10735/programmazione-iot-con-python-e-raspberry-pi

Capitolo 150: py.test

Examples

Impostazione di py.test

`py.test` è una delle numerose librerie di test di terze parti disponibili per Python. Può essere installato usando `pip` con

```
pip install pytest
```

Il codice da testare

Diciamo che stiamo testando una funzione di addizione in `projectroot/module/code.py` :

```
# projectroot/module/code.py
def add(a, b):
    return a + b
```

Il codice di prova

Creiamo un file di test in `projectroot/tests/test_code.py` . Il file **deve iniziare con `test_`** per essere riconosciuto come un file di test.

```
# projectroot/tests/test_code.py
from module import code


def test_add():
    assert code.add(1, 2) == 3
```

Esecuzione del test

Da `projectroot` eseguiamo semplicemente `py.test` :

```
# ensure we have the modules
$ touch tests/__init__.py
$ touch module/__init__.py
$ py.test
============================================== test session starts
=============================================
platform darwin -- Python 2.7.10, pytest-2.9.2, py-1.4.31, pluggy-0.3.1
rootdir: /projectroot, inifile:
collected 1 items

tests/test_code.py .
```

```
================================================ 1 passed in 0.01 seconds
================================================
```

Test falliti

Un test non funzionante fornirà un utile risultato su ciò che è andato storto:

```
# projectroot/tests/test_code.py
from module import code


def test_add__failing():
    assert code.add(10, 11) == 33
```

risultati:

```
$ py.test
================================================== test session starts
==================================================
platform darwin -- Python 2.7.10, pytest-2.9.2, py-1.4.31, pluggy-0.3.1
rootdir: /projectroot, inifile:
collected 1 items

tests/test_code.py F

======================================================== FAILURES
========================================================
_____________________________________________________ test_add__failing
_____________________________________________________

    def test_add__failing():
>       assert code.add(10, 11) == 33
E       assert 21 == 33
E        +  where 21 = <function add at 0x105d4d6e0>(10, 11)
E        +    where <function add at 0x105d4d6e0> = code.add

tests/test_code.py:5: AssertionError
================================================ 1 failed in 0.01 seconds
================================================
```

Introduzione alle partite di prova

Talvolta i test più complicati devono essere impostati prima di eseguire il codice che si desidera testare. È possibile farlo nella stessa funzione di test, ma poi si finisce con le funzioni di test di grandi dimensioni facendo così tanto che è difficile dire dove si ferma l'installazione e inizia il test. Puoi anche ottenere un sacco di codice di duplicazione tra le varie funzioni di test.

Il nostro file di codice:

```
# projectroot/module/stuff.py
class Stuff(object):
    def prep(self):
        self.foo = 1
        self.bar = 2
```

Il nostro file di test:

```
# projectroot/tests/test_stuff.py
import pytest
from module import stuff


def test_foo_updates():
    my_stuff = stuff.Stuff()
    my_stuff.prep()
    assert 1 == my_stuff.foo
    my_stuff.foo = 30000
    assert my_stuff.foo == 30000


def test_bar_updates():
    my_stuff = stuff.Stuff()
    my_stuff.prep()
    assert 2 == my_stuff.bar
    my_stuff.bar = 42
    assert 42 == my_stuff.bar
```

Questi sono esempi piuttosto semplici, ma se il nostro oggetto `Stuff` aveva bisogno di molte più impostazioni, sarebbe ingombrante. Vediamo che c'è un codice duplicato tra i nostri casi di test, quindi cerchiamo di farlo prima in una funzione separata.

```
# projectroot/tests/test_stuff.py
import pytest
from module import stuff


def get_prepped_stuff():
    my_stuff = stuff.Stuff()
    my_stuff.prep()
    return my_stuff


def test_foo_updates():
    my_stuff = get_prepped_stuff()
    assert 1 == my_stuff.foo
    my_stuff.foo = 30000
    assert my_stuff.foo == 30000


def test_bar_updates():
    my_stuff = get_prepped_stuff()
    assert 2 == my_stuff.bar
    my_stuff.bar = 42
    assert 42 == my_stuff.bar
```

Questo sembra migliore, ma abbiamo ancora la `my_stuff = get_prepped_stuff()` che ingombra le nostre funzioni di test.

py.test infissi in soccorso!

Le fixture sono versioni molto più potenti e flessibili delle funzioni di setup del test. Possono fare

molto più di quanto stiamo facendo leva qui, ma lo faremo un passo alla volta.

Per prima cosa cambiamo `get_prepped_stuff` in una fixture chiamata `prepped_stuff` . Vuoi nominare i tuoi dispositivi con nomi piuttosto che verbi a causa del modo in cui i proiettori finiranno per essere usati nelle funzioni di test più avanti. La `@pytest.fixture` indica che questa funzione specifica dovrebbe essere gestita come un dispositivo `@pytest.fixture` piuttosto che una funzione regolare.

```
@pytest.fixture
def prepped_stuff():
    my_stuff = stuff.Stuff()
    my_stuff.prep()
    return my_stuff
```

Ora dovremmo aggiornare le funzioni di test in modo che utilizzino il dispositivo. Questo viene fatto aggiungendo un parametro alla loro definizione che corrisponde esattamente al nome della fixture. Quando py.test viene eseguito, eseguirà la fixture prima di eseguire il test, quindi passerà il valore restituito dalla fixture alla funzione di test attraverso quel parametro. (Si noti che le fixture non hanno **bisogno** di restituire un valore, ma possono fare altre cose di setup, come chiamare una risorsa esterna, organizzare le cose sul filesystem, mettere i valori in un database, qualunque sia il test necessario per l'installazione)

```
def test_foo_updates(prepped_stuff):
    my_stuff = prepped_stuff
    assert 1 == my_stuff.foo
    my_stuff.foo = 30000
    assert my_stuff.foo == 30000


def test_bar_updates(prepped_stuff):
    my_stuff = prepped_stuff
    assert 2 == my_stuff.bar
    my_stuff.bar = 42
    assert 42 == my_stuff.bar
```

Ora puoi capire perché l'abbiamo chiamato con un nome. ma la linea `my_stuff = prepped_stuff` è praticamente inutile, quindi usiamo invece `prepped_stuff` direttamente.

```
def test_foo_updates(prepped_stuff):
    assert 1 == prepped_stuff.foo
    prepped_stuff.foo = 30000
    assert prepped_stuff.foo == 30000


def test_bar_updates(prepped_stuff):
    assert 2 == prepped_stuff.bar
    prepped_stuff.bar = 42
    assert 42 == prepped_stuff.bar
```

Ora stiamo usando i dispositivi! Possiamo andare oltre cambiando l'ambito della fixture (quindi viene eseguito solo una volta per modulo di test o sessione di esecuzione della suite di test invece di una volta per funzione di test), costruendo dispositivi che utilizzano altri dispositivi,

parametrizzando l'apparecchiatura (in modo che l'apparecchiatura e tutti i test che utilizzano quel dispositivo vengono eseguiti più volte, una volta per ogni parametro assegnato all'apparecchio), i dispositivi che leggono i valori dal modulo che li chiama ... come accennato in precedenza, gli apparecchi hanno molta più potenza e flessibilità di una normale funzione di configurazione.

Pulizia dopo i test.

Diciamo che il nostro codice è cresciuto e il nostro oggetto Stuff ora ha bisogno di una pulizia speciale.

```
# projectroot/module/stuff.py
class Stuff(object):
def prep(self):
    self.foo = 1
    self.bar = 2

def finish(self):
    self.foo = 0
    self.bar = 0
```

Potremmo aggiungere del codice per chiamare la pulizia nella parte inferiore di ogni funzione di test, ma i dispositivi forniscono un modo migliore per farlo. Se si aggiunge una funzione all'apparecchio e lo si registra come **finalizzatore** , il codice nella funzione finalizzatore verrà chiamato dopo che il test con l'apparecchiatura è stato eseguito. Se l'ambito della fixture è più grande di una singola funzione (come il modulo o la sessione), il finalizzatore verrà eseguito dopo che tutti i test in ambito sono stati completati, quindi dopo il completamento del modulo o alla fine dell'intera sessione di test in esecuzione .

```
@pytest.fixture
def prepped_stuff(request):  # we need to pass in the request to use finalizers
    my_stuff = stuff.Stuff()
    my_stuff.prep()
    def fin():  # finalizer function
        # do all the cleanup here
        my_stuff.finish()
    request.addfinalizer(fin)  # register fin() as a finalizer
    # you can do more setup here if you really want to
    return my_stuff
```

Usare la funzione finalizzatore all'interno di una funzione può essere un po 'difficile da capire a prima vista, specialmente quando si hanno proiettori più complicati. Puoi invece usare un **dispositivo di rendimento** per fare la stessa cosa con un flusso di esecuzione più leggibile. L'unica vera differenza è che invece di usare `return` usiamo un `yield` nella parte della fixture in cui viene eseguita la configurazione e il controllo dovrebbe andare a una funzione di test, quindi aggiungere tutto il codice cleanup dopo la `yield` . Lo decoriamo anche come `yield_fixture` modo che py.test sappia come gestirlo.

```
@pytest.yield_fixture
def prepped_stuff():  # it doesn't need request now!
    # do setup
    my_stuff = stuff.Stuff()
```

```
    my_stuff.prep()
    # setup is done, pass control to the test functions
    yield my_stuff
    # do cleanup
    my_stuff.finish()
```

E questo conclude l'Intro to Test Fixtures!

Per ulteriori informazioni, consultare la documentazione ufficiale su py.test e la documentazione ufficiale sulla resa

Leggi py.test online: https://riptutorial.com/it/python/topic/7054/py-test

Capitolo 151: pyaudio

introduzione

PyAudio fornisce collegamenti Python per PortAudio, la libreria I / O audio multipiattaforma. Con PyAudio, puoi facilmente usare Python per riprodurre e registrare l'audio su una varietà di piattaforme. PyAudio è ispirato da:

1.pyPortAudio / fastaudio: collegamenti Python per l'API PortAudio v18.

2.tkSnack: toolkit audio cross-platform per Tcl / Tk e Python.

Osservazioni

Nota: stream_callback è chiamato in un thread separato (dal thread principale). Le eccezioni che si verificano nello stream_callback:
1 .stampare un traceback sull'errore standard per facilitare il debug,
2 .queue l'eccezione da lanciare (ad un certo punto) nel thread principale, e
3 .return paAbort su PortAudio per interrompere lo streaming.
Nota: non chiamare Stream.read () o Stream.write () se si utilizza l'operazione non bloccante.
Vedi: Firma callback di PortAudio per ulteriori dettagli:
http://portaudio.com/docs/v19-doxydocs/portaudio_8h.html#a8a60fb2a5ec9cbade3f54a9c978e2710

Examples

Modalità di callback audio I / O

```
"""PyAudio Example: Play a wave file (callback version)."""

import pyaudio
import wave
import time
import sys

if len(sys.argv) < 2:
    print("Plays a wave file.\n\nUsage: %s filename.wav" % sys.argv[0])
    sys.exit(-1)

wf = wave.open(sys.argv[1], 'rb')

# instantiate PyAudio (1)
p = pyaudio.PyAudio()

# define callback (2)
def callback(in_data, frame_count, time_info, status):
    data = wf.readframes(frame_count)
    return (data, pyaudio.paContinue)
```

```
# open stream using callback (3)
stream = p.open(format=p.get_format_from_width(wf.getsampwidth()),
                channels=wf.getnchannels(),
                rate=wf.getframerate(),
                output=True,
                stream_callback=callback)

# start the stream (4)
stream.start_stream()

# wait for stream to finish (5)
while stream.is_active():
    time.sleep(0.1)

# stop stream (6)
stream.stop_stream()
stream.close()
wf.close()

# close PyAudio (7)
p.terminate()
```

In modalità callback, PyAudio chiamerà una funzione di callback specificata (2) ogni volta che ha bisogno di nuovi dati audio (da riprodurre) e / o quando sono disponibili nuovi dati audio (registrati). Si noti che PyAudio chiama la funzione di callback in un thread separato. La funzione ha il seguente `callback(<input_data>, <frame_count>, <time_info>, <status_flag>)` firma `callback(<input_data>, <frame_count>, <time_info>, <status_flag>)` e deve restituire una tupla contenente `frame_count` frame di dati audio e un flag che indica se ci sono più frame da riprodurre / registrare.

Inizia l'elaborazione del flusso audio usando **pyaudio.Stream.start_stream ()** (4), che chiamerà ripetutamente la funzione di callback fino a quando quella funzione non restituirà **pyaudio.paComplete** .

Per mantenere attivo il flusso, il thread principale non deve terminare, ad es. Dormendo (5).

Modalità I / O audio blocco

"" "Esempio di PyAudio: riproduzione di un file wave." ""

```
import pyaudio
import wave
import sys

CHUNK = 1024

if len(sys.argv) < 2:
    print("Plays a wave file.\n\nUsage: %s filename.wav" % sys.argv[0])
    sys.exit(-1)

wf = wave.open(sys.argv[1], 'rb')

# instantiate PyAudio (1)
p = pyaudio.PyAudio()
```

```
# open stream (2)
stream = p.open(format=p.get_format_from_width(wf.getsampwidth()),
                channels=wf.getnchannels(),
                rate=wf.getframerate(),
                output=True)

# read data
data = wf.readframes(CHUNK)

# play stream (3)
while len(data) > 0:
    stream.write(data)
    data = wf.readframes(CHUNK)

# stop stream (4)
stream.stop_stream()
stream.close()

# close PyAudio (5)
p.terminate()
```

Per usare PyAudio, prima istanzia PyAudio usando **pyaudio.PyAudio ()** (1), che configura il sistema portaudio.

Per registrare o riprodurre audio, apri uno stream sul dispositivo desiderato con i parametri audio desiderati usando **pyaudio.PyAudio.open ()** (2). Questo imposta un **pyaudio.Stream** per riprodurre o registrare l'audio.

Riproduci l'audio scrivendo i dati audio nello stream utilizzando **pyaudio.Stream.write ()** o leggi i dati audio dallo stream utilizzando **pyaudio.Stream.read ()** . (3)

Nota che in " *modalità di blocco* ", ogni **blocco pyaudio.Stream.write ()** o **pyaudio.Stream.read () si** blocca fino a quando tutti i fotogrammi dati / richiesti sono stati riprodotti / registrati. In alternativa, per generare dati audio al volo o elaborare immediatamente i dati audio registrati, utilizzare la "modalità di richiamata" (*fare riferimento all'esempio sulla modalità di richiamata*)

Utilizzare pyaudio.Stream.stop_stream () per sospendere la riproduzione / registrazione e **pyaudio.Stream.close ()** per terminare lo streaming. (4)

Infine, termina la sessione **portaudio** usando **pyaudio.PyAudio.terminate ()** (5)

Leggi pyaudio online: https://riptutorial.com/it/python/topic/10627/pyaudio

Capitolo 152: pygame

introduzione

Pygame è la libreria di riferimento per realizzare applicazioni multimediali, in particolare giochi, in Python. Il sito web ufficiale è http://www.pygame.org/ .

Sintassi

- pygame.mixer.init (frequenza = 22050, dimensione = -16, canali = 2, buffer = 4096)
- pygame.mixer.pre_init (frequenza, dimensione, canali, buffer)
- pygame.mixer.quit ()
- pygame.mixer.get_init ()
- pygame.mixer.stop ()
- pygame.mixer.pause ()
- pygame.mixer.unpause ()
- pygame.mixer.fadeout (tempo)
- pygame.mixer.set_num_channels (conteggio)
- pygame.mixer.get_num_channels ()
- pygame.mixer.set_reserved (conteggio)
- pygame.mixer.find_channel (forza)
- pygame.mixer.get_busy ()

Parametri

Parametro	Dettagli
contare	Un numero intero positivo che rappresenta qualcosa come il numero di canali che devono essere riservati.
vigore	Un valore booleano (`False` o `True`) che determina se `find_channel()` deve restituire un canale (inattivo o non) con `True` o meno (se non ci sono canali inattivi) con `False`

Examples

Installare pygame

Con `pip` :

```
pip install pygame
```

Con `conda` :

```
conda install -c tlatorre pygame=1.9.2
```

Download diretto dal sito Web: http://www.pygame.org/download.shtml

È possibile trovare gli installatori adatti per Windows e altri sistemi operativi.

I progetti possono anche essere trovati su http://www.pygame.org/

Il modulo mixer di Pygame

Il modulo `pygame.mixer` aiuta a controllare la musica usata nei programmi `pygame` . A partire da ora, ci sono 15 diverse funzioni per il modulo del `mixer` .

L'inizializzazione

Analogamente a come devi inizializzare `pygame` con `pygame.init()` , devi anche inizializzare `pygame.mixer` .

Usando la prima opzione, inizializziamo il modulo usando i valori predefiniti. Tuttavia, è possibile sovrascrivere queste opzioni predefinite. Usando la seconda opzione, possiamo inizializzare il modulo usando i valori che inseriamo manualmente in noi stessi. Valori standard:

```
pygame.mixer.init(frequency=22050, size=-16, channels=2, buffer=4096)
```

Per verificare se l'abbiamo inizializzato o no, possiamo usare `pygame.mixer.get_init()` , che restituisce `True` se lo è e `False` se non lo è. Per uscire / annullare l'inizializzazione, usa semplicemente `pygame.mixer.quit()` . Se si desidera continuare a riprodurre suoni con il modulo, potrebbe essere necessario reinizializzare il modulo.

Possibili azioni

Mentre suona il tuo suono, puoi metterlo temporaneamente in pausa con `pygame.mixer.pause()` . Per riprendere a suonare i tuoi suoni, usa semplicemente `pygame.mixer.unpause()` . Puoi anche eseguire il fadeout della fine del suono usando `pygame.mixer.fadeout()` . Ci vuole un argomento, che è il numero di millisecondi che ci vuole per completare la dissolvenza della musica.

canali

È possibile riprodurre il maggior numero di brani necessari a condizione che ci siano abbastanza canali aperti per supportarli. Per impostazione predefinita, ci sono 8 canali. Per cambiare il numero di canali che ci sono, usa `pygame.mixer.set_num_channels()` . L'argomento è un numero intero non negativo. Se il numero di canali diminuisce, qualsiasi suono riprodotto sui canali rimossi si interromperà immediatamente.

Per sapere quanti canali sono attualmente in uso, chiama `pygame.mixer.get_channels(count)` . L'output è il numero di canali che non sono attualmente aperti. Puoi anche prenotare canali per suoni che devono essere riprodotti usando `pygame.mixer.set_reserved(count)` . L'argomento è anche un numero intero non negativo. Qualsiasi suono riprodotto sui canali appena prenotati non verrà interrotto.

Puoi anche scoprire quale canale non viene utilizzato usando `pygame.mixer.find_channel(force)` . Il suo argomento è un bool: True o False. Se non ci sono canali che sono inattivi e la `force` è False, restituirà `None` . Se la `force` è vera, restituirà il canale che sta suonando da più tempo.

Leggi pygame online: https://riptutorial.com/it/python/topic/8761/pygame

Capitolo 153: pyglet

introduzione

Pyglet è un modulo Python utilizzato per immagini e suoni. Non ha dipendenze da altri moduli. Vedi [pyglet.org] [1] per le informazioni ufficiali. [1]: http://pyglet.org

Examples

Ciao mondo in Pyglet

```
import pyglet
window = pyglet.window.Window()
label = pyglet.text.Label('Hello, world',
                      font_name='Times New Roman',
                      font_size=36,
                      x=window.width//2, y=window.height//2,
                      anchor_x='center', anchor_y='center')
@window.event
def on_draw():
    window.clear()
    label.draw()
pyglet.app.run()
```

Installazione di Pyglet

Installa Python, vai nella riga di comando e digita:

Python 2:

```
pip install pyglet
```

Python 3:

```
pip3 install pyglet
```

Riproduzione audio in Pyglet

```
sound = pyglet.media.load(sound.wav)
sound.play()
```

Utilizzo di Pyglet per OpenGL

```
import pyglet
from pyglet.gl import *

win = pyglet.window.Window()
```

```
@win.event()
def on_draw():
    #OpenGL goes here. Use OpenGL as normal.

pyglet.app.run()
```

Punti di disegno usando Pyglet e OpenGL

```
import pyglet
from pyglet.gl import *

win = pyglet.window.Window()
glClear(GL_COLOR_BUFFER_BIT)

@win.event
def on_draw():
    glBegin(GL_POINTS)
    glVertex2f(x, y) #x is desired distance from left side of window, y is desired distance
from bottom of window
    #make as many vertexes as you want
    glEnd
```

Per connettere i punti, sostituire `GL_POINTS` con `GL_LINE_LOOP` .

Leggi pyglet online: https://riptutorial.com/it/python/topic/8208/pyglet

Capitolo 154: PyInstaller - Distribuzione del codice Python

Sintassi

- pyinstaller [opzioni] script [script ...] | specfile

Osservazioni

PyInstaller è un modulo utilizzato per raggruppare app python in un unico pacchetto insieme a tutte le dipendenze. L'utente può quindi eseguire l'app del pacchetto senza un interprete python o alcun modulo. Raggruppa correttamente molti pacchetti importanti come numpy, Django, OpenCv e altri.

Alcuni punti importanti da ricordare:

- Pyinstaller supporta Python 2.7 e Python 3.3+
- Pyinstaller è stato testato su Windows, Linux e Mac OS X.
- **NON** è un cross-compilatore. (Un'app di Windows non può essere impacchettata in Linux. Devi eseguire PyInstaller in Windows per raggruppare un'app per Windows)

Homepage Documenti ufficiali

Examples

Installazione e configurazione

Pyinstaller è un normale pacchetto python. Può essere installato usando pip:

```
pip install pyinstaller
```

Installazione in Windows
Per Windows, pywin32 o pypiwin32 è un prerequisito. Quest'ultimo viene installato automaticamente quando pyinstaller viene installato tramite pip.

Installazione in Mac OS X
PyInstaller funziona con il predefinito Python 2.7 fornito con Mac OS X corrente. Se si devono usare versioni successive di Python o se si devono usare pacchetti importanti come PyQT, Numpy, Matplotlib e simili, si consiglia di installarli usando MacPorts o Homebrew .

Installazione dall'archivio
Se pip non è disponibile, scaricare l'archivio compresso da PyPI .
Per testare la versione di sviluppo, scaricare l'archivio compresso dal ramo di *sviluppo* della pagina Download di PyInstaller .

Espandi l'archivio e trova lo script `setup.py` . Esegui `python setup.py install` con privilegio di amministratore per installare o aggiornare PyInstaller.

Verifica l'installazione
Il comando `pyinstaller` deve esistere sul percorso di sistema per tutte le piattaforme dopo un'installazione corretta.
Verificarlo digitando `pyinstaller --version` nella riga di comando. Questo stamperà la versione corrente di pyinstaller.

Utilizzando Pyinstaller

Nel caso d'uso più semplice, vai alla directory in cui si trova il tuo file e digita:

```
pyinstaller myfile.py
```

Pyinstaller analizza il file e crea:

- Un file **myfile.spec** nella stessa directory di `myfile.py`
- Una cartella di **compilazione** nella stessa directory di `myfile.py`
- Una cartella **dist** nella stessa directory di `myfile.py`
- Registrare i file nella cartella di **build**

L'app in bundle può essere trovata nella cartella **dist**

Opzioni
Esistono diverse opzioni che possono essere utilizzate con pyinstaller. Un elenco completo delle opzioni può essere trovato qui .

Una volta raggruppata, la tua app può essere avviata aprendo "dist \ myfile \ myfile.exe".

Raggruppamento in una cartella

Quando PyInstaller viene utilizzato senza opzioni per il bundle di `myscript.py` , l'output predefinito è una singola cartella (di nome `myscript`) contenente un eseguibile denominato `myscript` (`myscript.exe` in Windows) insieme a tutte le dipendenze necessarie.
L'app può essere distribuita comprimendo la cartella in un file zip.

Una modalità cartella può essere impostata in modo esplicito usando l'opzione `-D` o `--onedir`

```
pyinstaller myscript.py -D
```

vantaggi:

Uno dei principali vantaggi del raggruppamento in una singola cartella è che è più facile eseguire il debug dei problemi. Se qualche modulo non riesce a importare, può essere verificato ispezionando la cartella.
Un altro vantaggio si sente durante gli aggiornamenti. Se ci sono alcune modifiche nel codice ma le dipendenze utilizzate sono *esattamente* le stesse, i distributori possono semplicemente spedire

il file eseguibile (che è in genere più piccolo dell'intera cartella).

svantaggi

L'unico svantaggio di questo metodo è che gli utenti devono cercare l'eseguibile tra un numero elevato di file.
Inoltre, gli utenti possono cancellare / modificare altri file che potrebbero impedire alla app di funzionare correttamente.

Raggruppamento in un singolo file

```
pyinstaller myscript.py -F
```

Le opzioni per generare un singolo file sono `-F` o `--onefile` . Questo raggruppa il programma in un singolo file `myscript.exe` .

Il singolo file eseguibile è più lento del pacchetto a una cartella. Sono anche più difficili da eseguire il debug.

Leggi PyInstaller - Distribuzione del codice Python online:
https://riptutorial.com/it/python/topic/2289/pyinstaller---distribuzione-del-codice-python

Capitolo 155: Python Anti-Patterns

Examples

Overzealous tranne clausola

Le eccezioni sono potenti, ma una singola clausola eccessivamente zelante può portare via tutto in un'unica riga.

```
try:
    res = get_result()
    res = res[0]
    log('got result: %r' % res)
except:
    if not res:
        res = ''
    print('got exception')
```

Questo esempio dimostra 3 sintomi dell'antiPattern:

1. L' `except` con nessun tipo di eccezione (riga 5) otterrà eccezioni anche salutari, incluso `KeyboardInterrupt` . Ciò impedirà al programma di uscire in alcuni casi.
2. Il blocco except non genera un errore, il che significa che non saremo in grado di stabilire se l'eccezione proviene da `get_result` o perché `res` era una lista vuota.
3. Peggio ancora, se eravamo preoccupati che il risultato fosse vuoto, abbiamo causato qualcosa di molto peggio. Se `get_result` fallisce, `res` rimarrà completamente disinserito, e il riferimento a `res` nel blocco except, solleverà `NameError` , mascherando completamente l'errore originale.

Pensa sempre al tipo di eccezione che stai cercando di gestire. Dai una lettura alla pagina delle eccezioni e fatti un'idea delle eccezioni di base esistenti.

Ecco una versione fissa dell'esempio sopra:

```
import traceback

try:
    res = get_result()
except Exception:
    log_exception(traceback.format_exc())
    raise
try:
    res = res[0]
except IndexError:
    res = ''

log('got result: %r' % res)
```

Rileviamo eccezioni più specifiche, controrilanciare ove necessario. Qualche altra riga, ma infinitamente più corretta.

Guardando prima di saltare con la funzione intensiva del processore

Un programma può facilmente perdere tempo chiamando più volte una funzione intensiva del processore.

Ad esempio, prendi una funzione che assomiglia a questa: restituisce un intero se il `value` input può produrne uno, altrimenti `None` :

```
def intensive_f(value): # int -> Optional[int]
   # complex, and time-consuming code
   if process_has_failed:
       return None
   return integer_output
```

E potrebbe essere usato nel modo seguente:

```
x = 5
if intensive_f(x) is not None:
    print(intensive_f(x) / 2)
else:
    print(x, "could not be processed")

print(x)
```

Mentre questo funzionerà, ha il problema di chiamare `intensive_f` , che raddoppia il tempo di esecuzione del codice. Una soluzione migliore sarebbe ottenere in anticipo il valore di ritorno della funzione.

```
x = 5
result = intensive_f(x)
if result is not None:
    print(result / 2)
else:
    print(x, "could not be processed")
```

Tuttavia, un modo più chiaro e possibilmente più pitoni è usare le eccezioni, ad esempio:

```
x = 5
try:
    print(intensive_f(x) / 2)
except TypeError: # The exception raised if None + 1 is attempted
    print(x, "could not be processed")
```

Qui non è necessaria alcuna variabile temporanea. Spesso può essere preferibile utilizzare una dichiarazione `assert` e catturare l' `AssertionError` .

Chiavi del dizionario

Un esempio comune di dove questo può essere trovato è l'accesso alle chiavi del dizionario. Ad esempio confronta:

```
bird_speeds = get_very_long_dictionary()

if "european swallow" in bird_speeds:
    speed = bird_speeds["european swallow"]
else:
    speed = input("What is the air-speed velocity of an unladen swallow?")

print(speed)
```

con:

```
bird_speeds = get_very_long_dictionary()

try:
    speed = bird_speeds["european swallow"]
except KeyError:
    speed = input("What is the air-speed velocity of an unladen swallow?")

print(speed)
```

Il primo esempio deve esaminare il dizionario due volte e, poiché questo è un dizionario lungo, potrebbe richiedere molto tempo per farlo ogni volta. Il secondo richiede solo una ricerca attraverso il dizionario, risparmiando così molto tempo del processore.

Un'alternativa a questo è usare `dict.get(key, default)` *, tuttavia molte circostanze potrebbero richiedere operazioni più complesse da eseguire nel caso in cui la chiave non sia presente*

Leggi Python Anti-Patterns online: https://riptutorial.com/it/python/topic/4700/python-anti-patterns

Capitolo 156: Python ed Excel

Examples

Inserisci i dati dell'elenco in un file di Excel.

```
import os, sys
from openpyxl import Workbook
from datetime import datetime

dt = datetime.now()
list_values = [["01/01/2016", "05:00:00", 3], \
               ["01/02/2016", "06:00:00", 4], \
               ["01/03/2016", "07:00:00", 5], \
               ["01/04/2016", "08:00:00", 6], \
               ["01/05/2016", "09:00:00", 7]]

# Create a Workbook on Excel:
wb = Workbook()
sheet = wb.active
sheet.title = 'data'

# Print the titles into Excel Workbook:
row = 1
sheet['A'+str(row)] = 'Date'
sheet['B'+str(row)] = 'Hour'
sheet['C'+str(row)] = 'Value'

# Populate with data
for item in list_values:
    row += 1
    sheet['A'+str(row)] = item[0]
    sheet['B'+str(row)] = item[1]
    sheet['C'+str(row)] = item[2]

# Save a file by date:
filename = 'data_' + dt.strftime("%Y%m%d_%I%M%S") + '.xlsx'
wb.save(filename)

# Open the file for the user:
os.chdir(sys.path[0])
os.system('start excel.exe "%s\\%s"' % (sys.path[0], filename, ))
```

OpenPyXL

OpenPyXL è un modulo per la manipolazione e la creazione di cartelle di lavoro `xlsx/xlsm/xltx/xltm` in memoria.

Manipolazione e lettura di una cartella di lavoro esistente:

```
import openpyxl as opx
#To change an existing wookbook we located it by referencing its path
workbook = opx.load_workbook(workbook_path)
```

`load_workbook()` contiene il parametro `read_only` , impostandolo su `True` caricherà la cartella di lavoro come read_only, questo è utile quando si leggono file `xlsx` più `xlsx` :

```
workbook = opx.load_workbook(workbook_path, read_only=True)
```

Dopo aver caricato la cartella di lavoro in memoria, è possibile accedere ai singoli fogli utilizzando le `workbook.sheets`

```
first_sheet = workbook.worksheets[0]
```

Se si desidera specificare il nome di un foglio disponibile, è possibile utilizzare `workbook.get_sheet_names()` .

```
sheet = workbook.get_sheet_by_name('Sheet Name')
```

Infine, è possibile accedere alle righe del foglio utilizzando `sheet.rows` . Per scorrere le righe in un foglio, utilizzare:

```
for row in sheet.rows:
    print row[0].value
```

Poiché ogni `row` in `rows` è un elenco di `Cell` s, utilizzare `Cell.value` per ottenere il contenuto della cella.

Creare una nuova cartella di lavoro in memoria:

```
#Calling the Workbook() function creates a new book in memory
wb = opx.Workbook()

#We can then create a new sheet in the wb
ws = wb.create_sheet('Sheet Name', 0) #0 refers to the index of the sheet order in the wb
```

Diverse proprietà della scheda possono essere modificate tramite openpyxl, ad esempio `tabColor` :

```
ws.sheet_properties.tabColor = 'FFC0CB'
```

Per salvare la nostra cartella di lavoro creata finiamo con:

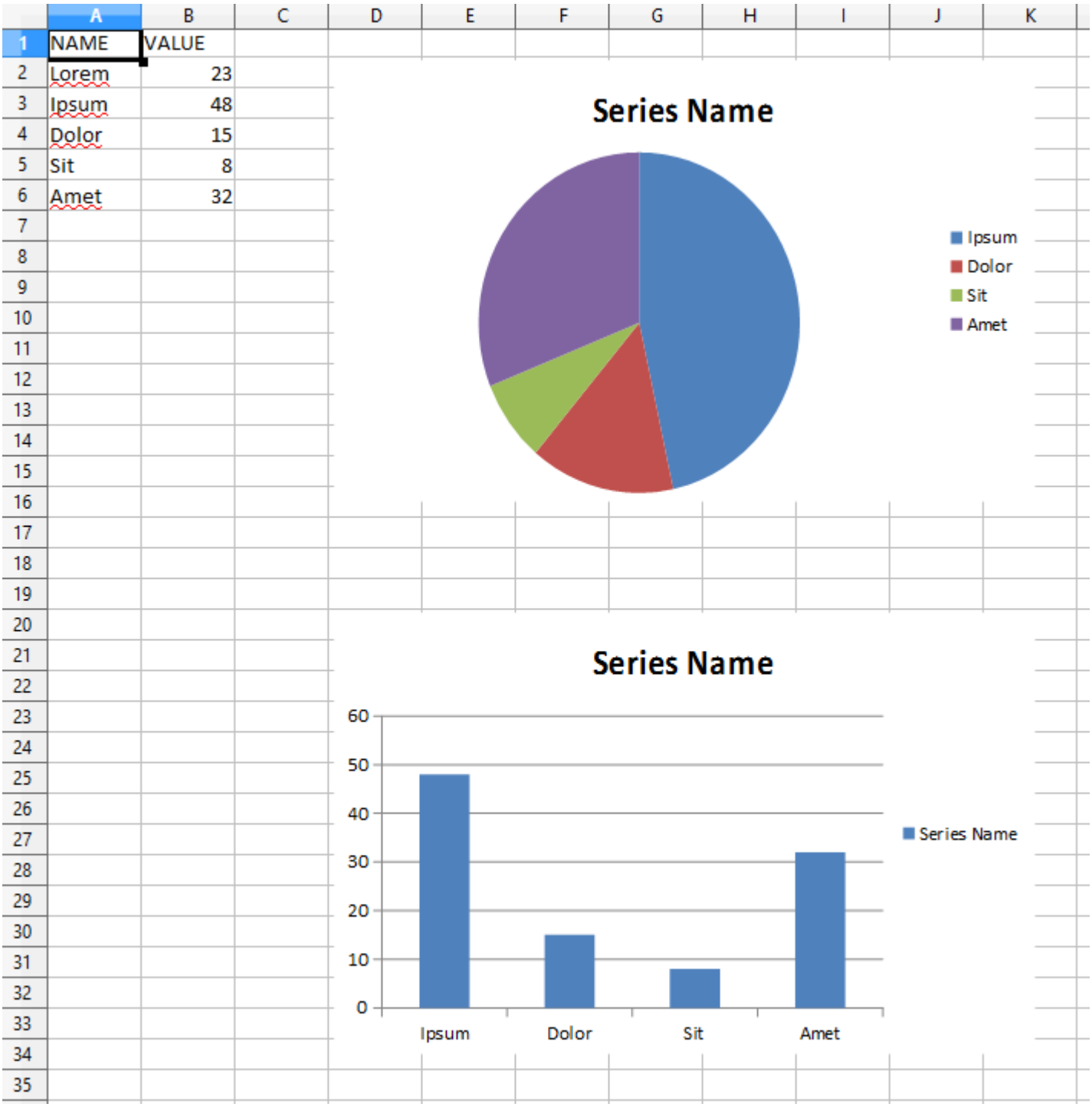
```
wb.save('filename.xlsx')
```

Crea grafici Excel con xlsxwriter

```
import xlsxwriter

# sample data
chart_data = [
    {'name': 'Lorem', 'value': 23},
    {'name': 'Ipsum', 'value': 48},
    {'name': 'Dolor', 'value': 15},
```

```
    {'name': 'Sit', 'value': 8},
    {'name': 'Amet', 'value': 32}
]

# excel file path
xls_file = 'chart.xlsx'

# the workbook
workbook = xlsxwriter.Workbook(xls_file)

# add worksheet to workbook
worksheet = workbook.add_worksheet()

row_ = 0
col_ = 0

# write headers
worksheet.write(row_, col_, 'NAME')
col_ += 1
worksheet.write(row_, col_, 'VALUE')
row_ += 1

# write sample data
for item in chart_data:
    col_ = 0
    worksheet.write(row_, col_, item['name'])
    col_ += 1
    worksheet.write(row_, col_, item['value'])
    row_ += 1

# create pie chart
pie_chart = workbook.add_chart({'type': 'pie'})

# add series to pie chart
pie_chart.add_series({
    'name': 'Series Name',
    'categories': '=Sheet1!$A$3:$A$%s' % row_,
    'values': '=Sheet1!$B$3:$B$%s' % row_,
    'marker': {'type': 'circle'}
})
# insert pie chart
worksheet.insert_chart('D2', pie_chart)

# create column chart
column_chart = workbook.add_chart({'type': 'column'})

# add serie to column chart
column_chart.add_series({
    'name': 'Series Name',
    'categories': '=Sheet1!$A$3:$A$%s' % row_,
    'values': '=Sheet1!$B$3:$B$%s' % row_,
    'marker': {'type': 'circle'}
})
# insert column chart
worksheet.insert_chart('D20', column_chart)

workbook.close()
```

Risultato:



Leggi i dati excel usando il modulo xlrd

La libreria xlrd Python serve per estrarre i dati dai file di foglio di calcolo Microsoft Excel (tm).

Installazione:-

```
pip install xlrd
```

Oppure puoi usare il file setup.py da pypi

https://pypi.python.org/pypi/xlrd

Lettura di un foglio Excel: - Importare il modulo xlrd e aprire il file excel usando il metodo open_workbook ().

```
import xlrd
book=xlrd.open_workbook('sample.xlsx')
```

Controlla il numero di fogli in Excel

```
print book.nsheets
```

Stampa i nomi dei fogli

```
print book.sheet_names()
```

Ottieni il foglio in base all'indice

```
sheet=book.sheet_by_index(1)
```

Leggi il contenuto di una cella

```
cell = sheet.cell(row,col) #where row=row number and col=column number
print cell.value #to print the cell contents
```

Ottieni il numero di righe e il numero di colonne in un foglio Excel

```
num_rows=sheet.nrows
num_col=sheet.ncols
```

Ottieni un foglio Excel per nome

```
sheets = book.sheet_names()
cur_sheet = book.sheet_by_name(sheets[0])
```

Formatta i file Excel con xlsxwriter

```
import xlsxwriter

# create a new file
workbook = xlsxwriter.Workbook('your_file.xlsx')

# add some new formats to be used by the workbook
percent_format = workbook.add_format({'num_format': '0%'})
percent_with_decimal = workbook.add_format({'num_format': '0.0%'})
bold = workbook.add_format({'bold': True})
red_font = workbook.add_format({'font_color': 'red'})
remove_format = workbook.add_format()

# add a new sheet
worksheet = workbook.add_worksheet()
```

```
# set the width of column A
worksheet.set_column('A:A', 30, )

# set column B to 20 and include the percent format we created earlier
worksheet.set_column('B:B', 20, percent_format)

# remove formatting from the first row (change in height=None)
worksheet.set_row('0:0', None, remove_format)

workbook.close()
```

Leggi Python ed Excel online: https://riptutorial.com/it/python/topic/2986/python-ed-excel

Capitolo 157: Python HTTP Server

Examples

Esecuzione di un semplice server HTTP

Python 2.x 2.3

```
python -m SimpleHTTPServer 9000
```

Python 3.x 3.0

```
python -m http.server 9000
```

L'esecuzione di questo comando serve i file della directory corrente sulla porta `9000` .

Se non viene fornito alcun argomento come numero di porta, il server verrà eseguito sulla porta predefinita `8000` .

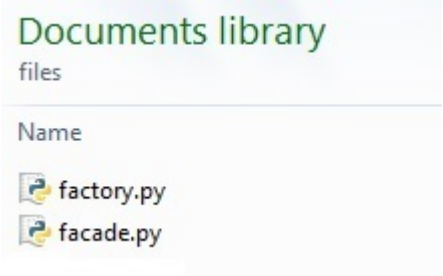
Il flag `-m` cercherà `sys.path` per il corrispondente file `.py` da eseguire come modulo.

Se vuoi servire solo su localhost devi scrivere un programma Python personalizzato come:

```
import sys
import BaseHTTPServer
from SimpleHTTPServer import SimpleHTTPRequestHandler

HandlerClass = SimpleHTTPRequestHandler
ServerClass  = BaseHTTPServer.HTTPServer
Protocol     = "HTTP/1.0"

if sys.argv[1:]:
   port = int(sys.argv[1])
else:
   port = 8000
server_address = ('127.0.0.1', port)

HandlerClass.protocol_version = Protocol
httpd = ServerClass(server_address, HandlerClass)

sa = httpd.socket.getsockname()
print "Serving HTTP on", sa[0], "port", sa[1], "..."
httpd.serve_forever()
```

Servire file

Supponendo di avere la seguente directory di file:



È possibile configurare un server Web per servire questi file come segue:

Python 2.x 2.3

```
import SimpleHTTPServer
import SocketServer

PORT = 8000

handler = SimpleHTTPServer.SimpleHTTPRequestHandler
httpd = SocketServer.TCPServer(("localhost", PORT), handler)
print "Serving files at port {}".format(PORT)
httpd.serve_forever()
```

Python 3.x 3.0

```
import http.server
import socketserver

PORT = 8000

handler = http.server.SimpleHTTPRequestHandler
httpd = socketserver.TCPServer(("", PORT), handler)
print("serving at port", PORT)
httpd.serve_forever()
```

Il modulo `SocketServer` fornisce le classi e le funzionalità per configurare un server di rete.

`SocketServer` s' `TCPServer` classe imposta un server utilizzando il protocollo TCP. Il costruttore accetta una tupla che rappresenta l'indirizzo del server (cioè l'indirizzo IP e la porta) e la classe che gestisce le richieste del server.

La `SimpleHTTPRequestHandler` classe del `SimpleHTTPServer` modulo consente i file nella directory corrente per essere servito.

Salva lo script nella stessa directory ed eseguilo.

Esegui il server HTTP:

Python 2.x 2.3

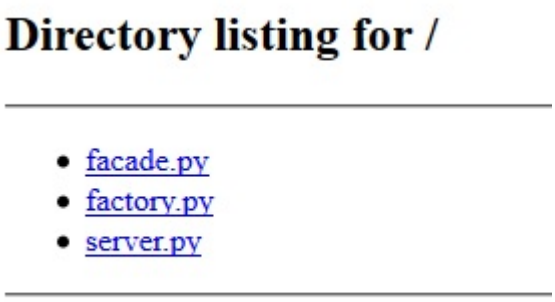
python -m SimpleHTTPServer 8000

Python 3.x 3.0

python -m http.server 8000

Il flag '-m' cercherà 'sys.path' per il corrispondente file '.py' da eseguire come modulo.

Apri localhost: 8000 nel browser, ti darà quanto segue:



API programmatica di SimpleHTTPServer

Cosa succede quando eseguiamo `python -m SimpleHTTPServer 9000` ?

Per rispondere a questa domanda dovremmo capire il costrutto di SimpleHTTerver (https://hg.python.org/cpython/file/2.7/Lib/SimpleHTTerver.py) e BaseHTTerver (https://hg.python.org/cpython/file /2.7/Lib/BaseHTTerver.py) .

In primo luogo, Python richiama il modulo `SimpleHTTPServer` con `9000` come argomento. Ora osservando il codice SimpleHTTerver,

```
def test(HandlerClass = SimpleHTTPRequestHandler,
         ServerClass = BaseHTTPServer.HTTPServer):
    BaseHTTPServer.test(HandlerClass, ServerClass)


if __name__ == '__main__':
    test()
```

La funzione di test è invocata dopo i gestori di richieste e ServerClass. Ora viene chiamato BaseHTTerver.test

```
def test(HandlerClass = BaseHTTPRequestHandler,
         ServerClass = HTTPServer, protocol="HTTP/1.0"):
"""Test the HTTP request handler class.

This runs an HTTP server on port 8000 (or the first command line
argument).

"""

if sys.argv[1:]:
    port = int(sys.argv[1])
else:
    port = 8000
server_address = ('', port)

HandlerClass.protocol_version = protocol
httpd = ServerClass(server_address, HandlerClass)
```

```
sa = httpd.socket.getsockname()
print "Serving HTTP on", sa[0], "port", sa[1], "..."
httpd.serve_forever()
```

Quindi qui il numero di porta, che l'utente ha passato come argomento viene analizzato ed è associato all'indirizzo host. Vengono eseguiti ulteriori passaggi di base per la programmazione dei socket con data porta e protocollo. Finalmente viene avviato il socket server.

Questa è una panoramica di base dell'ereditarietà della classe SocketServer in altre classi:

```
    +------------+
    | BaseServer |
    +------------+
          |
          v
    +-----------+        +------------------+
    | TCPServer |------->| UnixStreamServer |
    +-----------+        +------------------+
          |
          v
    +-----------+        +--------------------+
    | UDPServer |------->| UnixDatagramServer |
    +-----------+        +--------------------+
```

I link https://hg.python.org/cpython/file/2.7/Lib/BaseHTTerver.py e https://hg.python.org/cpython/file/2.7/Lib/SocketServer.py sono utili per trovare ulteriori informazione.

Gestione di base di GET, POST, PUT utilizzando BaseHTTPRequestHandler

```
# from BaseHTTPServer import BaseHTTPRequestHandler, HTTPServer # python2
from http.server import BaseHTTPRequestHandler, HTTPServer # python3
class HandleRequests(BaseHTTPRequestHandler):
    def _set_headers(self):
        self.send_response(200)
        self.send_header('Content-type', 'text/html')
        self.end_headers()

    def do_GET(self):
        self._set_headers()
        self.wfile.write("received get request")

    def do_POST(self):
        '''Reads post request body'''
        self._set_headers()
        content_len = int(self.headers.getheader('content-length', 0))
        post_body = self.rfile.read(content_len)
        self.wfile.write("received post request:<br>{}".format(post_body))

    def do_PUT(self):
        self.do_POST()

host = ''
port = 80
HTTPServer((host, port), HandleRequests).serve_forever()
```

Esempio di output usando `curl` :

```
$ curl http://localhost/
received get request%


$ curl -X POST http://localhost/
received post request:<br>%


$ curl -X PUT http://localhost/
received post request:<br>%


$ echo 'hello world' | curl --data-binary @- http://localhost/
received post request:<br>hello world
```

Leggi Python HTTP Server online: https://riptutorial.com/it/python/topic/4247/python-http-server

Capitolo 158: Python Lex-Yacc

introduzione

PLY è un'implementazione pure-Python dei popolari strumenti di costruzione del compilatore lex e yacc.

Osservazioni

Link aggiuntivi:

1. Documenti ufficiali
2. Github

Examples

Iniziare con PLY

Per installare PLY sulla tua macchina per python2 / 3, procedi come indicato di seguito:

1. Scarica il codice sorgente da qui .
2. Decomprimere il file zip scaricato
3. Passare alla cartella `ply-3.10` decompressa
4. Esegui il seguente comando nel tuo terminale: `python setup.py install`

Se hai completato tutto quanto sopra, ora dovresti essere in grado di utilizzare il modulo PLY. Puoi testarlo aprendo un interprete python e digitando `import ply.lex` .

Nota: *non* utilizzare `pip` per installare PLY, ma installerà una distribuzione interrotta sulla macchina.

"Ciao, mondo!" di PLY - A Simple Calculator

Dimostriamo la potenza di PLY con un semplice esempio: questo programma prenderà un'espressione aritmetica come input di stringa e tenterà di risolverlo.

Apri il tuo editor preferito e copia il seguente codice:

```
from ply import lex
import ply.yacc as yacc

tokens = (
    'PLUS',
    'MINUS',
    'TIMES',
    'DIV',
    'LPAREN',
```

```
    'RPAREN',
    'NUMBER',
)

t_ignore = ' \t'

t_PLUS    = r'\+'
t_MINUS   = r'-'
t_TIMES   = r'\*'
t_DIV     = r'/'
t_LPAREN  = r'\('
t_RPAREN  = r'\)'

def t_NUMBER( t ) :
    r'[0-9]+'
    t.value = int( t.value )
    return t

def t_newline( t ):
  r'\n+'
  t.lexer.lineno += len( t.value )

def t_error( t ):
  print("Invalid Token:",t.value[0])
  t.lexer.skip( 1 )

lexer = lex.lex()

precedence = (
    ( 'left', 'PLUS', 'MINUS' ),
    ( 'left', 'TIMES', 'DIV' ),
    ( 'nonassoc', 'UMINUS' )
)

def p_add( p ) :
    'expr : expr PLUS expr'
    p[0] = p[1] + p[3]

def p_sub( p ) :
    'expr : expr MINUS expr'
    p[0] = p[1] - p[3]

def p_expr2uminus( p ) :
    'expr : MINUS expr %prec UMINUS'
    p[0] = - p[2]

def p_mult_div( p ) :
    '''expr : expr TIMES expr
            | expr DIV expr'''

    if p[2] == '*' :
        p[0] = p[1] * p[3]
    else :
        if p[3] == 0 :
            print("Can't divide by 0")
            raise ZeroDivisionError('integer division by 0')
        p[0] = p[1] / p[3]

def p_expr2NUM( p ) :
    'expr : NUMBER'
    p[0] = p[1]
```

```
def p_parens( p ) :
    'expr : LPAREN expr RPAREN'
    p[0] = p[2]

def p_error( p ):
    print("Syntax error in input!")

parser = yacc.yacc()

res = parser.parse("-4*-(3-5)") # the input
print(res)
```

Salva questo file come `calc.py` ed `calc.py` .

Produzione:

```
-8
```

Qual è la risposta giusta per `-4 * - (3 - 5)` .

Parte 1: Tokenizing Input con Lex

Esistono due passaggi eseguiti dal codice dell'esempio 1: uno stava *tokenizzando* l'input, il che significa che cercava i simboli che costituiscono l'espressione aritmetica e il secondo passo era l' *analisi* , che comporta l'analisi dei token estratti e la valutazione del risultato.

Questa sezione fornisce un semplice esempio di come *tokenizzare* l'input dell'utente, e quindi scomposizione riga per riga.

```
    import ply.lex as lex

    # List of token names. This is always required
    tokens = [
       'NUMBER',
       'PLUS',
       'MINUS',
       'TIMES',
       'DIVIDE',
       'LPAREN',
       'RPAREN',
    ]

    # Regular expression rules for simple tokens
    t_PLUS    = r'\+'
    t_MINUS   = r'-'
    t_TIMES   = r'\*'
    t_DIVIDE  = r'/'
    t_LPAREN  = r'\('
    t_RPAREN  = r'\)'

    # A regular expression rule with some action code
    def t_NUMBER(t):
        r'\d+'
        t.value = int(t.value)
        return t
```

```
# Define a rule so we can track line numbers
def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)

# A string containing ignored characters (spaces and tabs)
t_ignore  = ' \t'

# Error handling rule
def t_error(t):
    print("Illegal character '%s'" % t.value[0])
    t.lexer.skip(1)

# Build the lexer
lexer = lex.lex()

# Give the lexer some input
lexer.input(data)

# Tokenize
while True:
    tok = lexer.token()
    if not tok:
        break      # No more input
    print(tok)
```

Salva questo file come `calclex.py` . Lo useremo quando costruiamo il nostro parser Yacc.

Abbattersi

1. Importa il modulo usando `import ply.lex`

2. Tutti i lexer devono fornire un elenco chiamato `tokens` che definisca tutti i possibili nomi di token che possono essere generati dal lexer. Questa lista è sempre richiesta.

```
tokens = [
   'NUMBER',
   'PLUS',
   'MINUS',
   'TIMES',
   'DIVIDE',
   'LPAREN',
   'RPAREN',
]
```

`tokens` potrebbero anche essere una tupla di stringhe (piuttosto che una stringa), in cui ogni stringa indica un token come prima.

3. La regola regex per ogni stringa può essere definita come una stringa o come una funzione. In entrambi i casi, il nome della variabile deve essere preceduto da t_ per indicare che si tratta di una regola per la corrispondenza dei token.

- Per i token semplici, l'espressione regolare può essere specificata come stringhe: `t_PLUS = r'\+'`

- Se è necessario eseguire un tipo di azione, è possibile specificare una regola token come funzione.

```
def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t
```

Nota, la regola è specificata come una stringa doc all'interno della funzione. La funzione accetta un argomento che è un'istanza di `LexToken` , esegue un'azione e quindi restituisce l'argomento.

Se si desidera utilizzare una stringa esterna come regola regex per la funzione anziché specificare una stringa doc, considerare il seguente esempio:

```
@TOKEN(identifier)          # identifier is a string holding the regex
def t_ID(t):
    ...      # actions
```

- `LexToken` dell'oggetto `LexToken` (chiamiamo questo oggetto `t`) ha i seguenti attributi:

    1. `t.type` che è il tipo di token (come una stringa) (ad esempio: `'NUMBER'` , `'PLUS'` , ecc.). Per impostazione predefinita, `t.type` è impostato sul nome che segue il prefisso `t_` .
    2. `t.value` che è il lessema (il testo vero e proprio corrisponde)
    3. `t.lineno` che è il numero di linea corrente (questo non viene aggiornato automaticamente, poiché il lexer non sa nulla dei numeri di riga). Aggiorna lineno usando una funzione chiamata `t_newline` .

```
def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)
```

    4. `t.lexpos` che è la posizione del token rispetto all'inizio del testo di input.

