Бесплатная электронная книга

УЧУСЬ Python Language

Free unaffiliated eBook created from **Stack Overflow contributors.**

#python

1
1: Python 2
2
3
Python 3.x 3
Python 2.x 3
Examples 4
4
, Python 4
, Python, IDLE 5
Hello World Python 6
Python 6
- 7
8
8
9
14
IDLE - Python 15
15
16
16
16
17
18
18
19
19
20
21
21

22

26

27

28

33

34

- str () repr () 35

() 35

() 36

pip 36

/ 37

37

38

Python 2.7.x 3.x 38

2: * args ** kwargs 42

42

h11 42

h12 42

h13 42

Examples 43

* args 43

** kwargs 43

* args 44

** kwargs 45

* args 45

46

kwarg 46

** kwargs 46

3: ArcPy 48

48

Examples 48

48

createDissolvedGDB gdb 48

4: ChemPy - python 49

..... 49

Examples 49

..... 49

..... 49

..... 49

..... 50

..... 50

( ) 50

5: Conditionals 52

..... 52

..... 52

Examples 52

if, elif, ... 52

(« ») 52

..... 53

Else statement 53

..... 54

..... 54

..... 54

..... 55

..... 55

..... 56

cmp, ... 57

List List 57

, None 58

6: ConfigParser 60

..... 60

..... 60

.....

60

Examples 60

.... 60

.... 61

7: ctypes **62**

.... 62

Examples 62

.... 62

.... 62

.... **62**

.... **63**

ctypes 63

ctypes 64

ctypes 65

.... 65

8: Enum **67**

.... 67

Examples 67

(Python 2.4 3.3) 67

.... 67

9: hashlib **68**

.... 68

Examples 68

MD5- 68

, OpenSSL 69

10: Heapq **70**

Examples 70

.... 70

.... 70

11: HTML- **72**

Examples 72

BeautifulSoup 72
CSS BeautifulSoup 72
PyQuery 73
12: ijson 75
75
Examples 75
75
13: kivy - Python NUI 76
76
Examples 76
76
14: Loops 79
79
79
79
Examples 79
79
81
81
81
break 81
continue 82
83
return break 83
«else» 84
? 86
86
87
88
89
89

90

«Half loop» do-while......90

......91

15: Mutable vs Immutable (Hashable) Python......92

Examples......92

Mutable vs Immutable......92

Immutables......92

......93

Mutables......93

......94

Mutable and Immutable......94

......95

16: Neo4j Cypher Py2Neo......97

Examples......97

......97

Neo4j Graph......97

Neo4j......97

1:......98

2:......98

Cypher......98

17: os.path......100

......100

......100

Examples......100

......100

......100

......101

......101

......101

, , , ,......101

18: Pandas Transform:......103

Examples......103

.... 103
.... 103
transform pandas 103
.... 104
transform , ... 104
19: pip: PyPI 106
.... 106
.... 106
.... 106
Examples 107
.... 107
.... 107
.... 107
, `pip` 108
.... 108
Linux 108
Windows 109
requirements.txt 109
requirements.txt virtualenv 109
Python pip 109
, ... 110
.... 112
.... 112
20: PostgreSQL 115
Examples 115
.... 115
.... 115
.... 115
21: py.test 117
Examples 117
py.test 117
....

117

........117

........117

........118

........118

py.test ! 120

. ...121

22: pyaudio **123**

........123

........123

Examples 123

........123

Audio I / O 124

23: Pygame **126**

........126

........126

........126

Examples 126

pygame 126

Pygame 127

........**127**

........**127**

........**127**

24: Pyglet **129**

........129

Examples 129

, Pyglet 129

Pyglet 129

Pyglet 129

Pyglet OpenGL 129

Pyglet OpenGL 130

25: PyInstaller - Python 131
..... 131
..... 131
Examples 131
..... 131
Pyinstaller 132
..... 132
: 132
..... 133
..... 133
26: Python Anti-Patterns 134
Examples 134
, ... 134
, ... 135
..... 136
27: Python HTTP Server 137
Examples 137
HTTP- 137
..... 137
API SimpleHTTPServer 139
GET, POST, PUT BaseHTTPRequestHandler 140
28: Python Lex-Yacc 142
..... 142
..... 142
Examples 142
PLY 142
«, !» PLY - 142
1: Lex 144
..... 145
h21 146
h22 146

h23 147
h24 147
h25 147
h26 147
h27 148
h28 148
h29 148
h210 149
2: Yacc 149
..... 150
h211 151
29: Python Excel 152
Examples 152
Excel 152
OpenPyXL 152
excel xlsxwriter 153
excel xlrd 155
Excel xlsxwriter 156
30: setup.py 158
..... 158
..... 158
Examples 158
setup.py 158
python 159
setup.py 160
..... 160
31: tempfile NamedTemporaryFile 162
..... 162
Examples 162
( ) , ... 162
32: Tkinter 164
.....

164

.....164

Examples.....164

tkinter.....164

.....165

.....165

.....166

.....166

33: Unicode.....168

Examples.....168

.....168

34: URLLIB.....169

Examples.....169

HTTP GET.....169

Python 2.....169

Python 3.....169

HTTP POST.....170

Python 2.....170

Python 3.....170

.....170

35: WebSockets.....172

Examples.....172

aiohttp.....172

Wrapper aiohttp.....172

Autobahn -.....173

36: 176

Examples.....176

python.....176

37: (abc).....178

Examples.....178

ABCMeta.....178

/ ABCMeta @abstractmethod 179

38: 181

.... 181

Examples 181

, : if / else. 181

.... 182

.... 182

.... 183

39: 185

Examples 185

Coroutine Delegation 185

.... 186

UVLoop 187

: 188

.... 188

.... 188

- 189

asyncio 189

40: 191

Examples 191

.... 191

WAV 191

winsound 191

.... 191

python ffmpeg 192

Windows 192

41: 193

.... 193

.... 193

.... 193

Examples 193

....

193

. 194

. 194

. 195

pycrypto 195

RSA pycrypto 196

RSA pycrypto 197

42: 199

. 199

. 199

Examples 199

. 199

Popen 200

. 200

. 200

. 200

. 200

. 200

. 201

. 201

43: 203

Examples 203

. 203

. 203

. 204

. 204

`and` ` or` 205

. 205

44: RabbitMQ AMQPStorm 206

. 206

Examples 206

RabbitMQ 206

RabbitMQ 207
RabbitMQ 208
45: , Pandas 211
.... 211
Examples 211
, Pan 211
46: 213
Examples 213
.... 213
.... 213
? 214
.... 214
47: - Python 216
.... 216
.... 216
Python - ( ) 216
.... 216
requests 216
requests-cache 216
scrapy 216
selenium 216
HTML 217
BeautifulSoup 217
lxml 217
Examples 217
lxml 217
- 217
Scrapy 218
Scrapy 218
BeautifulSoup4 219
Selenium WebDriver 219
- urllib.request 220
....

220
48: Python 222
Examples 222
Matplotlib 222
.... 223
MayaVi 226
Plotly 227
49: Python - virtualenv 230
.... 230
Examples 230
.... 230
.... 230
Virtualenv 231
virtualenv 231
50: virtualenvwrapper 232
.... 232
Examples 232
virtualenvwrapper 232
51: 234
.... 234
.... 234
Examples 234
.... 234
virtualenv 234
.... 234
.... 235
.... 235
.... 236
.... 236
.... 237
.... 237

python 238
virtualenvwrapper 239
239
239
240
, 241
python Unix / Linux 241
virtualenv 241
Anaconda 242
242
243
243
243
243
52: 244
244
Examples 244
: math.sqrt () cmath.sqrt 244
: ** pow () 245
: math.pow () 245
: math.exp () cmath.exp () 246
1: math.expm1 () 246
: , cmath 247
: pow () 249
: n- 249
250
53: 251
251
251
Examples 253
254
254

del 255
..... 256
..... 257
..... 257
..... 257
, ... 259
54: Python C # 260
..... 260
..... 260
Examples 261
Python, C # 261
C #, Python 262
55: `exec` ` eval` 264
..... 264
..... 264
..... 264
Examples 265
exec 265
eval 265
..... 265
eval 265
, Python ast.literal_eval 266
, exec, eval ast.li 266
56: 268
..... 268
..... 268
Examples 268
..... 268
next () 268
..... 269
..... 270
..... 270
.....

273

....273

- 274

....275

....275

: 276

....277

....277

....278

57: - 279

....279

Examples....279

PyDotPlus....279

....279

PyGraphviz....280

58: () 282

....282

....282

....282

....282

Examples....282

1....282

2....284

3....284

4....285

59: 287

....287

Examples....287

datetime, ...287

....287

datetime....288

....288

datetime, ... 289
, ... 290
( ) 292
.... 292
ISO 8601 293
.... 294
.... 294
.... 294
ISO 8601 295
.... **295**
, ... **295**
, ... **295**
60: **297**
.... 297
Examples 297
.... 297
.... 297
.... 297
61: **299**
.... 299
.... 299
.... 299
Examples 299
.... 299
.... 300
.... **301**
! **302**
.... 302
.... **303**
.... **303**
( ) 303
....

303

: 304

304

305

305

62: 307

Examples 307

307

308

63: 310

310

Examples 310

Hello World Django 310

64: 313

313

Examples 313

313

Setters, Getters & Properties 313

65: 316

316

Examples 316

MySQL MySQLdb 316

SQLite 317

SQLite: 318

318

h212 318

Connection 319

Cursor 320

SQLite Python 323

PostgreSQL psycopg2 323

323

: 324

: 324

Oracle 325

.... 326

sqlalchemy 327

66: - Python 329

Examples 329

- 329

.... 329

.... 329

, ... 329

, ... 330

.... 330

67: 332

.... 332

.... 332

Examples 332

.... 332

.... 333

.... 334

.... 335

.... 335

.... 335

.... 335

.... 336

.... 336

.... 337

.... 337

.... 337

.... 337

68: CSV 339

....

339
.... 339
.... 339
Examples 339
.... 340
CSV 340
69: 341
.... 341
.... 341
Examples 343
.... 343
, ... 343
70: 344
Examples 344
.... 344
.... 344
.... 344
«__main__», ... 345
71: 347
.... 347
.... 347
Examples 347
.... 347
.... 349
.... 349
__all__ 350
.... 351
.... 352
PEP8 352
.... 353
__import __ () 353
.... 353

Python 2 354
Python 3 354
72: **355**
.... 355
.... 355
.... 355
Examples 355
.... 355
.... 357
.... 357
: __getitem__, __setitem__ __delitem__ 357
.... 359
.... 359
.... 360
73: 2to3 .... **361**
.... 361
.... 361
.... 362
Examples 362
.... 362
.... 362
Windows 362
.... 363
Windows 363
74: ( ) **364**
Examples 364
.... 364
.... 364
Python 365
PYTHONSTARTUP 365
.... 365
.... 367

75: - (WSGI) 369
.... 369
Examples 369
() 369
76: 371
.... 371
.... 371
Examples 371
.... 371
.... 371
.... 372
.... 372
.... 373
.... 373
.... 376
.... 377
! 377
.... 378
.... 378
.... 378
.... 379
.... 379
77: 381
.... 381
Examples 381
Errors (SyntaxErrors) 381
IndentationError / SyntaxError: 381
.... 381
IndentationError / SyntaxError: unindent 382
.... 382
IndentationError: 382
....

382
IndentationError: 383
....383
....383
TypeErrors....383
Error: [/] ? , ? 383
....384
TypeError: []: '???' '???'....384
....384
TypeError: '???' / :....385
....385
TypeError: '???' 385
....385
NameError: name '???' 385
....386
, :....386
import :....386
Python LEGB:....386
....387
AssertError....387
KeyboardInterrupt....387
ZeroDivisionError....387
....388
78: «pip»: PyPI....390
....390
....390
Examples....391
....391
ImportError....391
....392
79: 393

.... 393
Examples 393
.... 393
80: 394
Examples 394
Iterable vs Generator 394
.... 395
.... 396
.... 396
.... 396
! 396
81: 398
.... 398
Examples 398
.... 398
, ... 399
.... 400
, ... 401
.... 403
.... 404
.... 405
.... 407
: 408
.... 410
.... 411
.... 412
.... 413
.... 414
Singleton 416
82: , ... 419
.... 419

Examples 419

419

83: 421

421

421

Examples 421

421

URL- 422

HTTP 422

423

Jinja Templating 424

425

URL 425

426

426

84: 427

427

427

Examples 427

, 427

docstrings 428

428

428

docstrings 429

docstrings 429

430

PEP 257 430

430

Google Python 431

85: ("with" Statement) 433

433

.... 433
.... 433
Examples 434
with 434
.... 434
.... 435
.... 436
.... 437
.... 437
86: 439
Examples 439
.... 439
.... 439
.... 439
.... 439
.... 440
87: 441
.... 441
.... 441
.... 441
Examples 441
.... 441
.... 442
- - 443
.... 443
.... 444
.... 445
Tuple 445
.... 445
.... 446
.... 446
.... 446

.....446
.....447
88: **448**
Examples.....448
Python.....448
.....449
89: XML.....**452**
.....452
Examples.....452
ElementTree.....452
XML.....452
XML-.....453
XML- iterparse (.....454
XML XPath.....454
90: **456**
.....456
.....456
Examples.....456
.....456
.....457
append ().....458
insert ().....458
python extend ().....458
, fromlist ().....458
remove ().....459
pop ().....459
, index ().....459
python reverse ().....459
buffer_info ().....460
count ().....460
tostring ().....460
python to.....460

char, fromstring () ... 460

91: ... **462**

Examples ... 462

: , , , trunc ... 462

! ... 463

, trunc ... 463

... 463

... 464

... 464

... 464

/ ... 464

, , ... 464

, ... 465

... 465

... 466

NaN (« ») ... 466

Pow ... 469

cmath ... 469

92: ... **473**

... 473

... 473

Examples ... 473

... 473

, ... 474

... 475

... **475**

Python 2 3 six ... **475**

... 475

... 476

? ... 476

... 476

, - ... 477

...

477

93: 479

Examples..479

....479

....480

94: 481

....481

Examples..481

....481

....482

....483

....484

()..484

while..485

95: 487

Examples..487

....487

....488

96: base64..489

....489

....489

....489

....491

Examples..491

Base64..492

Base32..493

16..493

ASCII85..494

Base85..495

97: Deque..496

....496

....496

.....496
Examples.....496
deque.....496
deque.....497
deque.....497
.....498
98: dis.....499
Examples.....499
.....499
- Python?.....499
.....500
99: Functools.....501
Examples.....501
.....501
total_ordering.....501
.....502
lru_cache.....502
cmp_to_key.....503
100: Itertools.....504
.....504
Examples.....504
.....504
.....505
itertools.product.....506
itertools.count.....507
itertools.takewhile.....507
itertools.dropwhile.....508
, ...509
Itertools.....509
.....510
itertools.repeat.....510
.....511
.....

511

itertools.permutations 511

101: JSON 513

513

513

513

-: 513

: 513

(-) 514

: 514

: 514

(): 515

Examples 515

JSON Python dict 515

Python dict JSON 516

516

516

`load` vs` load`, `dump` vs` dumps` 516

`json.tool` JSON 518

JSON 518

518

519

, 519

JSON- 519

102: os 521

521

521

521

Examples 521

521

521

521

.....522

(POSIX).....522

.....522

makedirs - 522

103: pyautogui.....524

.....524

Examples.....524

.....524

.....524

ScreenShot 524

104: Sqlite3.....525

Examples.....525

Sqlite3 - 525

.....525

105: Webbrowser.....527

.....527

.....527

.....527

.....528

Examples.....529

URL- 529

URL- 529

106: 531

.....531

.....531

Examples.....531

collections.Counter.....531

collections.defaultdict.....533

collections.OrderedDict.....534

collections.namedtuple.....535

collections.deque.....536

collections.ChainMap.....537

107: 539
..... 539
Examples 539
..... 539
108: 540
..... 540
Examples 540
..... 540
109: 541
Examples 541
Hello World C 541
C 542
C c ++ Boost 542
C ++ 542
110: GZip 545
..... 545
Examples 545
ZIP- GNU 545
111: (int, float, str, frozensets) 546
Examples 546
..... 546
Tuple 546
Frozenset 546
112: Python 547
Examples 547
IronPython 547
, ... 547
..... 547
Jython 547
, ... 548
..... 548

Transcrypt 548

..... 549

HTML 549

JavaScript DOM 549

JavaScript 550

Python JavaScript 550

..... 551

113: , Python 2 Python 3 552

..... 552

..... 552

Examples 553

..... 553

: Unicode 554

..... 556

..... 559

xrange 559

..... 561

..... 561

..... 563

.next () , ... 565

..... 566

..... 567

..... 567

exec Python 3 568

hasattr Python 2 569

..... 570

..... 570

..... 570

« » Python 3. 571

<> ``, ! = repr () 572

/ hex 572

cmp Python 3 573

574
() 575
filter (), map () zip () 576
/ 577
577
- 578
round () 579
() 579
() 580
, 580
- 581
long vs. int 581
Boolean Value 582
114: 583
583
Examples 583
583
584
585
586
586
del 587
del v 587
del v.name 587
del v[item] 588
del v[a:b] 588
588
? 588
? 589
590
global nonlocal (Python 3) 590
115: 592

.... 592

Examples 592

, ... 592

Mutable 595

.... 596

.... 601

int 602

.... 602

sys.argv [0] - 603

h14 603

.... 604

(GIL) 605

.... 606

.... 606

JSON 607

116: 608

.... 608

Examples 608

@property 608

@property 608

getter, setter deterter 609

.... 609

117: 612

Examples 612

.... 612

Methodcaller 612

Itemgetter 612

118: 614

Examples 614

.... 614

119: 615

.... 615

Examples 615

615

120: 618

618

Examples 618

PyTesseract 618

PyOCR 618

121: 620

Examples 620

input () raw_input () 620

620

621

621

stdin 622

623

122: Python 626

626

Examples 626

626

wrapper () 626

123: 628

628

628

: 628

628

Examples 629

629

(- , -): 629

629

Write- 630

124: 633

Examples 633

Python: _pdb_ 633
IPython ipdb 635
635
125: 637
637
Examples 637
637
Parent Children 637
C- 638
PyPar 638
126: Python 640
640
Examples 640
640
640
641
127: 644
Examples 644
/ Dunder 644
645
646
646
647
128: 651
Examples 651
651
129: 652
652
652
652
Examples 652
652

.... 653
.... 654
.... 654
.... 655
.... 656
.... 658
.... 659
.... 660
.... 661
, ... 663
.... 663
.... 664
130: 665
Examples 665
.... 665
.... 666
131: 668
.... 668
.... 668
Examples 668
.... 668
.... 668
XOR ( ) 669
.... 669
.... 670
.... 670
.... 672
132: / 674
.... 674
Examples 674
.... 674
.... 674

133: Python SQL Server 676
Examples 676
, , ... 676
134: Secure Shell Python 678
..... 678
Examples 678
ssh 678
135: CLI 679
..... 679
..... 679
Examples 679
( ) 679
argparse ( ) 680
argparse ( ) 681
136: 683
Examples 683
: collections.Counter 683
(-s): collections.Counter.most_common () 683
: list.count () tuple.count () 684
: str.count () 684
numpy 684
137: 686
Examples 686
..... 686
JPEG 686
138: 687
..... 687
Examples 687
: str.index (), str.rindex () str.find (), str.rfind () 687
..... 687
..... 688
..... 688

688

688

Dict 688

: list.index (), tuple.index () 688

() dict 689

: bisect.bisect_left () 690

690

: __contains__ __iter__ 691

139: 693

Examples 693

693

695

140: 697

Examples 697

697

698

141: Python (pyserial) 699

699

699

699

Examples 699

699

699

, 700

142: Matplotlib 701

701

Examples 701

Matplotlib 701

: , , , 702

, MATLAB 703

704

X, Y: doublex () 705

Y X twiny () 707

143: 710

710

710

Examples 710

Mixin 710

712

144: 713

713

713

Examples 714

python 714

145: 716

716

Examples 716

os.access 716

146: IoT Python PI 718

Examples 718

- 718

147: 721

721

721

721

Examples 721

721

722

722

723

725

725

726

....726

....727

....728

148: 729

Examples..729

%% timeit % timeit IPython..729

timeit ()..729

timeit..729

line_profiler 730

cProfile (Preferred Profiler)..730

149: 732

....732

Examples..732

....732

....734

....734

....735

....735

150: (GIL)..737

....737

GIL?..737

, GIL:..737

GIL..738

GIL..738

:..738

Examples..739

Multiprocessing.Pool..739

, GIL..739

Cython nogil:..740

, GIL..740

nogil ( ):..740

151: ZIP- 742
..... 742
..... 742
Examples 742
..... 742
Zipfile 742
zip- 743
..... 743
152: 745
..... 745
Examples 745
, argparse 745
docopt 746
argparse 746
argv 747
argparse 748
argparse.add_argument_group () 748
docopt docopt_dispatch 750
153: 752
Examples 752
Conda 752
154: 754
..... 754
Examples 754
..... 754
..... 754
155: 756
..... 756
Examples 756
Python ZipFile.extractall () ZIP- 756
Python TarFile.extractall () tarball 756
156: 757

Examples 757
py2app 757
cx_Freeze 758
157: ( ) 760
.... 760
.... 760
Examples 760
.... 761
.... 762
.... 762
.... 763
.... 763
.... 764
.... 765
.... 765
.... 765
.... 765
.... 765
.... 766
.... 766
.... 767
.... 767
Flags 767
.... 768
`re.finditer` 768
.... 769
158: 771
.... 771
Examples 771
1 n 771
, ... 771
....

775

776

- 777

778

159: 780

780

780

Examples 780

UDP 780

UDP 781

TCP 781

TCP Socket 782

Linux 783

160: 785

785

Examples 785

785

161: 789

789

789

789

Examples 790

JSON 790

Pickle 790

162: 792

792

792

792

792

pickle 793

Examples 793

Pickle 793
....793
....793
....793
....794
163: Python 796
....796
Examples 796
- Python 796
Http- 796
TCP- 797
UDP- 798
Simple HttpServer 798
164: 800
....800
....800
....800
Examples 800
....800
....800
....801
....801
165: Python 802
Examples 802
....802
....802
Deque 803
....804
... .804
166: 807
Examples 807
....807

167: 809

.... 809

Examples 809

.... 809

.... 810

168: 811

.... 811

Examples 811

: , ... 811

() 811

() 811

() 811

: randint, randrange, random 812

randint () 812

randrange () 812

.... 813

.... 813

: 813

.... 814

.... 815

.... 816

169: , Python 817

.... 817

Examples 817

SSE 817

Asyncio SSE 817

170: virtualenvwrapper 819

Examples 819

virtualenvwrapper 819

171: Python 821

.... 821

Examples....821

....821

PyPI....822

.pypirc....822

testpypi ()....822

....823

PyPI....823

....824

....824

....824

....824

172: Windows Python....826

....826

Examples....826

Python, ...826

- Flask 827

173: / 829

....829

....829

Examples....833

....833

....835

174: Python Requests....837

....837

Examples....837

....837

....838

....839

....839

....840

....841

175: ( ) **843**
.... 843
.... 843
Examples 843
«» 843
.... 843
.... 844
.... 844
176: , ... **846**
Examples 846
.... 846
.... 846
max, ... 847
: 847
.... **847**
.... 848
.... 848
.... 849
N N 852
177: __name__ **853**
.... 853
.... 853
Examples 853
__name__ == '__main__' 853
1 853
2 854
function_class_or_module .__ name__ 854
.... 855
178: **856**
.... 856
.... 856
.... 856

Examples 856
856
858
863
864
865
865
865
866
867
868
868
869
870
179: ( ) 871
Examples 871
871
871
871
872
872
873
873
874
875
180: 877
877
877
877
Examples 877
877
879

880

- 881

..... 882

..... 883

181: 884

..... 884

..... 884

Examples 884

..... 884

..... 885

..... 885

..... 886

..... 886

..... 887

`is` vs` == ` 887

..... 888

Common Gotcha: Python 889

182: 890

..... 890

..... 890

..... 890

Examples 890

Stack 890

..... 892

183: Python 893

..... 893

..... 893

Examples 893

Python 893

..... 894

184: 896

.....

....897
Examples....897
....897
str.casefold()....897
str.upper()....898
str.lower()....898
str.capitalize()....898
str.title()....898
str.swapcase()....898
str....898
....899
str.split(sep=None, maxsplit=-1)....899
str.rsplit(sep=None, maxsplit=-1)....900
....900
str.replace(old, new[, count]) :....900
str.format f-strings: 901
, ...902
str.count(sub[, start[, end]])....902
....903
str.startswith(prefix[, start[, end]])....903
str.endswith(prefix[, start[, end]])....904
, ...904
str.isalpha....904
str.isupper , str.islower , str.istitle....905
str.isdecimal , str.isdigit , str.isnumeric....905
str.isalnum....906
str.isspace....906
str.translate: 907
/ 907
str.strip([chars])....908
str.rstrip([chars]) str.lstrip([chars])....908
, ...908

...... 910
String 910
string.ascii_letters : 910
string.ascii_lowercase : 910
string.ascii_uppercase : 910
string.digits : 911
string.hexdigits : 911
string.octaldigits : 911
string.punctuation : 911
string.whitespace : 911
string.printable : 911
...... 912
...... 912
str Unicode 913
...... 914
185: : __str__ __repr__ 915
...... 915
...... 915
...... 915
Examples 916
...... 916
...... 917
(1) 917
(2) 918
... . 920
...... 921
, eval-round-trip __repr __ () 921
186: , : Python JavaScript 923
...... 923
Examples 923
`in` 923

187: ... 924

... 924

Examples ... 924

... 924

unittest.mock.create_autospec ... 925

unittest.TestCase ... 926

... 927

Unittests ... 928

pytest ... 929

188: ... 933

... 933

... 933

Examples ... 933

... 933

... 933

... 935

... 935

NamedTuple ... 936

... 936

189: Python ... 937

... 937

Examples ... 937

... 937

... 937

... 937

Tuple ... 938

... 938

... 938

190: ... 940

... 940

... 940

... 940

Examples 940

940

dict () 941

KeyError 941

942

943

943

943

944

: dict() 944

944

944

945

** 945

946

Python 3.5+ 946

Python 3.3+ 946

Python 2.x, 3.x 946

947

947

948

948

949

191: 951

Examples 951

Linked List python 951

192: 952

952

952

952

Examples 952

952
. . . 953
. . . 954
/ . . . 954
/ (,) . . . 954
193: I / O . . . 956
. . . 956
. . . 956
. . . 956
. . . 956
- . . . 956
Examples . . . 957
. . . 957
. . . 959
. . . 960
. . . 960
. . . 962
, . . . 962
. . . 963
() . . . 963
. . . 963
mmap . . . 964
. . . 964
, . . . 964
194: . . . 966
. . . 966
. . . 966
. . . 966
Examples . . . 966
. . . 966
. . . 967
. . . 967

: filterfalse, ifilterfalse 968

195: 970

Examples 970

970

datetime 970

datetime 970

196: 971

971

971

971

Examples 971

971

973

(f-) 974

974

Getitem Getattr 975

975

976

977

978

, 979

979

(Python 2.x) 979

(Python 3.2+) 980

: 980

197: 981

981

981

981

981

982

Examples 982

982

.... 984

.... 985

.... 985

.... 986

.... 986

.... 987

: 987

.... 987

.... 988

.... 989

.... 989

.... 989

.... 990

.... 990

.... 991

Lambda (Inline / Anonymous) 991

.... 994

.... 995

.... 996

.... 997

.... 997

.... 998

.... 1000

.... 1000

.... 1000

198: Python 1002

.... 1002

Examples 1002

- 1002

.... 1002

.... 1002

1003

199: 1004

1004

1004

1004

Examples 1004

, itertools.imap future_builtins.map 1004

1005

1006

: «» 1008

1008

200: 1011

Examples 1011

1011

1012

201: 1014

1014

1014

1014

1014

Examples 1014

1014

202: 1016

Examples 1016

Ninja Twist ( ) 1016

203: CSV 1017

Examples 1017

TSV 1017

1017

1017

1017

204: python **1018**
Examples 1018
.... 1018
.... 1018
205: **1019**
.... 1019
Examples 1019
.... 1019
Singleton 1020
.... 1022
206: **1025**
.... 1025
.... 1025
Examples 1025
.... 1025
.... **1025**
unicode **1026**
/ 1027
.... **1027**
.... **1027**
.... **1027**
- 1027
.... **1029**

Около

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: python-language

It is an unofficial and free Python Language ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Python Language.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

глава 1: Начало работы с языком Python

замечания

Python - широко используемый язык программирования. Это:

- **Высокий уровень** : Python автоматизирует операции низкого уровня, такие как управление памятью. Это оставляет программиста с меньшим контролем, но имеет много преимуществ, включая читаемость кода и минимальные выражения кода.
- **Универсальный** : Python построен для использования во всех контекстах и средах. Примером для языка общего пользования является PHP: он специально разработан как скриптовый язык для веб-разработки на стороне сервера. Напротив, Python *может* использоваться для веб-разработки на стороне сервера, но также для создания настольных приложений.
- **Динамически типизируется** : каждая переменная в Python может ссылаться на любые типы данных. Одно выражение может оценивать данные разных типов в разное время. В связи с этим возможен следующий код:

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

- **Сильно напечатан** : во время выполнения программы вам не разрешено делать что-либо, что несовместимо с типом данных, с которыми вы работаете. Например, нет скрытых преобразований от строк к числам; строка, сделанная из цифр, никогда не будет считаться номером, если вы не решите ее явно:

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

- **Начинающий дружелюбный :)** : Синтаксис и структура Python очень интуитивно понятны. Он является высокоуровневым и предоставляет конструкции, предназначенные для написания четких программ как в маленьком, так и в крупном масштабе. Python поддерживает несколько парадигм программирования, включая объектно-ориентированное, императивное и функциональное программирование или процедурные стили. Он имеет большую, всеобъемлющую стандартную библиотеку и множество простых в установке сторонних библиотек.

Его принципы дизайна изложены в *Zen of Python* .

В настоящее время существуют две основные ветви релиза Python, которые имеют некоторые существенные отличия. Python 2.x - это устаревшая версия, хотя она по-прежнему широко используется. Python 3.x создает набор несовместимых в обратную сторону изменений, которые направлены на сокращение дублирования функций. Чтобы помочь решить, какая версия лучше всего подходит вам, см. Эту статью .

Официальная документация Python также является исчерпывающим и полезным ресурсом, содержащим документацию для всех версий Python, а также руководства, которые помогут вам начать работу.

Существует одна официальная реализация языка, предоставляемого Python.org, обычно называемого CPython, и нескольких альтернативных реализаций языка на других платформах времени исполнения. К ним относятся IronPython (под управлением Python на платформе .NET), Jython (в среде исполнения Java) и PyPy (реализация Python в самом подмножестве).

Версии

Python 3.x

Версия	Дата выхода
[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

Версия	Дата выхода
2,7	2010-07-03

Версия	Дата выхода
2,6	2008-10-02
2.5	2006-09-19
2,4	2004-11-30
2,3	2003-07-29
2,2	2001-12-21
2,1	2001-04-15
2,0	2000-10-16

Examples

Начиная

Python - широко используемый высокоуровневый язык программирования для программирования общего назначения, созданный Guido van Rossum и впервые выпущенный в 1991 году. Python имеет динамическую систему типов и автоматическое управление памятью и поддерживает несколько программных парадигм, включая объектно-ориентированные, функционального программирования и процедурных стилей. Он имеет большую и всеобъемлющую стандартную библиотеку.

В настоящее время активно используются две основные версии Python:

- Python 3.x является текущей версией и находится в активной разработке.
- Python 2.x является устаревшей версией и будет получать только обновления для системы безопасности до 2020 года. Никаких новых функций не будет. Обратите внимание, что многие проекты по-прежнему используют Python 2, хотя переход на Python 3 становится проще.

Вы можете скачать и установить либо версию Python здесь . См. Python 3 и Python 2 для сравнения между ними. Кроме того, некоторые сторонние поставщики предлагают повторно упакованные версии Python, которые добавляют часто используемые библиотеки и другие функции для облегчения настройки для случаев общего использования, таких как математика, анализ данных или научное использование. См. Список на официальном сайте .

Проверьте, установлен ли Python

Чтобы убедиться, что Python установлен правильно, вы можете убедиться, что, выполнив следующую команду в своем любимом терминале (если вы используете ОС Windows, вам

нужно добавить путь к python в переменную окружения, прежде чем использовать его в командной строке):

```
$ python --version
```

Python 3.x 3.0

Если у вас установлен *Python 3* , и это ваша версия по умолчанию (см. **Раздел «Устранение неполадок»**), вы должны увидеть что-то вроде этого:

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

Python 2.x 2.7

Если у вас установлен *Python 2* , и это ваша версия по умолчанию (см. **Раздел «** Поиск и **устранение неисправностей»**), вы должны увидеть что-то вроде этого:

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

Если вы установили Python 3, но `$ python --version` выводит версию Python 2, у вас также установлен Python 2. Это часто бывает в MacOS и во многих дистрибутивах Linux. `$ python3` этого используйте `$ python3` чтобы явно использовать интерпретатор Python 3.

Привет, мир в Python, используя IDLE

IDLE - простой редактор для Python, который поставляется вместе с Python.

Как создать программу Hello, World в IDLE

- Откройте IDLE в вашей системе выбора.
  - В старых версиях Windows его можно найти в разделе « `All Programs` в меню « Windows».
  - В Windows 8+ найдите `IDLE` или найдите его в приложениях, присутствующих в вашей системе.
  - В системах на основе Unix (включая Mac) вы можете открыть его из оболочки, набрав `$ idle python_file.py` .
- Он откроет оболочку с параметрами вверху.

В оболочке есть подсказка из трех прямоугольных скобок:

```
>>>
```

Теперь напишите в подсказке следующий код:

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

Нажмите `Enter` .

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

Файл Hello World Python

Создайте новый файл `hello.py` который содержит следующую строку:

Python 3.x 3.0

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

Python 2.x 2.6

Вы можете использовать функцию `print` Python 3 в Python 2 со следующим оператором `import` :

```
from __future__ import print_function
```

Python 2 имеет ряд функций, которые могут быть импортированы из Python 3 с `__future__` модуля `__future__` , как описано здесь .

Python 2.x 2.7

Если вы используете Python 2, вы также можете ввести строку ниже. Обратите внимание, что это недействительно в Python 3 и, следовательно, не рекомендуется, потому что это уменьшает совместимость кода с перекрестной версией.

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

В своем терминале перейдите в каталог, содержащий файл `hello.py` .

Введите `python hello.py` , затем нажмите `клавишу Enter` .

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

Вы должны увидеть `Hello, World` напечатанную на консоли.

Вы также можете заменить `hello.py` на путь к вашему файлу. Например, если у вас есть файл в вашем домашнем каталоге, а ваш пользователь «пользователь» в Linux, вы можете ввести `python /home/user/hello.py` .

Запустить интерактивную оболочку Python

Выполняя (запуская) команду `python` в вашем терминале, вы получаете интерактивную оболочку Python. Это также известно как Python Interpreter или REPL (для «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
>>>
```

Если вы хотите запустить Python 3 из своего терминала, выполните команду `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
>>>
```

Кроме того, запустите интерактивную подсказку и загрузите файл с помощью `python -i <file.py>` .

В командной строке выполните:

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

Существует несколько способов закрыть оболочку Python:

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

или же

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

Кроме того, `CTRL + D` закроет оболочку и вернет вас в командную строку вашего терминала.

Если вы хотите отменить команду, находящуюся в середине ввода и вернуться к чистой командной строке, оставаясь внутри оболочки Interpreter, используйте `CTRL + C`.

Попробуйте интерактивную оболочку Python в Интернете .

Другие онлайн-оболочки

Различные веб-сайты предоставляют онлайн-доступ к оболочкам Python.

Онлайн-оболочки могут быть полезны для следующих целей:

- Запустите небольшой фрагмент кода с компьютера, на котором отсутствует установка python (смартфоны, планшеты и т. Д.).
- Изучайте базовый Python.
- Решите проблемы онлайн-судьи.

Примеры:

> Отказ от ответственности: автор (ы) документации не связаны ни с какими ресурсами, перечисленными ниже.

- https://www.python.org/shell/ - онлайн-оболочка Python, размещенная на официальном веб-сайте Python.
- https://ideone.com/ - Широко используется в сети для иллюстрации поведения фрагмента кода.
- https://repl.it/languages/python3 - Мощный и простой онлайн-компилятор, IDE и интерпретатор. Код, компилировать и запускать код в Python.
- https://www.tutorialspoint.com/execute_python_online.php - полнофункциональная оболочка UNIX и удобный проводник проекта.
- http://rextester.com/l/python3_online_compiler - Простая и простая в использовании среда разработки, которая показывает время выполнения

Выполнять команды как строку

Python может быть передан произвольным кодом в виде строки в оболочке:

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

Это может быть полезно при объединении результатов скриптов вместе в оболочке.

Оболочки и последующие

Управление пакетами. Рекомендованным PyPA инструментом для установки пакетов Python является PIP . Для установки в командной строке выполните команду `pip install <the package name>` . Например, `pip install numpy` . (Примечание. В окнах вы должны добавить pip в переменные среды PATH. Чтобы этого избежать, используйте `python -m pip install <the`

`package name>`).

Оболочки. До сих пор мы обсуждали разные способы запуска кода с использованием встроенной интерактивной оболочки Python. Оболочки используют интерпретационную способность Python для экспериментов с кодом в режиме реального времени. Альтернативные оболочки включают IDLE - предварительно объединенный графический интерфейс, IPython - известный для расширения интерактивного опыта и т. Д.

Программы. Для долговременного хранения вы можете сохранять контент в .py-файлах и редактировать / выполнять их как скрипты или программы с помощью внешних инструментов, таких как оболочка, IDE (например, PyCharm), ноутбуки Jupyter и т. Д. Промежуточные пользователи могут использовать эти инструменты; однако способы, описанные здесь, достаточны для начала.

Наставник Python позволяет вам пройти через код Python, чтобы вы могли визуализировать, как будет работать программа, и поможет вам понять, где ваша программа пошла не так.

PEP8 определяет правила форматирования кода Python. Очень важно правильно форматировать код, чтобы вы могли быстро прочитать, что делает код.

Создание переменных и назначение значений

Чтобы создать переменную в Python, все, что вам нужно сделать, это указать имя переменной и присвоить ей значение.

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

Python использует `=` для назначения значений переменным. Нет необходимости заранее объявлять переменную (или присваивать ей тип данных), присваивая значение самой переменной, объявляет и инициализирует переменную с этим значением. Невозможно объявить переменную без присвоения ей начального значения.

```
# 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
```

Переменное назначение работает слева направо. Таким образом, следующее приведет к синтаксической ошибке.

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

Вы не можете использовать ключевые слова python как допустимое имя переменной. Вы можете увидеть список ключевых слов:

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

Правила для именования переменных:

1. Имена переменных должны начинаться с буквы или подчеркивания.

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

 9x = False  # starts with numeral
 => SyntaxError: invalid syntax

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

2. Остальная часть вашего имени переменной может состоять из букв, цифр и символов подчеркивания.

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

3. Имена чувствительны к регистру.

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

Несмотря на то, что нет необходимости указывать тип данных при объявлении переменной в Python, при распределении необходимой области в памяти для переменной интерпретатор Python автоматически выбирает для нее наиболее подходящий встроенный тип :

```
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'>
```

Теперь вы знаете основы задания, давайте сделаем эту тонкость о назначении в python в сторону.

Когда вы используете `=` для выполнения операции присваивания, то, что слева от `=` - это ***имя*** для ***объекта*** справа. Наконец, what `=` is присваивает ***ссылку*** объекта справа на ***имя*** слева.

То есть:

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

Так, из многих примеров присваивания выше, если мы выбираем `pi = 3.14` , то `pi` этого имя (не имя, так как объект может иметь несколько имен) для объекта `3.14` . Если вы не понимаете что-то ниже, вернитесь к этому вопросу и прочитайте это снова! Кроме того, вы можете взглянуть на это для лучшего понимания.

Вы можете назначить несколько значений нескольким переменным в одной строке. Обратите внимание, что должно быть такое же количество аргументов в правой и левой сторонах оператора `=` :

```
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
```

Ошибка в последнем примере может быть устранена путем назначения оставшихся значений равным числу произвольных переменных. Эта фиктивная переменная может иметь любое имя, но условно использовать знак подчеркивания (`_`) для назначения нежелательных значений:

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

Обратите внимание, что число _ и количество оставшихся значений должны быть равны. В противном случае «слишком много значений для распаковки ошибки» выбрасывается, как указано выше:

```
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)
```

Вы также можете назначить одно значение нескольким переменным одновременно.

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

При использовании такого каскадного назначения важно отметить, что все три переменные `a` , `b` и `c` относятся к *одному и* тому *же объекту* в памяти, к объекту `int` со значением 1. Другими словами, `a` , `b` и `c` представляют собой три разных имени заданный одному и тому же объекту int. Назначение другого объекта одному из них впоследствии не изменяет других, как и ожидалось:

```
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.
```

Вышеизложенное также верно для изменяемых типов (например, `list` , `dict` и т. Д.), Так же как и для неизменяемых типов (например, `int` , `string` , `tuple` и т. Д.):

```
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]
```

Все идет нормально. Что-то немного изменилось, когда дело доходило до *модификации* объекта (в отличие от *назначения* имени другому объекту, который мы сделали выше), когда каскадное присвоение используется для изменяемых типов. Взгляните ниже, и вы увидите это из первых рук:

```
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
```

Вложенные списки также действительны в python. Это означает, что список может содержать другой список как элемент.

```
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
```

Наконец, переменные в Python не должны оставаться теми же типами, с которыми они были сначала определены - вы можете просто использовать `=` для назначения нового значения переменной, даже если это значение имеет другой тип.

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

a = "New value"
print(a)
# Output: New value
```

Если это вас беспокоит, подумайте о том, что то, что находится слева от `=` - это просто имя для объекта. Во- первых вы вызываете `int` объект со значением 2 , то вы передумали и

решили дать имя `a a` , к `string` объекту, имеющему значение «Нового значения». Простой, не так ли?

Вход пользователя

Интерактивный вход

Чтобы получить вход от пользователя, используйте функцию `input` (**обратите внимание** : в Python 2.x вместо этого функция называется `raw_input` , хотя Python 2.x имеет свою собственную версию `input` которая совершенно иная):

Python 2.x 2.3

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

> **Замечание по безопасности** Не используйте `input()` в Python2 - введенный текст будет оцениваться так, как если бы это было выражение Python (эквивалентное `eval(input())` в Python3), которое может легко стать уязвимостью. См. Эту статью для получения дополнительной информации о рисках использования этой функции.

Python 3.x 3.0

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

Остальная часть этого примера будет использовать синтаксис Python 3.

Функция принимает строковый аргумент, который отображает его как приглашение и возвращает строку. В приведенном выше коде содержится запрос, ожидающий ввода пользователем.

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

Если пользователь вводит «Боб» и попадает, `name` переменной будет присвоено строке `"Bob"` :

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

Обратите внимание, что `input` всегда имеет тип `str` , что важно, если вы хотите, чтобы пользователь вводил числа. Поэтому вам нужно преобразовать `str` прежде чем пытаться использовать его как число:

```
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
```

Примечание. Рекомендуется использовать блоки `try` / `except` чтобы перехватывать исключения при работе с пользовательскими вводами . Например, если ваш код хочет `raw_input` в `int` , и то, что пользователь пишет, является непередаваемым, оно вызывает `ValueError` .

IDLE - графический интерфейс Python

IDLE - это интегрированная среда разработки и обучения Python и является альтернативой командной строке. Как следует из названия, IDLE очень полезно для разработки нового кода или обучения python. В Windows это поставляется с интерпретатором Python, но в других операционных системах вам может потребоваться установить его через диспетчер пакетов.

Основными целями IDLE являются:

- Многострочный текстовый редактор с подсветкой синтаксиса, автозаполнением и интеллектуальным отступом
- Оболочка Python с подсветкой синтаксиса
- Интегрированный отладчик с степпингом, постоянными точками останова и видимостью стека вызовов
- Автоматический отступ (полезно для начинающих, изучающих отступ Python)
- Сохраняя программу Python как .py-файлы, запустите их и отредактируйте позже в любом из них с помощью IDLE.

В IDLE нажмите `F5` или `run Python Shell` чтобы запустить интерпретатор. Использование IDLE может быть лучшим опытом обучения для новых пользователей, потому что код интерпретируется как пользователь пишет.

Обратите внимание, что существует множество альтернатив, см., Например, эту дискуссию или этот список .

Поиск проблемы

- **Windows**

  Если вы работаете в Windows, команда по умолчанию - `python` . Если вы получаете ошибку `"'python' is not recognized"` , наиболее вероятной причиной является то, что местоположение Python не входит в `PATH` среды `PATH` вашей системы. Доступ к нему

можно получить, щелкнув правой кнопкой мыши на «Мой компьютер» и выбрав «Свойства» или перейдя в «Система» через «Панель управления». Нажмите «Дополнительные системные настройки», а затем «Переменные среды ...». Измените переменную `PATH` чтобы включить каталог вашей установки Python, а также папку сценария (обычно `C:\Python27;C:\Python27\Scripts`). Для этого требуются административные привилегии и может потребоваться перезагрузка.

При использовании нескольких версий Python на одном и том же компьютере возможным решением является переименование одного из файлов `python.exe` . Например, именование одной версии `python27.exe` приведет к тому, что `python27` станет командой Python для этой версии.

Вы также можете использовать Python Launcher для Windows, который доступен через установщик и поставляется по умолчанию. Он позволяет вам выбрать версию Python для запуска с помощью `py -[xy]` вместо `python[xy]` . Вы можете использовать последнюю версию Python 2, запустив скрипты с `py -2` и последней версией Python 3, запустив скрипты с `py -3` .

- **Debian / Ubuntu / MacOS**

  В этом разделе предполагается, что местоположение исполняемого файла `python` добавлено в `PATH` среды `PATH` .

  Если вы находитесь в Debian / Ubuntu / MacOS, откройте терминал и введите `python` для Python 2.x или `python3` для Python 3.x.

  Введите, `which python` будет видеть, какой интерпретатор Python будет использоваться.

- **Arch Linux**

  По умолчанию Python на Arch Linux (и потомки) - это Python 3, поэтому используйте `python` или `python3` для Python 3.x и `python2` для Python 2.x.

- **Другие системы**

  Python 3 иногда связан с `python` вместо `python3` . Чтобы использовать Python 2 в этих системах, где он установлен, вы можете использовать `python2` .

Типы данных

Встроенные типы

Булевы

`bool` : Логическое значение `True` или `False` . Логические операции, такие как `and` , `or` , `not` могут выполняться по булевым.

```
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
```

В Python 2.x и в Python 3.x логическое значение также является `int` . Тип `bool` является подклассом типа `int` а `True` и `False` являются его единственными экземплярами:

```
issubclass(bool, int) # True

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

Если в арифметических операциях используются логические значения, их целочисленные значения (`1` и `0` для `True` и `False`) будут использованы для возврата целочисленного результата:

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

чисел

- `int` : целое число

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

  Целые числа в Python имеют произвольные размеры.

  Примечание: в более старых версиях Python был доступен `long` тип, и это отличалось от `int` . Они были объединены.

- `float` : число с плавающей точкой; точность зависит от реализации и архитектуры системы, для CPython тип данных `float` соответствует C double.

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

- `complex` : комплексные номера

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

Операторы `<` , `<=` , `>` и `>=` будут вызывать исключение `TypeError` если любой операнд является сложным числом.

Струны

Python 3.x 3.0

- `str` : **строка** в **Юникоде** . Тип `'hello'`
- `bytes` : **байтовая строка** . Тип `b'hello'`

Python 2.x 2.7

- `str` : **строка байтов** . Тип `'hello'`
- `bytes` : синоним для `str`
- `unicode` : **строка** в **Юникоде** . Тип `u'hello'`

Последовательности и коллекции

Python различает упорядоченные последовательности и неупорядоченные коллекции (такие как `set` и `dict`).

- строки (`str` , `bytes` , `unicode`) являются последовательностями

- `reversed` : обратный порядок `str` с `reversed` функцией

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

- `tuple` : упорядоченный набор `n` значений любого типа (`n >= 0`).

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

  Поддерживает индексирование; неизменный; hashable, если все его члены являются хешируемыми

- `list` : упорядоченный набор `n` значений (`n >= 0`)

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

  Не хешируется; изменчивый.

- `set` : неупорядоченный набор уникальных значений. Элементы должны быть хешируемыми .

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

- `dict` : неупорядоченный набор уникальных пар ключ-значение; ключи должны быть хешируемыми .

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

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

Объект hashable, если он имеет значение хэша, которое никогда не изменяется в течение его жизненного `__hash__()` (ему нужен `__hash__()`), и его можно сравнить с другими объектами (ему нужен `__eq__()`). Объекты Hashable, которые сравнивают равенство, должны иметь одно и то же значение хэш-функции.

Встроенные константы

В сочетании со встроенными типами данных в встроенном пространстве имен имеется небольшое количество встроенных констант:

- `True` : истинное значение встроенного типа `bool`
- `False` : ложное значение встроенного типа `bool`
- `None` : Объект singleton используется для обозначения того, что значение отсутствует.
- `Ellipsis` или `...` : используется в основном Python3 + в любом месте и ограниченное использование в Python2.7 + как часть нотации массива. `numpy` и связанные с ним пакеты используют это как ссылку «включить все» в массивы.
- `NotImplemented` : singleton используется для указания Python, что специальный метод не поддерживает конкретные аргументы, а Python будет пытаться использовать альтернативы, если они доступны.

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

Python 3.x 3.0

`None` кого нет естественного заказа. Использование операторов сравнения заказов (`<` , `<=` , `>=` , `>`) больше не поддерживается и будет вызывать `TypeError` .

Python 2.x 2.7

`None` всегда меньше любого числа (`None < -32` оценивает значение `True`).

Тестирование типа переменных

В python мы можем проверить тип данных объекта с помощью встроенного `type` функции.

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

В условных операторах можно проверить тип данных с помощью `isinstance` . Однако обычно не рекомендуется полагаться на тип переменной.

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

Для получения информации о различиях между `type()` и `isinstance()` read: Различия между isinstance и типом в Python

Чтобы проверить, имеет ли что-то значение `NoneType` :

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

Преобразование между типами данных

Вы можете выполнить явное преобразование типов данных.

Например, «123» имеет тип `str` и может быть преобразован в целое число с использованием функции `int` .

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

Преобразование из строки с плавающей точкой, например, «123.456», может быть выполнено с помощью функции `float` .

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

Вы также можете конвертировать последовательности или типы коллекций

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

Явный тип строки при определении литералов

С одним буквенным надписью непосредственно перед кавычками вы можете указать, какую строку вы хотите определить.

- `b'foo bar'` : результаты `bytes` в Python 3, `str` в Python 2
- `u'foo bar'` : результаты `str` в Python 3, `unicode` в Python 2
- `'foo bar'` : results `str`
- `r'foo bar'` : результат так называемой сырой строки, где экранирование специальных символов не требуется, все принимается дословно, когда вы ввели

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

Переменные и неизменяемые типы данных

Объект называется *изменчивым,* если его можно изменить. Например, когда вы передаете список какой-либо функции, список можно изменить:

```
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
```

Объект называется *неизменяемым,* если его нельзя каким-либо образом изменить. Например, целые числа неизменяемы, поскольку их невозможно изменить:

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

Обратите внимание, что сами **переменные** изменяемы, поэтому мы можем переназначить *переменную* `x` , но это не изменяет объект, `x` который ранее указывал `x` . Он только сделал `x` указывать на новый объект.

Типы данных, экземпляры которых изменяемы, называются *изменяемыми типами данных* , а также для неизменяемых объектов и типов данных.

Примеры неизменяемых типов данных:

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

Примеры изменяемых типов данных:

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

Встроенные модули и функции

Модуль - это файл, содержащий определения и утверждения Python. Функция - это фрагмент кода, который выполняет некоторую логику.

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

Чтобы проверить встроенную функцию в python, мы можем использовать `dir()`. Если вызывается без аргумента, верните имена в текущую область. В противном случае верните алфавитный список имен, содержащих (некоторые) атрибут данного объекта и атрибуты, доступные из него.

```
>>> 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'
```

```
]
```

Чтобы узнать функциональность любой функции, мы можем использовать встроенную функцию `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.
```

Встроенные модули содержат дополнительные функции. Например, чтобы получить квадратный корень из числа, нам нужно включить `math` модуль.

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

Чтобы узнать все функции в модуле, мы можем назначить список функций переменной, а затем распечатать переменную.

```
>>> 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']
```

кажется `__doc__` полезно предоставить некоторую документацию, например, в функциях

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

В дополнение к функциям документация также может предоставляться в модулях. Итак, если у вас есть файл `helloWorld.py` вот так:

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

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

Вы можете получить доступ к своим docstrings следующим образом:

```
>>> import helloWorld
```

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

- Для любого пользовательского типа, его атрибутов, атрибутов его класса и рекурсивно атрибуты базовых классов его класса можно получить с помощью 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__']
```

Любой тип данных может быть просто преобразован в строку с помощью встроенной функции `str` . Эта функция вызывается по умолчанию, когда тип данных передается для `print`

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

Отступ блока

Python использует отступы для определения конструкций управления и контуров. Это способствует читаемости Python, однако для этого требуется, чтобы программист уделял пристальное внимание использованию пробелов. Таким образом, просчет редактора может привести к тому, что код, который ведет себя непредсказуемым образом.

Python использует символ двоеточия (`:`) и отступы для показа , где блоки кода начинаются и заканчивается (Если вы пришли из другого языка, не путайте это с каким - то образом быть связаны с троичным оператором). То есть блоки в Python, такие как функции, петля, `if` пункты и другие конструкции, не имеют идентификаторов концовки. Все блоки начинаются с двоеточия, а затем содержат отступы под ним.

Например:

```
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
```

или же

```
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
```

Блоки, которые содержат ровно один однострочный оператор, могут быть помещены в одну строку, хотя эта форма обычно не считается хорошим стилем:

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

Попытка сделать это более чем с одним заявлением *не* будет работать:

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

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

Пустой блок вызывает `IndentationError` . Используйте `pass` (команда, которая ничего не делает), когда у вас есть блок без содержимого:

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

Пробелы против вкладок

Короче: **всегда** используйте 4 пробела для отступов.

Использование вкладок исключительно возможно, но PEP 8 , руководство по стилю для кода Python, указывает, что пробелы предпочтительнее.

Python 3.x 3.0

Python 3 запрещает смешивать использование вкладок и пробелов для отступов. В этом случае генерируется ошибка времени компиляции: `Inconsistent use of tabs and spaces in indentation` а программа не запускается.

Python 2.x 2.7

Python 2 позволяет смешивать вкладки и пробелы в отступе; это сильно обескураживает. Символ табуляции завершает предыдущий отступ, чтобы быть кратным **8** пробелам . Поскольку обычно редакторы настроены на отображение вкладок в количестве, равном **4** местам, это может вызвать тонкие ошибки.

Цитируя PEP 8 :

> При вызове интерпретатора командной строки Python 2 с параметром `-t` он выдает предупреждения о коде, который незаконно смешивает вкладки и пробелы. При использовании `-tt` эти предупреждения становятся ошибками. Эти варианты настоятельно рекомендуется!