- Se non viene restituito nulla da una funzione di regola regex, il token viene scartato. Se vuoi scartare un token, puoi in alternativa aggiungere il prefisso t_ignore_ a una variabile della regola di espressione regolare invece di definire una funzione per la stessa regola.

```
def t_COMMENT(t):
    r'\#.*'
    pass
    # No return value. Token discarded
```

...Equivale a:

```
   t_ignore_COMMENT = r'\#.*'
```

Questo è ovviamente non valido se stai compiendo qualche azione quando vedi un commento. In tal caso, utilizzare una funzione per definire la regola regex.

Se non hai definito un token per alcuni personaggi ma desideri comunque ignorarlo, usa `t_ignore = "<characters to ignore>"` (questi prefissi sono necessari):

```
   t_ignore_COMMENT = r'\#.*'
   t_ignore  = ' \t'    # ignores spaces and tabs
```

- Quando si costruisce la regex master, lex aggiungerà le espressioni regolari specificate nel file come segue:

    1. I token definiti dalle funzioni vengono aggiunti nello stesso ordine in cui appaiono nel file.
    2. I token definiti dalle stringhe vengono aggiunti in ordine decrescente della lunghezza della stringa della stringa che definisce la regex per quel token.

    Se stai cercando `==` e `=` nello stesso file, approfitta di queste regole.

- I letterali sono token che vengono restituiti così come sono. Sia `t.type` che `t.value` saranno impostati sul carattere stesso. Definisci una lista di letterali in quanto tale:

```
literals = [ '+', '-', '*', '/' ]
```

o,

```
literals = "+-*/"
```

È possibile scrivere funzioni token che eseguono ulteriori azioni quando i valori letterali sono abbinati. Tuttavia, dovrai impostare il tipo di token in modo appropriato. Per esempio:

```
literals = [ '{', '}' ]

def t_lbrace(t):
    r'\{'
    t.type = '{'  # Set token type to the expected literal (ABSOLUTE MUST if this
is a literal)
    return t
```

- Gestire gli errori con la funzione t_error.

```
# Error handling rule
def t_error(t):
    print("Illegal character '%s'" % t.value[0])
    t.lexer.skip(1) # skip the illegal token (don't process it)
```

In generale, `t.lexer.skip(n)` salta n caratteri nella stringa di input.

4. Preparazioni finali:

   Costruisci il lexer usando `lexer = lex.lex()` .

   Puoi anche inserire tutto in una classe e chiamare l'istanza di utilizzo della classe per definire il lexer. Per esempio:

```
 import ply.lex as lex
 class MyLexer(object):
       ...     # everything relating to token rules and error handling comes here as
usual

       # Build the lexer
       def build(self, **kwargs):
           self.lexer = lex.lex(module=self, **kwargs)

       def test(self, data):
           self.lexer.input(data)
           for token in self.lexer.token():
               print(token)

       # Build the lexer and try it out

 m = MyLexer()
 m.build()           # Build the lexer
 m.test("3 + 4")     #
```

   Fornire input utilizzando `lexer.input(data)` cui i dati sono una stringa

   Per ottenere i token, utilizzare `lexer.token()` che restituisce i token corrispondenti. Puoi scorrere su lexer in un loop come in:

```
for i in lexer:
    print(i)
```

Parte 2: Parsing Input Tokenized con Yacc

Questa sezione spiega come viene elaborato l'input tokenizzato dalla Parte 1, che viene eseguito utilizzando Grammatiche Context Free (CFG). La grammatica deve essere specificata e i token vengono elaborati in base alla grammatica. Sotto il cofano, il parser usa un parser LALR.

```
# Yacc example

import ply.yacc as yacc

# Get the token map from the lexer. This is required.
from calclex import tokens

def p_expression_plus(p):
    'expression : expression PLUS term'
    p[0] = p[1] + p[3]
```

```
def p_expression_minus(p):
    'expression : expression MINUS term'
    p[0] = p[1] - p[3]

def p_expression_term(p):
    'expression : term'
    p[0] = p[1]

def p_term_times(p):
    'term : term TIMES factor'
    p[0] = p[1] * p[3]

def p_term_div(p):
    'term : term DIVIDE factor'
    p[0] = p[1] / p[3]

def p_term_factor(p):
    'term : factor'
    p[0] = p[1]

def p_factor_num(p):
    'factor : NUMBER'
    p[0] = p[1]

def p_factor_expr(p):
    'factor : LPAREN expression RPAREN'
    p[0] = p[2]

# Error rule for syntax errors
def p_error(p):
    print("Syntax error in input!")

# Build the parser
parser = yacc.yacc()

while True:
   try:
       s = raw_input('calc > ')
   except EOFError:
       break
   if not s: continue
   result = parser.parse(s)
   print(result)
```

Abbattersi

- Ogni regola grammaticale è definita da una funzione in cui la docstring a quella funzione contiene le specifiche grammaticali appropriate senza contesto. Le affermazioni che costituiscono il corpo della funzione implementano le azioni semantiche della regola. Ogni funzione accetta un singolo argomento p che è una sequenza contenente i valori di ciascun simbolo grammaticale nella regola corrispondente. I valori di `p[i]` sono mappati ai simboli grammaticali come mostrato qui:

  ```
  def p_expression_plus(p):
      'expression : expression PLUS term'
  ```

```
    #   ^            ^        ^    ^
    #  p[0]         p[1]     p[2] p[3]

    p[0] = p[1] + p[3]
```

- Per i token, il "valore" del corrispondente `p[i]` è uguale all'attributo `p.value` assegnato nel modulo lexer. Quindi, `PLUS` avrà il valore `+` .
- Per i non-terminali, il valore è determinato da qualunque cosa sia posizionata in `p[0]` . Se non viene inserito nulla, il valore è Nessuno. Inoltre, `p[-1]` non è lo stesso di `p[3]` , poiché `p` non è una lista semplice (`p[-1]` può specificare azioni incorporate (non discusse qui)).

Si noti che la funzione può avere qualsiasi nome, purché preceduto da `p_` .

- La `p_error(p)` è definita per `yyerror` errori di sintassi (come `yyerror` in yacc / bison).
- Più regole grammaticali possono essere combinate in un'unica funzione, che è una buona idea se le produzioni hanno una struttura simile.

```
def p_binary_operators(p):
    '''expression : expression PLUS term
                  | expression MINUS term
       term       : term TIMES factor
                  | term DIVIDE factor'''
    if p[2] == '+':
        p[0] = p[1] + p[3]
    elif p[2] == '-':
        p[0] = p[1] - p[3]
    elif p[2] == '*':
        p[0] = p[1] * p[3]
    elif p[2] == '/':
        p[0] = p[1] / p[3]
```

- I caratteri letterali possono essere usati al posto dei token.

```
def p_binary_operators(p):
    '''expression : expression '+' term
                  | expression '-' term
       term       : term '*' factor
                  | term '/' factor'''
    if p[2] == '+':
        p[0] = p[1] + p[3]
    elif p[2] == '-':
        p[0] = p[1] - p[3]
    elif p[2] == '*':
        p[0] = p[1] * p[3]
    elif p[2] == '/':
        p[0] = p[1] / p[3]
```

  Naturalmente, i valori letterali devono essere specificati nel modulo lexer.

- Le produzioni vuote hanno il `'''symbol : '''`
- Per impostare in modo esplicito il simbolo di partenza, utilizzare `start = 'foo'` , dove `foo` è

un non-terminale.

- L'impostazione della precedenza e dell'associatività può essere eseguita utilizzando la variabile di precedenza.

```
precedence = (
    ('nonassoc', 'LESSTHAN', 'GREATERTHAN'),  # Nonassociative operators
    ('left', 'PLUS', 'MINUS'),
    ('left', 'TIMES', 'DIVIDE'),
    ('right', 'UMINUS'),            # Unary minus operator
)
```

  I token sono ordinati dalla precedenza più bassa a quella più alta. `nonassoc` significa che quei token non si associano. Ciò significa che qualcosa come `a < b < c` è illegale mentre `a < b` è ancora legale.

- `parser.out` è un file di debug che viene creato quando il programma yacc viene eseguito per la prima volta. Ogni volta che si verifica uno spostamento / riduzione del conflitto, il parser si sposta sempre.

Leggi Python Lex-Yacc online: https://riptutorial.com/it/python/topic/10510/python-lex-yacc

Capitolo 159: Python Networking

Osservazioni

Esempio di socket client Python (molto) di base

Examples

Il più semplice esempio di client-server socket Python

Lato server:

```
import socket

serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
serversocket.bind(('localhost', 8089))
serversocket.listen(5) # become a server socket, maximum 5 connections

while True:
    connection, address = serversocket.accept()
    buf = connection.recv(64)
    if len(buf) > 0:
        print(buf)
    break
```

Dalla parte del cliente:

```
import socket

clientsocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
clientsocket.connect(('localhost', 8089))
clientsocket.send('hello')
```

Per prima cosa eseguire SocketServer.py e assicurarsi che il server sia pronto per ascoltare / ricevere sth Quindi il client invia le informazioni al server; Dopo che il server ha ricevuto sth, termina

Creazione di un server Http semplice

Per condividere file o per ospitare siti Web semplici (http e javascript) nella rete locale, è possibile utilizzare il modulo SimpleHTTPServer integrato di Python. Python dovrebbe essere nella tua variabile Path. Vai alla cartella in cui si trovano i tuoi file e digita:

Per `python 2` :

```
$ python -m SimpleHTTPServer <portnumber>
```

Per `python 3` :

```
$ python3 -m http.server <portnumber>
```

Se il numero di porta non è dato `8000` è la porta predefinita. Quindi l'output sarà:

> Servire HTTP su 0.0.0.0 porta 8000 ...

È possibile accedere ai propri file tramite qualsiasi dispositivo connesso alla rete locale digitando `http://hostipaddress:8000/` .

`hostipaddress` è il tuo indirizzo IP locale che probabilmente inizia con `192.168.xx`

Per completare il modulo, premi semplicemente `ctrl+c`.

Creazione di un server TCP

È possibile creare un server TCP utilizzando la libreria `socketserver` . Ecco un semplice server echo.

Lato server

```
from sockerserver import BaseRequestHandler, TCPServer

class EchoHandler(BaseRequestHandler):
    def handle(self):
        print('connection from:', self.client_address)
        while True:
            msg = self.request.recv(8192)
            if not msg:
                break
            self.request.send(msg)

if __name__ == '__main__':
    server = TCPServer(('', 5000), EchoHandler)
    server.serve_forever()
```

Dalla parte del cliente

```
from socket import socket, AF_INET, SOCK_STREAM
sock = socket(AF_INET, SOCK_STREAM)
sock.connect(('localhost', 5000))
sock.send(b'Monty Python')
sock.recv(8192)  # returns b'Monty Python'
```

`socketserver` rende relativamente facile creare semplici server TCP. Tuttavia, è necessario essere consapevoli del fatto che, per impostazione predefinita, i server sono a thread singolo e possono servire solo un client alla volta. Se si desidera gestire più client, creare un'istanza di `ThreadingTCPServer` .

```
from socketserver import ThreadingTCPServer
...
if __name__ == '__main__':
    server = ThreadingTCPServer(('', 5000), EchoHandler)
    server.serve_forever()
```

Creazione di un server UDP

Un server UDP è facilmente creato utilizzando la libreria `socketserver` .

un semplice server orario:

```
import time
from socketserver import BaseRequestHandler, UDPServer

class CtimeHandler(BaseRequestHandler):
    def handle(self):
    print('connection from: ', self.client_address)
    # Get message and client socket
    msg, sock = self.request
    resp = time.ctime()
    sock.sendto(resp.encode('ascii'), self.client_address)

if __name__ == '__main__':
    server = UDPServer(('', 5000), CtimeHandler)
    server.serve_forever()
```

test:

```
>>> from socket import socket, AF_INET, SOCK_DGRAM
>>> sock = socket(AF_INET, SOCK_DGRAM)
>>> sick.sendto(b'', ('localhost', 5000))
0
>>> sock.recvfrom(8192)
(b'Wed Aug 15 20:35:08 2012', ('127.0.0.1', 5000))
```

Avvia Simple HttpServer in una discussione e apri il browser

Utile se il tuo programma sta generando pagine web lungo la strada.

```
from http.server import HTTPServer, CGIHTTPRequestHandler
import webbrowser
import threading

def start_server(path, port=8000):
    '''Start a simple webserver serving path on port'''
    os.chdir(path)
    httpd = HTTPServer(('', port), CGIHTTPRequestHandler)
    httpd.serve_forever()

# Start the server in a new thread
port = 8000
daemon = threading.Thread(name='daemon_server',
                          target=start_server,
                          args=('.', port)
daemon.setDaemon(True) # Set as a daemon so it will be killed once the main thread is dead.
daemon.start()

# Open the web browser
webbrowser.open('http://localhost:{}'.format(port))
```

Leggi Python Networking online: https://riptutorial.com/it/python/topic/1309/python-networking

Capitolo 160: Python Serial Communication (pyserial)

Sintassi

- ser.read (size = 1)
- ser.readline ()
- ser.write ()

Parametri

parametro	dettagli
porta	Nome del dispositivo es. / Dev / ttyUSB0 su GNU / Linux o COM3 su Windows.
baudrate	baudrate type: int default: 9600 valori standard: 50, 75, 110, 134, 150, 200, 300, 600, 1200, 1800, 2400, 4800, 9600, 19200, 38400, 57600, 115200

Osservazioni

Per ulteriori dettagli, consultare la documentazione pyserial

Examples

Inizializza dispositivo seriale

```
import serial
#Serial takes these two parameters: serial device and baudrate
ser = serial.Serial('/dev/ttyUSB0', 9600)
```

Leggi dalla porta seriale

Inizializza dispositivo seriale

```
import serial
#Serial takes two parameters: serial device and baudrate
ser = serial.Serial('/dev/ttyUSB0', 9600)
```

leggere un singolo byte dal dispositivo seriale

```
data = ser.read()
```

leggere il numero dato di byte dal dispositivo seriale

```
data = ser.read(size=5)
```

leggere una riga dal dispositivo seriale.

```
data = ser.readline()
```

leggere i dati dal dispositivo seriale mentre qualcosa è stato scritto su di esso.

```
#for python2.7
data = ser.read(ser.inWaiting())

#for python3
ser.read(ser.inWaiting)
```

Verifica quali porte seriali sono disponibili sulla tua macchina

Utilizzare per ottenere un elenco di porte seriali disponibili

```
python -m serial.tools.list_ports
```

al prompt dei comandi o

```
from serial.tools import list_ports
list_ports.comports()  # Outputs list of available serial ports
```

dalla shell Python.

Leggi Python Serial Communication (pyserial) online:
https://riptutorial.com/it/python/topic/5744/python-serial-communication--pyserial-

Capitolo 161: Python velocità del programma

Examples

Notazione

Idea base

La notazione usata quando descrive la velocità del tuo programma Python è chiamata notazione Big-O. Diciamo che hai una funzione:

```
def list_check(to_check, the_list):
    for item in the_list:
        if to_check == item:
          return True
    return False
```

Questa è una semplice funzione per verificare se un elemento è in una lista. Per descrivere la complessità di questa funzione, dirai O (n). Questo significa "Ordine di n" poiché la funzione O è nota come funzione Ordine.

O (n) - generalmente n è il numero di elementi nel contenitore

O (k) - generalmente k è il valore del parametro o il numero di elementi nel parametro

Elenca le operazioni

Operazioni: caso medio (presuppone che i parametri siano generati casualmente)

Aggiungi: O (1)

Copia: O (n)

Del slice: O (n)

Elimina oggetto: O (n)

Inserisci: O (n)

Ottieni l'oggetto: O (1)

Set item: O (1)

Iterazione: O (n)

Ottieni una fetta: O (k)

Imposta slice: O (n + k)

Estendi: O (k)

Ordina: O (n log n)

Moltiplicare: O (nk)

x in s: O (n)

min (s), max (s): O (n)

Ottieni la lunghezza: O (1)

Operazioni di deque

Una deque è una coda a doppio attacco.

```
class Deque:
def __init__(self):
    self.items = []

def isEmpty(self):
    return self.items == []

def addFront(self, item):
    self.items.append(item)

def addRear(self, item):
    self.items.insert(0,item)

def removeFront(self):
    return self.items.pop()

def removeRear(self):
    return self.items.pop(0)

def size(self):
    return len(self.items)
```

Operazioni: caso medio (presuppone che i parametri siano generati casualmente)

Aggiungi: O (1)

Appendleft: O (1)

Copia: O (n)

Estendi: O (k)

Extendleft: O (k)

Pop: O (1)

Popleft: O (1)

Rimuovi: O (n)

Ruota: O (k)

Imposta le operazioni

Operazione: caso medio (presuppone che i parametri siano generati casualmente): caso peggiore

x in s: O (1)

Differenza s - t: O (len (s))

Intersezione s & t: O (min (len (s), len (t))): O (len (s) * len (t)

Intersezione multipla s1 & s2 & s3 & ... & sn:: (n-1) * O (l) dove l è max (len (s1), ..., len (sn))

s.difference_update (t): O (len (t)): O (len (t) * len (s))

s.symetric_difference_update (t): O (len (t))

Differenza simmetrica s ^ t: O (len (s)): O (len (s) * len (t))

Union s | t: O (len (s) + len (t))

Notifiche algoritmiche ...

Ci sono alcuni principi che si applicano all'ottimizzazione in qualsiasi linguaggio del computer, e Python non fa eccezione. **Non ottimizzare mentre procedi** : scrivi il tuo programma senza tener conto delle possibili ottimizzazioni, concentrandoti invece a fare in modo che il codice sia pulito, corretto e comprensibile. Se è troppo grande o troppo lento quando hai finito, puoi considerare di ottimizzarlo.

Ricorda la regola 80/20 : in molti campi puoi ottenere l'80% del risultato con il 20% dello sforzo (chiamato anche regola 90/10, dipende da chi stai parlando). Ogni volta che stai per ottimizzare il codice, utilizza la creazione di profili per scoprire dove sta andando l'80% del tempo di esecuzione, quindi sai dove concentrare i tuoi sforzi.

Esegui sempre i benchmark "prima" e "dopo" : in quale altro modo saprai che le tue ottimizzazioni hanno effettivamente fatto la differenza? Se il tuo codice ottimizzato risulta essere solo leggermente più veloce o più piccolo della versione originale, annulla le modifiche e torna al codice originale, chiaro.

Utilizzare gli algoritmi e le strutture dati corretti: non utilizzare un algoritmo di ordinamento di bolle O (n2) per ordinare migliaia di elementi quando è disponibile un quicksort O (n log n) disponibile. Allo stesso modo, non memorizzare un migliaio di elementi in una matrice che richiede una ricerca O (n) quando è possibile utilizzare un albero binario O (log n) o una tabella di hash Python O (1).

Per ulteriori informazioni visita il link sottostante ... Python Speed Up

Le seguenti 3 notazioni asintotiche sono usate principalmente per rappresentare la complessità temporale degli algoritmi.

1. **Θ Notazione** : la notazione theta limita una funzione dall'alto e dal basso, quindi definisce il comportamento asintotico esatto. Un modo semplice per ottenere la notazione Theta di un'espressione è di eliminare i termini di ordine basso e ignorare le costanti principali. Ad esempio, considera la seguente espressione. 3n3 + 6n2 + 6000 = Θ (n3) L'eliminazione dei termini di ordine inferiore è sempre soddisfacente perché ci sarà sempre un n0 dopo il quale Θ (n3) ha valori superiori a Θn2) indipendentemente dalle costanti coinvolte. Per una data funzione g (n), denotiamo che Θ (g (n)) sta seguendo un insieme di funzioni. Θ (g (n)) = {f (n): esistono costanti positive c1, c2 e n0 tali che 0 <= c1 *g (n)* <= *f (n)* <= *c2* g (n) per tutti n> = n0} La definizione sopra indica che se f (n) è theta di g (n), allora il valore f (n) è sempre tra c1 *g (n) e c2* g (n) per valori grandi di n (n> = n0). La definizione di theta richiede anche che f (n) debba essere non negativo per valori di n maggiori di n0.

2. **Notazione O grande** : La notazione O grande definisce un limite superiore di un algoritmo, limita una funzione solo dall'alto. Ad esempio, si consideri il caso di Insertion Sort. Nel caso peggiore ci vuole tempo lineare nel migliore dei casi e nel tempo quadratico. Possiamo tranquillamente affermare che la complessità temporale di Insertion sort è O (n ^ 2). Nota che O (n ^ 2) copre anche il tempo lineare. Se usiamo la notazione to per rappresentare la complessità temporale di Insertion sort, dobbiamo utilizzare due istruzioni per i casi migliori e peggiori:

1. La complessità temporale peggiore di Insertion Sort è Θ (n ^ 2).
2. La migliore complessità del tempo del caso di Insertion Sort è Θ (n).

La notazione Big O è utile quando abbiamo solo il limite superiore alla complessità temporale di un algoritmo. Molte volte troviamo facilmente un limite superiore semplicemente osservando l'algoritmo. O (g (n)) = {f (n): esistono costanti positive c e n0 tali che 0 <= f (n) <= cg (n) per tutto n> = n0}

3. **Notazione Ω** : proprio come la notazione Big O fornisce un limite superiore asintotico su una funzione, la notazione Ω fornisce un limite inferiore asintotico. Ω Notation <può essere utile quando abbiamo una complessità legata al tempo inferiore di un algoritmo. Come discusso nel post precedente, la migliore performance del caso di un algoritmo non è generalmente utile, la notazione Omega è la notazione meno utilizzata tra tutte e tre. Per una data funzione g (n), denotiamo con Ω (g (n)) l'insieme di funzioni. Ω (g (n)) = {f (n): esistono costanti positive c e n0 tali che 0 <= cg (n) <= f (n) per tutto n> = n0}. Consideriamo lo stesso esempio di Insertion sort qui. La complessità temporale di Insertion Sort può essere scritta come Ω (n), ma non è una informazione molto utile sull'inserimento sort, in quanto siamo generalmente interessati al caso peggiore ea volte nel caso medio.

Leggi Python velocità del programma online: https://riptutorial.com/it/python/topic/9185/python-velocita-del-programma

Capitolo 162: Python Virtual Environment - virtualenv

introduzione

Un ambiente virtuale ("virtualenv") è uno strumento per creare ambienti Python isolati. Mantiene le dipendenze richieste da diversi progetti in luoghi separati, creando per loro env Python virtuale. Risolve il "progetto A dipende dalla versione 2.xxx ma, il progetto B ha bisogno del dilemma 2.xxx" e mantiene la directory globale dei pacchetti del sito pulita e gestibile.

"virtualenv" crea una cartella che contiene tutte le librerie e le librerie necessarie per utilizzare i pacchetti necessari a un progetto Python.

Examples

Installazione

Installa virtualenv tramite pip / (apt-get):

```
pip install virtualenv
```

O

```
apt-get install python-virtualenv
```

Nota: in caso di problemi con i permessi, utilizzare sudo.

uso

```
$ cd test_proj
```

Crea un ambiente virtuale:

```
$ virtualenv test_proj
```

Per iniziare a utilizzare l'ambiente virtuale, è necessario attivarlo:

```
$ source test_project/bin/activate
```

Per uscire dal tuo virtualenv basta digitare "disattiva":

```
$ deactivate
```

Installa un pacchetto nel tuo Virtualenv

Se guardi la directory bin nel tuo virtualenv, vedrai easy_install che è stato modificato per mettere uova e pacchetti nella directory dei siti del sito virtualenv. Per installare un'app nel tuo ambiente virtuale:

```
$ source test_project/bin/activate
$ pip install flask
```

Al momento, non è necessario utilizzare sudo poiché i file verranno tutti installati nella directory locale dei pacchetti del sito virtualenv. Questo è stato creato come tuo account utente.

Altri comandi virtualenv utili

lsvirtualenv : elenca tutti gli ambienti.

cdvirtualenv : **naviga** nella directory dell'ambiente virtuale correntemente attivato, così puoi sfogliare i suoi pacchetti del sito, per esempio.

cdsitepackages : come sopra, ma direttamente nella directory dei pacchetti del sito.

lssitepackages : mostra il contenuto della directory dei pacchetti del sito.

Leggi Python Virtual Environment - virtualenv online:
https://riptutorial.com/it/python/topic/9782/python-virtual-environment---virtualenv

Capitolo 163: Raccolta dei rifiuti

Osservazioni

Al suo interno, il garbage collector di Python (a partire da 3.5) è una semplice implementazione di conteggio dei riferimenti. Ogni volta che si effettua un riferimento a un oggetto (ad esempio, `a = myobject`) il conteggio dei riferimenti su quell'oggetto (oggetto mio) viene incrementato. Ogni volta che un riferimento viene rimosso, il conteggio dei riferimenti viene decrementato e una volta che il conteggio dei riferimenti raggiunge `0` , sappiamo che nulla contiene un riferimento a quell'oggetto e possiamo deallocarlo!

Un comune malinteso su come funziona la gestione della memoria di Python è che la parola chiave `del` libera la memoria degli oggetti. Questo non è vero. Quello che succede in realtà è che la parola chiave `del` decrementa semplicemente il conto degli oggetti, il che significa che se lo chiami abbastanza volte perché il refcount raggiunga lo zero l'oggetto può essere garbage collection (anche se in realtà ci sono ancora riferimenti all'oggetto disponibile altrove nel tuo codice).

Python crea o pulisce aggressivamente gli oggetti la prima volta che ne ha bisogno Se eseguo il compito a = object (), la memoria per oggetto viene allocata in quel momento (a volte cpython riutilizzerà determinati tipi di oggetti, ad esempio liste sotto il cofano, ma per lo più non mantiene un pool di oggetti gratuito e eseguirà l'allocazione quando ne hai bisogno). Allo stesso modo, non appena il conto è decrementato a 0, GC lo pulisce.

Raccolta di rifiuti generazionale

Negli anni '60 John McCarthy scoprì un difetto fatale nel conteggiare la raccolta dei rifiuti quando implementò l'algoritmo di refcount utilizzato da Lisp: cosa succede se due oggetti si riferiscono l'un l'altro in un riferimento ciclico? Come puoi mai raccogliere i rifiuti di quei due oggetti, anche se non ci sono riferimenti esterni a loro se si riferiscono sempre a vicenda? Questo problema si estende anche a qualsiasi struttura ciclica dei dati, ad esempio un buffer circolare o due voci consecutive in una lista doppiamente collegata. Python tenta di risolvere questo problema usando una svolta leggermente interessante su un altro algoritmo di garbage collection chiamato **Generational Garbage Collection** .

In pratica, ogni volta che crei un oggetto in Python lo aggiunge alla fine di una lista doppiamente collegata. A volte Python scorre questo elenco, controlla quali oggetti si riferiscono anche agli oggetti nella lista, e se sono anche nella lista (vedremo perché potrebbero non essere in un momento), decrementa ulteriormente i loro conti. A questo punto (in realtà, ci sono alcune euristiche che determinano quando le cose vengono spostate, ma supponiamo che sia dopo una singola raccolta per mantenere le cose semplici) tutto ciò che ha ancora un conto maggiore di 0 viene promosso ad un'altra lista collegata chiamata "Generazione 1" (questo è il motivo per cui tutti gli oggetti non sono sempre nella lista di generazione 0) a cui questo ciclo si applica meno spesso. È qui che arriva la garbage collection generazionale. In Python ci sono 3 generazioni predefinite (tre liste di oggetti collegate): la prima lista (generazione 0) contiene tutti i nuovi oggetti;

se si verifica un ciclo GC e gli oggetti non vengono raccolti, vengono spostati nella seconda lista (generazione 1) e se un ciclo GC si verifica nella seconda lista e non vengono ancora raccolti vengono spostati nella terza lista (generazione 2). L'elenco di terza generazione (chiamato "generazione 2", dato che siamo indicizzati a zero) è garbage collection molto meno spesso dei primi due, l'idea è che se il tuo oggetto è longevo non è probabile che sia GCed, e non potrà mai essere GCed durante la vita della tua applicazione, quindi non c'è motivo di perdere tempo a controllarlo su ogni singola esecuzione di GC. Inoltre, si osserva che la maggior parte degli oggetti viene raccolta in modo relativamente rapido. D'ora in poi, chiameremo questi "oggetti buoni" poiché muoiono giovani. Questa è chiamata "debole ipotesi generazionale" e fu anche osservata per la prima volta negli anni '60.

Un rapido accostamento: a differenza delle prime due generazioni, la lunga lista di terza generazione non è raccolta di rifiuti su base regolare. Viene verificato quando il rapporto tra gli oggetti in sospeso di lunga durata (quelli che si trovano nell'elenco di terza generazione, ma non hanno ancora avuto un ciclo GC) per gli oggetti long life totali nell'elenco è superiore al 25%. Questo perché la terza lista è illimitata (le cose non vengono mai spostate da essa in un'altra lista, quindi vanno via solo quando sono effettivamente raccolte dalla spazzatura), il che significa che per le applicazioni in cui si stanno creando molti oggetti longevi, cicli GC sulla terza lista può diventare piuttosto lungo. Usando un rapporto otteniamo "prestazioni lineari ammortizzate nel numero totale di oggetti"; alias, più lunga è la lista, più GC è lungo, ma meno spesso eseguiamo GC (ecco la proposta originale del 2008 per questa euristica di Martin von Löwis per la lettura successiva). L'atto di eseguire una garbage collection sulla terza generazione o "matura" è chiamato "full garbage collection".

Quindi la raccolta di dati generazionali accelera tremendamente le cose non richiedendo di eseguire la scansione su oggetti che non hanno probabilmente bisogno di GC tutto il tempo, ma come ci aiutano a interrompere i riferimenti ciclici? Probabilmente non molto bene, si scopre. La funzione per interrompere effettivamente questi cicli di riferimento inizia così :

```
/* Break reference cycles by clearing the containers involved.  This is
 * tricky business as the lists can be changing and we don't know which
 * objects may be freed.  It is possible I screwed something up here.
 */
static void
delete_garbage(PyGC_Head *collectable, PyGC_Head *old)
```

Il motivo per cui la garbage collection generazionale aiuta è che possiamo mantenere la lunghezza della lista come un conteggio separato; ogni volta che aggiungiamo un nuovo oggetto alla generazione incrementiamo questo conteggio, e ogni volta che spostiamo un oggetto su un'altra generazione o dealloc decrementiamo il conteggio. Teoricamente alla fine di un ciclo di GC questo conteggio (sempre per le prime due generazioni) dovrebbe sempre essere 0. Se non lo è, qualsiasi cosa nella lista rimasta è una qualche forma di riferimento circolare e possiamo lasciarla cadere. Tuttavia, c'è un altro problema qui: Cosa succede se gli oggetti `__del__` il metodo magico di Python `__del__` su di essi? `__del__` è chiamato ogni volta che un oggetto Python viene distrutto. Tuttavia, se due oggetti in un riferimento circolare hanno metodi `__del__` , non possiamo essere sicuri che la distruzione di uno non interromperà il metodo `__del__` degli altri. Per un esempio forzato, immagina di aver scritto quanto segue:

```
class A(object):
    def __init__(self, b=None):
        self.b = b

    def __del__(self):
        print("We're deleting an instance of A containing:", self.b)

class B(object):
    def __init__(self, a=None):
        self.a = a

    def __del__(self):
        print("We're deleting an instance of B containing:", self.a)
```

e impostiamo un'istanza di A e un'istanza di B per puntare l'un l'altro e quindi finiscono nello stesso ciclo di raccolta dei rifiuti? Diciamo che ne prendiamo uno a caso e deallociamo la nostra istanza di A prima; Verrà chiamato il metodo `__del__` di A, verrà stampato, quindi A verrà liberato. Quindi arriviamo a B, chiamiamo il suo metodo `__del__` e oops! Segfault! A non esiste più. Potremmo risolvere questo problema chiamando tutto ciò che è rimasto prima i metodi `__del__` , poi facendo un altro passaggio per dealloc effettivamente tutto, tuttavia, questo introduce un altro problema: cosa succede se un oggetto `__del__` metodo salva un riferimento all'altro oggetto che sta per essere GCed e ci ha fatto riferimento da qualche altra parte? Abbiamo ancora un ciclo di riferimento, ma ora non è possibile eseguire effettivamente GC su un oggetto, anche se non sono più in uso. Si noti che anche se un oggetto non fa parte di una struttura di dati circolare, potrebbe rianimarsi nel proprio metodo `__del__` ; Python ha un controllo per questo e interromperà GCing se un account di oggetti è aumentato dopo che è stato chiamato il suo metodo `__del__` .

CPython si occupa di ciò bloccando quegli oggetti non-GC (qualsiasi cosa con qualche forma di riferimento circolare e un metodo `__del__`) su una lista globale di rifiuti non recuperabili e lasciandoli lì per l'eternità:

```
/* list of uncollectable objects */
static PyObject *garbage = NULL;
```

Examples

Conteggio di riferimento

La maggior parte della gestione della memoria di Python viene gestita con il conteggio dei riferimenti.

Ogni volta che un oggetto viene referenziato (ad es. Assegnato a una variabile), il suo conteggio dei riferimenti viene automaticamente aumentato. Quando è dereferenziato (ad es. La variabile esce dal campo di applicazione), il suo conteggio di riferimento viene automaticamente diminuito.

Quando il conteggio dei riferimenti raggiunge lo zero, l'oggetto viene **immediatamente distrutto** e la memoria viene immediatamente liberata. Pertanto, per la maggior parte dei casi, il garbage collector non è nemmeno necessario.

```
>>> import gc; gc.disable()  # disable garbage collector
>>> class Track:
        def __init__(self):
            print("Initialized")
        def __del__(self):
            print("Destructed")
>>> def foo():
        Track()
        # destructed immediately since no longer has any references
        print("---")
        t = Track()
        # variable is referenced, so it's not destructed yet
        print("---")
        # variable is destructed when function exits
>>> foo()
Initialized
Destructed
---
Initialized
---
Destructed
```

Per dimostrare ulteriormente il concetto di riferimenti:

```
>>> def bar():
        return Track()
>>> t = bar()
Initialized
>>> another_t = t  # assign another reference
>>> print("...")
...
>>> t = None          # not destructed yet - another_t still refers to it
>>> another_t = None  # final reference gone, object is destructed
Destructed
```

Garbage Collector per cicli di riferimento

L'unica volta che è necessario il garbage collector è se si dispone di un *ciclo di riferimento* . L'esempio semplice di un ciclo di riferimento è uno in cui A si riferisce a B e B si riferisce ad A, mentre nient'altro si riferisce ad A o B. Né A né B sono accessibili da qualsiasi parte del programma, quindi possono essere distrutti in modo sicuro, tuttavia i loro conteggi di riferimento sono 1 e quindi non possono essere liberati solo dall'algoritmo di conteggio di riferimento.

```
>>> import gc; gc.disable()  # disable garbage collector
>>> class Track:
        def __init__(self):
            print("Initialized")
        def __del__(self):
            print("Destructed")
>>> A = Track()
Initialized
>>> B = Track()
Initialized
>>> A.other = B
>>> B.other = A
>>> del A; del B  # objects are not destructed due to reference cycle
```

```
>>> gc.collect()  # trigger collection
Destructed
Destructed
4
```

Un ciclo di riferimento può essere arbitrario. Se A punta a B punta a C punta a ... punta a Z che punta ad A, quindi non verranno collezionate da A a Z, fino alla fase di garbage collection:

```
>>> objs = [Track() for _ in range(10)]
Initialized
Initialized
Initialized
Initialized
Initialized
Initialized
Initialized
Initialized
Initialized
Initialized
>>> for i in range(len(objs)-1):
...     objs[i].other = objs[i + 1]
...
>>> objs[-1].other = objs[0]  # complete the cycle
>>> del objs                  # no one can refer to objs now - still not destructed
>>> gc.collect()
Destructed
Destructed
Destructed
Destructed
Destructed
Destructed
Destructed
Destructed
Destructed
Destructed
20
```

Effetti del comando

Rimozione di un nome di variabile dall'ambito usando `del v` , o rimozione di un oggetto da una raccolta usando `del v[item]` o `del[i:j]` , o rimuovendo un attributo usando `del v.name` , o qualsiasi altro modo di rimuovere riferimenti a un oggetto, *non* innesca alcuna chiamata distruttore o alcuna memoria liberata in sé e per sé. Gli oggetti vengono distrutti solo quando il loro conteggio dei riferimenti raggiunge lo zero.

```
>>> import gc
>>> gc.disable()  # disable garbage collector
>>> class Track:
        def __init__(self):
            print("Initialized")
        def __del__(self):
            print("Destructed")
>>> def bar():
    return Track()
>>> t = bar()
Initialized
```

```
>>> another_t = t  # assign another reference
>>> print("...")
...
>>> del t          # not destructed yet - another_t still refers to it
>>> del another_t  # final reference gone, object is destructed
Destructed
```

Riutilizzo di oggetti primitivi

Una cosa interessante da notare, che può aiutare a ottimizzare le tue applicazioni, è che anche le primitive vengono effettivamente riconquistate. Diamo un'occhiata ai numeri; per tutti gli interi compresi tra -5 e 256, Python riutilizza sempre lo stesso oggetto:

```
>>> import sys
>>> sys.getrefcount(1)
797
>>> a = 1
>>> b = 1
>>> sys.getrefcount(1)
799
```

Si noti che il Refcount aumenta, il che significa che `a` e `b` riferimento allo stesso oggetto sottostante quando si riferiscono alla primitiva `1` . Tuttavia, per numeri più grandi, Python in realtà non riutilizza l'oggetto sottostante:

```
>>> a = 999999999
>>> sys.getrefcount(999999999)
3
>>> b = 999999999
>>> sys.getrefcount(999999999)
3
```

Poiché il refcount per `999999999` non cambia durante l'assegnazione ad `a` e `b` possiamo dedurre che si riferiscono a due diversi oggetti sottostanti, anche se entrambi viene assegnato lo stesso primitivo.

Visualizzazione del conto di un oggetto

```
>>> import sys
>>> a = object()
>>> sys.getrefcount(a)
2
>>> b = a
>>> sys.getrefcount(a)
3
>>> del b
>>> sys.getrefcount(a)
2
```

Disabilitare forzatamente gli oggetti

Puoi forzare gli oggetti deallocate anche se il loro refcount non è 0 in entrambi Python 2 e 3.

Entrambe le versioni usano il modulo `ctypes` per farlo.

ATTENZIONE: facendo questo *lascerà* il vostro ambiente Python instabile e soggetto a crash senza un traceback! L'utilizzo di questo metodo potrebbe anche introdurre problemi di sicurezza (alquanto improbabili) Distribuisci solo gli oggetti di cui sei sicuro di non fare mai più riferimento. Mai.

Python 3.x 3.0

```
import ctypes
deallocated = 12345
ctypes.pythonapi._Py_Dealloc(ctypes.py_object(deallocated))
```

Python 2.x 2.3

```
import ctypes, sys
deallocated = 12345
(ctypes.c_char * sys.getsizeof(deallocated)).from_address(id(deallocated))[:4] = '\x00' * 4
```

Dopo l'esecuzione, qualsiasi riferimento all'oggetto now deallocated farà sì che Python produca un comportamento non definito o crash - senza un traceback. C'era probabilmente una ragione per cui il garbage collector non ha rimosso quell'oggetto ...

Se deallocate `None` , viene visualizzato un messaggio speciale `Fatal Python error: deallocating None` prima dell'arresto anomalo.

Gestire la garbage collection

Esistono due approcci per influenzare quando viene eseguita una pulizia della memoria. Stanno influenzando la frequenza con cui viene eseguito il processo automatico e l'altro sta attivando manualmente una pulizia.

Il garbage collector può essere manipolato sintonizzando le soglie di raccolta che influiscono sulla frequenza di esecuzione del collector. Python utilizza un sistema di gestione della memoria basato sulla generazione. I nuovi oggetti vengono salvati nella nuova generazione - **generazione0** e con ciascuna raccolta sopravvissuta, gli oggetti vengono promossi alle generazioni precedenti. Dopo aver raggiunto l'ultima generazione - **generazione2** , non vengono più promossi.

Le soglie possono essere modificate utilizzando il seguente snippet:

```
import gc
gc.set_threshold(1000, 100, 10) # Values are just for demonstration purpose
```

Il primo argomento rappresenta la soglia per la raccolta di **generazione0** . Ogni volta che il numero di **allocazioni** supera il numero di **deallocazioni** per 1000, verrà chiamato il garbage collector.

Le generazioni precedenti non vengono pulite ad ogni analisi per ottimizzare il processo. Il secondo e il terzo argomento sono **facoltativi** e controllano la frequenza con cui vengono pulite le generazioni precedenti. Se la **generazione0 è** stata elaborata 100 volte senza eseguire la pulizia

di **generation1** , verrà **generata** la **generazione1** . Allo stesso modo, gli oggetti in **generation2** verranno elaborati solo quando quelli in **generation1** sono stati puliti 10 volte senza toccare **generation2** .

Un'istanza in cui l'impostazione manuale delle soglie è vantaggiosa è quando il programma assegna un sacco di piccoli oggetti senza deallocarli che conduce troppo spesso al garbage collector (ogni allocazione di oggetto **generation0_threshold**). Anche se il collezionista è piuttosto veloce, quando gira su un numero enorme di oggetti pone un problema di prestazioni. In ogni caso, non esiste una taglia adatta a tutte le strategie per la scelta delle soglie e il suo caso d'uso è affidabile.

L'attivazione manuale di una raccolta può essere eseguita come nel seguente snippet:

```
import gc
gc.collect()
```

La garbage collection viene automaticamente attivata in base al numero di allocazioni e deallocazioni, non sulla memoria consumata o disponibile. Di conseguenza, quando si lavora con oggetti di grandi dimensioni, la memoria potrebbe esaurirsi prima che venga avviata la pulizia automatica. Questo è un buon caso d'uso per chiamare manualmente il garbage collector.

Anche se è possibile, non è una pratica incoraggiata. Evitare perdite di memoria è l'opzione migliore. Ad ogni modo, nei grandi progetti l'individuazione della perdita di memoria può essere un compito ben preciso e l'attivazione manuale di una garbage collection può essere utilizzata come soluzione rapida fino a un ulteriore debugging.

Per i programmi con esecuzione prolungata, la garbage collection può essere attivata su base temporale o su base evento. Un esempio per il primo è un server Web che attiva una raccolta dopo un numero fisso di richieste. Per il futuro, un server Web che attiva una garbage collection quando viene ricevuto un determinato tipo di richiesta.

Non aspettare che la garbage collection sia pulita

Il fatto che la raccolta dei dati inutili venga eliminata non significa che è necessario attendere il ciclo di raccolta dei dati inutili per eseguire la pulizia.

In particolare, non è necessario attendere la garbage collection per chiudere handle di file, connessioni database e connessioni di rete aperte.

per esempio:

Nel seguente codice, si presuppone che il file verrà chiuso nel ciclo di garbage collection successivo, se f è l'ultimo riferimento al file.

```
>>> f = open("test.txt")
>>> del f
```

Un modo più esplicito per ripulire è chiamare `f.close()` . Puoi farlo ancora più elegante, utilizzando l'istruzione `with` , anche conosciuta come context manager :

```
>>> with open("test.txt") as f:
...     pass
...     # do something with f
>>> #now the f object still exists, but it is closed
```

L'istruzione `with` consente di indentare il codice sotto il file aperto. Ciò rende esplicito e più facile vedere per quanto tempo un file viene tenuto aperto. Inoltre, chiude sempre un file, anche se viene sollevata un'eccezione nel blocco `while` .

Leggi Raccolta dei rifiuti online: https://riptutorial.com/it/python/topic/2532/raccolta-dei-rifiuti

Capitolo 164: raggruppa per()

introduzione

In Python, il metodo `itertools.groupby()` consente agli sviluppatori di raggruppare i valori di una classe iterabile basata su una proprietà specificata in un altro insieme di valori iterabili.

Sintassi

- itertools.groupby (iterable, key = None o qualche funzione)

Parametri

Parametro	Dettagli
iterabile	Qualsiasi pitone iterabile
chiave	Funzione (criteri) su cui raggruppare l'iterabile

Osservazioni

groupby () è difficile ma una regola generale da tenere presente quando lo si usa è questa:

Ordinare sempre gli elementi che si desidera raggruppare con la stessa chiave che si desidera utilizzare per il raggruppamento

Si raccomanda al lettore di dare un'occhiata alla documentazione qui e vedere come viene spiegato usando una definizione di classe.

Examples

Esempio 1

Di 'che hai la corda

```
s = 'AAAABBBCCDAABBB'
```

e ti piacerebbe dividerlo in modo che tutte le 'A siano in una lista e quindi con tutte le' B 'e' C ', ecc. Potresti fare qualcosa di simile

```
s = 'AAAABBBCCDAABBB'
s_dict = {}
for i in s:
    if i not in s_dict.keys():
```

```
        s_dict[i] = [i]
    else:
        s_dict[i].append(i)
s_dict
```

Risultati in

```
{'A': ['A', 'A', 'A', 'A', 'A', 'A'],
 'B': ['B', 'B', 'B', 'B', 'B', 'B'],
 'C': ['C', 'C'],
 'D': ['D']}
```

Ma per il set di dati di grandi dimensioni si potrebbero costruire questi elementi in memoria. È qui che entra in gioco groupby ()

Potremmo ottenere lo stesso risultato in modo più efficiente facendo quanto segue

```
# note that we get a {key : value} pair for iterating over the items just like in python
dictionary
from itertools import groupby
s = 'AAAABBBCCDAABBB'
c = groupby(s)

dic = {}
for k, v in c:
    dic[k] = list(v)
dic
```

Risultati in

```
{'A': ['A', 'A'], 'B': ['B', 'B', 'B'], 'C': ['C', 'C'], 'D': ['D']}
```

Si noti che il numero di "A" nel risultato quando abbiamo usato il gruppo per è inferiore al numero effettivo di "A" nella stringa originale. Possiamo evitare quella perdita di informazioni ordinando gli elementi in s prima di passarli a c come mostrato di seguito

```
c = groupby(sorted(s))

dic = {}
for k, v in c:
    dic[k] = list(v)
dic
```

Risultati in

```
{'A': ['A', 'A', 'A', 'A', 'A', 'A'], 'B': ['B', 'B', 'B', 'B', 'B', 'B'], 'C': ['C', 'C'],
'D': ['D']}
```

Ora abbiamo tutte le nostre "A".

Esempio 2

Questo esempio illustra come viene scelta la chiave predefinita se non ne specificiamo alcuna

```
c = groupby(['goat', 'dog', 'cow', 1, 1, 2, 3, 11, 10, ('persons', 'man', 'woman')])
dic = {}
for k, v in c:
    dic[k] = list(v)
dic
```

Risultati in

```
{1: [1, 1],
 2: [2],
 3: [3],
 ('persons', 'man', 'woman'): [('persons', 'man', 'woman')],
 'cow': ['cow'],
 'dog': ['dog'],
 10: [10],
 11: [11],
 'goat': ['goat']}
```

Si noti qui che la tupla nel suo complesso conta come una chiave in questa lista

Esempio 3

Nota in questo esempio che mulato e cammello non appaiono nel nostro risultato. Viene mostrato solo l'ultimo elemento con la chiave specificata. L'ultimo risultato per c cancella in realtà due risultati precedenti. Ma guarda la nuova versione in cui ho i dati ordinati prima sulla stessa chiave.

```
list_things = ['goat', 'dog', 'donkey', 'mulato', 'cow', 'cat', ('persons', 'man', 'woman'), \
               'wombat', 'mongoose', 'malloo', 'camel']
c = groupby(list_things, key=lambda x: x[0])
dic = {}
for k, v in c:
    dic[k] = list(v)
dic
```

Risultati in

```
{'c': ['camel'],
 'd': ['dog', 'donkey'],
 'g': ['goat'],
 'm': ['mongoose', 'malloo'],
 'persons': [('persons', 'man', 'woman')],
 'w': ['wombat']}
```

Versione ordinata

```
list_things = ['goat', 'dog', 'donkey', 'mulato', 'cow', 'cat', ('persons', 'man', 'woman'), \
               'wombat', 'mongoose', 'malloo', 'camel']
sorted_list = sorted(list_things, key = lambda x: x[0])
print(sorted_list)
print()
c = groupby(sorted_list, key=lambda x: x[0])
```

```
dic = {}
for k, v in c:
    dic[k] = list(v)
dic
```

Risultati in

```
['cow', 'cat', 'camel', 'dog', 'donkey', 'goat', 'mulato', 'mongoose', 'malloo', ('persons',
'man', 'woman'), 'wombat']

{'c': ['cow', 'cat', 'camel'],
 'd': ['dog', 'donkey'],
 'g': ['goat'],
 'm': ['mulato', 'mongoose', 'malloo'],
 'persons': [('persons', 'man', 'woman')],
 'w': ['wombat']}
```

Esempio 4

In questo esempio vediamo cosa succede quando usiamo diversi tipi di iterabili.

```
things = [("animal", "bear"), ("animal", "duck"), ("plant", "cactus"), ("vehicle", "harley"),
\
          ("vehicle", "speed boat"), ("vehicle", "school bus")]
dic = {}
f = lambda x: x[0]
for key, group in groupby(sorted(things, key=f), f):
    dic[key] = list(group)
dic
```

Risultati in

```
{'animal': [('animal', 'bear'), ('animal', 'duck')],
 'plant': [('plant', 'cactus')],
 'vehicle': [('vehicle', 'harley'),
  ('vehicle', 'speed boat'),
  ('vehicle', 'school bus')]}
```

Questo esempio qui sotto è essenzialmente uguale a quello sopra. L'unica differenza è che ho modificato tutte le tuple alle liste.

```
things = [["animal", "bear"], ["animal", "duck"], ["vehicle", "harley"], ["plant", "cactus"],
\
          ["vehicle", "speed boat"], ["vehicle", "school bus"]]
dic = {}
f = lambda x: x[0]
for key, group in groupby(sorted(things, key=f), f):
    dic[key] = list(group)
dic
```

risultati

```
{'animal': [['animal', 'bear'], ['animal', 'duck']],
```

```
'plant': [['plant', 'cactus']],
'vehicle': [['vehicle', 'harley'],
 ['vehicle', 'speed boat'],
 ['vehicle', 'school bus']]}
```

Leggi raggruppa per() online: https://riptutorial.com/it/python/topic/8690/raggruppa-per--

Capitolo 165: Rappresentazioni di stringhe di istanze di classe: metodi __str__ e __repr__

Osservazioni

Una nota sull'implementazione di entrambi i metodi

Quando entrambi i metodi sono implementati, è piuttosto comune avere un metodo `__str__` che restituisca una rappresentazione umana (ad es. "Ace of Spaces") e `__repr__` restituisca una rappresentazione di `eval` -friendly.

In effetti, i documenti Python per `repr()` annotano esattamente questo:

> Per molti tipi, questa funzione fa un tentativo di restituire una stringa che produce un oggetto con lo stesso valore quando viene passato a eval (), altrimenti la rappresentazione è una stringa racchiusa tra parentesi angolari che contiene il nome del tipo dell'oggetto con informazioni aggiuntive spesso incluso il nome e l'indirizzo dell'oggetto.

Ciò significa che `__str__` potrebbe essere implementato per restituire qualcosa come "Ace of Spaces" come mostrato in precedenza, `__repr__` potrebbe essere implementato per restituire invece `Card('Spades', 1)`

Questa stringa potrebbe essere passata direttamente indietro in `eval` in un po 'di "round trip":

```
object -> string -> object
```

Un esempio di implementazione di tale metodo potrebbe essere:

```
def __repr__(self):
    return "Card(%s, %d)" % (self.suit, self.pips)
```

Gli appunti

[1] Questo output è specifico per l'implementazione. La stringa visualizzata proviene da cpython.

[2] Potresti aver già visto il risultato di questa divisione `str()` / `repr()` e non averlo conosciuto. Quando stringhe contenenti caratteri speciali come barre retroverse vengono convertite in stringhe tramite `str()` le barre retroverse appaiono così come sono (compaiono una volta). Quando vengono convertiti in stringhe tramite `repr()` (ad esempio, come elementi di una lista visualizzata),

i backslash sono sfuggiti e quindi appaiono due volte.

Examples

Motivazione

Quindi hai appena creato la tua prima classe in Python, una piccola e bella classe che incapsula una carta da gioco:

```
class Card:
    def __init__(self, suit, pips):
        self.suit = suit
        self.pips = pips
```

Altrove nel tuo codice, crei alcune istanze di questa classe:

```
ace_of_spades = Card('Spades', 1)
four_of_clubs = Card('Clubs',  4)
six_of_hearts = Card('Hearts', 6)
```

Hai persino creato un elenco di carte, per rappresentare una "mano":

```
my_hand = [ace_of_spades, four_of_clubs, six_of_hearts]
```

Ora, durante il debug, vuoi vedere come appare la tua mano, quindi fai ciò che viene naturale e scrivi:

```
print(my_hand)
```

Ma quello che ottieni è un mucchio di parole senza senso:

```
[<__main__.Card instance at 0x0000000002533788>,
 <__main__.Card instance at 0x00000000025B95C8>,
 <__main__.Card instance at 0x00000000025FF508>]
```

Confuso, provi solo a stampare una singola carta:

```
print(ace_of_spades)
```

E ancora, ottieni questa strana uscita:

```
<__main__.Card instance at 0x0000000002533788>
```

Non avere paura. Stiamo per risolvere questo problema.

Innanzitutto, è importante capire cosa sta succedendo qui. Quando hai scritto `print(ace_of_spades)` hai detto a Python che volevi che `print(ace_of_spades)` informazioni sull'istanza `Card` tuo codice chiama `ace_of_spades` . E per essere onesti, lo ha fatto.

Quell'output è composto da due bit importanti: il `type` dell'oggetto e l' `id` dell'oggetto. La seconda parte da sola (il numero esadecimale) è sufficiente per identificare univocamente l'oggetto al momento della chiamata di `print` . [1]

Ciò che è realmente accaduto è che hai chiesto a Python di "mettere in parole" l'essenza di quell'oggetto e poi mostrarlo a te. Una versione più esplicita dello stesso macchinario potrebbe essere:

```
string_of_card = str(ace_of_spades)
print(string_of_card)
```

Nella prima riga, provi a trasformare l'istanza di `Card` in una stringa, e nel secondo la visualizzi.

Il problema

Il problema che stai riscontrando deriva dal fatto che, mentre hai detto a Python tutto ciò che doveva sapere sulla classe `Card` per poter *creare le* carte, *non hai* detto come volevi convertire le istanze di `Card` in stringhe.

E poiché non sapeva, quando tu (implicitamente) hai scritto `str(ace_of_spades)` , ti ha dato ciò che hai visto, una rappresentazione generica dell'istanza `Card` .

La soluzione (parte 1)

Ma *possiamo* dire a Python come vogliamo che le istanze delle nostre classi personalizzate vengano convertite in stringhe. E il modo in cui lo facciamo è con il metodo `__str__` "dunder" (per doppia sottolineatura) o "magico".

Ogni volta che dici a Python di creare una stringa da un'istanza di classe, cercherà un metodo `__str__` sulla classe e lo chiamerà.

Considera la seguente versione aggiornata della nostra classe `Card` :

```
class Card:
    def __init__(self, suit, pips):
        self.suit = suit
        self.pips = pips

    def __str__(self):
        special_names = {1:'Ace', 11:'Jack', 12:'Queen', 13:'King'}

        card_name = special_names.get(self.pips, str(self.pips))

        return "%s of %s" % (card_name, self.suit)
```

Qui, abbiamo definito il metodo `__str__` sulla nostra classe `Card` che, dopo una semplice ricerca del dizionario per le face card, **restituisce** una stringa formattata, tuttavia decidiamo noi.

(Si noti che "restituisce" è in grassetto qui, per sottolineare l'importanza di restituire una stringa, e non semplicemente stamparla. La stampa può sembrare funzionare, ma poi si dovrebbe stampare la scheda quando si fa qualcosa come `str(ace_of_spades)` , senza nemmeno avere una chiamata di funzione di stampa nel programma principale. Per essere chiari, assicurarsi che `__str__` restituisca una stringa.).

Il metodo `__str__` è un metodo, quindi il primo argomento sarà `self` e non dovrebbe né accettare né passare argomenti addizionali.

Tornando al nostro problema di visualizzare la carta in un modo più user-friendly, se eseguiamo nuovamente:

```
ace_of_spades = Card('Spades', 1)
print(ace_of_spades)
```

Vedremo che il nostro output è molto migliore:

```
Ace of Spades
```

Così bello, abbiamo finito, giusto?

Beh, solo per coprire le nostre basi, ricontrolliamo che abbiamo risolto il primo problema che abbiamo incontrato, stampando l'elenco delle istanze di `Card` , la `hand` .

Quindi ricontrolliamo il seguente codice:

```
my_hand = [ace_of_spades, four_of_clubs, six_of_hearts]
print(my_hand)
```

E, con nostra sorpresa, otteniamo di nuovo quei divertenti codici esadecimali:

```
[<__main__.Card instance at 0x00000000026F95C8>,
 <__main__.Card instance at 0x000000000273F4C8>,
 <__main__.Card instance at 0x0000000002732E08>]
```

Cosa sta succedendo? Abbiamo detto a Python come volevamo visualizzare le nostre istanze di `Card` , perché apparentemente sembrava che dimenticasse?

La soluzione (parte 2)

Bene, il macchinario dietro le quinte è un po 'diverso quando Python vuole ottenere la rappresentazione di stringa degli elementi in una lista. Risulta che a Python non interessa `__str__` per questo scopo.

Invece, sembra per un metodo diverso, `__repr__` , e se *questo* non è trovato, cade di nuovo sul "cosa esadecimale". [2]

Quindi stai dicendo che devo fare due metodi per fare la stessa cosa? Uno per quando voglio

`print` *mia carta da solo e un altro quando è in una sorta di contenitore?*

No, ma prima diamo un'occhiata a come *sarebbe la* nostra classe se dovessimo implementare entrambi i metodi `__str__` e `__repr__` :

```
class Card:
    special_names = {1:'Ace', 11:'Jack', 12:'Queen', 13:'King'}

    def __init__(self, suit, pips):
        self.suit = suit
        self.pips = pips

    def __str__(self):
        card_name = Card.special_names.get(self.pips, str(self.pips))
        return "%s of %s (S)" % (card_name, self.suit)

    def __repr__(self):
        card_name = Card.special_names.get(self.pips, str(self.pips))
        return "%s of %s (R)" % (card_name, self.suit)
```

Qui, l'implementazione dei due metodi `__str__` e `__repr__` sono esattamente gli stessi, tranne per il fatto che, per distinguere tra i due metodi, `(S)` viene aggiunto alle stringhe restituite da `__str__` e `(R)` viene aggiunto alle stringhe restituite da `__repr__` .

Nota che proprio come il nostro metodo `__str__` , `__repr__` non accetta argomenti e **restituisce** una stringa.

Ora possiamo vedere quale metodo è responsabile per ogni caso:

```
ace_of_spades = Card('Spades', 1)
four_of_clubs = Card('Clubs',  4)
six_of_hearts = Card('Hearts', 6)

my_hand = [ace_of_spades, four_of_clubs, six_of_hearts]

print(my_hand)           # [Ace of Spades (R), 4 of Clubs (R), 6 of Hearts (R)]

print(ace_of_spades)     # Ace of Spades (S)
```

Come è stato trattato, il metodo `__str__` è stato chiamato quando abbiamo passato la nostra istanza `Card` per `print` e il metodo `__repr__` è stato chiamato quando abbiamo passato *un elenco delle nostre istanze* da `print` .

A questo punto vale la pena sottolineare che, come possiamo creare esplicitamente una stringa da un'istanza di classe personalizzata usando `str()` come abbiamo fatto in precedenza, possiamo anche creare esplicitamente una **rappresentazione** di **stringa** della nostra classe con una funzione incorporata chiamata `repr()` .

Per esempio:

```
str_card = str(four_of_clubs)
print(str_card)                     # 4 of Clubs (S)
```

```
repr_card = repr(four_of_clubs)
print(repr_card)                    # 4 of Clubs (R)
```

E inoltre, se definito, *potremmo* chiamare i metodi direttamente (anche se sembra un po 'poco chiaro ed inutile):

```
print(four_of_clubs.__str__())     # 4 of Clubs (S)

print(four_of_clubs.__repr__())    # 4 of Clubs (R)
```

Informazioni su quelle funzioni duplicate ...

Gli sviluppatori Python si sono resi conto, nel caso in cui volevi che stringhe identiche venissero restituite da `str()` e `repr()` , potresti dover duplicare funzionalmente i metodi, qualcosa che a nessuno piace.

Quindi, invece, c'è un meccanismo in atto per eliminare la necessità di ciò. Uno che ti ho fatto passare fino a questo punto. Si scopre che se una classe implementa il metodo `__repr__` *ma non* il metodo `__str__` e si passa un'istanza di quella classe a `str()` (sia implicitamente che esplicitamente), Python eseguirà il fallback sull'implementazione `__repr__` e lo userà.

Quindi, per essere chiari, considera la seguente versione della classe `Card` :

```
class Card:
    special_names = {1:'Ace', 11:'Jack', 12:'Queen', 13:'King'}

    def __init__(self, suit, pips):
        self.suit = suit
        self.pips = pips

    def __repr__(self):
        card_name = Card.special_names.get(self.pips, str(self.pips))
        return "%s of %s" % (card_name, self.suit)
```

Nota che questa versione implementa *solo* il metodo `__repr__` . Tuttavia, le chiamate a `str()` provocano la versione user-friendly:

```
print(six_of_hearts)            # 6 of Hearts  (implicit conversion)
print(str(six_of_hearts))       # 6 of Hearts  (explicit conversion)
```

come le chiamate a `repr()` :

```
print([six_of_hearts])          #[6 of Hearts] (implicit conversion)
print(repr(six_of_hearts))      # 6 of Hearts  (explicit conversion)
```

Sommario

Per consentire alle istanze di classe di "mostrarsi" in modo `__repr__` , ti consigliamo di implementare almeno il metodo `__repr__` della tua classe. Se la memoria serve, durante una `__repr__` Raymond Hettinger ha affermato che garantire l'implementazione di classi `__repr__` è una delle prime cose che cerca mentre si eseguono le revisioni del codice Python, e ormai dovrebbe essere chiaro il perché. La quantità di informazioni che *potresti* aver aggiunto alle istruzioni di debug, ai rapporti sugli arresti anomali o ai file di registro con un metodo semplice è schiacciante se confrontata con le informazioni irrisorie e spesso meno utili (tipo, id) fornite per impostazione predefinita.

Se desideri rappresentazioni *diverse* per quando, ad esempio, all'interno di un contenitore, dovrai implementare entrambi i metodi `__repr__` e `__str__` . (Ulteriori informazioni su come si potrebbero usare questi due metodi in modo diverso sotto).

Entrambi i metodi implementati, eval-round-trip style __repr __ ()

```
class Card:
    special_names = {1:'Ace', 11:'Jack', 12:'Queen', 13:'King'}

    def __init__(self, suit, pips):
        self.suit = suit
        self.pips = pips

    # Called when instance is converted to a string via str()
    # Examples:
    #   print(card1)
    #   print(str(card1)
    def __str__(self):
        card_name = Card.special_names.get(self.pips, str(self.pips))
        return "%s of %s" % (card_name, self.suit)

    # Called when instance is converted to a string via repr()
    # Examples:
    #   print([card1, card2, card3])
    #   print(repr(card1))
    def __repr__(self):
        return "Card(%s, %d)" % (self.suit, self.pips)
```

Leggi Rappresentazioni di stringhe di istanze di classe: metodi __str__ e __repr__ online: https://riptutorial.com/it/python/topic/4845/rappresentazioni-di-stringhe-di-istanze-di-classe--metodi---str---e---repr--

Capitolo 166: Raspare Web con Python

introduzione

Lo scraping Web è un processo programmatico automatizzato attraverso il quale i dati possono essere costantemente "raschiati" fuori dalle pagine Web. Conosciuto anche come screen scraping o web harvesting, il web scraping può fornire dati istantanei da qualsiasi pagina web accessibile al pubblico. Su alcuni siti Web, il web scraping potrebbe essere illegale.

Osservazioni

Utili pacchetti Python per lo scraping web (in ordine alfabetico)

Fare richieste e raccogliere dati

`requests`

Un semplice ma potente pacchetto per fare richieste HTTP.

`requests-cache`

Memorizzazione nella cache delle `requests` ; i dati di memorizzazione nella cache sono molto utili. In fase di sviluppo, significa che puoi evitare di colpire un sito inutilmente. Quando si esegue una vera raccolta, significa che se il tuo raschietto si blocca per qualche motivo (forse non hai gestito alcuni contenuti insoliti sul sito ...? Forse il sito è andato giù ...?) Puoi ripetere la raccolta molto velocemente da dove eri rimasto.

`scrapy`

Utile per creare web crawler, dove è necessario qualcosa di più potente dell'uso di `requests` e iterazione attraverso le pagine.

`selenium`

Collegamenti Python per Selenium WebDriver, per l'automazione del browser. L'utilizzo delle `requests` per effettuare direttamente richieste HTTP è spesso più semplice per il recupero di pagine Web. Tuttavia, questo rimane uno strumento utile quando non è possibile replicare il comportamento desiderato di un sito utilizzando solo le `requests` , in particolare quando è richiesto JavaScript per il rendering di elementi in una pagina.

Analisi HTML

BeautifulSoup

Interrogare i documenti HTML e XML, utilizzando un numero di parser diversi (Parser HTML incorporato di Python, `html5lib` , `lxml` o `lxml.html`)

lxml

Elabora HTML e XML. Può essere usato per interrogare e selezionare il contenuto da documenti HTML tramite selettori CSS e XPath.

Examples

Esempio di base di utilizzo di richieste e lxml per raschiare alcuni dati

```
# For Python 2 compatibility.
from __future__ import print_function

import lxml.html
import requests


def main():
    r = requests.get("https://httpbin.org")
    html_source = r.text
    root_element = lxml.html.fromstring(html_source)
    # Note root_element.xpath() gives a *list* of results.
    # XPath specifies a path to the element we want.
    page_title = root_element.xpath('/html/head/title/text()')[0]
    print(page_title)

if __name__ == '__main__':
    main()
```

Mantenimento della sessione di web scraping con le richieste

È consigliabile mantenere una sessione di web-scraping per mantenere i cookie e altri parametri. Inoltre, può comportare un *miglioramento delle prestazioni* poiché `requests.Session` riutilizza la connessione TCP sottostante a un host:

```
import requests

with requests.Session() as session:
    # all requests through session now have User-Agent header set
    session.headers = {'User-Agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_4)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/51.0.2704.103 Safari/537.36'}

    # set cookies
    session.get('http://httpbin.org/cookies/set?key=value')

    # get cookies
    response = session.get('http://httpbin.org/cookies')
    print(response.text)
```

Raschiare usando il framework Scrapy

Per prima cosa devi creare un nuovo progetto Scrapy. Inserisci una directory in cui desideri memorizzare il codice ed esegui:

```
scrapy startproject projectName
```

Per raschiare abbiamo bisogno di un ragno. Gli spider definiscono come verrà raschiato un determinato sito. Ecco il codice per uno spider che segue i link alle domande più votate su StackOverflow e cancella alcuni dati da ciascuna pagina (fonte):

```
import scrapy

class StackOverflowSpider(scrapy.Spider):
    name = 'stackoverflow'  # each spider has a unique name
    start_urls = ['http://stackoverflow.com/questions?sort=votes']  # the parsing starts from
a specific set of urls

    def parse(self, response):  # for each request this generator yields, its response is sent
to parse_question
        for href in response.css('.question-summary h3 a::attr(href)'):  # do some scraping
stuff using css selectors to find question urls
            full_url = response.urljoin(href.extract())
            yield scrapy.Request(full_url, callback=self.parse_question)

    def parse_question(self, response):
        yield {
            'title': response.css('h1 a::text').extract_first(),
            'votes': response.css('.question .vote-count-post::text').extract_first(),
            'body': response.css('.question .post-text').extract_first(),
            'tags': response.css('.question .post-tag::text').extract(),
            'link': response.url,
        }
```

Salva le tue classi spider nella directory `projectName\spiders` . In questo caso - `projectName\spiders\stackoverflow_spider.py` .

Ora puoi usare il tuo ragno. Ad esempio, prova a eseguire (nella directory del progetto):

```
scrapy crawl stackoverflow
```

Modifica agente utente Scrapy

A volte l'agente utente Scrapy predefinito (`"Scrapy/VERSION (+http://scrapy.org)"`) viene bloccato dall'host. Per cambiare l'agente utente predefinito, apri **settings.py** , decommenta e modifica la riga seguente per quello che vuoi.

```
#USER_AGENT = 'projectName (+http://www.yourdomain.com)'
```

Per esempio

```
USER_AGENT = 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_4) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/51.0.2704.103 Safari/537.36'
```

Raschiare usando BeautifulSoup4

```
from bs4 import BeautifulSoup
import requests

# Use the requests module to obtain a page
res = requests.get('https://www.codechef.com/problems/easy')

# Create a BeautifulSoup object
page = BeautifulSoup(res.text, 'lxml')   # the text field contains the source of the page

# Now use a CSS selector in order to get the table containing the list of problems
datatable_tags = page.select('table.dataTable')  # The problems are in the <table> tag,
                                                 # with class "dataTable"
# We extract the first tag from the list, since that's what we desire
datatable = datatable_tags[0]
# Now since we want problem names, they are contained in <b> tags, which are
# directly nested under <a> tags
prob_tags = datatable.select('a > b')
prob_names = [tag.getText().strip() for tag in prob_tags]

print prob_names
```

Raschiatura usando Selenium WebDriver

Alcuni siti Web non amano essere raschiati. In questi casi potrebbe essere necessario simulare un utente reale che lavora con un browser. Selenium lancia e controlla un browser web.

```
from selenium import webdriver

browser = webdriver.Firefox()  # launch firefox browser

browser.get('http://stackoverflow.com/questions?sort=votes')  # load url

title = browser.find_element_by_css_selector('h1').text  # page title (first h1 element)

questions = browser.find_elements_by_css_selector('.question-summary')  # question list

for question in questions:  # iterate over questions
    question_title = question.find_element_by_css_selector('.summary h3 a').text
    question_excerpt = question.find_element_by_css_selector('.summary .excerpt').text
    question_vote = question.find_element_by_css_selector('.stats .vote .votes .vote-count-
post').text

    print "%s\n%s\n%s votes\n-----------\n" % (question_title, question_excerpt,
question_vote)
```

Il selenio può fare molto di più. Può modificare i cookie del browser, compilare moduli, simulare clic del mouse, acquisire schermate di pagine Web ed eseguire JavaScript personalizzato.

Download semplice di contenuti Web con urllib.request

Il modulo di libreria standard `urllib.request` può essere utilizzato per scaricare contenuti web:

```
from urllib.request import urlopen

response = urlopen('http://stackoverflow.com/questions?sort=votes')
data = response.read()

# The received bytes should usually be decoded according the response's character set
encoding = response.info().get_content_charset()
html = data.decode(encoding)
```

Un modulo simile è disponibile anche in Python 2 .

Raschiando con arricciatura

importazioni:

```
from subprocess import Popen, PIPE
from lxml import etree
from io import StringIO
```

Download:

```
user_agent = 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_6) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/55.0.2883.95 Safari/537.36'
url = 'http://stackoverflow.com'
get = Popen(['curl', '-s', '-A', user_agent, url], stdout=PIPE)
result = get.stdout.read().decode('utf8')
```

`-s` : download silenzioso

`-A` : flag agente utente

di analisi:

```
tree = etree.parse(StringIO(result), etree.HTMLParser())
divs = tree.xpath('//div')
```

Leggi Raspare Web con Python online: https://riptutorial.com/it/python/topic/1792/raspare-web-con-python

Capitolo 167: Registrazione

Examples

Introduzione alla registrazione di Python

Questo modulo definisce funzioni e classi che implementano un sistema di registrazione eventi flessibile per applicazioni e librerie.

Il vantaggio principale di avere l'API di registrazione fornita da un modulo di libreria standard è che tutti i moduli Python possono partecipare alla registrazione, quindi il registro delle applicazioni può includere i propri messaggi integrati con i messaggi di moduli di terze parti.

Quindi iniziamo:

Esempio di configurazione direttamente nel codice

```
import logging

logger = logging.getLogger()
handler = logging.StreamHandler()
formatter = logging.Formatter(
        '%(asctime)s %(name)-12s %(levelname)-8s %(message)s')
handler.setFormatter(formatter)
logger.addHandler(handler)
logger.setLevel(logging.DEBUG)

logger.debug('this is a %s test', 'debug')
```

Esempio di uscita:

```
2016-07-26 18:53:55,332 root         DEBUG    this is a debug test
```

Esempio di configurazione tramite un file INI

Supponendo che il file sia denominato logging_config.ini. Ulteriori dettagli sul formato del file si trovano nella sezione di configurazione della registrazione del tutorial di registrazione .

```
[loggers]
keys=root

[handlers]
keys=stream_handler

[formatters]
keys=formatter

[logger_root]
level=DEBUG
handlers=stream_handler
```

```
[handler_stream_handler]
class=StreamHandler
level=DEBUG
formatter=formatter
args=(sys.stderr,)

[formatter_formatter]
format=%(asctime)s %(name)-12s %(levelname)-8s %(message)s
```

Quindi usa `logging.config.fileConfig()` nel codice:

```
import logging
from logging.config import fileConfig

fileConfig('logging_config.ini')
logger = logging.getLogger()
logger.debug('often makes a very good meal of %s', 'visiting tourists')
```

Esempio di configurazione tramite un dizionario

A partire da Python 2.7, è possibile utilizzare un dizionario con i dettagli di configurazione. PEP 391 contiene un elenco degli elementi obbligatori e facoltativi nel dizionario di configurazione.

```
import logging
from logging.config import dictConfig

logging_config = dict(
    version = 1,
    formatters = {
        'f': {'format':
              '%(asctime)s %(name)-12s %(levelname)-8s %(message)s'}
        },
    handlers = {
        'h': {'class': 'logging.StreamHandler',
              'formatter': 'f',
              'level': logging.DEBUG}
        },
    root = {
        'handlers': ['h'],
        'level': logging.DEBUG,
        },
)

dictConfig(logging_config)

logger = logging.getLogger()
logger.debug('often makes a very good meal of %s', 'visiting tourists')
```

Registrazione delle eccezioni

Se si desidera registrare eccezioni, è possibile e utilizzare il metodo `logging.exception(msg)` :

```
>>> import logging
>>> logging.basicConfig()
>>> try:
...     raise Exception('foo')
```

```
... except:
...     logging.exception('bar')
...
ERROR:root:bar
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
Exception: foo
```

Non passare l'eccezione come argomento:

Poiché `logging.exception(msg)` aspetta un `msg` arg, è un errore comune passare l'eccezione alla chiamata di registrazione come questa:

```
>>> try:
...     raise Exception('foo')
... except Exception as e:
...     logging.exception(e)
...
ERROR:root:foo
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
Exception: foo
```

Mentre potrebbe sembrare che questa sia la cosa giusta da fare in un primo momento, è in realtà problematico a causa del motivo per cui le eccezioni e le varie codifiche funzionano insieme nel modulo di registrazione:

```
>>> try:
...     raise Exception(u'föö')
... except Exception as e:
...     logging.exception(e)
...
Traceback (most recent call last):
  File "/.../python2.7/logging/__init__.py", line 861, in emit
    msg = self.format(record)
  File "/.../python2.7/logging/__init__.py", line 734, in format
    return fmt.format(record)
  File "/.../python2.7/logging/__init__.py", line 469, in format
    s = self._fmt % record.__dict__
UnicodeEncodeError: 'ascii' codec can't encode characters in position 1-2: ordinal not in
range(128)
Logged from file <stdin>, line 4
```

Cercando di registrare un'eccezione che contiene caratteri Unicode, in questo modo fallirà miseramente . Nasconderà lo stacktrace dell'eccezione originale sovrascrivendolo con uno nuovo che viene generato durante la formattazione della tua chiamata `logging.exception(e)` .

Ovviamente, nel tuo codice, potresti essere a conoscenza della codifica delle eccezioni. Tuttavia, le librerie di terze parti potrebbero gestirle in un modo diverso.

Uso corretto:

Se invece dell'eccezione si passa un messaggio e si lascia che Python faccia la sua magia, funzionerà:

```
>>> try:
...     raise Exception(u'föö')
... except Exception as e:
...     logging.exception('bar')
...
ERROR:root:bar
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
Exception: f\xf6\xf6
```

Come puoi vedere, in questo caso non usiamo `e` , la chiamata a `logging.exception(...)` formatta magicamente l'eccezione più recente.

Registrazione delle eccezioni con livelli di log non ERROR

Se si desidera registrare un'eccezione con un altro livello di registro diverso da ERRORE, è possibile utilizzare l'argomento `exc_info` dei logger predefiniti:

```
logging.debug('exception occurred', exc_info=1)
logging.info('exception occurred', exc_info=1)
logging.warning('exception occurred', exc_info=1)
```

Accedere al messaggio dell'eccezione

Siate consapevoli che le librerie disponibili potrebbero generare eccezioni con i messaggi come qualsiasi codice unicode o stringhe di byte (utf-8 se siete fortunati). Se hai davvero bisogno di accedere al testo di un'eccezione, l'unico modo affidabile, che funzionerà sempre, è usare `repr(e)` o la formattazione della stringa `%r` :

```
>>> try:
...     raise Exception(u'föö')
... except Exception as e:
...     logging.exception('received this exception: %r' % e)
...
ERROR:root:received this exception: Exception(u'f\xf6\xf6',)
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
Exception: f\xf6\xf6
```

Leggi Registrazione online: https://riptutorial.com/it/python/topic/4081/registrazione

Capitolo 168: Ricerca

Osservazioni

Tutti gli algoritmi di ricerca su iterabili contenenti `n` elementi hanno complessità `O(n)` . Solo algoritmi specializzati come `bisect.bisect_left()` possono essere più veloci con `O(log(n))` complessità.

Examples

Ottenere l'indice per le stringhe: str.index (), str.rindex () e str.find (), str.rfind ()

`String` anche un metodo `index` ma anche opzioni più avanzate e lo `str.find` aggiuntivo. Per entrambi c'è un metodo *inverso* complementare.

```
astring = 'Hello on StackOverflow'
astring.index('o')  # 4
astring.rindex('o') # 20

astring.find('o')   # 4
astring.rfind('o')  # 20
```

La differenza tra `index` / `rindex` e `find` / `rfind` è cosa succede se la sottostringa non viene trovata nella stringa:

```
astring.index('q') # ValueError: substring not found
astring.find('q')  # -1
```

Tutti questi metodi consentono un indice di inizio e fine:

```
astring.index('o', 5)    # 6
astring.index('o', 6)    # 6 - start is inclusive
astring.index('o', 5, 7) # 6
astring.index('o', 5, 6) #  - end is not inclusive
```

> ValueError: sottostringa non trovata

```
astring.rindex('o', 20) # 20
astring.rindex('o', 19) # 20 - still from left to right

astring.rindex('o', 4, 7) # 6
```

Alla ricerca di un elemento

Tutte le raccolte predefinite in Python implementano un modo per controllare l'appartenenza agli elementi usando `in` .

Elenco

```
alist = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
5 in alist   # True
10 in alist  # False
```

tuple

```
atuple = ('0', '1', '2', '3', '4')
4 in atuple    # False
'4' in atuple  # True
```

Stringa

```
astring = 'i am a string'
'a' in astring   # True
'am' in astring  # True
'I' in astring   # False
```

Impostato

```
aset = {(10, 10), (20, 20), (30, 30)}
(10, 10) in aset  # True
10 in aset        # False
```

dict

`dict` è un po 'speciale: la normale `in` solo controlli le *chiavi.* Se vuoi cercare nei *valori* devi specificarlo. Lo stesso se vuoi cercare coppie di *valori-chiave* .

```
adict = {0: 'a', 1: 'b', 2: 'c', 3: 'd'}
1 in adict                 # True   - implicitly searches in keys
'a' in adict               # False
2 in adict.keys()          # True   - explicitly searches in keys
'a' in adict.values()      # True   - explicitly searches in values
(0, 'a') in adict.items()  # True   - explicitly searches key/value pairs
```

Ottenere l'elenco degli indici e le tuple: list.index (), tuple.index ()

`list` e `tuple` hanno un metodo- `index` per ottenere la posizione dell'elemento:

```
alist = [10, 16, 26, 5, 2, 19, 105, 26]
# search for 16 in the list
alist.index(16) # 1
alist[1]        # 16

alist.index(15)
```

ValueError: 15 non è in elenco

Ma restituisce solo la posizione del primo elemento trovato:

```
atuple = (10, 16, 26, 5, 2, 19, 105, 26)
atuple.index(26)   # 2
atuple[2]          # 26
atuple[7]          # 26 - is also 26!
```

Ricerca di chiavi (s) per un valore in dict

`dict` non ha un metodo incorporato per cercare un valore o una chiave perché i *dizionari* non sono ordinati. È possibile creare una funzione che ottiene la chiave (o le chiavi) per un valore specificato:

```
def getKeysForValue(dictionary, value):
    foundkeys = []
    for keys in dictionary:
        if dictionary[key] == value:
            foundkeys.append(key)
    return foundkeys
```

Questo potrebbe anche essere scritto come una comprensione di lista equivalente:

```
def getKeysForValueComp(dictionary, value):
    return [key for key in dictionary if dictionary[key] == value]
```

Se ti interessa solo una chiave trovata:

```
def getOneKeyForValue(dictionary, value):
    return next(key for key in dictionary if dictionary[key] == value)
```

Le prime due funzioni restituiranno un `list` di tutte le `keys` con il valore specificato:

```
adict = {'a': 10, 'b': 20, 'c': 10}
getKeysForValue(adict, 10)     # ['c', 'a'] - order is random could as well be ['a', 'c']
getKeysForValueComp(adict, 10) # ['c', 'a'] - dito
getKeysForValueComp(adict, 20) # ['b']
getKeysForValueComp(adict, 25) # []
```

L'altro restituirà solo una chiave:

```
getOneKeyForValue(adict, 10)   # 'c'  - depending on the circumstances this could also be 'a'
getOneKeyForValue(adict, 20)   # 'b'
```

e `StopIteration` una `StopIteration` - `Exception` se il valore non è nel `dict` :

```
getOneKeyForValue(adict, 25)
```

StopIteration

Ottenere l'indice per le sequenze ordinate: bisect.bisect_left ()

Sequenze ordinate consentono l'uso di algoritmi di ricerca più veloci: `bisect.bisect_left()` [1] :

```
import bisect

def index_sorted(sorted_seq, value):
    """Locate the leftmost value exactly equal to x or raise a ValueError"""
    i = bisect.bisect_left(sorted_seq, value)
    if i != len(sorted_seq) and sorted_seq[i] == value:
        return i
    raise ValueError

alist = [i for i in range(1, 100000, 3)] # Sorted list from 1 to 100000 with step 3
index_sorted(alist, 97285) # 32428
index_sorted(alist, 4)     # 1
index_sorted(alist, 97286)
```

ValueError

Per **sequenze ordinate** molto grandi il guadagno di velocità può essere piuttosto elevato. In caso di prima ricerca approssimativamente 500 volte più veloce:

```
%timeit index_sorted(alist, 97285)
# 100000 loops, best of 3: 3 µs per loop
%timeit alist.index(97285)
# 1000 loops, best of 3: 1.58 ms per loop
```

Mentre è un po 'più lento se l'elemento è uno dei primi:

```
%timeit index_sorted(alist, 4)
# 100000 loops, best of 3: 2.98 µs per loop
%timeit alist.index(4)
# 1000000 loops, best of 3: 580 ns per loop
```

Ricerca di sequenze nidificate

La ricerca in sequenze annidate come un `list` di `tuple` richiede un approccio simile alla ricerca delle chiavi per i valori in `dict` ma richiede funzioni personalizzate.

L'indice della sequenza più esterna se il valore è stato trovato nella sequenza:

```
def outer_index(nested_sequence, value):
    return next(index for index, inner in enumerate(nested_sequence)
                      for item in inner
                      if item == value)

alist_of_tuples = [(4, 5, 6), (3, 1, 'a'), (7, 0, 4.3)]
outer_index(alist_of_tuples, 'a')  # 1
outer_index(alist_of_tuples, 4.3)  # 2
```

o l'indice della sequenza esterna e interna:

```
def outer_inner_index(nested_sequence, value):
    return next((oindex, iindex) for oindex, inner in enumerate(nested_sequence)
```

```
                                for iindex, item in enumerate(inner)
                                if item == value)

outer_inner_index(alist_of_tuples, 'a') # (1, 2)
alist_of_tuples[1][2]  # 'a'

outer_inner_index(alist_of_tuples, 7)   # (2, 0)
alist_of_tuples[2][0]  # 7
```

In generale (*non sempre*) usando `next` e **un'espressione di generatore** con condizioni per trovare la prima occorrenza del valore cercato è l'approccio più efficiente.

Ricerca in classi personalizzate: __contains__ e __iter__

Per consentire l'uso di `in` per classi personalizzate, la classe deve fornire il metodo magico `__contains__` o, in mancanza, un `__iter__` -method.

Supponiamo di avere una classe contenente un `list` di `list` s:

```
class ListList:
    def __init__(self, value):
        self.value = value
        # Create a set of all values for fast access
        self.setofvalues = set(item for sublist in self.value for item in sublist)

    def __iter__(self):
        print('Using __iter__.')
        # A generator over all sublist elements
        return (item for sublist in self.value for item in sublist)

    def __contains__(self, value):
        print('Using __contains__.')
        # Just lookup if the value is in the set
        return value in self.setofvalues

        # Even without the set you could use the iter method for the contains-check:
        # return any(item == value for item in iter(self))
```

L'utilizzo del test di appartenenza è possibile utilizzando `in` :

```
a = ListList([[1,1,1],[0,1,1],[1,5,1]])
10 in a    # False
# Prints: Using __contains__.
5 in a     # True
# Prints: Using __contains__.
```

anche dopo aver eliminato il metodo `__contains__` :

```
del ListList.__contains__
5 in a     # True
# Prints: Using __iter__.
```

Nota: il looping `in` (come in `for i in a`) userà sempre `__iter__` anche se la classe implementa un metodo `__contains__` .

Leggi Ricerca online: https://riptutorial.com/it/python/topic/350/ricerca

Capitolo 169: Richieste di richieste Python

introduzione

Documentazione per il modulo Richieste Python nel contesto del metodo POST HTTP e della relativa funzione Richieste

Examples

Post semplice

```
from requests import post

foo = post('http://httpbin.org/post', data = {'key':'value'})
```

Effettuerà una semplice operazione POST HTTP. I dati pubblicati possono essere i formati più interni, tuttavia le coppie di valori chiave sono prevalenti.

intestazioni

Le intestazioni possono essere visualizzate:

```
print(foo.headers)
```

Un esempio di risposta:

```
{'Content-Length': '439', 'X-Processed-Time': '0.000802993774414', 'X-Powered-By': 'Flask',
'Server': 'meinheld/0.6.1', 'Connection': 'keep-alive', 'Via': '1.1 vegur', 'Access-Control-
Allow-Credentials': 'true', 'Date': 'Sun, 21 May 2017 20:56:05 GMT', 'Access-Control-Allow-
Origin': '*', 'Content-Type': 'application/json'}
```

Le intestazioni possono anche essere preparate prima della pubblicazione:

```
headers = {'Cache-Control':'max-age=0',
        'Upgrade-Insecure-Requests':'1',
        'User-Agent':'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/54.0.2840.99 Safari/537.36',
        'Content-Type':'application/x-www-form-urlencoded',
        'Accept':'text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8',
        'Referer':'https://www.groupon.com/signup',
        'Accept-Encoding':'gzip, deflate, br',
        'Accept-Language':'es-ES,es;q=0.8'
        }

 foo = post('http://httpbin.org/post', headers=headers, data = {'key':'value'})
```

Codifica

La codifica può essere impostata e visualizzata più o meno allo stesso modo:

```
 print(foo.encoding)

'utf-8'

foo.encoding = 'ISO-8859-1'
```

Verifica SSL

Le richieste per impostazione predefinita convalida i certificati SSL dei domini. Questo può essere ignorato:

```
foo = post('http://httpbin.org/post', data = {'key':'value'}, verify=False)
```

reindirizzamento

Tutto il reindirizzamento verrà seguito (ad esempio da http a https) anche questo può essere modificato:

```
foo = post('http://httpbin.org/post', data = {'key':'value'}, allow_redirects=False)
```

Se l'operazione di post è stata reindirizzata, è possibile accedere a questo valore:

```
print(foo.url)
```

È possibile visualizzare una cronologia completa dei reindirizzamenti:

```
print(foo.history)
```

Form Dati codificati

```
from requests import post

payload = {'key1' : 'value1',
           'key2' : 'value2'
           }

foo = post('http://httpbin.org/post', data=payload)
```

Per passare i dati codificati del modulo con l'operazione di post, i dati devono essere strutturati come dizionario e forniti come parametro dei dati.

Se i dati non vogliono essere codificati in forma, basta passare una stringa o un intero al parametro dei dati.

Fornisci il dizionario al parametro json per Requests per formattare automaticamente i dati:

```
from requests import post

payload = {'key1' : 'value1', 'key2' : 'value2'}
```

```
foo = post('http://httpbin.org/post', json=payload)
```

Upload di file

Con il modulo Richieste, è solo necessario fornire un handle di file in contrapposizione ai contenuti recuperati con `.read()` :

```
from requests import post

files = {'file' : open('data.txt', 'rb')}

foo = post('http://http.org/post', files=files)
```

È inoltre possibile impostare nome file, content_type e intestazioni:

```
files = {'file': ('report.xls', open('report.xls', 'rb'), 'application/vnd.ms-excel',
{'Expires': '0'})}

foo = requests.post('http://httpbin.org/post', files=files)
```

Le stringhe possono anche essere inviate come file, a condizione che vengano fornite come parametro dei `files` .

File multipli

Più file possono essere forniti più o meno allo stesso modo di un file:

```
multiple_files = [
    ('images', ('foo.png', open('foo.png', 'rb'), 'image/png')),
    ('images', ('bar.png', open('bar.png', 'rb'), 'image/png'))]

foo = post('http://httpbin.org/post', files=multiple_files)
```

risposte

I codici di risposta possono essere visualizzati da un'operazione di post:

```
from requests import post

foo = post('http://httpbin.org/post', data={'data' : 'value'})
print(foo.status_code)
```

Dati restituiti

Accesso ai dati che vengono restituiti:

```
foo = post('http://httpbin.org/post', data={'data' : 'value'})
print(foo.text)
```

Risposte non elaborate

Nei casi in cui è necessario accedere all'url di risposta urllib3.HTTPResponse oggetto, questo può essere fatto come segue:

```
foo = post('http://httpbin.org/post', data={'data' : 'value'})
res = foo.raw

print(res.read())
```

Autenticazione

Autenticazione HTTP semplice

L'autenticazione HTTP semplice può essere ottenuta con quanto segue:

```
from requests import post

foo = post('http://natas0.natas.labs.overthewire.org', auth=('natas0', 'natas0'))
```

Questa è tecnicamente una mano breve per quanto segue:

```
from requests import post
from requests.auth import HTTPBasicAuth

foo = post('http://natas0.natas.labs.overthewire.org', auth=HTTPBasicAuth('natas0', 'natas0'))
```

Autenticazione del digest HTTP

L'autenticazione del digest HTTP viene eseguita in modo molto simile, Requests fornisce un oggetto diverso per questo:

```
from requests import post
from requests.auth import HTTPDigestAuth

foo = post('http://natas0.natas.labs.overthewire.org', auth=HTTPDigestAuth('natas0',
'natas0'))
```

Autenticazione personalizzata

In alcuni casi i meccanismi di autenticazione incorporati potrebbero non essere sufficienti, immagina questo esempio:

Un server è configurato per accettare l'autenticazione se il mittente ha la stringa utente-agente corretta, un determinato valore di intestazione e fornisce le credenziali corrette tramite l'autenticazione di base HTTP. Per ottenere ciò è necessario preparare una classe di autenticazione personalizzata, sottoclasse AuthBase, che è la base per le implementazioni di autenticazione di Requests:

```
from requests.auth import AuthBase
from requests.auth import _basic_auth_str
from requests._internal_utils import to_native_string
```

```
class CustomAuth(AuthBase):

    def __init__(self, secret_header, user_agent , username, password):
        # setup any auth-related data here
        self.secret_header =  secret_header
        self.user_agent = user_agent
        self.username = username
        self.password = password

    def __call__(self, r):
        # modify and return the request
        r.headers['X-Secret'] = self.secret_header
        r.headers['User-Agent'] = self.user_agent
        r.headers['Authorization'] = _basic_auth_str(self.username, self.password)

        return r
```

Questo può quindi essere utilizzato con il seguente codice:

```
foo = get('http://test.com/admin', auth=CustomAuth('SecretHeader', 'CustomUserAgent', 'user',
'password' ))
```

Proxy

Ogni operazione POST richiesta può essere configurata per utilizzare i proxy di rete

Proxy HTTP / S

```
from requests import post

proxies = {
  'http': 'http://192.168.0.128:3128',
  'https': 'http://192.168.0.127:1080',
   }

foo = requests.post('http://httpbin.org/post', proxies=proxies)
```

L'autenticazione di base HTTP può essere fornita in questo modo:

```
proxies = {'http': 'http://user:pass@192.168.0.128:312'}
foo = requests.post('http://httpbin.org/post', proxies=proxies)
```

Proxy SOCKS

L'uso di proxy di calze richiede `requests[socks]` dipendenza di terzi `requests[socks]` , una volta che i proxy di calze installati vengono utilizzati in modo molto simile a HTTPBasicAuth:

```
proxies = {
'http': 'socks5://user:pass@host:port',
'https': 'socks5://user:pass@host:port'
}

foo = requests.post('http://httpbin.org/post', proxies=proxies)
```

Leggi Richieste di richieste Python online: https://riptutorial.com/it/python/topic/10021/richieste-di-richieste-python

Capitolo 170: Riconoscimento ottico dei caratteri

introduzione

Il riconoscimento ottico dei caratteri sta convertendo le immagini del testo in testo reale. In questi esempi trovi modi per usare OCR in python.

Examples

PyTesseract

PyTesseract è un pacchetto python in sviluppo per OCR.

Usare PyTesseract è piuttosto semplice:

```
try:
    import Image
except ImportError:
    from PIL import Image

import pytesseract

    #Basic OCR
    print(pytesseract.image_to_string(Image.open('test.png')))

    #In French
    print(pytesseract.image_to_string(Image.open('test-european.jpg'), lang='fra'))
```

PyTesseract è open source e può essere trovato qui .

PyOCR

Un altro modulo di qualche uso è `PyOCR` , il cui codice sorgente è qui .

Anche semplice da usare e ha più funzionalità di `PyTesseract` .

Per inizializzare:

```
from PIL import Image
import sys

import pyocr
import pyocr.builders

tools = pyocr.get_available_tools()
# The tools are returned in the recommended order of usage
tool = tools[0]
```

```
langs = tool.get_available_languages()
lang = langs[0]
# Note that languages are NOT sorted in any way. Please refer
# to the system locale settings for the default language
# to use.
```

E alcuni esempi di utilizzo:

```
txt = tool.image_to_string(
    Image.open('test.png'),
    lang=lang,
    builder=pyocr.builders.TextBuilder()
)
# txt is a Python string

word_boxes = tool.image_to_string(
    Image.open('test.png'),
    lang="eng",
    builder=pyocr.builders.WordBoxBuilder()
)
# list of box objects. For each box object:
#   box.content is the word in the box
#   box.position is its position on the page (in pixels)
#
# Beware that some OCR tools (Tesseract for instance)
# may return empty boxes

line_and_word_boxes = tool.image_to_string(
    Image.open('test.png'), lang="fra",
    builder=pyocr.builders.LineBoxBuilder()
)
# list of line objects. For each line object:
#   line.word_boxes is a list of word boxes (the individual words in the line)
#   line.content is the whole text of the line
#   line.position is the position of the whole line on the page (in pixels)
#
# Beware that some OCR tools (Tesseract for instance)
# may return empty boxes

# Digits - Only Tesseract (not 'libtesseract' yet !)
digits = tool.image_to_string(
    Image.open('test-digits.png'),
    lang=lang,
    builder=pyocr.tesseract.DigitBuilder()
)
# digits is a python string
```

Leggi Riconoscimento ottico dei caratteri online:
https://riptutorial.com/it/python/topic/9302/riconoscimento-ottico-dei-caratteri

Capitolo 171: ricorsione

Osservazioni

La ricorsione ha bisogno di una condizione di stop `stopCondition` per uscire dalla ricorsione.

La variabile originale deve essere passata alla funzione ricorsiva in modo che venga archiviata.

Examples

Somma di numeri da 1 a n

Se volessi scoprire la somma dei numeri da `1` a `n` dove `n` è un numero naturale, posso fare `1 + 2 + 3 + 4 + ... + (several hours later) + n` . In alternativa, potrei scrivere un ciclo `for` :

```
n = 0
for i in range (1, n+1):
    n += i
```

O potrei usare una tecnica conosciuta come ricorsione:

```
def recursion(n):
    if n == 1:
        return 1
    return n + recursion(n - 1)
```

La ricorsione ha vantaggi rispetto ai due metodi precedenti. La ricorsione richiede meno tempo rispetto alla scrittura di `1 + 2 + 3` per una somma da 1 a 3. Per la `recursion(4)` , la ricorsione può essere utilizzata per lavorare all'indietro:

Chiamate di funzione: (4 -> 4 + 3 -> 4 + 3 + 2 -> 4 + 3 + 2 + 1 -> 10)

Mentre il ciclo `for` sta funzionando rigorosamente in avanti: (1 -> 1 + 2 -> 1 + 2 + 3 -> 1 + 2 + 3 + 4 -> 10). A volte la soluzione ricorsiva è più semplice della soluzione iterativa. Questo è evidente quando si implementa un'inversione di una lista collegata.

Il cosa, come e quando della ricorsione

La ricorsione si verifica quando una chiamata di funzione fa sì che la stessa funzione venga richiamata nuovamente prima che la chiamata della funzione originale termini. Ad esempio, considera la ben nota espressione matematica `x!` (cioè l'operazione fattoriale). L'operazione fattoriale è definita per tutti gli interi non negativi come segue:

- Se il numero è 0, la risposta è 1.
- Altrimenti, la risposta è quel numero moltiplicato per il fattoriale di un numero inferiore a quel numero.

In Python, un'implementazione ingenua dell'operazione fattoriale può essere definita come una funzione come segue:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

Le funzioni di ricorsione possono essere difficili da comprendere a volte, quindi passiamo a questa procedura dettagliata. Considera l'espressione `factorial(3)` . Queste e *tutte le* chiamate di funzione creano un nuovo **ambiente** . Un ambiente è fondamentalmente solo una tabella che mappa gli identificatori (ad esempio `n` , `factorial` , `print` , ecc.) Ai loro valori corrispondenti. In qualsiasi momento, è possibile accedere all'ambiente corrente utilizzando i `locals()` . Nella prima chiamata di funzione, l'unica variabile locale che viene definita è `n = 3` . Pertanto, la stampa dei `locals()` mostrerebbe `{'n': 3}` . Poiché `n == 3` , il valore restituito diventa `n * factorial(n - 1)` .

A questo punto seguente è dove le cose potrebbero diventare un po 'confuse. Guardando la nostra nuova espressione, sappiamo già che cosa è `n` . Tuttavia, non sappiamo ancora cosa sia `factorial(n - 1)` . Innanzitutto, `n - 1` restituisce `2` . Quindi, `2` è passato a `factorial` come valore per `n` . Poiché si tratta di una nuova chiamata di funzione, viene creato un secondo ambiente per memorizzare questo nuovo `n` . Sia *A* il primo ambiente e *B* il secondo ambiente. *Un* esiste ancora ed è uguale a `{'n': 3}` , tuttavia, *B* (che è uguale a `{'n': 2}`) è l'ambiente corrente. Guardando il corpo della funzione, il valore di ritorno è, di nuovo, `n * factorial(n - 1)` . Senza valutare questa espressione, sostituiamola nell'espressione di ritorno originale. In questo modo, stiamo mentalmente scartando *B,* quindi ricordatevi di sostituire `n` conseguenza (cioè i riferimenti a *B* 's `n` vengono sostituite con `n - 1` che utilizza *un'* s `n`). Ora, l'espressione di ritorno originale diventa `n * ((n - 1) * factorial((n - 1) - 1))` . Prenditi un secondo per assicurarti di capire perché è così.

Ora, valutiamo la parte `factorial((n - 1) - 1))` di quello. Dal momento che *A* è `n == 3` , stiamo passando a `1 factorial` . Pertanto, stiamo creando un nuovo ambiente *C* che equivale a `{'n': 1}` . Ancora, il valore restituito è `n * factorial(n - 1)` . Quindi sostituiamo `factorial((n - 1) - 1))` dell'espressione di ritorno "originale" in modo simile a come abbiamo regolato l'espressione di ritorno originale in precedenza. L'espressione "originale" è ora `n * ((n - 1) * ((n - 2) * factorial((n - 2) - 1)))` .

Quasi fatto. Ora, dobbiamo valutare `factorial((n - 2) - 1)` . Questa volta, stiamo passando `0` . Pertanto, questo vale `1` . Ora, eseguiamo la nostra ultima sostituzione. L'espressione di ritorno "originale" è ora `n * ((n - 1) * ((n - 2) * 1))` . Ricordando che l'espressione di ritorno originale è valutata sotto *A* , l'espressione diventa `3 * ((3 - 1) * ((3 - 2) * 1))` . Questo, ovviamente, valuta a 6. Per confermare che questa è la risposta corretta, ricorda che `3! == 3 * 2 * 1 == 6` . Prima di leggere ulteriormente, assicurati di comprendere appieno il concetto di ambiente e come si applicano alla ricorsione.

L'istruzione `if n == 0: return 1` è chiamata caso base. Questo perché, non mostra alcuna ricorsione. Un caso base è assolutamente necessario. Senza uno, ti imbatterai in una ricorsione infinita. Detto questo, se hai almeno un caso base, puoi avere tutti i casi che vuoi. Ad esempio, potremmo avere `factorial` scritto in modo equivalente come segue:

```
def factorial(n):
    if n == 0:
        return 1
    elif n == 1:
        return 1
    else:
        return n * factorial(n - 1)
```

Potresti anche avere più casi di ricorsione, ma non entreremo in questo dato che è relativamente raro ed è spesso difficile da elaborare mentalmente.

È anche possibile avere chiamate di funzioni ricorsive "parallele". Ad esempio, considera la sequenza di Fibonacci che è definita come segue:

- Se il numero è 0, la risposta è 0.
- Se il numero è 1, la risposta è 1.
- Altrimenti, la risposta è la somma dei due precedenti numeri di Fibonacci.

Possiamo definire questo è il seguente:

```
def fib(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib(n - 2) + fib(n - 1)
```

Non passerò attraverso questa funzione così accuratamente come ho fatto con `factorial(3)` , ma il valore di ritorno finale di `fib(5)` è equivalente alla seguente espressione (*sintatticamente* non valida):

```
(
  fib((n - 2) - 2)
  +
  (
    fib(((n - 2) - 1) - 2)
    +
    fib(((n - 2) - 1) - 1)
  )
)
+
(
  (
    fib(((n - 1) - 2) - 2)
    +
    fib(((n - 1) - 2) - 1)
  )
  +
  (
    fib(((n - 1) - 1) - 2)
    +
    (
      fib((((n - 1) - 1) - 1) - 2)
      +
      fib((((n - 1) - 1) - 1) - 1)
    )
  )
```

```
)
```

Questo diventa `(1 + (0 + 1)) + ((0 + 1) + (1 + (0 + 1)))` che ovviamente valuta a `5` .

Ora, copriamo alcuni termini del vocabolario:

- Una **coda di chiamata** è semplicemente una chiamata di funzione ricorsiva che è l'ultima operazione da eseguire prima di restituire un valore. Per essere chiari, `return foo(n - 1)` è una coda, ma `return foo(n - 1) + 1` non lo è (poiché l'aggiunta è l'ultima operazione).
- **Ottimizzazione chiamata coda** (TCO) è un modo per ridurre automaticamente la ricorsione nelle funzioni ricorsive.
- **L'eliminazione della chiamata di coda** (TCE) è la riduzione di una chiamata di coda a un'espressione che può essere valutata senza ricorsione. TCE è un tipo di TCO.

L'ottimizzazione della chiamata di coda è utile per una serie di motivi:

- L'interprete può minimizzare la quantità di memoria occupata dagli ambienti. Poiché nessun computer dispone di memoria illimitata, le chiamate eccessive di funzioni ricorsive comportano un **sovraccarico dello stack** .
- L'interprete può ridurre il numero di interruttori dello **stack frame** .

Python non ha alcuna forma di TCO implementata per una serie di motivi . Pertanto, sono necessarie altre tecniche per superare questa limitazione. Il metodo di scelta dipende dal caso d'uso. Con un po 'di intuizione, le definizioni di `factorial` e `fib` possono essere facilmente convertite in codice iterativo come segue:

```
def factorial(n):
    product = 1
    while n > 1:
        product *= n
        n -= 1
    return product

def fib(n):
    a, b = 0, 1
    while n > 0:
        a, b = b, a + b
        n -= 1
    return a
```

Questo è solitamente il modo più efficace per eliminare manualmente la ricorsione, ma può diventare piuttosto difficile per funzioni più complesse.

Un altro utile strumento è il decoratore lru_cache di Python che può essere utilizzato per ridurre il numero di calcoli ridondanti.

Ora hai un'idea su come evitare la ricorsione in Python, ma quando *dovresti* ricorrere alla ricorsione? La risposta è "non spesso". Tutte le funzioni ricorsive possono essere implementate in modo iterativo. Si tratta semplicemente di capire come farlo. Tuttavia, ci sono rari casi in cui la ricorsione è a posto. La ricorsione è comune in Python quando gli input previsti non causano un numero significativo di chiamate di funzioni ricorsive.

Se la ricorsione è un argomento che ti interessa, ti imploro di studiare linguaggi funzionali come Scheme o Haskell. In tali lingue, la ricorsione è molto più utile.

Si noti che l'esempio precedente per la sequenza di Fibonacci, sebbene sia efficace nel mostrare come applicare la definizione in python e l'uso successivo della cache lru, ha un tempo di esecuzione inefficiente poiché esegue 2 chiamate ricorsive per ciascun caso non base. Il numero di chiamate alla funzione aumenta esponenzialmente a `n` .
Piuttosto non intuitivamente, un'implementazione più efficiente userebbe la ricorsione lineare:

```
def fib(n):
    if n <= 1:
        return (n,0)
    else:
        (a, b) = fib(n - 1)
        return (a + b, a)
```

Ma quello ha il problema di restituire un *paio* di numeri. Ciò sottolinea che alcune funzioni in realtà non guadagnano molto dalla ricorsione.