У многих редакторов есть конфигурация «tabs to spaces». При настройке редактора следует различать *символ* табуляции ('\ t') и клавишу `Tab` .

- Вкладка *символ* должен быть настроен , чтобы показать 8 пробелов, чтобы соответствовать семантике языка - по крайней мере , в тех случаях , когда (случайно) смешанные отступы возможно. Редакторы также могут автоматически преобразовывать символ табуляции в пробелы.
- Однако было бы полезно настроить редактор так, чтобы нажатие клавиши `Tab` вставляло 4 пробела вместо того, чтобы вставлять символ табуляции.

Исходный код Python, написанный сочетанием вкладок и пробелов, или с нестандартным числом пространств отступа можно сделать pep8-совместимым с помощью autopep8 . (Менее мощная альтернатива поставляется с большинством установок Python: reindent.py)

Типы коллекций

В Python существует несколько типов коллекций. Хотя типы, такие как `int` и `str` содержат одно значение, типы коллекции содержат несколько значений.

Списки

Тип `list` вероятно, является наиболее часто используемым типом коллекции в Python. Несмотря на свое имя, список больше похож на массив на других языках, в основном на JavaScript. В Python список представляет собой просто упорядоченный набор действительных значений Python. Список может быть создан путем включения значений, разделенных запятыми, в квадратных скобках:

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

Список может быть пустым:

```
empty_list = []
```

Элементы списка не ограничены одним типом данных, что имеет смысл, учитывая, что Python является динамическим языком:

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

Список может содержать другой список:

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

Элементы списка могут быть доступны через *индекс* или числовое представление их позиции. Списки в Python имеют *нулевое индексирование,* означающее, что первый

элемент в списке имеет индекс 0, второй элемент - в индексе 1 и так далее:

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

Индексы также могут быть отрицательными, что означает подсчет с конца списка (`-1` - индекс последнего элемента). Итак, используя список из приведенного выше примера:

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

Списки изменяются, поэтому вы можете изменить значения в списке:

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

Кроме того, можно добавлять и / или удалять элементы из списка:

Добавить объект в конец списка с помощью `L.append(object)` , возвращает `None` .

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

Добавьте новый элемент для отображения по определенному индексу. `L.insert(index, object)`

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

Удалите первое вхождение значения с `L.remove(value)` , возвращает `None`

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

Получить индекс в списке первого элемента, значение которого равно x. Он покажет ошибку, если такого элемента нет.

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

Подсчитать длину списка

```
len(names)
6
```

подсчет количества элементов в списке

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

Изменить список

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

Удалите и верните элемент по индексу (по умолчанию последний элемент) с `L.pop([index])` , возвращает элемент

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

Вы можете перебирать элементы списка, как показано ниже:

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

Кортеж

`tuple` похож на список, за исключением того, что он является фиксированным и неизменным. Таким образом, значения в кортеже не могут быть изменены, а значения не будут добавлены или исключены из кортежа. Кортежи обычно используются для небольших коллекций значений, которые не нужно изменять, например, IP-адрес и порт. Кортежи представлены скобками вместо квадратных скобок:

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

Те же правила индексирования списков также применяются к кортежам. Кортежи также могут быть вложенными, и значения могут быть действительными действительными действительными Python.

Кортеж с одним членом должен быть определен (обратите внимание на запятую) следующим образом:

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

или же

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

или просто используя синтаксис `tuple`

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

Словари

`dictionary` в Python представляет собой набор пар ключ-значение. Словарь окружен фигурными фигурными скобками. Каждая пара разделяется запятой, а ключ и значение разделяются двоеточием. Вот пример:

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

Чтобы получить значение, обратитесь к нему по его ключу:

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

Вы также можете получить все ключи в словаре, а затем перебрать их:

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

Словари сильно напоминают синтаксис JSON. `json` модуль `json` в стандартной библиотеке Python может использоваться для преобразования между JSON и словарями.

задавать

`set` представляет собой набор элементов без повторов и без порядка вставки, но отсортированный порядок. Они используются в ситуациях, когда важно, чтобы некоторые вещи были сгруппированы вместе, а не какой порядок они были включены. Для больших групп данных гораздо быстрее проверить, находится ли элемент в `set` чем он должен делать то же самое для `list` .

Определение `set` очень похоже на определение `dictionary` :

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

Или вы можете построить `set` используя существующий `list` :

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

Проверьте членство в `set` с использованием `in` :

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

Вы можете перебирать `set` точно так же, как и список, но помните: значения будут в произвольном порядке реализации.

defaultdict

`defaultdict` - это словарь со значением по умолчанию для ключей, поэтому ключи, для которых не было явно определено значение, могут быть доступны без ошибок. `defaultdict` особенно полезен, когда значения в словаре представляют собой коллекции (списки, dicts и т. д.) в том смысле, что его не нужно инициализировать каждый раз, когда используется новый ключ.

Значение `defaultdict` никогда не вызовет KeyError. Любой ключ, который не существует, возвращает значение по умолчанию.

Например, рассмотрим следующий словарь

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

Если мы попытаемся получить доступ к несуществующему ключу, python возвращает нам ошибку следующим образом

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

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

KeyError: 'Alabama'
```

Давайте попробуем с `defaultdict` . Его можно найти в модуле коллекций.

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

То, что мы здесь сделали, - установить значение по умолчанию (**Boston**), если ключ-ключ не существует. Теперь заселите диктофон, как раньше:

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

Если мы попытаемся получить доступ к dict с несуществующим ключом, python вернет нам значение по умолчанию, то есть Boston

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

и возвращает созданные значения для существующего ключа, как обычный `dictionary`

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

Полезная утилита

Python имеет несколько функций, встроенных в интерпретатор. Если вы хотите получить информацию о ключевых словах, встроенные функции, модули или темы, откройте консоль Python и введите:

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

Вы получите информацию, непосредственно введя ключевые слова:

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

или внутри утилиты:

```
help> help
```

который покажет объяснение:

```
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)
```

Вы также можете запросить подклассы модулей:

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

Вы можете использовать помощь для доступа к docstrings из разных модулей, которые вы импортировали, например, попробуйте следующее:

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

и вы получите сообщение об ошибке

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

Теперь вы получите список доступных методов в модуле, но только ПОСЛЕ того, как вы его импортировали.

Закрой помощник с `quit`

Создание модуля

Модуль - это импортируемый файл, содержащий определения и утверждения.

Модуль можно создать, создав файл `.py` .

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

Функции в модуле можно использовать, импортируя модуль.

Для модулей, которые вы создали, они должны быть в том же каталоге, что и файл, в который вы их импортируете. (Однако вы также можете поместить их в каталог Python lib с предустановленными модулями, но по возможности следует избегать).

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

Модули могут быть импортированы другими модулями.

```
# greet.py
import hello
```

```
hello.say_hello()
```

Конкретные функции модуля можно импортировать.

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

Модули могут быть сглажены.

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

Модуль может быть автономным исполняемым скриптом.

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

Запустить его!

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

Если модуль находится внутри каталога и должен быть обнаружен с помощью python, каталог должен содержать файл с именем `__init__.py` .

Строковая функция - str () и repr ()

Существуют две функции, которые можно использовать для получения читаемого представления объекта.

`repr(x)` вызывает `x.__repr__()` : представление `x` . `eval` обычно преобразует результат этой функции обратно в исходный объект.

`str(x)` вызывает `x.__str__()` : человекочитаемая строка, описывающая объект. Это может привести к некоторым техническим деталям.

магнезии ()

Для многих типов эта функция пытается вернуть строку, которая даст объект с тем же значением при передаче в `eval()` . В противном случае представление представляет собой строку, заключенную в угловые скобки, которая содержит имя типа объекта вместе с дополнительной информацией. Это часто включает имя и адрес объекта.

ул ()

Для строк это возвращает строку. Разница между этим и представлением `repr(object)` заключается в том, что `str(object)` не всегда пытается вернуть строку, приемлемую для `eval()` . Скорее, его цель - вернуть печатную или «удобочитаемую» строку. Если аргумент не задан, это возвращает пустую строку, `''` .

Пример 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
```

Пример 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)'
```

При написании класса вы можете переопределить эти методы, чтобы делать все, что хотите:

```
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)
```

Используя вышеприведенный класс, мы можем увидеть результаты:

```
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
```

Установка внешних модулей с помощью pip

`pip`

- ваш друг, когда вам нужно установить любой пакет из множества вариантов, доступных в индексе пакета python (PyPI). `pip` уже установлен, если вы используете Python 2> = 2.7.9 или Python 3> = 3.4, загруженный с python.org. Для компьютеров под управлением Linux или другого * nix с собственным менеджером пакетов, `pip` часто должен быть установлен вручную.

В случаях, когда установлены оба Python 2 и Python 3, `pip` часто ссылается на Python 2 и `pip3` на Python 3. Использование `pip` будет устанавливать пакеты только для Python 2, а `pip3` будет устанавливать пакеты только для Python 3.

Поиск / установка пакета

Поиск пакета так же просто, как ввод текста

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

Установка пакета так же проста, как набирать *(в терминале / командной строке, а не в интерпретаторе 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
```

где `xxx` - номер версии пакета, который вы хотите установить.

Когда ваш сервер находится за прокси-сервером, вы можете установить пакет, используя следующую команду:

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

Обновление установленных пакетов

Когда появляются новые версии установленных пакетов, они автоматически не устанавливаются в вашу систему. Чтобы узнать, какой из установленных пакетов устарел, запустите:

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

Чтобы обновить использование определенного пакета

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

Обновление всех устаревших пакетов не является стандартной функциональностью `pip` .

Модернизация

Вы можете обновить существующую установку на пике, используя следующие команды:

- В Linux или macOS X:

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

  Возможно, вам понадобится использовать `sudo` with pip в некоторых Linux-системах

- В Windows:

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

  или же

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

Для получения дополнительной информации о пипе читайте здесь .

Установка Python 2.7.x и 3.x

> **Примечание** . Следующие инструкции написаны для Python 2.7 (если не указано): инструкции для Python 3.x аналогичны.

WINDOWS

Сначала загрузите последнюю версию Python 2.7 с официального сайта (https://www.python.org/downloads/) . Версия предоставляется как пакет MSI. Чтобы установить его вручную, просто дважды щелкните файл.

По умолчанию Python устанавливается в каталог:

```
C:\Python27\
```

Предупреждение: установка автоматически не изменяет переменную среды PATH.

Предполагая, что ваша установка Python находится в C: \ Python27, добавьте это в свой PATH:

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

Теперь, чтобы проверить правильность установки Python, напишите в cmd:

```
python --version
```

Python 2.x и 3.x Side-By-Side

Чтобы установить и использовать оба Python 2.x и 3.x рядом с Windows-машиной:

1. Установите Python 2.x с помощью установщика MSI.
   - Убедитесь, что Python установлен для всех пользователей.
   - Необязательно: добавьте Python в `PATH` чтобы сделать Python 2.x доступным из командной строки с помощью `python` .
2. Установите Python 3.x с помощью соответствующего установщика.
   - Опять же, убедитесь, что Python установлен для всех пользователей.
   - Необязательно: добавьте Python в `PATH` чтобы сделать Python 3.x доступным из командной строки с помощью `python` . Это может переопределить настройки Python 2.x `PATH` , поэтому дважды проверьте свой `PATH` и убедитесь, что он настроен на ваши предпочтения.
   - Обязательно установите `py launcher` установку `py launcher` для всех пользователей.

Python 3 установит пусковую установку Python, которая может использоваться для запуска Python 2.x и Python 3.x взаимозаменяемо из командной строки:

```
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.
>>>
```

Чтобы использовать соответствующую версию `pip` для конкретной версии Python, используйте:

```
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

Последние версии CentOS, Fedora, Redhat Enterprise (RHEL) и Ubuntu поставляются с Python 2.7.

Чтобы установить Python 2.7 на Linux вручную, просто выполните следующие действия в терминале:

```
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
```

Также добавьте путь к новому питону в переменной среды PATH. Если новый python находится в `/root/python-2.7.X` тогда выполните `export PATH = $PATH:/root/python-2.7.X`

Теперь, чтобы проверить правильность установки Python, напишите в терминале:

```
python --version
```

Ubuntu (From Source)

Если вам нужен Python 3.6, вы можете установить его из источника, как показано ниже (Ubuntu 16.10 и 17.04 имеют версию 3.6 в универсальном репозитории). Ниже приведены шаги для Ubuntu 16.04 и более низких версий:

```
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
```

Macos

Как мы говорим, macOS поставляется с Python 2.7.10, но эта версия устарела и немного изменена из обычного Python.

> Версия Python, поставляемая с OS X, отлично подходит для обучения, но это не хорошо для разработки. Версия, поставляемая с OS X, может быть устаревшей из официальной текущей версии Python, которая считается стабильной производственной версией. (источник)

Установить Homebrew :

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

Установите Python 2.7:

```
brew install python
```

Для Python 3.x используйте команду `brew install python3` .

Прочитайте Начало работы с языком Python онлайн: https://riptutorial.com/ru/python/topic/193/начало-работы-с-языком-python

глава 2: * args и ** kwargs

замечания

Здесь есть несколько замечаний:

1. Названия `args` и `kwargs` используются по соглашению, они не являются частью спецификации языка. Таким образом, они эквивалентны:

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

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

2. У вас может быть не более одного `args` или более одного параметра `kwargs` (однако они не требуются)

   ```
   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. Если какой-либо позиционный аргумент следует за аргументами `*args` , они являются аргументами только для ключевого слова, которые могут передаваться только по имени. Вместо `*args` можно использовать одну звезду, чтобы заставить значения использовать аргументы с ключевыми словами, не предоставляя список переменных параметров. Списки параметров только для ключевых слов доступны только в 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` должны быть последними в списке параметров.

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

Examples

Использование * args при написании функций

Вы можете использовать звезду * при написании функции для сбора всех позиционных (т. Е. Неназванных) аргументов в кортеже:

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

Метод вызова:

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

В этом вызове farg будет назначаться как всегда, а два других будут передаваться в корте args в том порядке, в котором они были получены.

Использование ** kwargs при записи функций

Вы можете определить функцию, которая принимает произвольное количество аргументов ключевого слова (named), используя двойную звездочку `**` перед именем параметра:

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

При вызове метода Python будет строить словарь всех аргументов ключевого слова и сделать его доступным в теле функции:

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

Обратите внимание, что параметр ** kwargs в определении функции всегда должен быть

последним параметром, и он будет соответствовать только аргументам, которые были переданы после предыдущих.

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

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

Внутри тела функции `kwargs` манипулируют так же, как словарь; для доступа к отдельным элементам в `kwargs` вы просто просматриваете их так же, как и с обычным словарем:

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

Теперь вызов `print_kwargs(a="two", b=1)` показывает следующий вывод:

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

Использование * args при вызове функций

Общим примером использования `*args` в определении функции является делегирование обработки либо завернутой, либо унаследованной функции. Типичный пример может быть в методе `__init__` класса

```
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
```

Здесь параметр обрабатывается дочерний класс после того, как все другие аргументы (позиционные и ключевые слова) передаются на - и обрабатываются - базового класса. `a`

Например:

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

Здесь происходит функция класса `B` `__init__` которая видит аргументы `1, 2, 3` . Он знает, что ему нужно взять один позиционный аргумент (`a`), поэтому он захватывает первый аргумент, переданный в (`1`), поэтому в области функции `a == 1` .

Затем он видит, что ему нужно принять произвольное количество позиционных аргументов (`*args`), чтобы он оставил остальные позиционные аргументы, переданные в (`1, 2`), и наполнил их в `*args` . Теперь (в рамках функции) `args == [2, 3]` .

Затем он вызывает функцию `__init__` класса `A` с `*args` . Python видит `*` перед args и «распаковывает» список в аргументы. В этом примере, когда класс `B` 's `__init__` вызовов функций класса `A` ' s `__init__` функцию, оно будет передано аргументы `2, 3` (т.е. `A(2, 3)`).

Наконец, он устанавливает собственное свойство `x` в первый позиционный аргумент `a` , равный `1` .

Использование ** kwargs при вызове функций

Вы можете использовать словарь для назначения значений параметрам функции; используя имя параметра в качестве ключей в словаре и значение этих аргументов, привязанных к каждому ключу:

```
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)
```

Использование * args при вызове функций

Эффект использования оператора `*` при аргументе при вызове функции заключается в распаковке списка или аргумента кортежа

```
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
```

Обратите внимание, что длина помеченного аргумента должна быть равна числу аргументов функции.

Общей идиомой python является использование оператора распаковки `*` с `zip` функцией для изменения его эффектов:

```
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)
```

Ключевое слово и требуемые по ключу аргументы

Python 3 позволяет вам определять аргументы функции, которые могут быть назначены только по ключевым словам, даже без значений по умолчанию. Это делается с помощью star * для использования дополнительных позиционных параметров без установки параметров ключевого слова. Все аргументы после * являются ключевыми словами (то есть непозиционными) аргументами. Обратите внимание, что если аргументы только для ключевого слова не заданы по умолчанию, они все равно необходимы при вызове функции.

```
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
```

Заполнение значений kwarg со словарем

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

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

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

** kwargs и значения по умолчанию

Использовать значения по умолчанию с ** kwargs

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

fun()
# print 0
```

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

Прочитайте * args и ** kwargs онлайн: https://riptutorial.com/ru/python/topic/2475/--args-и----kwargs

глава 3: ArcPy

замечания

В этом примере используется курсор поиска из модуля Data Access (da) ArcPy.

Не путайте синтаксис arcpy.da.SearchCursor с более ранней и медленной дугой. SearchCursor ().

Модуль доступа к данным (arcpy.da) доступен только с ArcGIS 10.1 для настольных компьютеров.

Examples

Печать значения одного поля для всех строк класса объектов в базе геоданных файлов с помощью Search Cursor

Чтобы напечатать тестовое поле (TestField) из тестового класса функций (TestFC) в базе геоданных тестового файла (Test.gdb), расположенной во временной папке (C: \ Temp):

```
with arcpy.da.SearchCursor(r"C:\Temp\Test.gdb\TestFC",["TestField"]) as cursor:
    for row in cursor:
        print row[0]
```

createDissolvedGDB для создания файла gdb в рабочей области

```
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
```

Прочитайте ArcPy онлайн: https://riptutorial.com/ru/python/topic/4693/arcpy

глава 4: ChemPy - пакет python

Вступление

ChemPy представляет собой пакет python, разработанный в основном для решения и решения проблем в физическом, аналитическом и неорганическом химическом составе. Это бесплатный инструментарий Python с открытым исходным кодом для приложений для химии, химической инженерии и материалов.

Examples

Формулы анализа

```
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
```

В композиции атомные числа (и 0 для заряда) используются в качестве ключей, и количество каждого вида становится соответствующим значением.

Балансирующая стехиометрия химической реакции

```
 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%'}
```

Реакции балансировки

```
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
```

Химические равновесия

```
 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
```

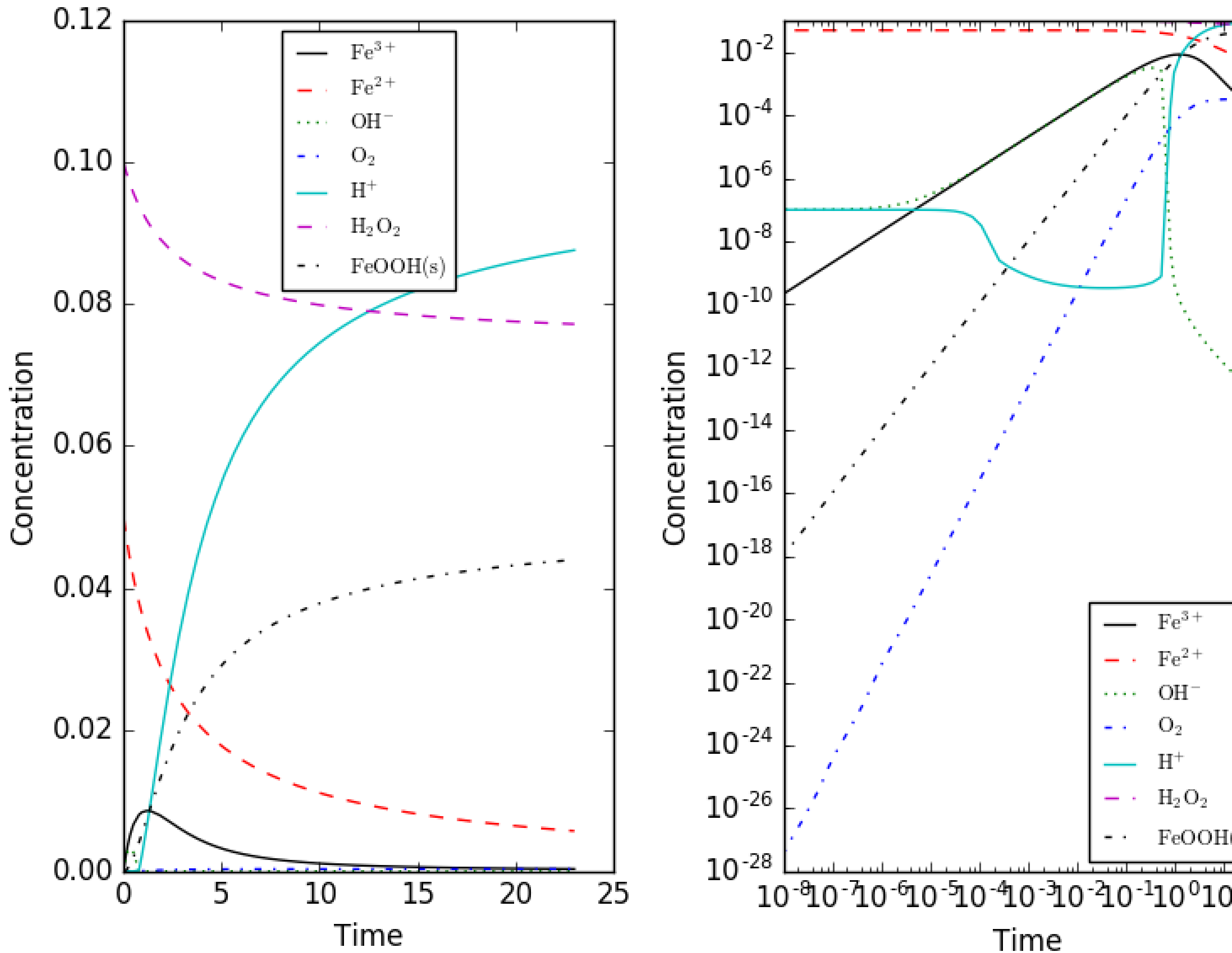
Ионная сила

```
 from chempy.electrolytes import ionic_strength
     ionic_strength({'Fe+3': 0.050, 'ClO4-': 0.150}) == .3
    True
```

Химическая кинетика (система обыкновенных дифференциальных уравнений)

```
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()
```



Прочитайте ChemPy - пакет python онлайн: https://riptutorial.com/ru/python/topic/10625/chempy---пакет-python

глава 5: Conditionals

Вступление

Условные выражения, включающие ключевые слова, такие как if, elif и другие, предоставляют программам Python возможность выполнять разные действия в зависимости от логического условия: True или False. В этом разделе рассматриваются использование условных выражений Python, логической логики и трехмерных выражений.

Синтаксис

- <выражение> if <условное> else <выражение> # Тернарный оператор

Examples

if, elif, и еще

В Python вы можете определить серию условных выражений, `if` для первого, `elif` для остальных, вплоть до окончательного (необязательного) `else` для чего-либо, не пойманного другими условными выражениями.

```
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.")
```

Выходы `Number is bigger than 2`

Использование `else if` вместо `elif` приведет к синтаксической ошибке и не будет разрешено.

Условное выражение (или «Тернарный оператор»)

Тернарный оператор используется для внутренних условных выражений. Его лучше всего использовать в простых, сжатых операциях, которые легко читать.

- Порядок аргументов отличается от многих других языков (таких как C, Ruby, Java и т. Д.), Что может привести к ошибкам, когда люди, незнакомые с «неожиданным» поведением Python, используют его (они могут отменить порядок).
- Некоторые считают это «громоздким», так как оно противоречит нормальному потоку

мысли (сначала думая о состоянии, а затем о последствиях).

```
n = 5

"Greater than 2" if n > 2 else "Smaller than or equal to 2"
# Out: 'Greater than 2'
```

Результат этого выражения будет таким, какой он читается на английском языке - если условное выражение True, тогда оно будет оцениваться с выражением слева, в противном случае - с правой стороны.

Десятичные операции также могут быть вложены, как здесь:

```
n = 5
"Hello" if n > 10 else "Goodbye" if n > 5 else "Good day"
```

Они также предоставляют метод включения условий в лямбда-функции .

Если утверждение

```
if condition:
    body
```

Операторы `if` проверяют условие. Если он принимает значение `True` , он выполняет тело оператора `if` . Если он оценивает значение `False` , он пропускает тело.

```
if True:
    print "It is true!"
>> It is true!

if False:
    print "This won't get printed.."
```

Условием может быть любое допустимое выражение:

```
if 2 + 2 == 4:
    print "I know math!"
>> I know math!
```

Else statement

```
if condition:
    body
else:
    body
```

Оператор else будет выполнять его тело только в том случае, если предыдущие условные операторы все оцениваются как 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!
```

Логические выражения логики

Логические логические выражения, помимо оценки `True` или `False` , возвращают *значение,* которое интерпретируется как `True` или `False` . Это Pythonic способ представлять логику, которая в противном случае могла бы потребовать проверку if-else.

И оператор

Оператор `and` вычисляет все выражения и возвращает последнее выражение, если все выражения оцениваются как `True` . В противном случае он возвращает первое значение, которое имеет значение `False` :

```
>>> 1 and 2
2

>>> 1 and 0
0

>>> 1 and "Hello World"
"Hello World"

>>> "" and "Pancakes"
""
```

Или оператор

Оператор `or` вычисляет выражения слева направо и возвращает первое значение, которое оценивается как `True` или последнее значение (если ни один не `True`).

```
>>> 1 or 2
1

>>> None or 1
```

```
1

>>> 0 or []
[]
```

Ленивая оценка

Когда вы используете этот подход, помните, что оценка ленива. Выражения, которые не требуется оценивать для определения результата, не оцениваются. Например:

```
>>> def print_me():
        print('I am here!')
>>> 0 and print_me()
0
```

В приведенном выше примере `print_me` никогда не выполняется, поскольку Python может определить, что все выражение `False` когда оно встречает `0` (`False`). Имейте это в виду, если `print_me` необходимо выполнить для обслуживания вашей логики программы.

Тестирование на несколько условий

Общей ошибкой при проверке нескольких условий является неправильное применение логики.

В этом примере мы пытаемся проверить, являются ли две переменные большими, чем 2. Оператор оценивается как - `if (a) and (b > 2)` . Это приводит к неожиданному результату, так как `bool(a)` оценивается как `True` когда `a` не равен нулю.

```
>>> a = 1
>>> b = 6
>>> if a and b > 2:
...     print('yes')
... else:
...     print('no')

yes
```

Каждая переменная должна сравниваться отдельно.

```
>>> if a > 2 and b > 2:
...     print('yes')
... else:
...     print('no')

no
```

Другая аналогичная ошибка возникает при проверке, является ли переменная одной из нескольких значений. Выражение в этом примере оценивается как - `if (a == 3) or (4) or (6)` . Это приводит к неожиданному результату, так как `bool(4)` и `bool(6)` оценивают значение `True`

```
>>> a = 1
>>> if a == 3 or 4 or 6:
...     print('yes')
... else:
...     print('no')

yes
```

Снова каждое сравнение должно быть сделано отдельно

```
>>> if a == 3 or a == 4 or a == 6:
...     print('yes')
... else:
...     print('no')

no
```

Использование оператора in является каноническим способом его записи.

```
>>> if a in (3, 4, 6):
...     print('yes')
... else:
...     print('no')

no
```

Значения правды

Следующие значения считаются ложными, поскольку они оцениваются как `False` при применении к булевому оператору.

- Никто
- Ложь
- `0` или любое числовое значение, равное нулю, например `0L` , `0.0` , `0j`
- Пустые последовательности: `''` , `""` , `()` , `[]`
- Пустые сопоставления: `{}`
- Пользовательские типы, в `__len__` методы `__bool__` или `__len__` возвращают 0 или `False`

Все остальные значения в Python оцениваются как `True` .

Примечание. Общей ошибкой является просто проверка фальшивости операции, которая возвращает разные значения Falsey, где разница имеет значение. Например,

использование `if foo()` а не более явное, `if foo() is None`

Используя функцию cmp, чтобы получить результат сравнения двух объектов

Python 2 включает в себя функцию `cmp` которая позволяет вам определить, меньше ли один объект, равный или больший, чем другой объект. Эта функция может использоваться для выбора выбора из списка на основе одного из этих трех вариантов.

Предположим, вам нужно напечатать `'greater than'` если `x > y`, `'less than'` если `x < y` и `'equal'` если `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)` возвращает следующие значения

сравнение	Результат
x <y	-1
x == y	0
x> y	1

Эта функция удаляется на Python 3. Вы можете использовать вспомогательную функцию `cmp_to_key(func)` расположенную в `functools` в Python 3, для преобразования старых функций сравнения в ключевые функции.

Оценка условного выражения с использованием методов List List

Python позволяет взломать списки для оценки условных выражений.

Например,

```
[value_false, value_true][<conditional-test>]
```

Пример:

```
>> n = 16
>> print [10, 20][n <= 15]
10
```

Здесь `n<=15` возвращает `False` (что равно 0 в Python). Так что Python оценивает:

```
[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

Встроенный метод `__cmp__` возвратил 3 возможных значения: 0, 1, -1, где cmp (x, y) возвращено 0: если оба объекта были одинаковыми 1: x> y -1: x <y

Это можно использовать со списком, чтобы вернуть первый (т. Е. Индекс 0), второй (т. Е. Индекс 1) и последний (т. Е. Индекс -1) элемент списка. Предоставление нам условного типа:

```
[value_equals, value_greater, value_less][<conditional-test>]
```

Наконец, во всех приведенных выше примерах Python оценивает обе ветви перед ее выбором. Чтобы оценить только выбранную ветку:

```
[lambda: value_false, lambda: value_true][<test>]()
```

где добавление `()` в конце гарантирует, что лямбда-функции только вызывается / оценивается в конце. Таким образом, мы оцениваем только выбранную ветвь.

Пример:

```
count = [lambda:0, lambda:N+1][count==N]()
```

Тестирование, если объект None и его назначение

Вы часто захотите присвоить объект объекту, если он `None` , указав, что он не был назначен. Мы будем использовать `aDate` .

Самый простой способ сделать это - использовать тест `is None` .

```
if aDate is None:
    aDate=datetime.date.today()
```

(Обратите внимание, что более Pythonic говорит, что `is None` вместо `== None` .)

Но это можно немного оптимизировать, используя понятие, что `not None` будет оценивать значение `True` в булевом выражении. Следующий код эквивалентен:

```
if not aDate:
    aDate=datetime.date.today()
```

Но есть более питонический путь. Следующий код также эквивалентен:

```
aDate=aDate or datetime.date.today()
```

Это делает оценку короткого замыкания . Если `aDate` инициализируется и `not None` является `not None` , тогда он присваивается самому себе без чистого эффекта. Если это `is None` , то `datetime.date.today()` присваивается `aDate` .

Прочитайте Conditionals онлайн: https://riptutorial.com/ru/python/topic/1111/conditionals

глава 6: ConfigParser

Вступление

Этот модуль предоставляет класс ConfigParser, который реализует базовый язык конфигурации в файлах INI. Вы можете использовать это для написания программ Python, которые могут быть легко настроены конечными пользователями.

Синтаксис

- Каждая новая строка содержит новую пару ключевых значений, разделенную знаком =
- Ключи можно разделять по разделам
- В файле INI заголовок каждого раздела записывается между скобками: []

замечания

Все возвращаемые значения из `ConfigParser.ConfigParser().get` являются строками. Он может быть преобразован в более распространенные типы благодаря `eval`

Examples

Основное использование

В config.ini:

```
[DEFAULT]
debug = True
name = Test
password = password

[FILES]
path = /path/to/file
```

В 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'
```

Создание конфигурационного файла программно

Файл конфигурации содержит разделы, каждый раздел содержит ключи и значения. Модуль configparser может использоваться для чтения и записи файлов конфигурации. Создание файла конфигурации: -

```
import configparser
config = configparser.ConfigParser()
config['settings']={'resolution':'320x240',
                    'color':'blue'}
with open('example.ini', 'w') as configfile:
    config.write(configfile)
```

Выходной файл содержит структуру ниже

```
[settings]
resolution = 320x240
color = blue
```

Если вы хотите изменить конкретное поле, получите поле и присвойте значение

```
settings=config['settings']
settings['color']='red'
```

Прочитайте ConfigParser онлайн: https://riptutorial.com/ru/python/topic/9186/configparser

глава 7: ctypes

Вступление

`ctypes` - это встроенная библиотека python, которая вызывает экспортированные функции из встроенных библиотек.

Примечание. Поскольку эта библиотека обрабатывает скомпилированный код, она зависит от ОС.

Examples

Основное использование

Предположим, мы хотим использовать функцию `ntohl` `libc` .

Во-первых, мы должны загрузить `libc.so` :

```
>>> from ctypes import *
>>> libc = cdll.LoadLibrary('libc.so.6')
>>> libc
<CDLL 'libc.so.6', handle baadf00d at 0xdeadbeef>
```

Затем мы получаем объект функции:

```
>>> ntohl = libc.ntohl
>>> ntohl
<_FuncPtr object at 0xbaadf00d>
```

И теперь мы можем просто вызвать функцию:

```
>>> ntohl(0x6C)
1811939328
>>> hex(_)
'0x6c000000'
```

Что делает именно то, что мы ожидаем от этого.

Обычные подводные камни

Не удалось загрузить файл

Первой возможной ошибкой является невозможность загрузить библиотеку. В этом случае OSError обычно поднимается.

Это происходит потому, что файл не существует (или не может быть найден ОС):

```
>>> 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
```

Как вы можете видеть, ошибка ясна и довольно показательна.

Вторая причина заключается в том, что файл найден, но не соответствует формату.

```
>>> 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
```

В этом случае файл является файлом сценария, а не файлом `.so` . Это также может произойти при попытке открыть `.dll` файл на машине Linux или 64-битном файле на 32-битном интерпретаторе python. Как вы можете видеть, в этом случае ошибка немного более расплывчата и требует некоторого рытья.

Отсутствие доступа к функции

Предположим, что мы успешно загрузили файл `.so` , тогда нам нужно получить доступ к нашей функции, как это было сделано в первом примере.

Когда используется несуществующая функция, возникает `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
```

Основной объект ctypes

Самый основной объект - это int:

```
>>> obj = ctypes.c_int(12)
>>> obj
c_long(12)
```

Теперь `obj` ссылается на кусок памяти, содержащий значение 12.

Доступ к этому значению можно получить напрямую и даже изменить:

```
>>> obj.value
12
>>> obj.value = 13
>>> obj
c_long(13)
```

Поскольку `obj` относится к куску памяти, мы также можем узнать его размер и местоположение:

```
>>> sizeof(obj)
4
>>> hex(addressof(obj))
'0xdeadbeef'
```

массивы ctypes

Как знает любой хороший программист C, одно значение не дойдет до вас. Что действительно вызовет нас, это массивы!

```
>>> c_int * 16
<class '__main__.c_long_Array_16'>
```

Это не реальный массив, но это довольно чертовски близко! Мы создали класс, который обозначает массив из 16 `int` s.

Теперь все, что нам нужно сделать, это инициализировать его:

```
>>> arr = (c_int * 16)(*range(16))
>>> arr
<__main__.c_long_Array_16 object at 0xbaddcafe>
```

Теперь `arr` является фактическим массивом, который содержит числа от 0 до 15.

Доступ к ним возможен, как и любой список:

```
>>> arr[5]
5
>>> arr[5] = 20
>>> arr[5]
20
```

И как и любой другой объект `ctypes` , он также имеет размер и местоположение:

```
>>> sizeof(arr)
64 # sizeof(c_int) * 16
>>> hex(addressof(arr))
'0xc00010ff'
```

Функции упаковки для ctypes

В некоторых случаях функция C принимает указатель на функцию. Как пользователи `ctypes` , мы хотели бы использовать эти функции и даже передавать функцию python в качестве аргументов.

Определим функцию:

```
>>> def max(x, y):
        return x if x >= y else y
```

Теперь эта функция принимает два аргумента и возвращает результат одного и того же типа. Для примера предположим, что тип - это int.

Как и в примере массива, мы можем определить объект, который обозначает этот прототип:

```
>>> CFUNCTYPE(c_int, c_int, c_int)
<CFunctionType object at 0xdeadbeef>
```

Этот прототип обозначает функцию, которая возвращает `c_int` (первый аргумент) и принимает два аргумента `c_int` (другие аргументы).

Теперь давайте обернем функцию:

```
>>> CFUNCTYPE(c_int, c_int, c_int)(max)
<CFunctionType object at 0xdeadbeef>
```

Прототипы функций имеют больше возможностей: они могут обернуть функцию `ctypes` (например, `libc.ntohl`) и убедиться, что при вызове функции используются правильные аргументы.

```
>>> 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)
```

Комплексное использование

Давайте объединим все приведенные выше примеры в один сложный сценарий: используя функцию `lfind` `libc` .

Подробнее о функции читайте на странице руководства . Я настоятельно рекомендую вам прочитать его перед продолжением.

Сначала мы определим правильные прототипы:

```
>>> 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)
```

Затем создадим переменные:

```
>>> key = c_int(12)
>>> arr = (c_int * 16)(*range(16))
>>> nmemb = c_uint(16)
```

И теперь мы определяем функцию сравнения:

```
>>> def compar(x, y):
        return x.contents.value - y.contents.value
```

Обратите внимание, что `x` и `y` являются `POINTER(c_int)` , поэтому нам нужно разыменовать их и принять их значения, чтобы фактически сравнить значение, хранящееся в памяти.

Теперь мы можем объединить все вместе:

```
>>> lfind = lfind_proto(libc.lfind)
>>> ptr = lfind(byref(key), byref(arr), byref(nmemb), sizeof(c_int), compar_proto(compar))
```

`ptr` - возвращаемый указатель пустоты. Если `key` не был найден в `arr` , значение будет `None` , но в этом случае мы получили действительное значение.

Теперь мы можем преобразовать его и получить доступ к значению:

```
>>> cast(ptr, POINTER(c_int)).contents
c_long(12)
```

Кроме того, мы видим, что `ptr` указывает на правильное значение внутри `arr` :

```
>>> addressof(arr) + 12 * sizeof(c_int) == ptr
True
```

Прочитайте ctypes онлайн: https://riptutorial.com/ru/python/topic/9050/ctypes

глава 8: Enum

замечания

Перечисления были добавлены в Python в версии 3.4 с помощью PEP 435 .

Examples

Создание перечисления (Python с 2.4 по 3.3)

Резервные копии были отправлены из Python 3.4 в Python 2.4 через Python 3.3. Вы можете получить это резервное копирование enum34 из PyPI.

```
pip install enum34
```

Создание перечисления идентично тому, как оно работает в 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
```

итерация

Перечисления повторяемы:

```
class Color(Enum):
    red = 1
    green = 2
    blue = 3

[c for c in Color]  # [<Color.red: 1>, <Color.green: 2>, <Color.blue: 3>]
```

Прочитайте Enum онлайн: https://riptutorial.com/ru/python/topic/947/enum

глава 9: hashlib

Вступление

hashlib реализует общий интерфейс для многих различных алгоритмов безопасного хэша и дайджеста сообщений. Включены алгоритмы безопасного хеша FIPS SHA1, SHA224, SHA256, SHA384 и SHA512.

Examples

MD5-хэш строки

Этот модуль реализует общий интерфейс для многих различных алгоритмов безопасного хэша и дайджеста сообщений. Включены алгоритмы безопасного хеша FIPS SHA1, SHA224, SHA256, SHA384 и SHA512 (определенные в FIPS 180-2), а также алгоритм MD5 MD5 (определенный в Internet RFC 1321).

Существует один метод-конструктор, который называется для каждого типа хэша. Все возвращают хэш-объект с тем же простым интерфейсом. Например: используйте `sha1()` для создания хэш-объекта SHA1.

```
hash.sha1()
```

Конструкторами для хэш-алгоритмов, которые всегда присутствуют в этом модуле, являются `md5()` , `sha1()` , `sha224()` , `sha256()` , `sha384()` и `sha512()` .

Теперь вы можете передать этот объект произвольными строками с помощью метода `update()` . В любой момент вы можете запросить у него дайджест конкатенации строк, `hexdigest()` ему до сих пор, используя методы `digest()` или `hexdigest()` .

```
hash.update(arg)
```

> Обновите хэш-объект с помощью строки arg. Повторные вызовы эквивалентны одному вызову с конкатенацией всех аргументов: m.update (a); m.update (b) эквивалентно m.update (a + b).

```
hash.digest()
```

> Верните дайджест строк, переданных методу update (). Это строка байтов digest_size, которая может содержать символы, отличные от ASCII, включая нулевые байты.

```
hash.hexdigest()
```

> Как digest (), за исключением того, что дайджест возвращается как строка двойной длины, содержащая только шестнадцатеричные цифры. Это можно использовать для безопасного обмена данными в электронной почте или других недвоичных средах.

Вот пример:

```
>>> 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
```

или же:

```
hashlib.md5("Nobody inspects the spammish repetition").hexdigest()
    'bb649c83dd1ea5c9d9dec9a18df0ffe9'
```

алгоритм, предоставляемый OpenSSL

Генерический конструктор `new()` который принимает имя строки желаемого алгоритма в качестве его первого параметра, также существует, чтобы разрешить доступ к перечисленным выше хэшам, а также любые другие алгоритмы, которые может предложить ваша библиотека OpenSSL. Именованные конструкторы намного быстрее, чем `new()` и должны быть предпочтительнее.

Используя `new()` с алгоритмом, предоставляемым OpenSSL:

```
>>> h = hashlib.new('ripemd160')
>>> h.update("Nobody inspects the spammish repetition")
>>> h.hexdigest()
'cc4a5ce1b3df48aec5d22d1f16b894a0b894eccc'
```

Прочитайте hashlib онлайн: https://riptutorial.com/ru/python/topic/8980/hashlib

глава 10: Heapq

Examples

Самые большие и мелкие предметы в коллекции

Для того, чтобы найти самые большие предметы в коллекции, `heapq` модуль имеет функцию под названием `nlargest` , мы передаем его два аргумента, то первый из них является количество элементов , которые мы хотим получить, второй это имя коллекции:

```
import heapq


numbers = [1, 4, 2, 100, 20, 50, 32, 200, 150, 8]
print(heapq.nlargest(4, numbers))  # [200, 150, 100, 50]
```

Аналогично, чтобы найти наименьшие элементы в коллекции, мы используем функцию `nsmallest` :

```
print(heapq.nsmallest(4, numbers))  # [1, 2, 4, 8]
```

Обе функции `nlargest` и `nsmallest` принимают необязательный аргумент (ключевой параметр) для сложных структур данных. В следующем примере показано использование свойства `age` для извлечения старшего и младшего людей из словаря `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'}]
```

Самый маленький предмет в коллекции

Самое интересное свойство `heap` состоит в том, что ее наименьший элемент всегда является первым элементом: `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]
```

Прочитайте Heapq онлайн: https://riptutorial.com/ru/python/topic/7489/heapq

глава 11: HTML-анализ

Examples

Найдите текст после элемента в BeautifulSoup

Представьте, что у вас есть следующий HTML:

```
<div>
    <label>Name:</label>
    John Smith
</div>
```

И вам нужно найти текст «John Smith» после элемента `label` .

В этом случае вы можете найти элемент `label` по тексту, а затем использовать свойство `.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())
```

Печатает `John Smith` .

Использование селекторов CSS в BeautifulSoup

BeautifulSoup имеет ограниченную поддержку селекторов CSS , но охватывает наиболее часто используемые. Используйте метод `select()` чтобы найти несколько элементов и `select_one()` чтобы найти один элемент.

Основной пример:

```
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())
```

Печать:

```
item1
item2
item3
```

PyQuery

pyquery - jQuery-подобная библиотека для python. Он очень хорошо поддерживает селекторов 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)
```

Прочитайте HTML-анализ онлайн: https://riptutorial.com/ru/python/topic/1384/html-анализ

глава 12: ijson

Вступление

ijson - отличная библиотека для работы с файлами JSON в Python. К сожалению, по умолчанию в качестве бэкэнд использует чистый анализатор Python JSON. Гораздо более высокая производительность может быть достигнута за счет использования C-сервера.

Examples

Простой пример

Пример Пример, взятый из одного бенчмаркинга

```
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

Прочитайте ijson онлайн: https://riptutorial.com/ru/python/topic/8342/ijson

глава 13: kivy - кроссплатформенная платформа Python для разработки NUI

Вступление

NUI: Естественный пользовательский интерфейс (NUI) - это система взаимодействия человека и компьютера, которую пользователь осуществляет посредством интуитивных действий, связанных с естественным повседневным поведением людей.

Kivy - это библиотека Python для разработки мультимедийных мультимедийных приложений с поддержкой multi-touch, которые могут быть установлены на разных устройствах. Multi-touch относится к способности сенсорной поверхности (обычно сенсорного экрана или трекпада) для обнаружения или распознавания ввода из двух или нескольких точек контакта одновременно.

Examples

Первое приложение

Чтобы создать приложение для kivy

1. подкласс класса **приложения**
2. Внедрите метод **сборки** , который вернет виджет.
3. Инстанцировать класс вызвать **запуск.**

```
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()
```

объяснение

```
from kivy.app import App
```

Вышеприведенный оператор импортирует **приложение** родительского класса. Это будет присутствовать в вашем каталоге установки your_installtion_directory / kivy / app.py

```
from kivy.uix.label import Label
```

Вышеприведенный оператор импортирует **метку** ux-элемента. Весь элемент ux присутствует в вашем каталоге установки your_installation_directory / kivy / uix /.

```
class Test(App):
```

Вышеприведенное заявление предназначено для создания вашего приложения, а имя класса будет вашим именем приложения. Этот класс наследуется родительским классом приложения.

```
def build(self):
```

Вышеприведенный оператор переопределяет метод сборки класса приложения. Что вернет виджет, который нужно отобразить, когда вы запустите приложение.

```
return Label(text='Hello world')
```

Вышеприведенный оператор является телом метода построения. Он возвращает ярлык со своим текстом **Hello world** .

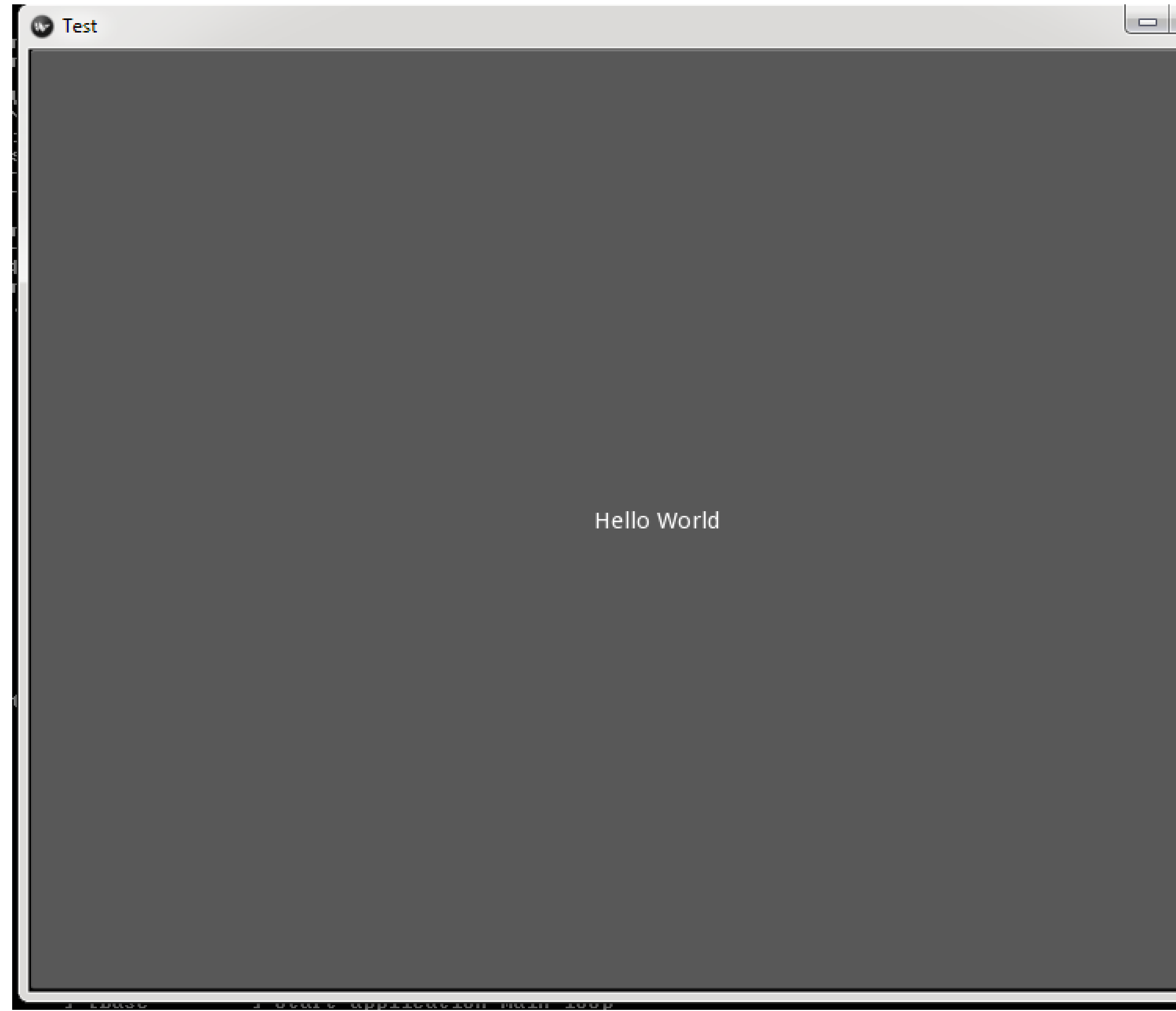
```
if __name__ == '__main__':
```

Вышеприведенный оператор является точкой входа, из которой интерпретатор python начинает выполнение вашего приложения.

```
Test().run()
```

Вышеприведенный оператор Инициализируйте свой тестовый класс, создав его экземпляр. И вызовите функцию класса приложения run ().

Ваше приложение будет выглядеть следующим образом.



Прочитайте kivy - кроссплатформенная платформа Python для разработки NUI онлайн: https://riptutorial.com/ru/python/topic/10743/kivy---кроссплатформенная-платформа-python-для-разработки-nui

глава 14: Loops

Вступление

Как одна из самых основных функций в программировании, петли - важная часть почти для каждого языка программирования. Петли позволяют разработчикам устанавливать определенные части своего кода для повторения через несколько циклов, которые называются итерациями. В этом разделе описывается использование нескольких типов циклов и приложений циклов в Python.

Синтаксис

- while <boolean expression>:
- для <variable> в <iterable>:
- для <variable> в диапазоне (<число>):
- для <variable> в диапазоне (<start_number>, <end_number>):
- для <variable> в диапазоне (<start_number>, <end_number>, <step_size>):
- для i, <variable> в перечислении (<iterable>): # с индексом i
- для <variable1>, <variable2> в zip (<iterable1>, <iterable2>):

параметры

параметр	подробности
логическое выражение	выражение, которое может быть оценено в булевом контексте, например `x < 10`
переменная	имя переменной для текущего элемента из `iterable`
итерируемый	все, что реализует итерации

Examples

Итерирование по спискам

Чтобы перебрать список, вы можете использовать `for` :

```
for x in ['one', 'two', 'three', 'four']:
    print(x)
```

Это напечатает элементы списка:

```
one
two
three
four
```

Функция `range` генерирует числа, которые также часто используются в цикле for.

```
for x in range(1, 6):
    print(x)
```

Результатом будет специальный тип последовательности диапазона в python> = 3 и список в python <= 2. Оба могут быть закодированы с использованием цикла for.

```
1
2
3
4
5
```

Если вы хотите зациклиться на обоих элементах списка *и* иметь индекс для этих элементов, вы можете использовать функцию `enumerate` Python:

```
for index, item in enumerate(['one', 'two', 'three', 'four']):
    print(index, '::', item)
```

`enumerate` будет генерировать кортежи, которые распаковываются в `index` (целое число) и `item` (фактическое значение из списка). Вышеприведенный цикл распечатает

```
(0, '::', 'one')
(1, '::', 'two')
(2, '::', 'three')
(3, '::', 'four')
```

Перейдем к списку с манипуляцией значениями с использованием `map` и `lambda` , т. Е. Примените лямбда-функцию для каждого элемента в списке:

```
x = map(lambda e :  e.upper(), ['one', 'two', 'three', 'four'])
print(x)
```

Выход:

```
['ONE', 'TWO', 'THREE', 'FOUR'] # Python 2.x
```

NB: на `map` Python 3.x вместо итератора вместо списка появляется итератор, поэтому если вам нужен список, вы должны указать результат `print(list(x))` (см. Http://www.riptutorial.com/python/ пример / 8186 / map-- в http://www.riptutorial.com/python/topic/809/incompatabilities-moving-from-python-2-to-python-3).

Для петель

`for` циклов итерации по набору элементов, таких как `list` или `dict` , и запускать блок кода с каждым элементом из коллекции.

```
for i in [0, 1, 2, 3, 4]:
    print(i)
```

Вышеприведенное `for` цикла повторяется над списком чисел.

Каждая итерация устанавливает значение `i` в следующий элемент списка. Итак, сначала это будет `0` , затем `1` , затем `2` и т. Д. Выход будет следующим:

```
0
1
2
3
4
```

`range` - это функция, которая возвращает серию чисел под итерируемой формой, поэтому ее можно использовать `for` циклов `for` :

```
for i in range(5):
    print(i)
```

дает тот же результат, что первый `for` цикл. Обратите внимание, что `5` не печатается, так как диапазон здесь - это первые пять чисел, считанных с `0` .

Итерируемые объекты и итераторы

`for` loop может выполнять итерацию по любому итерируемому объекту, который является объектом, который определяет функцию `__getitem__` или `__iter__` . Функция `__iter__` возвращает итератор, который является объектом со `next` функцией, которая используется для доступа к следующему элементу итерабельного.

Перерыв и продолжение в циклах

оператор `break`

Когда оператор `break` выполняется внутри цикла, поток управления «вырывается» из цикла немедленно:

```
i = 0
while i < 7:
```

```
    print(i)
    if i == 4:
        print("Breaking from loop")
        break
    i += 1
```

Условие цикла не будет оцениваться после выполнения инструкции `break` . Обратите внимание, что инструкции `break` допускаются только *внутри циклов* , синтаксически. Оператор `break` внутри функции не может использоваться для завершения циклов, которые вызывали эту функцию.

Выполнение следующих отпечатков каждой цифры до номера `4` когда выполняется оператор `break` и цикл останавливается:

```
0
1
2
3
4
Breaking from loop
```

операторы `break` также могут использоваться внутри `for` циклов, другая петлевая конструкция, предоставляемая Python:

```
for i in (0, 1, 2, 3, 4):
    print(i)
    if i == 2:
        break
```

Выполнение этого цикла теперь печатает:

```
0
1
2
```

Обратите внимание: 3 и 4 не печатаются, так как цикл закончился.

Если цикл имеет предложение `else` , он не выполняется, когда цикл завершается через оператор `break` .

`continue` заявление

Оператор `continue` пропустит следующую итерацию цикла, минуя остальную часть текущего блока, но продолжая цикл. Как и при `break` , `continue` может появляться только внутри циклов:

```
for i in (0, 1, 2, 3, 4, 5):
    if i == 2 or i == 4:
```

```
        continue
    print(i)

0
1
3
5
```

Обратите внимание: `2` и `4` не печатаются, потому что `continue` переходит к следующей итерации вместо продолжения `print(i)` когда `i == 2` или `i == 4` .

Вложенные петли

`break` и `continue` работать только на одном уровне цикла. Следующий пример сломается только из внутренних `for` цикла, а не внешние , `while` петель:

```
while True:
    for i in range(1,5):
        if i == 2:
            break    # Will only break out of the inner loop!
```

Python не имеет возможности вырваться из нескольких уровней цикла сразу - если это необходимо, рефакторинг одной или нескольких петель в функцию и замена `break` с `return` может быть способом.

Использовать `return` из функции в виде `break`

Оператор `return` выходит из функции, не выполняя код, который появляется после него.

Если у вас есть цикл внутри функции, использование `return` изнутри этого цикла эквивалентно `break` поскольку остальная часть кода цикла не выполняется (*обратите внимание, что любой код после цикла не выполняется*):

```
def break_loop():
    for i in range(1, 5):
        if (i == 2):
            return(i)
        print(i)
    return(5)
```

Если у вас есть вложенные циклы, оператор `return` будет разбивать все циклы:

```
def break_all():
    for j in range(1, 5):
        for i in range(1,4):
            if i*j == 6:
                return(i)
            print(i*j)
```

выведет:

```
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
```

Циклы с предложением «else»

`for` и в `while` операторы составных (петля) могут дополнительно иметь `else` пункт (на практике, это использование довольно редко).

Предложение `else` выполняется только после того, как цикл `for` завершается путем итерации до завершения или после того, `while` цикл while завершит его условное выражение, становясь ложным.