Esplorazione di alberi con ricorsione

Diciamo che abbiamo il seguente albero:

```
root
- A
  - AA
  - AB
- B
  - BA
  - BB
    - BBA
```

Ora, se vogliamo elencare tutti i nomi degli elementi, potremmo farlo con un semplice ciclo. Supponiamo che esista una funzione `get_name()` per restituire una stringa del nome di un nodo, una funzione `get_children()` per restituire un elenco di tutti i sub-nodi di un determinato nodo nell'albero e una funzione `get_root()` per ottenere il nodo radice.

```
root = get_root(tree)
for node in get_children(root):
    print(get_name(node))
    for child in get_children(node):
        print(get_name(child))
        for grand_child in get_children(child):
            print(get_name(grand_child))
# prints: A, AA, AB, B, BA, BB, BBA
```

Funziona bene e velocemente, ma cosa succede se i sub-nodi hanno dei sub-nodi propri? E quei sub-nodi potrebbero avere più sotto-nodi ... E se non sapessi in anticipo quanti ce ne saranno? Un metodo per risolvere questo è l'uso della ricorsione.

```
def list_tree_names(node):
```

```
    for child in get_children(node):
        print(get_name(child))
        list_tree_names(node=child)

list_tree_names(node=get_root(tree))
# prints: A, AA, AB, B, BA, BB, BBA
```

Forse non desideri stampare, ma restituisci un elenco semplice di tutti i nomi dei nodi. Questo può essere fatto passando una lista a rotazione come parametro.

```
def list_tree_names(node, lst=[]):
    for child in get_children(node):
        lst.append(get_name(child))
        list_tree_names(node=child, lst=lst)
    return lst

list_tree_names(node=get_root(tree))
# returns ['A', 'AA', 'AB', 'B', 'BA', 'BB', 'BBA']
```

Aumentare la profondità massima di ricorsione

C'è un limite alla profondità della possibile ricorsione, che dipende dall'implementazione di Python. Quando viene raggiunto il limite, viene sollevata un'eccezione RuntimeError:

```
RuntimeError: Maximum Recursion Depth Exceeded
```

Ecco un esempio di un programma che causerebbe questo errore:

```
def cursing(depth):
  try:
    cursing(depth + 1) # actually, re-cursing
  except RuntimeError as RE:
    print('I recursed {} times!'.format(depth))
cursing(0)
# Out: I recursed 1083 times!
```

È possibile modificare il limite di profondità di ricorsione utilizzando

```
sys.setrecursionlimit(limit)
```

Puoi controllare quali sono i parametri attuali del limite eseguendo:

```
sys.getrecursionlimit()
```

Eseguendo lo stesso metodo sopra con il nostro nuovo limite otteniamo

```
sys.setrecursionlimit(2000)
cursing(0)
# Out: I recursed 1997 times!
```

Da Python 3.5, l'eccezione è RecursionError, che deriva da RuntimeError.

Ricorsione di coda - Cattiva pratica

Quando l'unica cosa restituita da una funzione è una chiamata ricorsiva, viene indicata come ricorsione di coda.

Ecco un esempio di conto alla rovescia scritto utilizzando la ricorsione in coda:

```
def countdown(n):
    if n == 0:
        print "Blastoff!"
    else:
        print n
        countdown(n-1)
```

Qualsiasi computazione che può essere fatta usando l'iterazione può anche essere fatta usando la ricorsione. Ecco una versione di find_max scritta utilizzando la ricorsione di coda:

```
def find_max(seq, max_so_far):
    if not seq:
        return max_so_far
    if max_so_far < seq[0]:
        return find_max(seq[1:], seq[0])
    else:
        return find_max(seq[1:], max_so_far)
```

La ricorsione di coda è considerata una cattiva pratica in Python, dal momento che il compilatore Python non gestisce l'ottimizzazione per le chiamate ricorsive di coda. La soluzione ricorsiva in casi come questo utilizza più risorse di sistema rispetto alla soluzione iterativa equivalente.

Ottimizzazione della ricorsione della coda attraverso l'introspezione della pila

Di default, lo stack di ricorsione di Python non può superare i 1000 frame. Questo può essere modificato impostando `sys.setrecursionlimit(15000)` che è più veloce, tuttavia, questo metodo consuma più memoria. Invece, possiamo anche risolvere il problema di ricorsione della coda usando l'introspezione dello stack.

```
#!/usr/bin/env python2.4
# This program shows off a python decorator which implements tail call optimization. It
# does this by throwing an exception if it is it's own grandparent, and catching such
# exceptions to recall the stack.

import sys

class TailRecurseException:
    def __init__(self, args, kwargs):
        self.args = args
        self.kwargs = kwargs

def tail_call_optimized(g):
"""
This function decorates a function with tail call
optimization. It does this by throwing an exception
if it is it's own grandparent, and catching such
```

```
exceptions to fake the tail call optimization.

This function fails if the decorated
function recurses in a non-tail context.
"""

    def func(*args, **kwargs):
        f = sys._getframe()
        if f.f_back and f.f_back.f_back and f.f_back.f_back.f_code == f.f_code:
            raise TailRecurseException(args, kwargs)
        else:
            while 1:
                try:
                    return g(*args, **kwargs)
                except TailRecurseException, e:
                    args = e.args
                    kwargs = e.kwargs
    func.__doc__ = g.__doc__
    return func
```

Per ottimizzare le funzioni ricorsive, possiamo usare il decoratore `@tail_call_optimized` per chiamare la nostra funzione. Ecco alcuni degli esempi di ricorsione più comuni utilizzando il decoratore sopra descritto:

Esempio fattoriale:

```
@tail_call_optimized
def factorial(n, acc=1):
  "calculate a factorial"
  if n == 0:
    return acc
  return factorial(n-1, n*acc)

print factorial(10000)
# prints a big, big number,
# but doesn't hit the recursion limit.
```

Esempio di Fibonacci:

```
@tail_call_optimized
def fib(i, current = 0, next = 1):
  if i == 0:
    return current
  else:
    return fib(i - 1, next, current + next)

print fib(10000)
# also prints a big number,
# but doesn't hit the recursion limit.
```

Leggi ricorsione online: https://riptutorial.com/it/python/topic/1716/ricorsione

Capitolo 172: Ridurre

Sintassi

- riduci (funzione, iterabile [, inizializzatore])

Parametri

Parametro	Dettagli
funzione	funzione che viene utilizzata per ridurre l'iterabile (deve contenere due argomenti). (*solo posizionale*)
iterabile	iterabile che sarà ridotto. (*solo posizionale*)
initializer	valore iniziale della riduzione. (*opzionale* , *solo posizionale*)

Osservazioni

`reduce` potrebbe non essere sempre la funzione più efficiente. Per alcuni tipi ci sono funzioni o metodi equivalenti:

- `sum()` per la somma di una sequenza contenente elementi *aggiuntivi* (non stringhe):

```
sum([1,2,3])                                    # = 6
```

- `str.join` per la concatenazione di stringhe:

```
''.join(['Hello', ',', ' World'])               # = 'Hello, World'
```

- `next` insieme ad un generatore potrebbe essere una variante cortocircuito rispetto a `reduce` :

```
# First falsy item:
next((i for i in [100, [], 20, 0] if not i)) # = []
```

Examples

Panoramica

```
# No import needed


# No import required...
from functools import reduce # ... but it can be loaded from the functools module
```

```
from functools import reduce # mandatory
```

`reduce` riduce un iterable applicando ripetutamente una funzione sull'elemento successivo di un `iterable` e il risultato cumulativo finora.

```
def add(s1, s2):
    return s1 + s2

asequence = [1, 2, 3]

reduce(add, asequence)  # equivalent to: add(add(1,2),3)
# Out: 6
```

In questo esempio, abbiamo definito la nostra funzione di `add` . Tuttavia, Python viene fornito con una funzione equivalente standard nel modulo `operator` :

```
import operator
reduce(operator.add, asequence)
# Out: 6
```

`reduce` può anche essere passato un valore iniziale:

```
reduce(add, asequence, 10)
# Out: 16
```

Utilizzando ridurre

```
def multiply(s1, s2):
    print('{arg1} * {arg2} = {res}'.format(arg1=s1,
                                           arg2=s2,
                                           res=s1*s2))
    return s1 * s2

asequence = [1, 2, 3]
```

Dato un `initializer` la funzione viene avviata applicandola all'inizializzatore e al primo elemento iterabile:

```
cumprod = reduce(multiply, asequence, 5)
# Out: 5 * 1 = 5
#      5 * 2 = 10
#      10 * 3 = 30
print(cumprod)
# Out: 30
```

Senza il parametro `initializer` , la `reduce` inizia applicando la funzione ai primi due elementi dell'elenco:

```
cumprod = reduce(multiply, asequence)
```

```
# Out: 1 * 2 = 2
#      2 * 3 = 6
print(cumprod)
# Out: 6
```

Prodotto cumulativo

```
import operator
reduce(operator.mul, [10, 5, -3])
# Out: -150
```

Variante non a corto circuito di any / all

`reduce` non termina l'iterazione prima che l' `iterable` sia stato completamente ripetuto, in modo che possa essere usato per creare una funzione `any()` o `all()` senza cortocircuito:

```
import operator
# non short-circuit "all"
reduce(operator.and_, [False, True, True, True]) # = False

# non short-circuit "any"
reduce(operator.or_, [True, False, False, False]) # = True
```

Primo elemento di verità / falsy di una sequenza (o ultimo elemento se non ce n'è)

```
# First falsy element or last element if all are truthy:
reduce(lambda i, j: i and j, [100, [], 20, 10])    # = []
reduce(lambda i, j: i and j, [100, 50, 20, 10])    # = 10

# First truthy element or last element if all falsy:
reduce(lambda i, j: i or j, [100, [], 20, 0])     # = 100
reduce(lambda i, j: i or j, ['', {}, [], None])   # = None
```

Invece di creare una funzione `lambda` , in genere è consigliabile creare una funzione con nome:

```
def do_or(i, j):
    return i or j

def do_and(i, j):
    return i and j

reduce(do_or, [100, [], 20, 0])                   # = 100
reduce(do_and, [100, [], 20, 0])                  # = []
```

Leggi Ridurre online: https://riptutorial.com/it/python/topic/328/ridurre

Capitolo 173: Scrittura in formato CSV da stringa o elenco

introduzione

Scrivere in un file .csv non è diverso dalla scrittura in un file normale, per quanto riguarda gli aspetti, ed è abbastanza semplice. Farò al meglio delle mie possibilità l'approccio più semplice e più efficiente al problema.

Parametri

Parametro	Dettagli
open (**"/ percorso /"** , "modalità")	Specifica il percorso del tuo file CSV
aperto (percorso, **"modalità"**)	Specificare la modalità per aprire il file in (leggi, scrivi, ecc.)
csv.writer (**file** , delimitatore)	Passa qui il file CSV aperto
csv.writer (file, **delimitatore = "**)	Specifica il carattere o il modello del delimitatore

Osservazioni

```
open( path, "wb")
```

`"wb"` - Modalità di scrittura.

Il parametro `b` in `"wb"` che abbiamo usato è necessario solo se si desidera aprirlo in modalità binaria, che è necessaria solo in alcuni sistemi operativi come Windows.

```
csv.writer ( csv_file, delimiter=',' )
```

Qui il delimitatore che abbiamo usato è `,` perché vogliamo che ogni cella di dati di una riga contenga rispettivamente il nome, il cognome e l'età. Dal momento che anche la nostra lista è divisa `,` per noi risulta piuttosto conveniente.

Examples

Esempio di scrittura di base

```
import csv
```

```
#------ We will write to CSV in this function ------------

def csv_writer(data, path):

    #Open CSV file whose path we passed.
    with open(path, "wb") as csv_file:

        writer = csv.writer(csv_file, delimiter=',')
        for line in data:
            writer.writerow(line)



#---- Define our list here, and call function ------------

if __name__ == "__main__":

    """
    data = our list that we want to write.
    Split it so we get a list of lists.
    """
    data = ["first_name,last_name,age".split(","),
            "John,Doe,22".split(","),
            "Jane,Doe,31".split(","),
            "Jack,Reacher,27".split(",")
            ]

    # Path to CSV file we want to write to.
    path = "output.csv"
    csv_writer(data, path)
```

Aggiunta di una stringa come nuova riga in un file CSV

```
def append_to_csv(input_string):
    with open("fileName.csv", "a") as csv_file:
        csv_file.write(input_row + "\n")
```

Leggi Scrittura in formato CSV da stringa o elenco online:
https://riptutorial.com/it/python/topic/10862/scrittura-in-formato-csv-da-stringa-o-elenco

Capitolo 174: Scrivere estensioni

Examples

Ciao mondo con estensione C

Il seguente file sorgente C (che chiameremo `hello.c` a scopo dimostrativo) produce un modulo di estensione denominato `hello` che contiene una singola funzione `greet()` :

```
#include <Python.h>
#include <stdio.h>

#if PY_MAJOR_VERSION >= 3
#define IS_PY3K
#endif

static PyObject *hello_greet(PyObject *self, PyObject *args)
{
    const char *input;
    if (!PyArg_ParseTuple(args, "s", &input)) {
        return NULL;
    }
    printf("%s", input);
    Py_RETURN_NONE;
}

static PyMethodDef HelloMethods[] = {
    { "greet", hello_greet, METH_VARARGS, "Greet the user" },
    { NULL, NULL, 0, NULL }
};

#ifdef IS_PY3K
static struct PyModuleDef hellomodule = {
    PyModuleDef_HEAD_INIT, "hello", NULL, -1, HelloMethods
};

PyMODINIT_FUNC PyInit_hello(void)
{
    return PyModule_Create(&hellomodule);
}
#else
PyMODINIT_FUNC inithello(void)
{
    (void) Py_InitModule("hello", HelloMethods);
}
#endif
```

Per compilare il file con il compilatore `gcc` , esegui il seguente comando nel tuo terminale preferito:

```
gcc /path/to/your/file/hello.c -o /path/to/your/file/hello
```

Per eseguire la funzione `greet()` che abbiamo scritto in precedenza, creare un file nella stessa directory e chiamarlo `hello.py`

```
import hello          # imports the compiled library
hello.greet("Hello!") # runs the greet() function with "Hello!" as an argument
```

Passare un file aperto alle estensioni C

Passa un oggetto file aperto da Python al codice di estensione C.

È possibile convertire il file in un descrittore di file intero utilizzando la funzione `PyObject_AsFileDescriptor` :

```
PyObject *fobj;
int fd = PyObject_AsFileDescriptor(fobj);
if (fd < 0){
    return NULL;
}
```

Per convertire un descrittore di file intero in un oggetto python, usa `PyFile_FromFd` .

```
int fd; /* Existing file descriptor */
PyObject *fobj = PyFile_FromFd(fd, "filename","r",-1,NULL,NULL,NULL,1);
```

Estensione C con c ++ e Boost

Questo è un esempio di base di *un'estensione* C che utilizza C ++ e Boost .

Codice C ++

Codice C ++ messo in hello.cpp:

```
#include <boost/python/module.hpp>
#include <boost/python/list.hpp>
#include <boost/python/class.hpp>
#include <boost/python/def.hpp>

// Return a hello world string.
std::string get_hello_function()
{
   return "Hello world!";
}

// hello class that can return a list of count hello world strings.
class hello_class
{
public:

   // Taking the greeting message in the constructor.
   hello_class(std::string message) : _message(message) {}

   // Returns the message count times in a python list.
   boost::python::list as_list(int count)
   {
      boost::python::list res;
```

```
      for (int i = 0; i < count; ++i) {
         res.append(_message);
      }
      return res;
   }

private:
   std::string _message;
};


// Defining a python module naming it to "hello".
BOOST_PYTHON_MODULE(hello)
{
   // Here you declare what functions and classes that should be exposed on the module.

   // The get_hello_function exposed to python as a function.
   boost::python::def("get_hello", get_hello_function);

   // The hello_class exposed to python as a class.
   boost::python::class_<hello_class>("Hello", boost::python::init<std::string>())
      .def("as_list", &hello_class::as_list)
      ;
}
```

Per compilarlo in un modulo python occorrono le intestazioni python e le librerie boost. Questo esempio è stato realizzato su Ubuntu 12.04 usando python 3.4 e gcc. Boost è supportato su molte piattaforme. Nel caso di Ubuntu i pacchetti necessari sono stati installati usando:

```
sudo apt-get install gcc libboost-dev libpython3.4-dev
```

Compilare il file sorgente in un file .so che può essere successivamente importato come modulo purché si trovi sul percorso python:

```
gcc -shared -o hello.so -fPIC -I/usr/include/python3.4 hello.cpp -lboost_python-py34 -
lboost_system -l:libpython3.4m.so
```

Il codice python nel file example.py:

```
import hello

print(hello.get_hello())

h = hello.Hello("World hello!")
print(h.as_list(3))
```

Quindi `python3 example.py` darà il seguente risultato:

```
Hello world!
['World hello!', 'World hello!', 'World hello!']
```

Leggi Scrivere estensioni online: https://riptutorial.com/it/python/topic/557/scrivere-estensioni

Capitolo 175: Secure Shell Connection in Python

Parametri

Parametro	uso
hostname	Questo parametro indica l'host a cui deve essere stabilita la connessione
nome utente	nome utente richiesto per accedere all'host
porta	porta ospite
parola d'ordine	password per l'account

Examples

connessione ssh

```
from paramiko import client
ssh = client.SSHClient() # create a new SSHClient object
ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy()) #auto-accept unknown host keys
ssh.connect(hostname, username=username, port=port, password=password) #connect with a host
stdin, stdout, stderr = ssh.exec_command(command) # submit a command to ssh
print stdout.channel.recv_exit_status() #tells the status  1 - job failed
```

Leggi Secure Shell Connection in Python online: https://riptutorial.com/it/python/topic/5709/secure-shell-connection-in-python

Capitolo 176: Semplici operatori matematici

introduzione

Python fa da solo operatori matematici comuni, compresi interi e float, moltiplicazione, esponenziazione, addizione e sottrazione. Il modulo matematico (incluso in tutte le versioni standard di Python) offre funzionalità estese come funzioni trigonometriche, operazioni di root, logaritmi e molto altro.

Osservazioni

Tipi numerici e loro metaclassi

Il modulo `numbers` contiene i metaclassi astratti per i tipi numerici:

sottoclassi	numbers.Number	numbers.Integral	numbers.Rational	numbers.Real	nur
bool	✓	✓	✓	✓	✓
int	✓	✓	✓	✓	✓
fractions.Fraction	✓	-	✓	✓	✓
galleggiante	✓	-	-	✓	✓
complesso	✓	-	-	-	✓
decimal.Decimal	✓	-	-	-	-

Examples

aggiunta

```
a, b = 1, 2

# Using the "+" operator:
a + b                   # = 3

# Using the "in-place" "+=" operator to add and assign:
a += b                  # a = 3 (equivalent to a = a + b)

import operator         # contains 2 argument arithmetic functions for the examples

operator.add(a, b)      # = 5  since a is set to 3 right before this line
```

```
# The "+=" operator is equivalent to:
a = operator.iadd(a, b)    # a = 5 since a is set to 3 right before this line
```

Combinazioni possibili (tipi predefiniti):

- `int` e `int` (dà un `int`)
- `int` e `float` (dà un `float`)
- `int` e `complex` (dà un `complex`)
- `float` e `float` (dà un `float`)
- `float` e `complex` (dà un `complex`)
- `complex` e `complex` (dà un `complex`)

Nota: l'operatore `+` è anche usato per concatenare stringhe, liste e tuple:

```
"first string " + "second string"    # = 'first string second string'

[1, 2, 3] + [4, 5, 6]                # = [1, 2, 3, 4, 5, 6]
```

Sottrazione

```
a, b = 1, 2

# Using the "-" operator:
b - a                  # = 1


import operator        # contains 2 argument arithmetic functions
operator.sub(b, a)     # = 1
```

Combinazioni possibili (tipi predefiniti):

- `int` e `int` (dà un `int`)
- `int` e `float` (dà un `float`)
- `int` e `complex` (dà un `complex`)
- `float` e `float` (dà un `float`)
- `float` e `complex` (dà un `complex`)
- `complex` e `complex` (dà un `complex`)

Moltiplicazione

```
a, b = 2, 3

a * b                  # = 6

import operator
operator.mul(a, b)     # = 6
```

Combinazioni possibili (tipi predefiniti):

- `int` e `int` (dà un `int`)
- `int` e `float` (dà un `float`)
- `int` e `complex` (dà un `complex`)
- `float` e `float` (dà un `float`)
- `float` e `complex` (dà un `complex`)
- `complex` e `complex` (dà un `complex`)

Nota: l'operatore `*` viene anche utilizzato per la concatenazione ripetuta di stringhe, elenchi e tuple:

```
3 * 'ab'  # = 'ababab'
3 * ('a', 'b')  # = ('a', 'b', 'a', 'b', 'a', 'b')
```

Divisione

Python esegue la divisione integer quando entrambi gli operandi sono numeri interi. Il comportamento degli operatori di divisione di Python è cambiato da Python 2.xe 3.x (vedere anche Divisione Integer).

```
a, b, c, d, e = 3, 2, 2.0, -3, 10
```

Python 2.x 2.7

In Python 2 il risultato dell'operatore '/' dipende dal tipo di numeratore e denominatore.

```
a / b                   # = 1

a / c                   # = 1.5

d / b                   # = -2

b / a                   # = 0

d / e                   # = -1
```

Si noti che poiché sia `a` che `b` sono `int` s, il risultato è un `int` .

Il risultato è sempre arrotondato per difetto (pavimentato).

Poiché `c` è un float, il risultato di `a / c` è un `float` .

Puoi anche usare il modulo operatore:

```
import operator         # the operator module provides 2-argument arithmetic functions
operator.div(a, b)      # = 1
operator.__div__(a, b) # = 1
```

Python 2.x 2.2

Cosa succede se si desidera la divisione float:

Consigliato:

```
from __future__ import division # applies Python 3 style division to the entire module
a / b                   # = 1.5
a // b                  # = 1
```

Va bene (se non si desidera applicare all'intero modulo):

```
a / (b * 1.0)           # = 1.5
1.0 * a / b             # = 1.5
a / b * 1.0             # = 1.0    (careful with order of operations)

from operator import truediv
truediv(a, b)           # = 1.5
```

Non raccomandato (può sollevare l'errore TypeError, ad es. Se l'argomento è complesso):

```
float(a) / b            # = 1.5
a / float(b)            # = 1.5
```

Python 2.x 2.2

L'operatore '//' in Python 2 forza la divisione a pavimento indipendentemente dal tipo.

```
a // b                # = 1
a // c                # = 1.0
```

Python 3.x 3.0

In Python 3 l'operatore `/` esegue la divisione "true" indipendentemente dai tipi. L'operatore `//` esegue la divisione di piano e mantiene il tipo.

```
a / b                   # = 1.5
e / b                   # = 5.0
a // b                  # = 1
a // c                  # = 1.0

import operator             # the operator module provides 2-argument arithmetic functions
operator.truediv(a, b)      # = 1.5
operator.floordiv(a, b)     # = 1
operator.floordiv(a, c)     # = 1.0
```

Combinazioni possibili (tipi predefiniti):

- `int` e `int` (fornisce un `int` in Python 2 e un `float` in Python 3)
- `int` e `float` (dà un `float`)
- `int` e `complex` (dà un `complex`)
- `float` e `float` (dà un `float`)
- `float` e `complex` (dà un `complex`)

- `complex` e `complex` (dà un `complex`)

Vedere PEP 238 per ulteriori informazioni.

Exponentation

```
a, b = 2, 3

(a ** b)                # = 8
pow(a, b)               # = 8

import math
math.pow(a, b)          # = 8.0 (always float; does not allow complex results)

import operator
operator.pow(a, b)      # = 8
```

Un'altra differenza tra il `pow` `math.pow` e `math.pow` è che il `pow` incorporato può accettare tre argomenti:

```
a, b, c = 2, 3, 2

pow(2, 3, 2)            # 0, calculates (2 ** 3) % 2, but as per Python docs,
                        #    does so more efficiently
```

Funzioni speciali

La funzione `math.sqrt(x)` calcola la radice quadrata di `x` .

```
import math
import cmath
c = 4
math.sqrt(c)            # = 2.0 (always float; does not allow complex results)
cmath.sqrt(c)           # = (2+0j) (always complex)
```

Per calcolare altre radici, come una radice cubica, aumentare il numero al reciproco del grado della radice. Questo potrebbe essere fatto con qualsiasi funzione esponenziale o operatore.

```
 import math
 x = 8
 math.pow(x, 1/3) # evaluates to 2.0
 x**(1/3) # evaluates to 2.0
```

La funzione `math.exp(x)` calcola `e ** x` .

```
math.exp(0)  # 1.0
math.exp(1)  # 2.718281828459045 (e)
```

La funzione `math.expm1(x)` calcola `e ** x - 1` . Quando `x` è piccolo, questo fornisce una precisione significativamente migliore rispetto a `math.exp(x) - 1` .

```
math.expm1(0)       # 0.0

math.exp(1e-6) - 1  # 1.0000004999621837e-06
math.expm1(1e-6)    # 1.0000005000001665e-06
# exact result      # 1.0000005000001666667083333416666...
```

logaritmi

Per impostazione predefinita, la funzione `math.log` calcola il logaritmo di un numero, base e. Opzionalmente puoi specificare una base come secondo argomento.

```
import math
import cmath

math.log(5)         # = 1.6094379124341003
# optional base argument. Default is math.e
math.log(5, math.e) # = 1.6094379124341003
cmath.log(5)        # = (1.6094379124341003+0j)
math.log(1000, 10)   # 3.0 (always returns float)
cmath.log(1000, 10)  # (3+0j)
```

`math.log` varianti speciali della funzione `math.log` per diverse basi.

```
# Logarithm base e - 1 (higher precision for low values)
math.log1p(5)       # = 1.791759469228055

# Logarithm base 2
math.log2(8)        # = 3.0

# Logarithm base 10
math.log10(100)     # = 2.0
cmath.log10(100)    # = (2+0j)
```

Operazioni interne

È normale che all'interno delle applicazioni sia necessario avere un codice come questo:

```
a = a + 1
```

o

```
a = a * 2
```

C'è una scorciatoia efficace per queste operazioni sul posto:

```
a += 1
# and
a *= 2
```

Qualsiasi operatore matematico può essere utilizzato prima del carattere '=' per eseguire un'operazione inplace:

- `-=` decrementa la variabile in atto
- `+=` incrementa la variabile sul posto
- `*=` moltiplica la variabile sul posto
- `/=` divide la variabile in posizione
- `//=` floor divide la variabile al posto # Python 3
- `%=` restituisce il modulo della variabile sul posto
- `**=` aumenta a una potenza in atto

Esistono altri operatori sul posto per gli operatori bit a bit (`^` , `|` etc)

Funzioni trigonometriche

```
a, b = 1, 2

import math

math.sin(a)  # returns the sine of 'a' in radians
# Out: 0.8414709848078965

math.cosh(b)  # returns the inverse hyperbolic cosine of 'b' in radians
# Out: 3.7621956910836314

math.atan(math.pi)  # returns the arc tangent of 'pi' in radians
# Out: 1.2626272556789115

math.hypot(a, b) # returns the Euclidean norm, same as math.sqrt(a*a + b*b)
# Out: 2.23606797749979
```

> Si noti che `math.hypot(x, y)` è anche la lunghezza del vettore (o distanza euclidea) dall'origine `(0, 0)` al punto `(x, y)` .
>
> Per calcolare la distanza euclidea tra due punti `(x1, y1)` e `(x2, y2)` puoi usare `math.hypot` come segue
>
> ```
> math.hypot(x2-x1, y2-y1)
> ```

Per convertire da radianti -> gradi e gradi -> radianti usa rispettivamente `math.degrees` e `math.radians`

```
math.degrees(a)
# Out: 57.29577951308232

math.radians(57.29577951308232)
# Out: 1.0
```

Modulo

Come in molti altri linguaggi, Python usa l'operatore `%` per calcolare il modulo.

```
3 % 4     # 3
10 % 2    # 0
6 % 4     # 2
```

O utilizzando il modulo `operator` :

```
import operator

operator.mod(3 , 4)      # 3
operator.mod(10 , 2)     # 0
operator.mod(6 , 4)      # 2
```

Puoi anche usare numeri negativi.

```
-9 % 7      # 5
9 % -7      # -5
-9 % -7     # -2
```

Se è necessario trovare il risultato della divisione e del modulo intero, è possibile utilizzare la funzione `divmod` come scelta rapida:

```
quotient, remainder = divmod(9, 4)
# quotient = 2, remainder = 1 as 4 * 2 + 1 == 9
```

Leggi Semplici operatori matematici online: https://riptutorial.com/it/python/topic/298/semplici-operatori-matematici

Capitolo 177: Serializzazione dei dati

Sintassi

- unpickled_string = pickle.loads (stringa)
- unpickled_string = pickle.load (file_object)
- pickled_string = pickle.dumps ([('', 'cmplx'), {('oggetto',): Nessuno}], pickle.HIGHEST_PROTOCOL)
- pickle.dump (('', 'cmplx'), {('oggetto',): Nessuno}], file_oggetto, pickle.HIGHEST_PROTOCOL)
- unjsoned_string = json.loads (stringa)
- unjsoned_string = json.load (file_object)
- jsoned_string = json.dumps (('a', 'b', 'c', [1, 2, 3]))
- json.dump (('a', 'b', 'c', [1, 2, 3]), file_object)

Parametri

Parametro	Dettagli
`protocol`	Usando `pickle` o `cPickle` , è il metodo con cui gli oggetti vengono serializzati / non serializzati. Probabilmente vorrai utilizzare `pickle.HIGHEST_PROTOCOL` qui, il che significa che il metodo più recente.

Osservazioni

Perché usare JSON?

- Supporto per le lingue incrociate
- Leggibile dagli umani
- A differenza di pickle, non ha il rischio di eseguire codice arbitrario

Perché non usare JSON?

- Non supporta i tipi di dati Pythonic
- Le chiavi nei dizionari non devono essere diversi dai tipi di dati stringa.

Perché Pickle?

- Ottimo modo per serializzare Pythonic (tuple, funzioni, classi)
- Le chiavi nei dizionari possono essere di qualsiasi tipo di dati.

Perché non sottaceto?

- Manca il supporto per le lingue incrociate
- Non è sicuro per il caricamento di dati arbitrari

Examples

Serializzazione tramite JSON

JSON è un metodo cross-language, ampiamente utilizzato per serializzare i dati

Tipi di dati supportati: *int* , *float* , *boolean* , *string* , *list* e *dict* . Vedi -> JSON Wiki per ulteriori informazioni

Ecco un esempio che dimostra l'utilizzo di **base** di **JSON** : -

```
import json

families = (['John'], ['Mark', 'David', {'name': 'Avraham'}])

# Dumping it into string
json_families = json.dumps(families)
# [["John"], ["Mark", "David", {"name": "Avraham"}]]

# Dumping it to file
with open('families.json', 'w') as json_file:
    json.dump(families, json_file)

# Loading it from string
json_families = json.loads(json_families)

# Loading it from file
with open('families.json', 'r') as json_file:
    json_families = json.load(json_file)
```

Vedi Modulo JSON per informazioni dettagliate su JSON.

Serializzazione tramite Pickle

Ecco un esempio che dimostra l'utilizzo di **base** del **sottaceto** : -

```
# Importing pickle
try:
    import cPickle as pickle  # Python 2
except ImportError:
    import pickle  # Python 3

# Creating Pythonic object:
class Family(object):
    def __init__(self, names):
        self.sons = names

    def __str__(self):
        return ' '.join(self.sons)

my_family = Family(['John', 'David'])

# Dumping to string
pickle_data = pickle.dumps(my_family, pickle.HIGHEST_PROTOCOL)
```

```
# Dumping to file
with open('family.p', 'w') as pickle_file:
    pickle.dump(families, pickle_file, pickle.HIGHEST_PROTOCOL)

# Loading from string
my_family = pickle.loads(pickle_data)

# Loading from file
with open('family.p', 'r') as pickle_file:
    my_family = pickle.load(pickle_file)
```

Vedi Pickle per informazioni dettagliate su Pickle.

ATTENZIONE : la documentazione ufficiale per il sottaceto chiarisce che non ci sono garanzie di sicurezza. Non caricare dati di cui non ti fidi dell'origine.

Leggi Serializzazione dei dati online: https://riptutorial.com/it/python/topic/3347/serializzazione-dei-dati

Capitolo 178: Serializzazione dei dati sottaceti

Sintassi

- pickle.dump (oggetto, file, protocollo) # Per serializzare un oggetto
- pickle.load (file) # Per de-serializzare un oggetto
- pickle.dumps (object, protocol) # Per serializzare un oggetto in byte
- pickle.loads (buffer) # Per deserializzare un oggetto da byte

Parametri

Parametro	Dettagli
oggetto	L'oggetto che deve essere memorizzato
file	Il file aperto che conterrà l'oggetto
protocollo	Il protocollo utilizzato per il decapaggio dell'oggetto (parametro opzionale)
buffer	Un oggetto byte che contiene un oggetto serializzato

Osservazioni

Tipi pickleable

I seguenti oggetti sono selezionabili.

- `None` , `True` e `False`
- numeri (di tutti i tipi)
- archi (di tutti i tipi)
- `tuple` s, `list` s, `set` s e `dict` s contenenti solo oggetti selezionabili
- funzioni definite al livello più alto di un modulo
- funzioni integrate
- classi che sono definite al livello più alto di un modulo
    - istanze di tali classi il cui `__dict__` o il risultato della chiamata `__getstate__()` è selezionabile (vedere i documenti ufficiali per i dettagli).

Basato sulla documentazione ufficiale di Python .

`pickle` e sicurezza

Il modulo pickle **non** è **sicuro** . Non dovrebbe essere usato quando si ricevono i dati serializzati da una parte non fidata, come su Internet.

Examples

Usare Pickle per serializzare e deserializzare un oggetto

Il modulo `pickle` implementa un algoritmo per trasformare un oggetto Python arbitrario in una serie di byte. Questo processo è anche chiamato **serializzare** l'oggetto. Il flusso di byte che rappresenta l'oggetto può quindi essere trasmesso o memorizzato e successivamente ricostruito per creare un nuovo oggetto con le stesse caratteristiche.

Per il codice più semplice, utilizziamo le funzioni `dump()` e `load()` .

Per serializzare l'oggetto

```
import pickle

# An arbitrary collection of objects supported by pickle.
data = {
    'a': [1, 2.0, 3, 4+6j],
    'b': ("character string", b"byte string"),
    'c': {None, True, False}
}

with open('data.pickle', 'wb') as f:
    # Pickle the 'data' dictionary using the highest protocol available.
    pickle.dump(data, f, pickle.HIGHEST_PROTOCOL)
```

Per deserializzare l'oggetto

```
import pickle

with open('data.pickle', 'rb') as f:
    # The protocol version used is detected automatically, so we do not
    # have to specify it.
    data = pickle.load(f)
```

Usando oggetti pickle e byte

È anche possibile serializzare e deserializzare in su oggetti byte, utilizzando le `dumps` e `loads`

funzione, che sono equivalenti a `dump` e `load` .

```
serialized_data = pickle.dumps(data, pickle.HIGHEST_PROTOCOL)
# type(serialized_data) is bytes

deserialized_data = pickle.loads(serialized_data)
# deserialized_data == data
```

Personalizza dati sottodimensionati

Alcuni dati non possono essere decapitati. Altri dati non dovrebbero essere messi in salamoia per altri motivi.

Ciò che verrà sottoposto a `__getstate__` può essere definito nel metodo `__getstate__` . Questo metodo deve restituire qualcosa che è selezionabile.

Sul lato opposto è `__setstate__` : riceverà ciò che `__getstate__` creato e deve inizializzare l'oggetto.

```
class A(object):
    def __init__(self, important_data):
        self.important_data = important_data

        # Add data which cannot be pickled:
        self.func = lambda: 7

        # Add data which should never be pickled, because it expires quickly:
        self.is_up_to_date = False

    def __getstate__(self):
        return [self.important_data] # only this is needed

    def __setstate__(self, state):
        self.important_data = state[0]

        self.func = lambda: 7  # just some hard-coded unpicklable function

        self.is_up_to_date = False  # even if it was before pickling
```

Ora, questo può essere fatto:

```
>>> a1 = A('very important')
>>>
>>> s = pickle.dumps(a1)  # calls a1.__getstate__()
>>>
>>> a2 = pickle.loads(s)  # calls a1.__setstate__(['very important'])
>>> a2
<__main__.A object at 0x0000000002742470>
>>> a2.important_data
'very important'
>>> a2.func()
7
```

L'implementazione qui pikles una lista con un valore: `[self.important_data]` . Questo era solo un esempio, `__getstate__` potrebbe aver restituito tutto ciò che è picklable, a condizione che `__setstate__` sappia come fare il referendum. Una buona alternativa è un dizionario di tutti i valori:

`{'important_data': self.important_data}`.

Il costruttore non viene chiamato! Si noti che nell'istanza dell'esempio precedente `a2` stato creato in `pickle.loads` senza mai chiamare `A.__init__` , quindi `A.__setstate__` doveva inizializzare tutto ciò che `__init__` avrebbe inizializzato se fosse stato chiamato.

Leggi Serializzazione dei dati sottaceti online:
https://riptutorial.com/it/python/topic/2606/serializzazione-dei-dati-sottaceti

Capitolo 179: setup.py

Parametri

Parametro	uso
`name`	Nome della tua distribuzione.
`version`	Stringa di versione della tua distribuzione.
`packages`	Elenco di pacchetti Python (ovvero, directory contenenti moduli) da includere. Questo può essere specificato manualmente, ma in genere viene utilizzata una chiamata a `setuptools.find_packages()` .
`py_modules`	Elenco dei moduli Python di primo livello (ovvero, singoli file `.py`) da includere.

Osservazioni

Per ulteriori informazioni sulla confezione Python vedere:

introduzione

Per scrivere pacchetti ufficiali c'è una guida per l'utente della confezione .

Examples

Scopo di setup.py

Lo script di installazione è il centro di tutte le attività nella costruzione, distribuzione e installazione di moduli usando le Distutils. Lo scopo è la corretta installazione del software.

Se tutto quello che vuoi fare è distribuire un modulo chiamato foo, contenuto in un file foo.py, allora lo script di installazione può essere semplice come questo:

```
from distutils.core import setup

setup(name='foo',
      version='1.0',
      py_modules=['foo'],
      )
```

Per creare una distribuzione di origine per questo modulo, devi creare uno script di installazione, setup.py, contenente il codice precedente ed eseguire questo comando da un terminale:

```
python setup.py sdist
```

sdist creerà un file di archivio (ad esempio, tarball su Unix, file ZIP su Windows) contenente lo script di installazione setup.py e il modulo foo.py. Il file di archivio sarà chiamato foo-1.0.tar.gz (o .zip) e decomprimerà in una directory foo-1.0.

Se un utente finale desidera installare il tuo modulo foo, tutto ciò che deve fare è scaricare foo-1.0.tar.gz (o .zip), decomprimerlo e-dalla directory foo-1.0-run

```
python setup.py install
```

Aggiunta di script da riga di comando al pacchetto python

Gli script della riga di comando all'interno dei pacchetti python sono comuni. Puoi organizzare il tuo pacchetto in modo tale che quando un utente installa il pacchetto, lo script sarà disponibile sul loro percorso.

Se avessi il pacchetto dei `greetings` lo script della riga di comando `hello_world.py` .

```
greetings/
   greetings/
      __init__.py
      hello_world.py
```

È possibile eseguire quello script eseguendo:

```
python greetings/greetings/hello_world.py
```

Tuttavia se desideri eseguirlo in questo modo:

```
hello_world.py
```

È possibile ottenere ciò aggiungendo `scripts` al proprio `setup()` in `setup.py` questo modo:

```
from setuptools import setup
setup(
  name='greetings',
  scripts=['hello_world.py']
)
```

Quando installi il pacchetto di auguri ora, `hello_world.py` verrà aggiunto al tuo percorso.

Un'altra possibilità potrebbe essere quella di aggiungere un punto di ingresso:

```
entry_points={'console_scripts': ['greetings=greetings.hello_world:main']}
```

In questo modo devi solo eseguirlo come:

```
greetings
```

Utilizzo dei metadati del controllo del codice sorgente in setup.py

`setuptools_scm` è un pacchetto ufficialmente benedetto che può utilizzare i metadati Git o Mercurial per determinare il numero di versione del pacchetto e trovare i pacchetti Python e i dati del pacchetto da includere in esso.

```
from setuptools import setup, find_packages

setup(
    setup_requires=['setuptools_scm'],
    use_scm_version=True,
    packages=find_packages(),
    include_package_data=True,
)
```

Questo esempio utilizza entrambe le funzionalità; per utilizzare solo i metadati SCM per la versione, sostituire la chiamata a `find_packages()` con l'elenco dei pacchetti manuale o utilizzare solo il finder dei pacchetti, rimuovere `use_scm_version=True` .

Aggiunta di opzioni di installazione

Come visto negli esempi precedenti, l'uso di base di questo script è:

```
python setup.py install
```

Ma ci sono ancora più opzioni, come installare il pacchetto e avere la possibilità di cambiare il codice e testarlo senza doverlo reinstallare. Questo è fatto usando:

```
python setup.py develop
```

Se vuoi eseguire azioni specifiche come compilare una documentazione di *Sphinx* o creare un codice *fortran* , puoi creare la tua opzione in questo modo:

```
cmdclasses = dict()

class BuildSphinx(Command):

    """Build Sphinx documentation."""

    description = 'Build Sphinx documentation'
    user_options = []

    def initialize_options(self):
        pass

    def finalize_options(self):
        pass

    def run(self):
        import sphinx
        sphinx.build_main(['setup.py', '-b', 'html', './doc', './doc/_build/html'])
        sphinx.build_main(['setup.py', '-b', 'man', './doc', './doc/_build/man'])
```

```
cmdclasses['build_sphinx'] = BuildSphinx

setup(
...
cmdclass=cmdclasses,
)
```

`initialize_options` e `finalize_options` saranno eseguite prima e dopo la funzione di `run` come i loro nomi lo suggeriscono.

Dopodiché, potrai chiamare la tua opzione:

```
python setup.py build_sphinx
```

Leggi setup.py online: https://riptutorial.com/it/python/topic/1444/setup-py

Capitolo 180: Sicurezza e crittografia

introduzione

Python, essendo uno dei linguaggi più popolari nella sicurezza di computer e di rete, ha un grande potenziale in sicurezza e crittografia. Questo argomento tratta le funzionalità e le implementazioni crittografiche in Python dai suoi usi in sicurezza di computer e di rete a algoritmi di hashing e crittografia / decrittografia.

Sintassi

- hashlib.new (nome)
- hashlib.pbkdf2_hmac (nome, password, sale, colpi, dklen = Nessuno)

Osservazioni

Molti dei metodi in `hashlib` richiedono il passaggio di valori interpretabili come buffer di byte anziché stringhe. Questo è il caso di `hashlib.new().update()` e di `hashlib.pbkdf2_hmac` . Se si dispone di una stringa, è possibile convertirla in un buffer di byte anteponendo il carattere `b` all'inizio della stringa:

```
  "This is a string"
 b"This is a buffer of bytes"
```

Examples

Calcolo di un digest di messaggi

Il modulo `hashlib` consente di creare generatori di digest di messaggi tramite il `new` metodo. Questi generatori trasformeranno una stringa arbitraria in un digest a lunghezza fissa:

```
import hashlib

h = hashlib.new('sha256')
h.update(b'Nobody expects the Spanish Inquisition.')
h.digest()
# ==>
b'.\xdf\xda\xdaVR[\x12\x90\xff\x16\xfb\x17D\xcf\xb4\x82\xdd)\x14\xff\xbc\xb6Iy\x0c\x0eX\x9eF-
='
```

Nota che puoi chiamare `update` un numero arbitrario di volte prima di chiamare `digest` che è utile per cancellare un grosso file di blocco. Puoi anche ottenere il digest in formato esadecimale usando `hexdigest` :

```
h.hexdigest()
```

```
# ==> '2edfdada56525b1290ff16fb1744cfb482dd2914ffbcb649790c0e589e462d3d'
```

Algoritmi di hash disponibili

`hashlib.new` richiede il nome di un algoritmo quando lo chiami per produrre un generatore. Per scoprire quali algoritmi sono disponibili nell'attuale interprete Python, usa `hashlib.algorithms_available` :

```
import hashlib
hashlib.algorithms_available
# ==> {'sha256', 'DSA-SHA', 'SHA512', 'SHA224', 'dsaWithSHA', 'SHA', 'RIPEMD160', 'ecdsa-with-
SHA1', 'sha1', 'SHA384', 'md5', 'SHA1', 'MD5', 'MD4', 'SHA256', 'sha384', 'md4', 'ripemd160',
'sha224', 'sha512', 'DSA', 'dsaEncryption', 'sha', 'whirlpool'}
```

L'elenco restituito varierà in base alla piattaforma e all'interprete; assicurati di controllare che l'algoritmo sia disponibile.

Ci sono anche alcuni algoritmi che sono *garantiti* per essere disponibili su tutte le piattaforme e gli interpreti, che sono disponibili usando `hashlib.algorithms_guaranteed` :

```
hashlib.algorithms_guaranteed
# ==> {'sha256', 'sha384', 'sha1', 'sha224', 'md5', 'sha512'}
```

Hash password sicura

L' algoritmo PBKDF2 esposto dal modulo `hashlib` può essere utilizzato per eseguire l'hashing della password sicura. Sebbene questo algoritmo non possa impedire attacchi di forza bruta per recuperare la password originale dall'hash memorizzato, rende tali attacchi molto costosi.

```
import hashlib
import os

salt = os.urandom(16)
hash = hashlib.pbkdf2_hmac('sha256', b'password', salt, 100000)
```

PBKDF2 può funzionare con qualsiasi algoritmo di digest, l'esempio precedente usa SHA256 che è generalmente raccomandato. Il sale casuale deve essere memorizzato insieme alla password con hash, sarà necessario nuovamente per confrontare una password inserita con l'hash memorizzato. È essenziale che ogni password sia sottoposta a hash con un diverso salt. Per quanto riguarda il numero di round, si consiglia di impostarlo il più alto possibile per la tua applicazione .

Se si desidera il risultato in formato esadecimale, è possibile utilizzare il modulo `binascii` :

```
import binascii
hexhash = binascii.hexlify(hash)
```

Nota : mentre PBKDF2 non è male, bcrypt e soprattutto scrypt sono considerati più forti contro gli attacchi a forza bruta. Al momento non fanno parte della libreria standard Python.

File Hashing

Un hash è una funzione che converte una sequenza di lunghezza variabile di byte in una sequenza di lunghezza fissa. I file di hash possono essere vantaggiosi per molte ragioni. Gli hash possono essere usati per verificare se due file sono identici o per verificare che il contenuto di un file non sia stato corrotto o modificato.

È possibile utilizzare `hashlib` per generare un hash per un file:

```
import hashlib

hasher = hashlib.new('sha256')
with open('myfile', 'r') as f:
    contents = f.read()
    hasher.update(contents)

print hasher.hexdigest()
```

Per i file più grandi, è possibile utilizzare un buffer di lunghezza fissa:

```
import hashlib
SIZE = 65536
hasher = hashlib.new('sha256')
with open('myfile', 'r') as f:
    buffer = f.read(SIZE)
    while len(buffer) > 0:
        hasher.update(buffer)
        buffer = f.read(SIZE)
print(hasher.hexdigest())
```

Crittografia simmetrica usando pycrypto

La funzionalità crittografica incorporata di Python è attualmente limitata all'hashing. La crittografia richiede un modulo di terze parti come pycrypto . Ad esempio, fornisce l' algoritmo AES che è considerato lo stato dell'arte per la crittografia simmetrica. Il seguente codice crittograferà un dato messaggio usando una passphrase:

```
import hashlib
import math
import os

from Crypto.Cipher import AES

IV_SIZE = 16    # 128 bit, fixed for the AES algorithm
KEY_SIZE = 32   # 256 bit meaning AES-256, can also be 128 or 192 bits
SALT_SIZE = 16  # This size is arbitrary

cleartext = b'Lorem ipsum'
password = b'highly secure encryption password'
salt = os.urandom(SALT_SIZE)
derived = hashlib.pbkdf2_hmac('sha256', password, salt, 100000,
                              dklen=IV_SIZE + KEY_SIZE)
iv = derived[0:IV_SIZE]
```

```
key = derived[IV_SIZE:]

encrypted = salt + AES.new(key, AES.MODE_CFB, iv).encrypt(cleartext)
```

L'algoritmo AES accetta tre parametri: chiave di crittografia, vettore di inizializzazione (IV) e il messaggio effettivo da crittografare. Se si dispone di una chiave AES generata casualmente, è possibile utilizzarla direttamente e generare semplicemente un vettore di inizializzazione casuale. Tuttavia una passphrase non ha le dimensioni giuste, né sarebbe raccomandabile utilizzarla direttamente dato che non è veramente casuale e quindi ha un'entropia relativamente piccola. Al contrario, utilizziamo l' implementazione incorporata dell'algoritmo PBKDF2 per generare un vettore di inizializzazione a 128 bit e una chiave di crittografia a 256 bit dalla password.

Si noti il sale casuale che è importante avere un vettore di inizializzazione e una chiave diversi per ogni messaggio crittografato. Ciò garantisce in particolare che due messaggi uguali non determinino un testo crittografato identico, ma impedisce anche agli autori degli attacchi di riutilizzare il lavoro trascorso a indovinare una passphrase sui messaggi crittografati con un'altra passphrase. Questo sale deve essere memorizzato insieme al messaggio crittografato per ricavare lo stesso vettore di inizializzazione e la chiave per decrittografare.

Il seguente codice decodificherà nuovamente il nostro messaggio:

```
salt = encrypted[0:SALT_SIZE]
derived = hashlib.pbkdf2_hmac('sha256', password, salt, 100000,
                              dklen=IV_SIZE + KEY_SIZE)
iv = derived[0:IV_SIZE]
key = derived[IV_SIZE:]
cleartext = AES.new(key, AES.MODE_CFB, iv).decrypt(encrypted[SALT_SIZE:])
```

Generazione di firme RSA usando pycrypto

RSA può essere utilizzato per creare una firma del messaggio. Una firma valida può essere generata solo con l'accesso alla chiave RSA privata, la convalida d'altra parte è possibile solo con la chiave pubblica corrispondente. Quindi, se l'altra parte conosce la tua chiave pubblica, può verificare che il messaggio sia firmato da te e invariato, ad esempio un approccio utilizzato per la posta elettronica. Attualmente, per questa funzionalità è richiesto un modulo di terze parti come pycrypto .

```
import errno

from Crypto.Hash import SHA256
from Crypto.PublicKey import RSA
from Crypto.Signature import PKCS1_v1_5

message = b'This message is from me, I promise.'

try:
    with open('privkey.pem', 'r') as f:
        key = RSA.importKey(f.read())
except IOError as e:
    if e.errno != errno.ENOENT:
        raise
    # No private key, generate a new one. This can take a few seconds.
```

```
    key = RSA.generate(4096)
    with open('privkey.pem', 'wb') as f:
        f.write(key.exportKey('PEM'))
    with open('pubkey.pem', 'wb') as f:
        f.write(key.publickey().exportKey('PEM'))

hasher = SHA256.new(message)
signer = PKCS1_v1_5.new(key)
signature = signer.sign(hasher)
```

La verifica della firma funziona in modo simile ma utilizza la chiave pubblica anziché la chiave privata:

```
with open('pubkey.pem', 'rb') as f:
    key = RSA.importKey(f.read())
hasher = SHA256.new(message)
verifier = PKCS1_v1_5.new(key)
if verifier.verify(hasher, signature):
    print('Nice, the signature is valid!')
else:
    print('No, the message was signed with the wrong private key or modified')
```

Nota : gli esempi precedenti utilizzano l'algoritmo di firma PKCS # 1 v1.5 che è molto comune. pycrypto implementa anche il nuovo algoritmo PKCS # 1 PSS, sostituendo `PKCS1_v1_5` con `PKCS1_PSS` negli esempi dovrebbe funzionare se si desidera utilizzare quello. Attualmente sembra che ci siano poche ragioni per usarlo comunque.

Crittografia asimmetrica RSA usando pycrypto

La crittografia asimmetrica ha il vantaggio che un messaggio può essere crittografato senza scambiare una chiave segreta con il destinatario del messaggio. Il mittente ha semplicemente bisogno di conoscere la chiave pubblica dei destinatari, ciò consente di crittografare il messaggio in modo tale che solo il destinatario designato (che ha la chiave privata corrispondente) possa decrittografarlo. Attualmente, per questa funzionalità è richiesto un modulo di terze parti come pycrypto .

```
from Crypto.Cipher import PKCS1_OAEP
from Crypto.PublicKey import RSA

message = b'This is a very secret message.'

with open('pubkey.pem', 'rb') as f:
    key = RSA.importKey(f.read())
cipher = PKCS1_OAEP.new(key)
encrypted = cipher.encrypt(message)
```

Il destinatario può decifrare il messaggio, se ha la chiave privata giusta:

```
with open('privkey.pem', 'rb') as f:
    key = RSA.importKey(f.read())
cipher = PKCS1_OAEP.new(key)
decrypted = cipher.decrypt(encrypted)
```

Nota : gli esempi sopra riportati utilizzano lo schema di crittografia OAEP PKCS # 1. pycrypto implementa anche lo schema di crittografia PKCS # 1 v1.5, questo non è raccomandato per i nuovi protocolli, tuttavia a causa di avvertimenti noti .

Leggi Sicurezza e crittografia online: https://riptutorial.com/it/python/topic/2598/sicurezza-e-crittografia

Capitolo 181: Socket e crittografia / decrittografia dei messaggi tra client e server

introduzione

La crittografia viene utilizzata per motivi di sicurezza. Non ci sono molti esempi di crittografia / decodifica in Python utilizzando CTR di codifica IDEA di crittografia. **Scopo di questa documentazione:**

Estendere e implementare lo schema RSA Digital Signature nella comunicazione da stazione a stazione. Utilizzando l'hashing per l'integrità del messaggio, questo è SHA-1. Produce un semplice protocollo Key Transport. Cripta chiave con crittografia IDEA. La modalità di Block Cipher è Counter Mode

Osservazioni

Lingua utilizzata: Python 2.7 (Link per il download: https://www.python.org/downloads/)

Libreria utilizzata:

* **PyCrypto** (Link per il download: https://pypi.python.org/pypi/pycrypto)

* **PyCryptoPlus** (Link per il download: https://github.com/doegox/python-cryptoplus)

Installazione della libreria:

PyCrypto: decomprimere il file. Vai alla directory e apri il terminale per linux (alt + ctrl + t) e CMD (shift + tasto destro + seleziona prompt dei comandi aperto qui) per windows. Dopodiché scrivi python setup.py install (Make Sure Python Environment è impostato correttamente nel sistema operativo Windows)

PyCryptoPlus: come l'ultima libreria.