```
for i in range(3):
    print(i)
else:
    print('done')

i = 0
while i < 3:
    print(i)
    i += 1
else:
    print('done')
```

выход:

```
 0
 1
 2
 done
```

Предложение `else` *не* выполняется, если цикл завершает другой путь (через оператор `break` или путем создания исключения):

```
for i in range(2):
    print(i)
    if i == 1:
        break
else:
    print('done')
```

выход:

```
0
1
```

Большинство других языков программирования не хватает этого необязательного `else` положение петель. Использование ключевого слова `else` , в частности , часто считается запутанным.

Первоначальная концепция такого предложения восходит к Дональду Кнуту, и значение ключевого слова `else` становится ясным, если мы переписываем цикл в терминах операторов `if` и `goto` с предыдущих дней до структурированного программирования или с языка ассемблера более низкого уровня.

Например:

```
while loop_condition():
    ...
    if break_condition():
        break
    ...
```

эквивалентно:

```
# pseudocode

<<start>>:
if loop_condition():
    ...
    if break_condition():
        goto <<end>>
    ...
    goto <<start>>

<<end>>:
```

Они остаются эквивалентными, если мы присоединяем условие `else` к каждому из них.

Например:

```
while loop_condition():
    ...
    if break_condition():
        break
    ...
else:
    print('done')
```

эквивалентно:

```
# pseudocode

<<start>>:
if loop_condition():
    ...
    if break_condition():
        goto <<end>>
    ...
```

```
    goto <<start>>
else:
    print('done')

<<end>>:
```

То же, `for` цикл `for` с предложением `else` . Концептуально существует условие цикла, которое остается True, пока итерируемый объект или последовательность все еще имеют некоторые оставшиеся элементы.

Зачем использовать эту странную конструкцию?

Основным вариантом использования for `for...else` construct является краткая реализация поиска, например:

```
a = [1, 2, 3, 4]
for i in a:
    if type(i) is not int:
        print(i)
        break
else:
    print("no exception")
```

Чтобы сделать `else` в этой конструкции менее запутанным, можно подумать об этом как « *если не сломать* » или « *если не найдено* ».

Некоторые обсуждения по этому вопросу можно найти в [Python-ideas] Summary of for ... else threads. Почему python использует «else» после циклов for и while? , и Else Clauses on Loop Statement

Итерация над словарями

Учитывая следующий словарь:

```
d = {"a": 1, "b": 2, "c": 3}
```

Чтобы выполнить итерацию через свои ключи, вы можете использовать:

```
for key in d:
    print(key)
```

Выход:

```
"a"
"b"
"c"
```

Это эквивалентно:

```
for key in d.keys():
    print(key)
```

или в Python 2:

```
for key in d.iterkeys():
    print(key)
```

Для повторения своих значений используйте:

```
for value in d.values():
    print(value)
```

Выход:

```
1
2
3
```

Чтобы итератировать свои ключи и значения, используйте:

```
for key, value in d.items():
    print(key, "::", value)
```

Выход:

```
a :: 1
b :: 2
c :: 3
```

Обратите внимание, что в Python 2, `.keys()` , `.values()` и `.items()` возвращают объект `list` . Если вам просто нужно повторить результат, вы можете использовать эквивалентные `.iterkeys()` , `.itervalues()` и `.iteritems()` .

Разница между `.keys()` и `.iterkeys()` , `.values()` и `.itervalues()` , `.items()` и `.iteritems()` заключается в том, что методы `iter*` являются генераторами. Таким образом, элементы в словаре вводятся один за другим по мере их оценки. Когда объект `list` возвращается, все элементы упаковываются в список и затем возвращаются для дальнейшей оценки.

> Обратите также внимание на то, что в Python 3 порядок элементов, напечатанных указанным выше способом, не соответствует никакому порядку.

Пока цикл

В `while` цикл будет вызывать операторы цикла будет выполняться , пока условие цикла не falsey . Следующий код будет выполнять инструкции цикла в 4 раза.

```
i = 0
while i < 4:
    #loop statements
    i = i + 1
```

В то время как выше петли легко могут быть переведены в более элегантный `for` цикла, в `while` петли полезны для проверки , если какое - либо условие было выполнено. Следующий цикл продолжит выполнение до тех пор, пока `myObject` будет готов.

```
myObject = anObject()
while myObject.isNotReady():
    myObject.tryToGetReady()
```

`while` циклы также могут работать без условия, используя числа (сложные или реальные) или `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
```

Если условие всегда верно, цикл while будет выполняться вечно (бесконечный цикл), если он не завершен оператором break или return или исключением.

```
while True:
    print "Infinite loop"
# Infinite loop
# Infinite loop
# Infinite loop
# ...
```

Заявление о прохождении

`pass` является пустым оператором для того, когда оператор требует синтаксис Python (например, в пределах тела `for` или в `while` цикла), но никакие действий не требуются или желательны программист. Это может быть полезно в качестве заполнителя для кода, который еще не написан.

```
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
```

В этом примере ничего не произойдет. Цикл `for` будет завершен без ошибок, но никакие команды или код не будут действовать. `pass` позволяет нам успешно запускать наш код без полного выполнения всех команд и действий.

Аналогичный образом , `pass` может быть использована в `while` петли, а также в выборе и определении функций и т.д.

```
while x == y:
    pass
```

Итерирование другой части списка с разным размером шага

Предположим, у вас длинный список элементов, и вас интересует только каждый элемент списка. Возможно, вы хотите просмотреть только первый или последний элементы или определенный диапазон записей в своем списке. У Python есть сильные индексирующие встроенные возможности. Вот несколько примеров того, как достичь этих сценариев.

Вот простой список, который будет использоваться во всех примерах:

```
lst = ['alpha', 'bravo', 'charlie', 'delta', 'echo']
```

Итерация по всему списку

Для итерации по каждому элементу в списке может использоваться цикл `for` как показано ниже:

```
for s in lst:
    print s[:1] # print the first letter
```

Цикл `for` присваивает s для каждого элемента `lst` . Это напечатает:

```
a
b
c
d
e
```

Часто вам нужен как элемент, так и индекс этого элемента. Ключевое слово `enumerate` выполняет эту задачу.

```
for idx, s in enumerate(lst):
    print("%s has an index of %d" % (s, idx))
```

Индекс `idx` будет начинаться с нуля и приращения для каждой итерации, в то время как `s` будет содержать элемент обрабатывается. Вывод предыдущего фрагмента:

```
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
```

Итерация по подменю

Если мы хотим итерации по диапазону (помня, что Python использует индексирование с нулевым индексом), используйте ключевое слово `range` .

```
for i in range(2,4):
    print("lst at %d contains %s" % (i, lst[i]))
```

Это приведет к выводу:

```
lst at 2 contains charlie
lst at 3 contains delta
```

Список также можно нарезать. Следующая нотация фрагмента идет от элемента с индексом 1 до конца с шагом 2. Два `for` циклов дают одинаковый результат.

```
for s in lst[1::2]:
    print(s)

for i in range(1, len(lst), 2):
    print(lst[i])
```

Вышеприведенные фрагменты:

```
bravo
delta
```

Индексирование и нарезка - это отдельная тема.

«Half loop» do-while

В отличие от других языков, Python не имеет конструкцию do-until или do-while (это позволит выполнить код один раз до проверки условия). Однако вы можете комбинировать `while True` с `break` для достижения той же цели.

```
a = 10
while True:
    a = a-1
    print(a)
    if a<7:
        break
print('Done.')
```

Это напечатает:

```
9
8
7
6
Done.
```

Цикл и распаковка

Если вы хотите перебрать список кортежей, например:

```
collection = [('a', 'b', 'c'), ('x', 'y', 'z'), ('1', '2', '3')]
```

вместо того, чтобы делать что-то вроде этого:

```
for item in collection:
    i1 = item[0]
    i2 = item[1]
    i3 = item[2]
    # logic
```

или что-то вроде этого:

```
for item in collection:
    i1, i2, i3 = item
    # logic
```

Вы можете просто сделать это:

```
for i1, i2, i3 in collection:
    # logic
```

Это также будет работать для *большинства* типов итераций, а не только для кортежей.

Прочитайте Loops онлайн: https://riptutorial.com/ru/python/topic/237/loops

глава 15: Mutable vs Immutable (и Hashable) в Python

Examples

Mutable vs Immutable

В Python существует два типа типов. Неизменяемые типы и изменяемые типы.

Immutables

Объект неизменяемого типа нельзя изменить. Любая попытка изменить объект приведет к созданию копии.

Эта категория включает в себя: целые числа, поплавки, сложные, строки, байты, кортежи, диапазоны и фризодаты.

Чтобы выделить это свойство, давайте поиграем с встроенным `id` . Эта функция возвращает уникальный идентификатор объекта, переданного как параметр. Если идентификатор один и тот же, это тот же объект. Если он изменится, это другой объект. *(Некоторые говорят, что это на самом деле адрес памяти объекта, но остерегайтесь их, они из темной стороны силы ...)*

```
>>> a = 1
>>> id(a)
140128142243264
>>> a += 2
>>> a
3
>>> id(a)
140128142243328
```

Ладно, 1 не 3 ... Ломать новости ... Может быть, нет. Однако это поведение часто забывается, когда речь идет о более сложных типах, особенно в строках.

```
>>> stack = "Overflow"
>>> stack
'Overflow'
>>> id(stack)
140128123955504
>>> stack += " rocks!"
>>> stack
'Overflow rocks!'
```

Ага! Увидеть? Мы можем изменить его!

```
>>> id(stack)
140128123911472
```

Нет. Хотя кажется, что мы можем изменить строку, названную `stack` переменных, то, что мы на самом деле делаем, создает новый объект, содержащий результат конкатенации. Мы обманываем, потому что в процессе старый объект никуда не уходит, поэтому он уничтожается. В другой ситуации это было бы более очевидно:

```
>>> stack = "Stack"
>>> stackoverflow = stack + "Overflow"
>>> id(stack)
140128069348184
>>> id(stackoverflow)
140128123911480
```

В этом случае ясно, что если мы хотим сохранить первую строку, нам нужна копия. Но разве это так очевидно для других типов?

Упражнение

Теперь, зная, как работают неизменные типы, что бы вы сказали с помощью кода ниже? Это мудро?

```
s = ""
for i in range(1, 1000):
    s += str(i)
    s += ","
```

Mutables

Объект изменчивого типа может быть изменен и изменен *на месте* . Никаких неявных копий не производится.

В эту категорию входят: списки, словари, bytearrays и sets.

Давайте продолжим играть с нашей маленькой функцией `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
```

(В качестве побочного примечания я использую байты, содержащие данные ascii, чтобы сделать мой вопрос ясным, но помните, что байты не предназначены для хранения текстовых данных. Молитесь, простите меня.)

Что у нас есть? Мы создаем bytearray, модифицируем его и используем `id` , мы можем гарантировать, что это один и тот же объект, модифицированный. Не копия.

Конечно, если объект будет часто изменяться, изменяемый тип выполняет гораздо лучшую работу, чем неизменяемый тип. К сожалению, реальность этого свойства часто забывается, когда это больно больше всего.

```
>>> c = b
>>> c += b' rocks!'
>>> c
bytearray(b'StackOverflow rocks!')
```

Хорошо...

```
>>> b
bytearray(b'StackOverflow rocks!')
```

Вайит второй ...

```
>>> id(c) == id(b)
True
```

В самом деле. `c` не является копией `b` . `c` - `b` .

Упражнение

Теперь вам лучше понять, какой побочный эффект подразумевается изменчивым типом, можете ли вы объяснить, что в этом примере происходит неправильно?

```
>>> 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...
```

Mutable and Immutable как аргументы

Одним из основных вариантов использования, когда разработчику необходимо учитывать изменчивость, является передача аргументов функции. Это очень важно, потому что это определит способность функции изменять объекты, которые не относятся к ее области действия, или, другими словами, если функция имеет побочные эффекты. Это также

важно понять, где должен быть доступен результат функции.

```
>>> 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]
```

Здесь ошибка заключается в том, что `lin` , как параметр функции, может быть изменен локально. Вместо этого, `lin` и ссылка тот же объект. `a` Поскольку этот объект изменен, модификация выполняется на месте, что означает, что объект, на который ссылаются как `lin` и `a` , изменяется. `lin` действительно не нужно возвращать, потому что у нас уже есть ссылка на этот объект в виде `a` . `a` и `b` ссылающиеся на один и тот же объект.

Это не похоже на кортежи.

```
>>> 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)
```

В начале функции, `tin` и `a` качестве ссылки и тот же объект. Но это непреложный объект. Поэтому, когда функция пытается ее изменить, `tin` получает новый объект с модификацией, а `a` сохраняет ссылку на исходный объект. В этом случае возврат `tin` является обязательным, или новый объект будет потерян.

Упражнение

```
>>> def yoda(prologue, sentence):
    sentence.reverse()
    prologue += " ".join(sentence)
    return prologue

>>> focused = ["You must", "stay focused"]
>>> saying = "Yoda said: "
>>> yoda_sentence = yoda(saying, focused)
```

Примечание: `reverse` работает на месте.

Что вы думаете об этой функции? Имеет ли он побочные эффекты? Нужно ли

возвращение? После звонка, какова ценность `saying` ? `focused` ? Что произойдет, если функция снова вызвана с теми же параметрами?

Прочитайте Mutable vs Immutable (и Hashable) в Python онлайн: https://riptutorial.com/ru/python/topic/9182/mutable-vs-immutable--и-hashable--в-python

глава 16: Neo4j и Cypher используют Py2Neo

Examples

Импорт и аутентификация

```
from py2neo import authenticate, Graph, Node, Relationship
authenticate("localhost:7474", "neo4j", "<pass>")
graph = Graph()
```

Вы должны убедиться, что ваша база данных Neo4j существует на localhost: 7474 с соответствующими учетными данными.

объект `graph` - это ваш интерфейс к экземпляру neo4j в остальной части вашего кода на Python. Скорее спасибо, что это глобальная переменная, вы должны сохранить ее в методе `__init__` класса.

Добавление узлов в Neo4j Graph

```
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()
    [...]
```

Добавление узлов в график довольно просто, `graph.merge_one` важна, так как она предотвращает дублирование элементов. (Если вы дважды запускаете сценарий, то во второй раз он будет обновлять заголовок, а не создавать новые узлы для тех же статей)

`timestamp` должен быть целым числом, а не строкой даты, поскольку neo4j действительно не имеет типа даты. Это вызывает проблемы с сортировкой при сохранении даты как '05 -06-1989 '

`article.push()` - это вызов, который фактически совершает операцию в neo4j. Не забывайте этот шаг.

Добавление отношений к графику 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` важно для избежания дубликатов. Но в остальном это довольно простая операция. Имя отношения также важно, так как вы использовали бы его в расширенных случаях.

Запрос 1: автозаполнение заголовков новостей

```
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
```

Это пример запроса cypher для получения всех узлов с `name` свойства, которое начинается с `text` аргумента.

Запрос 2: получать новости по местоположению в определенную дату

```
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
```

Вы можете использовать этот запрос, чтобы найти все новостные статьи `(n)` связанные с местоположением `(l)` с помощью отношения.

Образцы запросов Cypher

Подсчитайте статьи, связанные с определенным человеком со временем

```
MATCH (n)-[]->(l)
```

```
where l.name='Donald Trump'
RETURN n.date,count(*) order by n.date
```

Поиск других людей / мест, связанных с такими же новостными статьями, как Trump с не менее чем 5 точками отношений.

```
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
```

Прочитайте Neo4j и Cypher используют Py2Neo онлайн: https://riptutorial.com/ru/python/topic/5841/neo4j-и-cypher-используют-py2neo

глава 17: os.path

Вступление

Этот модуль реализует некоторые полезные функции для путей. Параметры пути могут передаваться как строки, так и байты. Приложениям рекомендуется представлять имена файлов в виде (Unicode) символьных строк.

Синтаксис

- os.path.join (a, * p)
- os.path.basename (p)
- os.path.dirname (p)
- os.path.split (p)
- os.path.splitext (p)

Examples

Присоединительные пути

Чтобы объединить два или более компонентов пути вместе, сначала импортируйте os-модуль python, а затем используйте следующее:

```
import os
os.path.join('a', 'b', 'c')
```

Преимущество использования os.path заключается в том, что он позволяет коду оставаться совместимым во всех операционных системах, поскольку в нем используется разделитель, подходящий для платформы, на которой он работает.

Например, результатом этой команды в Windows будет:

```
>>> os.path.join('a', 'b', 'c')
'a\b\c'
```

В ОС Unix:

```
>>> os.path.join('a', 'b', 'c')
'a/b/c'
```

Абсолютный путь от относительного пути

Использовать `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'
```

Манипуляция компонентов пути

Чтобы разбить один компонент на путь:

```
>>> 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')
```

Получить родительский каталог

```
os.path.abspath(os.path.join(PATH_TO_GET_THE_PARENT, os.pardir))
```

Если данный путь существует.

проверить, существует ли данный путь

```
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
```

проверьте, является ли данный путь каталогом, файлом, символической ссылкой, точкой монтирования и т. д.

проверить, является ли данный путь каталогом

```
dirname = '/home/john/python'
os.path.isdir(dirname)
```

чтобы проверить, является ли данный путь файлом

```
filename = dirname + 'main.py'
os.path.isfile(filename)
```

чтобы проверить, является ли данный путь символической ссылкой

```
symlink = dirname + 'some_sym_link'
os.path.islink(symlink)
```

чтобы проверить, является ли данный путь точкой монтирования

```
mount_path = '/home'
os.path.ismount(mount_path)
```

Прочитайте os.path онлайн: https://riptutorial.com/ru/python/topic/1380/os-path

глава 18: Pandas Transform: операции надготовкой по группам и конкатенация результатов

Examples

Простое преобразование

Сначала создадим фиктивный фрейм

Мы предполагаем, что заказчик может иметь n заказов, заказ может иметь m элементов, а элементы можно заказать несколько раз

```
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
```

,
,

Теперь мы будем использовать функцию `transform` pandas для подсчета количества заказов на клиента

```
# 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
```

Несколько результатов для каждой группы

Использование функций `transform` , возвращающих подвычисления на группу

В предыдущем примере у нас был один результат для каждого клиента. Однако также могут применяться функции, возвращающие разные значения для группы.

```
# 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
```

Прочитайте Pandas Transform: операции надготовкой по группам и конкатенация результатов онлайн: https://riptutorial.com/ru/python/topic/10947/pandas-transform--операции-надготовкой-по-группам-и-конкатенация-результатов

глава 19: pip: Менеджер пакетов PyPI

Вступление

pip является наиболее широко используемым менеджером пакетов для индекса пакетов Python, установленным по умолчанию в последних версиях Python.

Синтаксис

- pip <command> [options], где <command> является одним из следующих:
  - устанавливать
    - Установка пакетов
  - деинсталляция
    - Удаление пакетов
  - замерзать
    - Выдавать установленные пакеты в формате требований
  - список
    - Список установленных пакетов
  - шоу
    - Показать информацию об установленных пакетах
  - поиск
    - Поиск PyPI для пакетов
  - рулевое колесо
    - Постройте колеса от ваших требований
  - застежка-молния
    - Почтовые индивидуальные пакеты (устаревшие)
  - расстегнуть молнию
    - Распакуйте отдельные пакеты (устаревшие)
  - сверток
    - Создание pybundles (устарело)
  - Помогите
    - Показать справку для команд

замечания

Иногда, `pip` будет выполнять ручную компиляцию собственного кода. В Linux python автоматически выберет доступный компилятор C в вашей системе. Обратитесь к приведенной ниже таблице для требуемой версии Visual Studio / Visual C ++ в Windows (новые версии не будут работать.).

Версия Python	Версия Visual Studio	Версия Visual C ++
2.6 - 3.2	Visual Studio 2008	Visual C ++ 9.0
3,3 - 3,4	Visual Studio 2010	Visual C ++ 10.0
3,5	Visual Studio 2015	Visual C ++ 14.0

Источник: wiki.python.org

Examples

Установить пакеты

Чтобы установить последнюю версию пакета с именем `SomePackage` :

```
$ pip install SomePackage
```

Чтобы установить определенную версию пакета:

```
$ pip install SomePackage==1.0.4
```

Чтобы указать минимальную версию для установки для пакета:

```
$ pip install SomePackage>=1.0.4
```

Если команды показывают разрешенную ошибку в Linux / Unix, тогда используйте `sudo` с командами

Установить из файлов требований

```
$ pip install -r requirements.txt
```

Каждая строка файла требований указывает что-то, что нужно установить, и как аргументы для установки pip. Подробности о форматах файлов приведены здесь: Формат файла требований .

После установки пакета вы можете проверить его с помощью команды `freeze` :

```
$ pip freeze
```

Удаление пакетов

Чтобы удалить пакет:

```
$ pip uninstall SomePackage
```

Чтобы просмотреть все пакеты, установленные с помощью `pip`

Чтобы указать установленные пакеты:

```
$ pip list
# example output
docutils (0.9.1)
Jinja2 (2.6)
Pygments (1.5)
Sphinx (1.1.2)
```

Чтобы просмотреть устаревшие пакеты и показать последнюю доступную версию:

```
$ pip list --outdated
# example output
docutils (Current: 0.9.1 Latest: 0.10)
Sphinx (Current: 1.1.2 Latest: 1.1.3)
```

Обновление пакетов

Бег

```
$ pip install --upgrade SomePackage
```

обновит пакет `SomePackage` и все его зависимости. Кроме того, pip автоматически удаляет старую версию пакета перед обновлением.

Чтобы обновить сам пипс, сделайте

```
$ pip install --upgrade pip
```

на Unix или

```
$ python -m pip install --upgrade pip
```

на машинах Windows.

Обновление всех устаревших пакетов в Linux

`pip` не содержит флаг, позволяющий пользователю обновлять все устаревшие пакеты за один снимок. Однако это может быть достигнуто с помощью команд piping вместе в среде Linux:

```
pip list --outdated --local | grep -v '^\-e' | cut -d = -f 1  | xargs -n1 pip install -U
```

Эта команда принимает все пакеты в локальном virtualenv и проверяет, устарели ли они. Из этого списка он получает имя пакета, а затем `pip install -U` команде `pip install -U` . В конце этого процесса необходимо обновить все локальные пакеты.

Обновление всех устаревших пакетов в Windows

`pip` не содержит флаг, позволяющий пользователю обновлять все устаревшие пакеты за один снимок. Однако это можно выполнить с помощью команд piping в среде Windows:

```
for /F "delims= " %i in ('pip list --outdated --local') do pip install -U %i
```

Эта команда принимает все пакеты в локальном virtualenv и проверяет, устарели ли они. Из этого списка он получает имя пакета, а затем `pip install -U` команде `pip install -U` . В конце этого процесса необходимо обновить все локальные пакеты.

Создайте файл requirements.txt всех пакетов в системе

`pip` помогает создавать файлы `requirements.txt` , предоставляя опцию `freeze` .

```
pip freeze > requirements.txt
```

Это сохранит список всех пакетов и их версию, установленных в системе, в файл с именем `requirements.txt` в текущей папке.

Создайте файл requirements.txt пакетов только в текущем virtualenv

`pip` помогает создавать файлы `requirements.txt` , предоставляя опцию `freeze` .

```
pip freeze --local > requirements.txt
```

Параметр `--local` только список пакетов и версий, которые локально устанавливаются в virtualenv. Глобальные пакеты не будут перечислены.

Использование определенной версии Python с помощью pip

Если у вас установлены оба Python 3 и Python 2, вы можете указать, какую версию Python вы хотите использовать для использования. Это полезно, если пакеты поддерживают только Python 2 или 3, или когда вы хотите протестировать оба.

Если вы хотите установить пакеты для Python 2, запустите либо:

```
pip install [package]
```

или же:

```
pip2 install [package]
```

Если вы хотите установить пакеты для Python 3, выполните следующие действия:

```
pip3 install [package]
```

Вы также можете вызвать установку пакета на определенную установку python с помощью:

```
\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
```

На платформах OS-X / Linux / Unix важно знать разницу между системной версией python (которая позволяет сделать рендеринг вашей операционной системой неработоспособной) и версией (-ами) пользователя python. Вы **можете** , в *зависимости от того, что вы пытаетесь обновить* , должны префикс этих команд с помощью `sudo` и ввода пароля.

Аналогично, в Windows некоторые установки python, особенно те, которые являются частью другого пакета, могут быть установлены в системных каталогах - те, которые вам придется обновить из командной строки, запущенной в режиме администратора, - если вы обнаружите, что это похоже на то, что вам нужно сделайте это, **очень** хорошо проверить, какую установку python вы пытаетесь обновить с помощью команды `python -c"import sys;print(sys.path);"` или `py -3.5 -c"import sys;print(sys.path);"` вы также можете проверить, какой пип вы пытаетесь запустить с помощью `pip --version`

В Windows, если у вас есть как Python 2 и установлен Python 3, и на вашем пути и вашего питона 3 больше , чем 3,4 , то вы, вероятно , также есть питон пусковая `py` на вашем системном пути. Затем вы можете делать трюки:

```
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
```

Если вы работаете и поддерживаете несколько версий python, я бы настоятельно рекомендовал ознакомиться с виртуальными виртуальными `venv` python `virtualenv` или `venv` которые позволяют вам изолировать как версию python, так и какие пакеты присутствуют.

Установка пакетов, еще не входящих в качестве колес

Многие, чистые python, пакеты пока недоступны в индексе пакета Python в качестве колес, но все еще устанавливаются нормально. Тем не менее, некоторые пакеты в Windows приводят к ужасной ошибке vcvarsall.bat.

Проблема в том, что пакет, который вы пытаетесь установить, содержит расширение C или C ++ и в *настоящее время* недоступно как предварительно построенное колесо из индекса пакета python, *pypi* и на окнах у вас нет цепочки инструментов, необходимой для сборки такие предметы.

Самый простой ответ - перейти на превосходный сайт Кристофа Гольке и найти **подходящую** версию библиотек, которые вам нужны. Соответственно в названии пакета а **-cp *NN* -** должно соответствовать вашей версии python, то есть, если вы используете Windows-битовый python *даже на win64,* имя должно содержать **-win32-,** и если использовать 64-битный питон, он должен включать **-win_amd64 -** а затем версия python должна соответствовать, то есть для Python 34 имя файла **должно** содержать **-cp *34-*** и т. д., это в основном магия, которую пип делает для вас на сайте pypi.

В качестве альтернативы вам необходимо получить соответствующий комплект для разработки Windows для используемой версии python, заголовки для любой библиотеки, к которой пакет пытается создавать интерфейсы, возможно, заголовки python для версии python и т. Д.

Python 2.7 использовал Visual Studio 2008, Python 3.3 и 3.4, используемые Visual Studio 2010, а Python 3.5+ использует Visual Studio 2015.

- Установите « Компилятор Visual C ++ для Python 2.7 », который доступен на веб-сайте Microsoft **или**
- Установите « Windows SDK для Windows 7 и .NET Framework 4 » (v7.1), который доступен на веб-сайте Microsoft **или**
- Установите Visual Studio 2015 Community Edition *(или любую более позднюю версию, когда они будут выпущены)* , **гарантируя, что вы выберете варианты установки поддержки C & C ++** *больше не по умолчанию - мне сказали, что* для загрузки и установки *может потребоваться до **8 часов*** поэтому убедитесь **,** что эти параметры установлены с первой попытки.

Тогда вам *может потребоваться* найти файлы заголовков *в соответствующей ревизии* для любых библиотек, к которым вы хотите подключить нужные ссылки, и загрузить их в соответствующие места.

Наконец, вы можете позволить pip сделать свою сборку - конечно, если у пакета есть зависимости, которые у вас еще нет, вам также может понадобиться найти файлы заголовков для них.

Альтернативы. Также стоит посмотреть, *как на pypi* , *так и на сайте Christop's* , для любой более ранней версии пакета, который вы ищете, это либо чистый питон, либо готовый для вашей платформы и версии python, и, возможно, с помощью тех, если , пока ваш пакет не станет доступен. Аналогично, если вы используете самую последнюю версию python, вы можете обнаружить, что разработчикам пакетов требуется немного времени, чтобы наверстать упущенное, поэтому для проектов, которым действительно **нужен**

определенный пакет, вам, возможно, придется использовать немного более старый python на данный момент. Вы также можете проверить исходный сайт пакетов, чтобы узнать, есть ли разветвленная версия, которая доступна заранее или как чистый питон, и поиск альтернативных пакетов, которые предоставляют требуемые функциональные возможности, но доступны. Один из примеров, который приходит на ум, - это Подушка , *активно поддерживается* , заменяет PIL, который в *настоящее время не обновляется через 6 лет и недоступен для python 3* .

Послесловие , я призываю любого, у кого есть эта проблема, перейти к отладчику ошибок для пакета и добавить к нему или поднять, если его еще нет, билет **вежливо** просит, чтобы сопровождающие пакета предоставили колесо на pypi для вашего конкретного сочетание платформы и python, если это будет сделано, то, как правило, со временем все будет лучше, некоторые сторонники пакетов не поймут, что они пропустили данную комбинацию, которую могут использовать люди.

Примечание по установке предварительных выпусков

Pip следует правилам Semantic Versioning и по умолчанию предпочитает выпущенные пакеты по предварительным выпускам. Поэтому, если данный пакет был выпущен как `V0.98` и есть также кандидат на выпуск `V1.0-rc1` поведение по умолчанию для `pip install` будет по `V0.98` установкой `V0.98` - если вы хотите установить кандидат на выпуск, *вам рекомендуется сначала проверить в виртуальной среде* , вы можете включить это с помощью `--pip install --pre` *package-name* или `--pip install --pre --upgrade` *package-name* . Во многих случаях перед выпуском или выпуском кандидатов могут не быть колес, созданных для всех комбинаций платформы и версии, поэтому вы, скорее всего, столкнетесь с вышеперечисленными проблемами.

Примечание по установке версий разработки

Вы также можете использовать pip для установки версий версий пакетов из github и других локаций, поскольку такой код в потоке очень маловероятен для создания колес, поэтому для любых нечистых пакетов потребуется наличие инструментов сборки, и они могут быть разбитым в любое время, поэтому пользователю **настоятельно** рекомендуется устанавливать такие пакеты только в виртуальной среде.

Для таких установок существуют три варианта:

1. Загрузите сжатый снимок, большинство онлайн-систем контроля версий имеют возможность загрузить сжатый снимок кода. Это можно загрузить вручную, а затем установить с помощью *пути* `pip install` */ to / download / file в* pip, который для большинства форматов сжатия будет обрабатывать распаковку в область кеша и т. Д.
2. Пусть pip обрабатывает загрузку и установку для вас: `pip install` *URL / of / package /*

repository - вам также может понадобиться использовать `--trusted-host`, `--client-cert` и / или `--proxy` чтобы это работало правильно, особенно в корпоративной среде. например:

```
    > 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
```

Обратите внимание на `git+` префикс URL.

3. Клон репозиторий с помощью `git` , `mercurial` или другой приемлемый инструмент, *предпочтительно инструмент DVCS* и использовать `pip install` *путь / к / клонированного / репо* - это будет **как** процесс , **так** любой файл requires.text и выполнять сборки и настройки шагов, *вы можете вручную изменить каталог в ваш клонированный репозиторий и запустите* `pip install -r requires.txt` *а затем* `python setup.py install` *чтобы получить тот же эффект.* Большие преимущества этого подхода заключаются в том, что, хотя начальная операция клонирования может занять больше времени, чем загрузка моментального снимка, которую вы можете обновить до последней версии, в случае git: `git pull origin master` и если текущая версия содержит ошибки, вы можете использовать `pip uninstall` *package-name,* затем используйте команды `git checkout` чтобы вернуться к истории хранилища до более ранних версий и повторить попытку.

Прочитайте pip: Менеджер пакетов PyPI онлайн:
https://riptutorial.com/ru/python/topic/1781/pip--менеджер-пакетов-pypi

глава 20: PostgreSQL

Examples

Начиная

PostgreSQL - это активно развитая и зрелая база данных с открытым исходным кодом. Используя модуль `psycopg2` , мы можем выполнять запросы в базе данных.

Установка с использованием пипса

```
pip install psycopg2
```

Основное использование

`my_table` у нас есть таблица `my_table` в базе данных `my_database` определенная следующим образом.

Я бы	имя	Фамилия
1	Джон	лань

Мы можем использовать модуль `psycopg2` для запуска запросов в базе данных следующим образом.

```
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')]
```

Прочитайте PostgreSQL онлайн: https://riptutorial.com/ru/python/topic/3374/postgresql

глава 21: py.test

Examples

Настройка py.test

`py.test` - одна из нескольких сторонних библиотек тестирования , доступных для Python. Он может быть установлен с помощью `pip` с

```
pip install pytest
```

Код для тестирования

Скажем, мы тестируем функцию добавления в `projectroot/module/code.py` :

```
# projectroot/module/code.py
def add(a, b):
    return a + b
```

Тестирование

Мы создаем тестовый файл в `projectroot/tests/test_code.py` . Файл **должен начинаться с `test_`** для распознавания в качестве файла тестирования.

```
# projectroot/tests/test_code.py
from module import code


def test_add():
    assert code.add(1, 2) == 3
```

Выполнение теста

Из `projectroot` мы просто запускаем `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
================================================
```

Неудачные тесты

Пробный тест даст полезную информацию о том, что пошло не так:

```
# projectroot/tests/test_code.py
from module import code


def test_add__failing():
    assert code.add(10, 11) == 33
```

Результаты:

```
$ 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
================================================
```

Введение в тестовые светильники

Более сложные тесты иногда должны иметь настройки, прежде чем запускать код, который вы хотите проверить. Это можно сделать в самой тестовой функции, но тогда вы оказываете большие функции тестирования, делая так много, что трудно сказать, где остановится установка, и начинается тест. Вы также можете получить много дублирующего кода установки между различными функциями тестирования.

Наш файл кода:

```
# projectroot/module/stuff.py
class Stuff(object):
```

```
    def prep(self):
        self.foo = 1
        self.bar = 2
```

Наш тестовый файл:

```
# 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
```

Это довольно простые примеры, но если нашему объекту `Stuff` понадобилось гораздо больше настроек, оно получило бы громоздкий характер. Мы видим, что между нашими тестовыми примерами существует дублированный код, поэтому давайте сначала переформулируем его в отдельную функцию.

```
# 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
```

Это выглядит лучше, но у нас все еще есть `my_stuff = get_prepped_stuff()` загромождающий

наши тестовые функции.

py.test приспособления на помощь!

Светильники - это гораздо более мощные и гибкие версии функций тестовой настройки. Они могут сделать намного больше, чем мы используем здесь, но мы будем делать это шаг за шагом.

Сначала мы меняем `get_prepped_stuff` на приспособление, называемое `prepped_stuff` . Вы хотите называть свои светильники именами, а не глаголами, из-за того, как приборы впоследствии будут использоваться в самих тестовых функциях. Параметр `@pytest.fixture` указывает, что эту конкретную функцию следует обрабатывать как привязку, а не как обычную функцию.

```
@pytest.fixture
def prepped_stuff():
    my_stuff = stuff.Stuff()
    my_stuff.prep()
    return my_stuff
```

Теперь мы должны обновить тестовые функции, чтобы они использовали прибор. Это делается путем добавления параметра к определению, которое точно соответствует имени прибора. Когда py.test выполняется, он запускает прибор перед запуском теста, а затем передает возвращаемое значение прибора в тестовую функцию через этот параметр. (Обратите внимание, что светильники не **должны** возвращать значение, они могут делать другие вещи настройки, например, вызывать внешний ресурс, упорядочивать вещи в файловой системе, помещать значения в базу данных, независимо от того, какие тесты необходимы для настройки)

```
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
```

Теперь вы можете понять, почему мы назвали его существительным. но `my_stuff = prepped_stuff` практически бесполезна, поэтому давайте просто использовать `prepped_stuff` прямо.

```
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
```

Теперь мы используем светильники! Мы можем пойти дальше, изменив область действия прибора (так что он запускается только один раз на тестовый модуль или сеанс выполнения тестового набора, а не один раз на каждую тестовую функцию), создавая светильники, которые используют другие приборы, параметризуя прибор (чтобы светильник и все тесты, использующие этот прибор, выполняются несколько раз, один раз для каждого параметра, присвоенного прибору), приборы, которые считывают значения из модуля, который их вызывает ..., как упоминалось ранее, светильники обладают гораздо большей мощностью и гибкостью, чем обычная функция настройки.

Очистка после проведения тестов.

Скажем, наш код вырос, и нашему объекту Stuff теперь нужна специальная очистка.

```
# 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
```

Мы могли бы добавить код для вызова очистки в нижней части каждой тестовой функции, но приспособления обеспечивают лучший способ сделать это. Если вы добавите функцию в прибор и зарегистрируете его в качестве **финализатора** , код в функции финализатора будет вызван после теста с использованием приспособления. Если объем устройства больше одной функции (например, модуля или сеанса), финализатор будет выполнен после завершения всех тестов в области видимости, поэтому после завершения работы модуля или в конце всего сеанса тестирования ,

```
@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
```

Использование функции финализатора внутри функции может быть немного сложно понять на первый взгляд, особенно когда у вас есть более сложные приборы. Вместо этого вы можете использовать **приспособление** с **урожаем,** чтобы сделать то же самое с более понятным для пользователя потоком выполнения. Единственное реальное отличие заключается в том, что вместо использования `return` мы используем `yield` в той части прибора, где выполняется настройка, и управление должно перейти к тестовой функции, а затем добавить весь код очистки после `yield` . Мы также украшаем его как `yield_fixture` так что py.test знает, как с ним справиться.

```
@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()
```

И это завершает Intro для тестирования светильников!

Для получения дополнительной информации см. Официальную документацию по документации на py.test и официальную документацию по инвентаризации

Прочитайте py.test онлайн: https://riptutorial.com/ru/python/topic/7054/py-test

глава 22: pyaudio

Вступление

PyAudio обеспечивает привязки Python для PortAudio, кросс-платформенной библиотеки аудио ввода-вывода. С помощью PyAudio вы можете легко использовать Python для воспроизведения и записи звука на различных платформах. PyAudio вдохновлен:

1.pyPortAudio / fastaudio: привязки Python для API PortAudio v18.

2.tkSnack: кросс-платформенный звуковой инструментарий для Tcl / Tk и Python.

замечания

Примечание: stream_callback вызывается в отдельном потоке (из основного потока). Исключения, которые происходят в stream_callback, будут:
1 .print traceback на стандартной ошибке, чтобы помочь отладке,
2. Запишите исключение, которое нужно бросить (в какой-то момент) в основной поток, и
3. Верните paAbort в PortAudio, чтобы остановить поток.
Примечание. Не используйте функцию Stream.read () или Stream.write (), если вы используете неблокирующую операцию.
См. Подпись обратного вызова PortAudio для получения дополнительной информации:
http://portaudio.com/docs/v19-doxydocs/portaudio_8h.html#a8a60fb2a5ec9cbade3f54a9c978e2710

Examples

Режим обратного вызова Аудиовход

```
"""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()
```

В режиме обратного вызова PyAudio вызовет указанную функцию обратного вызова (2), когда ему нужны новые аудиоданные (для воспроизведения) и / или когда имеются новые (записанные) аудиоданные. Обратите внимание, что PyAudio вызывает функцию обратного вызова в отдельном потоке. Функция имеет следующий `callback(<input_data>, <frame_count>, <time_info>, <status_flag>)` подписи `callback(<input_data>, <frame_count>, <time_info>, <status_flag>)` и должна возвращать кортеж, содержащий фреймы `frame_count` аудиоданных, и флаг, указывающий, есть ли больше кадров для воспроизведения / записи.

Начните обработку аудиопотока с помощью **pyaudio.Stream.start_stream ()** (4), который будет вызывать функцию обратного вызова повторно, пока эта функция не вернет **pyaudio.paComplete** .

Чтобы поддерживать поток, основной поток не должен заканчиваться, например, спящим (5).

Режим блокировки Audio I / O

"" "Пример PyAudio: Воспроизведение волнового файла." ""

```
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()
```

Чтобы использовать PyAudio, сначала создайте PyAudio, используя **pyaudio.PyAudio ()** (1), который устанавливает систему portaudio.

Для записи или воспроизведения звука откройте поток на нужном устройстве с требуемыми параметрами звука, используя **pyaudio.PyAudio.open ()** (2). Это создает **pyaudio.Stream** для воспроизведения или записи звука.

Воспроизведите аудио, записав аудиоданные в поток, используя **pyaudio.Stream.write ()** , или прочитайте аудиоданные из потока, используя **pyaudio.Stream.read ()** . (3)

Обратите внимание, что в *режиме блокировки* каждый **pyaudio.Stream.write ()** или **pyaudio.Stream.read ()** блокируется, пока все данные / запрошенные кадры не будут воспроизведены / записаны. В качестве альтернативы для генерации аудиоданных «на лету» или непосредственного обработки записанных аудиоданных используйте «режим обратного вызова» (*см. Пример в режиме обратного вызова*)

Используйте pyaudio.Stream.stop_stream (), чтобы приостановить воспроизведение / запись, и **pyaudio.Stream.close (),** чтобы завершить поток. (4)

Наконец, завершите сеанс **portaudio,** используя **pyaudio.PyAudio.terminate ()** (5)

Прочитайте pyaudio онлайн: https://riptutorial.com/ru/python/topic/10627/pyaudio

глава 23: Pygame

Вступление

Pygame - это библиотека для создания мультимедийных приложений, особенно игр, на Python. Официальный сайт - http://www.pygame.org/ .

Синтаксис

- pygame.mixer.init (частота = 22050, размер = -16, каналы = 2, буфер = 4096)
- pygame.mixer.pre_init (частота, размер, каналы, буфер)
- pygame.mixer.quit ()
- pygame.mixer.get_init ()
- pygame.mixer.stop ()
- pygame.mixer.pause ()
- pygame.mixer.unpause ()
- pygame.mixer.fadeout (время)
- pygame.mixer.set_num_channels (количество)
- pygame.mixer.get_num_channels ()
- pygame.mixer.set_reserved (количество)
- pygame.mixer.find_channel (сила)
- pygame.mixer.get_busy ()

параметры

параметр	подробности
подсчитывать	Положительное целое число, которое представляет собой нечто вроде количества каналов, которые необходимо зарезервировать.
сила	Логическое значение (`False` или `True`), которое определяет, `find_channel()` ли `find_channel()` возвращать канал (неактивный или нет) с помощью `True` или нет (если нет неактивных каналов) с `False`

Examples

Установка pygame

С `pip` :

```
pip install pygame
```

С `conda` :

```
conda install -c tlatorre pygame=1.9.2
```

Прямая загрузка с веб-сайта: http://www.pygame.org/download.shtml

Вы можете найти подходящих инсталляторов для окон и других операционных систем.

Проекты можно также найти по адресу http://www.pygame.org/

Модуль микшера Pygame

Модуль `pygame.mixer` помогает управлять музыкой, используемой в программах `pygame` . На данный момент для `mixer` модуля имеется 15 различных функций.

Инициализация

Подобно тому, как вы должны инициализировать `pygame` с помощью `pygame.init()` , вы также должны инициализировать `pygame.mixer` .

Используя первый вариант, мы инициализируем модуль, используя значения по умолчанию. Однако вы можете переопределить эти параметры по умолчанию. Используя второй вариант, мы можем инициализировать модуль, используя значения, которые мы вручную вносим сами. Стандартные значения:

```
pygame.mixer.init(frequency=22050, size=-16, channels=2, buffer=4096)
```

Чтобы проверить, были ли мы инициализированы это или нет, мы можем использовать `pygame.mixer.get_init()` , который возвращает `True` если он есть, и `False` если это не так. Чтобы завершить / отменить инициализацию, просто используйте `pygame.mixer.quit()` . Если вы хотите продолжить воспроизведение звуков с помощью модуля, вам может потребоваться повторная инициализация модуля.

Возможные действия

Когда ваш звук воспроизводится, вы можете временно приостановить его с помощью `pygame.mixer.pause()` . Чтобы возобновить воспроизведение ваших звуков, просто используйте `pygame.mixer.unpause()` . Вы также можете затухать в конце звука, используя `pygame.mixer.fadeout()` . Он принимает аргумент, который представляет собой количество миллисекунд, которое требуется, чтобы завершить затухание музыки.

каналы

Вы можете воспроизводить столько песен, сколько необходимо, пока есть достаточно открытых каналов для их поддержки. По умолчанию имеется 8 каналов. Чтобы изменить количество каналов, используйте `pygame.mixer.set_num_channels()` . Аргумент - неотрицательное целое число. Если количество каналов уменьшается, любые звуки, воспроизводимые на удаленных каналах, будут немедленно остановлены.

Чтобы узнать, сколько каналов используется в данный момент, вызовите `pygame.mixer.get_channels(count)` . Результатом является количество каналов, которые в настоящее время не открыты. Вы также можете резервировать каналы для звуков, которые должны воспроизводиться с помощью `pygame.mixer.set_reserved(count)` . Аргумент также является неотрицательным целым числом. Любые звуки, воспроизводимые на недавно зарезервированных каналах, не будут остановлены.

Вы также можете узнать, какой канал не используется, используя `pygame.mixer.find_channel(force)` . Его аргумент - это bool: True или False. Если нет каналов , которые находятся в режиме ожидания и `force` является False, то он не будет возвращать `None` . Если `force` верна, она вернет канал, который играл в течение самого долгого времени.

Прочитайте Pygame онлайн: https://riptutorial.com/ru/python/topic/8761/pygame

глава 24: Pyglet

Вступление

Pyglet - это модуль Python, используемый для визуализации и звука. Он не имеет зависимости от других модулей. См. [Pyglet.org] [1] для официальной информации. [1]: http://pyglet.org

Examples

Привет, мир в 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()
```

Установка Pyglet

Установите Python, перейдите в командную строку и введите:

Python 2:

```
pip install pyglet
```

Python 3:

```
pip3 install pyglet
```

Воспроизведение звука в Pyglet

```
sound = pyglet.media.load(sound.wav)
sound.play()
```

Использование Pyglet для 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()
```

Точки рисования с использованием Pyglet и 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
```

Чтобы подключить точки, замените `GL_POINTS` на `GL_LINE_LOOP` .

Прочитайте Pyglet онлайн: https://riptutorial.com/ru/python/topic/8208/pyglet

глава 25: PyInstaller - Распространение кода Python

Синтаксис

- pyinstaller [options] script [script ...] | файл спецификация

замечания

PyInstaller - это модуль, используемый для объединения приложений python в один пакет вместе со всеми зависимостями. Затем пользователь может запустить приложение пакета без интерпретатора python или любых модулей. Он правильно связывает многие крупные пакеты, такие как numpy, Django, OpenCv и другие.

Некоторые важные моменты, которые следует помнить:

- Pyinstaller поддерживает Python 2.7 и Python 3.3+
- Pyinstaller был протестирован против Windows, Linux и Mac OS X.
- Это **НЕ** перекрестный компилятор. (Приложение Windows не может быть упаковано в Linux. Вы должны запустить PyInstaller в Windows, чтобы связать приложение для Windows)

Главная Официальные документы

Examples

Установка и настройка

Pyinstaller - обычный пакет python. Он может быть установлен с использованием pip:

```
pip install pyinstaller
```

Установка в Windows
Для Windows pywin32 или pypiwin32 является обязательным условием. Последний устанавливается автоматически, когда pyinstaller устанавливается с помощью pip.

Установка в Mac OS X
PyInstaller работает с Python 2.7 по умолчанию с текущей Mac OS X. Если будут использоваться более поздние версии Python или если будут использоваться какие-либо основные пакеты, такие как PyQT, Numpy, Matplotlib и т. Д., Рекомендуется установить их, используя либо MacPorts, либо Homebrew .

Установка из архива

Если пип недоступен, загрузите сжатый архив из PyPI .
Чтобы протестировать версию разработки, загрузите сжатый архив из раздела *разработки* страницы загрузки PyInstaller .

Разверните архив и найдите скрипт `setup.py` . Выполните `python setup.py install` с правами администратора для установки или обновления PyInstaller.

Проверка установки

Командная команда `pyinstaller` должна существовать на системном пути для всех платформ после успешной установки.
Проверьте его, набрав `pyinstaller --version` в командной строке. Это будет печатать текущую версию pyinstaller.

Использование Pyinstaller

В простейшем случае использования просто перейдите к каталогу, в котором находится ваш файл, и введите:

```
pyinstaller myfile.py
```

Pyinstaller анализирует файл и создает:

- Файл **myfile.spec** в том же каталоге, что и `myfile.py`
- Папка **сборки** в том же каталоге, что и `myfile.py`
- Папка **dist** в том же каталоге, что и `myfile.py`
- Файлы журналов в папке **сборки**

Вложенное приложение можно найти в папке **dist**

Опции

Существует несколько вариантов, которые можно использовать с pyinstaller. Полный список опций можно найти здесь .

После установки вашего приложения можно запустить, открыв 'dist \ myfile \ myfile.exe'.

Объединение в одну папку

Когда PyInstaller используется без каких-либо параметров для `myscript.py` , выход по умолчанию представляет собой одну папку (называемую `myscript`), содержащую исполняемый файл с именем `myscript` (`myscript.exe` в окнах) вместе со всеми необходимыми зависимостями.
Приложение может быть распространено путем сжатия папки в zip-файл.

Режим One Folder может быть определенно определен с помощью опции `-D` или `--onedir`

```
pyinstaller myscript.py -D
```

Преимущества:

Одним из основных преимуществ объединения одной папки является то, что легче отлаживать проблемы. Если какие-либо модули не могут импортироваться, их можно проверить, проверив папку.
Еще одно преимущество ощущается во время обновлений. Если в коде несколько изменений, но используемые зависимости *точно* совпадают, дистрибуторы могут просто отправить исполняемый файл (который обычно меньше всей папки).

Недостатки

Единственным недостатком этого метода является то, что пользователи должны искать исполняемый файл из большого количества файлов.
Также пользователи могут удалять / изменять другие файлы, которые могут привести к тому, что приложение не сможет работать правильно.

Объединение в один файл

```
pyinstaller myscript.py -F
```

Параметры для создания одного файла: `-F` или `--onefile` . Это `myscript.exe` программу в один файл `myscript.exe` .

Исполняемый файл одного файла медленнее, чем пакет с одной папкой. Их также сложнее отлаживать.

Прочитайте PyInstaller - Распространение кода Python онлайн:
https://riptutorial.com/ru/python/topic/2289/pyinstaller---распространение-кода-python

глава 26: Python Anti-Patterns

Examples

Переусердствовать, кроме оговорки

Исключения являются мощными, но одна чрезмерно усредненная фраза может занять все это в одной строке.

```
try:
    res = get_result()
    res = res[0]
    log('got result: %r' % res)
except:
    if not res:
        res = ''
    print('got exception')
```

Этот пример демонстрирует 3 симптома антипаттера:

1. За `except` без указания типа исключения (строка 5) будет ловить даже здоровые исключения, в том числе `KeyboardInterrupt` . Это предотвратит выход из программы в некоторых случаях.
2. Блок except не регрессирует ошибку, что означает, что мы не сможем определить, произошло ли исключение из `get_result` или потому, что `res` был пустым.
3. Хуже всего, если бы мы были обеспокоены тем, что результат был пуст, мы привели к чему-то значительно худшему. Если `get_result` не удался, `res` останется полностью отключенным, а ссылка на `res` в `NameError` блоке повысит значение `NameError` , полностью маскируя исходную ошибку.

Всегда думайте о типе исключения, которое вы пытаетесь обработать. Дайте страницу исключений прочитанной и почувствуйте, какие основные исключения существуют.

Вот фиксированная версия приведенного выше примера:

```
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)
```

Мы улавливаем более конкретные исключения, перейзируя там, где это необходимо. Еще несколько строк, но бесконечно правильнее.

Глядя перед вами, прыгайте с интенсивной работой процессора

Программа может легко тратить время, вызывая многократно процессорную функцию.

Например, возьмите функцию, которая выглядит так: она возвращает целое число, если входное `value` может создать одно, иначе `None` :

```
def intensive_f(value): # int -> Optional[int]
   # complex, and time-consuming code
   if process_has_failed:
       return None
   return integer_output
```

И его можно использовать следующим образом:

```
x = 5
if intensive_f(x) is not None:
    print(intensive_f(x) / 2)
else:
    print(x, "could not be processed")

print(x)
```

Пока это будет работать, возникает проблема вызова `intensive_f` , что удваивает время выполнения кода. Лучшим решением было бы получить возвращаемое значение функции заранее.

```
x = 5
result = intensive_f(x)
if result is not None:
    print(result / 2)
else:
    print(x, "could not be processed")
```

Однако более ясный и, возможно, более питонический способ заключается в использовании исключений, например:

```
x = 5
try:
    print(intensive_f(x) / 2)
except TypeError: # The exception raised if None + 1 is attempted
    print(x, "could not be processed")
```

Здесь не требуется временная переменная. Часто может быть предпочтительнее использовать оператор `assert` и вместо этого ухватить `AssertionError` .

Словарь ключей

Общим примером того, где это можно найти, является доступ к клавишам словаря. Например, сравните:

```
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)
```

с:

```
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)
```

Первый пример должен дважды изучить словарь, и поскольку это длинный словарь, это может занять много времени, чтобы сделать это каждый раз. Второй требует только одного поиска через словарь и, таким образом, экономит много процессорного времени.