Implementazione delle attività: l'attività è suddivisa in due parti. Uno è il processo di handshake e un altro è il processo di comunicazione. Impostazione socket:

- Come la creazione di chiavi pubbliche e private e l'hashing della chiave pubblica, ora dobbiamo configurare il socket. Per configurare il socket, dobbiamo importare un altro modulo con "import socket" e connettere (per client) o bind (per server) l'indirizzo IP e la porta con il socket che viene prelevato dall'utente.

  ----------Dalla parte del cliente----------

```
server = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
host = raw_input("Server Address To Be Connected -> ")
port = int(input("Port of The Server -> "))
```

```
server.connect((host, port))
```

----------Lato server---------

```
try:
#setting up socket
server = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
server.bind((host,port))
server.listen(5)
except BaseException: print "-----Check Server Address or Port-----"
```

"Socket.AF_INET, socket.SOCK_STREAM" ci consentirà di utilizzare la funzione accept () e i fondamentali della messaggistica. Invece, possiamo usare "socket.AF_INET, socket.SOCK_DGRAM" anche se questa volta dovremo usare il setblocking (valore).

Processo di handshake:

- (CLIENT) Il primo compito è creare una chiave pubblica e privata. Per creare la chiave privata e pubblica, dobbiamo importare alcuni moduli. Sono: da Crypto import Random e da Crypto.PublicKey import RSA. Per creare le chiavi, dobbiamo scrivere poche semplici righe di codici:

```
random_generator = Random.new().read
        key = RSA.generate(1024,random_generator)
        public = key.publickey().exportKey()
```

random_generator è derivato dal modulo " **from Crypto import Random** ". La chiave è derivata da " **da Crypto.PublicKey import RSA** " che creerà una chiave privata, dimensione di 1024 generando caratteri casuali. Il pubblico sta esportando la chiave pubblica dalla chiave privata precedentemente generata.

- (CLIENT) Dopo aver creato la chiave pubblica e privata, dobbiamo hash la chiave pubblica da inviare al server usando l'hash SHA-1. Per usare l'hash SHA-1 dobbiamo importare un altro modulo scrivendo "import hashlib". Per cancellare la chiave pubblica abbiamo scritto due righe di codice:

```
hash_object = hashlib.sha1(public)
hex_digest = hash_object.hexdigest()
```

Qui hash_object e hex_digest è la nostra variabile. Dopodiché, il client invierà hex_digest e pubblico al server e Server li verificherà confrontando l'hash ottenuto dal client e il nuovo hash della chiave pubblica. Se il nuovo hash e l'hash del client corrispondono, passerà alla procedura successiva. Dato che il pubblico inviato dal client è in forma di stringa, non potrà essere utilizzato come chiave sul lato server. Per evitare ciò e convertire la chiave pubblica stringa in chiave pubblica rsa, dobbiamo scrivere `server_public_key = RSA.importKey(getpbk)` , qui getpbk è la chiave pubblica del client.

- (SERVER) Il prossimo passo è creare una chiave di sessione. Qui, ho usato il modulo "os"

per creare una chiave casuale "key = os.urandom (16)" che ci darà una chiave lunga 16 bit e dopo di che ho crittografato quella chiave in "AES.MODE_CTR" e l'ho cancellato di nuovo con SHA-1:

```
 #encrypt CTR MODE session key
 en = AES.new(key_128,AES.MODE_CTR,counter = lambda:key_128) encrypto =
en.encrypt(key_128)
 #hashing sha1
 en_object = hashlib.sha1(encrypto)
 en_digest = en_object.hexdigest()
```

Quindi en_digest sarà la nostra chiave di sessione.

- (SERVER) Per la parte finale del processo di handshake è necessario crittografare la chiave pubblica ricevuta dal client e la chiave di sessione creata sul lato server.

```
 #encrypting session key and public key
 E = server_public_key.encrypt(encrypto,16)
```

Dopo la crittografia, il server invierà la chiave al client come stringa.

- (CLIENT) Dopo aver ottenuto la stringa crittografata di (chiave pubblica e di sessione) dal server, il client la decrittografa utilizzando la chiave privata che è stata creata in precedenza insieme alla chiave pubblica. Dato che la chiave crittografata (pubblica e di sessione) era in forma di stringa, ora dobbiamo recuperarla come chiave utilizzando eval (). Se la decrittografia è stata eseguita, il processo di handshake è completato anche quando entrambi i lati confermano che stanno utilizzando le stesse chiavi. Per decifrare:

```
 en = eval(msg)
 decrypt = key.decrypt(en)
 # hashing sha1
 en_object = hashlib.sha1(decrypt) en_digest = en_object.hexdigest()
```

Ho usato lo SHA-1 qui in modo che sia leggibile nell'output.

Processo di comunicazione:

Per il processo di comunicazione, dobbiamo utilizzare la chiave di sessione da entrambi i lati come KEY per la crittografia IDEA MODE_CTR. Entrambe le parti cripteranno e decodificheranno i messaggi con IDEA.MODE_CTR utilizzando la chiave di sessione.

- (Crittografia) Per la crittografia IDEA, abbiamo bisogno di una chiave di 16 bit in dimensioni e contatore come deve essere richiamabile. Il contatore è obbligatorio in MODE_CTR. La chiave di sessione che abbiamo crittografato e con hash è ora una dimensione di 40 che supererà la chiave di limite della crittografia IDEA. Quindi, abbiamo bisogno di ridurre la dimensione della chiave di sessione. Per la riduzione, possiamo usare la normale stringa di funzione python incorporata [valore: valore]. Dove il valore può essere qualsiasi valore in base alla scelta dell'utente. Nel nostro caso, ho fatto "key [: 16]" dove ci vorranno da 0 a 16 valori dalla chiave. Questa conversione può essere eseguita in molti modi come chiave [1:17] o chiave [16:]. La parte successiva è creare una nuova funzione di crittografia IDEA

scrivendo IDEA.new () che richiederà 3 argomenti per l'elaborazione. Il primo argomento sarà KEY, il secondo argomento sarà la modalità della crittografia IDEA (nel nostro caso, IDEA.MODE_CTR) e il terzo argomento sarà il counter = che è una funzione callable obbligatoria. Il contatore = manterrà una dimensione di stringa che verrà restituita dalla funzione. Per definire il contatore =, dobbiamo usare valori ragionevoli. In questo caso, ho usato la dimensione della KEY definendo lambda. Invece di usare lambda, potremmo usare Counter.Util che genera un valore casuale per counter =. Per utilizzare Counter.Util, è necessario importare il modulo contatore da crypto. Quindi, il codice sarà:

```
ideaEncrypt = IDEA.new(key, IDEA.MODE_CTR, counter=lambda : key)
```

Una volta che definiamo "ideaEncrypt" come la nostra variabile di crittografia IDEA, possiamo usare la funzione di crittografia integrata per crittografare qualsiasi messaggio.

```
eMsg = ideaEncrypt.encrypt(whole)
#converting the encrypted message to HEXADECIMAL to readable eMsg =
eMsg.encode("hex").upper()
```

In questo segmento di codice, intero è il messaggio da crittografare e eMsg è il messaggio crittografato. Dopo aver crittografato il messaggio, l'ho convertito in HEXADECIMAL per renderlo leggibile e upper () è la funzione incorporata per rendere i caratteri maiuscoli. Successivamente, questo messaggio crittografato verrà inviato alla stazione opposta per la decrittografia.

- **(Decodificazione)**

Per decodificare i messaggi crittografati, sarà necessario creare un'altra variabile di crittografia utilizzando gli stessi argomenti e la stessa chiave, ma questa volta la variabile decodificherà i messaggi crittografati. Il codice per questo come l'ultima volta. Tuttavia, prima di decifrare i messaggi, abbiamo bisogno di decodificare il messaggio da esadecimale perché nella nostra parte di crittografia abbiamo codificato il messaggio crittografato in esadecimale per renderlo leggibile. Quindi, l'intero codice sarà:

```
decoded = newmess.decode("hex")
ideaDecrypt = IDEA.new(key, IDEA.MODE_CTR, counter=lambda: key)
dMsg = ideaDecrypt.decrypt(decoded)
```

Questi processi saranno eseguiti sia lato server che lato client per la crittografia e la decrittografia.

Examples

Implementazione lato server

```
import socket
import hashlib
import os
import time
import itertools
import threading
import sys
```

```
import Crypto.Cipher.AES as AES
from Crypto.PublicKey import RSA
from CryptoPlus.Cipher import IDEA

#server address and port number input from admin
host= raw_input("Server Address - > ")
port = int(input("Port - > "))
#boolean for checking server and port
check = False
done = False

def animate():
    for c in itertools.cycle(['....','.......','..........','............']):
        if done:
            break
        sys.stdout.write('\rCHECKING IP ADDRESS AND NOT USED PORT '+c)
        sys.stdout.flush()
        time.sleep(0.1)
    sys.stdout.write('\r -----SERVER STARTED. WAITING FOR CLIENT-----\n')
try:
    #setting up socket
    server = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
    server.bind((host,port))
    server.listen(5)
    check = True
except BaseException:
    print "-----Check Server Address or Port-----"
    check = False

if check is True:
    # server Quit
    shutdown = False
# printing "Server Started Message"
thread_load = threading.Thread(target=animate)
thread_load.start()

time.sleep(4)
done = True
#binding client and address
client,address = server.accept()
print ("CLIENT IS CONNECTED. CLIENT'S ADDRESS ->",address)
print ("\n-----WAITING FOR PUBLIC KEY & PUBLIC KEY HASH-----\n")

#client's message(Public Key)
getpbk = client.recv(2048)

#conversion of string to KEY
server_public_key = RSA.importKey(getpbk)

#hashing the public key in server side for validating the hash from client
hash_object = hashlib.sha1(getpbk)
hex_digest = hash_object.hexdigest()

if getpbk != "":
    print (getpbk)
    client.send("YES")
    gethash = client.recv(1024)
    print ("\n-----HASH OF PUBLIC KEY----- \n"+gethash)
if hex_digest == gethash:
    # creating session key
    key_128 = os.urandom(16)
```

```
    #encrypt CTR MODE session key
    en = AES.new(key_128,AES.MODE_CTR,counter = lambda:key_128)
    encrypto = en.encrypt(key_128)
    #hashing sha1
    en_object = hashlib.sha1(encrypto)
    en_digest = en_object.hexdigest()

    print ("\n-----SESSION KEY-----\n"+en_digest)

    #encrypting session key and public key
    E = server_public_key.encrypt(encrypto,16)
    print ("\n-----ENCRYPTED PUBLIC KEY AND SESSION KEY-----\n"+str(E))
    print ("\n-----HANDSHAKE COMPLETE-----")
    client.send(str(E))
    while True:
        #message from client
        newmess = client.recv(1024)
        #decoding the message from HEXADECIMAL to decrypt the ecrypted version of the message
only
        decoded = newmess.decode("hex")
        #making en_digest(session_key) as the key
        key = en_digest[:16]
        print ("\nENCRYPTED MESSAGE FROM CLIENT -> "+newmess)
        #decrypting message from the client
        ideaDecrypt = IDEA.new(key, IDEA.MODE_CTR, counter=lambda: key)
        dMsg = ideaDecrypt.decrypt(decoded)
        print ("\n**New Message**  "+time.ctime(time.time()) +" > "+dMsg+"\n")
        mess = raw_input("\nMessage To Client -> ")
        if mess != "":
            ideaEncrypt = IDEA.new(key, IDEA.MODE_CTR, counter=lambda : key)
            eMsg = ideaEncrypt.encrypt(mess)
            eMsg = eMsg.encode("hex").upper()
            if eMsg != "":
                print ("ENCRYPTED MESSAGE TO CLIENT-> " + eMsg)
            client.send(eMsg)
    client.close()
else:
    print ("\n-----PUBLIC KEY HASH DOESNOT MATCH-----\n")
```

Implementazione lato client

```
import time
import socket
import threading
import hashlib
import itertools
import sys
from Crypto import Random
from Crypto.PublicKey import RSA
from CryptoPlus.Cipher import IDEA

#animating loading
done = False
def animate():
    for c in itertools.cycle(['....','.......','..........','............']):
        if done:
            break
        sys.stdout.write('\rCONFIRMING CONNECTION TO SERVER '+c)
        sys.stdout.flush()
```

```
        time.sleep(0.1)

#public key and private key
random_generator = Random.new().read
key = RSA.generate(1024,random_generator)
public = key.publickey().exportKey()
private = key.exportKey()

#hashing the public key
hash_object = hashlib.sha1(public)
hex_digest = hash_object.hexdigest()

#Setting up socket
server = socket.socket(socket.AF_INET,socket.SOCK_STREAM)

#host and port input user
host = raw_input("Server Address To Be Connected -> ")
port = int(input("Port of The Server -> "))
#binding the address and port
server.connect((host, port))
# printing "Server Started Message"
thread_load = threading.Thread(target=animate)
thread_load.start()

time.sleep(4)
done = True

def send(t,name,key):
    mess = raw_input(name + " : ")
    key = key[:16]
    #merging the message and the name
    whole = name+" : "+mess
    ideaEncrypt = IDEA.new(key, IDEA.MODE_CTR, counter=lambda : key)
    eMsg = ideaEncrypt.encrypt(whole)
    #converting the encrypted message to HEXADECIMAL to readable
    eMsg = eMsg.encode("hex").upper()
    if eMsg != "":
        print ("ENCRYPTED MESSAGE TO SERVER-> "+eMsg)
    server.send(eMsg)
def recv(t,key):
    newmess = server.recv(1024)
    print ("\nENCRYPTED MESSAGE FROM SERVER-> " + newmess)
    key = key[:16]
    decoded = newmess.decode("hex")
    ideaDecrypt = IDEA.new(key, IDEA.MODE_CTR, counter=lambda: key)
    dMsg = ideaDecrypt.decrypt(decoded)
    print ("\n**New Message From Server**  " + time.ctime(time.time()) + " : " + dMsg + "\n")

while True:
    server.send(public)
    confirm = server.recv(1024)
    if confirm == "YES":
        server.send(hex_digest)

    #connected msg
    msg = server.recv(1024)
    en = eval(msg)
    decrypt = key.decrypt(en)
    # hashing sha1
    en_object = hashlib.sha1(decrypt)
    en_digest = en_object.hexdigest()
```

```
    print ("\n-----ENCRYPTED PUBLIC KEY AND SESSION KEY FROM SERVER-----")
    print (msg)
    print ("\n-----DECRYPTED SESSION KEY-----")
    print (en_digest)
    print ("\n-----HANDSHAKE COMPLETE-----\n")
    alais = raw_input("\nYour Name -> ")

    while True:
        thread_send = threading.Thread(target=send,args=("------Sending Message------
",alais,en_digest))
        thread_recv = threading.Thread(target=recv,args=("------Recieving Message------
",en_digest))
        thread_send.start()
        thread_recv.start()

        thread_send.join()
        thread_recv.join()
        time.sleep(0.5)
    time.sleep(60)
    server.close()
```

Leggi Socket e crittografia / decrittografia dei messaggi tra client e server online: https://riptutorial.com/it/python/topic/8710/socket-e-crittografia---decrittografia-dei-messaggi-tra-client-e-server

Capitolo 182: Sockets

introduzione

Molti linguaggi di programmazione utilizzano socket per comunicare tra processi o tra dispositivi. Questo argomento spiega come utilizzare correttamente il modulo socket in Python per facilitare l'invio e la ricezione di dati su protocolli di rete comuni.

Parametri

Parametro	Descrizione
socket.AF_UNIX	Socket UNIX
socket.AF_INET	IPv4
socket.AF_INET6	IPv6
socket.SOCK_STREAM	TCP
socket.SOCK_DGRAM	UDP

Examples

Invio di dati tramite UDP

UDP è un protocollo senza connessione. I messaggi ad altri processi o computer vengono inviati senza stabilire alcun tipo di connessione. Non c'è conferma automatica se il tuo messaggio è stato ricevuto. UDP viene solitamente utilizzato in applicazioni sensibili alla latenza o in applicazioni che inviano trasmissioni di rete.

Il seguente codice invia un messaggio ad un processo in ascolto sulla porta localhost 6667 usando UDP

Si noti che non è necessario "chiudere" il socket dopo l'invio, poiché UDP è senza connessione .

```
from socket import socket, AF_INET, SOCK_DGRAM
s = socket(AF_INET, SOCK_DGRAM)
msg = ("Hello you there!").encode('utf-8')  # socket.sendto() takes bytes as input, hence we
must encode the string first.
s.sendto(msg, ('localhost', 6667))
```

Ricezione di dati tramite UDP

UDP è un protocollo senza connessione. Ciò significa che i peer che inviano messaggi non

richiedono di stabilire una connessione prima di inviare messaggi. `socket.recvfrom` restituisce quindi una tupla (`msg` [il messaggio ricevuto dal socket], `addr` [l'indirizzo del mittente])

Un server UDP che utilizza esclusivamente il modulo `socket` :

```
from socket import socket, AF_INET, SOCK_DGRAM
sock = socket(AF_INET, SOCK_DGRAM)
sock.bind(('localhost', 6667))

while True:
    msg, addr = sock.recvfrom(8192)  # This is the amount of bytes to read at maximum
    print("Got message from %s: %s" % (addr, msg))
```

Di seguito è riportata un'implementazione alternativa utilizzando `socketserver.UDPServer` :

```
from socketserver import BaseRequestHandler, UDPServer

class MyHandler(BaseRequestHandler):
    def handle(self):
        print("Got connection from: %s" % self.client_address)
        msg, sock = self.request
        print("It said: %s" % msg)
        sock.sendto("Got your message!".encode(), self.client_address) # Send reply

serv = UDPServer(('localhost', 6667), MyHandler)
serv.serve_forever()
```

Di default, blocco `sockets` . Ciò significa che l'esecuzione dello script attenderà fino a quando il socket riceve i dati.

Invio di dati tramite TCP

L'invio di dati su Internet è reso possibile utilizzando più moduli. Il modulo socket fornisce un accesso a basso livello alle operazioni del sistema operativo sottostante responsabili dell'invio o della ricezione di dati da altri computer o processi.

Il seguente codice invia la stringa di byte `b'Hello'` a un server TCP in ascolto sulla porta 6667 sull'host local host e chiude la connessione al termine:

```
from socket import socket, AF_INET, SOCK_STREAM
s = socket(AF_INET, SOCK_STREAM)
s.connect(('localhost', 6667))  # The address of the TCP server listening
s.send(b'Hello')
s.close()
```

L'uscita socket sta bloccando di default, il che significa che il programma attenderà il collegamento e invierà le chiamate fino a quando l'azione non sarà completata. Per la connessione significa che il server effettivamente accetta la connessione. Per inviarlo significa solo che il sistema operativo ha abbastanza spazio nel buffer per accodare i dati da inviare in seguito.

Le prese dovrebbero essere sempre chiuse dopo l'uso.

TCP Socket Server multi-threaded

Quando viene eseguito senza argomenti, questo programma avvia un server socket TCP che ascolta le connessioni a `127.0.0.1` sulla porta `5000` . Il server gestisce ciascuna connessione in un thread separato.

Quando viene eseguito con l'argomento `-c` , questo programma si collega al server, legge l'elenco dei client e lo stampa. L'elenco dei client viene trasferito come una stringa JSON. Il nome del client può essere specificato passando l'argomento `-n` . Passando nomi diversi, si può osservare l'effetto sulla lista dei clienti.

client_list.py

```
import argparse
import json
import socket
import threading

def handle_client(client_list, conn, address):
    name = conn.recv(1024)
    entry = dict(zip(['name', 'address', 'port'], [name, address[0], address[1]]))
    client_list[name] = entry
    conn.sendall(json.dumps(client_list))
    conn.shutdown(socket.SHUT_RDWR)
    conn.close()

def server(client_list):
    print "Starting server..."
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    s.bind(('127.0.0.1', 5000))
    s.listen(5)
    while True:
        (conn, address) = s.accept()
        t = threading.Thread(target=handle_client, args=(client_list, conn, address))
        t.daemon = True
        t.start()

def client(name):
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect(('127.0.0.1', 5000))
    s.send(name)
    data = s.recv(1024)
    result = json.loads(data)
    print json.dumps(result, indent=4)

def parse_arguments():
    parser = argparse.ArgumentParser()
    parser.add_argument('-c', dest='client', action='store_true')
    parser.add_argument('-n', dest='name', type=str, default='name')
    result = parser.parse_args()
    return result

def main():
    client_list = dict()
    args = parse_arguments()
    if args.client:
```

```
        client(args.name)
    else:
        try:
            server(client_list)
        except KeyboardInterrupt:
            print "Keyboard interrupt"

if __name__ == '__main__':
    main()
```

Uscita del server

```
$ python client_list.py
Starting server...
```

Uscita client

```
$ python client_list.py -c -n name1
{
    "name1": {
        "address": "127.0.0.1",
        "port": 62210,
        "name": "name1"
    }
}
```

I buffer di ricezione sono limitati a 1024 byte. Se la rappresentazione della stringa JSON dell'elenco dei client supera questa dimensione, verrà troncata. Ciò causerà la seguente eccezione:

```
ValueError: Unterminated string starting at: line 1 column 1023 (char 1022)
```

Socket raw su Linux

Innanzitutto disabiliti il checksum automatico della scheda di rete:

```
sudo ethtool -K eth1 tx off
```

Quindi invia il tuo pacchetto, utilizzando un socket SOCK_RAW:

```
#!/usr/bin/env python
from socket import socket, AF_PACKET, SOCK_RAW
s = socket(AF_PACKET, SOCK_RAW)
s.bind(("eth1", 0))

# We're putting together an ethernet frame here,
# but you could have anything you want instead
# Have a look at the 'struct' module for more
# flexible packing/unpacking of binary data
# and 'binascii' for 32 bit CRC
src_addr = "\x01\x02\x03\x04\x05\x06"
dst_addr = "\x01\x02\x03\x04\x05\x06"
payload = ("["*30)+"PAYLOAD"+("]"*30)
```

```
checksum = "\x1a\x2b\x3c\x4d"
ethertype = "\x08\x01"

s.send(dst_addr+src_addr+ethertype+payload+checksum)
```

Leggi Sockets online: https://riptutorial.com/it/python/topic/1530/sockets

Capitolo 183: Somiglianze nella sintassi, differenze di significato: Python vs. JavaScript

introduzione

A volte capita che due lingue attribuiscano significati diversi alla stessa espressione di sintassi o simile. Quando entrambe le lingue sono di interesse per un programmatore, chiarire questi punti di biforcazione aiuta a comprendere meglio entrambe le lingue nelle loro basi e sottigliezze.

Examples

`in` con le liste

```
2 in [2, 3]
```

In Python questo vale per Vero, ma in JavaScript per falso. Questo perché in Python controlla se un valore è contenuto in una lista, quindi 2 è in [2, 3] come suo primo elemento. In JavaScript in viene utilizzato con oggetti e verifica se un oggetto contiene la proprietà con il nome espresso dal valore. Quindi JavaScript considera [2, 3] come un oggetto o una mappa valore-chiave come questa:

```
{'0': 2, '1': 3}
```

e controlla se ha una proprietà o una chiave '2' in essa. Intero 2 viene convertito in modo silenzioso alla stringa "2".

Leggi Somiglianze nella sintassi, differenze di significato: Python vs. JavaScript online: https://riptutorial.com/it/python/topic/10766/somiglianze-nella-sintassi--differenze-di-significato--python-vs--javascript

Capitolo 184: Sottocomandi CLI con output di aiuto preciso

introduzione

Diversi modi per creare sottocomandi come in `hg` o `svn` con l'esatta interfaccia a riga di comando e l'output di aiuto come mostrato nella sezione Commenti.

L'analisi degli argomenti della riga di comando copre argomenti più ampi di analisi degli argomenti.

Osservazioni

Diversi modi per creare sottocomandi come in `hg` o `svn` con l'interfaccia a riga di comando mostrata nel messaggio di aiuto:

```
usage: sub <command>

commands:

  status -  show status
  list   -  print list
```

Examples

Modo nativo (senza librerie)

```
"""
usage: sub <command>

commands:

  status -  show status
  list   -  print list
"""

import sys

def check():
    print("status")
    return 0

if sys.argv[1:] == ['status']:
    sys.exit(check())
elif sys.argv[1:] == ['list']:
    print("list")
else:
    print(__doc__.strip())
```

Uscita senza argomenti:

```
usage: sub <command>

commands:

  status -  show status
  list   -  print list
```

Professionisti:

- no deps
- tutti dovrebbero essere in grado di leggerlo
- controllo completo sulla formattazione della guida

argparse (formattatore di aiuto predefinito)

```
import argparse
import sys

def check():
    print("status")
    return 0

parser = argparse.ArgumentParser(prog="sub", add_help=False)
subparser = parser.add_subparsers(dest="cmd")

subparser.add_parser('status', help='show status')
subparser.add_parser('list', help='print list')

# hack to show help when no arguments supplied
if len(sys.argv) == 1:
    parser.print_help()
    sys.exit(0)

args = parser.parse_args()

if args.cmd == 'list':
    print('list')
elif args.cmd == 'status':
    sys.exit(check())
```

Uscita senza argomenti:

```
usage: sub {status,list} ...

positional arguments:
  {status,list}
    status       show status
    list         print list
```

Professionisti:

- viene fornito con Python
- l'analisi delle opzioni è inclusa

argparse (formattatore di aiuto personalizzato)

Versione estesa di http://www.Scriptutorial.com/python/example/25282/argparse--default-help-formatter- che ha risolto l'output della guida.

```
import argparse
import sys

class CustomHelpFormatter(argparse.HelpFormatter):
    def _format_action(self, action):
        if type(action) == argparse._SubParsersAction:
            # inject new class variable for subcommand formatting
            subactions = action._get_subactions()
            invocations = [self._format_action_invocation(a) for a in subactions]
            self._subcommand_max_length = max(len(i) for i in invocations)

        if type(action) == argparse._SubParsersAction._ChoicesPseudoAction:
            # format subcommand help line
            subcommand = self._format_action_invocation(action) # type: str
            width = self._subcommand_max_length
            help_text = ""
            if action.help:
                help_text = self._expand_help(action)
            return "  {:{width}} -  {}\n".format(subcommand, help_text, width=width)

        elif type(action) == argparse._SubParsersAction:
            # process subcommand help section
            msg = '\n'
            for subaction in action._get_subactions():
                msg += self._format_action(subaction)
            return msg
        else:
            return super(CustomHelpFormatter, self)._format_action(action)


def check():
    print("status")
    return 0

parser = argparse.ArgumentParser(usage="sub <command>", add_help=False,
             formatter_class=CustomHelpFormatter)

subparser = parser.add_subparsers(dest="cmd")
subparser.add_parser('status', help='show status')
subparser.add_parser('list', help='print list')

# custom help messge
parser._positionals.title = "commands"

# hack to show help when no arguments supplied
if len(sys.argv) == 1:
    parser.print_help()
    sys.exit(0)

args = parser.parse_args()

if args.cmd == 'list':
    print('list')
elif args.cmd == 'status':
```

```
    sys.exit(check())
```

Uscita senza argomenti:

```
usage: sub <command>

commands:

  status -  show status
  list   -  print list
```

Leggi Sottocomandi CLI con output di aiuto preciso online:
https://riptutorial.com/it/python/topic/7701/sottocomandi-cli-con-output-di-aiuto-preciso

Capitolo 185: Sovraccarico

Examples

Metodi Magici / Dunder

I metodi di Magic (chiamati anche dunder come abbreviazione per double-underscore) in Python hanno lo stesso scopo di sovraccaricare gli operatori in altre lingue. Consentono a una classe di definire il proprio comportamento quando viene utilizzata come operando in espressioni unate o binarie. Servono anche come implementazioni chiamate da alcune funzioni integrate.

Considerare questa implementazione di vettori bidimensionali.

```
import math

class Vector(object):
    # instantiation
    def __init__(self, x, y):
        self.x = x
        self.y = y

    # unary negation (-v)
    def __neg__(self):
        return Vector(-self.x, -self.y)

    # addition (v + u)
    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    # subtraction (v - u)
    def __sub__(self, other):
        return self + (-other)

    # equality (v == u)
    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

    # abs(v)
    def __abs__(self):
        return math.hypot(self.x, self.y)

    # str(v)
    def __str__(self):
        return '<{0.x}, {0.y}>'.format(self)

    # repr(v)
    def __repr__(self):
        return 'Vector({0.x}, {0.y})'.format(self)
```

Ora è possibile utilizzare naturalmente le istanze della classe `Vector` in varie espressioni.

```
v = Vector(1, 4)
u = Vector(2, 0)
```

```
u + v           # Vector(3, 4)
print(u + v)    # "<3, 4>" (implicit string conversion)
u - v           # Vector(1, -4)
u == v          # False
u + v == v + u  # True
abs(u + v)      # 5.0
```

Tipi di contenitori e di sequenza

È possibile emulare tipi di contenitori che supportano l'accesso ai valori per chiave o indice.

Considera questa ingenua implementazione di una lista sparsa, che memorizza solo i suoi elementi diversi da zero per conservare la memoria.

```
class sparselist(object):
    def __init__(self, size):
        self.size = size
        self.data = {}

    # l[index]
    def __getitem__(self, index):
        if index < 0:
            index += self.size
        if index >= self.size:
            raise IndexError(index)
        try:
            return self.data[index]
        except KeyError:
            return 0.0

    # l[index] = value
    def __setitem__(self, index, value):
        self.data[index] = value

    # del l[index]
    def __delitem__(self, index):
        if index in self.data:
            del self.data[index]

    # value in l
    def __contains__(self, value):
        return value == 0.0 or value in self.data.values()

    # len(l)
    def __len__(self):
        return self.size

    # for value in l: ...
    def __iter__(self):
        return (self[i] for i in range(self.size)) # use xrange for python2
```

Quindi, possiamo usare uno `sparselist` come una `list` normale.

```
l = sparselist(10 ** 6)  # list with 1 million elements
0 in l                   # True
10 in l                  # False
```

```
l[12345] = 10
10 in l                  # True
l[12345]                 # 10

for v in l:
    pass  # 0, 0, 0, ... 10, 0, 0 ... 0
```

Tipi chiamabili

```
class adder(object):
    def __init__(self, first):
        self.first = first

    # a(...)
    def __call__(self, second):
        return self.first + second

add2 = adder(2)
add2(1)  # 3
add2(2)  # 4
```

Gestione del comportamento non implementato

Se la classe non implementa uno specifico operatore sovraccarico per i tipi di argomento forniti, dovrebbe `return NotImplemented` (**notare** che questa è una costante speciale , non la stessa di `NotImplementedError`). Ciò consentirà a Python di tornare a provare altri metodi per far funzionare l'operazione:

> Quando viene restituito `NotImplemented` , l'interprete proverà l'operazione riflessa sull'altro tipo, o qualche altro fallback, a seconda dell'operatore. Se tutte le operazioni tentate restituiscono `NotImplemented` , l'interprete genererà un'eccezione appropriata.

Ad esempio, dato `x + y` , se `x.__add__(y)` restituisce non implementato, viene invece tentato `y.__radd__(x)` .

```
class NotAddable(object):

    def __init__(self, value):
        self.value = value

    def __add__(self, other):
        return NotImplemented


class Addable(NotAddable):

    def __add__(self, other):
        return Addable(self.value + other.value)

    __radd__ = __add__
```

Poiché questo è il metodo *riflesso* , dobbiamo implementare `__add__` **e** `__radd__` per ottenere il comportamento previsto in tutti i casi; fortunatamente, dato che entrambi stanno facendo la stessa

cosa in questo semplice esempio, possiamo prendere una scorciatoia.

In uso:

```
>>> x = NotAddable(1)
>>> y = Addable(2)
>>> x + x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'NotAddable' and 'NotAddable'
>>> y + y
<so.Addable object at 0x1095974d0>
>>> z = x + y
>>> z
<so.Addable object at 0x109597510>
>>> z.value
3
```

Sovraccarico dell'operatore

Di seguito sono riportati gli operatori che possono essere sovraccaricati in classi, insieme alle definizioni dei metodi richieste e un esempio dell'operatore in uso all'interno di un'espressione.

NB L'uso `other` come nome variabile non è obbligatorio, ma è considerato la norma.

Operatore	Metodo	Espressione
`+` Aggiunta	`__add__(self, other)`	`a1 + a2`
`-` Sottrazione	`__sub__(self, other)`	`a1 - a2`
`*` Moltiplicazione	`__mul__(self, other)`	`a1 * a2`
`@` Moltiplicazione della matrice	`__matmul__(self, other)`	`a1 @ a2` (*Python 3.5*)
`/` Divisione	`__div__(self, other)`	`a1 / a2` (*solo Python 2*)
`/` Divisione	`__truediv__(self, other)`	`a1 / a2` (*Python 3*)
`//` Floor Division	`__floordiv__(self, other)`	`a1 // a2`
`%` Modulo / resto	`__mod__(self, other)`	`a1 % a2`
`**` Potenza	`__pow__(self, other[, modulo])`	`a1 ** a2`
`<<` Bitwise Left Shift	`__lshift__(self, other)`	`a1 << a2`
`>>` Spostamento a destra bit a bit	`__rshift__(self, other)`	`a1 >> a2`
`&` Bitwise AND	`__and__(self, other)`	`a1 & a2`
`^` XOR bit a bit	`__xor__(self, other)`	`a1 ^ a2`

Operatore	Metodo	Espressione
`|` (OR bit a bit)	`__or__(self, other)`	`a1 | a2`
`-` Negazione (aritmetica)	`__neg__(self)`	`-a1`
`+` Positivo	`__pos__(self)`	`+a1`
`~` Bitwise NOT	`__invert__(self)`	`~a1`
`<` Meno di	`__lt__(self, other)`	`a1 < a2`
`<=` Minore o uguale a	`__le__(self, other)`	`a1 <= a2`
`==` Uguale a	`__eq__(self, other)`	`a1 == a2`
`!=` Non uguale a	`__ne__(self, other)`	`a1 != a2`
`>` Maggiore di	`__gt__(self, other)`	`a1 > a2`
`>=` Maggiore di o uguale a	`__ge__(self, other)`	`a1 >= a2`
`[index]` Operatore indice	`__getitem__(self, index)`	`a1[index]`
`in` operatore In	`__contains__(self, other)`	`a2 in a1`
`(*args, ...)` Chiamare	`__call__(self, *args, **kwargs)`	`a1(*args, **kwargs)`

Il parametro opzionale `modulo` per `__pow__` viene utilizzato solo dalla funzione integrata di `pow` .

Ciascuno dei metodi corrispondenti a un operatore *binario* ha un metodo "giusto" corrispondente che inizia con `__r` , ad esempio `__radd__` :

```
class A:
    def __init__(self, a):
        self.a = a
    def __add__(self, other):
        return self.a + other
    def __radd__(self, other):
        print("radd")
        return other + self.a

A(1) + 2  # Out:  3
2 + A(1)  # prints radd. Out: 3
```

così come una corrispondente versione inplace, a partire da `__i` :

```
class B:
    def __init__(self, b):
        self.b = b
    def __iadd__(self, other):
        self.b += other
```

```
        print("iadd")
        return self

b = B(2)
b.b       # Out: 2
b += 1    # prints iadd
b.b       # Out: 3
```

Poiché non c'è nulla di speciale in questi metodi, molte altre parti del linguaggio, parti della libreria standard e persino moduli di terze parti aggiungono metodi magici da soli, come metodi per eseguire il cast di un oggetto su un tipo o controllare le proprietà dell'oggetto. Ad esempio, la funzione builtin `str()` chiama il metodo `__str__` dell'oggetto, se esiste. Alcuni di questi usi sono elencati di seguito.

Funzione	Metodo	Espressione
Trasmissione a `int`	`__int__(self)`	`int(a1)`
Funzione assoluta	`__abs__(self)`	`abs(a1)`
Trasmissione a `str`	`__str__(self)`	`str(a1)`
Casting to `unicode`	`__unicode__(self)`	`unicode(a1)` (solo Python 2)
Rappresentazione delle stringhe	`__repr__(self)`	`repr(a1)`
Casting to `bool`	`__nonzero__(self)`	`bool(a1)`
Formattazione delle stringhe	`__format__(self, formatstr)`	`"Hi {:abc}".format(a1)`
hashing	`__hash__(self)`	`hash(a1)`
Lunghezza	`__len__(self)`	`len(a1)`
Reversed	`__reversed__(self)`	`reversed(a1)`
Pavimento	`__floor__(self)`	`math.floor(a1)`
Soffitto	`__ceil__(self)`	`math.ceil(a1)`

Esistono anche i metodi speciali `__enter__` e `__exit__` per i gestori di contesto e molti altri.

Leggi Sovraccarico online: https://riptutorial.com/it/python/topic/2063/sovraccarico

Capitolo 186: Strumento 2to3

Sintassi

- $ 2to3 [-opzioni] percorso / a / file.py

Parametri

Parametro	Descrizione
nome_file / nome_directory	2to3 accetta un elenco di file o directory che deve essere trasformato come argomento. Le directory sono attraversate in modo ricorsivo per le fonti Python.
Opzione	**Opzione Descrizione**
-f FIX, --fix = FIX	Specificare le trasformazioni da applicare; default: tutto Elenca le trasformazioni disponibili con `--list-fixes`
-j PROCESSI, --processi = PROCESSI	Esegui 2to3 contemporaneamente
-x NOFIX, --nofix = NOFIX	Escludere una trasformazione
-l, --list-fixes	Elenca le trasformazioni disponibili
-p, --print-function	Modificare la grammatica in modo che `print()` sia considerato una funzione
-v, --verbose	Output più dettagliato
--no-diff	Non emettere diffs del refactoring
-w	Scrivi i file modificati
-n, --nobackups	Non creare backup di file modificati
-o OUTPUT_DIR, --output-dir = OUTPUT_DIR	Posiziona i file di output in questa directory invece di sovrascrivere i file di input. Richiede il flag `-n` , poiché i file di backup non sono necessari quando i file di input non vengono modificati.
-W, --write-unchanged-files	Scrivi i file di output anche se non sono richieste modifiche. Utile con `-o` modo che un albero di sorgenti completo sia tradotto e copiato. Implica `-w` .
--add-suffisso =	Specificare una stringa da aggiungere a tutti i nomi file di output.

Parametro	Descrizione
ADD_SUFFIX	Richiede `-n` se non vuoto. Es .: `--add-suffix='3'` genererà file `.py3` .

Osservazioni

Lo strumento 2to3 è un programma python che viene utilizzato per convertire il codice scritto in Python 2.x in codice Python 3.x. Lo strumento legge il codice sorgente Python 2.x e applica una serie di fixer per trasformarlo in codice Python 3.x valido.

Lo strumento 2to3 è disponibile nella libreria standard come lib2to3 che contiene un ricco set di fixer che gestirà quasi tutto il codice. Dal momento che lib2to3 è una libreria generica, è possibile scrivere i propri fissatori per 2to3.

Examples

Uso di base

Si consideri il seguente codice Python2.x. Salva il file come `example.py`

Python 2.x 2.0

```
def greet(name):
    print "Hello, {0}!".format(name)
print "What's your name?"
name = raw_input()
greet(name)
```

Nel file precedente, ci sono diverse righe incompatibili. Il metodo `raw_input()` è stato sostituito con `input()` in Python 3.xe `print` non è più un'istruzione, ma una funzione. Questo codice può essere convertito in codice Python 3.x utilizzando lo strumento 2to3.

Unix

```
$ 2to3 example.py
```

finestre

```
> path/to/2to3.py example.py
```

L'esecuzione del codice sopra riportato mostrerà le differenze rispetto al file sorgente originale come mostrato di seguito.

```
RefactoringTool: Skipping implicit fixer: buffer
RefactoringTool: Skipping implicit fixer: idioms
RefactoringTool: Skipping implicit fixer: set_literal
```

```
RefactoringTool: Skipping implicit fixer: ws_comma
RefactoringTool: Refactored example.py
--- example.py    (original)
+++ example.py    (refactored)
@@ -1,5 +1,5 @@
 def greet(name):
-    print "Hello, {0}!".format(name)
-print "What's your name?"
-name = raw_input()
+    print("Hello, {0}!".format(name))
+print("What's your name?")
+name = input()
 greet(name)
RefactoringTool: Files that need to be modified:
RefactoringTool: example.py
```

Le modifiche possono essere riscritte nel file sorgente usando il flag -w. Viene creato un backup del file originale denominato `example.py.bak` , a meno che non venga fornito il flag -n.

Unix

```
$ 2to3 -w example.py
```

finestre

```
> path/to/2to3.py -w example.py
```

Ora il file `example.py` è stato convertito da Python 2.x al codice Python 3.x.

Una volta terminato, `example.py` conterrà il seguente codice Python3.x valido:

Python 3.x 3.0

```
def greet(name):
    print("Hello, {0}!".format(name))
print("What's your name?")
name = input()
greet(name)
```

Leggi Strumento 2to3 online: https://riptutorial.com/it/python/topic/5320/strumento-2to3

Capitolo 187: SYS

introduzione

Il modulo **sys** fornisce accesso a funzioni e valori relativi all'ambiente di runtime del programma, come i parametri della riga di comando in `sys.argv` o la funzione `sys.exit()` per terminare il processo corrente da qualsiasi punto nel flusso del programma.

Separato in modo pulito in un modulo, è in realtà integrato e, come tale, sarà sempre disponibile in circostanze normali.

Sintassi

- Importa il modulo sys e rendilo disponibile nello spazio dei nomi corrente:

```
import sys
```

- Importa una funzione specifica dal modulo sys direttamente nello spazio dei nomi corrente:

```
from sys import exit
```

Osservazioni

Per i dettagli su tutti i membri del modulo **sys** , fare riferimento alla documentazione ufficiale .

Examples

Argomenti della riga di comando

```
if len(sys.argv) != 4:         # The script name needs to be accounted for as well.
    raise RuntimeError("expected 3 command line arguments")

f = open(sys.argv[1], 'rb')    # Use first command line argument.
start_line = int(sys.argv[2])  # All arguments come as strings, so need to be
end_line = int(sys.argv[3])    # converted explicitly if other types are required.
```

Si noti che nei programmi più grandi e più lucidi si utilizzano moduli come il clic per gestire gli argomenti della riga di comando invece di farlo da soli.

Nome dello script

```
# The name of the executed script is at the beginning of the argv list.
print('usage:', sys.argv[0], '<filename> <start> <end>')

# You can use it to generate the path prefix of the executed program
```

```
# (as opposed to the current module) to access files relative to that,
# which would be good for assets of a game, for instance.
program_file = sys.argv[0]

import pathlib
program_path = pathlib.Path(program_file).resolve().parent
```

Flusso di errore standard

```
# Error messages should not go to standard output, if possible.
print('ERROR: We have no cheese at all.', file=sys.stderr)

try:
    f = open('nonexistent-file.xyz', 'rb')
except OSError as e:
    print(e, file=sys.stderr)
```

Termina prematuramente il processo e restituisce un codice di uscita

```
def main():
    if len(sys.argv) != 4 or '--help' in sys.argv[1:]:
        print('usage: my_program <arg1> <arg2> <arg3>', file=sys.stderr)

        sys.exit(1)    # use an exit code to signal the program was unsuccessful

    process_data()
```

Leggi SYS online: https://riptutorial.com/it/python/topic/9847/sys

Capitolo 188: tempfile NamedTemporaryFile

Parametri

param	descrizione
modalità	modalità per aprire il file, default = w + b
Elimina	Per eliminare il file alla chiusura, default = True
suffisso	nome file suffisso, default = ''
prefisso	prefisso nomefile, default = 'tmp'
dir	dirname per posizionare tempfile, default = None
buffsize	default = -1, (predefinito del sistema operativo usato)

Examples

Crea (e scrivi in) un file temporaneo persistente noto

È possibile creare file temporanei con un nome visibile sul file system a cui è possibile accedere tramite la proprietà `name` . Il file può, su sistemi unix, essere configurato per eliminare alla chiusura (impostato da `delete` param, il valore predefinito è True) o può essere riaperto in seguito.

Quanto segue creerà e aprirà un file temporaneo con nome e scriverà "Hello World!" a quel file. Il percorso file del file temporaneo è accessibile tramite il `name` , in questo esempio viene salvato nel `path` della variabile e stampato per l'utente. Il file viene quindi riaperto dopo aver chiuso il file e i contenuti del tempfile vengono letti e stampati per l'utente.

```
import tempfile

with tempfile.NamedTemporaryFile(delete=False) as t:
    t.write('Hello World!')
    path = t.name
    print path

with open(path) as t:
    print t.read()
```

Produzione:

```
/tmp/tmp6pireJ
Hello World!
```

Leggi tempfile NamedTemporaryFile online: https://riptutorial.com/it/python/topic/3666/tempfile-

namedtemporaryfile

Capitolo 189: Test unitario

Osservazioni

Esistono diversi strumenti di test delle unità per Python. Questo argomento di documentazione descrive il modulo di `unittest` base. Altri strumenti di test includono `py.test` e `nosetests` . Questa documentazione su Python confronta diversi di questi strumenti senza approfondire.

Examples

Test delle eccezioni

I programmi generano errori quando, ad esempio, viene fornito un input errato. Per questo motivo, è necessario assicurarsi che venga generato un errore quando viene fornito un input errato. Per questo motivo dobbiamo verificare un'eccezione esatta, per questo esempio useremo la seguente eccezione:

```
class WrongInputException(Exception):
    pass
```

Questa eccezione viene sollevata quando viene fornito un input errato, nel seguente contesto in cui ci si aspetta sempre un numero come input di testo.

```
def convert2number(random_input):
    try:
        my_input = int(random_input)
    except ValueError:
        raise WrongInputException("Expected an integer!")
    return my_input
```

Per verificare se è stata sollevata un'eccezione, utilizziamo `assertRaises` per verificare tale eccezione. `assertRaises` può essere utilizzato in due modi:

1. Usando la normale chiamata di funzione. Il primo argomento accetta il tipo di eccezione, il secondo un callable (di solito una funzione) e il resto degli argomenti viene passato a questo callable.
2. Usando una clausola `with` , dando solo il tipo di eccezione alla funzione. Ciò ha il vantaggio che è possibile eseguire più codice, ma dovrebbe essere usato con attenzione poiché più funzioni possono utilizzare la stessa eccezione che può essere problematica. Un esempio: con self.assertRaises (WrongInputException): convert2number ("non un numero")

Questo primo è stato implementato nel seguente caso di test:

```
import unittest

class ExceptionTestCase(unittest.TestCase):
```

```
    def test_wrong_input_string(self):
        self.assertRaises(WrongInputException, convert2number, "not a number")

    def test_correct_input(self):
        try:
            result = convert2number("56")
            self.assertIsInstance(result, int)
        except WrongInputException:
            self.fail()
```

Potrebbe anche esserci la necessità di verificare un'eccezione che non avrebbe dovuto essere generata. Tuttavia, un test fallirà automaticamente quando viene lanciata un'eccezione e quindi potrebbe non essere affatto necessaria. Solo per mostrare le opzioni, il secondo metodo di test mostra un caso su come si può verificare che non venga lanciata un'eccezione. Fondamentalmente, questo sta rilevando l'eccezione e quindi fallendo il test usando il metodo `fail` .

Funzioni di simulazione con unittest.mock.create_autospec

Un modo per prendere in giro una funzione è usare la funzione `create_autospec` , che metterà a punto un oggetto secondo le sue specifiche. Con le funzioni, possiamo usarlo per assicurarci che siano chiamati appropriatamente.

Con una funzione si `multiply` in `custom_math.py` :

```
def multiply(a, b):
    return a * b
```

E una funzione si `multiples_of` in `process_math.py` :

```
from custom_math import multiply


def multiples_of(integer, *args, num_multiples=0, **kwargs):
    """
    :rtype: list
    """
    multiples = []

    for x in range(1, num_multiples + 1):
        """
        Passing in args and kwargs here will only raise TypeError if values were
        passed to multiples_of function, otherwise they are ignored. This way we can
        test that multiples_of is used correctly. This is here for an illustration
        of how create_autospec works. Not recommended for production code.
        """
        multiple = multiply(integer,x, *args, **kwargs)
        multiples.append(multiple)

    return multiples
```

Possiamo testare i `multiples_of` da solo dividendo `multiply` . L'esempio seguente usa la libreria standard Python unittest, ma può essere usato anche con altri framework di testing, come pytest o

nose:

```
from unittest.mock import create_autospec
import unittest

# we import the entire module so we can mock out multiply
import custom_math
custom_math.multiply = create_autospec(custom_math.multiply)
from process_math import multiples_of


class TestCustomMath(unittest.TestCase):
    def test_multiples_of(self):
        multiples = multiples_of(3, num_multiples=1)
        custom_math.multiply.assert_called_with(3, 1)

    def test_multiples_of_with_bad_inputs(self):
        with self.assertRaises(TypeError) as e:
            multiples_of(1, "extra arg",  num_multiples=1) # this should raise a TypeError
```

Test Setup e Teardown in un unestest.TestCase

A volte vogliamo preparare un contesto per ogni test da eseguire. Il metodo `setUp` viene eseguito prima di ogni test della classe. `tearDown` viene eseguito alla fine di ogni test. Questi metodi sono opzionali. Ricorda che i TestCase sono spesso usati nell'ereditarietà multipla cooperativa quindi devi stare attento a chiamare sempre `super` in questi metodi in modo che anche i metodi `setUp` e `tearDown` della classe base vengano richiamati. L'implementazione base di `TestCase` fornisce metodi di `setUp` e `tearDown` vuoti in modo che possano essere richiamati senza generare eccezioni:

```
import unittest


class SomeTest(unittest.TestCase):
    def setUp(self):
        super(SomeTest, self).setUp()
        self.mock_data = [1,2,3,4,5]

    def test(self):
        self.assertEqual(len(self.mock_data), 5)

    def tearDown(self):
        super(SomeTest, self).tearDown()
        self.mock_data = []


if __name__ == '__main__':
    unittest.main()
```

Si noti che in python2.7 + esiste anche il metodo `addCleanup` che registra le funzioni da chiamare dopo l'esecuzione del test. Al contrario di `tearDown` che viene chiamato solo se `setUp` successo, le funzioni registrate tramite `addCleanup` verranno chiamate anche in caso di un'eccezione non gestita in `setUp` . Come esempio concreto, questo metodo può essere visto spesso rimuovendo vari mock che sono stati registrati mentre il test era in esecuzione:

```
import unittest
import some_module


class SomeOtherTest(unittest.TestCase):
    def setUp(self):
        super(SomeOtherTest, self).setUp()

        # Replace `some_module.method` with a `mock.Mock`
        my_patch = mock.patch.object(some_module, 'method')
        my_patch.start()

        # When the test finishes running, put the original method back.
        self.addCleanup(my_patch.stop)
```

Un altro vantaggio della registrazione delle pulizie in questo modo è che consente al programmatore di inserire il codice di pulizia accanto al codice di configurazione e che ti protegge nel caso in cui un subclasser si dimentichi di chiamare `super` in `tearDown` .

Asserendo sulle eccezioni

È possibile testare che una funzione genera un'eccezione con l'unittest integrato attraverso due metodi diversi.

Utilizzando un gestore di contesto

```
def division_function(dividend, divisor):
    return dividend / divisor


class MyTestCase(unittest.TestCase):
    def test_using_context_manager(self):
        with self.assertRaises(ZeroDivisionError):
            x = division_function(1, 0)
```

Questo eseguirà il codice all'interno del gestore di contesto e, se riesce, fallirà il test perché l'eccezione non è stata sollevata. Se il codice genera un'eccezione del tipo corretto, il test continuerà.

È anche possibile ottenere il contenuto dell'eccezione sollevata se si desidera eseguire ulteriori affermazioni su di esso.

```
class MyTestCase(unittest.TestCase):
    def test_using_context_manager(self):
        with self.assertRaises(ZeroDivisionError) as ex:
            x = division_function(1, 0)

        self.assertEqual(ex.message, 'integer division or modulo by zero')
```

Fornendo una funzione callable

```
def division_function(dividend, divisor):
    """
```

```
    Dividing two numbers.

    :type dividend: int
    :type divisor: int

    :raises: ZeroDivisionError if divisor is zero (0).
    :rtype: int
    """
    return dividend / divisor


class MyTestCase(unittest.TestCase):
    def test_passing_function(self):
        self.assertRaises(ZeroDivisionError, division_function, 1, 0)
```

L'eccezione da verificare deve essere il primo parametro e una funzione callable deve essere passata come secondo parametro. Qualsiasi altro parametro specificato verrà passato direttamente alla funzione chiamata, consentendo di specificare i parametri che attivano l'eccezione.

Scegliere le asserzioni all'interno di Unittests

Mentre Python ha un `assert` dichiarazione , il quadro unit testing Python ha una migliore affermazioni specializzati per le prove: sono più informativo sui fallimenti, e non dipendono modalità di debug della esecuzione.

Forse l'asserzione più semplice è `assertTrue` , che può essere usata in questo modo:

```
import unittest

class SimplisticTest(unittest.TestCase):
    def test_basic(self):
        self.assertTrue(1 + 1 == 2)
```

Ciò funzionerà correttamente, ma sostituendo la riga precedente con

```
        self.assertTrue(1 + 1 == 3)
```

avrà esito negativo.

L'affermazione L'affermazione `assertTrue` è molto probabilmente l'affermazione più generale, poiché qualsiasi cosa testata può essere lanciata come alcune condizioni booleane, ma spesso ci sono alternative migliori. Quando si prova per l'uguaglianza, come sopra, è meglio scrivere

```
        self.assertEqual(1 + 1, 3)
```

Quando il primo fallisce, il messaggio è

```
======================================================================

FAIL: test (__main__.TruthTest)
```

```
----------------------------------------------------------------------

Traceback (most recent call last):

  File "stuff.py", line 6, in test

    self.assertTrue(1 + 1 == 3)

AssertionError: False is not true
```

ma quando quest'ultimo fallisce, il messaggio è

```
======================================================================

FAIL: test (__main__.TruthTest)

----------------------------------------------------------------------

Traceback (most recent call last):

  File "stuff.py", line 6, in test

    self.assertEqual(1 + 1, 3)
AssertionError: 2 != 3
```

che è più informativo (in realtà ha valutato il risultato della parte sinistra).