Альтернативой этому является использование `dict.get(key, default)` , однако многим обстоятельствам может потребоваться выполнение более сложных операций в случае, если ключ отсутствует

Прочитайте Python Anti-Patterns онлайн: https://riptutorial.com/ru/python/topic/4700/python-anti-patterns

глава 27: Python HTTP Server

Examples

Запуск простого HTTP-сервера

Python 2.x 2.3

```
python -m SimpleHTTPServer 9000
```

Python 3.x 3.0

```
python -m http.server 9000
```

Выполнение этой команды служит для файлов текущего каталога на порту `9000` .

Если аргумент не указан как номер порта, сервер будет работать по умолчанию по умолчанию `8000` .

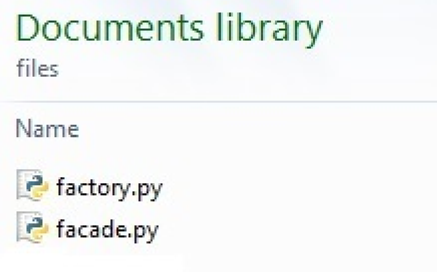
Флаг `-m` будет искать `sys.path` для соответствующего `.py` файла для запуска в качестве модуля.

Если вы хотите показывать только на локальном хосте, вам нужно написать пользовательскую программу Python, такую как:

```
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()
```

Служебные файлы

Предполагая, что у вас есть следующий каталог файлов:



Вы можете настроить веб-сервер для обслуживания этих файлов следующим образом:

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()
```

Модуль `SocketServer` предоставляет классы и функции для настройки сетевого сервера.

`SocketServer` `TCPServer` устанавливает сервер с использованием протокола TCP. Конструктор принимает кортеж, представляющий адрес сервера (т. Е. IP-адрес и порт), и класс, который обрабатывает запросы сервера.

`SimpleHTTPRequestHandler` класс `SimpleHTTPServer` модуля позволяет файлы в текущем каталоге будут обслужены.

Сохраните сценарий в том же каталоге и запустите его.

Запустите HTTP-сервер:

Python 2.x 2.3

python -m SimpleHTTPServer 8000

Python 3.x 3.0

python -m http.server 8000

Флаг '-m' будет искать 'sys.path' для соответствующего файла .py для запуска в качестве модуля.

Откройте localhost: 8000 в браузере, он даст вам следующее:

Directory listing for /

- facade.py
- factory.py
- server.py

Программный API SimpleHTTPServer

Что происходит, когда мы выполняем `python -m SimpleHTTPServer 9000` ?

Чтобы ответить на этот вопрос, мы должны понять конструкцию SimpleHTTPServer (https://hg.python.org/cpython/file/2.7/Lib/SimpleHTTPServer.py) и BaseHTTPServer (https://hg.python.org/cpython/file /2.7/Lib/BaseHTTPServer.py) .

Во-первых, Python вызывает модуль `SimpleHTTPServer` с `9000` в качестве аргумента. Теперь, наблюдая за кодом SimpleHTTPServer,

```
def test(HandlerClass = SimpleHTTPRequestHandler,
         ServerClass = BaseHTTPServer.HTTPServer):
    BaseHTTPServer.test(HandlerClass, ServerClass)


if __name__ == '__main__':
    test()
```

Функция тестирования вызывается после обработчиков запросов и ServerClass. Теперь вызывается BaseHTTPServer.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()
```

Следовательно, здесь номер порта, который пользователь передал как аргумент, анализируется и привязан к адресу хоста. Проводятся дополнительные базовые шаги программирования сокетов с заданным портом и протоколом. Наконец, запущен сервер сокетов.

Это базовый обзор наследования класса SocketServer для других классов:

```
    +------------+
    | BaseServer |
    +------------+
          |
          v
    +-----------+        +------------------+
    | TCPServer |------->| UnixStreamServer |
    +-----------+        +------------------+
          |
          v
    +-----------+        +--------------------+
    | UDPServer |------->| UnixDatagramServer |
    +-----------+        +--------------------+
```

Ссылки https://hg.python.org/cpython/file/2.7/Lib/BaseHTTPServer.py и https://hg.python.org/cpython/file/2.7/Lib/SocketServer.py полезны для поиска дальнейших Информация.

Основная обработка GET, POST, PUT с использованием 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()
```

Пример вывода с использованием `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
```

Прочитайте Python HTTP Server онлайн: https://riptutorial.com/ru/python/topic/4247/python-http-server

глава 28: Python Lex-Yacc

Вступление

PLY - это реализация на чистом Python популярных инструментов построения компилятора lex и yacc.

замечания

Дополнительные ссылки:

1. Официальные документы
2. Github

Examples

Начало работы с PLY

Чтобы установить PLY на свой компьютер для python2 / 3, выполните следующие действия:

1. Загрузите исходный код отсюда .
2. Разархивируйте загруженный zip-файл
3. Перейдите в папку с распакованным `ply-3.10`
4. Выполните следующую команду в своем терминале: `python setup.py install`

Если вы закончите все вышеперечисленное, вы должны теперь использовать модуль PLY. Вы можете проверить это, открыв интерпретатор python и набрав `import ply.lex` .

Примечание: *Не* используйте `pip` установить PLY, он установит битый дистрибутив на вашей машине.

«Привет, мир!» PLY - простой калькулятор

Давайте продемонстрируем мощь PLY с помощью простого примера: эта программа примет арифметическое выражение как строковый ввод и попытается его решить.

Откройте свой любимый редактор и скопируйте следующий код:

```
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)
```

Сохраните этот файл как `calc.py` и запустите его.

Выход:

```
-8
```

Какой правильный ответ для `-4 * - (3 - 5)` .

Часть 1: токенизация ввода с помощью Lex

Существует два этапа выполнения кода из примера 1: один - *токенирование* ввода, что означает, что он ищет символы, которые составляют арифметическое выражение, а второй - *синтаксический анализ* , который включает анализ извлеченных токенов и оценку результата.

В этом разделе приведен простой пример того, как *вводить токены* пользователя, а затем *разбивать его* по *строкам* .

```
    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)
```

Сохраните этот файл как `calclex.py` . Мы будем использовать это при создании нашего анализатора Yacc.

Сломать

1. Импортируйте модуль с помощью `import ply.lex`

2. Все лексеры должны предоставить список, называемый `tokens` который определяет все возможные имена токенов, которые могут быть созданы лексером. Этот список всегда необходим.

```
tokens = [
   'NUMBER',
   'PLUS',
   'MINUS',
   'TIMES',
   'DIVIDE',
   'LPAREN',
   'RPAREN',
]
```

`tokens` также могут быть кортежем строк (а не строки), где каждая строка обозначает токен, как и раньше.

3. Правило регулярных выражений для каждой строки может быть определено как строка или как функция. В любом случае имя переменной должно иметь префикс t_, чтобы обозначить это правило для сопоставления токенов.

   - Для простых токенов регулярное выражение может быть указано как строки: `t_PLUS = r'\+'`
   - Если необходимо выполнить какое-либо действие, в качестве функции можно указать правило токена.

     ```
     def t_NUMBER(t):
         r'\d+'
         t.value = int(t.value)
         return t
     ```

     Обратите внимание, что правило задается как строка документа внутри функции. Функция принимает один аргумент, который является экземпляром `LexToken` , выполняет какое-то действие и затем возвращает аргумент.

     Если вы хотите использовать внешнюю строку в качестве правила регулярного выражения для функции вместо указания строки doc, рассмотрите следующий пример:

     ```
     @TOKEN(identifier)         # identifier is a string holding the regex
     def t_ID(t):
         ...      # actions
     ```

   - Экземпляр объекта `LexToken` (назовем этот объект `t`) имеет следующие атрибуты:

     1. `t.type` который является типом маркера (как строка) (например: `'NUMBER'` , `'PLUS'` и т. д.). По умолчанию `t.type` устанавливается на имя, следующее за префиксом `t_` .
     2. `t.value` который является лексемой (фактический текст соответствует)
     3. `t.lineno` который является текущим номером строки (это не обновляется автоматически, так как лексер ничего не знает о номерах строк). Обновите lineno, используя функцию `t_newline` .

     ```
     def t_newline(t):
         r'\n+'
         t.lexer.lineno += len(t.value)
     ```

     4. `t.lexpos` который является позицией маркера относительно начала входного текста.

- Если ничего не возвращается из функции правила регулярного выражения, токен отбрасывается. Если вы хотите отменить токен, вы можете вместо этого добавить префикс t_ignore_ к переменной правила regex вместо определения функции для того же правила.

  ```
  def t_COMMENT(t):
      r'\#.*'
      pass
      # No return value. Token discarded
  ```

  ...Такой же как:

  ```
  t_ignore_COMMENT = r'\#.*'
  ```

  Это, конечно, недействительно, если вы выполняете некоторые действия, когда видите комментарий. В этом случае используйте функцию для определения правила регулярного выражения.

  Если вы не определили токен для некоторых символов, но все же хотите его игнорировать, используйте `t_ignore = "<characters to ignore>"` (эти префиксы необходимы):

  ```
  t_ignore_COMMENT = r'\#.*'
  t_ignore  = ' \t'    # ignores spaces and tabs
  ```

- При создании главного регулярного выражения lex добавляет регулярные выражения, указанные в файле, следующим образом:

  1. Токены, определенные функциями, добавляются в том же порядке, что и в файле.
  2. Токены, определенные строками, добавляются в порядке убывания длины строки строки, определяющей регулярное выражение для этого токена.

  Если вы соответствуете `==` и `=` в том же файле, воспользуйтесь этими правилами.

- Литералы - это жетоны, которые возвращаются так, как они есть. И `t.type` и `t.value` будут установлены самим символом. Определите список литералов как таковых:

  ```
  literals = [ '+', '-', '*', '/' ]
  ```

  или же,

  ```
  literals = "+-*/"
  ```

Можно записывать функции токена, которые выполняют дополнительные действия при сопоставлении литералов. Однако вам необходимо установить тип токена соответствующим образом. Например:

```
literals = [ '{', '}' ]

def t_lbrace(t):
    r'\{'
    t.type = '{'  # Set token type to the expected literal (ABSOLUTE MUST if this
is a literal)
    return t
```

- Обработать ошибки с помощью функции 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)
```

В общем случае `t.lexer.skip(n)` пропускает n символов во входной строке.

4. Заключительные приготовления:

Создайте лексер, используя `lexer = lex.lex()` .

Вы также можете поместить все внутри класса и вызвать экземпляр класса для определения lexer. Например:

```
 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")     #
```

Предоставление ввода с использованием `lexer.input(data)` где данные являются строкой

Чтобы получить маркеры, используйте `lexer.token()` который возвращает маркеры. Вы

можете перебирать лексер в цикле, как в:

```
for i in lexer:
    print(i)
```

Часть 2: Анализ токенизированных входных данных с помощью Yacc

В этом разделе объясняется, как обработанный токен из части 1 обрабатывается - это делается с использованием Context Free Grammars (CFG). Грамматика должна быть указана, а токены обрабатываются в соответствии с грамматикой. Под капотом анализатор использует парсер 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)
```

Сломать

- Каждое правило грамматики определяется функцией, где docstring к этой функции содержит соответствующую контекстуальную грамматическую спецификацию. Операторы, составляющие тело функции, реализуют семантические действия правила. Каждая функция принимает один аргумент p, который представляет собой последовательность, содержащую значения каждого символа грамматики в соответствующем правиле. Значения `p[i]` отображаются на символы грамматики, как показано здесь:

```
def p_expression_plus(p):
    'expression : expression PLUS term'
    #   ^            ^        ^    ^
    #  p[0]         p[1]     p[2] p[3]

    p[0] = p[1] + p[3]
```

- Для токенов «значение» соответствующего `p[i]` такое же, как атрибут `p.value` назначенный в модуле lexer. Таким образом, `PLUS` будет иметь значение `+` .

- Для нетерминалов значение определяется тем, что помещено в `p[0]` . Если ничего не установлено, значение равно None. Кроме того, `p[-1]` не совпадает `p[3]` , так как `p` не является простым списком (`p[-1]` может указывать встроенные действия (здесь не обсуждается)).

Обратите внимание, что функция может иметь любое имя, если она предшествует `p_` .

- `p_error(p)` определено для улавливания синтаксических ошибок (таких же, как `yyerror` в yacc / bison).

- Несколько правил грамматики могут быть объединены в одну функцию, что является хорошей идеей, если постановки имеют сходную структуру.

```
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]
```

- Символьные литералы могут использоваться вместо токенов.

```
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]
```

  Конечно, литералы должны быть указаны в лексерском модуле.

- Пустые постановки имеют форму `'''symbol : '''`

- Чтобы явно установить символ начала, используйте `start = 'foo'` , где `foo` - некоторый нетерминальный.

- Установка приоритета и ассоциативности может быть выполнена с использованием переменной приоритета.

```
precedence = (
    ('nonassoc', 'LESSTHAN', 'GREATERTHAN'),  # Nonassociative operators
    ('left', 'PLUS', 'MINUS'),
    ('left', 'TIMES', 'DIVIDE'),
    ('right', 'UMINUS'),            # Unary minus operator
)
```

  Токены упорядочены с самого низкого до старшего приоритета. `nonassoc` означает, что эти токены не ассоциируются. Это означает, что что-то вроде `a < b < c` является незаконным, тогда `a < b` все еще является законным.

- `parser.out` - файл отладки, который создается, когда программа yacc выполняется в первый раз. Всякий раз, когда возникает конфликт сдвига / уменьшения, парсер всегда сдвигается.

Прочитайте Python Lex-Yacc онлайн: https://riptutorial.com/ru/python/topic/10510/python-lex-yacc

глава 29: Python и Excel

Examples

Поместите данные списка в файл 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 - это модуль для управления и создания `xlsx/xlsm/xltx/xltm` в памяти.

Манипуляция и чтение существующей книги:

```
import openpyxl as opx
#To change an existing wookbook we located it by referencing its path
workbook = opx.load_workbook(workbook_path)
```

`load_workbook()` содержит параметр `read_only` , установка этого параметра в `True` будет загружать книгу как read_only, это полезно при чтении больших файлов `xlsx` :

```
workbook = opx.load_workbook(workbook_path, read_only=True)
```

После того, как вы загрузили книгу в память, вы можете получить доступ к отдельным листам, используя `workbook.sheets`

```
first_sheet = workbook.worksheets[0]
```

Если вы хотите указать имя доступных листов, вы можете использовать `workbook.get_sheet_names()` .

```
sheet = workbook.get_sheet_by_name('Sheet Name')
```

Наконец, строки листа могут быть доступны с помощью `sheet.rows` . Чтобы перебрать строки в листе, используйте:

```
for row in sheet.rows:
    print row[0].value
```

Поскольку каждая `row` в `rows` является списком `Cell` s, используйте `Cell.value` для получения содержимого Cell.

Создание новой книги в памяти:

```
#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
```

Несколько свойств вкладки могут быть изменены через openpyxl, например `tabColor` :

```
ws.sheet_properties.tabColor = 'FFC0CB'
```

Чтобы сохранить нашу созданную книгу, мы закончим:

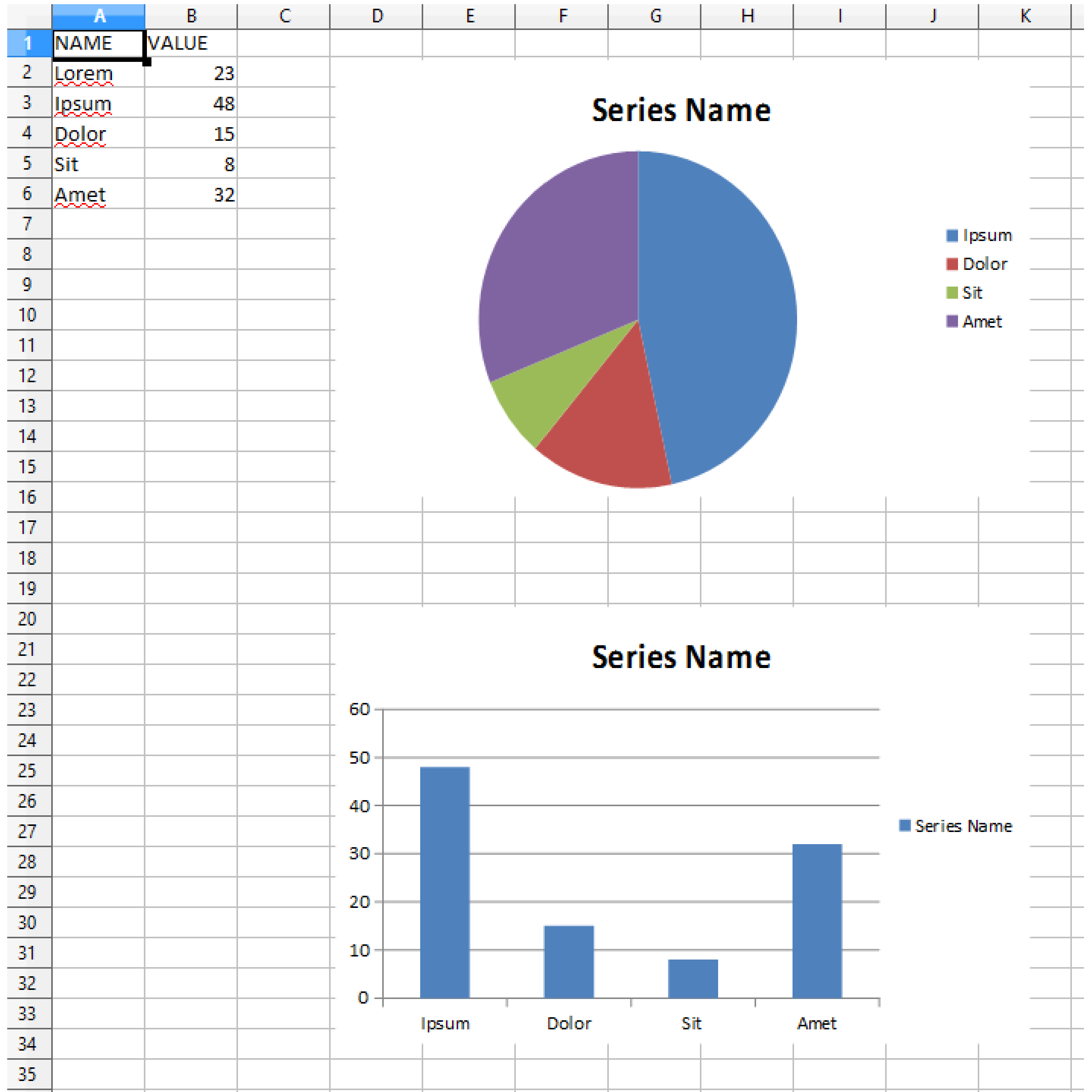
```
wb.save('filename.xlsx')
```

Создание графиков excel с помощью 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()
```

Результат:



Чтение данных excel с использованием модуля xlrd

Библиотека Python xlrd предназначена для извлечения данных из файлов электронной таблицы Microsoft Excel (tm).

Монтаж:-

```
pip install xlrd
```

Или вы можете использовать файл setup.py из pypi

https://pypi.python.org/pypi/xlrd

Чтение листа excel: - Импортируйте модуль xlrd и откройте файл excel, используя метод open_workbook ().

```
import xlrd
book=xlrd.open_workbook('sample.xlsx')
```

Проверьте количество листов в Excel

```
print book.nsheets
```

Печать имен листов

```
print book.sheet_names()
```

Получить лист по индексу

```
sheet=book.sheet_by_index(1)
```

Прочитайте содержимое ячейки

```
cell = sheet.cell(row,col) #where row=row number and col=column number
print cell.value #to print the cell contents
```

Получить количество строк и количество столбцов в листе excel

```
num_rows=sheet.nrows
num_col=sheet.ncols
```

Получить лист Excel по имени

```
sheets = book.sheet_names()
cur_sheet = book.sheet_by_name(sheets[0])
```

Форматировать файлы Excel с помощью 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()
```

Прочитайте Python и Excel онлайн: https://riptutorial.com/ru/python/topic/2986/python-и-excel

глава 30: setup.py

параметры

параметр	использование
`name`	Название вашего дистрибутива.
`version`	Строка версии вашего дистрибутива.
`packages`	Список пакетов Python (то есть каталогов, содержащих модули) для включения. Это можно указать вручную, но вместо этого обычно используется вызов `setuptools.find_packages()` .
`py_modules`	Список модулей Python верхнего уровня (то есть отдельных файлов `.py`) для включения.

замечания

Для получения дополнительной информации о упаковке python см.

Вступление

Для написания официальных пакетов есть руководство пользователя для упаковки .

Examples

Назначение setup.py

Сценарий установки является центром всей деятельности по созданию, распределению и установке модулей с использованием Distutils. Целью является правильная установка программного обеспечения.

Если все, что вы хотите сделать, это распространять модуль под названием foo, содержащийся в файле foo.py, тогда ваш сценарий установки может быть таким же простым, как это:

```
from distutils.core import setup

setup(name='foo',
      version='1.0',
      py_modules=['foo'],
      )
```

Чтобы создать исходный дистрибутив для этого модуля, вы должны создать сценарий установки, setup.py, содержащий вышеуказанный код, и запустить эту команду с терминала:

```
python setup.py sdist
```

sdist создаст файл архива (например, tarball в Unix, ZIP-файл в Windows), содержащий ваш скрипт setup.py и ваш модуль foo.py. Файл архива будет называться foo-1.0.tar.gz (или .zip) и распакуется в каталог foo-1.0.

Если конечный пользователь хочет установить ваш модуль foo, все, что ей нужно сделать, это загрузить foo-1.0.tar.gz (или .zip), распаковать его и-из запуска каталога foo-1.0

```
python setup.py install
```

Добавление сценариев командной строки в пакет python

Скрипты командной строки внутри пакетов python являются общими. Вы можете организовать свой пакет таким образом, чтобы, когда пользователь устанавливает пакет, скрипт будет доступен по их пути.

Если у вас есть пакет `greetings` , в котором был скрипт командной строки `hello_world.py` .

```
greetings/
   greetings/
      __init__.py
      hello_world.py
```

Вы можете запустить этот скрипт, выполнив:

```
python greetings/greetings/hello_world.py
```

Однако, если вы хотите запустить его так:

```
hello_world.py
```

Вы можете добиться этого, добавив `scripts` в свою `setup()` в `setup.py` следующим образом:

```
from setuptools import setup
setup(
  name='greetings',
  scripts=['hello_world.py']
)
```

Когда вы установите пакет приветствия, `hello_world.py` будет добавлен в ваш путь.

Другая возможность - добавить точку входа:

```
entry_points={'console_scripts': ['greetings=greetings.hello_world:main']}
```

Таким образом, вам просто нужно запустить его, как:

```
greetings
```

Использование метаданных управления версиями в файле setup.py

setuptools_scm - официально благословленный пакет, который может использовать метаданные Git или Mercurial для определения номера версии вашего пакета и поиска пакетов Python и данных пакета для включения в него.

```
from setuptools import setup, find_packages

setup(
    setup_requires=['setuptools_scm'],
    use_scm_version=True,
    packages=find_packages(),
    include_package_data=True,
)
```

В этом примере используются обе функции; для использования только метаданных SCM для версии, замените вызов `find_packages()` на ваш список пакетов вручную или используйте только `use_scm_version=True` пакетов, удалите `use_scm_version=True` .

Добавление параметров установки

Как видно из предыдущих примеров, базовое использование этого скрипта:

```
python setup.py install
```

Но есть еще больше возможностей, таких как установка пакета и возможность изменить код и протестировать его, не переустанавливая его. Это делается с использованием:

```
python setup.py develop
```

Если вы хотите выполнить определенные действия, такие как компиляция документации *Sphinx* или создание кода *fortran* , вы можете создать свой собственный вариант следующим образом:

```
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` и `finalize_options` будут выполняться до и после функции `run` поскольку их имена предполагают это.

После этого вы сможете позвонить по своему выбору:

```
python setup.py build_sphinx
```

Прочитайте setup.py онлайн: https://riptutorial.com/ru/python/topic/1444/setup-py

глава 31: tempfile NamedTemporaryFile

параметры

пары	описание
Режим	режим открытия файла, по умолчанию = w + b
удалять	Чтобы удалить файл при закрытии, значение по умолчанию = True
суффикс	filename suffix, default = ''
префикс	префикс имени файла, default = 'tmp'
реж	dirname для размещения tempfile, default = None
buffsize	default = -1 (используется по умолчанию для операционной системы)

Examples

Создать (и написать в) известный, постоянный временный файл

Вы можете создавать временные файлы с видимым именем в файловой системе, к которым можно получить доступ через свойство `name` . Файл может на unix-системах настраиваться на удаление при закрытии (устанавливается параметром `delete` , по умолчанию - True) или может быть снова открыт позже.

Следующее создаст и откроет именованный временный файл и напишет «Hello World!». к этому файлу. Доступ к файловому пути временного файла можно получить через `name` , в этом примере он сохраняется в `path` переменной и печатается для пользователя. Затем файл открывается после закрытия файла, и содержимое файла temp считывается и распечатывается для пользователя.

```
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()
```

Выход:

```
/tmp/tmp6pireJ
Hello World!
```

Прочитайте tempfile NamedTemporaryFile онлайн: https://riptutorial.com/ru/python/topic/3666/tempfile-namedtemporaryfile

глава 32: Tkinter

Вступление

Выпущен в Tkinter - это самая популярная библиотека графического интерфейса пользователя (Python). В этом разделе объясняется правильное использование этой библиотеки и ее функций.

замечания

Капитализация модуля tkinter отличается между Python 2 и 3. Для Python 2 используйте следующее:

```
from Tkinter import *  # Capitalized
```

Для Python 3 используйте следующее:

```
from tkinter import *  # Lowercase
```

Для кода, который работает с Python 2 и 3, вы можете либо сделать

```
try:
    from Tkinter import *
except ImportError:
    from tkinter import *
```

или же

```
from sys import version_info
if version_info.major == 2:
    from Tkinter import *
elif version_info.major == 3:
    from tkinter import *
```

> См. Документацию tkinter для более подробной информации.

Examples

Минимальное приложение tkinter

`tkinter` - это инструментарий GUI, который предоставляет оболочку вокруг библиотеки GUI Tk / Tcl и входит в состав Python. Следующий код создает новое окно с использованием `tkinter` и помещает некоторый текст в тело окна.

Примечание. В Python 2 заглавная буква может немного отличаться, см. Раздел «Примечания» ниже.

```
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()
```

Руководители геометрии

Tkinter имеет три механизма управления геометрией: `place` , `pack` и `grid` .

Менеджер `place` использует абсолютные пиксельные координаты.

Менеджер `pack` помещает виджеты в одну из четырех сторон. Новые виджеты размещаются рядом с существующими виджетами.

Менеджер `grid` помещает виджеты в сетку, аналогичную динамически изменяющейся электронной таблице.

Место

Наиболее распространенными аргументами ключевого слова для `widget.place` являются:

- `x` , абсолютная x-координата виджета
- `y` , абсолютная y-координата виджета
- `height` , абсолютная высота виджета
- `width` , абсолютная ширина виджета

Пример кода с использованием `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()
```

пак

`widget.pack` может принимать следующие аргументы ключевых слов:

- `expand` , заполнить или не заполнить пространство, оставшееся от родителя
- `fill` , расширять ли, чтобы заполнить все пространство (NONE (по умолчанию), X, Y или ОБОИХ)
- `side` , сторона для упаковки против (TOP (по умолчанию), BOTTOM, LEFT или RIGHT)

сетка

Наиболее часто используемые аргументы ключевого слова `widget.grid` следующие:

- `row` , строка виджета (по умолчанию самая маленькая незанятая)
- `rowspan` , количество колонок в виджетах (по умолчанию 1)
- `column` , столбец виджета (по умолчанию 0)
- `columnspan` , количество столбцов, в которых виджет охватывает (по умолчанию 1)
- `sticky` , где размещать виджет, если ячейка сетки больше (комбинация N, NE, E, SE, S, SW, W, NW)

Строки и столбцы нулевые индексируются. Строки увеличиваются, а столбцы увеличиваются.

Пример кода с использованием `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()
```

Никогда не смешивайте `pack` и `grid` в одном раме! Это приведет к запуску приложения!

Прочитайте Tkinter онлайн: https://riptutorial.com/ru/python/topic/7574/tkinter

глава 33: Unicode

Examples

Кодирование и декодирование

Всегда *кодировать* из юникода в байты. В этом направлении **вы можете выбрать кодировку** .

```
>>> u'🐍'.encode('utf-8')
'\xf0\x9f\x90\x8d'
```

Другой способ - *декодировать* от байтов до unicode. В этом направлении **вы должны знать, что такое кодировка** .

```
>>> b'\xf0\x9f\x90\x8d'.decode('utf-8')
u'\U0001f40d'
```

Прочитайте Unicode онлайн: https://riptutorial.com/ru/python/topic/5618/unicode

глава 34: URLLIB

Examples

HTTP GET

Python 2.x 2.7

Python 2

```
import urllib
response = urllib.urlopen('http://stackoverflow.com/documentation/')
```

Использование `urllib.urlopen()` вернет объект ответа, который можно обрабатывать аналогично файлу.

```
print response.code
# Prints: 200
```

Код `response.code` представляет собой возвращаемое значение http. 200 в порядке, 404 - NotFound и т. Д.

```
print response.read()
'<!DOCTYPE html>\r\n<html>\r\n<head>\r\n\r\n<title>Documentation - Stack. etc'
```

`response.read()` и `response.readlines()` могут использоваться для чтения фактического файла html, возвращенного из запроса. Эти методы работают аналогично `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>
```

Модуль обновлен для Python 3.x, но варианты использования остаются в основном

одинаковыми. `urllib.request.urlopen` вернет аналогичный файл-подобный объект.

HTTP POST

В данные POST передаются аргументы закодированного запроса как данные в 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'
```

Декодировать принятые байты согласно кодировке типа контента

Полученные байты должны быть декодированы с правильной кодировкой символов, которые должны интерпретироваться как текст:

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)
```

Прочитайте URLLIB онлайн: https://riptutorial.com/ru/python/topic/2645/urllib

глава 35: WebSockets

Examples

Простое эхо с aiohttp

`aiohttp` предоставляет асинхронные веб-узлы.

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())
```

Класс Wrapper с aiohttp

`aiohttp.ClientSession` может использоваться как родительский для пользовательского класса WebSocket.

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()
```

Использование Autobahn в качестве фабрики для веб-мастеров

Пакет Autobahn можно использовать для серверов сервера веб-сокетов Python.

Документация пакета Python Autobahn

Для установки обычно используется команда терминала

(Для Linux):

```
sudo pip install autobahn
```

(Для Windows):

```
python -m pip install autobahn
```

Затем в сценарии Python может быть создан простой эхо-сервер:

```
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()
```

В этом примере сервер создается на localhost (127.0.0.1) на порте 9000. Это IP-адрес прослушивания и порт. Это важная информация. Используя это, вы можете идентифицировать локальный адрес и порт вашего компьютера с вашего модема, хотя на всех маршрутизаторах у вас есть компьютер. Затем, используя Google для исследования вашего WAN IP, вы можете создать свой веб-сайт для отправки сообщений WebSocket на ваш IP-адрес WAN на порте 9000 (в этом примере).

Важно, чтобы вы перешли с вашего модема обратно, что означает, что если у вас есть маршрутизаторы, привязанные к модему, введите параметры конфигурации модема, порт, переходящий от модема к подключенному маршрутизатору, и так далее до окончательного маршрутизатора вашего компьютера подключен к тому, что передается информация,

принимаемая на модемном порту 9000 (в этом примере).

Прочитайте WebSockets онлайн: https://riptutorial.com/ru/python/topic/4751/websockets

глава 36: Абстрактное синтаксическое дерево

Examples

Анализ функций в скрипте python

Это анализирует скрипт python и для каждой определенной функции сообщает номер строки, где начинается функция, где заканчивается подпись, где заканчивается docstring, и где заканчивается определение функции.

```
#!/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)
```

Прочитайте Абстрактное синтаксическое дерево онлайн:
https://riptutorial.com/ru/python/topic/5370/абстрактное-синтаксическое-дерево

глава 37: Абстрактные базовые классы (abc)

Examples

Установка метакласса ABCMeta

Абстрактные классы - это классы, которые должны быть унаследованы, но избегать реализации определенных методов, оставляя за собой только сигнатуры методов, которые должны реализовывать подклассы.

Абстрактные классы полезны для определения и обеспечения соблюдения абстракций класса на высоком уровне, аналогично понятию интерфейсов на типизированных языках, без необходимости реализации метода.

Один концептуальный подход к определению абстрактного класса состоит в том, чтобы заглушить методы класса, а затем поднять NotImplementedError при доступе. Это предотвращает доступ дочерних классов к родительским методам без переопределения их в первую очередь. Вот так:

```
class Fruit:

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


class Apple(Fruit):
    pass


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

Создание абстрактного класса таким образом предотвращает неправильное использование методов, которые не переопределяются, и, конечно же, поощряет методы, которые должны быть определены в дочерних классах, но не обеспечивает их определения. С помощью модуля `abc` мы можем предотвратить создание экземпляров дочерних классов, когда они не могут переопределить методы абстрактного класса своих родителей и предков:

```
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.
```

Теперь можно просто подклассы и переопределить:

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

Почему / Как использовать ABCMeta и @abstractmethod

Абстрактные базовые классы (ABC) обеспечивают, какие производные классы реализуют определенные методы из базового класса.

Чтобы понять, как это работает и почему мы должны его использовать, давайте рассмотрим пример, который понравится Ван Россу. Предположим, у нас есть базовый класс «MontyPython» с двумя методами (joke & punchline), которые должны быть реализованы всеми производными классами.

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

    def punchline(self):
        raise NotImplementedError()

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

Когда мы создаем экземпляр объекта и называем его двумя методами, мы получим ошибку (как и ожидалось) с помощью метода `punchline()` .

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

Тем не менее, это все еще позволяет нам создать объект класса ArgumentClinic, не получив ошибку. На самом деле мы не получаем ошибку, пока не найдем punchline ().

Этого можно избежать, используя модуль абстрактного базового класса (ABC). Посмотрим, как это работает с тем же примером:

```
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"
```

На этот раз, когда мы пытаемся создать экземпляр объекта из неполного класса, мы сразу получаем TypeError!

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

В этом случае легко завершить класс, чтобы избежать каких-либо TypeErrors:

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

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

На этот раз, когда вы создаете экземпляр объекта, он работает!

Прочитайте Абстрактные базовые классы (abc) онлайн: https://riptutorial.com/ru/python/topic/5442/абстрактные-базовые-классы--abc-

глава 38: Альтернативы для переключения оператора с других языков

замечания

В python *нет* инструкции switch в качестве выбора языка. Был PEP (PEP-3103), охватывающий тему, которая была отклонена.

Вы можете найти множество рецептов о том, как делать свои собственные инструкции switch в python, и здесь я пытаюсь предложить наиболее разумные варианты. Вот несколько мест для проверки:

- 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

Используйте то, что предлагает язык: конструкция if / else.

Ну, если вам нужна конструкция `switch` / `case` , наиболее простой способ - использовать старую добрую конструкцию `if` / `else` :

```
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!")
```

он может выглядеть излишним и не всегда красивым, но это, безусловно, самый эффективный способ, и он выполняет эту работу:

```
>>> 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
```

Используйте опцию функций

Еще один простой способ - создать словарь функций:

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

то вы добавляете функцию по умолчанию:

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

и вы используете метод get dictionary для получения функции, заданной для проверки и запуска ее. Если значение не существует в словаре, запускается `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
```

вы также можете сделать синтаксический сахар, чтобы переключатель выглядел лучше:

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

>>> run_switch(1)
one
```

Использовать интроспекцию класса

Вы можете использовать класс для имитации структуры переключателя / случая. Следующее использует интроспекцию класса (используя `getattr()` которая разрешает строку в методе bound в экземпляре), чтобы разрешить «случайную» часть.

Затем этот метод интроспекции сглаживается методом `__call__` чтобы перегрузить оператор `()` .

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

```
    __call__ = switch
```

Затем, чтобы сделать его более привлекательным, мы подклассифицируем класс `SwitchBase` (но это может быть сделано в одном классе), и там мы определяем все `case` как методы:

```
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!')
```

поэтому мы можем, наконец, использовать его:

```
>>> 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!
```

Использование диспетчера контекстов

Другим способом, который очень читабельным и изящным, но гораздо менее эффективным, чем структура if / else, является построение класса, такого как next, который будет считывать и сохранять значение, с которым можно сравнивать, выставлять себя в контексте как вызываемый, вернет true, если оно соответствует сохраненному значению:

```
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
```

то определение случаев почти соответствует реальной конструкции `switch` / `case` (отображается внутри функции ниже, чтобы было легче показать):

```
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!')
```

Таким образом, выполнение будет:

```
>>> 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
```

Нота Бене :

- Это решение предлагается в виде *коммутационного* модуля, доступного на pypi .

Прочитайте Альтернативы для переключения оператора с других языков онлайн: https://riptutorial.com/ru/python/topic/4268/альтернативы-для-переключения-оператора-с-других-языков

глава 39: Асинхронный модуль

Examples

Синтаксис Coroutine и Delegation

Перед выпуском Python 3.5+ модуль `asyncio` использовал генераторы для имитации асинхронных вызовов и, следовательно, имел другой синтаксис, чем текущий выпуск Python 3.5.

Python 3.x 3.5

Python 3.5 представил ключевые слова `async` и `await` . Обратите внимание на отсутствие круглых скобок вокруг вызова `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

Перед Python 3.5 для определения сопрограммы использовался `@asyncio.coroutine` . Выход из выражения использовался для делегирования генератора. Обратите внимание на круглые скобки вокруг `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

Вот пример, показывающий, как две функции могут выполняться асинхронно:

```
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)
```

Асинхронные исполнители

Примечание. Использует синтаксис aync / wait Python 3.5+

`asyncio` поддерживает использование объектов `Executor` найденных в `concurrent.futures` для планирования задач асинхронно. Циклы событий имеют функцию `run_in_executor()` которая принимает параметры объекта `Executor` , `Callable` и Callable.

Планирование задания для `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))
```

В каждом цикле событий также есть слот `Executor` умолчанию, который может быть назначен `Executor` . Чтобы назначить `Executor` и назначить задачи из цикла, вы используете метод `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))
```

Существует два основных типа `Executor` в `concurrent.futures` , `ThreadPoolExecutor` и `ProcessPoolExecutor` . `ThreadPoolExecutor` содержит пул потоков, который может быть либо вручную настроен на определенное количество потоков через конструктор, либо по умолчанию равен числу ядер в машинных моментах 5. `ThreadPoolExecutor` использует пул потоков для выполнения назначенных ему задач и как правило, лучше для операций с привязкой к ЦП, а не для операций, связанных с I / O. Сравните это с `ProcessPoolExecutor` который порождает новый процесс для каждой назначенной ему задачи. `ProcessPoolExecutor` может принимать только задачи и параметры, которые могут быть выбраны. Наиболее распространенными задачами, которые не учитываются, являются методы объектов. Если вы должны запланировать метод объекта как задачу в `Executor` вы должны использовать `ThreadPoolExecutor` .

Использование UVLoop

`uvloop` - это реализация для `asyncio.AbstractEventLoop` на основе libuv (Используется nodejs). Он совместим с 99% функций `asyncio` и намного быстрее, чем традиционный `asyncio.EventLoop` . `uvloop` в настоящее время недоступен в Windows, установите его с помощью `pip install uvloop` .

```
import asyncio
import uvloop

if __name__ == "__main__":
    asyncio.set_event_loop(uvloop.new_event_loop())
    # Do your stuff here ...
```

Можно также изменить фабрику циклов событий, установив `EventLoopPolicy` в ту, которая находится в `uvloop` .

```
import asyncio
import uvloop

if __name__ == "__main__":
    asyncio.set_event_loop_policy(uvloop.EventLoopPolicy())
    loop = asyncio.new_event_loop()
```

Примитив синхронизации: событие

концепция

Используйте `Event` чтобы **синхронизировать планирование нескольких сопрограмм** .

Проще говоря, событие похоже на выстрел из пушки в бегущей гонке: он позволяет бегунам покинуть стартовые блоки.

пример

```
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)
```

Выход:

> Ожидание потребителя B
> Потребитель Ожидание
> СОБЫТИЕ
> Потребитель B активирован

Потребитель A

Простой веб-узел

Здесь мы делаем простой echo websocket, используя `asyncio` . Мы определяем сопрограммы для подключения к серверу и отправки / получения сообщений. Передача websocket выполняется в `main` сопрограмме, которая управляется контуром событий. Этот пример изменен из предыдущего сообщения .

```
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())
```

Общее Заблуждение о asyncio

вероятно, наиболее распространенное заблуждение о `asnycio` является то , что она позволяет запускать любую задачу параллельно - обходя GIL (глобальная блокировка интерпретатора) и , следовательно , выполнять блокирующие работу параллельно (в отдельных потоках). это **не так** !

`asyncio` (и библиотеки, созданные для совместной работы с `asyncio`), построены на сопрограмме: функции, которые (совместно) возвращают поток управления вызывающей функции. `asyncio.sleep` приведенных выше примерах обратите внимание на `asyncio.sleep` . это пример неблокирующей сопрограммы, которая ждет «в фоновом режиме» и возвращает управляющий поток вызывающей функции (при вызове с `await`). `time.sleep` является примером функции блокировки. поток выполнения программы будет просто остановлен и только вернется после окончания `time.sleep` .

реальным примером является библиотека `requests` которая состоит (пока) только для функций блокировки. нет параллелизма, если вы вызываете любую из своих функций в `asyncio` . `aiohttp` с другой стороны, был построен с `asyncio` . его сопрограммы будут выполняться одновременно.

- если у вас есть длительные задачи с привязкой к процессору, которые вы хотели бы запускать параллельно, `asyncio` **не** для вас. для этого вам нужны `threads` или `multiprocessing` .
- если у вас запущены задания с привязкой к IO, вы *можете* запускать их одновременно с помощью `asyncio` .

Прочитайте Асинхронный модуль онлайн: https://riptutorial.com/ru/python/topic/1319/асинхронный-модуль

глава 40: аудио

Examples

Аудио с пиглетом

```
import pyglet
audio = pyglet.media.load("audio.wav")
audio.play()
```

Для получения дополнительной информации см. Pyglet

Работа с файлами WAV

winsound

- Среда Windows

```
import winsound
winsound.PlaySound("path_to_wav_file.wav", winsound.SND_FILENAME)
```

волна

- Поддержка моно / стерео
- Не поддерживает сжатие / декомпрессию

```
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)
```

Преобразование любого звукового файла с помощью python и 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()
```

нота:

- http://superuser.com/questions/507386/why-would-i-choose-libav-over-ffmpeg-or-is-there-even-a-difference
- Каковы различия и сходства между ffmpeg, libav и avconv?

Воспроизведение звуков Windows

Windows предоставляет явный интерфейс, через который модуль `winsound` позволяет вам воспроизводить звуковые сигналы с заданной частотой и продолжительностью.

```
import winsound
freq = 2500 # Set frequency To 2500 Hertz
dur = 1000 # Set duration To 1000 ms == 1 second
winsound.Beep(freq, dur)
```

Прочитайте аудио онлайн: https://riptutorial.com/ru/python/topic/8189/аудио

глава 41: Безопасность и криптография

Вступление

Python, являясь одним из самых популярных языков в области компьютерной и сетевой безопасности, обладает большим потенциалом в области безопасности и криптографии. В этом разделе рассматриваются криптографические функции и реализации в Python от его использования в компьютерной и сетевой безопасности до алгоритмов хэширования и шифрования / дешифрования.

Синтаксис

- hashlib.new (имя)
- hashlib.pbkdf2_hmac (имя, пароль, соль, раунды, dklen = нет)

замечания

Многие из методов в `hashlib` потребуют от вас передавать значения, интерпретируемые как буферы байтов, а не строки. Это относится к `hashlib.new().update()` а также `hashlib.pbkdf2_hmac` . Если у вас есть строка, вы можете преобразовать ее в буфер байта, добавив символ `b` в начало строки:

```
 "This is a string"
b"This is a buffer of bytes"
```

Examples

Вычисление сообщения

Модуль `hashlib` позволяет создавать генераторы дайджеста сообщений с помощью `new` метода. Эти генераторы превратят произвольную строку в дайджест фиксированной длины:

```
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-
='
```

Обратите внимание, что вы можете вызывать `update` произвольным количеством раз,

прежде чем вызывать `digest` который полезен для хеширования большого фрагмента файла куском. Вы также можете получить дайджест в шестнадцатеричном формате с помощью `hexdigest` :

```
h.hexdigest()
# ==> '2edfdada56525b1290ff16fb1744cfb482dd2914ffbcb649790c0e589e462d3d'
```

Доступные алгоритмы хеширования

`hashlib.new` требует имя алгоритма, когда вы вызываете его для создания генератора. Чтобы узнать, какие алгоритмы доступны в текущем интерпретаторе Python, используйте `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'}
```

Возвращаемый список будет отличаться в зависимости от платформы и интерпретатора; убедитесь, что вы проверили, что ваш алгоритм доступен.

Существуют также некоторые алгоритмы, которые *гарантированно* будут доступны на всех платформах и интерпретаторах, которые доступны с использованием `hashlib.algorithms_guaranteed` :

```
hashlib.algorithms_guaranteed
# ==> {'sha256', 'sha384', 'sha1', 'sha224', 'md5', 'sha512'}
```

Безопасное шифрование паролей

Алгоритм PBKDF2 , `hashlib` модулем `hashlib` может использоваться для выполнения безопасного хеширования паролей. Хотя этот алгоритм не может предотвратить атаки грубой силы, чтобы восстановить исходный пароль из хранимого хеша, он делает такие атаки очень дорогими.

```
import hashlib
import os

salt = os.urandom(16)
hash = hashlib.pbkdf2_hmac('sha256', b'password', salt, 100000)
```

PBKDF2 может работать с любым алгоритмом дайджеста, в приведенном выше примере используется SHA256, который обычно рекомендуется. Случайную соль следует хранить вместе с хешированным паролем, вам понадобится снова, чтобы сравнить введенный пароль с сохраненным хэшем. Очень важно, чтобы каждый пароль хэшировал с другой солью. Что касается количества раундов, рекомендуется установить его как можно выше

для вашего приложения .

Если вы хотите получить результат в шестнадцатеричном формате, вы можете использовать модуль `binascii` :

```
import binascii
hexhash = binascii.hexlify(hash)
```

Примечание . В то время как PBKDF2 не плох, bcrypt и особенно scrypt считаются более сильными против атак с грубой силой. На данный момент эта часть не является стандартной библиотекой Python.

Хеширование файлов

Хэш - это функция, которая преобразует последовательность переменных длины байтов в последовательность фиксированной длины. Хеширование файлов может быть выгодным по многим причинам. Хэши могут использоваться для проверки идентичности двух файлов или проверки того, что содержимое файла не было повреждено или изменено.

Вы можете использовать `hashlib` для генерации хэша для файла:

```
import hashlib

hasher = hashlib.new('sha256')
with open('myfile', 'r') as f:
    contents = f.read()
    hasher.update(contents)

print hasher.hexdigest()
```

Для больших файлов можно использовать буфер фиксированной длины:

```
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())
```

Симметричное шифрование с использованием pycrypto

Встроенная криптосистема Python в настоящее время ограничена хешированием. Для шифрования требуется сторонний модуль, например, pycrypto . Например, он обеспечивает алгоритм AES, который считается современным для симметричного шифрования. Следующий код будет шифровать данное сообщение, используя кодовую фразу:

```
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)
```

Алгоритм AES использует три параметра: ключ шифрования, вектор инициализации (IV) и фактическое сообщение для шифрования. Если у вас случайно сформированный AES-ключ, вы можете использовать его напрямую и просто генерировать случайный вектор инициализации. Однако кодовая фраза не имеет нужного размера и не рекомендуется использовать ее напрямую, учитывая, что она не является действительно случайной и, следовательно, имеет сравнительно небольшую энтропию. Вместо этого мы используем встроенную реализацию алгоритма PBKDF2 для генерации 128-битного вектора инициализации и 256-битного ключа шифрования из пароля.

Обратите внимание на случайную соль, которая важна для того, чтобы иметь различный вектор инициализации и ключ для каждого зашифрованного сообщения. Это гарантирует, в частности, что два равных сообщения не приведут к идентичному зашифрованному тексту, но также предотвращают повторное использование злоумышленниками работы, угадывающей одну кодовую фразу на сообщениях, зашифрованных другой ключевой фразой. Эта соль должна храниться вместе с зашифрованным сообщением, чтобы получить один и тот же вектор инициализации и ключ для дешифрования.

Следующий код расшифровывает наше сообщение еще раз:

```
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:])
```

Создание сигнатур RSA с использованием pycrypto

RSA может использоваться для создания сигнатуры сообщения. Действительная подпись может генерироваться только с доступом к частному ключу RSA, с другой стороны, проверка с помощью всего лишь соответствующего открытого ключа. Поэтому, пока другая

сторона знает ваш открытый ключ, они могут проверить сообщение, подписанное вами и неизменным, - например, подход, используемый для электронной почты. В настоящее время для этой функции требуется сторонний модуль, такой как 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)
```

Проверка подписи работает аналогично, но использует открытый ключ, а не закрытый ключ:

```
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')
```

Примечание . В приведенных выше примерах используется алгоритм подписи PKCS # 1 v1.5, который очень распространен. pycrypto также реализует новый PSS-алгоритм PKCS # 1, заменяя `PKCS1_v1_5` на `PKCS1_PSS` в примерах, должен работать, если вы хотите использовать этот. В настоящее время, похоже, нет оснований для его использования .

Асимметричное шифрование RSA с использованием pycrypto

Преимущество асимметричного шифрования состоит в том, что сообщение может быть зашифровано без обмена секретным ключом с получателем сообщения. Отправитель просто должен знать открытый ключ получателей, это позволяет шифровать сообщение таким образом, что только назначенный получатель (имеющий соответствующий закрытый ключ) может его расшифровать. В настоящее время для этой функции требуется

сторонний модуль, такой как 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)
```

Получатель может расшифровать сообщение, если у них есть правильный закрытый ключ:

```
with open('privkey.pem', 'rb') as f:
    key = RSA.importKey(f.read())
cipher = PKCS1_OAEP.new(key)
decrypted = cipher.decrypt(encrypted)
```

Примечание . В приведенных выше примерах используется схема шифрования OAEP PKCS # 1. pycrypto также реализует схему шифрования PKCS # 1 v1.5, однако это не рекомендуется для новых протоколов, однако из-за известных оговорок .

Прочитайте Безопасность и криптография онлайн: https://riptutorial.com/ru/python/topic/2598/безопасность-и-криптография

глава 42: Библиотека подпроцессов

Синтаксис

- subprocess.call (args, *, stdin = None, stdout = None, stderr = None, shell = False, timeout = None)
- subprocess.Popen (args, bufsize = -1, executable = None, stdin = None, stdout = None, stderr = None, preexec_fn = None, close_fds = True, shell = False, cwd = None, env = None, universal_newlines = False , startupinfo = None, creationflags = 0, restore_signals = True, start_new_session = False, pass_fds = ())

параметры

параметр	подробности
`args`	Один исполняемый файл или последовательность исполняемых файлов и аргументов - `'ls'` , `['ls', '-la']`
`shell`	Выполнить под оболочкой? По умолчанию оболочка для `/bin/sh` в POSIX.
`cwd`	Рабочий каталог дочернего процесса.

Examples

Вызов внешних команд

Простейшим вариантом использования является использование функции `subprocess.call` . Он принимает список в качестве первого аргумента. Первым элементом в списке должно быть внешнее приложение, которое вы хотите вызвать. Другие элементы в списке - это аргументы, которые будут переданы этому приложению.

```
subprocess.call([r'C:\path\to\app.exe', 'arg1', '--flag', 'arg'])
```

Для команд оболочки установите `shell=True` и укажите команду как строку вместо списка.

```
subprocess.call('echo "Hello, world"', shell=True)
```

Обратите внимание, что две команды выше возвращают только `exit status` выхода подпроцесса. Кроме того, обратите внимание при использовании `shell=True` поскольку он обеспечивает проблемы безопасности (см. Здесь).

Если вы хотите получить стандартный вывод подпроцесса, замените `subprocess.call` на

`subprocess.check_output` . Для более продвинутого использования обратитесь к этому .

Больше гибкости с Popen

Использование `subprocess.Popen` дает более мелкозернистый контроль запущенных процессов, чем `subprocess.call` .

Запуск подпроцесса

```
process = subprocess.Popen([r'C:\path\to\app.exe', 'arg1', '--flag', 'arg'])
```

Подпись для `Popen` очень похожа на функцию `call` ; однако `Popen` немедленно вернется, вместо того, чтобы ждать завершения подпроцесса, например `call` .

Ожидание завершения подпроцесса

```
process = subprocess.Popen([r'C:\path\to\app.exe', 'arg1', '--flag', 'arg'])
process.wait()
```

Чтение вывода из подпроцесса

```
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
```

Интерактивный доступ к работающим подпроцессам

Вы можете читать и писать на `stdin` и `stdout` даже если подпроцесс не завершен. Это может быть полезно при автоматизации функций в другой программе.

Запись в подпроцесс

```
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.
```

Однако, если вам нужен только один набор входных и выходных данных, а не динамическое взаимодействие, вы должны использовать `communicate()` вместо прямого доступа к `stdin` и `stdout` .

Чтение потока из подпроцесса

Если вы хотите видеть вывод подпроцесса по очереди, вы можете использовать следующий фрагмент:

```
process = subprocess.Popen(<your_command>, stdout=subprocess.PIPE)
while process.poll() is None:
    output_line = process.stdout.readline()
```

в случае, если вывод подкоманды не имеет символа EOL, приведенный выше фрагмент не работает. Затем вы можете прочитать выходной символ по символу следующим образом:

```
process = subprocess.Popen(<your_command>, stdout=subprocess.PIPE)
while process.poll() is None:
    output_line = process.stdout.read(1)
```

`1` указывается в качестве аргумента к `read` методу говорит чтение для чтения 1 знака времени. Вы можете указать, чтобы читать столько символов, сколько хотите, используя другое число. Отрицательное число или 0 указывает `read` считаться одной строкой до тех пор, пока не встретится EOF (см. Здесь).

В обоих вышеописанных фрагментах `process.poll()` имеет значение `None` пока не завершится подпроцесс. Это используется для выхода из цикла, когда больше нет выхода для чтения.

Та же процедура может быть применена к `stderr` подпроцесса.

Как создать аргумент списка команд

Метод подпроцесса, который позволяет запускать команды, нуждается в команде в виде списка (по крайней мере, с использованием `shell_mode=True`).

Правила создания списка не всегда просты, особенно с помощью сложных команд. К счастью, есть очень полезный инструмент, который позволяет сделать это: `shlex` . Самый простой способ создания списка, который будет использоваться в качестве команды, следующий:

```
import shlex
cmd_to_subprocess = shlex.split(command_used_in_the_shell)
```

Простой пример:

```
import shlex
shlex.split('ls --color -l -t -r')

out: ['ls', '--color', '-l', '-t', '-r']
```

Прочитайте Библиотека подпроцессов онлайн: https://riptutorial.com/ru/python/topic/1393/библиотека-подпроцессов

глава 43: Булевы операторы

Examples

а также

Оценивает второй аргумент тогда и только тогда, когда оба аргумента являются правдивыми. В противном случае вычисляется первый аргумент false.

```
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)
```

`1` в приведенном выше примере можно изменить на любое правдоподобное значение, а `0` можно изменить на любое значение false.

или же

Оценивает первый аргумент правды, если один из аргументов правдив. Если оба аргумента ложны, вычисляется второй аргумент.

```
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)
```

`1` в приведенном выше примере можно изменить на любое правдоподобное значение, а `0` можно изменить на любое значение false.

не

Он возвращает противоположность следующего утверждения:

```
x = True
y = not x # y = False

x = False
y = not x # y = True
```

Оценка короткого замыкания

Python минимально оценивает булевы выражения.

```
>>> 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
```

`and` и` or` не гарантированно возвращают логическое

Когда вы используете `or` , он либо вернет первое значение в выражении, если оно истинно, иначе оно будет вслепую возвращать второе значение. То есть `or` эквивалентно:

```
def or_(a, b):
    if a:
        return a
    else:
        return b
```

Для `and` , оно вернет свое первое значение, если оно ложно, иначе оно вернет последнее значение:

```
def and_(a, b):
    if not a:
        return a
    else:
        return b
```

Простой пример

В Python вы можете сравнить один элемент, используя два бинарных оператора: один с обеих сторон:

```
if 3.14 < x < 3.142:
    print("x is near pi")
```

На многих языках (в большинстве?) Это будет оцениваться способом, противоречащим обычной математике: `(3.14 < x) < 3.142` , но в Python он рассматривается как `3.14 < x and x < 3.142` , как и большинство не-программистов ожидал бы.

Прочитайте Булевы операторы онлайн: https://riptutorial.com/ru/python/topic/1731/булевы-операторы

глава 44: Введение в RabbitMQ с использованием AMQPStorm

замечания

Последняя версия AMQPStorm доступна на pypi или вы можете установить ее с помощью pip

```
pip install amqpstorm
```

Examples

Как потреблять сообщения от RabbitMQ

Начните с импорта библиотеки.

```
from amqpstorm import Connection
```

При использовании сообщений мы сначала должны определить функцию обработки входящих сообщений. Это может быть любая вызываемая функция и должна принимать объект сообщения или кортеж сообщения (в зависимости от параметра `to_tuple` определенного в `start_consuming`).

Помимо обработки данных из входящего сообщения, нам также необходимо подтвердить или отклонить сообщение. Это важно, так как нам нужно сообщить RabbitMQ, что мы правильно получили и обработали сообщение.

```
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)
```

Затем нам нужно настроить соединение с сервером RabbitMQ.

```
connection = Connection('127.0.0.1', 'guest', 'guest')
```

После этого нам нужно настроить канал. Каждое соединение может иметь несколько каналов, и, как правило, при выполнении многопоточных задач рекомендуется (но не обязательно) иметь по одному на поток.

```
channel = connection.channel()
```

Как только мы настроим наш канал, нам нужно сообщить RabbitMQ, что мы хотим начать использовать сообщения. В этом случае мы будем использовать нашу ранее определенную функцию `on_message` для обработки всех наших потребляемых сообщений.

Очередь, которую мы будем слушать на сервере RabbitMQ, будет `simple_queue` , и мы также сообщаем RabbitMQ, что мы будем признавать все входящие сообщения, как только мы с ними закончим.

```
channel.basic.consume(callback=on_message, queue='simple_queue', no_ack=False)
```

Наконец, нам нужно запустить цикл ввода-вывода, чтобы начать обработку сообщений, предоставляемых сервером RabbitMQ.

```
channel.start_consuming(to_tuple=False)
```

Как публиковать сообщения в RabbitMQ

Начните с импорта библиотеки.

```
from amqpstorm import Connection
from amqpstorm import Message
```

Затем нам нужно открыть соединение с сервером RabbitMQ.

```
connection = Connection('127.0.0.1', 'guest', 'guest')
```

После этого нам нужно настроить канал. Каждое соединение может иметь несколько каналов, и, как правило, при выполнении многопоточных задач рекомендуется (но не обязательно) иметь по одному на поток.

```
channel = connection.channel()
```

Как только мы настроим наш канал, мы можем начать готовить наше сообщение.

```
# Message Properties.
properties = {
    'content_type': 'text/plain',
    'headers': {'key': 'value'}
```

```
}

# Create the message.
message = Message.create(channel=channel, body='Hello World!', properties=properties)
```

Теперь мы можем опубликовать сообщение, просто позвонив в `publish` и предоставив `routing_key` . В этом случае мы отправим сообщение в очередь с именем `simple_queue` .

```
message.publish(routing_key='simple_queue')
```

Как создать задержанную очередь в RabbitMQ

Сначала нам нужно настроить два основных канала: один для основной очереди и один для очереди задержки. В моем примере в конце я включаю пару дополнительных флагов, которые не требуются, но делает код более надежным; таких как `confirm delivery` , `delivery_mode` и `durable` . Вы можете найти более подробную информацию о них в RabbitMQ руководстве .

После того, как мы установили каналы, мы добавим привязку к основному каналу, которую мы можем использовать для отправки сообщений с канала задержки в нашу основную очередь.

```
channel.queue.bind(exchange='amq.direct', routing_key='hello', queue='hello')
```

Затем нам нужно настроить наш канал задержки для пересылки сообщений в основную очередь по истечении срока их действия.

```
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)*

  Обычно это используется для автоматического удаления старых сообщений в очереди по истечении определенной продолжительности, но, добавив два необязательных аргумента, мы можем изменить это поведение, и вместо этого этот параметр определяет в миллисекундах, сколько сообщений будет оставаться в очереди ожидания.

- x анкерного письмо маршрутизации ключ

  Эта переменная позволяет нам передавать сообщение в другую очередь, как только они истекли, вместо того, чтобы по умолчанию полностью удалять его.

- x-буквальный обмен

Эта переменная определяет, какой Exchange используется для передачи сообщения с hello_delay в очередь hello.

Публикация в очередь ожидания

Когда мы закончим настройку всех основных параметров Pika, вы просто отправляете сообщение в очередь ожидания, используя базовую публикацию.

```
delay_channel.basic.publish(exchange='',
                            routing_key='hello_delay',
                            body='test',
                            properties={'delivery_mod': 2})
```

После выполнения сценария вы должны увидеть следующие очереди, созданные в вашем модуле управления RabbitMQ.

Overview					Messages			Messa	
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	

Пример.

```
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")
```

Прочитайте Введение в RabbitMQ с использованием AMQPStorm онлайн: https://riptutorial.com/ru/python/topic/3373/введение-в-rabbitmq-с-использованием-amqpstorm

глава 45: Ввод, подмножество и вывод внешних файлов данных с использованием Pandas

Вступление

В этом разделе показан базовый код для чтения, подстановки и записи внешних файлов данных с использованием панд.

Examples

Основной код для импорта, подмножества и записи файлов внешних данных с использованием Pandas

```
# 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)
```

Прочитайте Ввод, подмножество и вывод внешних файлов данных с использованием Pandas онлайн: https://riptutorial.com/ru/python/topic/8854/ввод--подмножество-и-вывод-внешних-файлов-данных-с-использованием-pandas

глава 46: вдавливание

Examples

Ошибки отступов

Интервал должен быть четным и равномерным. Неправильное отступы могут вызвать `IndentationError` или заставить программу сделать что-то неожиданное. В следующем примере возникает параметр `IndentationError` :

```
a = 7
if a > 5:
  print "foo"
else:
  print "bar"
 print "done"
```

Или, если строка, следующая за двоеточием, не имеет `IndentationError` будет также поднят параметр `IndentationError` :

```
if True:
print "true"
```

Если вы добавите отступ, где он не принадлежит, будет поднят параметр `IndentationError` :

```
if  True:
    a = 6
        b = 5
```

Если вы забудете отменить абзац, функции могут быть потеряны. В этом примере вместо ожидаемого `False` возвращается `None` :

```
def isEven(a):
    if a%2 ==0:
        return True
        #this next line should be even with the if
        return False
print isEven(7)
```

Простой пример

Для Python Гвидо ван Россум основывал группировку высказываний на отступы. Причины этого объясняются в первом разделе «Часто задаваемые вопросы о дизайне и истории Python» . Colons, `:` , используются для объявления отступов кода , например, в следующем примере:

```
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])
```

Пробелы или вкладки?

Рекомендуемый отступ - 4 пробела, но можно использовать вкладки или пробелы, если они совместимы. ***Не смешивайте вкладки и пробелы в Python, так*** как это вызовет ошибку в Python 3 и может вызвать ошибки в Python 2 .

Как отступы разобраны

Пробел обрабатывается лексическим анализатором перед анализом.

Лексический анализатор использует стек для хранения уровней отступов. В начале стек содержит только значение 0, которое является самым левым положением. Всякий раз, когда начинается вложенный блок, новый уровень отступов помещается в стек, а токен INDENT вставляется в поток токенов, который передается парсеру. В строке не может быть более одного «INDENT» токена (`IndentationError`).

Когда строка встречается с меньшим уровнем отступов, значения выводятся из стека до тех пор, пока значение не будет на вершине, которое равно новому уровню отступов (если ни один не найден, возникает синтаксическая ошибка). Для каждого значения выдается токен «DEDENT». Очевидно, что могут быть несколько токенов «DEDENT» подряд.

Лексический анализатор пропускает пустые строки (содержащие только пробелы и, возможно, комментарии), и никогда не будет генерировать для них токены «INDENT» или « DEDENT».

В конце исходного кода токены «DEDENT» генерируются для каждого уровня отступов, оставшегося в стеке, до тех пор, пока не останется только 0.

Например:

```
if foo:
    if bar:
        x = 42
else:
    print foo
```

анализируется как:

```
<if> <foo> <:>                    [0]
<INDENT> <if> <bar> <:>           [0, 4]
<INDENT> <x> <=> <42>             [0, 4, 8]
<DEDENT> <DEDENT> <else> <:>      [0]
<INDENT> <print> <foo>            [0, 2]
<DEDENT>
```

Парсер, чем обрабатывает токены «INDENT» и «DEDENT» в качестве разделителей блоков.

Прочитайте вдавливание онлайн: https://riptutorial.com/ru/python/topic/2597/вдавливание

глава 47: Веб-скребок с Python

Вступление

Веб-соскабливание - это автоматизированный программный процесс, посредством которого данные могут постоянно «очищаться» от веб-страниц. Также известный как очистка экрана или сборка веб-страниц, веб-скребок может предоставлять мгновенные данные с любой общедоступной веб-страницы. На некоторых сайтах веб-скребок может быть незаконным.

замечания

Полезные пакеты Python для веб-соскабливания (в алфавитном порядке)

Выполнение запросов и сбор данных

`requests`

Простой, но мощный пакет для создания HTTP-запросов.

`requests-cache`

Кэширование `requests` ; данные кэширования очень полезны. В разработке это означает, что вы можете без необходимости удалять сайт. При запуске реальной коллекции это означает, что если ваш скребок сработает по какой-то причине (возможно, вы не обрабатывали какой-то необычный контент на сайте ...? Возможно, сайт опустился ...?), Вы можете очень быстро повторить сбор откуда вы остановились.

`scrapy`

Полезно для создания веб-сканеров, где вам нужно что-то более мощное, чем использование `requests` и итерация по страницам.

`selenium`

Python для Selenium WebDriver для автоматизации браузера. Использование `requests` для прямого HTTP-запроса часто бывает проще для получения веб-страниц. Однако это остается полезным инструментом, когда невозможно воспроизвести желаемое поведение сайта с использованием только `requests` , особенно когда JavaScript требуется для отображения элементов на странице.

Разбор HTML

BeautifulSoup

Запросить HTML и XML-документы, используя ряд различных парсеров (встроенный HTML-анализатор `html5lib` , `lxml` , `lxml` или `lxml.html`)

lxml

Процессы HTML и XML. Может использоваться для запроса и выбора содержимого из HTML-документов с помощью селекторов CSS и XPath.

Examples

Основной пример использования запросов и lxml для очистки некоторых данных

```
# 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()
```

Ведение сеанса веб-очистки с запросами

Рекомендуется сохранять сеанс веб-очистки для сохранения файлов cookie и других параметров. Кроме того, это может привести в *повышению производительности* , так как `requests.Session` повторно использует TCP - соединение с узлом:

```
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)
```

Скребок с использованием схемы Scrapy

Сначала вам нужно создать новый проект Scrapy. Введите каталог, в котором вы хотите сохранить код и выполните его:

```
scrapy startproject projectName
```

Чтобы очистить, нам нужен паук. Пауки определяют, как будет очищаться определенный сайт. Вот код для паука, который следует за ссылками на верхние голосовые вопросы по StackOverflow и сбрасывает некоторые данные с каждой страницы (источник):

```
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,
        }
```

Сохраните свои классы `projectName\spiders` каталоге `projectName\spiders` . В этом случае - `projectName\spiders\stackoverflow_spider.py` .

Теперь вы можете использовать своего паука. Например, попробуйте запустить (в каталоге проекта):

```
scrapy crawl stackoverflow
```

Изменить пользовательский агент Scrapy

Иногда пользовательский агент `"Scrapy/VERSION (+http://scrapy.org)"` по умолчанию (

`"Scrapy/VERSION (+http://scrapy.org)"`) блокируется хостом. Чтобы изменить пользовательский агент по умолчанию, откройте **settings.py** , раскомментируйте и отредактируйте следующую строку, что бы вы ни пожелали.

```
#USER_AGENT = 'projectName (+http://www.yourdomain.com)'
```

Например

```
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'
```

Скребок с использованием 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
```

Скребок с использованием Selenium WebDriver

Некоторые веб-сайты не любят соскабливаться. В этих случаях вам может потребоваться смоделировать реального пользователя, работающего с браузером. Selenium запускает и управляет веб-браузером.

```
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)
```

Селен может сделать гораздо больше. Он может изменять файлы cookie браузера, заполнять формы, имитировать щелчки мыши, делать скриншоты веб-страниц и запускать собственный JavaScript.

Простая загрузка веб-контента с помощью urllib.request

Стандартный библиотечный модуль `urllib.request` можно использовать для загрузки веб-контента:

```
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)
```

Аналогичный модуль также доступен в Python 2 .

Скребок с завитом

импорт:

```
from subprocess import Popen, PIPE
from lxml import etree
from io import StringIO
```

Загрузка:

```
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` : silent скачать

`-A` : флаг агента пользователя

Синтаксический:

```
tree = etree.parse(StringIO(result), etree.HTMLParser())
divs = tree.xpath('//div')
```

Прочитайте Веб-скребок с Python онлайн: https://riptutorial.com/ru/python/topic/1792/веб-скребок-с-python

глава 48: Визуализация данных с помощью Python

Examples

Matplotlib

Matplotlib - это математическая библиотека построения шрифтов для Python, которая предоставляет множество различных функций построения графиков.

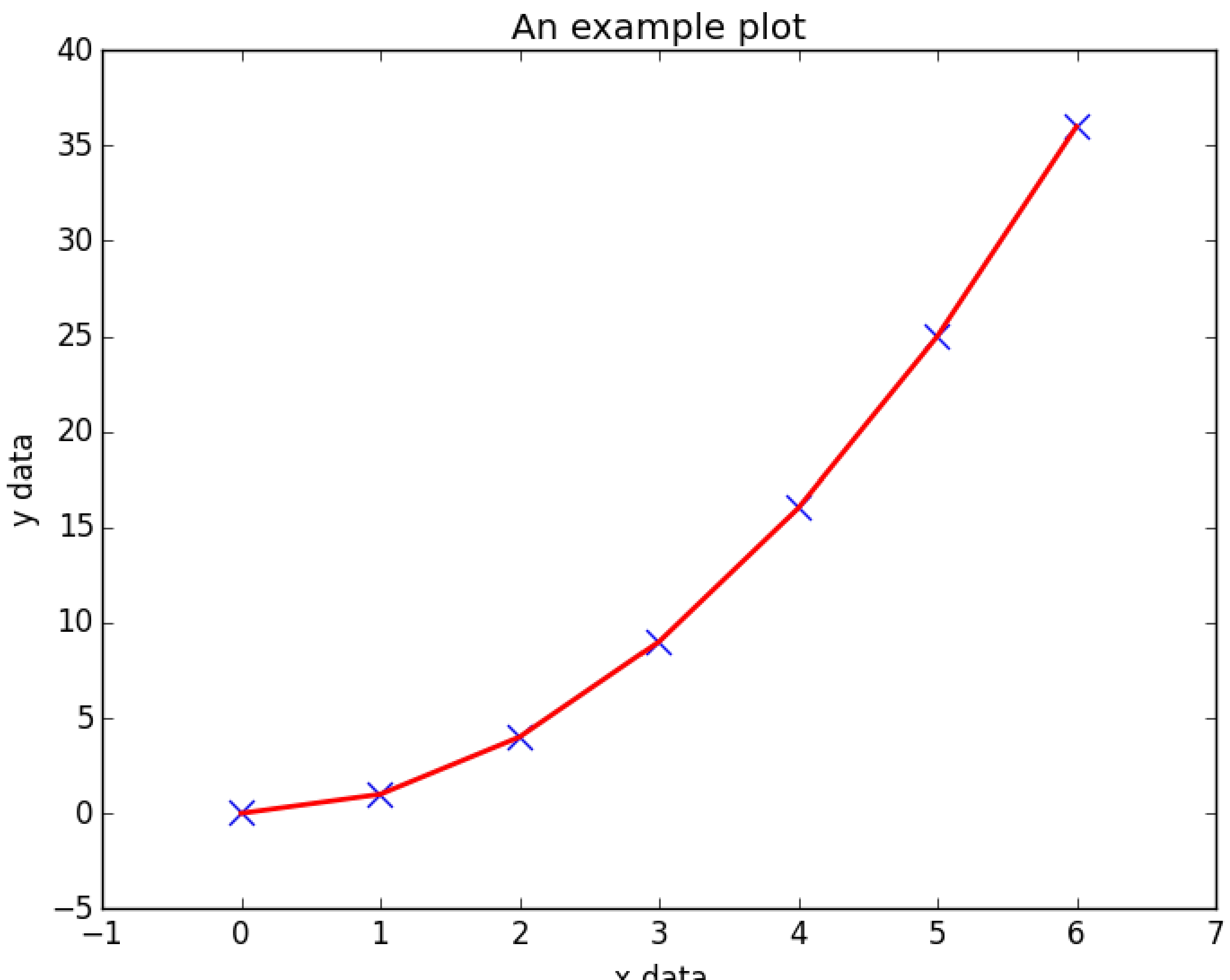
Документацию Matplotlib можно найти здесь , с SO Docs будет доступна здесь .

Matplotlib предоставляет два различных метода построения, хотя они взаимозаменяемы по большей части:

- Во-первых, matplotlib предоставляет интерфейс `pyplot` , прямой и простой в использовании интерфейс, который позволяет строить сложные графики в стиле MATLAB.
- Во-вторых, matplotlib позволяет пользователю управлять различными аспектами (осями, линиями, тиками и т. Д.) Непосредственно с использованием объектной системы. Это сложнее, но позволяет полностью контролировать весь участок.

Ниже приведен пример использования интерфейса `pyplot` для построения некоторых сгенерированных данных:

```
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()
```



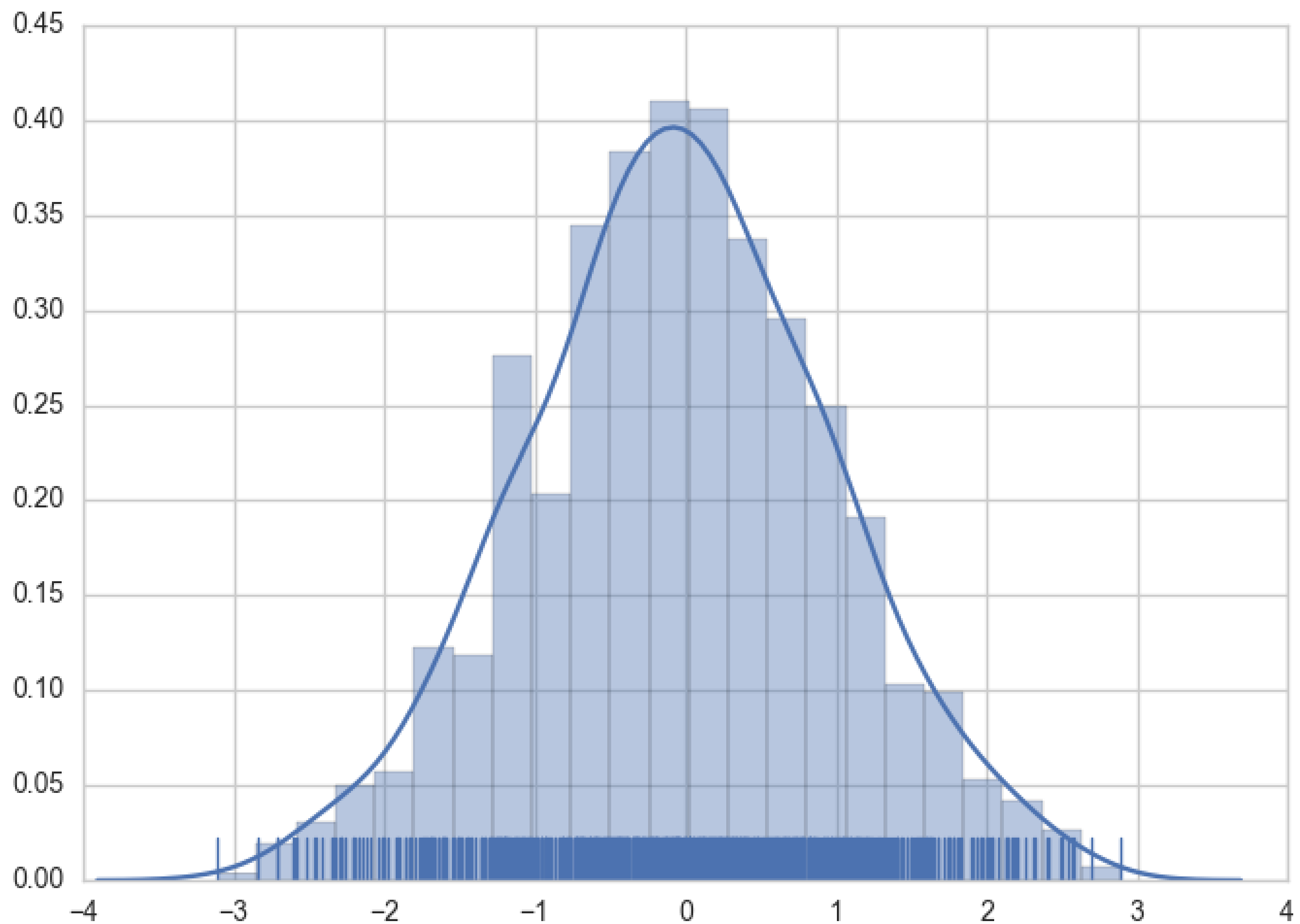
Обратите внимание, что `plt.show()` как известно, проблематичен в некоторых средах из-за запуска `matplotlib.pyplot` в интерактивном режиме, и если это так, поведение блокировки может быть явно переопределено путем передачи необязательного аргумента `plt.show(block=True)` , чтобы облегчить эту проблему.

рожденное море

Seaborn - обертка вокруг Matplotlib, которая упрощает создание общих статистических участков. Список поддерживаемых графиков включает одномерные и двумерные графики распределения, графики регрессии и ряд методов построения категориальных переменных. Полный список участков, предоставляемых Seaborn, находится в их справочной информации по API .

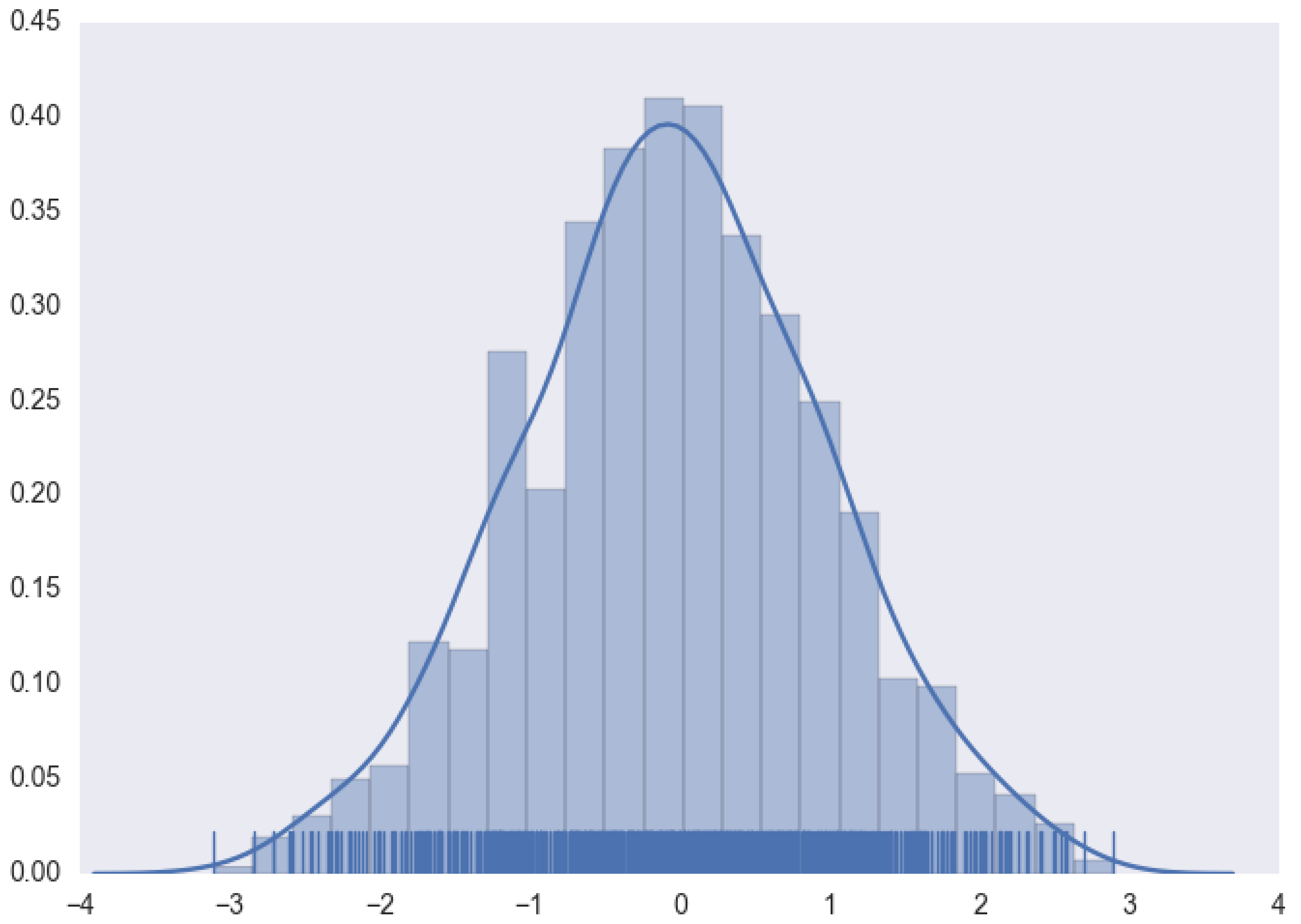
Создание графиков в Seaborn так же просто, как вызов соответствующей функции графического отображения. Ниже приведен пример создания гистограммы, оценки плотности ядра и графика для случайно генерируемых данных.

```
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)
```



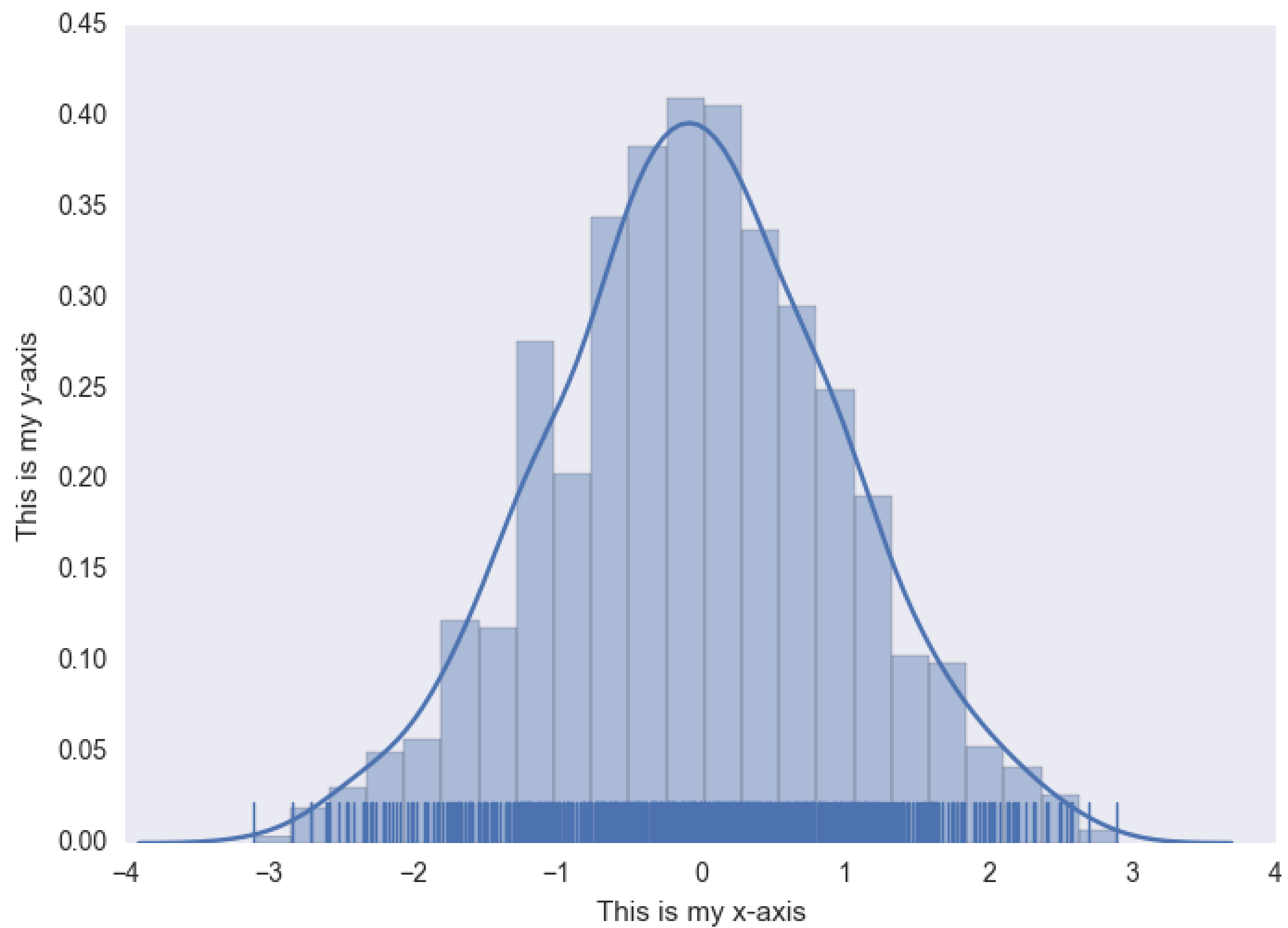
Стиль сюжета также можно контролировать с помощью декларативного синтаксиса.

```
# 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)
```



В качестве дополнительного бонуса, нормальные команды matplotlib все еще могут быть применены к сайтам Seaborn. Вот пример добавления названий осей в ранее созданную гистограмму.

```
# 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 - инструмент 3D-визуализации для научных данных. Он использует набор инструментов визуализации или VTK под капотом. Используя мощность VTK , **MayaVI** способен производить множество 3-мерных графиков и рисунков. Он доступен как отдельное программное приложение, а также как библиотека. Подобно Matplotlib , эта библиотека предоставляет интерфейс объектно-ориентированного программирования для создания графиков без необходимости знать о **VTK** .

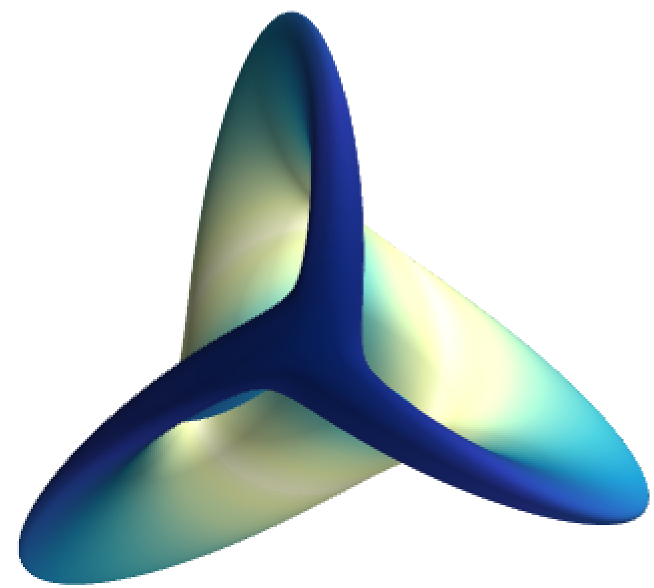
MayaVI доступен только в серии Python 2.7x! Ожидается, что он скоро появится в серии Python 3-x! (Хотя некоторый успех замечен при использовании его зависимостей в Python 3)

Документацию можно найти здесь . Некоторые примеры галереи найдены здесь

Вот образец, созданный с использованием **MayaVI** из документации.

```
# 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 - современная платформа для построения графиков и визуализации данных. Полезно для создания различных графиков, особенно для наук о данных, **Plotly** доступен в виде библиотеки для **Python** , **R** , **JavaScript** , **Julia** и **MATLAB** . Он также может использоваться как веб-приложение с этими языками.

Пользователи могут установить библиотеку plotly и использовать ее в автономном режиме после аутентификации пользователей. Установка этой библиотеки и в автономном режиме аутентификации дается здесь . Кроме того, графики могут быть сделаны и в **ноутбуках Jupyter** .

Для использования этой библиотеки требуется учетная запись с именем пользователя и

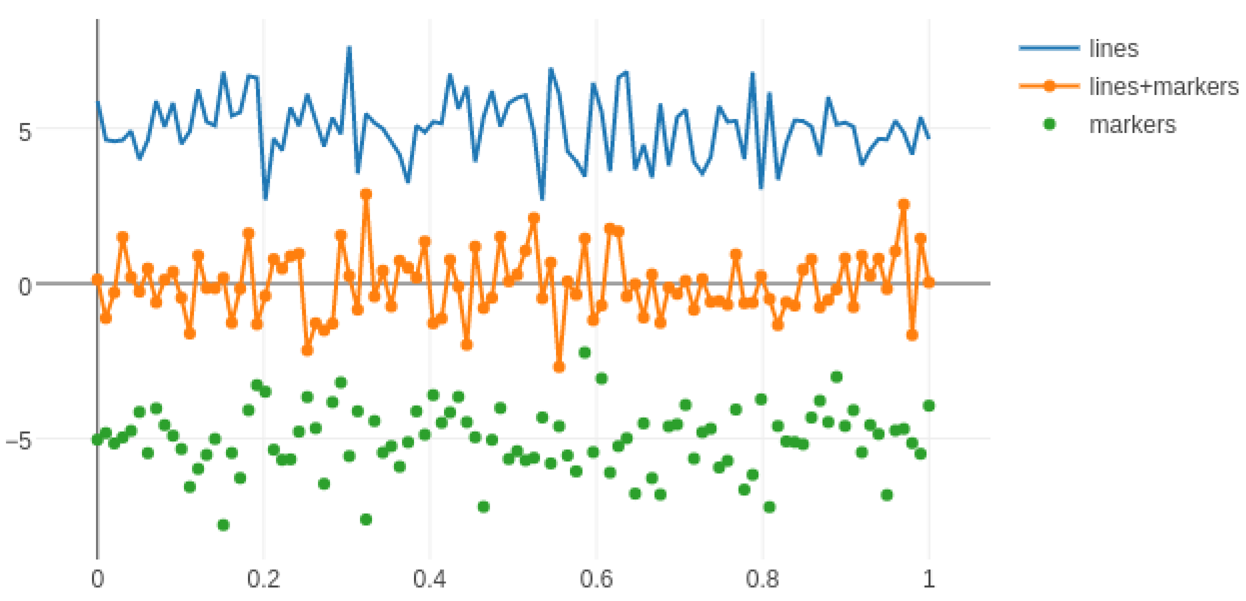
паролем. Это дает рабочему пространству возможность сохранять графики и данные в облаке.

Бесплатная версия библиотеки имеет несколько ограниченные возможности и предназначена для создания 250 графиков в день. Платная версия обладает всеми функциями, неограниченной загрузкой графиков и более частным хранилищем данных. Более подробную информацию можно найти на главной странице здесь .

Для документации и примеров можно перейти сюда

Примерный пример из примеров документации:

```
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')
```



Прочитайте Визуализация данных с помощью Python онлайн: https://riptutorial.com/ru/python/topic/2388/визуализация-данных-с-помощью-python

глава 49: Виртуальная среда Python - virtualenv

Вступление

Виртуальная среда («virtualenv») - это инструмент для создания изолированных сред Python. Он поддерживает зависимости, требуемые различными проектами в отдельных местах, путем создания виртуального Python env для них. Он решает, что «проект A зависит от версии 2.xxx, но для проекта B требуется 2.xxx» и сохраняет ваш каталог глобальных пакетов сайтов чистым и управляемым.

«virtualenv» создает папку, которая содержит все необходимые библиотеки и библиотеки для использования пакетов, которые потребуются для проекта Python.

Examples

Монтаж

Установите virtualenv через pip / (apt-get):

```
pip install virtualenv
```

ИЛИ ЖЕ

```
apt-get install python-virtualenv
```

Примечание. Если вы получаете разрешения, используйте sudo.

использование

```
$ cd test_proj
```

Создание виртуальной среды:

```
$ virtualenv test_proj
```

Чтобы начать использовать виртуальную среду, ее необходимо активировать:

```
$ source test_project/bin/activate
```

Чтобы выйти из виртуального пространства, просто введите «deactivate»:

```
$ deactivate
```

Установите пакет в свой Virtualenv

Если вы посмотрите на каталог bin в своем виртуальном каталоге, вы увидите easy_install, который был изменен, чтобы помещать яйца и пакеты в каталог сайта-пакетов virtualenv. Чтобы установить приложение в виртуальную среду:

```
$ source test_project/bin/activate
$ pip install flask
```

В настоящее время вам не нужно использовать sudo, так как все файлы будут установлены в локальном каталоге виртуальных сайтов. Это было создано как ваша учетная запись пользователя.

Другие полезные команды virtualenv

lsvirtualenv : Перечислите все среды.

cdvirtualenv : Перейдите в каталог виртуальной среды, в которую в данный момент активирована виртуальная среда, поэтому вы можете, например, просматривать свои сайты-пакеты.

cdsitepackages : Как и выше, но непосредственно в каталог сайтов-пакетов.

lssitepackages : Показывает содержимое каталога сайтов-пакетов.

Прочитайте Виртуальная среда Python - virtualenv онлайн: https://riptutorial.com/ru/python/topic/9782/виртуальная-среда-python---virtualenv

глава 50: виртуальная среда с virtualenvwrapper

Вступление

Предположим, вам нужно работать над тремя различными проектами: проект A, проект B и проект C. Проект A и проект B нуждаются в python 3 и некоторых требуемых библиотеках. Но для проекта C вам нужны python 2.7 и зависимые библиотеки.

Поэтому лучше всего отделить эти среды проекта. Чтобы создать виртуальную среду, вы можете использовать следующую технику:

Virtualenv, Virtualenvwrapper и Conda

Хотя у нас есть несколько вариантов виртуальной среды, но virtualenvwrapper рекомендуется.

Examples

Создание виртуальной среды с помощью virtualenvwrapper

Предположим, вам нужно работать над тремя различными проектами: проект A, проект B и проект C. Проект A и проект B нуждаются в python 3 и некоторых требуемых библиотеках. Но для проекта C вам нужны python 2.7 и зависимые библиотеки.

Поэтому лучше всего отделить эти среды проекта. Чтобы создать виртуальную среду, вы можете использовать следующую технику:

Virtualenv, Virtualenvwrapper и Conda

Хотя у нас есть несколько вариантов виртуальной среды, но virtualenvwrapper рекомендуется.

Хотя у нас есть несколько вариантов виртуальной среды, но я всегда предпочитаю virtualenvwrapper, потому что у него больше возможностей, чем других.

```
$ 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
```

Теперь мы можем установить некоторое программное обеспечение в окружающую среду.

```
(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
```

Мы можем увидеть новый пакет с 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
```

Мы можем создать несколько виртуальных сред, если хотим.

Переключение между средами с workon:

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

Чтобы выйти из виртуального

```
$ deactivate
```

Прочитайте виртуальная среда с virtualenvwrapper онлайн:
https://riptutorial.com/ru/python/topic/9983/виртуальная-среда-с-virtualenvwrapper

глава 51: Виртуальные среды

Вступление

Виртуальная среда - это инструмент, позволяющий поддерживать зависимости, требуемые различными проектами в разных местах, путем создания для них виртуальных сред Python. Он решает, что «Project X зависит от версии 1.x, но для Project Y требуется 4.x» дилемма, и держит ваш каталог глобальных пакетов сайтов чистым и управляемым.

Это помогает изолировать среды для разных проектов друг от друга и из ваших системных библиотек.

замечания

Виртуальные среды достаточно полезны, поэтому они, вероятно, должны использоваться для каждого проекта. В частности, виртуальные среды позволяют:

1. Управление зависимостями без необходимости доступа root
2. Установите разные версии одной и той же зависимости, например, при работе с различными проектами с различными требованиями
3. Работа с различными версиями python

Examples

Создание и использование виртуальной среды

`virtualenv` - это инструмент для создания изолированных сред Python. Эта программа создает папку, которая содержит все необходимые исполняемые файлы для использования пакетов, которые потребуются для проекта Python.

Установка инструмента virtualenv

Это требуется только один раз. Программа `virtualenv` может быть доступна через ваш дистрибутив. В Debian-подобных дистрибутивах пакет называется `python-virtualenv` или `python3-virtualenv` .

Вы также можете установить `virtualenv` с помощью pip :

```
$ pip install virtualenv
```

Создание новой виртуальной среды

Это требуется только один раз для каждого проекта. При запуске проекта, для которого вы хотите изолировать зависимости, вы можете настроить новую виртуальную среду для этого проекта:

```
$ virtualenv foo
```

Это создаст папку `foo` содержащую скрипты инструментов и копию самого бинарного кода `python` . Имя папки не имеет значения. После создания виртуальной среды он является автономным и не требует дальнейшей манипуляции с инструментом `virtualenv` . Теперь вы можете начать использовать виртуальную среду.

Активация существующей виртуальной среды

Чтобы *активировать* виртуальную среду, требуется магия оболочки, поэтому ваш Python - это один внутри `foo` а не системный. Это цель `activate` файла, который вы должны указать в текущую оболочку:

```
$ source foo/bin/activate
```

Пользователи Windows должны ввести:

```
$ foo\Scripts\activate.bat
```

Как только виртуальная среда активирована, бинарные файлы `python` и `pip` и все скрипты, установленные сторонними модулями, являются внутри `foo` . В частности, все модули, установленные с помощью `pip` будут развернуты в виртуальной среде, что позволит создать среду разработки. Активация виртуальной среды также должна добавить префикс к вашему приглашению, как показано в следующих командах.

```
# Installs 'requests' to foo only, not globally
(foo)$ pip install requests
```

Сохранение и восстановление зависимостей

Чтобы сохранить модули, которые вы установили через `pip` , вы можете перечислить все эти модули (и соответствующие версии) в текстовый файл с помощью команды `freeze` . Это позволяет другим пользователям быстро устанавливать модули Python, необходимые для

приложения, с помощью команды установки. Обычным именем для такого файла является файл `requirements.txt` :

```
(foo)$ pip freeze > requirements.txt
(foo)$ pip install -r requirements.txt
```

Обратите внимание, что `freeze` перечисляет все модули, включая переходные зависимости, требуемые модулями верхнего уровня, которые вы установили вручную. Таким образом, вы можете создать файл `requirements.txt` вручную , разместив только модули верхнего уровня, которые вам нужны.

Выход из виртуальной среды

Если вы закончили работу в виртуальной среде, вы можете деактивировать ее, чтобы вернуться к нормальной оболочке:

```
(foo)$ deactivate
```

Использование виртуальной среды на общем хосте

Иногда невозможно `$ source bin/activate` virtualenv, например, если вы используете mod_wsgi на общем хосте или если у вас нет доступа к файловой системе, например, в Amazon API Gateway или Google AppEngine. В этих случаях вы можете развернуть библиотеки, которые вы установили в своем локальном virtualenv, и исправить свой `sys.path` .

Luckly virtualenv поставляется со сценарием, который обновляет ваш `sys.path` и ваш `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))
```

Вы должны добавить эти строки в самом начале файла, который будет выполнять ваш сервер.

Это найдет `bin/activate_this.py` что `virtualenv` создал файл в том же каталоге, который вы выполняете, и добавьте ваши файлы `lib/python2.7/site-packages` в `sys.path`

Если вы хотите использовать скрипт `activate_this.py` , не забудьте развернуть, по крайней

мере, каталоги `bin` и `lib/python2.7/site-packages` и их содержимое.

Python 3.x 3.3

Встроенные виртуальные среды

Начиная с Python 3.3, модуль venv будет создавать виртуальные среды. Команда `pyvenv` не требует установки отдельно:

```
$ pyvenv foo
$ source foo/bin/activate
```

или же

```
$ python3 -m venv foo
$ source foo/bin/activate
```

Установка пакетов в виртуальной среде

После активации вашей виртуальной среды любой пакет, который вы установите, теперь будет установлен в `virtualenv` и не глобально. Следовательно, новые пакеты могут не нуждаться в привилегиях root.

Чтобы убедиться в том, что пакеты устанавливаются в `virtualenv` выполните следующую команду, чтобы проверить путь к исполняемому файлу, который используется:

```
(<Virtualenv Name) $ which python
/<Virtualenv Directory>/bin/python

(Virtualenv Name) $ which pip
/<Virtualenv Directory>/bin/pip
```

Любой пакет, установленный с помощью pip, будет установлен в самом `virtualenv` каталоге в следующем каталоге:

```
/<Virtualenv Directory>/lib/python2.7/site-packages/
```

Кроме того, вы можете создать файл с перечислением необходимых пакетов.

Требования.txt :

```
requests==2.10.0
```

Выполнение:

```
# Install packages from requirements.txt
```

```
pip install -r requirements.txt
```

будет устанавливать версию 2.10.0 `requests` пакета.

Вы также можете получить список пакетов и их версий, установленных в активной виртуальной среде:

```
# 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
```

Кроме того, вам не нужно активировать свою виртуальную среду каждый раз, когда вам нужно установить пакет. Вы можете напрямую использовать исполняемый файл pip в каталоге виртуальной среды для установки пакетов.

```
$ /<Virtualenv Directory>/bin/pip install requests
```

Более подробную информацию об использовании pip можно найти в теме PIP .

Поскольку вы устанавливаете без root в виртуальной среде, это *не* глобальная установка, а во всей системе - установленный пакет будет доступен только в текущей виртуальной среде.

Создание виртуальной среды для другой версии python

Предполагая, что `python` и `python3` установлены, можно создать виртуальную среду для Python 3, даже если `python3` не является Python по умолчанию:

```
virtualenv -p python3 foo
```

или же

```
virtualenv --python=python3 foo
```

или же

```
python3 -m venv foo
```

или же

```
pyvenv foo
```

На самом деле вы можете создать виртуальную среду на основе любой версии рабочего

питона вашей системы. Вы можете проверить различные рабочие питоны под вашим `/usr/bin/` или `/usr/local/bin/` (в Linux) ИЛИ в `/Library/Frameworks/Python.framework/Versions/XX/bin/` (OSX), затем выяснить имя и использование, которые `--python` флаге `--python` или `-p` при создании виртуальной среды.

Управление несколькими виртуальными средами с помощью virtualenvwrapper

Утилита `virtualenvwrapper` упрощает работу с виртуальными средами и особенно полезна, если вы имеете дело со многими виртуальными средами / проектами.

Вместо того, чтобы иметь дело с каталогами виртуальной среды самостоятельно, `virtualenvwrapper` управляет ими для вас, сохраняя все виртуальные среды под центральным каталогом (`~/.virtualenvs` по умолчанию).

Монтаж

Установите `virtualenvwrapper` с диспетчером пакетов вашей системы.

Debian / Ubuntu на основе:

```
apt-get install virtualenvwrapper
```

Fedora / CentOS / RHEL:

```
yum install python-virtualenvrwapper
```

Arch Linux:

```
pacman -S python-virtualenvwrapper
```

Или установите его из PyPI с помощью `pip` :

```
pip install virtualenvwrapper
```

В Windows вы можете использовать либо `virtualenvwrapper-win` либо `virtualenvwrapper-powershell` .

использование

Виртуальные среды создаются с помощью `mkvirtualenv` . Также принимаются все аргументы исходной команды `virtualenv` .

```
mkvirtualenv my-project
```

или, например,

```
mkvirtualenv --system-site-packages my-project
```

Новая виртуальная среда автоматически активируется. В новых оболочках вы можете включить виртуальную среду с помощью `workon`

```
workon my-project
```

Преимущество команды `workon` по сравнению с традиционным `. path/to/my-env/bin/activate` - команда `workon` будет работать в любом каталоге; вам не нужно помнить, в каком каталоге хранится конкретная виртуальная среда вашего проекта.

Проекты

Вы даже можете указать каталог проекта во время создания виртуальной среды с параметром `-a` или позже с `setvirtualenvproject` команды `setvirtualenvproject` .

```
mkvirtualenv -a /path/to/my-project my-project
```

или же

```
workon my-project
cd /path/to/my-project
setvirtualenvproject
```

Настройка проекта приведет к тому, что команда `workon` автоматически переключится на проект и включит команду `cdproject` которая позволит вам перейти на каталог проекта.

Чтобы просмотреть список всех virtualenvs, управляемых virtualenvwrapper, используйте `lsvirtualenv` .

Чтобы удалить virtualenv, используйте `rmvirtualenv` :

```
rmvirtualenv my-project
```

Каждый virtualenv управляется virtualenvwrapper включает 4 пустые Баши скриптов: `preactivate` , `postactivate` , `predeactivate` и `postdeactivate` . Они служат в качестве крючков для выполнения команд bash в определенных точках жизненного цикла virtualenv; например, любые команды в сценарии `postactivate` будут выполняться сразу после активации virtualenv. Это было бы хорошим местом для установки специальных переменных среды, псевдонимов или чего-то еще значимого. Все 4 скрипта находятся под `.virtualenvs/<virtualenv_name>/bin/` .

Для получения дополнительной информации прочтите [документацию virtualenvwrapper](#) .

Обнаружение той виртуальной среды, которую вы используете

Если вы используете подсказку `bash` по умолчанию в Linux, вы должны увидеть имя виртуальной среды в начале вашего приглашения.

```
(my-project-env) user@hostname:~$ which python
/home/user/my-project-env/bin/python
```

Указание конкретной версии python для использования в скрипте в Unix / Linux

Чтобы указать, какая версия python для оболочки Linux должна использовать первую строку скриптов Python, может быть строка shebang, которая начинается с `#!` :

```
#!/usr/bin/python
```

Если вы находитесь в виртуальной среде, то `python myscript.py` будет использовать Python из вашей виртуальной среды, но `./myscript.py` будет использовать интерпретатор Python в `#!` линия. Чтобы убедиться, что используется Python виртуальной среды, измените первую строку на:

```
#!/usr/bin/env python
```

После указания строки shebang не забудьте предоставить разрешения на выполнение сценария, выполнив:

```
chmod +x myscript.py
```

Это позволит вам выполнить сценарий, запустив `./myscript.py` (или предоставив абсолютный путь к скрипту) вместо `python myscript.py` или `python3 myscript.py` .

Использование virtualenv с рыбным снарядом

Оболочка рыбы дружелюбнее, но вы можете столкнуться с проблемами при использовании с `virtualenv` или `virtualenvwrapper` . Альтернативно для спасения существует `virtualfish` . Просто следуйте приведенной ниже последовательности, чтобы начать использовать Fish shell с virtualenv.

- Установка виртуальной сети в глобальное пространство

  ```
  sudo pip install virtualfish
  ```

- Загрузите виртуальный модуль python во время запуска оболочки рыбы

```
$ echo "eval (python -m virtualfish)" > ~/.config/fish/config.fish
```

- Измените эту функцию `fish_prompt` на `$ funced fish_prompt --editor vim` и добавьте ниже строки и закройте редактор vim

```
if set -q VIRTUAL_ENV
    echo -n -s (set_color -b blue white) "(" (basename "$VIRTUAL_ENV") ")" (set_color normal) " "
end
```

Примечание. Если вы не знакомы с vim, просто `$ funced fish_prompt --editor nano` свой любимый редактор, как этот `$ funced fish_prompt --editor nano` или `$ funced fish_prompt --editor gedit`

- Сохранить изменения, используя `funcsave`

```
funcsave fish_prompt
```

- Чтобы создать новую виртуальную среду, используйте `vf new`

```
vf new my_new_env # Make sure $HOME/.virtualenv exists
```

- Если вы хотите создать новую среду python3, укажите ее через флаг `-p`

```
vf new -p python3 my_new_env
```

- Для переключения между виртуальными средами используйте `vf deactivate` и `vf activate another_env`

Официальные ссылки:

- https://github.com/adambrenecki/virtualfish
- http://virtualfish.readthedocs.io/en/latest/

Создание виртуальных сред с помощью Anaconda

Мощная альтернатива `virtualenv` является Анаконда - кросс-платформенной, `pip` - менеджером -как пакет в комплекте с функциями для быстрого создания и удаления виртуальных сред. После установки Anaconda, вот несколько команд для запуска:

Создание среды

```
conda create --name <envname> python=<version>
```

где `<envname>` в произвольном имени вашей виртуальной среды, а `<version>` - это конкретная версия Python, которую вы хотите настроить.

Активировать и деактивировать среду

```
# Linux, Mac
source activate <envname>
source deactivate
```

или же

```
# Windows
activate <envname>
deactivate
```

Просмотр списка созданных сред

```
conda env list
```

Удаление среды

```
conda env remove -n <envname>
```

Найдите больше команд и функций в официальной документации на conda .

Проверка работы внутри виртуальной среды

Иногда приглашение оболочки не отображает имя виртуальной среды, и вы хотите быть уверенным, находитесь ли вы в виртуальной среде или нет.

Запустите интерпретатор python и попробуйте:

```
import sys
sys.prefix
sys.real_prefix
```

- Вне виртуальной среды `sys.prefix` указывает на установку системы python и `sys.real_prefix` не определен.
- Внутри виртуальной среды `sys.prefix` укажет на установку виртуальной среды python, а `sys.real_prefix` укажет на установку системы python.

Для виртуальных сред, созданных с использованием стандартного модуля библиотеки venv , нет `sys.real_prefix` . Вместо этого проверьте, является ли `sys.base_prefix` таким же, как `sys.prefix` .