È possibile trovare l'elenco di asserzioni nella documentazione standard . In generale, è una buona idea scegliere l'asserzione più adatta alla condizione. Quindi, come mostrato sopra, per affermare che `1 + 1 == 2` è meglio usare `assertEqual` di `assertTrue` . Allo stesso modo, per affermare che `a is None` , è meglio usare `assertIsNone` che `assertEqual` .

Si noti inoltre che le asserzioni hanno forme negative. Così `assertEqual` ha la sua controparte negativa `assertNotEqual` , e `assertIsNone` ha la sua controparte negativa `assertIsNotNone` . Ancora una volta, usando le controparti negative, se appropriato, porterà a messaggi di errore più chiari.

Test unitari con pytest

installazione pytest:

```
pip install pytest
```

preparando i test:

```
mkdir tests
touch tests/test_docker.py
```

Funzioni da testare in `docker_something/helpers.py` :

```
from subprocess import Popen, PIPE
# this Popen is monkeypatched with the fixture `all_popens`
```

```
def copy_file_to_docker(src, dest):
    try:
        result = Popen(['docker','cp', src, 'something_cont:{}'.format(dest)], stdout=PIPE,
stderr=PIPE)
        err = result.stderr.read()
        if err:
            raise Exception(err)
    except Exception as e:
        print(e)
    return result

def docker_exec_something(something_file_string):
    fl = Popen(["docker", "exec", "-i", "something_cont", "something"], stdin=PIPE,
stdout=PIPE, stderr=PIPE)
    fl.stdin.write(something_file_string)
    fl.stdin.close()
    err = fl.stderr.read()
    fl.stderr.close()
    if err:
        print(err)
        exit()
    result = fl.stdout.read()
    print(result)
```

Il test importa `test_docker.py` :

```
import os
from tempfile import NamedTemporaryFile
import pytest
from subprocess import Popen, PIPE

from docker_something import helpers
copy_file_to_docker = helpers.copy_file_to_docker
docker_exec_something = helpers.docker_exec_something
```

deridere un file come oggetto in `test_docker.py` :

```
class MockBytes():
    '''Used to collect bytes
    '''
    all_read = []
    all_write = []
    all_close = []

    def read(self, *args, **kwargs):
        # print('read', args, kwargs, dir(self))
        self.all_read.append((self, args, kwargs))

    def write(self, *args, **kwargs):
        # print('wrote', args, kwargs)
        self.all_write.append((self, args, kwargs))

    def close(self, *args, **kwargs):
        # print('closed', self, args, kwargs)
        self.all_close.append((self, args, kwargs))

    def get_all_mock_bytes(self):
        return self.all_read, self.all_write, self.all_close
```

Patch di scimmia con pytest in `test_docker.py` :

```
@pytest.fixture
def all_popens(monkeypatch):
    '''This fixture overrides / mocks the builtin Popen
        and replaces stdin, stdout, stderr with a MockBytes object

        note: monkeypatch is magically imported
    '''
    all_popens = []

    class MockPopen(object):
        def __init__(self, args, stdout=None, stdin=None, stderr=None):
            all_popens.append(self)
            self.args = args
            self.byte_collection = MockBytes()
            self.stdin = self.byte_collection
            self.stdout = self.byte_collection
            self.stderr = self.byte_collection
            pass
    monkeypatch.setattr(helpers, 'Popen', MockPopen)

    return all_popens
```

I test di esempio, devono iniziare con il prefisso `test_` nel file `test_docker.py` :

```
def test_docker_install():
    p = Popen(['which', 'docker'], stdout=PIPE, stderr=PIPE)
    result = p.stdout.read()
    assert 'bin/docker' in result

def test_copy_file_to_docker(all_popens):
    result = copy_file_to_docker('asdf', 'asdf')
    collected_popen = all_popens.pop()
    mock_read, mock_write, mock_close = collected_popen.byte_collection.get_all_mock_bytes()
    assert mock_read
    assert result.args == ['docker', 'cp', 'asdf', 'something_cont:asdf']


def test_docker_exec_something(all_popens):

    docker_exec_something(something_file_string)

    collected_popen = all_popens.pop()
    mock_read, mock_write, mock_close = collected_popen.byte_collection.get_all_mock_bytes()
    assert len(mock_read) == 3
    something_template_stdin = mock_write[0][1][0]
    these = [os.environ['USER'], os.environ['password_prod'], 'table_name_here', 'test_vdm',
'col_a', 'col_b', '/tmp/test.tsv']
    assert all([x in something_template_stdin for x in these])
```

eseguendo i test uno alla volta:

```
py.test -k test_docker_install tests
py.test -k test_copy_file_to_docker tests
py.test -k test_docker_exec_something tests
```

eseguendo tutti i test nella cartella `tests` :

```
py.test -k test_ tests
```

Leggi Test unitario online: https://riptutorial.com/it/python/topic/631/test-unitario

Capitolo 190: The Interpreter (Command Line Console)

Examples

Ottenere aiuto generale

Se la funzione di `help` viene chiamata nella console senza alcun argomento, Python presenta una console di guida interattiva, in cui è possibile trovare informazioni sui moduli Python, simboli, parole chiave e altro.

```
>>> help()

Welcome to Python 3.4's help utility!

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http://docs.python.org/3.4/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type
"modules", "keywords", "symbols", or "topics".  Each module also comes
with a one-line summary of what it does; to list the modules whose name
or summary contain a given string such as "spam", type "modules spam".
```

Riferendosi all'ultima espressione

Per ottenere il valore dell'ultimo risultato dall'ultima espressione nella console, utilizzare un carattere di sottolineatura _ .

```
>>> 2 + 2
4
>>> _
4
>>> _ + 6
10
```

Questo valore di sottolineatura magica viene aggiornato solo quando si utilizza un'espressione python che restituisce un valore. La definizione di funzioni o cicli non modifica il valore. Se l'espressione genera un'eccezione, non ci saranno modifiche a _ .

```
>>> "Hello, {0}".format("World")
'Hello, World'
>>> _
'Hello, World'
>>> def wontchangethings():
...     pass
```

```
>>> _
'Hello, World'
>>> 27 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> _
'Hello, World'
```

Ricorda, questa variabile magica è disponibile solo nell'interprete python interattivo. L'esecuzione di script non lo farà.

Apertura della console Python

La console per la versione primaria di Python può essere aperta in genere digitando `py` nella tua console di windows o `python` su altre piattaforme.

```
$ py
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Se hai più versioni, per impostazione predefinita i loro eseguibili verranno mappati rispettivamente su `python2` o `python3` .

Questo ovviamente dipende dagli eseguibili Python presenti nel PATH.

La variabile PYTHONSTARTUP

Puoi impostare una variabile d'ambiente chiamata PYTHONSTARTUP per la console di Python. Ogni volta che si accede alla console Python, questo file verrà eseguito, consentendo di aggiungere funzionalità aggiuntive alla console, ad esempio l'importazione automatica dei moduli utilizzati più comunemente.

Se la variabile PYTHONSTARTUP è stata impostata sul percorso di un file contenente questo:

```
print("Welcome!")
```

Quindi l'apertura della console Python produrrebbe questo output extra:

```
$ py
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
Welcome!
>>>
```

Argomenti della riga di comando

Python ha una varietà di opzioni da riga di comando che possono essere passate a `py` . Questi possono essere trovati eseguendo `py --help` , che fornisce questo output su Python 3.4:

```
Python Launcher

usage: py [ launcher-arguments ] [ python-arguments ] script [ script-arguments ]

Launcher arguments:

-2     : Launch the latest Python 2.x version
-3     : Launch the latest Python 3.x version
-X.Y   : Launch the specified Python version
-X.Y-32: Launch the specified 32bit Python version

The following help text is from Python:

usage: G:\Python34\python.exe [option] ... [-c cmd | -m mod | file | -] [arg] ...
Options and arguments (and corresponding environment variables):
-b     : issue warnings about str(bytes_instance), str(bytearray_instance)
         and comparing bytes/bytearray with str. (-bb: issue errors)
-B     : don't write .py[co] files on import; also PYTHONDONTWRITEBYTECODE=x
-c cmd : program passed in as string (terminates option list)
-d     : debug output from parser; also PYTHONDEBUG=x
-E     : ignore PYTHON* environment variables (such as PYTHONPATH)
-h     : print this help message and exit (also --help)
-i     : inspect interactively after running script; forces a prompt even
         if stdin does not appear to be a terminal; also PYTHONINSPECT=x
-I     : isolate Python from the user's environment (implies -E and -s)
-m mod : run library module as a script (terminates option list)
-O     : optimize generated bytecode slightly; also PYTHONOPTIMIZE=x
-OO    : remove doc-strings in addition to the -O optimizations
-q     : don't print version and copyright messages on interactive startup
-s     : don't add user site directory to sys.path; also PYTHONNOUSERSITE
-S     : don't imply 'import site' on initialization
-u     : unbuffered binary stdout and stderr, stdin always buffered;
         also PYTHONUNBUFFERED=x
         see man page for details on internal buffering relating to '-u'
-v     : verbose (trace import statements); also PYTHONVERBOSE=x
         can be supplied multiple times to increase verbosity
-V     : print the Python version number and exit (also --version)
-W arg : warning control; arg is action:message:category:module:lineno
         also PYTHONWARNINGS=arg
-x     : skip first line of source, allowing use of non-Unix forms of #!cmd
-X opt : set implementation-specific option
file   : program read from script file
-      : program read from stdin (default; interactive mode if a tty)
arg ...: arguments passed to program in sys.argv[1:]

Other environment variables:
PYTHONSTARTUP: file executed on interactive startup (no default)
PYTHONPATH   : ';'-separated list of directories prefixed to the
               default module search path.  The result is sys.path.
PYTHONHOME   : alternate <prefix> directory (or <prefix>;<exec_prefix>).
               The default module search path uses <prefix>\lib.
PYTHONCASEOK : ignore case in 'import' statements (Windows).
PYTHONIOENCODING: Encoding[:errors] used for stdin/stdout/stderr.
PYTHONFAULTHANDLER: dump the Python traceback on fatal errors.
PYTHONHASHSEED: if this variable is set to 'random', a random value is used
   to seed the hashes of str, bytes and datetime objects.  It can also be
   set to an integer in the range [0,4294967295] to get hash values with a
   predictable seed.
```

Ottenere aiuto su un oggetto

La console Python aggiunge una nuova funzione, `help` , che può essere utilizzata per ottenere informazioni su una funzione o un oggetto.

Per una funzione, `help` stampare la sua firma (argomenti) e la sua docstring, se la funzione ne ha uno.

```
>>> help(print)
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

Per un oggetto, `help` elenca la docstring dell'oggetto e le diverse funzioni membro che l'oggetto ha.

```
>>> x = 2
>>> help(x)
Help on int object:

class int(object)
 |  int(x=0) -> integer
 |  int(x, base=10) -> integer
 |
 |  Convert a number or string to an integer, or return 0 if no arguments
 |  are given.  If x is a number, return x.__int__().  For floating point
 |  numbers, this truncates towards zero.
 |
 |  If x is not a number or if base is given, then x must be a string,
 |  bytes, or bytearray instance representing an integer literal in the
 |  given base.  The literal can be preceded by '+' or '-' and be surrounded
 |  by whitespace.  The base defaults to 10.  Valid bases are 0 and 2-36.
 |  Base 0 means to interpret the base from the string as an integer literal.
 |  >>> int('0b100', base=0)
 |  4
 |
 |  Methods defined here:
 |
 |  __abs__(self, /)
 |      abs(self)
 |
 |  __add__(self, value, /)
 |      Return self+value...
```

Leggi The Interpreter (Command Line Console) online:
https://riptutorial.com/it/python/topic/2473/the-interpreter--command-line-console-

Capitolo 191: Tipi di dati immutabili (int, float, str, tuple e frozensets)

Examples

I singoli caratteri delle stringhe non sono assegnabili

```
foo = "bar"
foo[0] = "c" # Error
```

Il valore variabile immutabile non può essere modificato una volta creati.

I singoli membri di Tuple non sono assegnabili

```
foo = ("bar", 1, "Hello!",)
foo[1] = 2 # ERROR!!
```

La seconda riga restituirebbe un errore poiché i membri della tupla una volta creati non sono assegnabili. A causa dell'immutabilità di tuple.

I Frozenset sono immutabili e non assegnabili

```
foo = frozenset(["bar", 1, "Hello!"])
foo[2] = 7 # ERROR
foo.add(3) # ERROR
```

La seconda riga restituirebbe un errore poiché i membri di frozenset una volta creati non sono assegnabili. La terza riga restituirebbe un errore poiché i frozenset non supportano le funzioni che possono manipolare i membri.

Leggi Tipi di dati immutabili (int, float, str, tuple e frozensets) online:
https://riptutorial.com/it/python/topic/4806/tipi-di-dati-immutabili--int--float--str--tuple-e-frozensets-

Capitolo 192: Tipi di dati Python

introduzione

I tipi di dati non sono altro che variabili utilizzati per riservare spazio nella memoria. Le variabili Python non hanno bisogno di una dichiarazione esplicita per riservare spazio di memoria. La dichiarazione avviene automaticamente quando si assegna un valore a una variabile.

Examples

Tipo di dati numerici

I numeri hanno quattro tipi in Python. Int, float, complesso e lungo.

```
int_num = 10    #int value
float_num = 10.2    #float value
complex_num = 3.14j    #complex value
long_num = 1234567L    #long value
```

Tipo di dati stringa

Le stringhe sono identificate come un insieme contiguo di caratteri rappresentati tra virgolette. Python consente sia coppie di virgolette singole o doppie. Le stringhe sono tipi di dati di sequenza immutabili, ovvero ogni volta che si apportano modifiche a una stringa, viene creato un oggetto stringa completamente nuovo.

```
a_str = 'Hello World'
print(a_str)    #output will be whole string. Hello World
print(a_str[0])    #output will be first character. H
print(a_str[0:5])    #output will be first five characters. Hello
```

Elenca il tipo di dati

Un elenco contiene elementi separati da virgole e racchiusi tra parentesi quadre []. Gli elenchi sono quasi simili agli array di C. Una differenza è che tutti gli elementi appartenenti a un elenco possono essere di diverso tipo di dati.

```
list = [123,'abcd',10.2,'d']    #can be a array of any data type or single data type.
list1 = ['hello','world']
print(list)    #will ouput whole list. [123,'abcd',10.2,'d']
print(list[0:2])    #will output first two element of list. [123,'abcd']
print(list1 * 2)    #will gave list1 two times. ['hello','world','hello','world']
print(list + list1)    #will gave concatenation of both the lists.
[123,'abcd',10.2,'d','hello','world']
```

Tuple Data Type

Gli elenchi sono racchiusi tra parentesi [] e i loro elementi e dimensioni possono essere modificati, mentre le tuple sono racchiuse tra parentesi () e non possono essere aggiornate. Le tuple sono immutabili.

```
tuple = (123,'hello')
tuple1 = ('world')
print(tuple)    #will output whole tuple. (123,'hello')
print(tuple[0])    #will output first value. (123)
print(tuple + tuple1)    #will output (123,'hello','world')
tuple[1]='update'    #this will give you error.
```

Dizionario Tipo di dati

Il dizionario consiste di coppie chiave-valore. È racchiuso tra parentesi graffe {} e i valori possono essere assegnati e consultati usando parentesi quadre [].

```
dic={'name':'red','age':10}
print(dic)    #will output all the key-value pairs. {'name':'red','age':10}
print(dic['name'])    #will output only value with 'name' key. 'red'
print(dic.values())    #will output list of values in dic. ['red',10]
print(dic.keys())    #will output list of keys. ['name','age']
```

Imposta i tipi di dati

Gli insiemi sono raccolte non ordinate di oggetti unici, ci sono due tipi di set:

1. Insiemi - Sono mutabili e nuovi elementi possono essere aggiunti una volta definiti i set

   ```
   basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
   print(basket)            # duplicates will be removed
   > {'orange', 'banana', 'pear', 'apple'}
   a = set('abracadabra')
   print(a)                 # unique letters in a
   > {'a', 'r', 'b', 'c', 'd'}
   a.add('z')
   print(a)
   > {'a', 'c', 'r', 'b', 'z', 'd'}
   ```

2. Insiemi congelati - Sono immutabili e non è possibile aggiungere nuovi elementi dopo la sua definizione.

   ```
   b = frozenset('asdfagsa')
   print(b)
   > frozenset({'f', 'g', 'd', 'a', 's'})
   cities = frozenset(["Frankfurt", "Basel","Freiburg"])
   print(cities)
   > frozenset({'Frankfurt', 'Basel', 'Freiburg'})
   ```

Leggi Tipi di dati Python online: https://riptutorial.com/it/python/topic/9366/tipi-di-dati-python

Capitolo 193: Tipo Suggerimenti

Sintassi

- typing.Callable [[int, str], None] -> def func (a: int, b: str) -> Nessuno
- typing.Mapping [str, int] -> {"a": 1, "b": 2, "c": 3}
- typing.List [int] -> [1, 2, 3]
- typing.Set [int] -> {1, 2, 3}
- typing.Optional [int] -> None o int
- typing.Sequence [int] -> [1, 2, 3] o (1, 2, 3)
- digitando. Qualsiasi -> Qualsiasi tipo
- typing.Union [int, str] -> 1 o "1"
- T = typing.TypeVar ('T') -> Tipo generico

Osservazioni

Type Hinting, come specificato in `PEP 484` , è una soluzione formalizzata per indicare staticamente il tipo di un valore per il codice Python. Visualizzando il modulo di `typing` , type-hints offre agli utenti Python la possibilità di annotare il loro codice, aiutando i controllori di tipi mentre, indirettamente, documentando il loro codice con ulteriori informazioni.

Examples

Tipi generici

`typing.TypeVar` è una fabbrica di tipo generico. L'obiettivo principale è quello di fungere da parametro / segnaposto per annotazioni di funzioni / classi / metodi generiche:

```
import typing

T = typing.TypeVar("T")

def get_first_element(l: typing.Sequence[T]) -> T:
    """Gets the first element of a sequence."""
    return l[0]
```

Aggiunta di tipi a una funzione

Prendiamo un esempio di una funzione che riceve due argomenti e restituisce un valore che indica la loro somma:

```
def two_sum(a, b):
    return a + b
```

Osservando questo codice, non si può sicuramente e senza dubbio indicare il tipo di argomenti

per la funzione `two_sum` . Funziona sia quando è fornito con valori `int` :

```
print(two_sum(2, 1))  # result: 3
```

e con le stringhe:

```
print(two_sum("a", "b"))  # result: "ab"
```

e con altri valori, come `list` s, `tuple` s et cetera.

A causa di questa natura dinamica dei tipi Python, in cui molti sono applicabili per una determinata operazione, qualsiasi controllo di tipo non sarebbe in grado di affermare ragionevolmente se una chiamata per questa funzione debba essere consentita o meno.

Per aiutare il nostro correttore di tipi, ora possiamo fornire suggerimenti sui tipi nella definizione della funzione che indica il tipo che consentiamo.

Per indicare che vogliamo solo consentire i tipi `int` possiamo modificare la nostra definizione di funzione in modo che assomigli:

```
def two_sum(a: int, b: int):
    return a + b
```

Le annotazioni seguono il nome dell'argomento e sono separate da un `:` carattere.

Allo stesso modo, per indicare che sono consentiti solo i tipi `str` , cambieremo la nostra funzione per specificarlo:

```
def two_sum(a: str, b: str):
    return a + b
```

Oltre a specificare il tipo di argomenti, si potrebbe anche indicare il valore di ritorno di una chiamata di funzione. Questo viene fatto aggiungendo il carattere `->` seguito dal tipo dopo la parentesi chiusa nella lista degli argomenti *ma* prima `:` alla fine della dichiarazione della funzione:

```
def two_sum(a: int, b: int) -> int:
    return a + b
```

Ora abbiamo indicato che il valore restituito quando si chiama `two_sum` dovrebbe essere di tipo `int` . Allo stesso modo possiamo definire valori appropriati per `str` , `float` , `list` , `set` e altri.

Sebbene i suggerimenti tipo siano principalmente utilizzati dai controllori dei tipi e dagli IDE, a volte potrebbe essere necessario recuperarli. Questo può essere fatto usando l' `__annotations__` speciale `__annotations__` :

```
two_sum.__annotations__
# {'a': <class 'int'>, 'b': <class 'int'>, 'return': <class 'int'>}
```

Membri e metodi della classe

```
class A:
    x = None  # type: float
    def __init__(self, x: float) -> None:
        """
        self should not be annotated
        init should be annotated to return None
        """
        self.x = x

    @classmethod
    def from_int(cls, x: int) -> 'A':
        """
        cls should not be annotated
        Use forward reference to refer to current class with string literal 'A'
        """
        return cls(float(x))
```

Il riferimento diretto della classe corrente è necessario poiché le annotazioni vengono valutate quando la funzione è definita. I riferimenti diretti possono anche essere utilizzati quando si fa riferimento a una classe che causerebbe un'importazione circolare se importata.

Variabili e attributi

Le variabili sono annotate usando i commenti:

```
x = 3  # type: int
x = negate(x)
x = 'a type-checker might catch this error'
```

Python 3.x 3.6

A partire da Python 3.6, c'è anche una nuova sintassi per le annotazioni variabili . Il codice sopra potrebbe usare il modulo

```
x: int = 3
```

A differenza dei commenti, è anche possibile aggiungere un suggerimento tipo a una variabile che non è stata dichiarata in precedenza, senza impostarne un valore:

```
y: int
```

Inoltre se questi sono usati nel modulo o nel livello di classe, i tipi hint possono essere recuperati usando `typing.get_type_hints(class_or_module)` :

```
class Foo:
    x: int
    y: str = 'abc'

print(typing.get_type_hints(Foo))
# ChainMap({'x': <class 'int'>, 'y': <class 'str'>}, {})
```

In alternativa, è possibile accedervi utilizzando la variabile o l'attributo speciale `__annotations__` :

```
x: int
print(__annotations__)
# {'x': <class 'int'>}

class C:
    s: str
print(C.__annotations__)
# {'s': <class 'str'>}
```

namedtuple

La creazione di un namedtuple con hint del tipo viene eseguita utilizzando la funzione NamedTuple dal modulo di `typing` :

```
import typing
Point = typing.NamedTuple('Point', [('x', int), ('y', int)])
```

Si noti che il nome del tipo risultante è il primo argomento della funzione, ma dovrebbe essere assegnato a una variabile con lo stesso nome per facilitare il lavoro dei correttori di tipi.

Digita suggerimenti per gli argomenti delle parole chiave

```
def hello_world(greeting: str = 'Hello'):
    print(greeting + ' world!')
```

Nota gli spazi attorno al segno di uguale rispetto al modo in cui gli argomenti delle parole chiave sono solitamente in stile.

Leggi Tipo Suggerimenti online: https://riptutorial.com/it/python/topic/1766/tipo-suggerimenti

Capitolo 194: Tkinter

introduzione

Rilasciato in Tkinter è la libreria GUI (Graphical User Interface) più popolare di Python. Questo argomento spiega l'uso corretto di questa libreria e delle sue funzionalità.

Osservazioni

La capitalizzazione del modulo tkinter è diversa tra Python 2 e 3. Per Python 2 utilizzare quanto segue:

```
from Tkinter import *  # Capitalized
```

Per Python 3 usa il seguente:

```
from tkinter import *  # Lowercase
```

Per il codice che funziona con Python 2 e 3, puoi farlo

```
try:
    from Tkinter import *
except ImportError:
    from tkinter import *
```

o

```
from sys import version_info
if version_info.major == 2:
    from Tkinter import *
elif version_info.major == 3:
    from tkinter import *
```

> Vedi la documentazione di tkinter per maggiori dettagli

Examples

Un'applicazione tkinter minimale

`tkinter` è un toolkit GUI che fornisce un wrapper attorno alla libreria GUI Tk / Tcl ed è incluso in Python. Il codice seguente crea una nuova finestra usando `tkinter` e inserisce del testo nel corpo della finestra.

> Nota: in Python 2, le lettere maiuscole potrebbero essere leggermente diverse, vedere la sezione Note sotto.

```
import tkinter as tk

# GUI window is a subclass of the basic tkinter Frame object
class HelloWorldFrame(tk.Frame):
    def __init__(self, master):
        # Call superclass constructor
        tk.Frame.__init__(self, master)
        # Place frame into main window
        self.grid()
        # Create text box with "Hello World" text
        hello = tk.Label(self, text="Hello World! This label can hold strings!")
        # Place text box into frame
        hello.grid(row=0, column=0)

# Spawn window
if __name__ == "__main__":
    # Create main window object
    root = tk.Tk()
    # Set title of window
    root.title("Hello World!")
    # Instantiate HelloWorldFrame object
    hello_frame = HelloWorldFrame(root)
    # Start GUI
    hello_frame.mainloop()
```

Geometry Manager

Tkinter ha tre meccanismi per la gestione della geometria: `place` , `pack` e `grid` .

Il `place` gestore utilizza coordinate assolute pixel.

Il gestore del `pack` inserisce i widget in uno dei 4 lati. I nuovi widget sono posizionati accanto ai widget esistenti.

Il gestore della `grid` posiziona i widget in una griglia simile a un foglio di calcolo con ridimensionamento dinamico.

Posto

Gli argomenti delle parole chiave più comuni per `widget.place` sono i seguenti:

- `x` , la coordinata x assoluta del widget
- `y` , la coordinata y assoluta del widget
- `height` , l'altezza assoluta del widget
- `width` , la larghezza assoluta del widget

Un esempio di codice che utilizza il `place` :

```
class PlaceExample(Frame):
    def __init__(self,master):
        Frame.__init__(self,master)
        self.grid()
        top_text=Label(master,text="This is on top at the origin")
```

```
        #top_text.pack()
        top_text.place(x=0,y=0,height=50,width=200)
        bottom_right_text=Label(master,text="This is at position 200,400")
        #top_text.pack()
        bottom_right_text.place(x=200,y=400,height=50,width=200)
# Spawn Window
if __name__=="__main__":
    root=Tk()
    place_frame=PlaceExample(root)
    place_frame.mainloop()
```

pacco

`widget.pack` può prendere i seguenti argomenti di parole chiave:

- `expand` , se riempire o meno lo spazio lasciato dal genitore
- `fill` , se espandere per riempire tutto lo spazio (NESSUNO (predefinito), X, Y o ENTRAMBI)
- `side` , il lato da imballare contro (TOP (predefinito), BOTTOM, SINISTRA o DESTRA)

Griglia

Gli argomenti delle parole chiave più utilizzati di `widget.grid` sono i seguenti:

- `row` , la riga del widget (la più piccola di default non occupata)
- `rowspan` , il numero di colonne che un widget si estende (predefinito 1)
- `column` , la colonna del widget (default 0)
- `columnspan` , il numero di colonne di un widget (predefinito 1)
- `sticky` , dove posizionare il widget se la cella della griglia è più grande di essa (combinazione di N, NE, E, SE, S, SW, W, NW)

Le righe e le colonne sono indicizzate a zero. Le righe aumentano andando giù e le colonne aumentano andando a destra.

Un esempio di codice usando la `grid` :

```
from tkinter import *

class GridExample(Frame):
    def __init__(self,master):
        Frame.__init__(self,master)
        self.grid()
        top_text=Label(self,text="This text appears on top left")
        top_text.grid() # Default position 0, 0
        bottom_text=Label(self,text="This text appears on bottom left")
        bottom_text.grid() # Default position 1, 0
        right_text=Label(self,text="This text appears on the right and spans both rows",
                         wraplength=100)
        # Position is 0,1
        # Rowspan means actual position is [0-1],1
        right_text.grid(row=0,column=1,rowspan=2)
```

```
# Spawn Window
if __name__=="__main__":
    root=Tk()
    grid_frame=GridExample(root)
    grid_frame.mainloop()
```

Non mischiare mai `pack` e `grid` nello stesso frame! Fare così porterà ad un deadlock dell'applicazione!

Leggi Tkinter online: https://riptutorial.com/it/python/topic/7574/tkinter

Capitolo 195: Tracciare con Matplotlib

introduzione

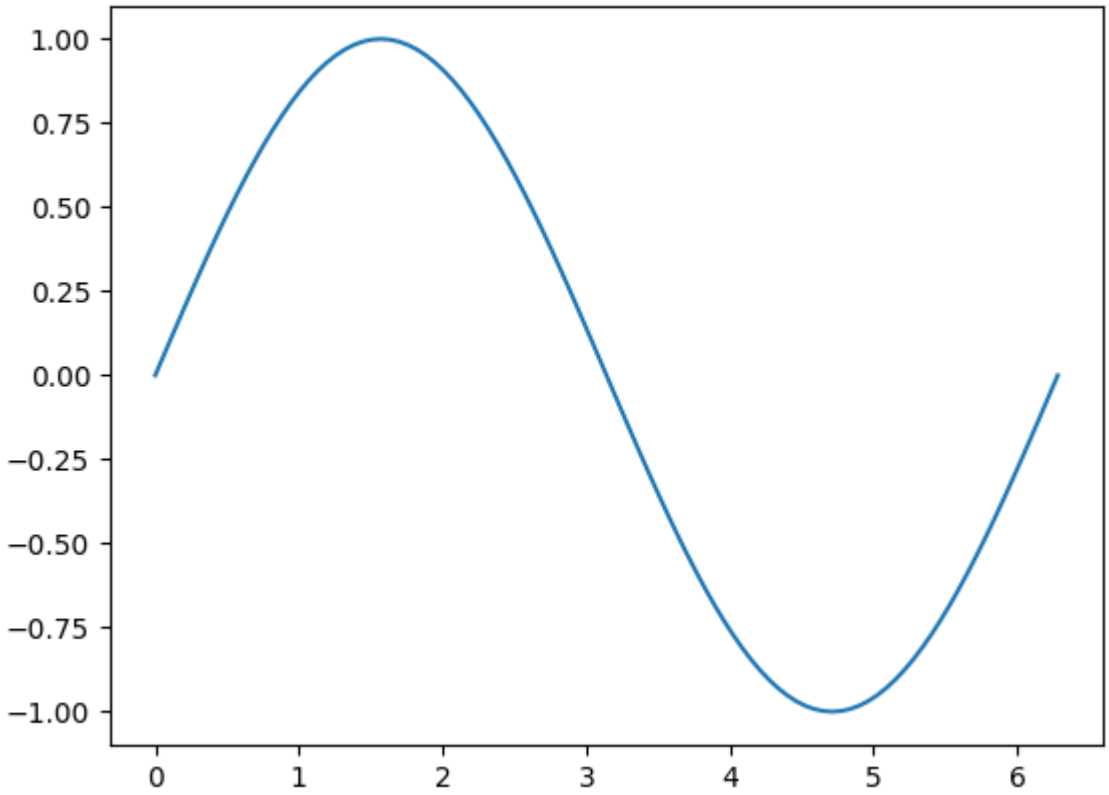
Matplotlib (https://matplotlib.org/) è una libreria per il disegno 2D basata su NumPy. Ecco alcuni esempi di base. Altri esempi possono essere trovati nella documentazione ufficiale (https://matplotlib.org/2.0.2/gallery.html e https://matplotlib.org/2.0.2/examples/index.html) così come in http: //www.riptutorial.com/topic/881

Examples

Una trama semplice in Matplotlib

Questo esempio illustra come creare una curva sinusoidale semplice usando **Matplotlib**

```
# Plotting tutorials in Python
# Launching a simple plot

import numpy as np
import matplotlib.pyplot as plt

# angle varying between 0 and 2*pi
x = np.linspace(0, 2.0*np.pi, 101)
y = np.sin(x)                          # sine function

plt.plot(x, y)
plt.show()
```

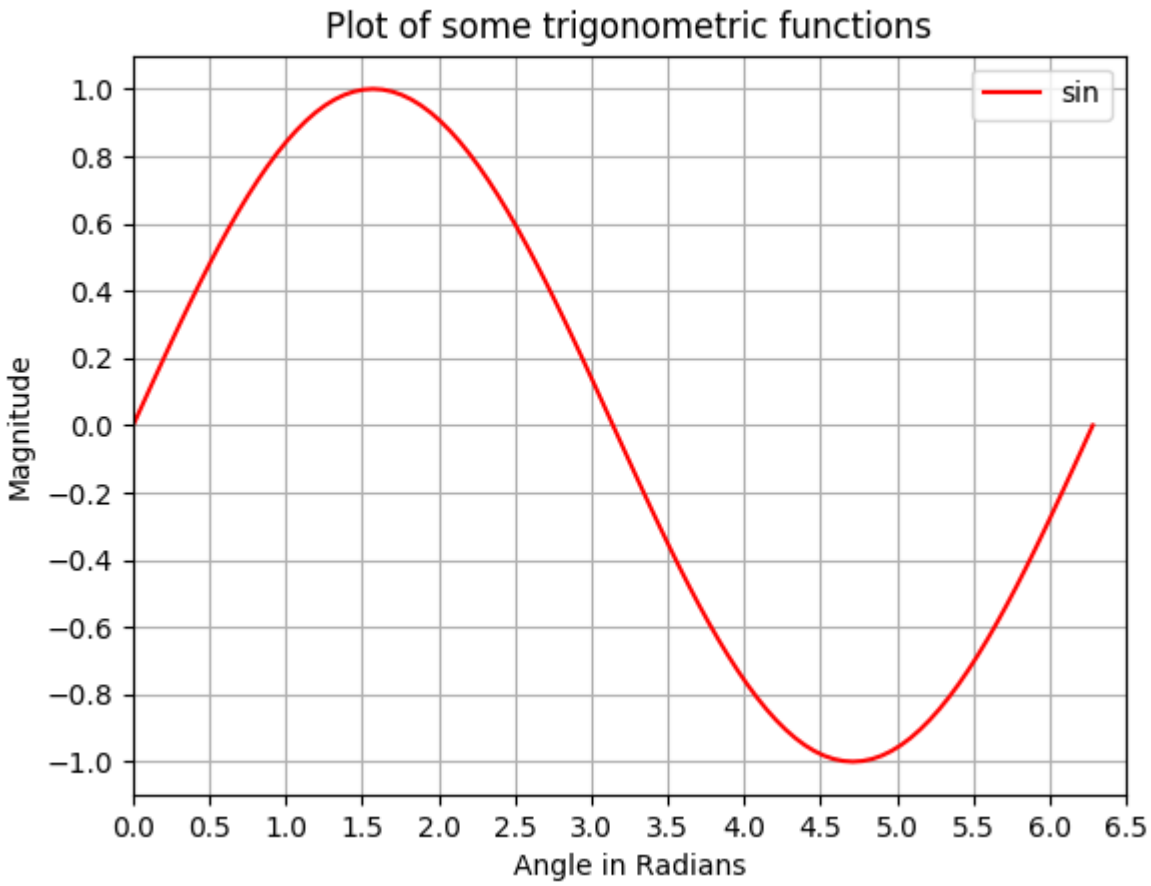


Aggiunta di più funzioni a un grafico semplice: etichette dell'asse, titolo, segni di graduazione, griglia e legenda

In questo esempio, prendiamo una trama sinusoidale e aggiungiamo più funzioni ad essa; ovvero il titolo, le etichette degli assi, il titolo, le tacche degli assi, la griglia e la legenda.

```
# Plotting tutorials in Python
# Enhancing a plot

import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2.0*np.pi, 101)
y = np.sin(x)

# values for making ticks in x and y axis
xnumbers = np.linspace(0, 7, 15)
ynumbers = np.linspace(-1, 1, 11)

plt.plot(x, y, color='r', label='sin') # r - red colour
plt.xlabel("Angle in Radians")
plt.ylabel("Magnitude")
plt.title("Plot of some trigonometric functions")
plt.xticks(xnumbers)
plt.yticks(ynumbers)
plt.legend()
plt.grid()
plt.axis([0, 6.5, -1.1, 1.1]) # [xstart, xend, ystart, yend]
```

```
plt.show()
```

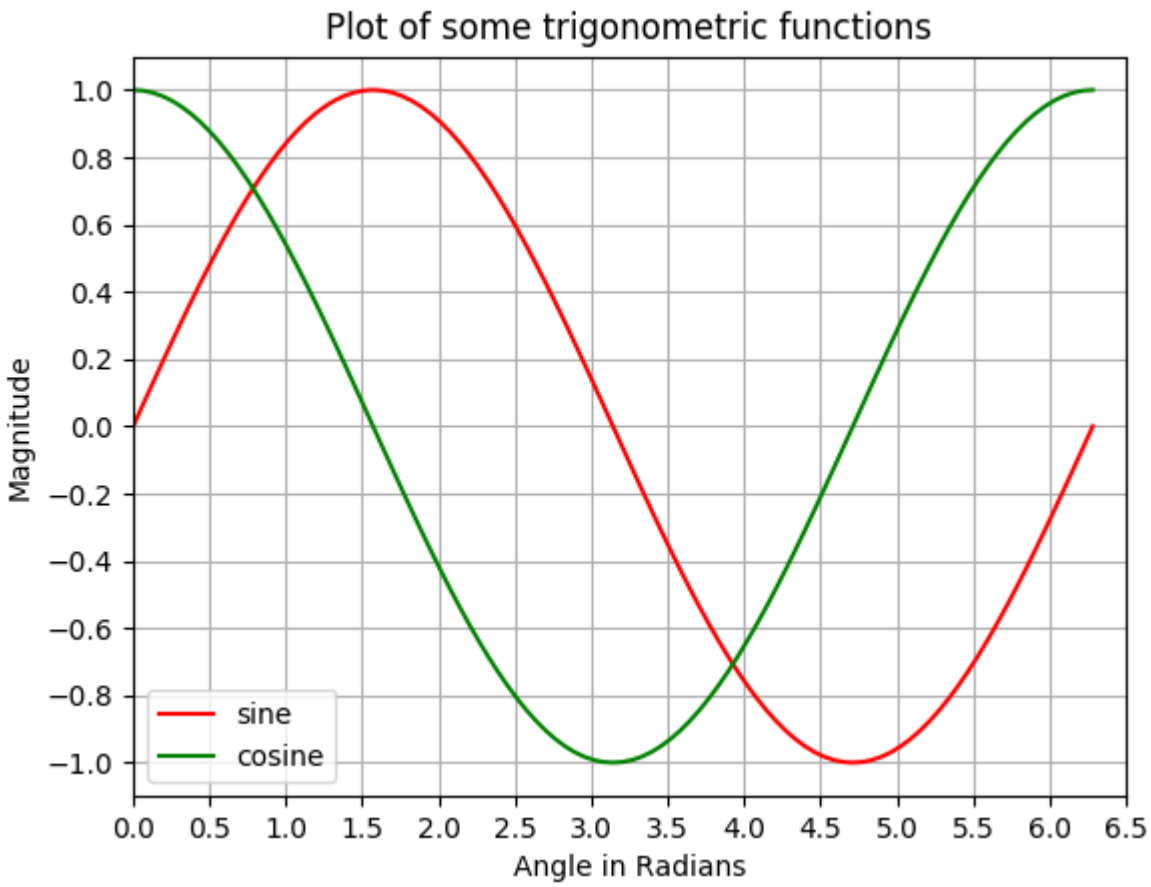


Realizzare trame multiple nella stessa figura per sovrapposizione simile a MATLAB

In questo esempio, una curva sinusoidale e una curva coseno sono tracciate nella stessa figura sovrapponendo i grafici l'uno sopra l'altro.

```
# Plotting tutorials in Python
# Adding Multiple plots by superimposition
# Good for plots sharing similar x, y limits
# Using single plot command and legend

import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2.0*np.pi, 101)
y = np.sin(x)
z = np.cos(x)

# values for making ticks in x and y axis
xnumbers = np.linspace(0, 7, 15)
ynumbers = np.linspace(-1, 1, 11)

plt.plot(x, y, 'r', x, z, 'g') # r, g - red, green colour
plt.xlabel("Angle in Radians")
plt.ylabel("Magnitude")
```

```
plt.title("Plot of some trigonometric functions")
plt.xticks(xnumbers)
plt.yticks(ynumbers)
plt.legend(['sine', 'cosine'])
plt.grid()
plt.axis([0, 6.5, -1.1, 1.1]) # [xstart, xend, ystart, yend]
plt.show()
```

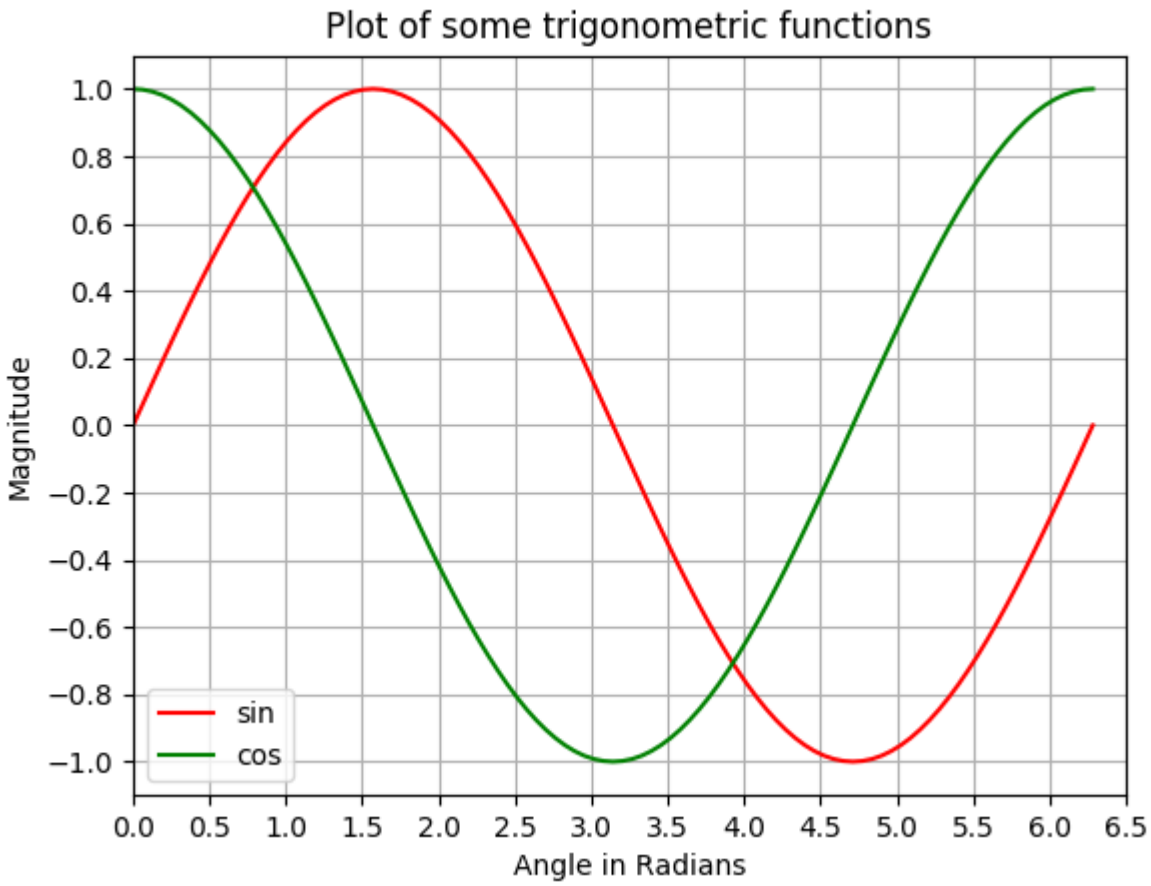


Creare più grafici nella stessa figura usando la sovrapposizione della trama con comandi di stampa separati

Simile all'esempio precedente, qui, una curva seno e una curva coseno sono tracciate sulla stessa figura utilizzando comandi di stampa separati. Questo è più Pythonic e può essere usato per ottenere maniglie separate per ogni trama.

```
# Plotting tutorials in Python
# Adding Multiple plots by superimposition
# Good for plots sharing similar x, y limits
# Using multiple plot commands
# Much better and preferred than previous

import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2.0*np.pi, 101)
y = np.sin(x)
z = np.cos(x)
```

```
# values for making ticks in x and y axis
xnumbers = np.linspace(0, 7, 15)
ynumbers = np.linspace(-1, 1, 11)

plt.plot(x, y, color='r', label='sin') # r - red colour
plt.plot(x, z, color='g', label='cos') # g - green colour
plt.xlabel("Angle in Radians")
plt.ylabel("Magnitude")
plt.title("Plot of some trigonometric functions")
plt.xticks(xnumbers)
plt.yticks(ynumbers)
plt.legend()
plt.grid()
plt.axis([0, 6.5, -1.1, 1.1]) # [xstart, xend, ystart, yend]
plt.show()
```

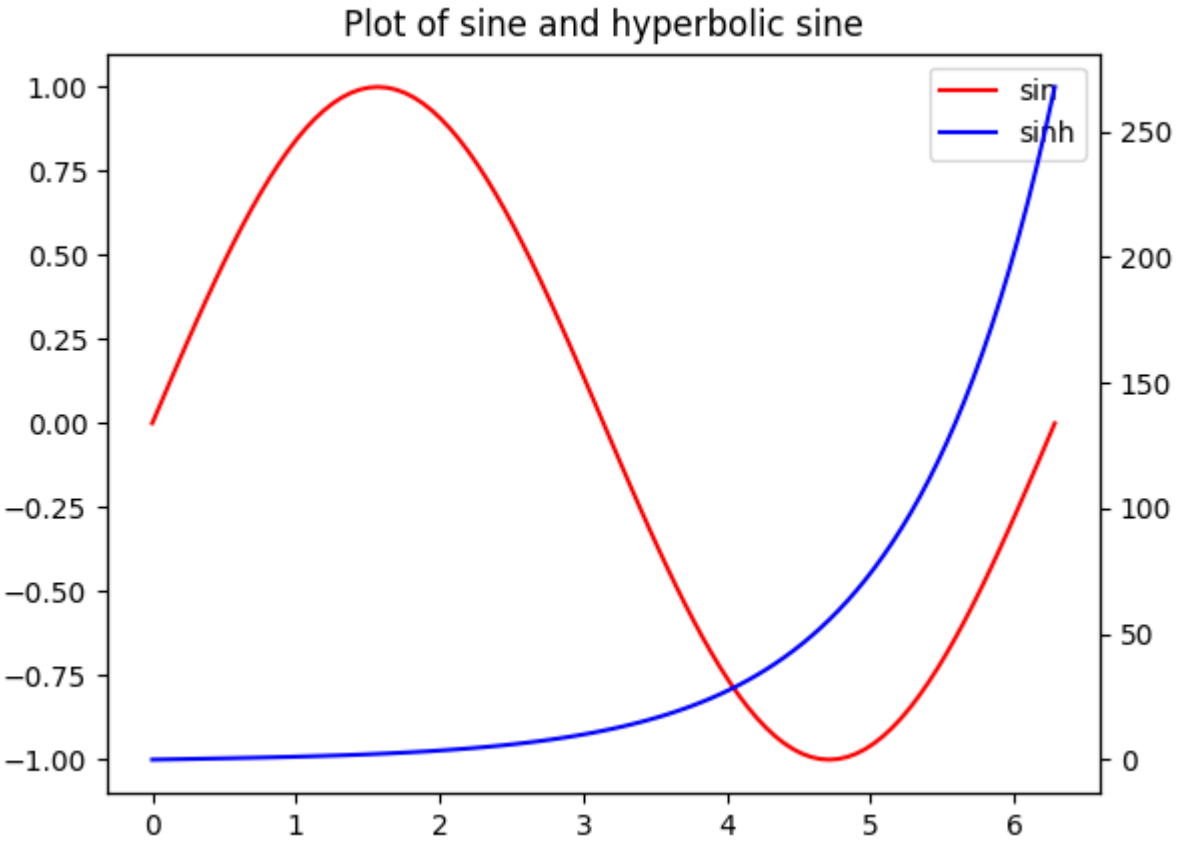


Grafici con asse X comune ma asse Y diverso: utilizzo di twinx ()

In questo esempio, tracciamo una curva sinusoidale e una curva sinusoidale iperbolica nello stesso grafico con un asse x comune con asse y diverso. Ciò è ottenuto dall'uso del comando **twinx ()** .

```
# Plotting tutorials in Python
# Adding Multiple plots by twin x axis
# Good for plots having different y axis range
# Separate axes and figure objects
```

```
# replicate axes object and plot curves
# use axes to set attributes

# Note:
# Grid for second curve unsuccessful : let me know if you find it! :(

import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2.0*np.pi, 101)
y = np.sin(x)
z = np.sinh(x)

# separate the figure object and axes object
# from the plotting object
fig, ax1 = plt.subplots()

# Duplicate the axes with a different y axis
# and the same x axis
ax2 = ax1.twinx() # ax2 and ax1 will have common x axis and different y axis

# plot the curves on axes 1, and 2, and get the curve handles
curve1, = ax1.plot(x, y, label="sin", color='r')
curve2, = ax2.plot(x, z, label="sinh", color='b')

# Make a curves list to access the parameters in the curves
curves = [curve1, curve2]

# add legend via axes 1 or axes 2 object.
# one command is usually sufficient
# ax1.legend() # will not display the legend of ax2
# ax2.legend() # will not display the legend of ax1
ax1.legend(curves, [curve.get_label() for curve in curves])
# ax2.legend(curves, [curve.get_label() for curve in curves]) # also valid

# Global figure properties
plt.title("Plot of sine and hyperbolic sine")
plt.show()
```

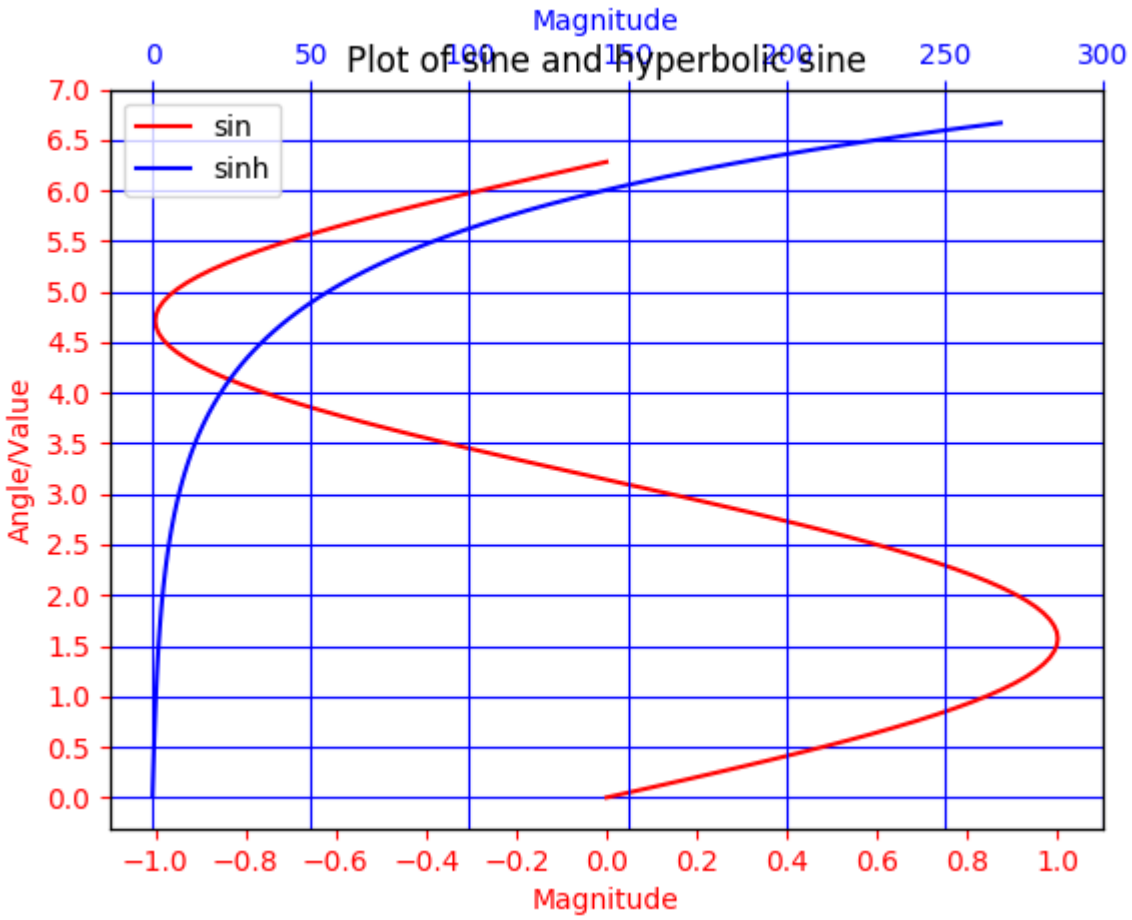


Grafici con asse Y comune e asse X diverso con twiny ()

In questo esempio, un grafico con curve aventi l'asse y comune ma un asse x diverso viene dimostrato usando il metodo **twiny ()** . Inoltre, alcune caratteristiche aggiuntive come titolo, legenda, etichette, griglie, segni di graduazione e colori vengono aggiunti alla trama.

```
# Plotting tutorials in Python
# Adding Multiple plots by twin y axis
# Good for plots having different x axis range
# Separate axes and figure objects
# replicate axes object and plot curves
# use axes to set attributes

import numpy as np
import matplotlib.pyplot as plt

y = np.linspace(0, 2.0*np.pi, 101)
x1 = np.sin(y)
x2 = np.sinh(y)

# values for making ticks in x and y axis
ynumbers = np.linspace(0, 7, 15)
xnumbers1 = np.linspace(-1, 1, 11)
xnumbers2 = np.linspace(0, 300, 7)

# separate the figure object and axes object
# from the plotting object
fig, ax1 = plt.subplots()
```

```
# Duplicate the axes with a different x axis
# and the same y axis
ax2 = ax1.twiny() # ax2 and ax1 will have common y axis and different x axis

# plot the curves on axes 1, and 2, and get the axes handles
curve1, = ax1.plot(x1, y, label="sin", color='r')
curve2, = ax2.plot(x2, y, label="sinh", color='b')

# Make a curves list to access the parameters in the curves
curves = [curve1, curve2]

# add legend via axes 1 or axes 2 object.
# one command is usually sufficient
# ax1.legend() # will not display the legend of ax2
# ax2.legend() # will not display the legend of ax1
# ax1.legend(curves, [curve.get_label() for curve in curves])
ax2.legend(curves, [curve.get_label() for curve in curves]) # also valid

# x axis labels via the axes
ax1.set_xlabel("Magnitude", color=curve1.get_color())
ax2.set_xlabel("Magnitude", color=curve2.get_color())

# y axis label via the axes
ax1.set_ylabel("Angle/Value", color=curve1.get_color())
# ax2.set_ylabel("Magnitude", color=curve2.get_color()) # does not work
# ax2 has no property control over y axis

# y ticks - make them coloured as well
ax1.tick_params(axis='y', colors=curve1.get_color())
# ax2.tick_params(axis='y', colors=curve2.get_color()) # does not work
# ax2 has no property control over y axis

# x axis ticks via the axes
ax1.tick_params(axis='x', colors=curve1.get_color())
ax2.tick_params(axis='x', colors=curve2.get_color())

# set x ticks
ax1.set_xticks(xnumbers1)
ax2.set_xticks(xnumbers2)

# set y ticks
ax1.set_yticks(ynumbers)
# ax2.set_yticks(ynumbers) # also works

# Grids via axes 1 # use this if axes 1 is used to
# define the properties of common x axis
# ax1.grid(color=curve1.get_color())

# To make grids using axes 2
ax1.grid(color=curve2.get_color())
ax2.grid(color=curve2.get_color())
ax1.xaxis.grid(False)

# Global figure properties
plt.title("Plot of sine and hyperbolic sine")
plt.show()
```



Leggi Tracciare con Matplotlib online: https://riptutorial.com/it/python/topic/10264/tracciare-con-matplotlib

Capitolo 196: Trasformazione di Panda: operazioni preforme su gruppi e concatenare i risultati

Examples

Trasformazione semplice

Innanzitutto, creiamo un dataframe fittizio

Supponiamo che un cliente possa avere n ordini, che un ordine possa avere m articoli e che gli articoli possano essere ordinati più volte

```
orders_df = pd.DataFrame()
orders_df['customer_id'] = [1,1,1,1,1,2,2,3,3,3,3,3]
orders_df['order_id'] = [1,1,1,2,2,3,3,4,5,6,6,6]
orders_df['item'] = ['apples', 'chocolate', 'chocolate', 'coffee', 'coffee', 'apples',
                     'bananas', 'coffee', 'milkshake', 'chocolate', 'strawberry',
'strawberry']

# And this is how the dataframe looks like:
print(orders_df)
#     customer_id  order_id        item
# 0             1         1      apples
# 1             1         1   chocolate
# 2             1         1   chocolate
# 3             1         2      coffee
# 4             1         2      coffee
# 5             2         3      apples
# 6             2         3     bananas
# 7             3         4      coffee
# 8             3         5   milkshake
# 9             3         6   chocolate
# 10            3         6  strawberry
# 11            3         6  strawberry
```

.
.