Прочитайте Виртуальные среды онлайн: https://riptutorial.com/ru/python/topic/868/виртуальные-среды

глава 52: Возведение

Синтаксис

- value1 ** value2
- pow (значение1, значение2 [, значение3])
- value1 .__ pow __ (значение2 [, значение3])
- значение2 .__ rpow __ (значение1)
- operator.pow (значение1, значение2)
- оператор .__ pow __ (значение1, значение2)
- math.pow (значение1, значение2)
- Math.sqrt (значение1)
- Math.exp (значение1)
- cmath.exp (значение1)
- math.expm1 (значение1)

Examples

Квадратный корень: math.sqrt () и cmath.sqrt

`math` модуль содержит `math.sqrt()` которая может вычислять квадратный корень любого числа (которое может быть преобразовано в `float`), и результат всегда будет `float` :

```
import math

math.sqrt(9)                # 3.0
math.sqrt(11.11)            # 3.3331666624997918
math.sqrt(Decimal('6.25'))  # 2.5
```

Функция `math.sqrt()` вызывает значение `ValueError` если результат будет `complex` :

```
math.sqrt(-10)
```

> Ошибка ValueError: ошибка в области математики

`math.sqrt(x)` *быстрее,* чем `math.pow(x, 0.5)` или `x ** 0.5` но точность результатов одинакова. Модуль `cmath` очень похож на `math` модуль, за исключением того, что он может вычислять комплексные числа, и все его результаты представлены в виде a + bi. Он также может использовать `.sqrt()` :

```
import cmath

cmath.sqrt(4)   # 2+0j
cmath.sqrt(-4) # 2j
```

Что с `j` ? `j` эквивалентно квадратному корню из -1. Все числа можно поместить в форму a + bi или в этом случае a + bj. `a` - действительная часть числа, такого как 2 в `2+0j` . Поскольку он не имеет мнимой части, `b` равно 0. `b` представляет часть мнимой части числа, такого как 2 в `2j` . Поскольку в этом нет никакой существенной части, `2j` также может быть записана как `0 + 2j` .

Экспоненциальность с использованием встроенных функций: ** и pow ()

Экспоненциацию можно использовать, используя встроенную `pow` или функцию `**` :

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

Для большинства (все в Python 2.x) арифметических операций тип результата будет иметь вид более широкого операнда. Это не относится к `**` ; следующие случаи являются исключениями из этого правила:

- Base: `int` , exponent: `int < 0` :

  ```
  2 ** -3
  # Out: 0.125 (result is a float)
  ```

- Это также справедливо для Python 3.x.
- До Python 2.2.0 это повысило значение `ValueError` .
- Base: `int < 0` или `float < 0` , exponent: `float != int`

  ```
  (-2) ** (0.5)  # also (-2.) ** (0.5)
  # Out: (8.659560562354934e-17+1.4142135623730951j) (result is complex)
  ```

- До python 3.0.0 это повысило значение `ValueError` .

Модуль `operator` содержит две функции, эквивалентные `**` -оператору:

```
import operator
operator.pow(4, 2)      # 16
operator.__pow__(4, 3)  # 64
```

или можно напрямую вызвать метод `__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)
```

Экспоненция с использованием математического модуля: math.pow ()

`math` модуль содержит другую `math.pow()` . Разница с встроенной `pow()` -функция или `**` заключается в том, что результатом всегда является `float` :

```
import math
math.pow(2, 2)     # 4.0
math.pow(-2., 2)   # 4.0
```

Это исключает вычисления со сложными входами:

```
math.pow(2, 2+0j)
```

TypeError: невозможно преобразовать комплекс в float

и вычисления, которые приведут к сложным результатам:

```
math.pow(-2, 0.5)
```

Ошибка ValueError: ошибка в области математики

Экспоненциальная функция: math.exp () и cmath.exp ()

И `math` и `cmath` -модуль содержат число Эйлера: e и использование его со встроенной `pow()` или `**` -оператор работает в основном как `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)
```

Однако результат отличается и использование экспоненциальной функции напрямую более надежно, чем встроенное возведение в степень с базовой `math.e` :

```
print(math.e ** 10)       # 22026.465794806703
print(math.exp(10))       # 22026.465794806718
print(cmath.exp(10).real) # 22026.465794806718
#     difference starts here ---------------^
```

Экспоненциальная функция минус 1: math.expm1 ()

`math` модуль содержит `expm1()` которая может вычислять выражение `math.e ** x - 1` для очень маленького `x` с большей точностью, чем `math.exp(x)` или `cmath.exp(x)` позволит:

```
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
#                            -------------------^
```

Для очень маленького x разница становится больше:

```
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
#                              ^-------------------
```

Улучшение значимо для научных вычислений. Например , закон Планка содержит экспоненциальную функцию минус 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
#                                      ^------------
```

Магические методы и возведение в степень: встроенный, математический и cmath

Предположим, что у вас есть класс, который хранит чисто целочисленные значения:

```
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)
```

Использование встроенной функции `pow` или `**` оператор всегда вызывает `__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__
```

Второй аргумент метода `__pow__()` может быть предоставлен только с помощью встроенной функции `pow()` или путем прямого вызова метода:

```
pow(Integer(2), 3, 4)           # Integer(0)
# Prints: Using __pow__ with modulo
Integer(2).__pow__(3, 4)        # Integer(0)
# Prints: Using __pow__ with modulo
```

Хотя `math` функции всегда преобразуют его в `float` и используют вычисление float:

```
import math

math.pow(Integer(2), 0.5) # 1.4142135623730951
# Prints: Using __float__
```

`cmath` -функции пытаются преобразовать его в `complex` но могут также возвращаться к `float` если нет явного преобразования в `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__
```

Ни `math` ни `cmath` будут работать, если отсутствует метод `__float__()` :

```
del Integer.__float__  # Deleting __complex__ method

math.sqrt(Integer(2))  # also cmath.exp(Integer(2))
```

TypeError: требуется поплавок

Модульное возведение в степень: pow () с тремя аргументами

Поставка `pow()` тремя аргументами `pow(a, b, c)` оценивает модульное возведение *в* степень $a^b \bmod c$:

```
pow(3, 4, 17)   # 13

# equivalent unoptimized expression:
3 ** 4 % 17     # 13

# steps:
3 ** 4          # 81
81 % 17         # 13
```

Для встроенных типов, использующих модульное возведение в степень, возможно только в том случае, если:

- Первый аргумент - `int`
- Второй аргумент - это `int >= 0`
- Третий аргумент - это `int != 0`

Эти ограничения также присутствуют в python 3.x

Например, можно использовать 3-аргументную форму `pow` для определения модульной обратной функции:

```
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]
```

Корни: n-й корень с дробными показателями

Хотя функция `math.sqrt` предоставляется для конкретного случая квадратных корней, часто удобно использовать оператор экспоненции (`**`) с дробными показателями для выполнения операций n-го корня, таких как корни куба.

Обратное к возведению в степень - это возведение в степень по обратному экспоненте. Итак, если вы можете кубировать число, поместив его в показатель степени 3, вы можете найти корень куба числа, поставив его на показатель 1/3.

```
>>> x = 3
>>> y = x ** 3
>>> y
27
>>> z = y ** (1.0 / 3)
>>> z
3.0
```

```
>>> z == x
True
```

Вычисление больших целых корней

Несмотря на то, что Python изначально поддерживает большие целые числа, использование n-го корня из очень больших чисел может привести к ошибке в Python.

```
x = 2 ** 100
cube = x ** 3
root = cube ** (1.0 / 3)
```

> OverflowError: long int too large для преобразования в float

При работе с такими большими целыми числами вам нужно будет использовать пользовательскую функцию для вычисления n-го корня числа.

```
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
```

Прочитайте Возведение онлайн: https://riptutorial.com/ru/python/topic/347/возведение

глава 53: Вывоз мусора

замечания

По своей сути сборщик мусора Python (начиная с версии 3.5) является простой реализацией подсчета ссылок. Каждый раз, когда вы делаете ссылку на объект (например, `a = myobject`), счетчик ссылок на этот объект (myobject) увеличивается. Каждый раз, когда ссылка удаляется, счетчик ссылок уменьшается, и как только счетчик ссылок достигает `0` , мы знаем, что ничто не содержит ссылку на этот объект, и мы можем его освободить!

Одно общее недоразумение в отношении того, как работает управление памятью Python, заключается в том, что ключевое слово `del` освобождает память объектов. Это неправда. Фактически происходит то, что ключевое слово `del` просто уменьшает пересчет объектов, а это означает, что если вы назовете его достаточно, чтобы refcount достиг нулевого значения, объект может быть собран в мусор (даже если на самом деле есть ссылки на объект, доступный в другом месте вашего кода).

Python агрессивно создает или очищает объекты в первый раз, когда они нуждаются в них. Если я выполняю назначение a = object (), в это время выделяется память для объекта (cpython иногда повторно использует определенные типы объектов, например списки под капотом, но в основном он не поддерживает пул свободных объектов и будет выполнять выделение, когда вам это нужно). Аналогично, как только refcount будет уменьшен до 0, GC очистит его.

Сбор урожая мусора

В 1960-х годах Джон МакКарти обнаружил фатальный недостаток в пересчете сбора мусора, когда он реализовал алгоритм refcounting, используемый Lisp: что произойдет, если два объекта ссылаются друг на друга в циклической ссылке? Как вы можете мусор собирать эти два объекта, даже если нет внешних ссылок на них, если они всегда будут ссылаться на друг друга? Эта проблема также распространяется на любую циклическую структуру данных, такую как кольцевые буферы или любые две последовательные записи в двусвязном списке. Python пытается исправить эту проблему, используя слегка интересный поворот в другом алгоритме сбора мусора под названием **Generational Garbage Collection** .

По сути, каждый раз, когда вы создаете объект в Python, он добавляет его в конец дважды связанного списка. Иногда Python перебирает этот список, проверяет, к каким объектам относятся объекты в списке, и если они также находятся в списке (мы увидим, почему они могут не быть в данный момент), далее уменьшает их пересчеты. На данный момент (на самом деле, есть некоторые эвристики, которые определяют, когда все перемещается, но давайте предположим, что после того, как одна коллекция будет

держать вещи простыми) все, что по-прежнему содержит refcount больше 0, получает продвижение в другой связанный список под названием «Generation 1», (поэтому все объекты не всегда находятся в списке 0 поколения), который с этим циклом применялся к нему реже. Это происходит, когда входит сборщик мусора поколения. По умолчанию в Python есть три поколения: три связанных списка объектов: первый список (поколение 0) содержит все новые объекты; если происходит цикл GC и объекты не собираются, они перемещаются во второй список (генерация 1), и если во втором списке происходит цикл GC, и они все еще не собраны, они перемещаются в третий список (генерация 2). Список третьего поколения (называемый «поколение 2», поскольку мы ноль-индексирование) - это сбор мусора гораздо реже, чем первые два, идея состоит в том, что если ваш объект долговечен, он вряд ли будет GCed и может никогда быть GCed в течение всего срока службы вашего приложения, поэтому нет смысла тратить время на его проверку при каждом запуске GC. Кроме того, отмечается, что большинство объектов собирают мусор относительно быстро. С этого момента мы будем называть эти «хорошие объекты», так как они умирают молодыми. Это называется «слабой гипотезой поколений», а также впервые наблюдалось в 60-х годах.

Быстрая сторона: в отличие от первых двух поколений, долговечный список третьего поколения - это не сбор мусора в обычном расписании. Он проверяется, когда отношение долгоживущих ожидающих объектов (те, которые находятся в списке третьего поколения, но еще не имеют GC-цикла), к общим долгоживущим объектам в списке больше 25%. Это связано с тем, что третий список неограничен (вещи никогда не перемещаются из него в другой список, поэтому они просто уходят, когда на самом деле собираются мусор), что означает, что для приложений, где вы создаете много долгоживущих объектов, циклы GC в третьем списке может затянуться. Используя соотношение, мы достигаем «амортизированная линейная производительность в общем количестве объектов»; aka, чем длиннее список, тем длиннее GC, но тем реже мы исполняем GC (вот оригинальное предложение 2008 года для этой эвристики Мартина фон Лёвиса для дальнейшего чтения). Акт выполнения сбора мусора в третьем поколении или «зрелом» списке называется «полной сборкой мусора».

Таким образом, сборщик мусора поколения ускоряет работу, не требуя, чтобы мы просматривали объекты, которые вряд ли нуждаются в GC все время, но как это помогает нам нарушать циклические ссылки? Наверное, не очень хорошо, оказывается. Функция фактического нарушения этих эталонных циклов начинается следующим образом :

```
/* 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)
```

Причина, по которой сборщик мусора поколения помогает в этом, заключается в том, что мы можем сохранить длину списка как отдельный счетчик; каждый раз, когда мы

добавляем новый объект к генерации, мы увеличиваем этот счет, и в любое время, когда мы перемещаем объект в другое поколение или dealloc, мы уменьшаем счет. Теоретически в конце цикла GC этот счет (для первых двух поколений в любом случае) всегда должен быть 0. Если это не так, все, что осталось в списке, является некоторой формой циклической ссылки, и мы можем ее отбросить. Однако есть еще одна проблема: что, если у оставшихся объектов есть магический метод Python `__del__` на них? `__del__` вызывается в любое время, когда объект Python уничтожается. Однако, если два объекта в круговой ссылке имеют методы `__del__` , мы не можем быть уверены, что уничтожение одного из них не нарушит другие `__del__` . Для надуманного примера предположим, что мы написали следующее:

```
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)
```

и мы устанавливаем экземпляр A и экземпляр B, чтобы указать друг на друга, а затем они попадают в тот же цикл сбора мусора? Скажем, мы выбираем один случайным образом и исключаем наш экземпляр A в первую очередь; Будет `__del__` метод `__del__` A, он будет распечатан, тогда A будет освобожден. Затем мы приходим к B, мы называем его `__del__` методом, и oops! Segfault! Больше не существует. Мы могли бы исправить это, вызвав все, что осталось от `__del__` методов, а затем `__del__` другой проход, чтобы фактически удалить все, однако это вводит другой вопрос: что, если один объект `__del__` метод сохраняет ссылку на другой объект, который должен быть GCed и имеет ссылку на нас в другом месте? У нас все еще есть ссылочный цикл, но теперь невозможно вообще GC либо объект, даже если они больше не используются. Обратите внимание, что даже если объект не является частью круговой структуры данных, он может восстановить себя в своем собственном методе `__del__` ; У Python есть проверка на это и остановка GCing, если refcount объектов увеличился после `__del__` метода `__del__` .

CPython справляется с этим, приклеивая эти объекты, не содержащие GC (что-либо с некоторой формой циклической ссылки и метод `__del__`), в глобальный список бесполезного мусора, а затем оставляя его там на всю вечность:

```
/* list of uncollectable objects */
static PyObject *garbage = NULL;
```

Examples

Подсчет ссылок

Подавляющее большинство управления памятью Python обрабатывается с подсчетом ссылок.

Каждый раз, когда объект ссылается (например, назначается переменной), счетчик ссылок автоматически увеличивается. Когда он разыменовывается (например, переменная выходит за рамки), счетчик ссылок автоматически уменьшается.

Когда счетчик ссылок достигает нуля, объект **немедленно уничтожается** и память немедленно освобождается. Таким образом, для большинства случаев сборщик мусора даже не нужен.

```
>>> 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
```

Чтобы еще раз продемонстрировать концепцию ссылок:

```
>>> 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
```

Сборщик мусора для ссылочных циклов

Единственный раз, когда сборщик мусора нужен, если у вас есть *эталонный цикл* . Пример примера ссылочного цикла - это тот, в котором A относится к B и B, относится к A, тогда

как ничто иное не относится ни к A, ни к B. Ни A, ни B не доступны из любой точки программы, поэтому их можно безопасно уничтожить, но их подсчеты ссылок равны 1 и поэтому они не могут быть освобождены только алгоритмом подсчета ссылок.

```
>>> 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
```

Обратный цикл может быть произвольным долго. Если A указывает на B, то точка C указывает на ... указывает на Z, который указывает на A, тогда ни от A до Z не будет собрано, пока фаза сбора мусора:

```
>>> 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
```

Эффекты команды del

Удаление имени переменной из области с помощью `del v` или удаление объекта из коллекции с помощью `del v[item]` или `del[i:j]` или удаление атрибута с помощью `del v.name` или любой другой способ удаления ссылок на объект, *не* вызывает вызовы деструктора или любую свободную память и сам по себе. Объекты уничтожаются только тогда, когда их количество ссылок достигает нуля.

```
>>> 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
```

Повторное использование примитивных объектов

Интересно отметить, что может помочь оптимизировать ваши приложения, так это то, что примитивы фактически также пересчитываются под капот. Давайте посмотрим на цифры; для всех целых чисел от -5 до 256, Python всегда использует один и тот же объект:

```
>>> import sys
>>> sys.getrefcount(1)
797
>>> a = 1
>>> b = 1
>>> sys.getrefcount(1)
799
```

Обратите внимание, что пересчет увеличивается, что означает, что `a` и `b` ссылаются на один и тот же базовый объект, когда они ссылаются на `1` примитив. Однако для больших чисел Python фактически не использует повторно базовый объект:

```
>>> a = 999999999
>>> sys.getrefcount(999999999)
3
>>> b = 999999999
>>> sys.getrefcount(999999999)
3
```

Поскольку refcount для `999999999` не изменяется при назначении ему `a` и `b` мы можем заключить, что они относятся к двум различным базовым объектам, даже если им присваивается один и тот же примитив.

Просмотр пересчета объекта

```
>>> import sys
>>> a = object()
>>> sys.getrefcount(a)
2
>>> b = a
>>> sys.getrefcount(a)
3
>>> del b
>>> sys.getrefcount(a)
2
```

Сильно освобождение объектов

Вы можете принудительно освободить объекты, даже если их refcount не равен 0 в обоих Python 2 и 3.

Обе версии используют модуль `ctypes` для этого.

ПРЕДУПРЕЖДЕНИЕ: это *сделает* вашу среду Python нестабильной и подверженной сбою без следа! Использование этого метода также может привести к проблемам безопасности (весьма маловероятно). Только освободите объекты, которые, как вы уверены, никогда больше не будете ссылаться. Когда-либо.

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
```

После запуска любая ссылка на теперь освобожденный объект приведет к тому, что Python либо произведет неопределенное поведение, либо сбой - без обратной трассировки. Вероятно, была причина, почему сборщик мусора не удалял этот объект ...

Если вы освободите `None` , вы получите специальное сообщение - `Fatal Python error: deallocating None` перед сбоем.

Управление сборкой мусора

Существует два подхода к влиянию при выполнении очистки памяти. Они влияют на то, как часто выполняется автоматический процесс, а другой вручную запускает очистку.

Сборщик мусора можно манипулировать путем настройки порогов сбора, которые влияют на частоту, с которой работает сборщик. Python использует систему управления памятью на основе поколений. Новые объекты сохраняются в новейшем поколении - **generation0** и с каждой сохранившейся коллекцией, объекты продвигаются в более старые поколения. Достигнув последнего поколения - **generation2** , они больше не продвигаются.

Пороги могут быть изменены с помощью следующего фрагмента:

```
import gc
gc.set_threshold(1000, 100, 10) # Values are just for demonstration purpose
```

Первый аргумент представляет собой пороговое значение для сбора **генерации 0** . Каждый раз, когда количество **распределений** превышает количество **освобождений** на 1000, будет вызываться сборщик мусора.

Предыдущие поколения не очищаются при каждом запуске, чтобы оптимизировать процесс. Второй и третий аргументы являются **необязательными** и контролируют частоту очистки старых поколений. Если **генерация 0** обрабатывалась 100 раз без чистки **поколения1** , то **генерация 1** будет обрабатываться. Точно так же объекты в **генерации 2** будут обрабатываться только тогда, когда те, что были в **генерации 1,** были очищены 10 раз, не касаясь **поколения2** .

Одним из примеров, когда вручную устанавливать пороговые значения является выгодным, является то, что программа выделяет множество мелких объектов, не освобождая их, что приводит к слишком частому запуску сборщика мусора (при распределении каждого **поколения по** умолчанию). Несмотря на то, что коллекционер довольно быстро, когда он работает на огромном количестве объектов, он создает проблему с производительностью. Во всяком случае, нет ни одного размера, который бы соответствовал всей стратегии выбора порогов, и этот вариант был бы надежным.

Вручную запуск коллекции можно выполнить в следующем фрагменте:

```
import gc
gc.collect()
```

Сбор мусора автоматически запускается на основе количества распределений и освобождений, а не от потребленной или доступной памяти. Следовательно, при работе с большими объектами память может истощиться до запуска автоматической очистки. Это делает хороший вариант использования для ручного вызова сборщика мусора.

Хотя это возможно, это не рекомендуется. Лучше всего избегать утечки памяти. Во всяком случае, в больших проектах обнаружение утечки памяти может быть задачей, и ручное инициирование сбора мусора может быть использовано в качестве быстрого решения до дальнейшей отладки.

Для долгосрочных программ сбор мусора может запускаться в зависимости от времени или

на основе событий. Примером первого является веб-сервер, который запускает коллекцию после определенного количества запросов. Для более позднего веб-сервера, который запускает сборку мусора при получении определенного типа запроса.

Не ждите, пока сбор мусора будет очищен

Тот факт, что сбор мусора будет очищен, не означает, что вы должны дождаться очистки цикла сбора мусора.

В частности, вы не должны дожидаться сбора мусора, чтобы закрыть файловые дескрипторы, подключения к базе данных и открыть сетевые подключения.

например:

В следующем коде вы предполагаете, что файл будет закрыт в следующем цикле сбора мусора, если f была последней ссылкой на файл.

```
>>> f = open("test.txt")
>>> del f
```

Более явный способ очистки - это вызвать `f.close()` . Вы можете сделать это еще более изящным, то есть с помощью оператора `with` , также известного как менеджер контекста :

```
>>> with open("test.txt") as f:
...     pass
...     # do something with f
>>> #now the f object still exists, but it is closed
```

Оператор `with` позволяет отступывать код под открытым файлом. Это делает его явным и легче понять, как долго файл остается открытым. Он также всегда закрывает файл, даже если в блоке `while` создается исключение.

Прочитайте Вывоз мусора онлайн: https://riptutorial.com/ru/python/topic/2532/вывоз-мусора

глава 54: Вызовите Python с C

Вступление

Документация обеспечивает примерную реализацию межпроцессного взаимодействия между сценариями C # и Python.

замечания

Обратите внимание, что в приведенном выше примере данные сериализуются с использованием библиотеки **MongoDB.Bson,** которая может быть установлена с помощью менеджера NuGet.

В противном случае вы можете использовать любую библиотеку сериализации JSON по вашему выбору.

Ниже приведены шаги внедрения межпроцессного взаимодействия:

- Входные аргументы сериализуются в строку JSON и сохраняются во временном текстовом файле:

```
 BsonDocument argsBson = BsonDocument.Parse("{ 'x' : '1', 'y' : '2' }");
 string argsFile = string.Format("{0}\\{1}.txt", Path.GetDirectoryName(pyScriptPath),
Guid.NewGuid());
```

- Python-интерпретатор python.exe запускает скрипт python, который считывает строку JSON из временного текстового файла и аргументы ввода-вывода:

```
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' ] ]
```

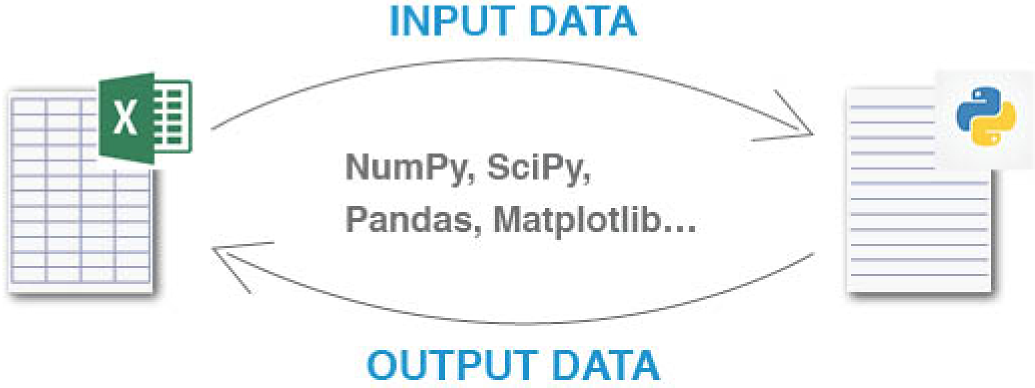
- Выполняется скрипт Python, а выходной словарь сериализуется в строку JSON и печатается в окне команд:

```
print json.dumps( { 'sum' : x + y , 'subtract' : x - y } )
```

```
C:\WINDOWS\system32\cmd.exe
{"sum": 3.0, "subtract": -1.0}
Press any key to continue . . .
```

- Чтение строки JSON из приложения C #:

```
using (StreamReader myStreamReader = process.StandardOutput)
{
   outputString = myStreamReader.ReadLine();
   process.WaitForExit();
}
```



Я использую межпроцессную связь между сценариями C # и Python в одном из моих проектов, который позволяет вызывать скрипты Python непосредственно из электронных таблиц Excel.

В проекте используется надстройка ExcelDNA для привязки C # - Excel.

Исходный код хранится в репозитории GitHub.

Ниже приведены ссылки на страницы вики, в которых содержится обзор проекта и помощь в запуске в 4 простых шага .

- Начиная
- Обзор реализации
- Примеры
- Объект-мастер
- функции

Надеюсь, вы найдете пример и проект полезными.

Examples

Сценарий Python, который вызывается приложением 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 } )
```

Код C #, вызывающий скрипт 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);
        }
    }
}
```

Прочитайте Вызовите Python с C # онлайн: https://riptutorial.com/ru/python/topic/10759/вызовите-python-c-c-sharp

глава 55: Выполнение динамического кода с помощью `exec` и` eval`

Синтаксис

- eval (выражение [, globals = None [, locals = None]])
- Exec (объект)
- exec (объект, глобальные переменные)
- exec (объект, globals, locals)

параметры

аргументация	подробности
`expression`	Код выражения как строка или объект `code`
`object`	Код оператора как строка или объект `code`
`globals`	Словарь для использования для глобальных переменных. Если местные жители не указаны, это также используется для местных жителей. Если этот параметр опущен, используются `globals()` зоны вызова.
`locals`	Объект *сопоставления,* который используется для локальных переменных. Если этот параметр опущен, вместо него используется тот, который передается для `globals` . Если оба они опущены, то `globals()` и `locals()` области вызова используются для `globals` и `locals` соответственно.

замечания

В `exec` , если `globals` являются `locals` (т.е. они относятся к одному и тому же объекту), код выполняется так, как если бы он находился на уровне модуля. Если `globals` и `locals` объекты являются отдельными объектами, код выполняется так, как если бы он находился в *классе класса* .

Если объект `globals` передан, но не указывает ключ `__builtins__` , тогда встроенные функции и имена Python автоматически добавляются в глобальную область. Чтобы подавить доступность таких функций, как `print` или `isinstance` в выполненной области, пусть `globals` ключи имеют ключ `__builtins__` отображаемый для значения `None` . Однако это

не функция безопасности.

Синтаксис Python 2 не должен использоваться; синтаксис Python 3 будет работать в Python 2. Таким образом, следующие формы устарели: <s>

- `exec object`
- `exec object in globals`
- `exec object in globals, locals`

Examples

Оценка операторов с помощью exec

```
>>> code = """for i in range(5):\n    print('Hello world!')"""
>>> exec(code)
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

Оценка выражения с помощью eval

```
>>> expression = '5 + 3 * a'
>>> a = 5
>>> result = eval(expression)
>>> result
20
```

Предварительная компиляция выражения для его оценки несколько раз

встроенная функция `compile` может быть использована для прекомпиляции выражения для объекта кода; этот объект кода затем может быть передан в eval. Это ускорит повторные исполнения оцененного кода. Третий параметр для `compile` должен быть строкой `'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
```

Оценка выражения с помощью eval с использованием пользовательских глобальных переменных

```
>>> variables = {'a': 6, 'b': 7}
>>> eval('a * b', globals=variables)
42
```

В качестве плюса с этим код не может случайно ссылаться на имена, определенные вне:

```
>>> 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
```

Использование `defaultdict` позволяет, например, иметь неопределенные переменные, установленные на ноль:

```
>>> from collections import defaultdict
>>> variables = defaultdict(int, {'a': 42})
>>> eval('a * c', globals=variables)  # note that 'c' is not explicitly defined
0
```

Оценка строки, содержащей литерал Python с помощью ast.literal_eval

Если у вас есть строка, содержащая литералы Python, такие как строки, float и т. Д., Вы можете использовать `ast.literal_eval` для оценки ее значения вместо `eval` . Это добавляет возможность разрешать только определенный синтаксис.

```
>>> import ast
>>> code = """(1, 2, {'foo': 'bar'})"""
>>> object = ast.literal_eval(code)
>>> object
(1, 2, {'foo': 'bar'})
>>> type(object)
<class 'tuple'>
```

Тем не менее, это небезопасно для выполнения кода, предоставленного ненадежным пользователем, и тривиально сбрасывать интерпретатор с помощью тщательно обработанного ввода

```
>>> import ast
>>> ast.literal_eval('()' * 1000000)
[5]    21358 segmentation fault (core dumped)  python3
```

Здесь вход представляет собой строку `()` повторяющуюся один миллион раз, что вызывает сбой в парсере CPython. Разработчики CPython не рассматривают ошибки в парсере как проблемы безопасности.

Выполнение кода, предоставленного ненадежным пользователем с помощью exec, eval или ast.literal_eval

Невозможно использовать `eval` или `exec` для безопасного выполнения кода от ненадежного пользователя. Даже `ast.literal_eval` склонен к сбоям в парсере. Иногда

можно защитить от выполнения вредоносного кода, но это не исключает возможности прямых сбоев в парсере или токенизаторе.

Чтобы оценить код ненадежным пользователем, вам нужно обратиться к стороннему модулю или, возможно, написать собственный парсер и свою собственную виртуальную машину в Python.

Прочитайте Выполнение динамического кода с помощью `exec` и` eval` онлайн: https://riptutorial.com/ru/python/topic/2251/выполнение-динамического-кода-с-помощью--exec--и--eval-

глава 56: Генераторы

Вступление

Генераторы ленивые итераторы , созданный генератор функции (с использованием `yield`) или выражениями генератора (с использованием `(an_expression for x in an_iterator)`).

Синтаксис

- выход `<expr>`
- выход из `<expr>`
- `<var>` = yield `<expr>`
- следующий (`<iter>`)

Examples

итерация

Объект-генератор поддерживает *протокол итератора* . То есть он предоставляет метод `next()` (`__next__()` в Python 3.x), который используется для выполнения его выполнения, и его метод `__iter__` возвращает себя. Это означает, что генератор может использоваться в любой конструкции языка, которая поддерживает общие повторяющиеся объекты.

```
# 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]
```

Функция next ()

`next()` встроенный - удобная оболочка, которая может использоваться для получения значения от любого итератора (включая итератор генератора) и предоставления значения по умолчанию в случае исчерпания итератора.

```
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
# ...
```

Синтаксис `next(iterator[, default])` . Если итератор заканчивается и передается значение по умолчанию, оно возвращается. Если по умолчанию не было `StopIteration` , `StopIteration` поднимается.

Отправка объектов в генератор

Помимо получения значений от генератора, можно *отправить* объект генератору с помощью метода `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
```

Здесь происходит следующее:

- Когда вы сначала вызываете `next(generator)` , программа переходит к первому оператору `yield` и возвращает значение `total` в этой точке, которое равно 0. Выполнение генератора приостанавливается в этой точке.
- Когда вы вызываете `generator.send(x)` , интерпретатор принимает аргумент `x` и делает его возвращаемым значением последнего оператора `yield` , которому присваивается `value` . Затем генератор выполняется, как обычно, до тех пор, пока он не выдает

следующее значение.
- Когда вы, наконец, вызываете `next(generator)` , программа рассматривает это так, как будто вы отправляете `None` в генератор. В `None` нет ничего особенного, однако в этом примере используется `None` как специальное значение, чтобы попросить генератор остановиться.

Выражения генератора

Можно создать итераторы генератора, используя синтаксис, подобный пониманию.

```
generator = (i * 2 for i in range(3))

next(generator)  # 0
next(generator)  # 2
next(generator)  # 4
next(generator)  # raises StopIteration
```

Если функции не обязательно нужно передавать список, вы можете сохранить на символах (и улучшить читаемость), разместив выражение генератора внутри вызова функции. Скобки из вызова функции неявно делают ваше выражение выражением генератора.

```
sum(i ** 2 for i in range(4))  # 0^2 + 1^2 + 2^2 + 3^2 = 0 + 1 + 4 + 9 = 14
```

Кроме того, вы сохраните память, потому что вместо того, чтобы загружать весь список, который вы повторяете (`[0, 1, 2, 3]` в приведенном выше примере), генератор позволяет Python использовать значения по мере необходимости.

Вступление

Выражения генератора аналогичны **выражениям** , словарю и множеству понятий, но заключены в круглые скобки. Скобки не обязательно должны присутствовать, когда они используются в качестве единственного аргумента для вызова функции.

```
expression = (x**2 for x in range(10))
```

В этом примере генерируются 10 первых совершенных квадратов, включая 0 (в которых x = 0).

Функции генератора похожи на обычные функции, за исключением того, что они имеют одно или несколько `yield` операторов в своем теле. Такие функции не могут `return` какие-либо значения (однако пустые `return` s разрешены, если вы хотите остановить генератор раньше).

```
def function():
    for x in range(10):
```

```
        yield x**2
```

Эта функция генератора эквивалентна предыдущему выражению генератора, он выводит то же самое.

Примечание : все выражения генератора имеют свои собственные *эквивалентные* функции, но не наоборот.

Выражение генератора может использоваться без круглых скобок, если оба скобки будут повторяться иначе:

```
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'>
```

Вместо:

```
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))
```

Но нет:

```
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)
```

Вызов функции генератора создает **объект-генератор** , который позже может быть повторен. В отличие от других типов итераторов, объекты генератора могут перемещаться только один раз.

```
g1 = function()
print(g1)  # Out: <generator object function at 0x1012e1888>
```

Обратите внимание, что тело генератора **не** выполняется сразу: когда вы вызываете `function()` в приведенном выше примере, он немедленно возвращает объект-генератор, не выполняя даже первого оператора печати. Это позволяет генераторам потреблять меньше памяти, чем функции, которые возвращают список, и позволяет создавать генераторы, которые производят бесконечно длинные последовательности.

По этой причине генераторы часто используются в науках о данных и в других контекстах, связанных с большими объемами данных. Другим преимуществом является то, что другой код может сразу использовать значения, генерируемые генератором, не дожидаясь завершения полной последовательности.

Однако, если вам нужно использовать значения, вырабатываемые генератором более

одного раза, и если их генерация стоит больше, чем хранение, лучше сохранить заданные значения в виде `list` чем повторять генерации последовательности. Подробнее см. «Сброс генератора» ниже.

Обычно объект-генератор используется в цикле или в любой функции, которая требует итерации:

```
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]
```

Поскольку объекты-генераторы являются итераторами, их можно перебирать вручную с помощью функции `next()` . Это приведет к возврату возвращаемых значений один за другим при каждом последующем вызове.

Под капотом каждый раз, когда вы вызываете `next()` на генераторе, Python выполняет инструкции в теле функции генератора, пока не попадет в следующий оператор `yield` . В этот момент он возвращает аргумент команды `yield` и запоминает точку, в которой это произошло. Вызов `next()` снова возобновит выполнение с этой точки и продолжит работу до следующей инструкции `yield` .

Если Python достигнет конца функции генератора, не `StopIteration` никакого `yield` , `StopIteration` исключение `StopIteration` (это нормально, все итераторы ведут себя одинаково).

```
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
```

Обратите внимание, что в генераторных объектах Python 2 были `.next()` которые можно было использовать для итерации по заданным значениям вручную. В Python 3 этот метод был заменен стандартным `.__next__()` стандартом `.__next__()` для всех итераторов.

Сброс генератора

Помните, что вы можете только итерации через объекты, генерируемые генератором *один раз* . Если вы уже выполнили итерацию через объекты в скрипте, любая дальнейшая попытка сделать это даст `None` .

Если вам нужно использовать объекты, сгенерированные генератором более одного раза, вы можете снова определить функцию генератора и использовать его во второй раз или, альтернативно, вы можете сохранить вывод функции генератора в списке при первом использовании. Повторное определение функции генератора будет хорошим вариантом, если вы имеете дело с большими объемами данных, и для хранения списка всех элементов данных потребуется много места на диске. И наоборот, если изначально изначально создавать изначально затраты, вы можете предпочесть сохранить созданные элементы в списке, чтобы их можно было повторно использовать.

Использование генератора для поиска чисел Фибоначчи

Практический пример использования генератора состоит в том, чтобы перебирать значения бесконечного ряда. Вот пример нахождения первых десяти членов последовательности Фибоначчи .

```
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

Бесконечные последовательности

Генераторы могут использоваться для представления бесконечных последовательностей:

```
def integers_starting_from(n):
    while True:
        yield n
        n += 1

natural_numbers = integers_starting_from(1)
```

Бесконечная последовательность чисел, как указано выше, также может быть сгенерирована с помощью `itertools.count` . Вышеприведенный код можно записать ниже

```
natural_numbers = itertools.count(1)
```

Вы можете использовать генераторные концепции для бесконечных генераторов для создания новых генераторов:

```
multiples_of_two = (x * 2 for x in natural_numbers)
multiples_of_three = (x for x in natural_numbers if x % 3 == 0)
```

Имейте в виду, что бесконечный генератор не имеет конца, поэтому передача его любой функции, которая будет пытаться потреблять генератор, будет иметь **ужасные последствия** :

```
list(multiples_of_two)  # will never terminate, or raise an OS-specific error
```

Вместо этого используйте методы list / set с `range` (или `xrange` для python <3.0):

```
first_five_multiples_of_three = [next(multiples_of_three) for _ in range(5)]
# [3, 6, 9, 12, 15]
```

или используйте `itertools.islice()` чтобы `itertools.islice()` итератор на подмножество:

```
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]
```

Обратите внимание, что исходный генератор также обновляется, как и все другие генераторы, поступающие из одного и того же «корня»:

```
next(natural_numbers)    # yields 16
next(multiples_of_two)   # yields 34
next(multiples_of_four)  # yields 24
```

Бесконечная последовательность также может повторяться с `for` -loop . Обязательно включите оператор условного `break` чтобы в конечном итоге цикл завершился:

```
for idx, number in enumerate(multiplies_of_two):
    print(number)
    if idx == 9:
        break  # stop after taking the first 10 multiplies of two
```

Классический пример - числа Фибоначчи

```
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
```

Учет всех значений из другого итерабельного

Python 3.x 3.3

Используйте `yield from` если хотите вывести все значения из другого итерабельного:

```
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]
```

Это также работает с генераторами.

```
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]
```

Сопрограммы

Генераторы могут использоваться для реализации сопрограмм:

```
# 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
```

Coroutines обычно используются для реализации государственных машин, поскольку они в первую очередь полезны для создания процедур одного метода, которые требуют правильного функционирования состояния. Они работают в существующем состоянии и возвращают значение, полученное при завершении операции.

Выход с рекурсией: рекурсивно перечисление всех файлов в каталоге

Сначала импортируйте библиотеки, которые работают с файлами:

```
from os import listdir
from os.path import isfile, join, exists
```

Вспомогательная функция для чтения только файлов из каталога:

```
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
```

Другая вспомогательная функция для получения только подкаталогов:

```
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
```

Теперь используйте эти функции для рекурсивного получения всех файлов в каталоге и во всех его подкаталогах (с использованием генераторов):

```
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
```

Эта функция может быть упрощена, используя `yield from` :

```
def get_files_recursive(directory):
    yield from get_files(directory)
    for subdirectory in get_directories(directory):
        yield from get_files_recursive(subdirectory)
```

Итерация по генераторам параллельно

Для параллельной обработки нескольких генераторов используйте встроенный `zip` :

```
for x, y in zip(a,b):
    print(x,y)
```

Результаты в:

```
1 x
2 y
3 z
```

В python 2 вместо этого вы должны использовать `itertools.izip` . Здесь мы также видим, что все `zip` функции дают кортежи.

Обратите внимание, что zip остановит итерацию, как только закончится один из повторяющихся элементов. Если вы хотите выполнить итерацию до тех пор, пока она будет самой длинной, используйте `itertools.zip_longest()` .

Рефакторинг для составления списка

Предположим, у вас есть сложный код, который создает и возвращает список, начиная с пустого списка и многократно добавляя его:

```
def create():
    result = []
    # logic here...
    result.append(value) # possibly in several places
    # more logic...
    return result # possibly in several places

values = create()
```

Когда нецелесообразно заменять внутреннюю логику пониманием списка, вы можете превратить всю функцию в генератор на месте, а затем собрать результаты:

```
def create_gen():
    # logic...
    yield value
    # more logic
    return # not needed if at the end of the function, of course

values = list(create_gen())
```

Если логика рекурсивна, используйте `yield from` для включения всех значений из рекурсивного вызова в результат «сплющенного»:

```
def preorder_traversal(node):
```

```
    yield node.value
    for child in node.children:
        yield from preorder_traversal(child)
```

поиск

`next` функция полезна даже без итерации. Передача выражения генератора для `next` - это быстрый способ поиска первого вхождения элемента, соответствующего некоторому предикату. Процедурный код

```
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)
```

можно заменить на:

```
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.
```

Для этой цели может быть желательно создать псевдоним, например `first = next` , или функцию-оболочку для преобразования исключения:

```
def first(generator):
    try:
        return next(generator)
    except StopIteration:
        raise ValueError
```

Прочитайте Генераторы онлайн: https://riptutorial.com/ru/python/topic/292/генераторы

глава 57: граф-инструмент

Вступление

Инструменты python могут использоваться для создания графика

Examples

PyDotPlus

PyDotPlus - улучшенная версия старого проекта pydot, который предоставляет интерфейс Python для языка Dot Graphviz.

Монтаж

Для последней стабильной версии:

```
pip install pydotplus
```

Для версии разработки:

```
pip install https://github.com/carlos-jenkins/pydotplus/archive/master.zip
```

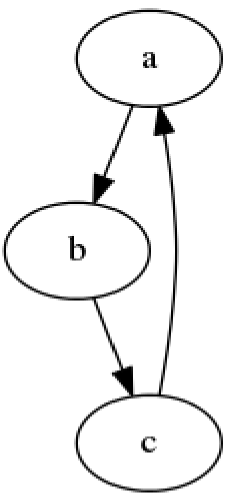
Загрузить график, определенный в файле DOT

- Предполагается, что файл находится в формате DOT. Он будет загружен, проанализирован и будет возвращен класс Dot, представляющий график. Например, простой 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.
```

Вы получите svg (Масштабируемая векторная графика), как это:



PyGraphviz

Получите PyGraphviz из Индекса пакета Python по адресу
http://pypi.python.org/pypi/pygraphviz

или установить его с помощью:

```
pip install pygraphviz
```

и будет предпринята попытка найти и установить соответствующую версию, соответствующую вашей операционной системе и версии Python.

Вы можете установить версию разработки (на github.com) с помощью:

```
pip install git://github.com/pygraphviz/pygraphviz.git#egg=pygraphviz
```

Получите PyGraphviz из Индекса пакета Python по адресу
http://pypi.python.org/pypi/pygraphviz

или установить его с помощью:

```
easy_install pygraphviz
```

и будет предпринята попытка найти и установить соответствующую версию, соответствующую вашей операционной системе и версии Python.

Загрузить график, определенный в файле DOT

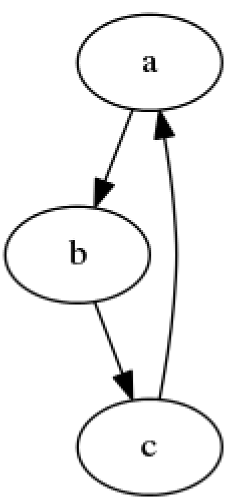
- Предполагается, что файл находится в формате DOT. Он будет загружен, проанализирован и будет возвращен класс Dot, представляющий график. Например, простой demo.dot:

  digraph demo1 {a -> b -> c; c -> a; }

- Загрузите его и нарисуйте.

```
import pygraphviz as pgv
G = pgv.AGraph("demo.dot")
G.draw('test', format='svg', prog='dot')
```

Вы получите svg (Масштабируемая векторная графика), как это:



Прочитайте граф-инструмент онлайн: https://riptutorial.com/ru/python/topic/9483/граф-инструмент

глава 58: группа по()

Вступление

В Python метод `itertools.groupby()` позволяет разработчикам группировать значения итерируемого класса на основе указанного свойства в другой итерируемый набор значений.

Синтаксис

- itertools.groupby (iterable, key = None или некоторая функция)

параметры

параметр	подробности
итерируемый	Любой python iterable
ключ	Функция (критерии), по которой можно группировать итерацию

замечания

groupby () является сложным, но общее правило, которое следует учитывать при его использовании, таково:

Всегда сортируйте элементы, которые хотите группировать, с тем же ключом, который вы хотите использовать для группировки

Рекомендуется, чтобы читатель взглянул на документацию здесь и посмотрел, как это объясняется с помощью определения класса.

Examples

Пример 1

Скажем, у вас есть строка

```
s = 'AAAABBBCCDAABBB'
```

и вы хотели бы разбить его так, чтобы все «A» были в одном списке и так со всеми «B» и « C» и т. д. Вы могли бы сделать что-то вроде этого

```
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
```

Результаты в

```
{'A': ['A', 'A', 'A', 'A', 'A', 'A'],
 'B': ['B', 'B', 'B', 'B', 'B', 'B'],
 'C': ['C', 'C'],
 'D': ['D']}
```

Но для большого набора данных вы будете наращивать эти элементы в памяти. Здесь groupby () входит в

Мы могли бы получить тот же результат более эффективным образом, выполнив следующие

```
# 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
```

Результаты в

```
{'A': ['A', 'A'], 'B': ['B', 'B', 'B'], 'C': ['C', 'C'], 'D': ['D']}
```

Обратите внимание, что число «A» в результате, когда мы использовали группу, меньше фактического числа «A» в исходной строке. Мы можем избежать этой потери информации, сортируя элементы в s, прежде чем передавать их на c, как показано ниже

```
c = groupby(sorted(s))

dic = {}
for k, v in c:
    dic[k] = list(v)
dic
```

Результаты в

```
{'A': ['A', 'A', 'A', 'A', 'A', 'A'], 'B': ['B', 'B', 'B', 'B', 'B', 'B'], 'C': ['C', 'C'],
'D': ['D']}
```

Теперь у нас есть все наши «А».

Пример 2.

В этом примере показано, как выбран ключ по умолчанию, если мы не укажем какой-либо

```
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
```

Результаты в

```
{1: [1, 1],
 2: [2],
 3: [3],
 ('persons', 'man', 'woman'): [('persons', 'man', 'woman')],
 'cow': ['cow'],
 'dog': ['dog'],
 10: [10],
 11: [11],
 'goat': ['goat']}
```

Обратите внимание, что кортеж в целом считается одним из ключевых в этом списке

Пример 3.

Обратите внимание на этот пример, что мулато и верблюд не появляются в нашем результате. Появляется только последний элемент с указанным ключом. Последний результат для с фактически уничтожает два предыдущих результата. Но посмотрите новую версию, где у меня есть данные, отсортированные сначала на одном и том же ключе.

```
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
```

Результаты в

```
{'c': ['camel'],
 'd': ['dog', 'donkey'],
 'g': ['goat'],
 'm': ['mongoose', 'malloo'],
 'persons': [('persons', 'man', 'woman')],
 'w': ['wombat']}
```

Сортировка

```
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
```

Результаты в

```
['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']}
```

Пример 4.

В этом примере мы видим, что происходит, когда мы используем разные типы итераций.

```
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
```

Результаты в

```
{'animal': [('animal', 'bear'), ('animal', 'duck')],
 'plant': [('plant', 'cactus')],
 'vehicle': [('vehicle', 'harley'),
  ('vehicle', 'speed boat'),
  ('vehicle', 'school bus')]}
```

Этот пример ниже, по существу, такой же, как и над ним. Единственное различие заключается в том, что я изменил все кортежи на списки.

```
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
```

Результаты

```
{'animal': [['animal', 'bear'], ['animal', 'duck']],
 'plant': [['plant', 'cactus']],
 'vehicle': [['vehicle', 'harley'],
  ['vehicle', 'speed boat'],
  ['vehicle', 'school bus']]}
```

Прочитайте группа по() онлайн: https://riptutorial.com/ru/python/topic/8690/группа-по--

глава 59: Дата и время

замечания

Python предоставляет встроенные методы и внешние библиотеки для создания, модификации, разбора и управления датами и временем.

Examples

Разбор строки в объекте datetime, посвященном часовому поясу

Python 3.2+ поддерживает формат `%z` при разборе строки в объект `datetime` .

> UTC в форме `+HHMM` или `-HHMM` (пустая строка, если объект наивен).

Python 3.x 3.2

```
import datetime
dt = datetime.datetime.strptime("2016-04-15T08:27:18-0500", "%Y-%m-%dT%H:%M:%S%z")
```

Для других версий Python вы можете использовать внешнюю библиотеку, такую как `dateutil` , которая `dateutil` анализирует строку с `dateutil` в объект `datetime` .

```
import dateutil.parser
dt = dateutil.parser.parse("2016-04-15T08:27:18-0500")
```

Теперь переменная `dt` представляет собой объект `datetime` со следующим значением:

```
datetime.datetime(2016, 4, 15, 8, 27, 18, tzinfo=tzoffset(None, -18000))
```

Простая арифметика даты

Даты не существуют изолированно. Обычно вам нужно найти количество времени между датами или определить, какая дата будет завтра. Это может быть выполнено с использованием объектов `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)
```

Это приведет к результатам, аналогичным:

```
Today: 2016-04-15
Yesterday: 2016-04-14
Tomorrow: 2016-04-16
Difference between tomorrow and yesterday: 2 days, 0:00:00
```

Использование базовых объектов datetime

Модуль datetime содержит три основных типа объектов - дату, время и дату и время.

```
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)
```

Арифметические операции для этих объектов поддерживаются только внутри одного и того же типа данных, и выполнение простой арифметики с экземплярами разных типов приведет к типу 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)
```

Итерация по датам

Иногда вы хотите перебирать диапазон дат от даты начала до некоторой даты окончания. Вы можете сделать это с использованием библиотеки `datetime` и объекта `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)
```

Что производит:

```
2016-07-21
2016-07-22
2016-07-23
2016-07-24
2016-07-25
2016-07-26
2016-07-27
```

Разбор строки с коротким именем часового пояса в объекте datetime, относящемся к часовому поясу

Используя библиотеку `dateutil` как в предыдущем примере, при разборе временных меток времени , можно также разобрать временные метки с указанным «коротким» именем часового пояса.

Для дат, отформатированных с короткими названиями зон или аббревиатурами, которые обычно неоднозначны (например, КНТ, которая может быть Центральным стандартным временем, Стандартным временем в Китае, Стандартным временем Кубы и т. Д. - здесь можно найти) или не обязательно доступна в стандартной базе данных , необходимо указать отображение между аббревиатурой часового пояса и объектом `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)
```

После выполнения этого:

```
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'))
```

Стоит отметить, что при использовании часового пояса `pytz` с этим методом он *не* будет правильно локализован:

```
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})
```

Это просто привязывает часовой пояс `pytz` к дате и времени:

```
dt.tzinfo # Will be in Local Mean Time!
# <DstTzInfo 'America/New_York' LMT-1 day, 19:04:00 STD>
```

Если вы используете этот метод, вы, вероятно, должны повторно `localize` наивную часть даты и времени после разбора:

```
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>)
```

Построение временных дат, относящихся к часовому поясу

По умолчанию все объекты `datetime` наивны. Чтобы они были осведомлены о часовом поясе, вы должны прикрепить объект `tzinfo` , который обеспечивает сокращение UTC и сокращение временной зоны как функцию даты и времени.

Исправлены смещенные временные зоны

Для часовых поясов, которые являются фиксированным смещением от UTC, в Python 3.2+, модуль `datetime` предоставляет класс `timezone` , конкретную реализацию `tzinfo` , которая принимает параметр `timedelta` и (необязательный) name:

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'
```

Для версий Python до 3.2 необходимо использовать стороннюю библиотеку, такую как `dateutil` . `dateutil` предоставляет эквивалентный класс `tzoffset` , который (`tzoffset` с версии 2.5.3) принимает аргументы формы `dateutil.tz.tzoffset(tzname, offset)` , где `offset` указано в секундах:

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'
```

Зоны с летним временем

Для зон с летним временем стандартные стандартные библиотеки python не предоставляют стандартный класс, поэтому необходимо использовать стороннюю библиотеку. `pytz` и `dateutil` популярные библиотеки , обеспечивающие классы часовых поясов.

Помимо статических часовых поясов, `dateutil` предоставляет классы часовых поясов, которые используют летнее время (см. Документацию для модуля `tz`). Вы можете использовать метод `tz.gettz()` для получения объекта часового пояса, который затем может быть передан непосредственно конструктору `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
```

ПРЕДУПРЕЖДЕНИЕ . Начиная с версии 2.5.3, `dateutil` обрабатывает неоднозначные даты и всегда будет по умолчанию для более *поздней* даты. `dateutil` построить объект с `dateutil` представляющим, например, `2015-11-01 1:30 EDT-4` , так как это происходит *во* время

перехода на летнее время.

При использовании `pytz` все обработанные края обрабатываются должным образом, но часовые пояса `pytz` *не* должны быть непосредственно привязаны к часовым поясам через конструктор. Вместо этого временная зона `pytz` должна быть присоединена с использованием метода `localize` часового пояса:

```
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
```

Имейте в виду, что если вы выполняете арифметику datetime в часовом поясе `pytz` -aware, вы должны либо выполнить вычисления в формате UTC (если хотите абсолютное истекшее время), либо вы должны вызвать `normalize()` для результата:

```
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
```

Нечеткий анализ времени и времени (извлечение даты и времени из текста)

Можно извлечь дату из текста, используя синтаксический анализатор `dateutil` в режиме «нечеткого», в котором компоненты строки, не признанной как часть даты, игнорируются.

```
from dateutil.parser import parse

dt = parse("Today is January 1, 2047 at 8:21:00AM", fuzzy=True)
print(dt)
```

`dt` теперь является *объектом* `datetime` и вы увидите `datetime.datetime(2047, 1, 1, 8, 21)` .

Переключение между часовыми поясами

Чтобы переключаться между часовыми поясами, вам нужны объекты datetime, которые имеют информацию о часовом поясе.

```
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.
```

Разбор произвольной временной метки ISO 8601 с минимальными библиотеками

Python имеет ограниченную поддержку для разбора временных меток ISO 8601. Для `strptime` вам нужно точно знать, в каком формате он находится. В качестве осложнения строкой `datetime` является временная метка ISO 8601 с пространством в качестве разделителя и 6-разрядной фракцией:

```
str(datetime.datetime(2016, 7, 22, 9, 25, 59, 555555))
# '2016-07-22 09:25:59.555555'
```

но если фракция равна 0, дробная часть не выводится

```
str(datetime.datetime(2016, 7, 22, 9, 25, 59, 0))
# '2016-07-22 09:25:59'
```

Но эти 2 формы нуждаются в *другом* формате для `strptime` . Кроме того, `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` не может.

Существует библиотека с одним файлом, называемая `iso8601` которая правильно анализирует временные метки ISO 8601 и только их.

Он поддерживает дроби и часовые пояса, а `T` разделитель - с одной функцией:

```
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' ...>)
```

Если часовой пояс не установлен, значение `iso8601.parse_date` соответствует UTC. Зона по умолчанию может быть изменена с помощью аргумента ключевого слова `default_zone` . Примечательно, что если вместо значения по умолчанию выбрано значение « `None` , тогда

те временные метки, которые не имеют явного часового пояса, возвращаются вместо наивных datetimes:

```
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>)
```

Преобразование временной метки в дату и время

Модуль `datetime` может преобразовать `timestamp` POSIX в объект `datetime` ИТЦ.

Эпоха - 1 января 1970 года в полночь.

```
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)
```

Вычитание месяцев с даты точно

Использование модуля `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)
```

Использование модуля `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)
```

Вычисление временных разниц

модуль `timedelta` пригодится для вычисления различий между временами:

```
from datetime import datetime, timedelta
now = datetime.now()
then = datetime(2016, 5, 23)     # datetime.datetime(2016, 05, 23, 0, 0, 0)
```

Указание времени является необязательным при создании нового объекта `datetime`

```
delta = now-then
```

`delta` имеет тип `timedelta`

```
print(delta.days)
# 60
print(delta.seconds)
# 40826
```

Чтобы получить n день после и n дней до даты, мы могли бы использовать:

n день после даты:

```
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 день до даты:

```
 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)
```

Получить временную метку ISO 8601

Без часовой пояс с микросекундами

```
from datetime import datetime

datetime.now().isoformat()
# Out: '2016-07-31T23:08:20.886783'
```

С часовым поясом, с микросекундами

```
from datetime import datetime
from dateutil.tz import tzlocal

datetime.now(tzlocal()).isoformat()
# Out: '2016-07-31T23:09:43.535074-07:00'
```

С часовым поясом, без микросекунд

```
from datetime import datetime
from dateutil.tz import tzlocal

datetime.now(tzlocal()).replace(microsecond=0).isoformat()
# Out: '2016-07-31T23:10:30-07:00'
```

См. ISO 8601 для получения дополнительной информации о формате ISO 8601.

Прочитайте Дата и время онлайн: https://riptutorial.com/ru/python/topic/484/дата-и-время

глава 60: Двоичные данные

Синтаксис

- pack (fmt, v1, v2, ...)
- unpack (fmt, buffer)

Examples

Форматировать список значений в байтовый объект

```
from struct import pack

print(pack('I3c', 123, b'a', b'b', b'c'))  # b'{\x00\x00\x00abc'
```

Распаковать байтовый объект в соответствии со строкой формата

```
from struct import unpack

print(unpack('I3c', b'{\x00\x00\x00abc'))  # (123, b'a', b'b', b'c')
```

Упаковка структуры

Модуль « **struct** » предоставляет возможность собирать объекты python как непрерывный фрагмент байтов или сбрасывать кусок байтов в структуры python.

Функция pack принимает строку формата и один или несколько аргументов и возвращает двоичную строку. Это очень похоже на то, что вы форматируете строку, за исключением того, что вывод не является строкой, а блоком байтов.

```
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)
```

Выход:

> Собственный байт: маленький байтовый кусок: '\ x03 \ x00 \ x00 \ x00 \ x04 \ x00 \ x05' Байт chunk unpacked: (3, 4, 5) Byte chunk: '\ x03 \ x00 \ x00 \ x00 \ x04 \ x00 \ x05 \ x00'

Вы можете использовать сетевой порядок байтов с данными, полученными из сетевых или пакетных данных, чтобы отправить их в сеть.

```
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)
```

Выход:

> Байт байта: порядковый номер байта: '\ x03 \ x00 \ x04 \ x00 \ x05 \ x00'
>
> Байт байтового сетевого байтового порядка: '\ x00 \ x03 \ x00 \ x04 \ x00 \ x05'

Вы можете оптимизировать, избегая накладных расходов на выделение нового буфера, предоставляя буфер, который был создан ранее.

```
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)
```

Выход:

> Байтовый фрагмент: '\ x03 \ x00 \ x04 \ x00 \ x05 \ x00 \ x00 \ x00'
>
> Byte chunk: '\ x00 \ x00 \ x03 \ x00 \ x04 \ x00 \ x05 \ x00'

Прочитайте Двоичные данные онлайн: https://riptutorial.com/ru/python/topic/2978/двоичные-данные

глава 61: Декораторы

Вступление

Функции декоратора - это шаблоны проектирования программного обеспечения. Они динамически изменяют функциональные возможности функции, метода или класса без непосредственного использования подклассов или изменения исходного кода декорированной функции. При правильном использовании декораторы могут стать мощными инструментами в процессе разработки. В этом разделе рассматриваются реализации и применения функций декоратора в Python.

Синтаксис

- def decorator_function (f): pass # определяет декоратор с именем decorator_function
- @decorator_function
  def decoration_function (): pass # функция теперь завернута (украшена) decorator_function
- decor_function = decorator_function (`@decorator_function`) # это эквивалентно использованию синтаксического сахара `@decorator_function`

параметры

параметр	подробности
е	Функция, которая будет украшена (завернута)

Examples

Функция декоратора

Декораторы дополняют поведение других функций или методов. Любая функция, которая принимает функцию в качестве параметра и возвращает расширенную функцию, может использоваться в качестве **декоратора** .

```
# 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.")
```

`@` -Notation - синтаксический сахар, который эквивалентен следующему:

```
my_function = super_secret_function(my_function)
```

Это важно иметь в виду, чтобы понять, как работают декораторы. Этот «unsugared» синтаксис дает понять, почему функция декоратора принимает функцию в качестве аргумента и почему она должна возвращать другую функцию. Он также демонстрирует, что произойдет, если вы *не* вернете функцию:

```
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
```

Таким образом, мы обычно определяем *новую функцию* внутри декоратора и возвращаем ее. Эта новая функция сначала сделает то, что ей нужно сделать, затем вызовет исходную функцию и, наконец, обработает возвращаемое значение. Рассмотрим эту простую функцию декоратора, которая печатает аргументы, которые получает исходная функция, а затем вызывает ее.

```
#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.
```

Класс декоратора

Как упоминалось во введении, декоратор - это функция, которая может быть применена к

другой функции, чтобы увеличить ее поведение. Синтаксический сахар эквивалентен следующему: `my_func = decorator(my_func)` . Но что, если `decorator` был вместо класса? Синтаксис все равно будет работать, за исключением того, что теперь `my_func` заменяется экземпляром класса `decorator` . Если этот класс реализует магический метод `__call__()` , тогда все же можно будет использовать `my_func` как если бы это была функция:

```
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.
```

Обратите внимание, что функция, украшенная декоратором класса, больше не будет считаться «функцией» с точки зрения проверки типов:

```
import types
isinstance(testfunc, types.FunctionType)
# False
type(testfunc)
# <class '__main__.Decorator'>
```

Методы отделки

Для методов декорирования вам необходимо определить дополнительный метод `__get__` :

```
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()
```

Внутри декоратора.

Предупреждение!

Class Decorators производят только один экземпляр для определенной функции, поэтому для украшения метода с помощью декоратора класса будет использоваться один и тот же декоратор между всеми экземплярами этого класса:

```
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
```

Создание декоратора выглядит как украшенная функция

Декораторы обычно сбрасывают метаданные функции, так как они не совпадают. Это может вызвать проблемы при использовании метапрограммирования для динамического доступа к метаданным функций. Метаданные также включают в себя функциональные docstrings и его имя. `functools.wraps` заставляет украшенную функцию выглядеть как оригинальная функция, копируя несколько атрибутов в функцию обертки.

```
from functools import wraps
```

Два метода обертывания декоратора - это то же самое, что скрывать, что оригинальная функция была украшена. Нет причин предпочитать версию функции версии класса, если вы уже не используете ее над другой.

Как функция

```
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__
```

'тестовое задание'

Как класс

```
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__
```

«Докстринк теста».

Декоратор с аргументами (фабрика декораторов)

Декоратор принимает только один аргумент: функцию, которую нужно украсить. Невозможно передать другие аргументы.

Но часто требуются дополнительные аргументы. Трюк заключается в том, чтобы сделать функцию, которая принимает произвольные аргументы и возвращает декоратор.

Функции декоратора

```
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()
```

Декоратор хочет сказать вам: Hello World

Важная заметка:

С такими фабриками декораторов вы **должны** называть декоратор парой круглых скобок:

```
@decoratorfactory # Without parentheses
def test():
    pass

test()
```

TypeError: decorator () отсутствует 1 требуемый позиционный аргумент: 'func'

Классы декораторов

```
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()
```

Внутри декоратора с аргументами (10,)

Создать одноэлементный класс с декоратором

Синглтон - это шаблон, который ограничивает экземпляр класса одним экземпляром / объектом. Используя декоратор, мы можем определить класс как singleton, заставив класс либо вернуть существующий экземпляр класса, либо создать новый экземпляр (если он не существует).

```
def singleton(cls):
    instance = [None]
    def wrapper(*args, **kwargs):
        if instance[0] is None:
            instance[0] = cls(*args, **kwargs)
        return instance[0]

    return wrapper
```

Этот декоратор можно добавить к любому объявлению класса и убедиться, что создается не более одного экземпляра класса. Любые последующие вызовы возвращают уже существующий экземпляр класса.

```
@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
```

Поэтому не имеет значения, ссылаетесь ли вы на экземпляр класса через локальную переменную или создаете ли вы другой «экземпляр», вы всегда получаете один и тот же объект.

Используя декоратор для выполнения функции

```
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()
```

Прочитайте Декораторы онлайн: https://riptutorial.com/ru/python/topic/229/декораторы

глава 62: дескриптор

Examples

Простой дескриптор

Существует два разных типа дескрипторов. Дескрипторы данных определяются как объекты, которые определяют как метод `__get__()` и `__set__()` , тогда как дескрипторы без данных определяют только метод `__get__()` . Это различие важно при рассмотрении переопределений и пространства имен словаря экземпляра. Если дескриптор данных и запись в словаре экземпляра имеют одно и то же имя, дескриптор данных будет иметь приоритет. Однако, если вместо этого дескриптор не данных и запись в словаре экземпляра имеют одно и то же имя, запись словаря экземпляра будет иметь приоритет.

Чтобы создать дескриптор данных только для чтения, определите как **get** (), так и **set** () с помощью **set** (), вызывающего AttributeError при вызове. Определить метод **set** () с помощью заполнителя исключения, достаточного для того, чтобы сделать его дескриптором данных.

```
descr.__get__(self, obj, type=None) --> value
descr.__set__(self, obj, value) --> None
descr.__delete__(self, obj) --> None
```

Приведенный пример:

```
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
```

Двусторонние конверсии

Объекты дескриптора могут позволить связанным объектным атрибутам реагировать на изменения автоматически.

Предположим, мы хотим смоделировать осциллятор с заданной частотой (в герцах) и периоде (в секундах). Когда мы обновляем частоту, которую хотим обновить, и когда мы обновляем период, мы хотим, чтобы частота обновлялась:

```
 >>> 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
```

Мы выбираем одно из значений (частота, в Hertz) как «якорь», то есть тот, который можно установить без преобразования, и написать для него класс дескриптора:

```
class Hertz(object):
    def __get__(self, instance, owner):
        return self.value

    def __set__(self, instance, value):
        self.value = float(value)
```

«Другое» значение (период, в секундах) определяется в терминах якоря. Мы пишем класс дескриптора, который делает наши преобразования:

```
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)
```

Теперь мы можем написать класс осциллятора:

```
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
```

Прочитайте дескриптор онлайн: https://riptutorial.com/ru/python/topic/3405/дескриптор

глава 63: Джанго

Вступление

Django - это высокоуровневая система Python Web, которая поощряет быстрое развитие и чистый, прагматичный дизайн. Построенный опытными разработчиками, он заботится о многих проблемах веб-разработки, поэтому вы можете сосредоточиться на написании своего приложения без необходимости изобретать колесо. Это бесплатно и с открытым исходным кодом.

Examples

Hello World с Django

Сделайте простой пример `Hello World` используя ваш django.

давайте удостовериться, что у вас есть django, установленный на вашем ПК.

откройте терминал и введите: python -c "import django"
-> если ошибка не возникает, значит, django уже установлен.

Теперь давайте создадим проект в django. Для этого напишите ниже команду на терминале:
django-admin startproject HelloWorld

Команда Above создаст каталог HelloWorld.
Структура каталога будет выглядеть так:
Привет, мир
| --helloworld
| | - **INIT** .py
| | --settings.py
| | --urls.py
| | --wsgi.py
| --manage.py

Написание видов (ссылка из документации django)

Функция просмотра или короткое представление - это просто функция Python, которая принимает веб-запрос и возвращает ответ Web. Этот ответ может быть содержимым HTML-страницы веб-страницы или чего-то еще. Документация говорит, что мы можем писать функции вида где угодно, но лучше писать в views.py, размещенном в нашем каталоге проекта.

Вот представление, которое возвращает приветственное мировое сообщение. (Views.py)

```
from django.http import HttpResponse

define helloWorld(request):
    return HttpResponse("Hello World!! Django Welcomes You.")
```

давайте разобраться в коде, шаг за шагом.

- Сначала мы импортируем класс HttpResponse из модуля django.http.
- Затем мы определяем функцию helloWorld. Это функция просмотра. Каждая функция просмотра принимает объект HttpRequest в качестве своего первого параметра, который обычно называется запросом.

  Обратите внимание, что имя функции просмотра не имеет значения; он не должен быть определен определенным образом, чтобы Django мог его распознать. мы назвали его helloWorld здесь, так что, будет ясно, что он делает.

- Представление возвращает объект HttpResponse, содержащий сгенерированный ответ. Каждая функция просмотра отвечает за возврат объекта HttpResponse.

Для получения дополнительной информации о представлениях django нажмите здесь

Отображение URL-адресов в представлениях
Чтобы отобразить это представление по определенному URL-адресу, вам нужно создать URLconf;

До этого давайте поймем, как django обрабатывает запросы.

- Django определяет используемый корневой модуль URLconf.
- Django загружает этот модуль Python и ищет переменные urlpatterns. Это должен быть список Python экземпляров django.conf.urls.url ().
- Django запускает каждый шаблон URL по порядку и останавливается на первом, который соответствует запрашиваемому URL.
- Когда одно из регулярных выражений совпадает, Django импортирует и вызывает данное представление, которое является простой функцией Python.

Вот как выглядит наш URLconf:

```
from django.conf.urls import url
from . import views #import the views.py from current directory

urlpatterns = [
   url(r'^helloworld/$', views.helloWorld),
]
```

Для получения дополнительной информации о django Urls нажмите здесь

Теперь измените каталог на HelloWorld и напишите ниже команду на терминале.
python manage.py runningerver

по умолчанию сервер будет запущен на уровне 127.0.0.1:8000

Откройте браузер и введите 127.0.0.1:8000/helloworld/. На странице вы увидите «Hello World !! Django Welcomes You».

Прочитайте Джанго онлайн: https://riptutorial.com/ru/python/topic/8994/джанго

глава 64: Доступ к атрибутам

Синтаксис

- `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

Доступ к базовому атрибуту с использованием точечной нотации

Возьмем образец класса.

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

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

В Python вы можете получить доступ к *названию* атрибута класса с помощью точечной нотации.

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

Если атрибут не существует, Python выдает ошибку:

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

Setters, Getters & Properties

Для инкапсуляции данных иногда вы хотите иметь атрибут, значение которого исходит от других атрибутов или, в общем, какое значение должно вычисляться в данный момент. Стандартный способ справиться с этой ситуацией - создать метод, называемый getter или setter.

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

В приведенном выше примере легко увидеть, что произойдет, если мы создадим новую книгу, содержащую название и автора. Если у всех книг, которые мы должны добавить в нашу Библиотеку, есть авторы и названия, мы можем пропустить геттеры и сеттеры и использовать точечную нотацию. Однако предположим, что у нас есть несколько книг, у которых нет автора, и мы хотим установить автора в «Неизвестный». Или, если у них есть несколько авторов, и мы планируем вернуть список авторов.

В этом случае мы можем создать getter и setter для атрибута *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
```

Эта схема не рекомендуется.

Одна из причин заключается в том, что есть улов: предположим, что мы разработали наш класс с общедоступным атрибутом и без методов. Люди уже много использовали, и они написали такой код:

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

Теперь у нас есть проблема. Потому что *автор* не является атрибутом! Python предлагает решение этой проблемы, называемой свойствами. Метод получения свойств украшен параметром @property перед его заголовком. Метод, который мы хотим использовать в качестве сеттера, украшен атрибутом @ attributeName.setter перед ним.

Помня об этом, теперь у нас есть новый обновленный класс.

```
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
```

Обратите внимание, что обычно Python не позволяет вам иметь несколько методов с тем же именем и различным количеством параметров. Однако в этом случае Python допускает это из-за используемых декораторов.

Если мы проверим код:

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

Прочитайте Доступ к атрибутам онлайн: https://riptutorial.com/ru/python/topic/4392/доступ-к-атрибутам

глава 65: Доступ к базе данных

замечания

Python может обрабатывать множество различных типов баз данных. Для каждого из этих типов существует другой API. Поэтому поощряем сходство между этими различными API, PEP 249.

> Этот API был определен для поощрения сходства между модулями Python, которые используются для доступа к базам данных. Делая это, мы надеемся достичь согласованности, ведущей к более понятным модулям, коду, который, как правило, более переносим по базам данных и более широкому доступу к подключению к базе данных от Python. PEP-249

Examples

Доступ к базе данных MySQL с использованием MySQLdb

Первое, что вам нужно сделать, это создать соединение с базой данных с помощью метода connect. После этого вам понадобится курсор, который будет работать с этим соединением.

Используйте метод execute курсора для взаимодействия с базой данных, и каждый раз в то время фиксируйте изменения, используя метод commit объекта соединения.

Как только все будет сделано, не забудьте закрыть курсор и соединение.

Вот класс Dbconnect со всем, что вам нужно.

```
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()
```

Взаимодействие с базой данных простое. После создания объекта просто используйте метод execute.

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

Если вы хотите вызвать хранимую процедуру, используйте следующий синтаксис. Обратите внимание, что список параметров не является обязательным.

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

По завершении запроса вы можете получить доступ к результатам несколькими способами. Объект cursor - это генератор, который может извлекать все результаты или зацикливаться.

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

Если вы хотите, чтобы цикл использовал непосредственно генератор:

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

Если вы хотите зафиксировать изменения в базе данных:

```
db.commit_db()
```

Если вы хотите закрыть курсор и соединение:

```
db.close_db()
```

SQLite

SQLite - это легкая база данных на основе дисков. Поскольку для него не требуется отдельный сервер базы данных, он часто используется для прототипирования или для небольших приложений, которые часто используются одним пользователем или одним пользователем в данный момент времени.

```
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()
```

Приведенный выше код подключается к базе данных, хранящейся в файле с именем `users.db` , `users.db` создавая файл, если он еще не существует. Вы можете взаимодействовать с базой данных с помощью операторов SQL.

Результатом этого примера должно быть:

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

Синтаксис SQLite: углубленный анализ

Начиная

1. Импортируйте модуль sqlite, используя

   ```
   >>> import sqlite3
   ```

2. Чтобы использовать модуль, вы должны сначала создать объект Connection, который представляет базу данных. Здесь данные будут сохранены в файле example.db:

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

   Кроме того, вы также можете указать специальное имя `:memory:` создать временную базу данных в ОЗУ следующим образом:

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

3. После того, как у вас есть `Connection` , вы можете создать объект `Cursor` и вызвать его метод `execute()` для выполнения команд 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()
```

Важные атрибуты и функции `Connection`

1. `isolation_level`

   Это атрибут, используемый для получения или установки текущего уровня изоляции. Нет для режима автосохранения или один из `DEFERRED` , `IMMEDIATE` или `EXCLUSIVE` .

2. `cursor`

   Объект cursor используется для выполнения SQL-команд и запросов.

3. `commit()`

   Выполняет текущую транзакцию.

4. `rollback()`

   Отбрасывает любые изменения, сделанные с предыдущего вызова `commit()`

5. `close()`

   Закрывает соединение с базой данных. Он не вызывает `commit()` автоматически. Если `close()` вызывается без первого вызова `commit()` (при условии, что вы не находитесь в режиме autocommit), все сделанные изменения будут потеряны.

6. `total_changes`

   Атрибут, который регистрирует общее количество строк, измененных, удаленных или вставленных с момента открытия базы данных.

7. `execute` , `executemany` и `executescript`

   Эти функции выполняются так же, как и объекты курсора. Это ярлык, поскольку вызов этих функций через объект соединения приводит к созданию промежуточного объекта курсора и вызывает соответствующий метод объекта курсора

8. `row_factory`

   Вы можете изменить этот атрибут на вызываемый, который принимает курсор и исходную строку как кортеж и вернет строку реального результата.

   ```
   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
```

Важные функции `Cursor`

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

   Выполняет *один* оператор SQL. Оператор SQL может быть параметризован (т. Е. Заполнители вместо SQL-литералов). Модуль sqlite3 поддерживает два типа заполнителей: вопросительные знаки `?` («Стиль qmark») и названные заполнители `:name` («named style»).

   ```
   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())
   ```

   Остерегайтесь: не используйте `%s` для вставки строк в команды SQL, так как это может сделать вашу программу уязвимой для атаки SQL-инъекции (см. SQL Injection).

2. `executemany(sql, seq_of_parameters)`

   Выполняет команду SQL для всех последовательностей параметров или сопоставлений, найденных в последовательности sql. Модуль sqlite3 также позволяет использовать итератор, приводящий параметры вместо последовательности.

   ```
   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)
   ```

Вы также можете передавать объекты-итераторы в качестве параметра в executeemany, и функция будет перебирать каждый кортеж значений, возвращаемых итератором. Итератор должен возвращать кортеж значений.

```
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)`

Это нестандартный метод удобства для выполнения сразу нескольких операторов SQL. Сначала он выдает инструкцию `COMMIT` , затем выполняет SQL-скрипт, который он получает в качестве параметра.

`sql_script` может быть экземпляром `str` или `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
    );
    """)
```

Следующий набор функций используется вместе с `SELECT` в SQL. Чтобы получить данные после выполнения `SELECT` , вы можете обрабатывать курсор как итератор, вызывать метод `fetchone()` курсора для извлечения одной подходящей строки или вызвать `fetchall()` чтобы получить список соответствующих строк.

Пример формы итератора:

```
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()`

Выбирает следующую строку набора результатов запроса, возвращает одну последовательность или None, когда больше нет данных.

```
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)`

Выбирает следующий набор строк результата запроса (задается по размеру), возвращая список. Если размер опущен, fetchmany возвращает одну строку. Пустой список возвращается, когда больше строк не доступно.

```
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()`

   Выбирает все (остальные) строки результата запроса, возвращая список.

```
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)]
```

Типы данных SQLite и Python

SQLite поддерживает следующие типы: NULL, INTEGER, REAL, TEXT, BLOB.

Так преобразуются типы данных при переходе с SQL на Python или наоборот.

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

Доступ к базе данных PostgreSQL с помощью psycopg2

psycopg2 - самый популярный адаптер базы данных PostgreSQL, который является легким и эффективным. Это текущая реализация адаптера PostgreSQL.

> Его основными функциями являются полная реализация спецификации API Python DB API 2.0 и безопасности потоков (несколько потоков могут использовать одно и то же соединение)

Установление соединения с базой данных и создание таблицы

```
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()
```

Вставка данных в таблицу:

```
# 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)""")
```

Получение данных таблицы:

```
# 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]))
```

Результатом вышесказанного будет:

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

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

Итак, вот вы, вы теперь знаете половину всего, что вам нужно знать о ***psycopg2*** ! :)

База данных Oracle

Предпосылки:

- пакет cx_Oracle - см. здесь для всех версий
- Мгновенный клиент Oracle - для Windows x64 , Linux x64

Настроить:

- Установите пакет cx_Oracle как:

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

- Извлеките мгновенный клиент Oracle и установите переменные среды следующим образом:

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

Создание соединения:

```
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()
```

Получить версию базы данных:

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

Пример: ['12', '1', '0', '2', '0']

Выполнить запрос: SELECT

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

Выход будет в кортежах Python:

(10, «SYSADMIN», «IT-INFRA», 7)

(23, «HR ASSOCIATE», «ЛЮДСКИЕ РЕСУРСЫ», 6)

Выполнить запрос: INSERT

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

Когда вы выполняете операции вставки / обновления / удаления в базе данных Oracle, изменения доступны только в пределах вашего сеанса до тех пор, пока не будет выпущена `commit` . Когда обновленные данные привязаны к базе данных, они затем доступны для других пользователей и сеансов.

Выполнить запрос: INSERT с использованием переменных Bind

Ссылка

Переменные Bind позволяют повторно выполнять операторы с новыми значениями без накладных расходов на повторный анализ оператора. Переменные привязки улучшают перепрограммирование кода и могут снизить риск атаки 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()
```

Закрыть соединение:

```
_db_connection.close()
```

Метод close () закрывает соединение. Любые соединения, явно не закрытые, будут автоматически освобождены при завершении сценария.

соединение

Создание соединения

Согласно PEP 249, соединение с базой данных должно быть установлено с помощью конструктора `connect()` , который возвращает объект `Connection` . Аргументы для этого конструктора зависят от базы данных. Для соответствующих аргументов обратитесь к конкретным темам базы данных.

```
import MyDBAPI

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

Этот объект соединения имеет четыре метода:

1: закрыть

```
con.close()
```

Немедленно закрывает соединение. Обратите внимание, что соединение автоматически закрывается, если `Connection.__del__` метод `Connection.__del__` . Любые незавершенные транзакции будут скрытно отклонены.

2: фиксация

```
con.commit()
```

Записывает любую ожидающую транзакцию в базу данных.

3: откат

```
con.rollback()
```

Возврат к началу любой ожидающей транзакции. Другими словами: это отменяет любую транзакцию без привязки к базе данных.

4: курсор

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

Возвращает объект `Cursor` . Это используется для транзакций в базе данных.

Использование sqlalchemy

Использовать sqlalchemy для базы данных:

```
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
```

Теперь этот движок можно использовать: например, с помощью pandas для извлечения данных из базы данных непосредственно из mysql

```
import pandas as pd

con = engine.connect()
```

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

Прочитайте Доступ к базе данных онлайн: https://riptutorial.com/ru/python/topic/4240/доступ-к-базе-данных

глава 66: Доступ к исходному коду и байт-коду Python

Examples

Отображение байт-кода функции

Интерпретатор Python компилирует код в байт-код перед выполнением его на виртуальной машине Python (см. Также « Что такое байт-код python?» .

Вот как просмотреть байт-код функции 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)
```

Функция `dis.dis` в модуле dis вернет декомпилированный байт-код переданной ему функции.

Изучение объекта кода функции

CPython позволяет получить доступ к объекту кода для объекта функции.

Объект `__code__` содержит исходный байт-код (`co_code`) функции, а также другую информацию, такую как константы и имена переменных.

```
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__)
```

Отобразить исходный код объекта

Объекты, которые не встроены

Чтобы распечатать исходный код объекта Python, `inspect` . Обратите внимание, что это не

будет работать для встроенных объектов или для объектов, определенных в интерактивном режиме. Для этого вам понадобятся другие методы, объясненные позже.

Вот как напечатать исходный код метода `randint` из `random` модуля:

```
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)
```

Чтобы просто распечатать строку документации

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

Распечатайте полный путь к файлу, где определен метод `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
```

Объекты, определенные интерактивно

Если объект определен в интерактивном режиме `inspect` не может предоставить исходный код , но вы можете использовать `dill.source.getsource` вместо

```
# 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
```

Встроенные объекты

Исходный код для встроенных функций Python написан на **c** и доступен только при просмотре исходного кода Python (размещенном на Mercurial или загружаемом с https://www.python.org/downloads/source/) .

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

Прочитайте Доступ к исходному коду и байт-коду Python онлайн: https://riptutorial.com/ru/python/topic/4351/доступ-к-исходному-коду-и-байт-коду-python

глава 67: Задавать

Синтаксис

- empty_set = set () # инициализировать пустой набор
- literal_set = {'foo', 'bar', 'baz'} # построить набор из 3 строк внутри него
- set_from_list = set (['foo', 'bar', 'baz']) # вызвать функцию set для нового набора
- set_from_iter = set (x для x в диапазоне (30)) # использовать произвольные итерации для создания набора
- set_from_iter = {x для x в [random.randint (0,10) для i в диапазоне (10)]} # альтернативная нотация

замечания

Наборы *неупорядочены* и имеют *очень быстрое время поиска* (амортизируются O (1), если вы хотите получить техническую информацию). Это здорово использовать, когда у вас есть коллекция вещей, порядок не имеет значения, и вы будете искать предметы по имени много. Если имеет смысл искать элементы по номеру индекса, подумайте о том, чтобы использовать список. Если порядок имеет значение, рассмотрите также список.

Наборы являются *изменяемыми* и, следовательно, не могут быть хэшированы, поэтому вы не можете использовать их в качестве ключей словаря или помещать их в другие наборы или где-либо еще, что требует типов хеширования. В таких случаях вы можете использовать неизменяемый `frozenset` .

Элементы набора должны быть *хешируемыми* . Это означает, что они имеют правильный метод `__hash__` , что согласуется с `__eq__` . В общем, изменяемые типы, такие как `list` или `set` , не являются хешируемыми и не могут быть помещены в набор. Если вы столкнулись с этой проблемой, рассмотрите использование ключей `dict` и immutable.

Examples

Получить уникальные элементы списка

Допустим, у вас есть список ресторанов - возможно, вы прочитали его из файла. Вы заботитесь о *уникальных* ресторанах в списке. Лучший способ получить уникальные элементы из списка - превратить его в набор:

```
restaurants = ["McDonald's", "Burger King", "McDonald's", "Chicken Chicken"]
unique_restaurants = set(restaurants)
print(unique_restaurants)
# prints {'Chicken Chicken', "McDonald's", 'Burger King'}
```

Обратите внимание, что набор не находится в том же порядке, что и исходный список; это потому, что наборы *неупорядочены* , точно так же, как `dict` s.

Это можно легко преобразовать в `List` с встроенной функцией `list` Python, предоставив другой список, который является тем же самым списком, что и оригинал, но без дубликатов:

```
list(unique_restaurants)
# ['Chicken Chicken', "McDonald's", 'Burger King']
```

Также принято рассматривать это как одну строку:

```
# Removes all duplicates and returns another list
list(set(restaurants))
```

Теперь любые операции, которые могут быть выполнены в исходном списке, могут быть выполнены снова.

Операции над наборами

с другими наборами

```
# 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
```

с отдельными элементами

```
# 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!
```

Заданные операции возвращают новые наборы, но имеют соответствующие версии на месте:

метод	работа на месте	метод на месте
союз	s | = t	Обновить
пересечение	s & = t	intersection_update
разница	s - = t	difference_update
symmetric_difference	s ^ = t	symmetric_difference_update

Например:

```
s = {1, 2}
s.update({3, 4})   # s == {1, 2, 3, 4}
```

Установки по сравнению с мультимножествами

Наборы представляют собой неупорядоченные коллекции отдельных элементов. Но иногда мы хотим работать с неупорядоченными наборами элементов, которые не обязательно различны и отслеживают множественность элементов.

Рассмотрим этот пример:

```
>>> setA = {'a','b','b','c'}
>>> setA
set(['a', 'c', 'b'])
```

Сохраняя строки `'a'` , `'b'` , `'b'` , `'c'` в заданную структуру данных, мы потеряли информацию о том, что `'b'` происходит дважды. Конечно, сохранение элементов в списке сохранит эту информацию

```
>>> listA = ['a','b','b','c']
```

```
>>> listA
['a', 'b', 'b', 'c']
```

но структура данных списка вводит дополнительное ненужное упорядочение, которое замедлит наши вычисления.

Для реализации мультимножеств Python предоставляет класс `Counter` из модуля `collections` (начиная с версии 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` - это словарь, в котором где элементы хранятся в виде словарных ключей, а их счетчики хранятся в виде значений словаря. И как все словари, это неупорядоченная коллекция.

Установить операции с использованием методов и встроенных

Определим два множества `a` и `b`

```
>>> a = {1, 2, 2, 3, 4}
>>> b = {3, 3, 4, 4, 5}
```

> ПРИМЕЧАНИЕ. `{1}` создает набор из одного элемента, но `{}` создает пустой `dict` . Правильный способ создания пустого набора `set()` .

пересечение

`a.intersection(b)` возвращает новый набор с элементами, присутствующими как в `a` и в `b`

```
>>> a.intersection(b)
{3, 4}
```

союз

`a.union(b)` возвращает новый набор с элементами, присутствующими в `a` или `b`

```
>>> a.union(b)
{1, 2, 3, 4, 5}
```

разница

`a.difference(b)` возвращает новый набор с элементами, присутствующими в `a` но не в `b`

```
>>> a.difference(b)
{1, 2}
>>> b.difference(a)
{5}
```

Симметричная разница

`a.symmetric_difference(b)` возвращает новый набор с элементами, присутствующими в `a` или `b` но не в обоих

```
>>> a.symmetric_difference(b)
{1, 2, 5}
>>> b.symmetric_difference(a)
{1, 2, 5}
```

ПРИМЕЧАНИЕ : `a.symmetric_difference(b) == b.symmetric_difference(a)`

Подмножество и надмножество

`c.issubset(a)` проверяет, находится ли каждый элемент `c` в `a` .

`a.issuperset(c)` проверяет, находится ли каждый элемент `c` в `a` .

```
>>> c = {1, 2}
>>> c.issubset(a)
True
>>> a.issuperset(c)
True
```

Последние операции имеют эквивалентные операторы, как показано ниже:

метод	оператор
`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`

метод	оператор
a.issuperset(b)	a >= b

Непересекающиеся множества

Устанавливает `a` и `d` не пересекаются, если ни один элемент в `a` также не находится в `d` и наоборот.

```
>>> 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
```

Тестирование членства

Встроенный `in` поисках ключевых слов для появлений

```
>>> 1 in a
True
>>> 6 in a
False
```

длина

Функция builtin `len()` возвращает количество элементов в наборе

```
>>> len(a)
4
>>> len(b)
3
```

Набор наборы

```
{{1,2}, {3,4}}
```

приводит к:

```
TypeError: unhashable type: 'set'
```

Вместо этого используйте `frozenset` :

```
{frozenset({1, 2}), frozenset({3, 4})}
```

Прочитайте Задавать онлайн: https://riptutorial.com/ru/python/topic/497/задавать

глава 68: Запись в CSV из строки или списка

Вступление

Запись в CSV-файл не похожа на запись в обычный файл в большинстве случаев и довольно проста. Я, насколько это возможно, рассмотрю самый простой и эффективный подход к проблеме.

параметры

параметр	подробности
открыть (**«/ путь** /» , «режим»)	Укажите путь к файлу CSV
open (путь, **«режим»**)	Укажите режим открытия файла (чтение, запись и т. Д.).
csv.writer (**файл** , разделитель)	Пропустите открытый CSV-файл здесь
csv.writer (файл, **разделитель = ''**)	Укажите символ или шаблон разделителя

замечания

```
open( path, "wb")
```

`"wb"` - режим записи.

Параметр `b` в `"wb"` мы использовали, необходим, только если вы хотите открыть его в двоичном режиме, который требуется только в некоторых операционных системах, таких как Windows.

```
csv.writer ( csv_file, delimiter=',' )
```

Здесь разделитель, который мы использовали, заключается в том , что мы хотим, чтобы каждая ячейка данных в строке содержала первое имя, фамилию и возраст соответственно. Поскольку наш список разделен вдоль , тоже, оказывается весьма удобно для нас.

Examples

Пример базовой записи

```
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)
```

Добавление строки в качестве новой строки в файле CSV

```
def append_to_csv(input_string):
    with open("fileName.csv", "a") as csv_file:
        csv_file.write(input_row + "\n")
```

Прочитайте Запись в CSV из строки или списка онлайн:
https://riptutorial.com/ru/python/topic/10862/запись-в-csv-из-строки-или-списка

глава 69: Заявление о прохождении

Синтаксис

- проходить

замечания

Зачем вам когда-либо хотеть сказать переводчику явно ничего не делать? Python имеет синтаксическое требование, чтобы блоки кода (после `if` , `except` , `def` , `class` и т. Д.) Не могут быть пустыми.

Но иногда пустой блок кода полезен сам по себе. Пустой блок `class` может определять новый, другой класс, например исключение, которое можно поймать. Пустым `except` блока может быть самый простой способ выразить «просить прощения позже», если нечего просить прощения. Если итератор делает всю тяжелую работу, пустой `for` цикла , чтобы просто запустить итератор может быть полезным.

Поэтому, если в блоке кода ничего не должно происходить, для такого блока необходим `pass` чтобы не создавать `IndentationError` . В качестве альтернативы можно использовать любой оператор (включая только термин, который нужно оценить, например, литерал `Ellipsis` `...` или строку, чаще всего docstring), но в `pass` четко указано, что на самом деле ничего не должно происходить и не нужно чтобы быть фактически оценены и (по крайней мере временно) сохранены в памяти. Вот небольшой аннотированный сборник наиболее частых применений `pass` , которые пересекли мой путь - вместе с некоторыми комментариями по хорошей и плохой pratice.

- Игнорирование (все или) определенного типа `Exception` (пример из `xml`):

  ```
  try:
      self.version = "Expat %d.%d.%d" % expat.version_info
  except AttributeError:
      pass # unknown
  ```

  Примечание. Игнорирование всех типов повышений, как в следующем примере из `pandas` , обычно считается плохой практикой, поскольку оно также ловит исключения, которые, вероятно, должны быть переданы вызывающему абоненту, например `KeyboardInterrupt` или `SystemExit` (или даже `HardwareIsOnFireError` Как вы знаете вы не работаете в настраиваемом ящике с определенными определенными ошибками, о которых будет знать какое-либо вызывающее приложение?).

  ```
  try:
      os.unlink(filename_larry)
  ```

```
except:
    pass
```

Вместо этого используйте, по крайней мере, `except Error:` или в этом случае лучше, чем `except OSError:` считается гораздо лучшей практикой. Быстрый анализ всех модулей python, которые я установил, дал мне более 10% всех, `except ...: pass` инструкции `except ...: pass` выхватывают все исключения, поэтому он все еще является частым шаблоном в программировании на python.

- Получение класса исключений, который не добавляет нового поведения (например, в `scipy`):

```
class CompileError(Exception):
    pass
```

Аналогично, классы, предназначенные как абстрактный базовый класс, часто имеют явные пустые `__init__` или другие методы, которые должны получать подклассы. (например, `pebl`)

```
class _BaseSubmittingController(_BaseController):
    def submit(self, tasks): pass
    def retrieve(self, deferred_results): pass
```

- Тестирование этого кода выполняется правильно для нескольких тестовых значений, не заботясь о результатах (из `mpmath`):

```
for x, error in MDNewton(mp, f, (1,-2), verbose=0,
                         norm=lambda x: norm(x, inf)):
    pass
```

- В определениях классов или функций часто док-строка уже существует, поскольку *обязательное утверждение* должно выполняться как единственное в блоке. В таких случаях блок может содержать `pass` *в дополнение* к docstring, чтобы сказать: «Это действительно предназначено, чтобы ничего не делать», например, в `pebl` :

```
class ParsingError(Exception):
    """Error encountered while parsing an ill-formed datafile."""
    pass
```

- В некоторых случаях `pass` используется в качестве заполнителя, чтобы сказать: «Этот метод / класс / if-block / ... еще не реализован, но это будет место для этого», хотя я лично предпочитаю литерал `Ellipsis` `...` (ПРИМЕЧАНИЕ: только python-3), чтобы строго различать это и преднамеренное «no-op» в предыдущем примере. Например, если я пишу модель с широкими штрихами, я могу написать

```
def update_agent(agent):
    ...
```

где другие могут иметь

```
def update_agent(agent):
    pass
```

до

```
def time_step(agents):
    for agent in agents:
        update_agent(agent)
```

как напоминание о том, чтобы заполнить функцию `update_agent` в более поздней точке, но запустите несколько тестов, чтобы проверить, ведет ли остальная часть кода, как предполагалось. (Третьим вариантом для этого случая является `raise NotImplementedError` . Это полезно, в частности, для двух случаев: *«Этот абстрактный метод должен быть реализован каждым подклассом, нет общего способа определить его в этом базовом классе»* или *«Эта функция , с этим именем, еще не реализовано в этом выпуске, но это то, как его подпись будет выглядеть »*)

Examples

Игнорировать исключение

```
try:
    metadata = metadata['properties']
except KeyError:
    pass
```

Создайте новое Исключение, которое можно поймать

```
class CompileError(Exception):
    pass
```

Прочитайте Заявление о прохождении онлайн: https://riptutorial.com/ru/python/topic/6891/заявление-о-прохождении

глава 70: Идиомы

Examples

Инициализация ключа словаря

Предпочитайте метод `dict.get` если вы не уверены, присутствует ли ключ. Он позволяет вернуть значение по умолчанию, если ключ не найден. Традиционный метод `dict[key]` приведет к `KeyError` исключения `KeyError` .

Вместо того, чтобы делать

```
def add_student():
    try:
        students['count'] += 1
    except KeyError:
        students['count'] = 1
```

Делать

```
def add_student():
        students['count'] = students.get('count', 0) + 1
```

Переключение переменных

Чтобы переключить значение двух переменных, вы можете использовать распаковку кортежа.

```
x = True
y = False
x, y = y, x
x
# False
y
# True
```

Использовать тестирование ценности истины

Python будет неявно преобразовывать любой объект в логическое значение для тестирования, поэтому используйте его везде, где это возможно.

```
# 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
```

Это обычно дает более читаемый код и обычно намного безопаснее при работе с неожиданными типами.

Щелкните здесь, чтобы узнать, что будет оценено для `False` .

Проверить «__main__», чтобы избежать непредвиденного выполнения кода

Хорошая практика - проверить переменную `__name__` вызывающей программы перед выполнением кода.

```
import sys

def main():
    # Your code starts here

    # Don't forget to provide a return code
    return 0

if __name__ == "__main__":
    sys.exit(main())
```

Использование этого шаблона гарантирует, что ваш код будет выполнен только тогда, когда вы ожидаете его; например, при явном запуске файла:

```
python my_program.py
```

Преимущество, однако, возникает, если вы решите `import` файл в другую программу (например, если вы пишете его как часть библиотеки). Затем вы можете `import` свой файл, а ловушка `__main__` гарантирует, что никакой код не будет выполнен неожиданно:

```
# 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()
```

Прочитайте Идиомы онлайн: https://riptutorial.com/ru/python/topic/3070/идиомы

глава 71: Импорт модулей

Синтаксис

- import *module_name*
- import *module_name.submodule_name*
- from *module_name* import *
- от *module_name* импорта *submodule_name* [, *class_name, ИМЯ_ФУНКЦИИ,* ... и т.д.]
- from *module_name* import *some_name* как *new_name*
- от *module_name.submodule_name* импорта *class_name* [, *ИМЯ_ФУНКЦИИ,* ... и т.д.]

замечания

Импорт модуля позволит Python оценить весь код верхнего уровня в этом модуле, чтобы он *изучил* все функции, классы и переменные, которые содержит модуль. Если вы хотите, чтобы ваш модуль был импортирован где-то в другом месте, будьте осторожны с кодом верхнего уровня и инкапсулируйте его в `if __name__ == '__main__':` если вы не хотите, чтобы он выполнялся при импорте модуля.

Examples

Импорт модуля

Используйте оператор `import` :

```
>>> import random
>>> print(random.randint(1, 10))
4
```

`import module` импортирует модуль, а затем позволяет ссылаться на его объекты - значения, функции и классы, например, - используя синтаксис `module.name` . В приведенном выше примере импортируется `random` модуль, который содержит функцию `randint` . Поэтому, импортируя `random` вы можете вызвать `randint` с `random.randint` .

Вы можете импортировать модуль и назначить его другому имени:

```
>>> import random as rn
>>> print(rn.randint(1, 10))
4
```

Если файл python `main.py` находится в той же папке, что и `custom.py` . Вы можете импортировать его следующим образом:

```
import custom
```

Также можно импортировать функцию из модуля:

```
>>> from math import sin
>>> sin(1)
0.8414709848078965
```

Чтобы импортировать определенные функции глубже в модуль, оператор точки может использоваться **только** с левой стороны ключевого слова `import` :

```
from urllib.request import urlopen
```

В python у нас есть два способа вызова функции с верхнего уровня. Один - `import` а другой - `from` . Мы должны использовать `import` если у нас есть возможность столкновения имен. Предположим , что мы имеем `hello.py` файл и `world.py` файлы , имеющие ту же самую функцию с именем `function` . Тогда `import` заявка будет работать хорошо.

```
from hello import function
from world import function

function() #world's function will be invoked. Not hello's
```

В общем случае `import` предоставит вам пространство имен.

```
import hello
import world

hello.function() # exclusively hello's function will be invoked
world.function() # exclusively world's function will be invoked
```

Но если вы уверены, что в вашем проекте нет никакого имени функции, которую вы должны использовать `from` инструкции

Несколько импорта могут быть сделаны в одной строке:

```
>>> # 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
```

Ключевые слова и синтаксис, показанные выше, также могут использоваться в

комбинациях:

```
>>> 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
```

Импортирование определенных имен из модуля

Вместо импорта полного модуля вы можете импортировать только указанные имена:

```
from random import randint # Syntax "from MODULENAME import NAME1[, NAME2[, ...]]"
print(randint(1, 10))      # Out: 5
```

`from random` , потому что интерпретатор python должен знать, из какого ресурса он должен импортировать функцию или класс, а `import randint` указывает функцию или класс.

Другой пример ниже (аналогично приведенному выше):

```
from math import pi
print(pi)                  # Out: 3.14159265359
```

Следующий пример вызовет ошибку, потому что мы не импортировали модуль:

```
random.randrange(1, 10)    # works only if "import random" has been run before
```

Выходы:

```
NameError: name 'random' is not defined
```

Интерпретатор python не понимает, что вы имеете в виду со `random` . Он должен быть объявлен путем добавления `import random` примера к примеру:

```
import random
random.randrange(1, 10)
```

Импорт всех имен из модуля

```
from module_name import *
```

например:

```
from math import *
sqrt(2)    # instead of math.sqrt(2)
ceil(2.7)  # instead of math.ceil(2.7)
```

Это импортирует все имена, определенные в `math` модуле, в глобальное пространство имен, кроме имен, начинающихся с подчеркивания (что указывает на то, что автор считает, что он предназначен только для внутреннего использования).

Предупреждение . Если функция с тем же именем уже была определена или импортирована, она будет **перезаписана** . Почти всегда импортировать только определенные имена `from math import sqrt, ceil` **рекомендуется использовать** `from math import sqrt, ceil` :

```
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
```

Помеченный импорт разрешен только на уровне модуля. Попытки выполнить их в определениях классов или функций приводят к `SyntaxError` .

```
def f():
    from math import *
```

а также

```
class A:
    from math import *
```

оба терпят неудачу:

```
SyntaxError: import * only allowed at module level
```

Специальная переменная __all__

Модули могут иметь специальную переменную с именем `__all__` чтобы ограничить, какие переменные импортируются при использовании `from mymodule import *` .

Учитывая следующий модуль:

```
# mymodule.py
```

```
__all__ = ['imported_by_star']

imported_by_star = 42
not_imported_by_star = 21
```

Только `imported_by_star` импортируется при использовании `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
```

Однако `not_imported_by_star` можно импортировать явно:

```
>>> from mymodule import not_imported_by_star
>>> not_imported_by_star
21
```

Программный импорт

Python 2.x 2.7

Чтобы импортировать модуль через вызов функции, используйте модуль `importlib` (включенный в Python, начиная с версии 2.7):

```
import importlib
random = importlib.import_module("random")
```

Функция `importlib.import_module()` также импортирует подмодуль пакета напрямую:

```
collections_abc = importlib.import_module("collections.abc")
```

Для более старых версий Python используйте модуль `imp` .

Python 2.x 2.7

Используйте функции `imp.find_module` и `imp.load_module` для выполнения программного импорта.

Взято из стандартной библиотечной документации

```
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()
```

НЕ используйте `__import__()` для программного импорта модулей! Существуют тонкие детали, связанные с `sys.modules` , аргументом `fromlist` и т. `sys.modules` , `fromlist` легко упускать из вида, которые `importlib.import_module()` обрабатывает для вас.

Импорт модулей из произвольного расположения файловой системы

Если вы хотите импортировать модуль, который еще не существует в качестве встроенного модуля в стандартной библиотеке Python или в качестве побочного пакета, вы можете сделать это, добавив путь к каталогу, в котором ваш модуль находится в `sys.path` . Это может быть полезно, когда на хосте существует несколько сред python.

```
import sys
sys.path.append("/path/to/directory/containing/your/module")
import mymodule
```

Важно, чтобы вы добавили путь к *каталогу,* в котором `mymodule` найден `mymodule` , а не по пути к самому модулю.

Правила PEP8 для импорта

Некоторые рекомендуемые правила стиля PEP8 для импорта:

1. Импорт должен быть на отдельных линиях:

   ```
   from math import sqrt, ceil      # Not recommended
   from math import sqrt            # Recommended
   from math import ceil
   ```

2. Закажите импорт в верхней части модуля следующим образом:

   - Импорт стандартных библиотек
   - Связанный импорт третьей стороны
   - Локальный импорт приложений / библиотек

3. Следует избегать импорта подстановочных знаков, поскольку это приводит к путанице в именах в текущем пространстве имен. Если вы выполняете `from module import *` , может быть неясно, связано ли какое-либо имя в вашем коде с `module` или нет. Это вдвойне верно, если у вас есть несколько `from module import *` операторов `from module import *` .

4. Избегайте использования относительного импорта; вместо этого используйте явный импорт.

Импорт подмодулей

```
from module.submodule import function
```

Это импортирует `function` из `module.submodule` .

Функция __import __ ()

Функция `__import__()` может использоваться для импорта модулей, где имя известно только во время выполнения

```
if user_input == "os":
    os = __import__("os")

# equivalent to import os
```

Эта функция также может использоваться для указания пути файла к модулю

```
mod = __import__(r"C:/path/to/file/anywhere/on/computer/module.py")
```

Повторный импорт модуля

При использовании интерактивного интерпретатора вам может понадобиться перезагрузить модуль. Это может быть полезно, если вы редактируете модуль и хотите импортировать новейшую версию, или если вы обезврежили элемент существующего модуля и хотите вернуть свои изменения.

Обратите внимание, что вы **не можете** просто повторно `import` модуль для возврата:

```
import math
math.pi = 3
print(math.pi)    # 3
import math
print(math.pi)    # 3
```

Это объясняется тем, что интерпретатор регистрирует каждый импортируемый модуль. И когда вы пытаетесь переименовать модуль, интерпретатор видит его в регистре и ничего не делает. Таким образом, сложный способ reimport - использовать `import` после удаления соответствующего элемента из реестра:

```
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
```

Но есть более простой и простой способ.

Python 2

Используйте функцию `reload` :

Python 2.x 2.3

```
import math
math.pi = 3
print(math.pi)    # 3
reload(math)
print(math.pi)    # 3.141592653589793
```

Python 3

Функция `reload` переместилась в `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
```

Прочитайте Импорт модулей онлайн: https://riptutorial.com/ru/python/topic/249/импорт-модулей

глава 72: Индексация и нарезка

Синтаксис

- OBJ [начать: остановка: шаг]
- ломтик (стоп)
- срез (начало, остановка [, шаг])

параметры

Parameter	Описание
`obj`	Объект, из которого вы хотите извлечь «под-объект» из
`start`	Индекс `obj` , с которого вы хотите, чтобы суб-объект начинался (помните, что Python имеет нуль-индексирование, что означает, что первый элемент `obj` имеет индекс `0`). Если опустить, значение по умолчанию равно `0` .
`stop`	Индекс (не включительно) `obj` который вы хотите, чтобы подпоследовательный объект заканчивался. Если опущено, по умолчанию используется `len(obj)` .
`step`	Позволяет выбрать только каждый элемент `step` . Если опущено, значение по умолчанию равно `1` .

замечания

Вы можете унифицировать концепцию секущих строк с разбиением других последовательностей, просмотрев строки как неизменную коллекцию символов, с оговоркой, что символ Юникода представлен строкой длиной 1.

В математической нотации вы можете рассмотреть нарезку, чтобы использовать полуоткрытый интервал `[start, end)` , то есть начало включено, но конец не является. Полуоткрытая природа интервала имеет то преимущество, что `len(x[:n])` = `n` где `len(x)` > = `n` , а интервал, закрытый в начале, имеет то преимущество, что `x[n:n+1]` = `[x[n]]` где `x` - список с `len(x) >= n` , сохраняя при этом согласованность между индексацией и обозначением нарезки.

Examples

Базовая нарезка

Для любого итеративного (например, строки, списка и т. Д.) Python позволяет вам срезать и возвращать подстроку или подсписку своих данных.

Формат для нарезки:

```
iterable_name[start:stop:step]
```

где,

- `start` - это первый индекс среза. По умолчанию 0 (индекс первого элемента)
- `stop` один из последних индексов среза. По умолчанию len (iterable)
- `step` - размер шага (лучше объясняется примерами ниже)

Примеры:

```
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))
```

Кроме того, любой из вышеперечисленных может использоваться с заданным размером шага:

```
a[::2]       # "ace" (every 2nd element)
a[1:4:2]     # "bd" (from index 1, to index 4 (excluded), every 2nd element)
```

Индексы могут быть отрицательными, и в этом случае они вычисляются с конца последовательности

```
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())
```

Размеры шагов также могут быть отрицательными, и в этом случае срез будет перебираться по списку в обратном порядке:

```
a[3:1:-1]   # "dc" (from index 2 to None (default), in reverse order)
```

Эта конструкция полезна для изменения итерации

```
a[::-1]     # "fedcba" (from last element (default len()-1), to first, in reverse order(-1))
```

Обратите внимание на то, что для отрицательных шагов по умолчанию `end_index` не `None` (см

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)
```

Создание мелкой копии массива

Быстрый способ сделать копию массива (в отличие от назначения переменной с другой ссылкой на исходный массив):

```
arr[:]
```

Рассмотрим синтаксис. `[:]` означает, что `start` , `end` и `slice` опущены. По умолчанию они равны `0` , `len(arr)` и `1` соответственно, что означает, что подмассиво, которое мы запрашиваем, будет иметь все элементы `arr` от начала до самого конца.

На практике это выглядит примерно так:

```
arr = ['a', 'b', 'c']
copy = arr[:]
arr.append('d')
print(arr)    # ['a', 'b', 'c', 'd']
print(copy)   # ['a', 'b', 'c']
```

Как вы можете видеть, `arr.append('d')` добавил `d` к `arr` , но `copy` осталась неизменной!

Обратите внимание, что это делает *мелкую* копию и идентично `arr.copy()` .

Реверсирование объекта

Вы можете использовать срезы, чтобы очень легко отменить `str` , `list` или `tuple` (или в основном любой объект коллекции, который реализует срез с параметром step). Ниже приведен пример изменения строки, хотя это в равной степени относится к другим типам, перечисленным выше:

```
s = 'reverse me!'
s[::-1]    # '!em esrever'
```

Давайте быстро посмотрим на синтаксис. `[::-1]` означает, что срез должен быть от начала до конца строки (поскольку `start` и `end` опущены), а шаг `-1` означает, что он должен перемещаться по строке в обратном порядке.

Индексирование пользовательских классов: __getitem__, __setitem__ и __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]
```

Это позволяет разрезать и индексировать доступ к элементу:

```
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
```

В то время как установка и удаление элементов допускает только *разделенное* целочисленное индексирование (без нарезки):

```
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]
```

Назначение среза

Еще одна опрятная функция с использованием срезов - это назначение срезов. Python позволяет назначать новые срезы для замены старых фрагментов списка за одну операцию.