Ora, utilizzeremo la funzione di `transform` panda per contare il numero di ordini per cliente

```
# First, we define the function that will be applied per customer_id
count_number_of_orders = lambda x: len(x.unique())

# And now, we can tranform each group using the logic defined above
orders_df['number_of_orders_per_cient'] = (               # Put the results into a new column
```

```
that is called 'number_of_orders_per_cient'
                    orders_df                              # Take the original dataframe
                   .groupby(['customer_id'])['order_id'] # Create a seperate group for each
customer_id & select the order_id
                   .transform(count_number_of_orders))    # Apply the function to each group
seperatly

# Inspecting the results ...
print(orders_df)
#     customer_id  order_id        item  number_of_orders_per_cient
# 0             1         1      apples                           2
# 1             1         1   chocolate                           2
# 2             1         1   chocolate                           2
# 3             1         2      coffee                           2
# 4             1         2      coffee                           2
# 5             2         3      apples                           1
# 6             2         3     bananas                           1
# 7             3         4      coffee                           3
# 8             3         5   milkshake                           3
# 9             3         6   chocolate                           3
# 10            3         6  strawberry                           3
# 11            3         6  strawberry                           3
```

Più risultati per gruppo

Utilizzo delle funzioni di `transform` che restituiscono calcoli secondari per gruppo

Nell'esempio precedente, avevamo un risultato per cliente. Tuttavia, è possibile applicare anche funzioni che restituiscono valori diversi per il gruppo.

```
# Create a dummy dataframe
orders_df = pd.DataFrame()
orders_df['customer_id'] = [1,1,1,1,1,2,2,3,3,3,3,3]
orders_df['order_id'] = [1,1,1,2,2,3,3,4,5,6,6,6]
orders_df['item'] = ['apples', 'chocolate', 'chocolate', 'coffee', 'coffee', 'apples',
                     'bananas', 'coffee', 'milkshake', 'chocolate', 'strawberry',
'strawberry']


# Let's try to see if the items were ordered more than once in each orders

# First, we define a fuction that will be applied per group
def multiple_items_per_order(_items):
    # Apply .duplicated, which will return True is the item occurs more than once.
    multiple_item_bool = _items.duplicated(keep=False)
    return(multiple_item_bool)

# Then, we transform each group according to the defined function
orders_df['item_duplicated_per_order'] = (                    # Put the results into a new
column
                        orders_df                             # Take the orders dataframe
                        .groupby(['order_id'])['item']        # Create a seperate group for
each order_id & select the item
                        .transform(multiple_items_per_order)) # Apply the defined function to
```

```
each group separately

# Inspecting the results ...
print(orders_df)
#     customer_id  order_id        item  item_duplicated_per_order
# 0             1         1      apples                      False
# 1             1         1   chocolate                       True
# 2             1         1   chocolate                       True
# 3             1         2      coffee                       True
# 4             1         2      coffee                       True
# 5             2         3      apples                      False
# 6             2         3     bananas                      False
# 7             3         4      coffee                      False
# 8             3         5   milkshake                      False
# 9             3         6   chocolate                      False
# 10            3         6  strawberry                       True
# 11            3         6  strawberry                       True
```

Leggi Trasformazione di Panda: operazioni preforme su gruppi e concatenare i risultati online: https://riptutorial.com/it/python/topic/10947/trasformazione-di-panda--operazioni-preforme-su-gruppi-e-concatenare-i-risultati

Capitolo 197: tuple

introduzione

Una tupla è una lista immutabile di valori. Le tuple sono uno dei tipi di raccolta più semplici e comuni di Python e possono essere creati con l'operatore virgola (`value = 1, 2, 3`).

Sintassi

- (1, a, "ciao") # a deve essere una variabile
- () # una tupla vuota
- (1,) # a tupla a 1 elemento. `(1)` non è una tupla.
- 1, 2, 3 # la tupla a 3 elementi (1, 2, 3)

Osservazioni

Le parentesi sono necessarie solo per le tuple vuote o quando vengono utilizzate in una chiamata di funzione.

Una tupla è una sequenza di valori. I valori possono essere di qualsiasi tipo e sono indicizzati da numeri interi, quindi sotto questo aspetto le tuple sono molto simili alle liste. L'importante differenza è che le tuple sono immutabili e sono lavabili, quindi possono essere utilizzate in insiemi e mappe

Examples

Tuple di indicizzazione

```
x = (1, 2, 3)
x[0]  # 1
x[1]  # 2
x[2]  # 3
x[3]  # IndexError: tuple index out of range
```

L'indicizzazione con numeri negativi inizierà dall'ultimo elemento come -1:

```
x[-1] # 3
x[-2] # 2
x[-3] # 1
x[-4] # IndexError: tuple index out of range
```

Indicizzazione di un intervallo di elementi

```
print(x[:-1])   # (1, 2)
print(x[-1:])   # (3,)
print(x[1:3])   # (2, 3)
```

Le tuple sono immutabili

Una delle principali differenze tra le `list` s e le `tuple` in Python è che le tuple sono immutabili, cioè non è possibile aggiungere o modificare elementi una volta che la tupla è stata inizializzata. Per esempio:

```
>>> t = (1, 4, 9)
>>> t[0] = 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Allo stesso modo, le tuple non hanno metodi `.append` e `.extend` come fa la `list` . Usare `+=` è possibile, ma cambia il binding della variabile, e non la tupla stessa:

```
>>> t = (1, 2)
>>> q = t
>>> t += (3, 4)
>>> t
(1, 2, 3, 4)
>>> q
(1, 2)
```

Fai attenzione quando posizioni oggetti mutabili, come `lists` , all'interno di tuple. Questo può portare a risultati molto confusi quando li si cambia. Per esempio:

```
>>> t = (1, 2, 3, [1, 2, 3])
(1, 2, 3, [1, 2, 3])
>>> t[3] += [4, 5]
```

Entrambi genereranno un errore e cambieranno il contenuto della lista all'interno della tupla:

```
TypeError: 'tuple' object does not support item assignment
>>> t
(1, 2, 3, [1, 2, 3, 4, 5])
```

Puoi usare l'operatore `+=` per "aggiungere" ad una tupla - questo funziona creando una nuova tupla con il nuovo elemento che hai "aggiunto" e assegnandolo alla sua variabile corrente; la vecchia tupla non è cambiata, ma sostituita!

Questo evita la conversione da e verso un elenco, ma questo è lento ed è una cattiva pratica, specialmente se si intende aggiungere più volte.

Tuple Are Element-wise hashable ed Equitable

```
hash( (1, 2) )  # ok
```

```
hash( ([], {"hello"})  # not ok, since lists and sets are not hashabe
```

Quindi una tupla può essere messa dentro un `set` o come una chiave in un `dict` solo se ognuno dei suoi elementi può.

```
{ (1, 2) } #  ok
{ ([], {"hello"}) ) # not ok
```

tuple

Sintatticamente, una tupla è un elenco di valori separati da virgole:

```
t = 'a', 'b', 'c', 'd', 'e'
```

Sebbene non sia necessario, è comune racchiudere le tuple tra parentesi:

```
t = ('a', 'b', 'c', 'd', 'e')
```

Crea una tupla vuota con parentesi:

```
t0 = ()
type(t0)            # <type 'tuple'>
```

Per creare una tupla con un singolo elemento, devi includere una virgola finale:

```
t1 = 'a',
type(t1)              # <type 'tuple'>
```

Si noti che un singolo valore tra parentesi non è una tupla:

```
t2 = ('a')
type(t2)              # <type 'str'>
```

Per creare una tupla singleton è necessario avere una virgola finale.

```
t2 = ('a',)
type(t2)              # <type 'tuple'>
```

Si noti che per le tuple singleton è consigliabile (vedere PEP8 sulle virgole finali) per utilizzare le parentesi. Inoltre, nessuno spazio vuoto dopo la virgola finale (vedi PEP8 sugli spazi bianchi)

```
t2 = ('a',)           # PEP8-compliant
t2 = 'a',             # this notation is not recommended by PEP8
t2 = ('a', )          # this notation is not recommended by PEP8
```

Un altro modo per creare una tupla è la funzione `tuple` integrata.

```
t = tuple('lupins')
```

```
print(t)              # ('l', 'u', 'p', 'i', 'n', 's')
t = tuple(range(3))
print(t)              # (0, 1, 2)
```

Questi esempi sono basati sul materiale del libro **Think Python** di Allen B. Downey .

Imballaggio e disimballaggio Tuples

Le tuple in Python sono valori separati da virgole. Includere parentesi per l'immissione di tuple è facoltativo, quindi i due compiti

```
a = 1, 2, 3   # a is the tuple (1, 2, 3)
```

e

```
a = (1, 2, 3) # a is the tuple (1, 2, 3)
```

sono equivalenti. L'assegnazione `a = 1, 2, 3` viene anche chiamata *packing* perché racchiude i valori insieme in una tupla.

Nota che una tupla a un valore è anche una tupla. Per dire a Python che una variabile è una tupla e non un singolo valore puoi usare una virgola finale

```
a = 1  # a is the value 1
a = 1, # a is the tuple (1,)
```

È necessaria una virgola anche se si utilizzano le parentesi

```
a = (1,) # a is the tuple (1,)
a = (1)  # a is the value 1 and not a tuple
```

Per decomprimere i valori da una tupla e utilizzare più assegnazioni

```
# unpacking AKA multiple assignment
x, y, z = (1, 2, 3)
# x == 1
# y == 2
# z == 3
```

Il simbolo _ può essere usato come un nome di variabile usa e getta se uno ha bisogno solo di alcuni elementi di una tupla, che agisce come un segnaposto:

```
a = 1, 2, 3, 4
_, x, y, _ = a
# x == 2
# y == 3
```

Tuple a singolo elemento:

```
x, = 1,  # x is the value 1
x  = 1,  # x is the tuple (1,)
```

In Python 3 una variabile di destinazione con un prefisso `*` può essere usata come variabile *catch-all* (vedi Unpacking Iterables):

Python 3.x 3.0

```
first, *more, last = (1, 2, 3, 4, 5)
# first == 1
# more == [2, 3, 4]
# last == 5
```

Inversione di elementi

Elementi inversi all'interno di una tupla

```
colors = "red", "green", "blue"
rev = colors[::-1]
# rev: ("blue", "green", "red")
colors = rev
# colors: ("blue", "green", "red")
```

O usando l'inverso (invertito dà un iterabile che viene convertito in una tupla):

```
rev = tuple(reversed(colors))
# rev: ("blue", "green", "red")
colors = rev
# colors: ("blue", "green", "red")
```

Funzioni tuple integrate

Le tuple supportano le seguenti funzioni built-in

Confronto

Se gli elementi sono dello stesso tipo, python esegue il confronto e restituisce il risultato. Se gli elementi sono di tipo diverso, controlla se si tratta di numeri.

- Se i numeri, eseguire il confronto.
- Se uno degli elementi è un numero, viene restituito l'altro elemento.
- Altrimenti, i tipi sono ordinati alfabeticamente.

Se abbiamo raggiunto la fine di uno degli elenchi, la lista più lunga è "più grande". Se entrambe le liste sono uguali, restituisce 0.

```
tuple1 = ('a', 'b', 'c', 'd', 'e')
tuple2 = ('1','2','3')
tuple3 = ('a', 'b', 'c', 'd', 'e')
```

```
cmp(tuple1, tuple2)
Out: 1

cmp(tuple2, tuple1)
Out: -1

cmp(tuple1, tuple3)
Out: 0
```

Lunghezza della tupla

La funzione `len` restituisce la lunghezza totale della tupla

```
len(tuple1)
Out: 5
```

Max di una tupla

La funzione `max` restituisce l'elemento dalla tupla con il valore massimo

```
max(tuple1)
Out: 'e'

max(tuple2)
Out: '3'
```

Minimo di una tupla

La funzione min restituisce l'elemento dalla tupla con il valore minimo

```
min(tuple1)
Out: 'a'

min(tuple2)
Out: '1'
```

Convertire una lista in tupla

La funzione integrata `tuple` converte una lista in una tupla.

```
list = [1,2,3,4,5]
tuple(list)
Out: (1, 2, 3, 4, 5)
```

Concatenazione di tupla

Usa `+` per concatenare due tuple

```
tuple1 + tuple2
Out: ('a', 'b', 'c', 'd', 'e', '1', '2', '3')
```

Leggi tuple online: https://riptutorial.com/it/python/topic/927/tuple

Capitolo 198: Unicode

Examples

Codifica e decodifica

Codifica sempre da unicode a byte. In questa direzione, **puoi scegliere la codifica** .

```
>>> u'🐍'.encode('utf-8')
'\xf0\x9f\x90\x8d'
```

L'altro modo è quello di *decodificare* da byte a unicode. In questa direzione, **devi sapere qual è la codifica** .

```
>>> b'\xf0\x9f\x90\x8d'.decode('utf-8')
u'\U0001f40d'
```

Leggi Unicode online: https://riptutorial.com/it/python/topic/5618/unicode

Capitolo 199: Unicode e byte

Sintassi

- str.encode (encoding, errors = 'strict')
- bytes.decode (encoding, errors = 'strict')
- open (nomefile, modo, codifica = Nessuno)

Parametri

Parametro	Dettagli
codifica	La codifica da usare, ad esempio `'ascii'` , `'utf8'` , ecc ...
errori	La modalità errori, ad esempio `'replace'` per sostituire i caratteri non validi con punti interrogativi, `'ignore'` per ignorare i caratteri errati, ecc ...

Examples

Nozioni di base

In Python 3 `str` è il tipo per le stringhe abilitate per Unicode, mentre i `bytes` è il tipo per le sequenze di byte non elaborati.

```
type("f") == type(u"f")  # True, <class 'str'>
type(b"f")               # <class 'bytes'>
```

In Python 2 una stringa casuale era una sequenza di byte grezzi per impostazione predefinita e la stringa unicode era ogni stringa con prefisso "u".

```
type("f") == type(b"f")  # True, <type 'str'>
type(u"f")               # <type 'unicode'>
```

Unicode in byte

Le stringhe Unicode possono essere convertite in byte con `.encode(encoding)` .

Python 3

```
>>> "£13.55".encode('utf8')
b'\xc2\xa313.55'
>>> "£13.55".encode('utf16')
```

```
b'\xff\xfe\xa3\x001\x003\x00.\x005\x005\x00'
```

Python 2

in py2 la codifica della console predefinita è `sys.getdefaultencoding() == 'ascii'` e non `utf-8` come in py3, quindi la stampa come nell'esempio precedente non è direttamente possibile.

```
>>> print type(u"£13.55".encode('utf8'))
<type 'str'>
>>> print u"£13.55".encode('utf8')
SyntaxError: Non-ASCII character '\xc2' in...

# with encoding set inside a file

# -*- coding: utf-8 -*-
>>> print u"£13.55".encode('utf8')
┬ú13.55
```

Se la codifica non può gestire la stringa, viene sollevato un `UnicodeEncodeError`:

```
>>> "£13.55".encode('ascii')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'ascii' codec can't encode character '\xa3' in position 0: ordinal not in
range(128)
```

Byte per unicode

I byte possono essere convertiti in stringhe unicode con `.decode(encoding)` .

Una sequenza di byte può essere convertita in una stringa unicode solo tramite la codifica appropriata!

```
>>> b'\xc2\xa313.55'.decode('utf8')
'£13.55'
```

Se la codifica non è in grado di gestire la stringa, viene generato un `UnicodeDecodeError` :

```
>>> b'\xc2\xa313.55'.decode('utf16')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/csaftoiu/csaftoiu-github/yahoo-groups-
backup/.virtualenv/bin/../lib/python3.5/encodings/utf_16.py", line 16, in decode
    return codecs.utf_16_decode(input, errors, True)
UnicodeDecodeError: 'utf-16-le' codec can't decode byte 0x35 in position 6: truncated data
```

Gestione degli errori di codifica / decodifica

`.encode` e `.decode` hanno entrambi modalità di errore.

L'impostazione predefinita è `'strict'` , che solleva eccezioni in caso di errore. Altre modalità sono più indulgenti.

Codifica

```
>>> "£13.55".encode('ascii', errors='replace')
b'?13.55'
>>> "£13.55".encode('ascii', errors='ignore')
b'13.55'
>>> "£13.55".encode('ascii', errors='namereplace')
b'\\N{POUND SIGN}13.55'
>>> "£13.55".encode('ascii', errors='xmlcharrefreplace')
b'&#163;13.55'
>>> "£13.55".encode('ascii', errors='backslashreplace')
b'\\xa313.55'
```

decodifica

```
>>> b = "£13.55".encode('utf8')
>>> b.decode('ascii', errors='replace')
'��13.55'
>>> b.decode('ascii', errors='ignore')
'13.55'
>>> b.decode('ascii', errors='backslashreplace')
'\\xc2\\xa313.55'
```

Morale

Da quanto sopra è chiaro che è fondamentale mantenere le codifiche corrette quando si ha a che fare con unicode e byte.

File I / O

I file aperti in modalità non binaria (es. `'r'` o `'w'`) gestiscono le stringhe. La codifica sorda è `'utf8'` .

```
open(fn, mode='r')                    # opens file for reading in utf8
open(fn, mode='r', encoding='utf16')  # opens file for reading utf16

# ERROR: cannot write bytes when a string is expected:
open("foo.txt", "w").write(b"foo")
```

I file aperti in modalità binaria (ad esempio `'rb'` o `'wb'`) gestiscono i byte. Nessun argomento di codifica può essere specificato in quanto non esiste alcuna codifica.

```
open(fn, mode='wb')  # open file for writing bytes

# ERROR: cannot write string when bytes is expected:
open(fn, mode='wb').write("hi")
```

Leggi Unicode e byte online: https://riptutorial.com/it/python/topic/1216/unicode-e-byte

Capitolo 200: Unzipping dei file

introduzione

Per estrarre o decomprimere un file tarball, ZIP o gzip, vengono forniti rispettivamente i file tarfile, zipfile e gzip di Python. Il modulo `TarFile.extractall(path=".", members=None)` Python fornisce la funzione `TarFile.extractall(path=".", members=None)` per l'estrazione da un file tarball. Il modulo `ZipFile.extractall([path[, members[, pwd]]])` Python fornisce la funzione `ZipFile.extractall([path[, members[, pwd]]])` per estrarre o decomprimere file compressi ZIP. Infine, il modulo gzip di Python fornisce la classe GzipFile per la decompressione.

Examples

Utilizzando Python ZipFile.extractall () per decomprimere un file ZIP

```
file_unzip = 'filename.zip'
unzip = zipfile.ZipFile(file_unzip, 'r')
unzip.extractall()
unzip.close()
```

Usando Python TarFile.extractall () per decomprimere un tarball

```
file_untar = 'filename.tar.gz'
untar = tarfile.TarFile(file_untar)
untar.extractall()
untar.close()
```

Leggi Unzipping dei file online: https://riptutorial.com/it/python/topic/9505/unzipping-dei-file

Capitolo 201: urllib

Examples

HTTP GET

Python 2.x 2.7

Python 2

```
import urllib
response = urllib.urlopen('http://stackoverflow.com/documentation/')
```

L'uso di `urllib.urlopen()` restituirà un oggetto risposta, che può essere gestito in modo simile a un file.

```
print response.code
# Prints: 200
```

`response.code` rappresenta il valore di ritorno http. 200 è OK, 404 è NotFound, ecc.

```
print response.read()
'<!DOCTYPE html>\r\n<html>\r\n<head>\r\n\r\n<title>Documentation - Stack. etc'
```

`response.read()` e `response.readlines()` possono essere utilizzati per leggere il file html effettivo restituito dalla richiesta. Questi metodi funzionano in modo simile a `file.read*`

Python 3.x 3.0

Python 3

```
import urllib.request

print(urllib.request.urlopen("http://stackoverflow.com/documentation/"))
# Prints: <http.client.HTTPResponse at 0x7f37a97e3b00>

response = urllib.request.urlopen("http://stackoverflow.com/documentation/")

print(response.code)
# Prints: 200
print(response.read())
# Prints: b'<!DOCTYPE html>\r\n<html>\r\n<head>\r\n\r\n<title>Documentation - Stack
Overflow</title>
```

Il modulo è stato aggiornato per Python 3.x, ma i casi d'uso rimangono sostanzialmente gli stessi. `urllib.request.urlopen` restituirà un oggetto simile a un file simile.

HTTP POST

Per i dati POST passare gli argomenti di query codificati come dati a urlopen ()

Python 2.x 2.7

Python 2

```
import urllib
query_parms = {'username':'stackoverflow', 'password':'me.me'}
encoded_parms = urllib.urlencode(query_parms)
response = urllib.urlopen("https://stackoverflow.com/users/login", encoded_parms)
response.code
# Output: 200
response.read()
# Output: '<!DOCTYPE html>\r\n<html>\r\n<head>\r\n\r\n<title>Log In - Stack Overflow'
```

Python 3.x 3.0

Python 3

```
import urllib
query_parms = {'username':'stackoverflow', 'password':'me.me'}
encoded_parms = urllib.parse.urlencode(query_parms).encode('utf-8')
response = urllib.request.urlopen("https://stackoverflow.com/users/login", encoded_parms)
response.code
# Output: 200
response.read()
# Output: b'<!DOCTYPE html>\r\n<html>....etc'
```

Decodifica i byte ricevuti in base alla codifica del tipo di contenuto

I byte ricevuti devono essere decodificati con la corretta codifica dei caratteri per essere interpretati come testo:

Python 3.x 3.0

```
import urllib.request

response = urllib.request.urlopen("http://stackoverflow.com/")
data = response.read()

encoding = response.info().get_content_charset()
html = data.decode(encoding)
```

Python 2.x 2.7

```
import urllib2
response = urllib2.urlopen("http://stackoverflow.com/")
data = response.read()

encoding = response.info().getencoding()
```

```
html = data.decode(encoding)
```

Leggi urllib online: https://riptutorial.com/it/python/topic/2645/urllib

Capitolo 202: Utilizzo dei loop all'interno delle funzioni

introduzione

In Python la funzione verrà restituita non appena l'esecuzione raggiunge l'istruzione "return".

Examples

Dichiarazione di ritorno all'interno del loop in una funzione

In questo esempio, la funzione verrà restituita non appena il valore var ha 1

```
def func(params):
    for value in params:
        print ('Got value {}'.format(value))

        if value == 1:
            # Returns from function as soon as value is 1
            print (">>>> Got 1")
            return

        print ("Still looping")

    return "Couldn't find 1"

func([5, 3, 1, 2, 8, 9])
```

produzione

```
Got value 5
Still looping
Got value 3
Still looping
Got value 1
>>>> Got 1
```

Leggi Utilizzo dei loop all'interno delle funzioni online:
https://riptutorial.com/it/python/topic/10883/utilizzo-dei-loop-all-interno-delle-funzioni

Capitolo 203: Utilizzo del modulo "pip": PyPI Package Manager

introduzione

A volte potrebbe essere necessario utilizzare il gestore di pacchetti pip all'interno di python ad es. quando alcune importazioni possono generare `ImportError` e si desidera gestire l'eccezione. Se si scompatta in Windows `Python_root/Scripts/pip.exe` all'interno viene memorizzato il file `__main__.py` , dove viene importata la classe `main` dal pacchetto `pip` . Questo significa che il pacchetto pip viene utilizzato ogni volta che si utilizza pip eseguibile. Per l'utilizzo di pip come eseguibile, consultare: pip: PyPI Package Manager

Sintassi

- pip. <function | attribute | class> dove function è una delle seguenti:
  - completamento automatico ()
    - Comando e completamento dell'opzione per il parser dell'opzione principale (e le opzioni) e relativi sottocomandi (e opzioni). Abilita procurando uno degli script di shell di completamento (bash, zsh o fish).
  - check_isolated (args)
    - param args {lista}
    - restituisce {booleano}
  - create_main_parser ()
    - restituisce {pip.baseparser.ConfigOptionParser object}
  - principali (args = None)
    - param args {lista}
    - restituisce {intero} Se non è fallito, restituisce 0
  - parseopts (args)
    - param args {lista}
  - get_installed_distributions ()
    - restituisce {elenco}
  - get_similar_commands (nome)
    - Il nome del comando è auto-corretto.
    - nome parametro {stringa}
    - restituisce {booleano}
  - get_summaries (ordinato = True)
    - Rende ordinate (nome del comando, riepilogo del comando) tuple.
  - get_prog ()
    - restituisce {string}
  - dist_is_editable (dist)
    - La distribuzione è un'installazione modificabile?
    - param dist {oggetto}
    - restituisce {booleano}

- commands_dict
  - attributo {dizionario}

Examples

Esempio di utilizzo di comandi

```
import pip

command = 'install'
parameter = 'selenium'
second_param = 'numpy' # You can give as many package names as needed
switch = '--upgrade'

pip.main([command, parameter, second_param, switch])
```

Solo i parametri necessari sono obbligatori, quindi sia `pip.main(['freeze'])` che `pip.main(['freeze', '', ''])` sono accettabili.

Installazione batch

È possibile passare molti nomi di pacchetti in un'unica chiamata, ma se un aggiornamento / installazione fallisce, l'intero processo di installazione si interrompe e termina con lo stato '1'.

```
import pip

installed = pip.get_installed_distributions()
list = []
for i in installed:
    list.append(i.key)

pip.main(['install']+list+['--upgrade'])
```

Se non vuoi fermarti quando alcune installazioni falliscono, chiama l'installazione in loop.

```
for i in installed:
        pip.main(['install']+i.key+['--upgrade'])
```

Gestire l'eccezione di ImportError

Quando si usa il file python come modulo non è necessario controllare sempre se il pacchetto è installato ma è comunque utile per gli script.

```
if __name__ == '__main__':
    try:
        import requests
    except ImportError:
        print("To use this module you need 'requests' module")
        t = input('Install requests? y/n: ')
        if t == 'y':
            import pip
            pip.main(['install', 'requests'])
```

```
            import requests
            import os
            import sys
            pass
        else:
            import os
            import sys
            print('Some functionality can be unavailable.')
else:
    import requests
    import os
    import sys
```

Forza installazione

Molti pacchetti, ad esempio sulla versione 3.4, funzionerebbero su 3.6, ma se non ci sono distribuzioni per piattaforma specifica, non possono essere installati, ma c'è una soluzione alternativa. Nella convenzione di denominazione dei file .whl (conosciuta come ruote) è possibile decidere se è possibile installare il pacchetto sulla piattaforma specificata. Per esempio. `scikit_learn-0.18.1-cp36-cp36m-win_amd64.whl` [`scikit_learn-0.18.1-cp36-cp36m-win_amd64.whl`] - [versione] - [interprete python] - [interprete python] - [Sistema operativo] .whl. Se il nome del file wheel viene modificato, quindi la piattaforma non corrisponde, pip tenta di installare il pacchetto anche se la versione platform o python non corrisponde. Rimuovendo la piattaforma o l'interprete dal nome, si verificherà un errore nell'ultimo verso del modulo pip `kjhfkjdf.whl is not a valid wheel filename. .`

In alternativa, il file .whl può essere decompresso utilizzando un archiviatore come 7-zip. - Di solito contiene la cartella meta di distribuzione e la cartella con i file di origine. Questi file sorgente possono essere semplicemente decompressi nella directory `site-packges` meno che questa rotella non contenga script di installazione, in tal caso, deve essere eseguita per prima.

Leggi Utilizzo del modulo "pip": PyPI Package Manager online:
https://riptutorial.com/it/python/topic/10730/utilizzo-del-modulo--pip---pypi-package-manager

Capitolo 204: Verifica dell'esistenza e delle autorizzazioni del percorso

Parametri

Parametro	Dettagli
os.F_OK	Valore da passare come parametro della modalità di accesso () per verificare l'esistenza del percorso.
os.R_OK	Valore da includere nel parametro mode di access () per verificare la leggibilità del percorso.
os.W_OK	Valore da includere nel parametro mode di access () per verificare la scrivibilità del percorso.
os.X_OK	Valore da includere nel parametro mode di access () per determinare se il path può essere eseguito.

Examples

Esegui i controlli usando os.access

`os.access` è una soluzione molto migliore per verificare se la directory esiste ed è accessibile per la lettura e la scrittura.

```
import os
path = "/home/myFiles/directory1"

## Check if path exists
os.access(path, os.F_OK)

## Check if path is Readable
os.access(path, os.R_OK)

## Check if path is Wriable
os.access(path, os.W_OK)

## Check if path is Execuatble
os.access(path, os.E_OK)
```

inoltre è possibile eseguire tutti i controlli insieme

```
os.access(path, os.F_OK & os.R_OK & os.W_OK & os.E_OK)
```

Tutto quanto sopra restituisce `True` se l'accesso è consentito e `False` se non consentito. Questi

sono disponibili su Unix e Windows.

Leggi Verifica dell'esistenza e delle autorizzazioni del percorso online: https://riptutorial.com/it/python/topic/1262/verifica-dell-esistenza-e-delle-autorizzazioni-del-percorso

Capitolo 205: Visualizzazione dei dati con Python

Examples

matplotlib

Matplotlib è una libreria di plottaggio matematica per Python che offre una varietà di diverse funzionalità di tracciamento.

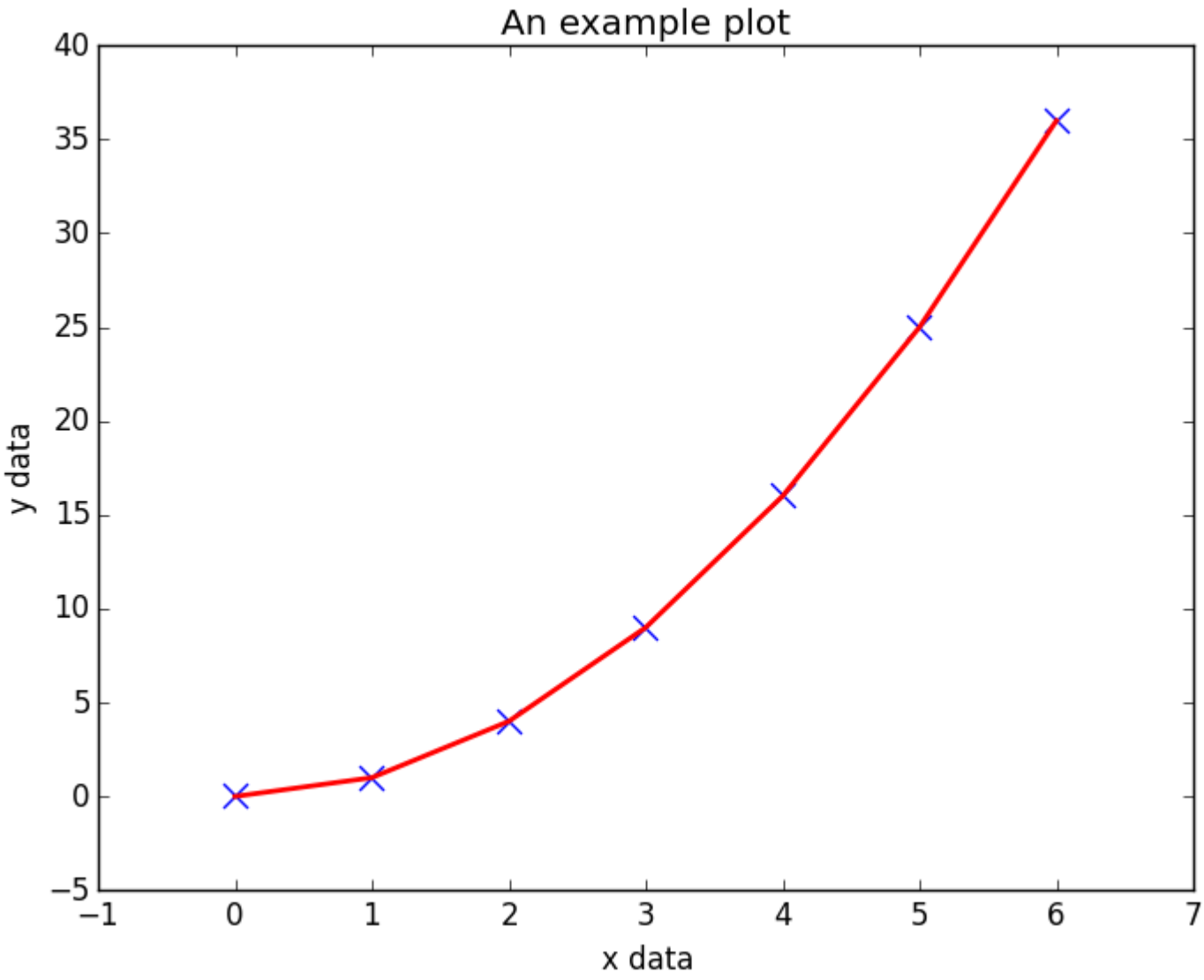
La documentazione di matplotlib può essere trovata qui , con i documenti SO disponibili qui .

Matplotlib fornisce due metodi distinti per la stampa, sebbene siano per lo più intercambiabili:

- In primo luogo, matplotlib fornisce l'interfaccia di `pyplot` , un'interfaccia diretta e di semplice utilizzo che consente di tracciare grafici complessi in uno stile simile a MATLAB.
- In secondo luogo, matplotlib consente all'utente di controllare i diversi aspetti (assi, linee, zecche, ecc.) Direttamente utilizzando un sistema basato su oggetti. Questo è più difficile ma consente il controllo completo dell'intera trama.

Di seguito è riportato un esempio di utilizzo dell'interfaccia di `pyplot` per tracciare alcuni dati generati:

```
import matplotlib.pyplot as plt

# Generate some data for plotting.
x = [0, 1, 2, 3, 4, 5, 6]
y = [i**2 for i in x]

# Plot the data x, y with some keyword arguments that control the plot style.
# Use two different plot commands to plot both points (scatter) and a line (plot).

plt.scatter(x, y, c='blue', marker='x', s=100) # Create blue markers of shape "x" and size 100
plt.plot(x, y, color='red', linewidth=2) # Create a red line with linewidth 2.

# Add some text to the axes and a title.
plt.xlabel('x data')
plt.ylabel('y data')
plt.title('An example plot')

# Generate the plot and show to the user.
plt.show()
```



Si noti che `plt.show()` è noto per essere problematico in alcuni ambienti a causa dell'esecuzione di `matplotlib.pyplot` in modalità interattiva e, in tal caso, il comportamento di blocco può essere sovrascritto esplicitamente passando in un argomento facoltativo, `plt.show(block=True)` , per alleviare il problema.

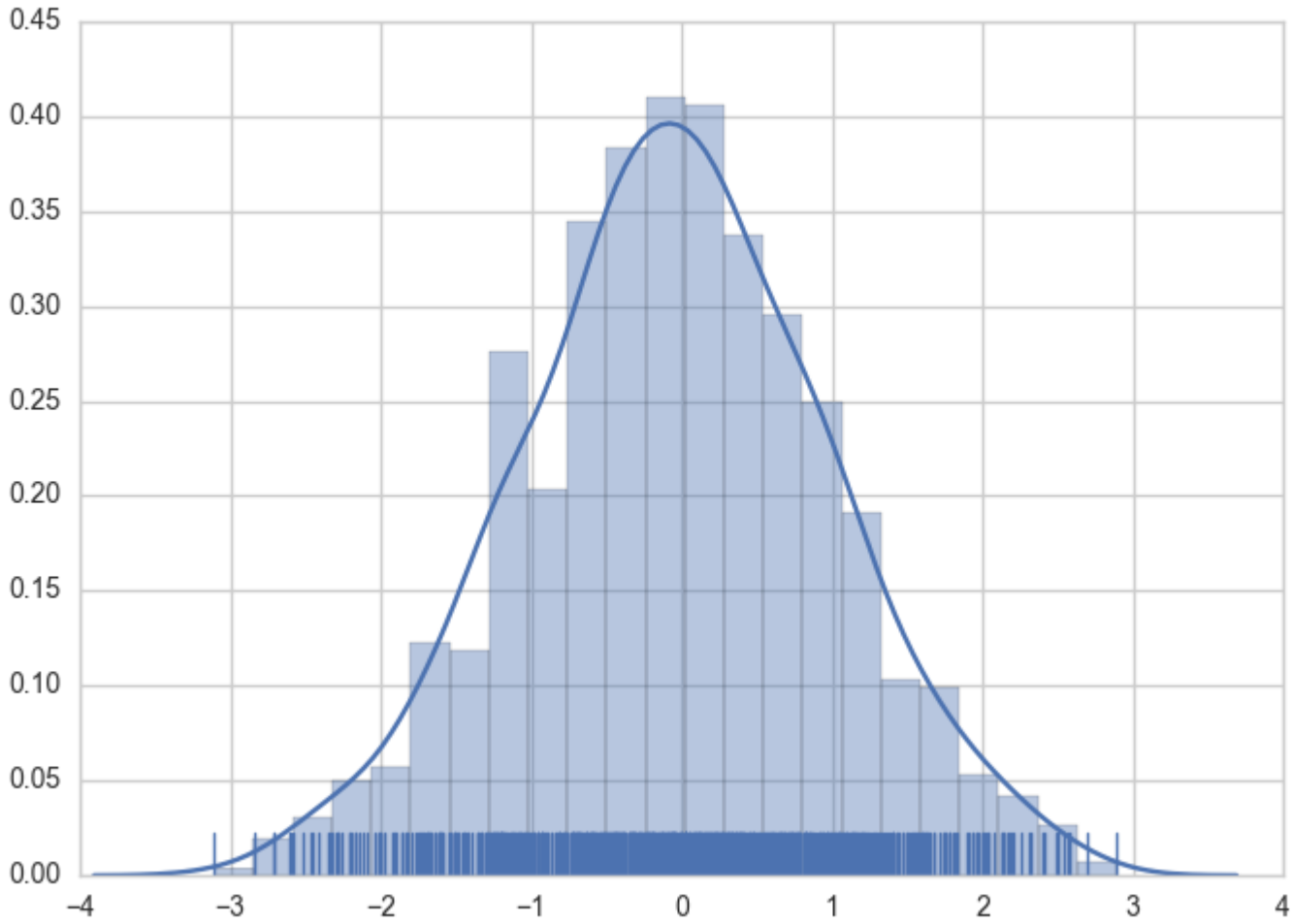
Seaborn

Seaborn è un wrapper di Matplotlib che semplifica la creazione di grafici statistici comuni. L'elenco di grafici supportati include grafici di distribuzione univariati e bivariati, diagrammi di regressione e un numero di metodi per il tracciamento di variabili categoriali. L'elenco completo dei grafici forniti da Seaborn si trova nel riferimento API .

Creare grafici in Seaborn è semplice come chiamare la funzione grafica appropriata. Ecco un esempio di creazione di un istogramma, stima della densità del kernel e trama del tappeto per dati generati casualmente.

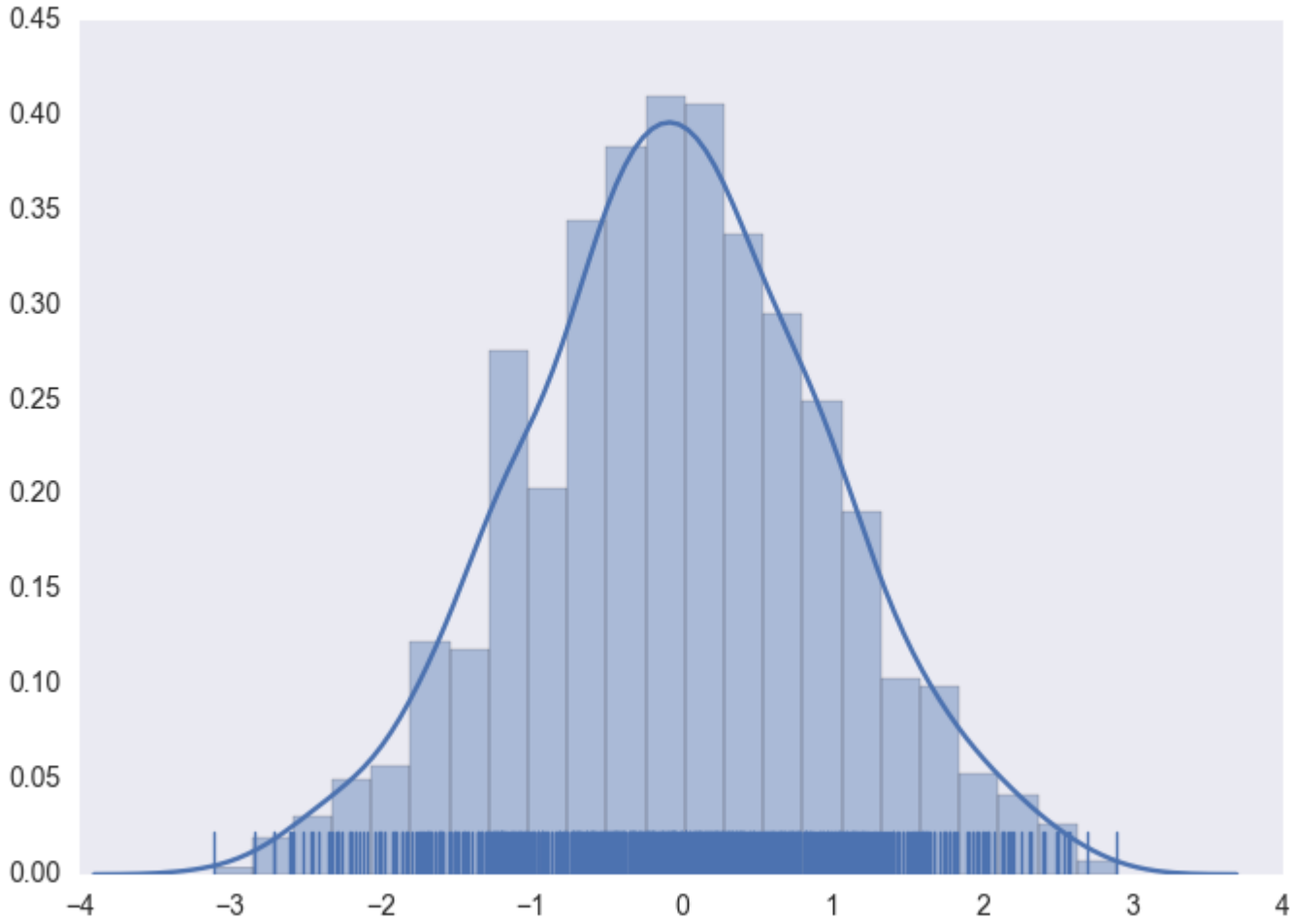
```
import numpy as np  # numpy used to create data from plotting
import seaborn as sns  # common form of importing seaborn
```

```
# Generate normally distributed data
data = np.random.randn(1000)

# Plot a histogram with both a rugplot and kde graph superimposed
sns.distplot(data, kde=True, rug=True)
```



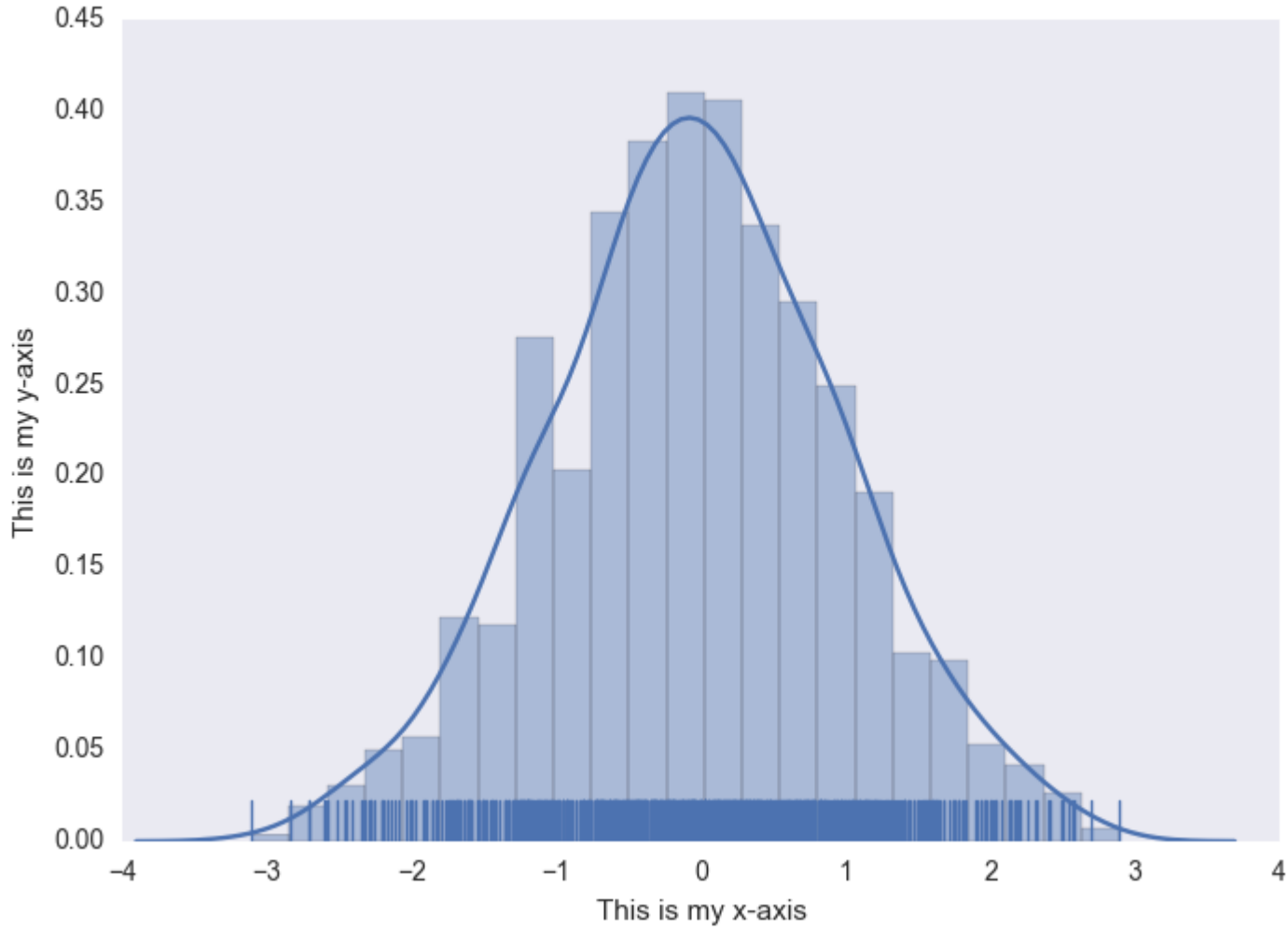
Lo stile del grafico può anche essere controllato usando una sintassi dichiarativa.

```
# Using previously created imports and data.

# Use a dark background with no grid.
sns.set_style('dark')
# Create the plot again
sns.distplot(data, kde=True, rug=True)
```



Come bonus aggiuntivo, i normali comandi matplotlib possono ancora essere applicati ai grafici di Seaborn. Ecco un esempio di aggiunta di titoli di assi al nostro istogramma creato in precedenza.

```
# Using previously created data and style

# Access to matplotlib commands
import matplotlib.pyplot as plt

# Previously created plot.
sns.distplot(data, kde=True, rug=True)
# Set the axis labels.
plt.xlabel('This is my x-axis')
plt.ylabel('This is my y-axis')
```



MayaVi

MayaVI è uno strumento di visualizzazione 3D per dati scientifici. Utilizza il Visualization Tool Kit o VTK sotto il cofano. Usando la potenza di VTK , **MayaVI** è in grado di produrre una varietà di grafici e figure tridimensionali. È disponibile come applicazione software separata e anche come libreria. Simile a Matplotlib , questa libreria fornisce un'interfaccia linguistica di programmazione orientata agli oggetti per creare grafici senza dover conoscere **VTK** .

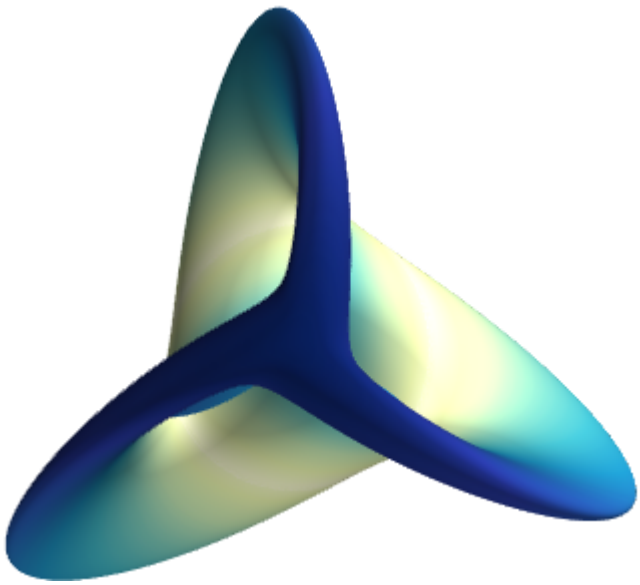
MayaVI è disponibile solo nella serie Python 2.7x! Si spera di essere presto disponibile nella serie Python 3-x! (Sebbene si noti un certo successo quando si usano le sue dipendenze in Python 3)

La documentazione può essere trovata qui . Alcuni esempi di gallerie sono disponibili qui

Ecco un tracciato di esempio creato usando **MayaVI** dalla documentazione.

```
# Author: Gael Varoquaux <gael.varoquaux@normalesup.org>
# Copyright (c) 2007, Enthought, Inc.
# License: BSD Style.
```

```
from numpy import sin, cos, mgrid, pi, sqrt
from mayavi import mlab

mlab.figure(fgcolor=(0, 0, 0), bgcolor=(1, 1, 1))
u, v = mgrid[- 0.035:pi:0.01, - 0.035:pi:0.01]

X = 2 / 3. * (cos(u) * cos(2 * v)
        + sqrt(2) * sin(u) * cos(v)) * cos(u) / (sqrt(2) -
                                                 sin(2 * u) * sin(3 * v))
Y = 2 / 3. * (cos(u) * sin(2 * v) -
        sqrt(2) * sin(u) * sin(v)) * cos(u) / (sqrt(2)
        - sin(2 * u) * sin(3 * v))
Z = -sqrt(2) * cos(u) * cos(u) / (sqrt(2) - sin(2 * u) * sin(3 * v))
S = sin(u)

mlab.mesh(X, Y, Z, scalars=S, colormap='YlGnBu', )

# Nice view from the front
mlab.view(.0, - 5.0, 4)
mlab.show()
```



Plotly

Plotly è una piattaforma moderna per il tracciamento e la visualizzazione dei dati. Utile per produrre una varietà di trame, in particolare per le scienze di dati, **Plotly** è disponibile come libreria per **Python** , **R** , **JavaScript** , **Julia** e, **MATLAB** . Può anche essere usato come applicazione web con queste lingue.

Gli utenti possono installare la libreria grafica e utilizzarla offline dopo l'autenticazione dell'utente. L'installazione di questa libreria e l'autenticazione offline viene fornita qui . Inoltre, le trame possono essere fatte anche nei **quaderni Jupyter** .

L'utilizzo di questa libreria richiede un account con nome utente e password. Ciò consente all'area di lavoro di salvare grafici e dati sul cloud.

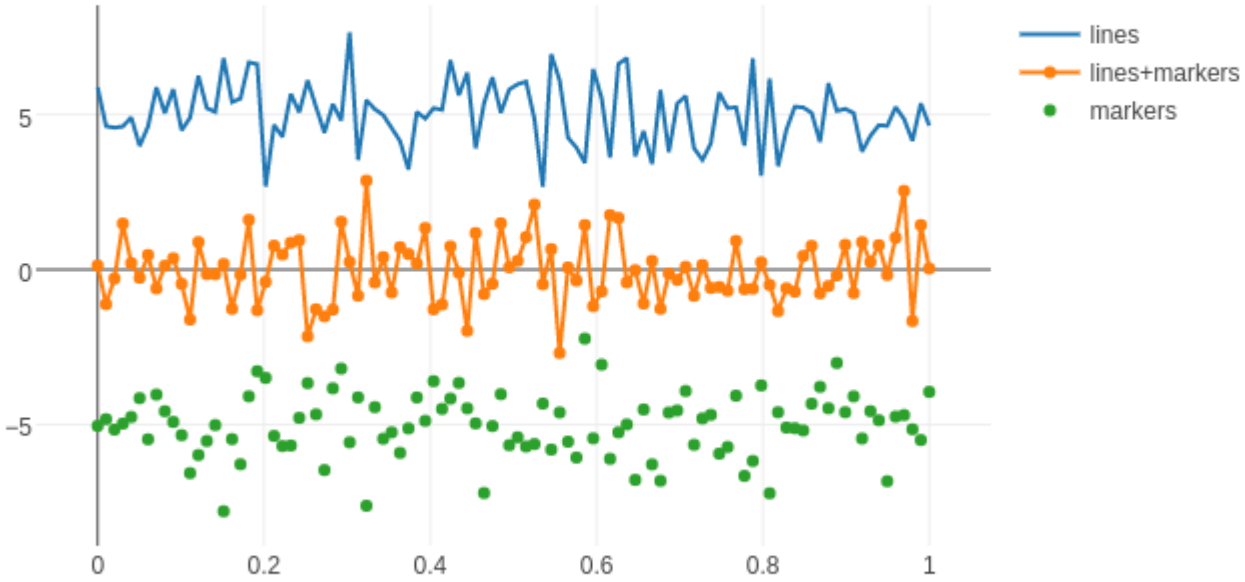
La versione gratuita della libreria ha alcune funzionalità leggermente limitate e progettata per fare 250 grafici al giorno. La versione a pagamento ha tutte le caratteristiche, download di trama

illimitati e più spazio di archiviazione privato. Per maggiori dettagli, si può visitare la pagina principale qui .

Per documentazione ed esempi, si può andare qui

Un grafico di esempio dagli esempi di documentazione:

```
import plotly.graph_objs as go
import plotly as ply

# Create random data with numpy
import numpy as np

N = 100
random_x = np.linspace(0, 1, N)
random_y0 = np.random.randn(N)+5
random_y1 = np.random.randn(N)
random_y2 = np.random.randn(N)-5

# Create traces
trace0 = go.Scatter(
    x = random_x,
y = random_y0,
mode = 'lines',
name = 'lines'
)
trace1 = go.Scatter(
    x = random_x,
    y = random_y1,
    mode = 'lines+markers',
    name = 'lines+markers'
)
trace2 = go.Scatter(
    x = random_x,
    y = random_y2,
    mode = 'markers',
    name = 'markers'
)
data = [trace0, trace1, trace2]

ply.offline.plot(data, filename='line-mode')
```



Leggi Visualizzazione dei dati con Python online:
https://riptutorial.com/it/python/topic/2388/visualizzazione-dei-dati-con-python

Capitolo 206: WebSockets

Examples

Eco semplice con aiohttp

`aiohttp` fornisce `aiohttp` asincroni.

Python 3.x 3.5

```
import asyncio
from aiohttp import ClientSession

with ClientSession() as session:
    async def hello_world():

        websocket = await session.ws_connect("wss://echo.websocket.org")

        websocket.send_str("Hello, world!")

        print("Received:", (await websocket.receive()).data)

        await websocket.close()

    loop = asyncio.get_event_loop()
    loop.run_until_complete(hello_world())
```

Classe wrapper con aiohttp

`aiohttp.ClientSession` può essere utilizzato come genitore per una classe WebSocket personalizzata.

Python 3.x 3.5

```
import asyncio
from aiohttp import ClientSession

class EchoWebSocket(ClientSession):

    URL = "wss://echo.websocket.org"

    def __init__(self):
        super().__init__()
        self.websocket = None

    async def connect(self):
        """Connect to the WebSocket."""
        self.websocket = await self.ws_connect(self.URL)

    async def send(self, message):
        """Send a message to the WebSocket."""
        assert self.websocket is not None, "You must connect first!"
        self.websocket.send_str(message)
        print("Sent:", message)
```

```
    async def receive(self):
        """Receive one message from the WebSocket."""
        assert self.websocket is not None, "You must connect first!"
        return (await self.websocket.receive()).data

    async def read(self):
        """Read messages from the WebSocket."""
        assert self.websocket is not None, "You must connect first!"

        while self.websocket.receive():
            message = await self.receive()
            print("Received:", message)
            if message == "Echo 9!":
                break

async def send(websocket):
    for n in range(10):
        await websocket.send("Echo {}!".format(n))
        await asyncio.sleep(1)

loop = asyncio.get_event_loop()

with EchoWebSocket() as websocket:

    loop.run_until_complete(websocket.connect())

    tasks = (
        send(websocket),
        websocket.read()
    )

    loop.run_until_complete(asyncio.wait(tasks))

    loop.close()
```

Utilizzando Autobahn come una Websocket Factory

Il pacchetto Autobahn può essere utilizzato per le fabbriche di server socket web Python.

Documentazione del pacchetto Autobahn Python

Per installare, in genere si dovrebbe semplicemente utilizzare il comando del terminale

(Per Linux):

```
sudo pip install autobahn
```

(Per Windows):

```
python -m pip install autobahn
```

Quindi, un semplice server echo può essere creato in uno script Python:

```
from autobahn.asyncio.websocket import WebSocketServerProtocol
```

```
class MyServerProtocol(WebSocketServerProtocol):
    '''When creating server protocol, the
    user defined class inheriting the
    WebSocketServerProtocol needs to override
    the onMessage, onConnect, et-c events for
    user specified functionality, these events
    define your server's protocol, in essence'''
    def onMessage(self,payload,isBinary):
        '''The onMessage routine is called
        when the server receives a message.
        It has the required arguments payload
        and the bool isBinary. The payload is the
        actual contents of the "message" and isBinary
        is simply a flag to let the user know that
        the payload contains binary data. I typically
        elsewise assume that the payload is a string.
        In this example, the payload is returned to sender verbatim.'''
        self.sendMessage(payload,isBinary)
if__name__=='__main__':
    try:
        importasyncio
    except ImportError:
        '''Trollius = 0.3 was renamed'''
        import trollius as asyncio
    from autobahn.asyncio.websocketimportWebSocketServerFactory
    factory=WebSocketServerFactory()
    '''Initialize the websocket factory, and set the protocol to the
    above defined protocol(the class that inherits from
    autobahn.asyncio.websocket.WebSocketServerProtocol)'''
    factory.protocol=MyServerProtocol
    '''This above line can be thought of as "binding" the methods
    onConnect, onMessage, et-c that were described in the MyServerProtocol class
    to the server, setting the servers functionality, ie, protocol'''
    loop=asyncio.get_event_loop()
    coro=loop.create_server(factory,'127.0.0.1',9000)
    server=loop.run_until_complete(coro)
    '''Run the server in an infinite loop'''
    try:
        loop.run_forever()
    except KeyboardInterrupt:
        pass
    finally:
        server.close()
        loop.close()
```

In questo esempio, un server viene creato sull'host locale (127.0.0.1) sulla porta 9000. Questo è l'IP e la porta di ascolto. Si tratta di informazioni importanti, in quanto utilizzando questo, è possibile identificare l'indirizzo LAN del computer e il port forwarding dal modem, indipendentemente dai router che si hanno sul computer. Quindi, utilizzando google per indagare sul tuo IP WAN, potresti progettare il tuo sito Web per inviare messaggi WebSocket al tuo indirizzo IP WAN, sulla porta 9000 (in questo esempio).

È importante eseguire il port forwarding dal tuo modem, il che significa che se hai router collegati a margherita al modem, inserisci le impostazioni di configurazione del modem, porta in avanti dal modem al router connesso e così via fino al router finale del tuo computer è collegato a sta ricevendo le informazioni ricevute sulla porta modem 9000 (in questo esempio) inoltrate ad esso.

Leggi WebSockets online: https://riptutorial.com/it/python/topic/4751/websockets

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare con Python Language	A. Raza, Aaron Critchley, Abhishek Jain, AER, afeique, Akshay Kathpal, alejosocorro, Alessandro Trinca Tornidor, Alex Logan, ALinuxLover, Andrea, Andrii Abramov, Andy, Andy Hayden, angussidney, Ani Menon, Anthony Pham, Antoine Bolvy, Aquib Javed Khan, Ares, Arpit Solanki, B8vrede, Baaing Cow, baranskistad, Brian C, Bryan P, BSL-5, BusyAnt, Cbeb24404, ceruleus, ChaoticTwist, Charlie H, Chris Midgley, Christian Ternus, Claudiu, Clíodhna, CodenameLambda, cLDs EED, Community, Conrad.Dean, Daksh Gupta, Dania, Daniel Minnaar, Darth Shadow, Dartmouth, deeenes, Delgan, depperm, DevD, dodell, Douglas Starnes, duckman_1991, Eamon Charles, edawine, Elazar, eli-bd, Enrico Maria De Angelis, Erica, Erica, ericdwang, Erik Godard, EsmaeelE, Filip Haglund, Firix, fox, Franck Dernoncourt, Fred Barclay, Freddy, Gerard Roche, glS, GoatsWearHats, GThamizh, H. Pauwelyn, hardmooth, hayalci, hichris123, Ian, IanAuld, icesin, Igor Raush, Ilyas Mimouni, itsthejoker, J F, Jabba, jalanb, James, James Taylor, Jean-Francois T., jedwards, Jeffrey Lin, jfunez, JGreenwell, Jim Fasarakis Hilliard, jim opleydulven, jimsug, jmunsch, Johan Lundberg, John Donner, John Slegers, john400, jonrsharpe, Joseph True, JRodDynamite, jtbandes, Juan T, Kamran Mackey, Karan Chudasama, KerDam, Kevin Brown, Kiran Vemuri, kisanme, Lafexlos, Leon, Leszek Kicior, LostAvatar, Majid, manu, MANU, Mark Miller, Martijn Pieters, Mathias711, matsjoyce, Matt, Mattew Whitt, mdegis, Mechanic, Media, mertyildiran, metahost, Mike Driscoll, MikJR, Miljen Mikic, mnoronha, Morgoth, moshemeirelles, MSD, MSeifert, msohng, msw, muddyfish, Mukund B, Muntasir Alam, Nathan Arthur, Nathaniel Ford, Ned Batchelder, Ni., niyasc, noɥʎd ʎzɐɹƆ, numbermaniac, orvi, Panda, Patrick Haugh, Pavan Nath, Peter Masiar, PSN, PsyKzz, pylang, pzp, Qchmqs, Quill, Rahul Nair, Rakitić, Ram Grandhi, rfkortekaas, rick112358, Robotski, rrao, Ryan Hilbert, Sam Krygsheld, Sangeeth Sudheer, SashaZd, Selcuk, Severiano Jaramillo Quintanar, Shiven, Shoe, Shog9, Sigitas Mockus, Simplans, Slayther, stark, StuxCrystal, SuperBiasedMan, Sнаđoшƒa, taylor swift, techydesigner, Tejus Prasad, TerryA, The_Curry_Man, TheGenie OfTruth, Timotheus.Kampik, tjohnson, Tom Barron, Tom de Geus, Tony Suffolk 66, tonyo, TPVasconcelos,

		user2314737, user2853437, user312016, Utsav T, vaichidrewar, vasili111, Vin, W.Wong, weewooquestionaire, Will, wintermute, Yogendra Sharma, Zach Janicki, Zags
2	* args e ** kwargs	cjds, Eric Zhang, ericmarkmartin, Geeklhem, J F, Jeff Hutchins, Jim Fasarakis Hilliard, JuanPablo, kdopen, loading..., Marlon Abeykoon, Mattew Whitt, Pasha, pcurry, PsyKzz, Scott Mermelstein, user2314737, Valentin Lorentz, Veedrac
3	Abstract Base Classes (abc)	Akshat Mahajan, Alessandro Trinca Tornidor, JGreenwell, Kevin Brown, Mattew Whitt, mkrieger1, SashaZd, Stephen Leppik
4	accantonare	Biswa_9937
5	Accesso agli attributi	Elazar, SashaZd, SuperBiasedMan
6	Accesso al codice sorgente e al bytecode Python	muddyfish, StuxCrystal, user2314737
7	Accesso al database	Alessandro Trinca Tornidor, Antonio, bee-sting, cldseed, D. Alveno, John Y, LostAvatar, mbsingh, Michel Touw, qwertyuip9, RamenChef, rrawat, Stephen Leppik, Stephen Nyamweya, sumitroy, user2314737, valeas, zweiterlinde
8	Albero di sintassi astratto	Teepeemm
9	Alternative per cambiare dichiarazione da altre lingue	davidism, J F, zmo, Валерий Павлов
10	ambiente virtuale con virtualenvwrapper	Sirajus Salayhin
11	Ambienti virtuali	Adrian17, Artem Kolontay, ArtOfCode, Bhargav, brennan, Dair, Daniil Ryzhkov, Darkade, Darth Shadow, edwinksl, Fernando, ghostarbeiter, ha_1694, Hans Then, Iancnorden, J F, Majid, Marco Pashkov, Matt Giltaji, Mattew Whitt, nehemiah, Nuhil Mehdy, Ortomala Lokni, Preston, pylang, qwertyuip9, RamenChef, Régis B., Sebastian Schrader, Serenity, Shantanu Alshi, Shrey Gupta, Simon Fraser, Simplans, wrwrwr, ychaouche, zopieux, zvezda
12	Ambito e legame variabili	Anthony Pham, davidism, Elazar, Esteis, Mike Driscoll, SuperBiasedMan, user2314737, zvone

13	Analisi HTML	alecxe, talhasch
14	Analizzare gli argomenti della riga di comando	amblina, Braiam, Claudiu, cledoux, Elazar, Gerard Roche, krato, loading..., Marco Pashkov, Or Duan, Pasha, RamenChef , rfkortekaas, Simplans, Thomas Gerot, Topperfalkon, zmo, zondo
15	ArcPy	Midavalo, PolyGeo, Zhanping Shi
16	Array	Andy, Pavan Nath, RamenChef, Vin
17	Audio	blueberryfields, Comrade SparklePony, frankyjuang, jmunsch, orvi, qwertyuip9, Stephen Leppik, Thomas Gerot
18	Aumentare errori / eccezioni personalizzati	naren
19	Blocchi di codice, frame di esecuzione e spazi dei nomi	Jeremy, Mohammed Salman
20	Calcolo parallelo	Akshat Mahajan, Dair, Franck Dernoncourt, J F, Mahdi, nlsdfnbch, Ryan Smith, Vinzee, Xavier Combelle
21	Caratteristiche nascoste	Aaron Hall, Akshat Mahajan, Anthony Pham, Antti Haapala, Byte Commander, dermen, Elazar, Ellis, ericmarkmartin, Fermi paradox, Ffisegydd, japborst, Jim Fasarakis Hilliard, jonrsharpe , Justin, kramer65, Lafexlos, LDP, Morgan Thrapp, muddyfish, nico, OrangeTux, pcurry, Pythonista, Selcuk, Serenity, Tejas Jadhav, tobias_k, Vlad Shcherbina, Will
22	ChemPy - pacchetto python	Biswa_9937
23	Chiama Python da C #	Julij Jegorov
24	Classi	Aaron Hall, Ahsanul Haque, Akshat Mahajan, Andrzej Pronobis , Anthony Pham, Avantol13, Camsbury, cfi, Community, Conrad.Dean, Daksh Gupta, Darth Shadow, Dartmouth, depperm, Elazar, Ffisegydd, Haris, Igor Raush, InitializeSahib, J F, jkdev, jlarsch, John Militer, Jonas S, Jonathan, Kallz, KartikKannapur, Kevin Brown, Kinifwyne, Leo, Liteye, lmiguelvargasf, Mailerdaimon, Martijn Pieters, Massimiliano Kraus, Mattew Whitt, MrP01, Nathan Arthur, ojas mohril, Pasha, Peter Steele, pistache, Preston, pylang, Richard Fitzhugh, rohittk239, Rushy Panchal, Sempoo, Simplans, Soumendra Kumar Sahoo, SuperBiasedMan, techydesigner,

		then0rTh, Thomas Gerot, Tony Suffolk 66, tox123, UltraBob, user2314737, wrwrwr, Yogendra Sharma
25	Commenti e documentazione	Ani Menon, FunkySayu, MattCorr, SuperBiasedMan, TuringTux
26	Comprensione delle liste	3442, 4444, acdr, Ahsanul Haque, Akshay Anand, Akshit Soota, Alleo, Amir Rachum, André Laszlo, Andy Hayden, Ankit Kumar Singh, Antoine Bolvy, APerson, Ashwinee K Jha, B8vrede, bfontaine, Brian Cline, Brien, Casebash, Celeo, cfi, ChaoticTwist, Charles, Charlie H, Chong Tang, Community, Conrad.Dean, Dair, Daniel Stradowski, Darth Shadow, Dartmouth, David Heyman, Delgan, Dima Tisnek, eenblam, Elazar, Emma, enrico.bacis, EOL, ericdwang, ericmarkmartin, Esteis, Faiz Halde, Felk, Fermi paradox, Florian Bender, Franck Dernoncourt, Fred Barclay, freidrichen, G M, Gal Dreiman, garg10may, ghostarbeiter, GingerHead, griswolf, Hannele, Harry, Hurkyl, IanAuld, iankit, Infinity, intboolstring, J F, J0HN, James, JamesS, Jamie Rees, jedwards, Jeff Langemeier, JGreenwell, JHS, jjwatt, JKillian, JNat, joel3000, John Slegers, Jon, jonrsharpe, Josh Caswell, JRodDynamite, Julian, justhalf, Kamyar Ghasemlou, kdopen, Kevin Brown, KIDJourney, Kwarrtz, Lafexlos, lapis, Lee Netherton, Liteye, Locane, Lyndsy Simon, machine yearning, Mahdi, Marc, Markus Meskanen, Martijn Pieters, Matt, Matt Giltaji, Matt S, Mattew Whitt, Maximillian Laumeister, mbrig, Mirec Miskuf, Mitch Talmadge, Morgan Thrapp, MSeifert, muddyfish, n8henrie, Nathan Arthur, nehemiah, noɥʎɑʎzɐɔ, Or East, Ortomala Lokni, pabouk, Panda, Pasha, pktangyue, Preston, Pro Q, pylang, R Nar, Rahul Nair, rap-2-h, Riccardo Petraglia, rll, Rob Fagen, rrao, Ryan Hilbert, Ryan Smith, ryanyuyu, Samuel McKay, sarvajeetsuman, Sayakiss, Sebastian Kreft, Shoe, SHOWMEWHATYOUGOT, Simplans, Slayther, Slickytail, solidcell, StuxCrystal, sudo bangbang, Sunny Patel, SuperBiasedMan, syb0rg, Symmitchry, The_Curry_Man, theheadofabroom, Thomas Gerot, Tim McNamara, Tom Barron, user2314737, user2357112, Utsav T, Valentin Lorentz, Veedrac, viveksyngh, vog, W.P. McNeill, Will, Will, Wladimir Palant, Wolf, XCoder Real, yurib, Yury Fedorov, Zags, Zaz
27	Concorrenza Python	David Heyman, Faiz Halde, Iván Rodríguez Torres, J F, Thomas Moreau, Tyler Gubala
28	Condizionali	Andy Hayden, BusyAnt, Chris Larson, deepakkt, Delgan, Elazar, evuez, Ffisegydd, Geeklhem, Hannes Karppila, James, Kevin Brown, krato, Max Feng, noɥʎɑʎzɐɔ, rajah9, rrao, SashaZd, Simplans, Slayther, Soumendra Kumar Sahoo, Thomas Gerot, Trimax, Valentin Lorentz, Vinzee, wwii, xgord,

		Zack
29	ConfigParser	Chinmay Hegde, Dunatotatos
30	confronti	Anthony Pham, Ares, Elazar, J F, MSeifert, Shawn Mehan, SuperBiasedMan, Will, Xavier Combelle
31	Connessione di Python a SQL Server	metmirr
32	Conteggio	Andy Hayden, MSeifert, Peter Mølgaard Pallesen, pylang
33	Copia dei dati	hashcode55, StuxCrystal
34	Crea un ambiente virtuale con virtualenvwrapper in windows	Sirajus Salayhin
35	Creare un servizio Windows usando Python	Simon Hibbs
36	Creazione di pacchetti Python	Claudiu, KeyWeeUsr, Marco Pashkov, pylang, SuperBiasedMan, Thtu
37	ctypes	Or East
38	Cuscino	Razik
39	Data e ora	Ajean, alecxe, Andy, Antti Haapala, BusyAnt, Conrad.Dean, Elazar, ghostarbeiter, J F, Jeffrey Lin, jonrsharpe, Kevin Brown , Nicole White, nlsdfnbch, Ohad Eytan, Paul, paulmorriss, proprius, RahulHP, RamenChef, sagism, Simplans, Sirajus Salayhin, Suku, Will
40	Dati binari	Eleftheria, evuez, mnoronha
41	Debug	Aldo, B8vrede, joel3000, Sardathrion, Sardorbek Imomaliev, Vlad Bezden
42	decoratori	Alessandro Trinca Tornidor, ChaoticTwist, Community, Dair, doratheexplorer0911, Emolga, greut, iankit, JGreenwell, jonrsharpe, kefkius, Kevin Brown, Mattew Whitt, MSeifert, muddyfish, Mukunda Modell, Nearoo, Nemo, Nuno André, Pasha, Rob Bednark, seenu s, Shreyash S Sarnayak, Simplans, StuxCrystal, Suhas K, technusm1, Thomas Gerot, tyteen4a03, Wladimir Palant, zvone
43	Definire funzioni con	zenlc2000

	argomenti lista	
44	dentellatura	Alessandro Trinca Tornidor, depperm, J F, JGreenwell, Matt Giltaji, Pasha, RamenChef, Stephen Leppik
45	descrittore	bbayles, cizixs, Nemo, pylang, SuperBiasedMan
46	Differenza tra modulo e pacchetto	DeepSpace, Simplans, tjohnson
47	Distribuzione	Alessandro Trinca Tornidor, JGreenwell, metahost, Pigman168, RamenChef, Stephen Leppik
48	Dizionario	Amir Rachum, Anthony Pham, APerson, ArtOfCode, BoppreH, Burhan Khalid, Chris Mueller, cizixs, depperm, Ffisegydd, Gareth Latty, Guy, helpful, iBelieve, Igor Raush, Infinity, James, JGreenwell, jonrsharpe, Karsten 7., kdopen, machine yearning, Majid, mattgathu, Mechanic, MSeifert, muddyfish, Nathan, nlsdfnbch, noɥʎdʌzɐɔ, ronrest, Roy Iacob, Shawn Mehan, Simplans, SuperBiasedMan, TehTris, Valentin Lorentz, viveksyngh, Xavier Combelle
49	Django	code_geek, orvi
50	eccezioni	Adrian Antunez, Alessandro Trinca Tornidor, Alfe, Andy, Benjamin Hodgson, Brian Rodriguez, BusyAnt, Claudiu, driax, Elazar, flazzarini, ghostarbeiter, Ilia Barahovski, J F, Marco Pashkov, muddyfish, noɥʎdʌzɐɔ, Paul Weaver, Rahul Nair, RamenChef, Shawn Mehan, Shiven, Shkelqim Memolla, Simplans, Slickytail, Stephen Leppik, Sudip Bhandari, SuperBiasedMan, user2314737
51	Eccezioni del Commonwealth	Juan T, TemporalWolf
52	Elenco	Adriano, Alexander, Anthony Pham, Ares, Barry, blueenvelope, Bosoneando, BusyAnt, Çağatay Uslu, caped114, Chandan Purohit, ChaoticTwist, cizixs, Daniel Porteous, Darth Kotik, deeenes, Delgan, Elazar, Ellis, Emma, evuez, exhuma, Ffisegydd, Flickerlight, Gal Dreiman, ganesh gadila, ghostarbeiter, Igor Raush, intboolstring, J F, j3485, jalanb, James, James Elderfield, jani, jimsug, jkdev, JNat, jonrsharpe, KartikKannapur, Kevin Brown, Lafexlos, LDP, Leo Thumma, Luke Taylor, lukewrites, lxer, Majid, Mechanic, MrP01, MSeifert, muddyfish, n12312, noɥʎdʌzɐɔ, Oz Bar-Shalom, Pasha, Pavan Nath, poke, RamenChef, ravigadila, ronrest, Serenity, Severiano Jaramillo Quintanar, Shawn Mehan, Simplans, sirin, solarc, SuperBiasedMan, textshell, The_Cthulhu_Kid,

		user2314737, user6457549, Utsav T, Valentin Lorentz, vaultah , Will, wythagoras, Xavier Combelle
53	Elenco delle comprensioni	3442, Akshit Soota, André Laszlo, Andy Hayden, Annonymous , Ari, Bhargav, Chris Mueller, Darth Shadow, Dartmouth, Delgan, enrico.bacis, Franck Dernoncourt, garg10may, intboolstring, Jeff Langemeier, Josh Caswell, JRodDynamite, justhalf, kdopen, Ken T, Kevin Brown, kiliantics, longyue0521, Martijn Pieters, Mattew Whitt, Moinuddin Quadri, MSeifert, muddyfish, noɥʎdʎzɐɹƆ, pktangyue, Pyth0nicPenguin, Rahul Nair, Riccardo Petraglia, SashaZd, shrishinde, Simplans, Slayther, sudo bangbang, theheadofabroom, then0rTh, Tim McNamara, Udi, Valentin Lorentz, Veedrac, Zags
54	Elenco delle sezioni (selezione di parti di elenchi)	Greg, JakeD
55	Elenco di destrutturazione (ovvero imballaggio e disimballaggio)	J F, sth, zmo
56	elevamento a potenza	Anthony Pham, intboolstring, jtbandes, Luke Taylor, MSeifert, Pasha, supersam654
57	enum	Andy, Elazar, evuez, Martijn Pieters, techydesigner
58	Esecuzione di codice dinamico con `exec` e` eval`	Antti Haapala, Ilja Everilä
59	Espressioni regolari (Regex)	Aidan, alejosocorro, andandandand, Andy Hayden, ashes999, B8vrede, Claudiu, Darth Shadow, driax, Fermi paradox, ganesh gadila, goodmami, Jan, Jeffrey Lin, jonrsharpe, Julien Spronck, Kevin Brown, Md.Sifatul Islam, Michael M., mnoronha , Nander Speerstra, nrusch, Or East, orvi, regnarg, sarvajeetsuman, Simplans, SN Ravichandran KR, SuperBiasedMan, user2314737, zondo
60	Eventi inviati dal server Python	Nick Humrich
61	File e cartelle I / O	Ajean, Anthony Pham, avb, Benjamin Hodgson, Bharel, Charles, crhodes, David Cullen, Dov, Esteis, ilse2005, isvforall, jfsturtz, Justin, Kevin Brown, mattgathu, MSeifert, nlsdfnbch, Ozair Kafray, PYPL, pzp, RamenChef, Ronen Ness, rrao, Serenity, Simplans, SuperBiasedMan, Tasdik Rahman, Thomas Gerot, Umibozu, user2314737, Will, WombatPM,

		xgord
62	Filtro	APerson, cfi, J Atkin, MSeifert, rajah9, SuperBiasedMan
63	Formattazione della data	surfthecity
64	Formattazione di stringhe	4444, Aaron Christiansen, Adam_92, ADITYA, Akshit Soota, aldanor, alecxe, Alessandro Trinca Tornidor, Andy Hayden, Ani Menon, B8vrede, Bahrom, Bhargav, Charles, Chris, Darth Shadow, Dartmouth, Dave J, Delgan, dreftymac, evuez, Franck Dernoncourt, Gal Dreiman, gerrit, Giannis Spiliopoulos, GiantsLoveDeathMetal, goyalankit, Harrison, James Elderfield, Jean-Francois T., Jeffrey Lin, jetpack_guy, JL Peyret, joel3000, Jonatan, JRodDynamite, Justin, Kevin Brown, knight, krato, Marco Pashkov, Mark, Matt, Matt Giltaji, mu , MYGz, Nander Speerstra, Nathan Arthur, Nour Chawich, orion_tvv, ragesz, SashaZd, Serenity, serv-inc, Simplans, Slayther, Sometowngeek, SuperBiasedMan, Thomas Gerot, tobias_k, Tony Suffolk 66, UloPe, user2314737, user312016, Vin, zondo
65	Funzione mappa	APerson, cfi, Igor Raush, Jon Ericson, Karl Knechtel, Marco Pashkov, MSeifert, noɥʎɑʎzɐɹƆ, Parousia, Simplans, SuperBiasedMan, tlama, user2314737
66	funzioni	Adriano, Akshat Mahajan, AlexV, Andy, Andy Hayden, Anthony Pham, Arkady, B8vrede, Benjamin Hodgson, btel, CamelBackNotation, Camsbury, Chandan Purohit, ChaoticTwist, Charlie H, Chris Larson, Community, D. Alveno, danidee, DawnPaladin, Delgan, duan, duckman_1991, elegent , Elodin, Emma, EsmaeelE, Ffisegydd, Gal Dreiman, ghostarbeiter, Hurkyl, J F, James, Jeffrey Lin, JGreenwell, Jim Fasarakis Hilliard, jkitchen, Jossie Calderon, Justin, Kevin Brown, L3viathan, Lee Netherton, Martijn Pieters, Martin Thurau, Matt Giltaji, Mike - SMT, Mike Driscoll, MSeifert, muddyfish, Murphy4, nd., noɥʎɑʎzɐɹƆ, Pasha, pylang, pzp, Rahul Nair, Severiano Jaramillo Quintanar, Simplans, Slayther, Steve Barnes, Steven Maude, SuperBiasedMan, textshell, then0rTh, Thomas Gerot, user2314737, user3333708, user405 , Utsav T, vaultah, Veedrac, Will, Will, zxxz, λuser
67	Funzioni parziali	FrankBr
68	generatori	2Cubed, Ahsanul Haque, Akshat Mahajan, Andy Hayden, Arthur Dent, ArtOfCode, Augustin, Barry, Chankey Pathak, Claudiu, CodenameLambda, Community, deeenes, Delgan, Devesh Saini, Elazar, ericmarkmartin, Ernir, ForceBru, Igor Raush, Ilia Barahovski, J0HN, jackskis, Jim Fasarakis Hilliard,

		Juan T, Julius Bullinger, Karl Knechtel, Kevin Brown, Kronen, Luc M, Lyndsy Simon, machine yearning, Martijn Pieters, Matt Giltaji, max, MSeifert, nlsdfnbch, Pasha, Pedro, PsyKzz, pzp, satsumas, sevenforce, Signal, Simplans, Slayther, StuxCrystal, tversteeg, Valentin Lorentz, Will, William Merrill, xtreak, Zaid Ajaj, zarak, λuser
69	Gestori di contesto ("con" istruzione)	Abhijeet Kasurde, Alessandro Trinca Tornidor, Andy Hayden, Antoine Bolvy, carrdelling, Conrad.Dean, Dartmouth, David Marx, DeepSpace, Elazar, Kevin Brown, magu_, Majid, Martijn Pieters, Matthew, nlsdfnbch, Pasha, Peter Brittain, petrs, Shuo, Simplans, SuperBiasedMan, The_Cthulhu_Kid, Thomas Gerot, tyteen4a03, user312016, Valentin Lorentz, vaultah, λuser
70	Grafica tartaruga	Luca Van Oort, Stephen Leppik
71	grafico-utensile	xiaoyi
72	hashlib	Mark Omo, xiaoyi
73	Heapq	ettanany
74	idiomi	Benjamin Hodgson, Elazar, Faiz Halde, J F, Lee Netherton, loading..., Mister Mister
75	ijson	Prem Narain
76	Il modulo base64	Thomas Gerot
77	Il modulo dis	muddyfish, user2314737
78	Il modulo locale	Will, XonAether
79	Il modulo os	Andy, Christian Ternus, JelmerS, JL Peyret, mnoronha, Vinzee
80	Implementazioni Python non ufficiali	Jacques de Hooge, Squidward
81	Importazione di moduli	angussidney, Anthony Pham, Antonis Kalou, Brett Cannon, BusyAnt, Casebash, Christian Ternus, Community, Conrad.Dean, Daniel, Dartmouth, Esteis, Ffisegydd, FMc, Gerard Roche, Gideon Buckwalter, J F, JGreenwell, Kinifwyne, languitar, Lex Scarisbrick, Matt Giltaji, MSeifert, niyasc, nlsdfnbch, Paulo Freitas, pylang, Rahul Nair, Saiful Azad, Serenity, Simplans, StardustGogeta, StuxCrystal, SuperBiasedMan, techydesigner, the_cat_lady, Thomas Gerot, Tony Meyer, Tushortz, user2683246, Valentin Lorentz, Valor Naram, vaultah, wnnmaw
82	Impostato	Andrzej Pronobis, Andy Hayden, Bahrom, Cimbali, Cody

		Piersall, Conrad.Dean, Elazar, evuez, J F, James, Or East, pylang, RahulHP, RamenChef, Simplans, user2314737
83	Incompatibilità che si spostano da Python 2 a Python 3	671620616, Abhishek Kumar, Akshit Soota, Alex Gaynor, Allan Burleson, Alleo, Amarpreet Singh, Andy Hayden, Ani Menon, Antoine Bolvy, AntsySysHack, Antti Haapala, Antwan, arekolek , Ares, asmeurer, B8vrede, Bakuriu, Bharel, Bhargav Rao, bignose, bitchaser, Bluethon, Cache Staheli, Cameron Gagnon , Charles, Charlie H, Chris Sprague, Claudiu, Clayton Wahlstrom, cʟdseed, Colin Yang, Cometsong, Community, Conrad.Dean, danidee, Daniel Stradowski, Darth Shadow, Dartmouth, Dave J, David Cullen, David Heyman, deeenes, DeepSpace, Delgan, DoHe, Duh-Wayne-101, Dunno, dwanderson, Ekeyme Mo, Elazar, enderland, enrico.bacis, erewok, ericdwang, ericmarkmartin, Ernir, ettanany, Everyone_Else, evuez, Franck Dernoncourt, Fred Barclay, garg10may, Gavin, geoffspear, ghostarbeiter, GoatsWearHats, H. Pauwelyn, Haohu Shen, holdenweb, iScrE4m, Iván C., J F, J. C. Leitão, James Elderfield, James Thiele, jarondl, jedwards, Jeffrey Lin, JGreenwell, Jim Fasarakis Hilliard, Jimmy Song, John Slegers, Jojodmo, jonrsharpe, Josh, Juan T, Justin, Justin M. Ucar, Kabie, kamalbanga, Karl Knechtel, Kevin Brown, King's jester, Kunal Marwaha, Lafexlos, lenz, linkdd, l'L'l, Mahdi, Martijn Pieters, Martin Thoma, masnun, Matt, Matt Dodge, Matt Rowland, Mattew Whitt, Max Feng, mgwilliams, Michael Recachinas, mkj, mnoronha, Moinuddin Quadri, muddyfish, Nathaniel Ford, niemmi, niyasc, noɥʎdʎzɐɹƆ, OrangeTux, Pasha, Paul Weaver, Paulo Freitas, pcurry, pktangyue, poppie, pylang, python273, Pythonista, RahulHP, Rakitić, RamenChef, Rauf, René G, rfkortekaas, rrao, Ryan, sblair, Scott Mermelstein, Selcuk, Serenity, Seth M. Larson, ShadowRanger, Simplans, Slayther, solarc, sricharan, Steven Hewitt, sth, SuperBiasedMan, Tadhg McDonald-Jensen, techydesigner, Thomas Gerot, Tim, tobias_k, Tyler, tyteen4a03 , user2314737, user312016, Valentin Lorentz, Veedrac, Ven, Vinayak, Vlad Shcherbina, VPfB, WeizhongTu, Wieland, wim, Wolf, Wombatz, xtreak, zarak, zcb, zopieux, zurfyx, zvezda
84	Indicizzazione e affettatura	Alleo, amblina, Antoine Bolvy, Bonifacio2, Ffisegydd, Guy, Igor Raush, Jonatan, Martec, MSeifert, MUSR, pzp, RahulHP, Reut Sharabani, SashaZd, Sayed M Ahamad, SuperBiasedMan, theheadofabroom, user2314737, yurib
85	iniziare con GZip	orvi
86	Input e output di base	Doraemon, GoatsWearHats, J F, JNat, Marco Pashkov, Mark Miller, Martijn Pieters, Nathaniel Ford, Nicolás, pcurry, pzp,

		SashaZd, SuperBiasedMan, Vilmar
87	Input, Subset e Output File di dati esterni che utilizzano Panda	Mark Miller
88	Insidie comuni	abukaj, ADITYA, Alec, Alessandro Trinca Tornidor, Alex, Antoine Bolvy, Baaing Cow, Bhargav Rao, Billy, bixel, Charles, Cheney, Christophe Roussy, Dartmouth, DeepSpace, DhiaTN, Dilettant, fox, Fred Barclay, Gerard Roche, greatwolf, hiro protagonist, Jeffrey Lin, JGreenwell, Jim Fasarakis Hilliard, Lafexlos, maazza, Malt, Mark, matsjoyce, Matt Dodge, MervS, MSeifert, ncmathsadist, omgimanerd, Patrick Haugh, pylang, RamenChef, Reut Sharabani, Rob Bednark, rrao, SashaZd, Shihab Shahriar, Simplans, SuperBiasedMan, Tim D, Tom Dunbavan, tyteen4a03, user2314737, Will Vousden, Wombatz
89	Interfaccia Web Server Gateway (WSGI)	David Heyman, Kevin Brown, Preston, techydesigner
90	Introduzione a RabbitMQ usando AMQPStorm	eandersson
91	Iterables e Iterators	4444, Conrad.Dean, demonplus, Ilia Barahovski, Pythonista
92	kivy - Framework Python multipiattaforma per lo sviluppo NUI	dhimanta
93	La dichiarazione del passaggio	Anaphory
94	La funzione di stampa	Beall619, Frustrated, Justin, Leon Z., lukewrites, SuperBiasedMan, Valentin Lorentz
95	La variabile speciale __name__	Annonymous, BusyAnt, Christian Ternus, jonrsharpe, Lutz Prechelt, Steven Elliott
96	Lavorare attorno al Global Interpreter Lock (GIL)	Scott Mermelstein
97	Lavorare con gli archivi ZIP	Chinmay Hegde, ghostarbeiter, Jeffrey Lin, SuperBiasedMan

98	Lettura e scrittura CSV	Adam Matan, Franck Dernoncourt, Martin Valgur, mnoronha, ravigadila, Setu
99	Libreria di sottoprocesso	Adam Matan, Andrew Schade, Brendan Abel, jfs, jmunsch, Riccardo Petraglia
100	Liste collegate	Nemo
101	Loops	Adriano, Alex L, alfonso.kim, Alleo, Anthony Pham, Antti Haapala, Chris Hunt, Christian Ternus, Darth Kotik, DeepSpace, Delgan, DhiaTN, ebo, Elazar, Eric Finn, Felix D., Ffisegydd, Gal Dreiman, Generic Snake, ghostarbeiter, GoatsWearHats, Guy, Inbar Rose, intboolstring, J F, James, Jeffrey Lin, JGreenwell, Jim Fasarakis Hilliard, jrast, Karl Knechtel, machine yearning, Mahdi, manetsus, Martijn Pieters, Math, Mathias711, MSeifert, pnhgiol, rajah9, Rishabh Gupta, Ryan, sarvajeetsuman, sevenforce, SiggyF, Simplans, skrrgwasme, SuperBiasedMan, textshell, The_Curry_Man, Thomas Gerot, Tom, Tony Suffolk 66, user1349663, user2314737, Vinzee, Will
102	Maledizioni di base con Python	4444, Guy, kollery, Vinzee
103	Manipolazione di XML	4444, Brad Larson, Chinmay Hegde, Francisco Guimaraes, greuze, heyhey2k, Rob Murray
104	Matematica complessa	Adeel Ansari, Bosoneando, bpachev
105	Matrici multidimensionali	boboquack, Buzz, rrao
106	metaclassi	2Cubed, Amir Rachum, Antoine Pinsard, Camsbury, Community, driax, Igor Raush, InitializeSahib, Marco Pashkov, Martijn Pieters, Mattew Whitt, OozeMeister, Pasha, Paulo Scardine, RamenChef, Rob Bednark, Simplans, sisanared, zvone
107	Metodi definiti dall'utente	Alessandro Trinca Tornidor, Beall619, mnoronha, RamenChef, Stephen Leppik, Sun Qingyao
108	Metodi di stringa	Amitay Stern, Andy Hayden, Ares, Bhargav Rao, Brien, BusyAnt, Cache Staheli, caped114, ChaoticTwist, Charles, Dartmouth, David Heyman, depperm, Doug Henderson, Elazar , ganesh gadila, ghostarbeiter, GoatsWearHats, idjaw, Igor Raush, Ilia Barahovski, j__, Jim Fasarakis Hilliard, JL Peyret, Kevin Brown, krato, MarkyPython, Metasomatism, Mikail Land, MSeifert, mu , Nathaniel Ford, OliPro007, orvi, pzp, ronrest,

		Shrey Gupta, Simplans, SuperBiasedMan, theheadofabroom, user1349663, user2314737, Veedrac, WeizhongTu, wnnmaw
109	Metodo Overriding	DeepSpace, James
110	mixins	Doc, Rahul Nair, SashaZd
111	Modelli di progettazione	Charul, denvaar, djaszczurowski
112	Modelli in python	4444, Alessandro Trinca Tornidor, Fred Barclay, RamenChef, Ricardo, Stephen Leppik
113	Modulo Asyncio	2Cubed, Alessandro Trinca Tornidor, Cimbali, hiro protagonist, obust, pylang, RamenChef, Seth M. Larson, Simplans, Stephen Leppik, Udi
114	Modulo casuale	Alex Gaynor, Andrzej Pronobis, Anthony Pham, Community, David Robinson, Delgan, giucal, Jim Fasarakis Hilliard, michaelrbock, MSeifert, Nobilis, ppperry, RamenChef, Simplans, SuperBiasedMan
115	Modulo collezioni	asmeurer, Community, Elazar, jmunsch, kon psych, Marco Pashkov, MSeifert, RamenChef, Shawn Mehan, Simplans, Steven Maude, Symmitchry, void, XCoder Real
116	Modulo Deque	Anthony Pham, BusyAnt, matsjoyce, ravigadila, Simplans, Thomas Ahle, user2314737
117	Modulo di coda	Prem Narain
118	Modulo Functools	Alessandro Trinca Tornidor, enrico.bacis, flamenco, RamenChef, Shrey Gupta, Simplans, Stephen Leppik, StuxCrystal
119	Modulo Itertools	ADITYA, Alessandro Trinca Tornidor, Andy Hayden, balki, bpachev, Ffisegydd, jackskis, Julien Spronck, Kevin Brown, machine yearning, nlsdfnbch, pylang, RahulHP, RamenChef, Simplans, Stephen Leppik, Symmitchry, Wickramaranga, wnnmaw
120	Modulo JSON	Indradhanush Gupta, Leo, Martijn Pieters, pzp, theheadofabroom, Underyx, Wolfgang
121	Modulo matematico	Anthony Pham, ArtOfCode, asmeurer, Christofer Ohlsson, Ellis , fredley, ghostarbeiter, Igor Raush, intboolstring, J F, James Elderfield, JGreenwell, MSeifert, niyasc, RahulHP, rajah9, Simplans, StardustGogeta, SuperBiasedMan, yurib
122	Modulo operatore	MSeifert

123	modulo pyautogui	Damien, Rednivrug
124	Modulo Sqlite3	Chinmay Hegde, Simplans
125	Modulo Webbrowser	Thomas Gerot
126	multiprocessing	Alon Alexander, Nander Speerstra, unutbu, Vinzee, Will
127	multithreading	Alu, cldseed, juggernaut, Kevin Brown, Kristof, mattgathu, Nabeel Ahmed, nlsdfnbch, Rahul, Rahul Nair, Riccardo Petraglia, Thomas Gerot, Will, Yogendra Sharma
128	Mutevole vs Immutabile (e Lavabile) in Python	Cilyan
129	Neo4j e Cypher utilizzano Py2Neo	Wingston Sharon
130	Nodo di elenco collegato	orvi
131	Oggetti di proprietà	Alessandro Trinca Tornidor, Darth Shadow, DhiaTN, J F, Jacques de Hooge, Leo, Martijn Pieters, mnoronha, Priya, RamenChef, Stephen Leppik
132	Operatori bit a bit	Abhishek Jain, boboquack, Charles, Gal Dreiman, intboolstring , JakeD, JNat, Kevin Brown, Matías Brignone, nemesisfixx, poke, R Colmenares, Shawn Mehan, Simplans, Thomas Gerot, tmr232, Tony Suffolk 66, viveksyngh
133	Operatori booleani	boboquack, Brett Cannon, Dair, Ffisegydd, John Zwinck, Severiano Jaramillo Quintanar, Steven Maude
134	Ordinamento, minimo e massimo	Antti Haapala, APerson, GoatsWearHats, Mirec Miskuf, MSeifert, RamenChef, Simplans, Valentin Lorentz
135	os.path	Claudiu, Fábio Perez, girish946, Jmills, Szabolcs Dombi, VJ Magar
136	Ottimizzazione delle prestazioni	A. Ciclet, RamenChef, user2314737
137	Pallone	Stephen Leppik, Thomas Gerot
138	Persistenza di Python	RamenChef, user2728397
139	Pila	ADITYA, boboquack, Chromium, cjds, depperm, Hannes Karppila, JGreenwell, Jonatan, kdopen, OliPro007, orvi,

		SashaZd, Snađoшfa, textshell, Thomas Ahle, user2314737
140	pip: PyPI Package Manager	Andy, Arpit Solanki, Community, InitializeSahib, JNat, Mahdi, Majid, Matt Giltaji, Nathaniel Ford, Rápli András, SerialDev, Simplans, Steve Barnes, StuxCrystal, tlo
141	Plugin ed Extension Classes	2Cubed, proprefenetre, pylang, rrao, Simon Hibbs, Simplans
142	Polimorfismo	Benedict Bunting, DeepSpace, depperm, Simplans, skrrgwasme, Vinzee
143	PostgreSQL	Alessandro Trinca Tornidor, RamenChef, Stephen Leppik, user2027202827
144	Precedenza dell'operatore	HoverHell, JGreenwell, MathSquared, SashaZd, Shreyash S Sarnayak
145	Processi e discussioni	Claudiu, Thomas Gerot
146	profiling	J F, keiv.fly, SashaZd
147	Programmazione funzionale in Python	Imran Bughio, mvis89, Rednivrug
148	Programmazione IoT con Python e Raspberry PI	dhimanta
149	py.test	Andy, Claudiu, Ffisegydd, Kinifwyne, Matt Giltaji
150	pyaudio	Biswa_9937
151	pygame	Anthony Pham, Aryaman Arora, Pavan Nath
152	pyglet	Comrade SparklePony, Stephen Leppik
153	PyInstaller - Distribuzione del codice Python	ChaoticTwist, Eric, mnoronha
154	Python Anti-Patterns	Alessandro Trinca Tornidor, Annonymous, eenblam, Mahmoud Hashemi, RamenChef, Stephen Leppik
155	Python ed Excel	bee-sting, Chinmay Hegde, GiantsLoveDeathMetal, hackvan, Majid, talhasch, user2314737, Will
156	Python HTTP Server	Arpit Solanki, J F, jmunsch, Justin Chadwell, Mark, MervS, orvi , quantummind, Raghav, RamenChef, Sachin Kalkur, Simplans

		, techydesigner
157	Python Lex-Yacc	cldseed
158	Python Networking	atayenel, ChaoticTwist, David, Geeklhem, mattgathu, mnoronha, thsecmaniac
159	Python Serial Communication (pyserial)	Alessandro Trinca Tornidor, Ani Menon, girish946, mnoronha, Saranjith, user2314737
160	Python velocità del programma	ADITYA, Antonio, Elodin, Neil A., Vinzee
161	Python Virtual Environment - virtualenv	Vikash Kumar Jain
162	Raccolta dei rifiuti	bogdanciobanu, Claudiu, Conrad.Dean, Elazar, FazeL, J F, James Elderfield, lukess, muddyfish, Sam Whited, SiggyF, Stephen Leppik, SuperBiasedMan, Xavier Combelle
163	raggruppa per()	Parousia, Thomas Gerot
164	Rappresentazioni di stringhe di istanze di classe: metodi __str__ e __repr__	Alessandro Trinca Tornidor, jedwards, JelmerS, RamenChef, Stephen Leppik
165	Raspare Web con Python	alecxe, Amitay Stern, jmunsch, mrtuovinen, Ni., RamenChef, Saiful Azad, Saqib Shamsi, Simplans, Steven Maude, sth, sytech, talhasch, Thomas Gerot
166	Registrazione	Gal Dreiman, Jörn Hees, sxnwlfkk
167	Ricerca	Dan Sanderson, Igor Raush, MSeifert
168	Richieste di richieste Python	Ken Y-N, RandomHash
169	Riconoscimento ottico dei caratteri	rassar
170	ricorsione	Bastian, japborst, JGreenwell, Jossie Calderon, mbomb007, SashaZd, Tyler Crompton
171	Ridurre	APerson, Igor Raush, Martijn Pieters, MSeifert
172	Scrittura in formato CSV da stringa o	Hriddhi Dey, Thomas Crowley

	elenco	
173	Scrivere estensioni	Dartmouth, J F, mattgathu, Nathan Osman, techydesigner, ygram
174	Secure Shell Connection in Python	mnoronha, Shijo
175	Semplici operatori matematici	amin, blueenvelope, Bryce Frank, Camsbury, David, DeepSpace, Elazar, J F, James, JGreenwell, Jon Ericson, Kevin Brown, Lafexlos, matsjoyce, Mechanic, Milo P, MSeifert, numbermaniac, sarvajeetsuman, Simplans, techydesigner, Tony Suffolk 66, Undo, user2314737, wythagoras, Zenadix
176	Serializzazione dei dati	Devesh Saini, Infinity, rfkortekaas
177	Serializzazione dei dati sottaceti	J F, Majid, Or East, RahulHP, rfkortekaas, zvone
178	setup.py	Adam Brenecki, amblina, JNat, ravigadila, strpeter, user2027202827, Y0da
179	Sicurezza e crittografia	adeora, ArtOfCode, BSL-5, Kevin Brown, matsjoyce, SuperBiasedMan, Thomas Gerot, Wladimir Palant, wrwrwr
180	Socket e crittografia / decrittografia dei messaggi tra client e server	Mohammad Julfikar
181	Sockets	David Cullen, Dev, MattCorr, nlsdfnbch, Rob H, StuxCrystal, textshell, Thomas Gerot, Will
182	Somiglianze nella sintassi, differenze di significato: Python vs. JavaScript	user2683246
183	Sottocomandi CLI con output di aiuto preciso	Alessandro Trinca Tornidor, anatoly techtonik, Darth Shadow
184	Sovraccarico	Andy Hayden, Darth Shadow, ericmarkmartin, Ffisegydd, Igor Raush, Jonas S, jonrsharpe, L3viathan, Majid, RamenChef, Simplans, Valentin Lorentz
185	Strumento 2to3	Alessandro Trinca Tornidor, Dartmouth, Firix, Kevin Brown, Naga2Raja, Stephen Leppik

186	SYS	blubberdiblub
187	tempfile NamedTemporaryFile	Alessandro Trinca Tornidor, amblina, Kevin Brown, Stephen Leppik
188	Test unitario	Alireza Savand, Ami Tavory, antimatter15, Arpit Solanki, bijancn, Claudiu, Dartmouth, engineercoding, Ffisegydd, J F, JGreenwell, jmunsch, joel3000, Kevin Brown, Kinifwyne, Mario Corchero, Matt Giltaji, Mattew Whitt, mgilson, muddyfish, pylang, strpeter
189	The Interpreter (Command Line Console)	Aaron Christiansen, David, Elazar, Peter Shinners, ppperry
190	Tipi di dati immutabili (int, float, str, tuple e frozensets)	Alessandro Trinca Tornidor, FazeL, Ganesh K, RamenChef, Stephen Leppik
191	Tipi di dati Python	Gavin, lorenzofeliz, Pike D., Rednivrug
192	Tipo Suggerimenti	alecxe, Annonymous, Antti Haapala, Elazar, Jim Fasarakis Hilliard, Jonatan, RamenChef, Seth M. Larson, Simplans, Stephen Leppik
193	Tkinter	Dartmouth, rlee827, Thomas Gerot, TidB
194	Tracciare con Matplotlib	Arun, user2314737
195	Trasformazione di Panda: operazioni preforme su gruppi e concatenare i risultati	Dee
196	tuple	Anthony Pham, Antoine Bolvy, BusyAnt, Community, Elazar, James, Jim Fasarakis Hilliard, Joab Mendes, Majid, Md.Sifatul Islam, Mechanic, mezzode, nlsdfnbch, noɥʎdʎzɐɹƆ, Selcuk, Simplans, textshell, tobias_k, Tony Suffolk 66, user2314737
197	Unicode	wim
198	Unicode e byte	Claudiu, KeyWeeUsr
199	Unzipping dei file	andrew
200	urllib	Amitay Stern, ravigadila, sth, Will
201	Utilizzo dei loop all'interno delle	naren

	funzioni	
202	Utilizzo del modulo "pip": PyPI Package Manager	Zydnar
203	Verifica dell'esistenza e delle autorizzazioni del percorso	Esteis, Marlon Abeykoon, mnoronha, PYPL
204	Visualizzazione dei dati con Python	Aquib Javed Khan, Arun, ChaoticTwist, cledoux, Ffisegydd, ifma
205	WebSockets	2Cubed, Stephen Leppik, Tyler Gubala