Это означает, что если у вас есть список, вы можете заменить несколько членов в одном назначении:

```
lst = [1, 2, 3]
lst[1:3] = [4, 5]
print(lst) # Out: [1, 4, 5]
```

Назначение также не должно совпадать по размеру, поэтому, если вы хотите заменить старый фрагмент новым срезом, который отличается по размеру, вы можете:

```
lst = [1, 2, 3, 4, 5]
lst[1:4] = [6]
print(lst) # Out: [1, 6, 5]
```

Также можно использовать известный синтаксис разреза, чтобы делать такие вещи, как замена всего списка:

```
lst = [1, 2, 3]
lst[:] = [4, 5, 6]
print(lst) # Out: [4, 5, 6]
```

Или только последние два члена:

```
lst = [1, 2, 3]
lst[-2:] = [4, 5, 6]
print(lst) # Out: [1, 4, 5, 6]
```

Объекты среза

Срезы являются объектами сами по себе и могут храниться в переменных со встроенной функцией `slice()` . Переменные среза могут использоваться, чтобы сделать ваш код более читаемым и способствовать повторному использованию.

```
>>> 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']
```

Базовая индексация

Списки Python основаны на 0, *т. Е.* К первому элементу в списке может обращаться индекс `0`

```
arr = ['a', 'b', 'c', 'd']
print(arr[0])
>> 'a'
```

Вы можете получить второй элемент в списке по индексу `1` , третьему элементу по индексу `2` и так далее:

```
print(arr[1])
>> 'b'
print(arr[2])
>> 'c'
```

Вы также можете использовать отрицательные индексы для доступа к элементам в конце списка. например. index `-1` даст вам последний элемент списка, а index `-2` предоставит вам второй элемент списка:

```
print(arr[-1])
>> 'd'
print(arr[-2])
>> 'c'
```

Если вы попытаетесь получить доступ к индексу, которого нет в списке, будет `IndexError` :

```
print arr[6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Прочитайте Индексация и нарезка онлайн: https://riptutorial.com/ru/python/topic/289/индексация-и-нарезка

глава 73: Инструмент 2to3

Синтаксис

- $ 2to3 [-options] path / to / file.py

параметры

параметр	Описание
имя_файла / имя_каталога	2to3 принимает список файлов или каталогов, которые должны быть преобразованы в качестве аргумента. Каталоги рекурсивно пройдены для источников Python.
вариант	**Вариант Описание**
-f FIX, --fix = FIX	Укажите применяемые преобразования; default: все. Список доступных преобразований с помощью `--list-fixes`
-j ПРОЦЕССЫ, --processes = ПРОЦЕССЫ	Запуск 2to3 одновременно
-x NOFIX, --nofix = NOFIX	Исключить преобразование
-l, --list-fixes	Список доступных преобразований
-p, --print-function	Измените грамматику так, чтобы `print()` считался функцией
-v, --verbose	Более подробный вывод
--no-дифференциалы	Не выводить разности рефакторинга
-w	Записывать измененные файлы
-n, --nobackups	Не создавать резервные копии измененных файлов
-o OUTPUT_DIR, --output-dir = OUTPUT_DIR	Поместите выходные файлы в этот каталог вместо перезаписывания входных файлов. Требуется флаг `-n` , поскольку резервные файлы не нужны, если входные файлы не изменены.
-W, --write-unchanged-files	Запись выходных файлов даже не требуется. Полезно с `-o`

параметр	Описание
	так что полное исходное дерево переводится и копируется. Подразумевает `-w` .
--add-суффикс = ADD_SUFFIX	Укажите строку, которая будет добавлена ко всем именам выходных файлов. Требуется `-n` если не пусто. Пример: `--add-suffix='3'` будет генерировать файлы `.py3` .

замечания

Инструмент 2to3 представляет собой программу python, которая используется для преобразования кода, написанного в Python 2.x, в код Python 3.x. Инструмент читает исходный код Python 2.x и применяет ряд исправлений, чтобы преобразовать его в действительный код Python 3.x.

Инструмент 2to3 доступен в стандартной библиотеке как lib2to3, который содержит богатый набор исправлений, которые будут обрабатывать практически весь код. Поскольку lib2to3 является общей библиотекой, можно написать свои собственные фиксаторы для 2to3.

Examples

Основное использование

Рассмотрим следующий код Python2.x. Сохраните файл как `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)
```

В приведенном выше файле имеется несколько несовместимых строк. Метод `raw_input()` был заменен на `input()` в Python 3.x, и `print` больше не является оператором, а функцией. Этот код можно преобразовать в код Python 3.x с помощью инструмента 2to3.

Юникс

```
$ 2to3 example.py
```

Windows

```
> path/to/2to3.py example.py
```

Выполнение вышеуказанного кода приведет к различиям в исходном исходном файле, как показано ниже.

```
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
```

Изменения могут быть записаны обратно в исходный файл с использованием флага -w. Создается резервная копия исходного файла с именем `example.py.bak` , если не указан флаг -n.

Юникс

```
$ 2to3 -w example.py
```

Windows

```
> path/to/2to3.py -w example.py
```

Теперь файл `example.py` был преобразован из Python 2.x в код Python 3.x.

После завершения `example.py` будет содержать следующий действительный код Python3.x:

Python 3.x 3.0

```
def greet(name):
    print("Hello, {0}!".format(name))
print("What's your name?")
name = input()
greet(name)
```

Прочитайте Инструмент 2to3 онлайн: https://riptutorial.com/ru/python/topic/5320/инструмент-2to3

глава 74: Интерпретатор (консоль командной строки)

Examples

Получение общей помощи

Если `help` функция вызывается в консоли без каких-либо аргументов, Python представляет собой интерактивную консоль справки, где вы можете узнать о модулях, символах, ключевых словах и т. Д. Python.

```
>>> 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".
```

Ссылаясь на последнее выражение

Чтобы получить значение последнего результата из вашего последнего выражения в консоли, используйте знак подчеркивания _ .

```
>>> 2 + 2
4
>>> _
4
>>> _ + 6
10
```

Это знаковое значение подчеркивания обновляется только при использовании выражения python, которое приводит к значению. Определение функций или циклов не изменяет значения. Если выражение вызывает исключение, изменений в _ .

```
>>> "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'
```

Помните, что эта волшебная переменная доступна только в интерактивном интерпретаторе python. Запуск сценариев не будет.

Открытие консоли Python

Консоль для основной версии Python обычно можно открыть, набрав `py` в консоль Windows или `python` на других платформах.

```
$ 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.
>>>
```

Если у вас несколько версий, то по умолчанию их исполняемые файлы будут сопоставлены с `python2` или `python3` соответственно.

Это, конечно, зависит от исполняемых файлов Python, находящихся в вашем PATH.

Переменная PYTHONSTARTUP

Вы можете установить переменную среды PYTHONSTARTUP для консоли Python. Всякий раз, когда вы входите в консоль Python, этот файл будет выполнен, что позволит вам добавить дополнительные функции в консоль, например, автоматически импортировать часто используемые модули.

Если для переменной PYTHONSTARTUP задано местоположение файла, содержащего это:

```
print("Welcome!")
```

Тогда открытие консоли Python приведет к дополнительному выводу:

```
$ 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!
>>>
```

Аргументы командной строки

Python имеет множество ключей командной строки, которые можно передать в `py` . Их можно найти, выполнив `py --help` , который дает этот вывод на 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.
```

Получение справки об объекте

Консоль Python добавляет новую функцию, `help` , которая может использоваться для получения информации о функции или объекте.

Для функции `help` печатает свою подпись (аргументы) и ее docstring, если функция имеет один.

```
>>> 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.
```

Для объекта `help` отображает docstring объекта и различные функции-члены, которые имеет объект.

```
>>> 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...
```

Прочитайте Интерпретатор (консоль командной строки) онлайн:

https://riptutorial.com/ru/python/topic/2473/интерпретатор--консоль-командной-строки-

глава 75: Интерфейс шлюза веб-сервера (WSGI)

параметры

параметр	подробности
start_response	Функция, используемая для обработки старта

Examples

Объект сервера (метод)

Каждому серверному объекту присваивается параметр «приложение», который может быть любым вызываемым объектом приложения (см. Другие примеры). Он сначала записывает заголовки, затем тело данных, возвращаемых нашим приложением, на стандартный вывод системы.

```
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()'
```

Прочитайте Интерфейс шлюза веб-сервера (WSGI) онлайн:
https://riptutorial.com/ru/python/topic/5315/интерфейс-шлюза-веб-сервера--wsgi-

глава 76: Исключения

Вступление

Ошибки, обнаруженные во время выполнения, называются исключениями и не являются безоговорочно фатальными. Большинство исключений не обрабатываются программами; можно писать программы, которые обрабатывают выбранные исключения. В Python есть специальные функции для обработки исключений и логики исключений. Кроме того, исключения имеют иерархию богатого типа, все наследуют от типа `BaseException` .

Синтаксис

- *исключение*
- raise # re-raise исключение, которое уже было поднято
- повысить *исключение* из *причины* # Python 3 - установить исключение причины
- *исключить исключение* из None # Python 3 - подавить весь контекст исключения
- пытаться:
- кроме *[типов исключений] [в* качестве *идентификатора]* :
- еще:
- в конце концов:

Examples

Повышение исключений

Если ваш код встречает условие, которое он не знает, как обращаться с ним, например неправильный параметр, он должен поднять соответствующее исключение.

```
def even_the_odds(odds):
    if odds % 2 != 1:
        raise ValueError("Did not get an odd number")

    return odds + 1
```

Устранение исключений

Используйте `try...except:` для исключения исключений. Вы должны указать как точное исключение, как можете:

```
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.
```

Указанный класс исключений - в этом случае - `ZeroDivisionError` - ловит любое исключение из этого класса или любого подкласса этого исключения.

Например, `ZeroDivisionError` является подклассом `ArithmeticError` :

```
>>> ZeroDivisionError.__bases__
(<class 'ArithmeticError'>,)
```

И вот, все еще поймают `ZeroDivisionError` :

```
try:
    5 / 0
except ArithmeticError:
    print("Got arithmetic error")
```

Запуск кода очистки с окончательным

Иногда вы можете захотеть, чтобы что-то произошло независимо от того, что произошло, например, если вам нужно очистить некоторые ресурсы.

`finally` блок условия `try` будет происходить независимо от того, были ли сделаны какие-либо исключения.

```
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)
```

Этот шаблон часто лучше обрабатывается с помощью менеджеров контекста (используя оператор `with`).

Переопределение исключений

Иногда вы хотите получить исключение только для его проверки, например, для ведения журнала. После проверки вы хотите, чтобы исключение продолжало распространяться, как и раньше.

В этом случае просто используйте оператор `raise` без параметров.

```
try:
    5 / 0
except ZeroDivisionError:
    print("Got an error")
    raise
```

Имейте в виду, однако, что кто-то еще в стеке вызывающего абонента все еще может поймать исключение и обработать его каким-то образом. Выполненный вывод может быть неприятным в этом случае, потому что это произойдет в любом случае (пойман или не пойман). Поэтому лучше было бы создать другое исключение, содержащее ваш комментарий о ситуации, а также оригинальное исключение:

```
try:
    5 / 0
except ZeroDivisionError as e:
    raise ZeroDivisionError("Got an error", e)
```

Но у этого есть недостаток в сокращении следа исключения до именно этого `raise` то время как `raise` без аргумента сохраняет исходный след исключения.

В Python 3 вы можете сохранить исходный стек с помощью синтаксиса `raise from` :

```
    raise ZeroDivisionError("Got an error") from e
```

Исключение цепочки с рейзом из

В процессе обработки исключения вы можете создать другое исключение. Например, если вы получаете `IOError` во время чтения из файла, вы можете захотеть поднять конкретную ошибку приложения для представления пользователям вашей библиотеки.

Python 3.x 3.0

Вы можете связать исключения, чтобы показать, как выполняется обработка исключений:

```
>>> 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
```

Иерархия исключений

Обработка исключений происходит на основе иерархии исключений, определяемой структурой наследования классов исключений.

Например, `IOError` и `OSError` являются подклассами `EnvironmentError` . Код, который ловит `IOError` , не поймает `OSError` . Однако код, который ловит объект `EnvironmentError` будет захватывать как `IOError` s, так и `OSError` s.

Иерархия встроенных исключений:

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
```

Исключения также являются объектами

Исключения - это просто обычные объекты Python, которые наследуются от встроенного `BaseException` . Сценарий Python может использовать оператор `raise` для прерывания выполнения, заставляя Python печатать трассировку стека стека вызовов в этой точке и представление экземпляра исключения. Например:

```
>>> 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!
```

в котором говорится, что `ValueError` с сообщением `'Example error!'` был поднят нашей `failing_function()` , которая была выполнена в интерпретаторе.

Код вызова может выбирать для обработки любых и всех типов исключений, которые может вызвать вызов:

```
>>> try:
...     failing_function()
... except ValueError:
...     print('Handled the error')
Handled the error
```

Вы можете получить объекты исключений, назначив их в части `except...` из кода обработки исключений:

```
>>> try:
...     failing_function()
... except ValueError as e:
...     print('Caught exception', repr(e))
Caught exception ValueError('Example error!',)
```

Полный список встроенных исключений Python вместе с их описаниями можно найти в документации на Python: https://docs.python.org/3.5/library/exceptions.html . И вот полный список, упорядоченный иерархически: Иерархия исключений .

Создание настраиваемых типов исключений

Создайте класс, наследующий от `Exception` :

```
class FooException(Exception):
    pass
try:
    raise FooException("insert description here")
except FooException:
    print("A FooException was raised.")
```

или другой тип исключения:

```
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))
```

Не поймайте все!

Хотя часто возникает соблазн поймать каждое `Exception` :

```
try:
    very_difficult_function()
except Exception:
    # log / try to reconnect / exit gratiously
finally:
    print "The END"
    # it runs no matter what execute.
```

Или даже все (включая `BaseException` и все его дети, включая `Exception`):

```
try:
    even_more_difficult_function()
except:
    pass  # do whatever needed
```

В большинстве случаев это плохая практика. Это может `SystemExit` больше, чем предполагалось, например `SystemExit` , `KeyboardInterrupt` и `MemoryError` - каждый из которых обычно должен обрабатываться иначе, чем обычные системные или логические ошибки. Это также означает, что нет четкого понимания того, что внутренний код может сделать неправильно и как правильно восстановиться из этого условия. Если вы поймаете каждую

ошибку, вы не будете знать, какая ошибка произошла или как ее исправить.

Это чаще всего называют «маскировкой ошибок», и его следует избегать. Пусть ваша программа вылетает из строя вместо того, чтобы тихо провалиться или даже хуже, не справиться на более глубоком уровне исполнения. (Представьте, что это транзакционная система)

Обычно эти конструкции используются на самом внешнем уровне программы и будут регистрировать детали ошибки, чтобы ошибка могла быть исправлена, или ошибка может быть обработана более конкретно.

Захват нескольких исключений

Существует несколько способов уловить несколько исключений .

Первый заключается в создании кортежа типов исключений, которые вы хотите поймать и обработать тем же способом. В этом примере код будет игнорировать исключения `KeyError` и `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.")
```

Если вы хотите обрабатывать различные исключения по-разному, вы можете предоставить отдельный блок исключений для каждого типа. В этом примере мы по-прежнему захватываем `KeyError` и `AttributeError` , но обрабатываем исключения в разных манерах.

```
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)
```

Практические примеры обработки исключений

Вход пользователя

Представьте, что вы хотите, чтобы пользователь вводил число через `input` . Вы хотите, чтобы вход был числом. Вы можете использовать `try` / `except` :

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.')
```

Примечание: Python 2.x вместо этого использует `raw_input` ; `input` функции существует в Python 2.x, но имеет другую семантику. В приведенном выше примере `input` также принимает выражения, такие как `2 + 2` которые вычисляют число.

Если вход не может быть преобразован в целое число, повышается значение `ValueError` . Вы можете поймать его `except` . Если никакое исключение не возбуждается, `break` выпрыгивает из цикла. После цикла `nb` содержит целое число.

Словари

Представьте, что вы повторяете список последовательных целых чисел, например `range(n)` , и у вас есть список словарей `d` , содержащий информацию о том, что нужно делать, когда вы сталкиваетесь с определенными целыми числами, скажем, *пропустите* `d[i]` *следующие* .

```
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
```

`KeyError` будет поднят, когда вы попытаетесь получить значение из словаря для ключа, которого не существует.

еще

Код в блоке else будет выполняться только в том случае, если код в блоке `try` восстановлен. Это полезно, если у вас есть код, который вы не хотите запускать, если генерируется исключение, но вы не хотите, чтобы исключения, брошенные этим кодом, были пойманы.

Например:

```
try:
    data = {1: 'one', 2: 'two'}
    print(data[1])
except KeyError as e:
    print('key not found')
```

```
else:
    raise ValueError()
# Output: one
# Output: ValueError
```

Обратите внимание, что этот вид `else:` нельзя комбинировать с `if` начиная с else-clause, в `elif`. Если у вас есть следующее , `if` это необходимо , чтобы остаться с отступом ниже, `else:` :

```
try:
    ...
except ...:
    ...
else:
    if ...:
        ...
    elif ...:
        ...
    else:
        ...
```

Прочитайте Исключения онлайн: https://riptutorial.com/ru/python/topic/1788/исключения

глава 77: Исключения из Содружества

Вступление

Здесь, в Stack Overflow, мы часто видим дубликаты, говорящие о тех же ошибках: `"ImportError: No module named '??????'` , `SyntaxError: invalid syntax` или `NameError: name '???' is not defined` . Это попытка уменьшить их и иметь ссылку на документацию.

Examples

ОтступыErrors (или отступы SyntaxErrors)

В большинстве других языков отступы не являются обязательными, но в Python (и на других языках: ранние версии FORTRAN, Makefiles, Whitespace (эзотерический язык) и т. Д.), Что не так, что может сбивать с толку, если вы пришли с другого языка, если вы копировали код из примера в свой собственный или просто, если вы новичок.

IndentationError / SyntaxError: неожиданный отступ

Это исключение возникает, когда уровень отступов увеличивается без причины.

пример

Нет причин для повышения уровня здесь:

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")
```

Здесь есть две ошибки: последняя и что отступ не соответствует уровню отступов. Однако показано только одно:

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 не соответствует внешнему уровню отступа

Похоже, ты не отказался полностью.

пример

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: ожидается отложенный блок

После двоеточия (а затем новой строки) уровень отступов должен увеличиваться. Эта ошибка возникает, когда этого не происходит.

пример

```
if ok:
doStuff()
```

Примечание . Используйте ключевое слово `pass` (что абсолютно ничего), чтобы просто положить `if` , `else` , `except` , `class` , `method` или `definition` но не сказать, что произойдет, если inv / condition истинно (но сделайте это позже или в случае `except` : просто ничего не

делать):

```
def foo():
    pass
```

IndentationError: непоследовательное использование вкладок и пробелов в отступе

пример

```
def foo():
    if ok:
      return "Two != Four != Tab"
        return "i dont care i do whatever i want"
```

Как избежать этой ошибки

Не используйте вкладки. Это обескураживает `PEP8` , руководство по стилю для Python.

1. Установите для вашего редактора четыре **пробела** для отступов.
2. Сделайте поиск и замените, чтобы заменить все вкладки на 4 пробела.
3. Убедитесь, что ваш редактор настроен на **отображение** вкладок в виде 8 пробелов, чтобы вы могли легко понять эту ошибку и исправить ее.

См. Этот вопрос, если вы хотите узнать больше.

TypeErrors

Эти исключения возникают, когда тип какого-либо объекта должен быть разным

ТипError: [определение / метод] занимает? позиционные аргументы, но? было дано

Функция или метод вызывались с более (или менее) аргументами, чем те, которые он может принять.

пример

Если даны дополнительные аргументы:

```
def foo(a): return a
foo(a,b,c,d) #And a,b,c,d are defined
```

Если даны меньше аргументов:

```
def foo(a,b,c,d): return a += b + c + d
foo(a) #And a is defined
```

Примечание . Если вы хотите использовать неизвестное количество аргументов, вы можете использовать `*args` или `**kwargs` . См. * Args и ** kwargs

---

TypeError: неподдерживаемый тип операндов для [операнда]: '???' а также '???'

Некоторые типы не могут работать вместе, в зависимости от операнда.

пример

Например: `+` используется для конкатенации и добавления, но вы не можете использовать их для обоих типов. Например, попытка создания `set` путем конкатенации (`+` ing) `'set1'` и `'tuple1'` дает ошибку. Код:

```
set1, tuple1 = {1,2}, (3,4)
a = set1 + tuple1
```

Некоторые типы (например: `int` и `string`) используют оба `+` но для разных вещей:

```
b = 400 + 'foo'
```

Или они могут быть даже не использованы ни для чего:

```
c = ["a","b"] - [1,2]
```

Но вы можете, например, добавить `float` в `int` :

```
d = 1 + 1.0
```

TypeError: '???' объект не является итерируемым / индексируемым:

Для того, чтобы объект был итерируемым, он может принимать последовательные индексы, начиная с нуля, до тех пор, пока индексы перестанут быть действительными, а `IndexError` будет поднят (более технически: он должен иметь метод `__iter__` который возвращает `__iterator__` или который определяет метод `__getitem__` который делает что было упомянуто ранее).

пример

Здесь мы говорим, что `bar` - это нулевой элемент 1. Чепуха:

```
foo = 1
bar = foo[0]
```

Это более дискретная версия: в этом примере `for` попытки установить `x` для `amount[0]` первый элемент в итерируемой, но он не может, поскольку сумма является int:

```
amount = 10
for x in amount: print(x)
```

TypeError: '???' объект не может быть вызван

Вы определяете переменную и вызываете ее позже (например, что вы делаете с помощью функции или метода)

пример

```
foo = "notAFunction"
foo()
```

NameError: name '???' не определено

Повышается, когда вы пытались использовать переменную, метод или функцию, которая

не инициализирована (по крайней мере, не раньше). Другими словами, он возникает, когда запрашиваемое локальное или глобальное имя не найдено. Возможно, вы пропустили имя объекта или забыли что-то `import` . Также, возможно, это в другой сфере. Мы рассмотрим их отдельными примерами.

Это просто не определено нигде в коде

Возможно, вы забыли инициализировать его, особенно если он является постоянным

```
foo   # This variable is not defined
bar() # This function is not defined
```

Возможно, это определено позже:

```
baz()

def baz():
    pass
```

Или это не было `import` :

```
#needs import math

def sqrt():
    x = float(input("Value: "))
    return math.sqrt(x)
```

Области Python и правило LEGB:

В так называемом правиле LEGB говорится о области Python. Это имя основано на разных областях, упорядоченных по соответствующим приоритетам:

```
Local → Enclosed → Global → Built-in.
```

- **L** ocal: Переменные, не объявленные глобальными или назначенные в функции.
- **E** nclosing: Переменные, определенные в функции, которая завернута в другую функцию.
- **G** lobal: переменные объявлены глобальными или назначаются на верхнем уровне файла.
- **B** uilt-in: Переменные, назначенные во встроенном модуле имен.

В качестве примера:

```
for i in range(4):
    d = i * 2
print(d)
```

`d` является доступным, потому что цикл `for` не отмечает новую область видимости, но если бы это произошло, у нас была бы ошибка, и ее поведение было бы похоже на:

```
def noaccess():
    for i in range(4):
        d = i * 2
noaccess()
print(d)
```

Python говорит `NameError: name 'd' is not defined`

Другие ошибки

AssertError

Утверждение `assert` существует почти на всех языках программирования. Когда вы выполните:

```
assert condition
```

или же:

```
assert condition, message
```

Это эквивалентно этому:

```
if __debug__:
    if not condition: raise AssertionError(message)
```

Утверждения могут включать необязательное сообщение, и вы можете отключить их, когда вы закончите отладку.

Примечание : встроенная переменная **debug** True при нормальных обстоятельствах, False при запросе оптимизации (опция командной строки -O). Задания для **отладки** являются незаконными. Значение встроенной переменной определяется при запуске интерпретатора.

KeyboardInterrupt

Ошибка при нажатии на клавишу прерывания, обычно `Ctrl` + `C` или `del` .

ZeroDivisionError

Вы пытались вычислить `1/0` который не определен. См. Этот пример, чтобы найти делители числа:

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)
```

Он вызывает `ZeroDivisionError` потому что цикл `for` присваивает этому значению значение `x` . Вместо этого это должно быть:

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)
```

Синтаксическая ошибка при хорошем коде

В большинстве случаев синтаксическая ошибка, указывающая на неинтересную строку, означает, что перед ней стоит проблема (в этом примере это недостающая скобка):

```
def my_print():
    x = (1 + 1
    print(x)
```

Возвращает

```
  File "<input>", line 3
    print(x)
        ^
```

```
SyntaxError: invalid syntax
```

Наиболее распространенная причина этой проблемы - несогласованные скобки / скобки, как показывает пример.

Существует одна серьезная оговорка для операторов печати в Python 3:

Python 3.x 3.0

```
 >>> print "hello world"
  File "<stdin>", line 1
    print "hello world"
                      ^
SyntaxError: invalid syntax
```

Поскольку оператор `print` был заменен функцией `print()` , вы хотите:

```
print("hello world")  # Note this is valid for both Py2 & Py3
```

Прочитайте Исключения из Содружества онлайн: https://riptutorial.com/ru/python/topic/9300/исключения-из-содружества

глава 78: Использование модуля «pip»: диспетчер пакетов PyPI

Вступление

Иногда вам может понадобиться использовать диспетчер пакетов pip внутри python, например. когда некоторый импорт может увеличить `ImportError` и вы хотите обработать исключение. Если вы распаковываете на Windows `Python_root/Scripts/pip.exe` внутри, то хранится файл `__main__.py` , где импортируется `main` класс из пакета `pip` . Это означает, что пакет pip используется всякий раз, когда вы используете исполняемый файл pip. Для использования pip в качестве исполняемого файла см .: pip: PyPI Package Manager

Синтаксис

- pip. <function | attribute | class>, где функция является одной из следующих:
  - автополный ()
    - Завершение команды и опции для основного парсера параметров (и опций) и его подкоманд (и параметров). Включить, используя один из сценариев оболочки завершения (bash, zsh или fish).
  - check_isolated (apr)
    - param args {list}
    - возвращает {boolean}
  - create_main_parser ()
    - возвращает {pip.baseparser.ConfigOptionParser object}
  - Основной (apr = нет)
    - param args {list}
    - возвращает {integer} Если не сработало, чем возвращает 0
  - parseopts (apr)
    - param args {list}
  - get_installed_distributions ()
    - возвращает {list}
  - get_similar_commands (имя)
    - Имя команды автокорректно.
    - param name {string}
    - возвращает {boolean}
  - get_summaries (упорядоченная = True)
    - Умножается сортировка (имя команды, сводка команд) кортежей.
  - get_prog ()
    - возвращает {строка}
  - dist_is_editable (расстояние)

        ◦ Является ли дистрибутив доступной для редактирования установкой?
        ◦ param dist {object}
        ◦ возвращает {boolean}
    ◦ commands_dict
        ◦ attribute {dictionary}

Examples

Пример использования команд

```
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])
```

Обязательны только необходимые параметры, поэтому возможны и `pip.main(['freeze'])` и `pip.main(['freeze', '', ''])` .

Пакетная установка

В одном вызове можно передавать много имен пакетов, но если одна установка / обновление не выполняется, весь процесс установки останавливается и заканчивается статусом «1».

```
import pip

installed = pip.get_installed_distributions()
list = []
for i in installed:
    list.append(i.key)

pip.main(['install']+list+['--upgrade'])
```

Если вы не хотите останавливаться при сбое некоторых установок, вызовите установку в цикле.

```
for i in installed:
        pip.main(['install']+i.key+['--upgrade'])
```

Обработка исключения ImportError

Когда вы используете файл python как модуль, нет необходимости всегда проверять, установлен ли пакет, но он по-прежнему полезен для скриптов.

```
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
```

Силовая установка

Многие пакеты, например, в версии 3.4, будут работать на 3.6 просто отлично, но если для конкретной платформы нет дистрибутивов, они не могут быть установлены, но есть обходное решение. В .whl файлах (называемых колесами) соглашение об именах решает, можно ли устанавливать пакет на указанной платформе. Например. `scikit_learn-0.18.1-cp36-cp36m-win_amd64.whl` [`scikit_learn-0.18.1-cp36-cp36m-win_amd64.whl`] - [версия] - [интерпретатор python] - [python-interpreter] - [Операционная система] .whl. Если имя файла колес изменено, поэтому платформа соответствует, pip пытается установить пакет, даже если версия платформы или python не соответствует. Удаление платформы или интерпретатора от имени приведет к ошибке в новейшей версии модуля pip `kjhfkjdf.whl is not a valid wheel filename.` ,

Альтернативный файл .whl может быть распакован с использованием архиватора как 7-zip. - Обычно он содержит папку метаданных и папку с исходными файлами. Эти исходные файлы могут быть просто распакованы в каталог `site-packges` если только это колесо не содержит сценарий установки, если это так, его нужно запустить первым.

Прочитайте Использование модуля «pip»: диспетчер пакетов PyPI онлайн: https://riptutorial.com/ru/python/topic/10730/использование-модуля--pip---диспетчер-пакетов-pypi

глава 79: Использование петель внутри функций

Вступление

Функция Python будет возвращена, как только выполнение будет выглядеть как «return».

Examples

Оператор возврата внутри цикла в функции

В этом примере функция вернется, как только значение var имеет 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])
```

выход

```
Got value 5
Still looping
Got value 3
Still looping
Got value 1
>>>> Got 1
```

Прочитайте Использование петель внутри функций онлайн:
https://riptutorial.com/ru/python/topic/10883/использование-петель-внутри-функций

глава 80: Итераторы и итераторы

Examples

Итератор против Iterable vs Generator

Итерируемый объект, который может возвращать **итератор** . Любой объект с состоянием, который имеет метод `__iter__` и возвращает итератор, является итерируемым. Он также может быть объектом *без* состояния, реализующим метод `__getitem__` . - Метод может принимать индексы (начиная с нуля) и поднимать `IndexError` когда индексы более недействительны.

Класс `str` класса Python является примером `__getitem__` iterable.

Итератор - это объект, который выдает следующее значение в последовательности, когда вы вызываете `next(*object*)` на какой-либо объект. Более того, любой объект с методом `__next__` является итератором. Итератор вызывает `StopIteration` после исчерпания итератора и *не может* быть повторно использован в этой точке.

Итерируемые классы:

`__iter__` Iterable определяет `__iter__` и `__next__` . Пример итеративного класса:

```
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())
```

Попытка создать экземпляр абстрактного класса из модуля `collections` чтобы лучше это видеть.

Пример:

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__
```

Управляйте совместимостью Python 3 для итерируемых классов в Python 2, выполняя следующие действия:

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
```

Оба они теперь итераторы и могут быть зациклены:

```
ex1 = MyIterableClass()
ex2 = MySequence()

for (item) in (ex1): #code
for (item) in (ex2): #code
```

Генераторы - это простые способы создания итераторов. Генератор *является* итератором, и итератор является итерируемым.

Что можно итерабельно

Iterable может быть любым, для которого элементы принимаются *один за другим, только вперед* . Встроенные коллекции Python являются итерабельными:

```
[1, 2, 3]     # list, iterate over items
(1, 2, 3)     # tuple
{1, 2, 3}     # set
{1: 2, 3: 4}  # dict, iterate over keys
```

Генераторы возвращают итерации:

```
def foo():  # foo isn't iterable yet...
    yield 1

res = foo()  # ...but res already is
```

Итерирование по всему итерируемому

```
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]
```

Проверить только один элемент в истребителе

Используйте распаковку для извлечения первого элемента и убедитесь, что он единственный:

```
a, = iterable

def foo():
    yield 1

a, = foo()  # a = 1

nums = [1, 2, 3]
a, = nums  # ValueError: too many values to unpack
```

Извлечение значений по одному

Начните с встроенного `iter()` чтобы получить **итератор** поверх iterable и использовать `next()` для получения элементов один за другим до `StopIteration` пор, пока `StopIteration` будет поднят, означающий конец:

```
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
```

Итератор не реентерабелен!

```
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()
```

Прочитайте Итераторы и итераторы онлайн: https://riptutorial.com/ru/python/topic/2343/итераторы-и-итераторы

глава 81: Классы

Вступление

Python предлагает себя не только как популярный язык сценариев, но также поддерживает парадигму объектно-ориентированного программирования. Классы описывают данные и предоставляют методы для управления этими данными, все они охватываются одним объектом. Кроме того, классы позволяют абстрагироваться путем отделения конкретных деталей реализации от абстрактных представлений данных.

Кодекс, использующий классы, обычно легче читать, понимать и поддерживать.

Examples

Основное наследование

Наследование в Python основано на аналогичных идеях, используемых в других объектно-ориентированных языках, таких как Java, C ++ и т. Д. Новый класс может быть получен из существующего класса следующим образом.

```
class BaseClass(object):
    pass

class DerivedClass(BaseClass):
    pass
```

`BaseClass` - уже существующий (*родительский*) класс, а `DerivedClass` - новый (*дочерний*) класс, который наследует (или *подклассы*) атрибуты из `BaseClass` . **Примечание** . Начиная с Python 2.2, все классы неявно наследуют от класса `object` , который является базовым классом для всех встроенных типов.

Мы определяем родительский класс `Rectangle` в приведенном ниже примере, который неявно наследует от `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)
```

Класс `Rectangle` может использоваться как базовый класс для определения `Square` класса,

так как квадрат является частным случаем прямоугольника.

```
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
```

Класс `Square` автоматически наследует все атрибуты класса `Rectangle` а также класс объекта. `super()` используется для вызова метода `__init__()` класса `Rectangle` , по существу вызывающего любой переопределенный метод базового класса. **Примечание** : в Python 3 `super()` не требует аргументов.

Объекты с производными классами могут получать и изменять атрибуты базовых классов:

```
r.area()
# Output: 12
r.perimeter()
# Output: 14

s.area()
# Output: 4
s.perimeter()
# Output: 8
```

Встроенные функции, которые работают с наследованием

`issubclass(DerivedClass, BaseClass)` : возвращает `True` если `DerivedClass` является подклассом `BaseClass`

`isinstance(s, Class)` : возвращает `True` если s является экземпляром `Class` или любого из производных классов `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
```

Переменные класса и экземпляра

Переменные экземпляра уникальны для каждого экземпляра, а переменные класса - для всех экземпляров.

```
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
```

Доступ к переменным класса можно получить в экземплярах этого класса, но присвоение атрибуту class создаст переменную экземпляра, которая затеняет переменную класса

```
c2.x = 4
c2.x
# 4
C.x
# 2
```

Обратите внимание, что *мутирующие* переменные класса из экземпляров могут привести к некоторым неожиданным последствиям.

```
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]
```

Связанные, несвязанные и статические методы

Идея связанных и несвязанных методов была удалена в Python 3 . В Python 3, когда вы объявляете метод внутри класса, вы используете ключевое слово `def` , создавая таким образом объект функции. Это регулярная функция, и окружающий класс работает как пространство имен. В следующем примере мы объявляем метод `f` в классе `A` , и он становится функцией `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)
```

В Python 2 поведение было иным: функциональные объекты внутри класса были неявно заменены объектами типа `instancemethod` , которые назывались *несвязанными методами,* потому что они не были привязаны к какому-либо конкретному экземпляру класса. Можно было получить доступ к основной функции, используя свойство `.__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 ...>
```

Последнее поведение подтверждается проверкой - методы распознаются как функции в Python 3, а различие поддерживается в 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
```

В обеих версиях функции / метода Python `Af` можно вызывать напрямую, при условии, что вы передаете экземпляр класса `A` в качестве первого аргумента.

```
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
```

Предположим теперь, что `a` является экземпляром класса `A` , что тогда `af` ? Ну, интуитивно это должен быть тот же метод `f` класса `A` , только он должен каким-то образом «знать», что он был применен к объекту `a` - в Python, это называется методом, *связанным* с `a` .

В суровых буднях детали следующим образом : запись `af` вызывает магический `__getattribute__` метод , который сначала проверяет , является ли имеет атрибут с именем `a` `a f` (не), а затем проверяет , класс `A` , содержит ли это метод с таким именем (он делает), и создает новый объект `m` типа `method` , который имеет ссылку на исходные `Af` в `m.__func__` и ссылку на объект `a` в `m.__self__` . Когда этот объект вызывается как функция, он просто выполняет следующее: `m(...) => m.__func__(m.__self__, ...)` . Таким образом, этот объект называется **связанным методом,** потому что при вызове он знает, что для обеспечения объекта он был связан как первый аргумент. (Эти вещи работают одинаково в Python 2 и 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
```

Наконец, Python имеет **методы класса** и **статические методы** - специальные виды методов. Методы класса работают так же, как и обычные методы, за исключением того, что при вызове на объект они привязываются к *классу* объекта, а не к объекту. Таким образом, `m.__self__ = type(a)` . Когда вы вызываете такой связанный метод, он передает класс `a` в качестве первого аргумента. Статические методы еще проще: они вообще ничего не связывают и просто возвращают базовую функцию без каких-либо преобразований.

```
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
```

Обратите внимание, что методы класса привязаны к классу даже при доступе к экземпляру:

```
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
```

Стоит отметить, что на самом низком уровне функции, методы, staticmethods и т. Д. Фактически являются дескрипторами, которые вызывают `__get__` , `__set__` __ и, необязательно, специальные методы `__del__` . Дополнительные сведения о методах класса и staticmethods:

- В чем разница между @staticmethod и @classmethod в Python?
- Значение @classmethod и @staticmethod для начинающих?

Классы нового стиля и старого стиля

Python 2.x 2.2.0

Классы *нового стиля* были введены в Python 2.2 для унификации *классов* и *типов* . Они наследуются от типа `object` верхнего уровня. *Класс нового стиля является определяемым пользователем типом* и очень похож на встроенные типы.

```
# 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
```

Классы *старого стиля* **не** наследуются от `object` . Старые экземпляры всегда реализуются со встроенным типом `instance` .

```
# 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

В Python 3 были удалены классы старого стиля.

Классы нового стиля в Python 3 неявно наследуют от `object` , поэтому больше не нужно указывать `MyClass(object)` .

```
class MyClass:
    pass

my_inst = MyClass()

type(my_inst)
# <class '__main__.MyClass'>
my_inst.__class__
# <class '__main__.MyClass'>
issubclass(MyClass, object)
# True
```

Значения по умолчанию для переменных экземпляра

Если переменная содержит значение неизменяемого типа (например, строку), тогда можно назначить значение по умолчанию, подобное этому

```
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
```

Нужно быть осторожным при инициализации изменяемых объектов, таких как списки в конструкторе. Рассмотрим следующий пример:

```
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
```

Такое поведение вызвано тем, что параметры Python по умолчанию привязаны к выполнению функции, а не к объявлению функции. Чтобы получить переменную экземпляра по умолчанию, которая не используется совместно с экземплярами, следует использовать такую конструкцию:

```
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
```

См. Также « Разрешимые аргументы по умолчанию» и «Наименьшее удивление» и параметр Mutable Default Argument .

Многократное наследование

Python использует алгоритм линеаризации C3 для определения порядка, в котором разрешаются атрибуты класса, включая методы. Это называется порядком разрешения

метода (MRO).

Вот простой пример:

```
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'
```

Теперь, если мы создаем FooBar, если мы найдем атрибут foo, мы увидим, что атрибут Foo найден первым

```
fb = FooBar()
```

а также

```
>>> fb.foo
'attr foo of Foo'
```

Вот MRO FooBar:

```
>>> FooBar.mro()
[<class '__main__.FooBar'>, <class '__main__.Foo'>, <class '__main__.Bar'>, <type 'object'>]
```

Можно просто сказать, что алгоритм MRO Python

1. Сначала глубина (например, `FooBar` затем `Foo`), если только
2. общий родитель (`object`) блокируется дочерним элементом (`Bar`) и
3. не допускаются круговые отношения.

То есть, например, Bar не может наследовать FooBar, а FooBar наследуется от Bar.

Подробный пример в Python см. В записи wikipedia .

Еще одна мощная функция в наследовании - `super` . super может извлекать функции родительских классов.

```
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()
```

Множественное наследование с методом init метода, когда каждый класс имеет собственный метод init, мы пытаемся выполнить множественный ineritance, тогда только метод init получает вызов класса, который наследуется первым.

для примера ниже только метод **инициализации** класса Foo, получивший вызов **Bar** class init, не получивший вызов

```
    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()
```

Выход:

```
    foobar init
    foo init
```

Но это не значит, что класс **Bar** не наследуется. Экземпляр конечного класса **FooBar** также является экземпляром класса **Bar и** класса **Foo** .

```
print isinstance(a,FooBar)
print isinstance(a,Foo)
print isinstance(a,Bar)
```

Выход:

```
True
True
True
```

Дескрипторы и пунктирные образы

Дескрипторы - это объекты, которые являются (обычно) атрибутами классов и имеют какие-либо из специальных методов `__get__` , `__set__` или `__delete__` .

Дескрипторы данных имеют любой из `__set__` или `__delete__`

Они могут контролировать точечный поиск экземпляра и использоваться для реализации функций, `staticmethod` , `classmethod` и `property` . Точечный поиск (например, экземпляр `foo` класса `Foo` поиска атрибута `bar` то есть `foo.bar`) использует следующий алгоритм:

1. `bar` просматривается в классе, `Foo` . Если он есть, и это **дескриптор данных** , то используется дескриптор данных. Вот как `property` может контролировать доступ к данным в экземпляре, и экземпляры не могут переопределить это. Если **дескриптора данных** нет, тогда

2. `bar` просматривается в экземпляре `__dict__` . Вот почему мы можем переопределять или блокировать методы, вызываемые из экземпляра с точным поиском. Если в экземпляре существует `bar` , он используется. Если нет, мы тогда

3. посмотрите в классе `Foo` для `bar` . Если это **дескриптор** , используется протокол дескриптора. Вот как реализованы функции (в этом контексте, несвязанные методы), `classmethod` и `staticmethod` . Иначе он просто возвращает объект там, или есть `AttributeError`

Методы класса: альтернативные инициализаторы

Методы класса представляют альтернативные способы создания экземпляров классов. Чтобы проиллюстрировать это, давайте рассмотрим пример.

Предположим, что у нас относительно простой класс `Person` :

```
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 + ".")
```

Возможно, было бы удобно создать экземпляры этого класса, указав полное имя вместо имени и фамилии отдельно. Один из способов сделать это `last_name` бы в том, чтобы `last_name` был необязательным параметром и считал, что если он не указан, мы передали полное имя в:

```
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 + ".")
```

Однако в этом бите кода есть две основные проблемы:

1. Параметры `first_name` и `last_name` теперь вводят в заблуждение, так как вы можете ввести полное имя для `first_name` . Кроме того, если есть больше случаев и / или более параметров, обладающих такой гибкостью, ветвление if / elif / else может раздражать быстро.

2. Не так важно, но все же стоит указать: что, если `last_name` - `None` , но `first_name` не разбивается на две или более вещи через пробелы? У нас есть еще один уровень проверки ввода и / или обработки исключений ...

Введите методы класса. Вместо того, чтобы иметь один инициализатор, мы создадим отдельный инициализатор, называемый `from_full_name` , и `from_full_name` его (встроенным) декоратором `classmethod` .

```
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 + ".")
```

Обратите внимание на `cls` вместо `self` в качестве первого аргумента `from_full_name` . Методы класса применяется к общему классу, *не* является экземпляром данного класса (что `self` обычно обозначает). Итак, если `cls` - это наш класс `Person` , тогда возвращаемое значение из `from_full_name` класса `from_full_name` - `Person(first_name, last_name, age)` , которое использует `__init__` `Person` для создания экземпляра класса `Person` . В частности, если мы должны были создать подкласс `Employee` of `Person` , то `from_full_name` будет работать и в классе `Employee` .

Чтобы показать, что это работает, как ожидается, давайте создадим экземпляры `Person` более чем одним способом без ветвления в `__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.
```

Другие ссылки:

- Python @classmethod и @staticmethod для начинающих?
- https://docs.python.org/2/library/functions.html#classmethod
- https://docs.python.org/3.5/library/functions.html#classmethod

Состав класса

Композиция класса допускает явные отношения между объектами. В этом примере люди живут в городах, принадлежащих странам. Композиция позволяет людям получить доступ к числу людей, проживающих в их стране:

```
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
```

Патч обезьяны

В этом случае «патч обезьяны» означает добавление новой переменной или метода в класс после того, как он был определен. Например, предположим, что мы определили класс `A` как

```
class A(object):
    def __init__(self, num):
        self.num = num

    def __add__(self, other):
        return A(self.num + other.num)
```

Но теперь мы хотим добавить еще одну функцию позже в код. Предположим, что эта функция такова.

```
def get_num(self):
    return self.num
```

Но как мы можем добавить это как метод в `A` ? Это просто, мы просто по существу помещаем эту функцию в `A` с оператором присваивания.

```
A.get_num = get_num
```

Почему это работает? Поскольку функции являются объектами, как и любой другой объект, а методы - это функции, принадлежащие классу.

Функция `get_num` должна быть доступна всем существующим (уже созданным), а также новым экземплярам `A`

Эти дополнения доступны для всех экземпляров этого класса (или его подклассов) автоматически. Например:

```
foo = A(42)

A.get_num = get_num

bar = A(6);

foo.get_num() # 42
```

```
bar.get_num() # 6
```

Обратите внимание, что в отличие от некоторых других языков этот метод не работает для определенных встроенных типов, и он не считается хорошим стилем.

Список всех участников класса

Функция `dir()` может использоваться для получения списка членов класса:

```
dir(Class)
```

Например:

```
>>> 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']
```

Общепринято смотреть только на «не магических» членов. Это можно сделать, используя простое понимание, в котором перечислены члены с именами, не начинающимися с `__` :

```
>>> [m for m in dir(list) if not m.startswith('__')]
['append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse',
'sort']
```

Предостережения:

Классы могут определять `__dir__()` . Если этот метод существует, вызов `dir()` вызовет `__dir__()` , в противном случае Python попытается создать список членов класса. Это означает, что функция dir может иметь неожиданные результаты. Две цитаты, имеющие важное значение из официальной документации python :

> Если объект не предоставляет **dir** (), функция пытается собрать информацию из атрибута **dict** объекта, если он определен, и из его объекта типа. Полученный список не обязательно завершен и может быть неточным, если у объекта есть пользовательский **getattr** ().
>
> **Примечание.** Поскольку dir () предоставляется в основном как удобство для использования в интерактивном приглашении, он пытается предоставить интересный набор имен больше, чем пытается предоставить строго или последовательно определенный набор имен, а его подробное поведение может релизы. Например, атрибуты metaclass не входят в список результатов, когда аргумент является классом.

Введение в классы

Класс функционирует как шаблон, определяющий основные характеристики конкретного объекта. Вот пример:

```
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))
```

При взгляде на приведенный выше пример есть несколько вещей.

1. Класс состоит из *атрибутов* (данных) и *методов* (функций).
2. Атрибуты и методы просто определяются как обычные переменные и функции.
3. Как отмечено в соответствующей docstring, метод `__init__()` называется *инициализатором* . Это эквивалентно конструктору на других объектно-ориентированных языках, и это метод, который сначала запускается при создании нового объекта или нового экземпляра класса.
4. Атрибуты, которые относятся ко всему классу, определяются сначала и называются *атрибутами класса* .
5. Атрибуты, относящиеся к конкретному экземпляру класса (объекта), называются *атрибутами экземпляра* . Они обычно определяются внутри `__init__()` ; это необязательно, но рекомендуется (поскольку атрибуты, определенные за пределами `__init__()` подвергаются риску доступа к ним до их определения).
6. Каждый метод, включенный в определение класса, передает этот объект в качестве его первого параметра. Слово « `self` используется для этого параметра (использование « `self` на самом деле условно, так как слово « `self` не имеет неотъемлемого значения в Python, но это одна из самых уважаемых конвенций Python, и вы всегда должны следовать за ней).
7. Те, которые используются для объектно-ориентированного программирования на других языках, могут быть удивлены несколькими вещами. Во-первых, у Python нет реальной концепции `private` элементов, поэтому все, по умолчанию, имитирует

поведение `public` ключевого слова C ++ / Java. Для получения дополнительной информации см. Пример «Члены частного класса» на этой странице.
8. Некоторые из методов класса имеют следующий вид: `__functionname__(self, other_stuff)` . Все такие методы называются «магическими методами» и являются важной частью классов в Python. Например, перегрузка операторов в Python осуществляется с помощью магических методов. Для получения дополнительной информации см. Соответствующую документацию .

Теперь давайте сделаем несколько экземпляров нашего класса `Person` !

```
>>> # Instances
>>> kelly = Person("Kelly")
>>> joseph = Person("Joseph")
>>> john_doe = Person("John Doe")
```

В настоящее время у нас есть три объекта `Person` , `kelly` , `joseph` и `john_doe` .

Мы можем получить доступ к атрибутам класса из каждого экземпляра, используя оператор точки `.` Еще раз обратите внимание на разницу между атрибутами класса и экземпляра:

```
>>> # Attributes
>>> kelly.species
'Homo Sapiens'
>>> john_doe.species
'Homo Sapiens'
>>> joseph.species
'Homo Sapiens'
>>> kelly.name
'Kelly'
>>> joseph.name
'Joseph'
```

Мы можем выполнять методы класса с использованием одного и того же точечного оператора `.` :

```
>>> # Methods
>>> john_doe.__str__()
'John Doe'
>>>  print(john_doe)
'John Doe'
>>>  john_doe.rename("John")
'Now my name is John'
```

свойства

Классы Python поддерживают **свойства** , которые выглядят как обычные объектные переменные, но с возможностью установки пользовательского поведения и документации.

```
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
```

Объект класса `MyClass` , *похоже* , имеет свойство `.string` , однако его поведение теперь жестко контролируется:

```
mc = MyClass()
mc.string = "String!"
print(mc.string)
del mc.string
```

Помимо полезного синтаксиса, как указано выше, синтаксис свойств позволяет проверять или добавлять другие атрибуты к этим атрибутам. Это может быть особенно полезно для общедоступных API-интерфейсов, где пользователю должен быть предоставлен уровень помощи.

Другим распространенным использованием свойств является включение класса для представления «виртуальных атрибутов» - атрибутов, которые фактически не хранятся, а вычисляются только по запросу.

```
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
```

Класс Singleton

Синглтон - это шаблон, который ограничивает экземпляр класса одним экземпляром / объектом. Для получения дополнительной информации о шаблонах проектирования синглтона python см. Здесь .

```
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
```

Другой способ - украсить ваш класс. Следуя примеру из этого ответа, создайте класс 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)
```

Для использования вы можете использовать метод `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
```

Прочитайте Классы онлайн: https://riptutorial.com/ru/python/topic/419/классы

глава 82: Кодовые блоки, кадры выполнения и пространства имен

Вступление

Блок кода представляет собой часть текста программы Python, которая может быть выполнена как единое целое, например модуль, определение класса или тело функции. Некоторые блоки кода (например, модули) обычно выполняются только один раз, другие (например, тела функций) могут выполняться много раз. Блоки кода могут содержать текстовые блоки других кодов. Блоки кода могут вызывать другие блоки кода (которые могут или не могут содержаться в них в тексте) как часть их выполнения, например, путем вызова (вызова) функции.

Examples

Пространства имен блоков кода

Тип блока кода	Глобальное пространство имен	Местное пространство имен
модуль	ns для модуля	как и глобальные
Сценарий (файл или команда)	ns для `__main__`	как и глобальные
Интерактивная команда	ns для `__main__`	как и глобальные
Определение класса	глобальные ns содержащего блока	новое пространство имен
Тело функции	глобальные ns содержащего блока	новое пространство имен
Строка передана в оператор `exec`	глобальные ns содержащего блока	локальное пространство имен содержащего блока
Строка передана `eval()`	глобальный номер вызывающего абонента	локальный номер вызывающего абонента
Файл, прочитанный `execfile()`	глобальный номер вызывающего абонента	локальный номер вызывающего абонента

Тип блока кода	Глобальное пространство имен	Местное пространство имен
Выражение, прочитанное `input()`	глобальный номер вызывающего абонента	локальный номер вызывающего абонента

Прочитайте Кодовые блоки, кадры выполнения и пространства имен онлайн: https://riptutorial.com/ru/python/topic/10741/кодовые-блоки--кадры-выполнения-и-пространства-имен

глава 83: колба

Вступление

Flask - это веб-инфраструктура Python, используемая для управления основными веб-сайтами, включая Pintrest, Twilio и Linkedin. В этом разделе объясняется и демонстрируется разнообразие функций Flask для разработки веб-сайтов как на передней, так и на задней панели.

Синтаксис

- @ app.route ("/ urlpath", methods = ["GET", "POST", "DELETE", "PUTS", "HEAD", "OPTIONS"])
- @ app.route ("/ urlpath / <param>", methods = ["GET", "POST", "DELETE", "PUTS", "HEAD", "OPTIONS"])

Examples

Основы

Следующий пример - пример базового сервера:

```
# 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()
```

Запуск этого сценария (при всех установленных зависимостях) должен запустить локальный сервер. Хост - `127.0.0.1` обычно известный как **localhost** . Этот сервер по умолчанию работает на порту **5000** . Чтобы получить доступ к веб-серверу, откройте веб-браузер и введите URL-адрес `localhost:5000` или `127.0.0.1:5000` (без разницы). В настоящее время только ваш компьютер может получить доступ к веб-серверу.

`app.run()` имеет три параметра: **хост** , **порт** и **отладка** . Хост по умолчанию `127.0.0.1` , но

установка этого `0.0.0.0` сделает ваш веб-сервер доступным с любого устройства в вашей сети, используя ваш частный IP-адрес в URL-адресе. порт по умолчанию 5000, но если параметр установлен на порт `80` , пользователям не нужно указывать номер порта, так как браузер использует порт 80 по умолчанию. Что касается опции отладки, то в процессе разработки (никогда в производстве) это помогает установить этот параметр в значение «Истина», так как ваш сервер будет перезагружен при внесении изменений в проект Flask.

```
if __name__ == "__main__":
    app.run(host="0.0.0.0", port=80, debug=True)
```

Маршрутизация URL-адресов

С Flask маршрутизация URL-адресов традиционно выполняется с помощью декораторов. Эти декораторы могут использоваться для статической маршрутизации, а также для маршрутизации URL-адресов с параметрами. В следующем примере представьте, что этот сценарий Flask работает на веб-сайте `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"
```

С помощью этого последнего маршрута вы можете видеть, что с учетом URL-адреса с / users / и именем профиля мы можем вернуть профиль. Поскольку было бы ужасно неэффективно и беспорядочно включать `@app.route()` для каждого пользователя, Flask предлагает взять параметры из URL-адреса:

```
@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
```

Методы HTTP

Двумя наиболее распространенными методами HTTP являются **GET** и **POST** . Флажок может запускать другой код с одного и того же URL-адреса в зависимости от

используемого метода HTTP. Например, в веб-службе с учетными записями наиболее удобно маршрутизировать страницу входа и вход в систему с помощью одного и того же URL-адреса. Запрос GET, то же самое, что и при открытии URL-адреса в вашем браузере, должен содержать форму входа в систему, а запрос POST (с регистрационными данными) должен обрабатываться отдельно. Маршрут также создается для обработки метода DELETE и 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"
```

Чтобы немного упростить код, мы можем импортировать пакет `request` из фляжки.

```
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"
```

Чтобы получить данные из запроса POST, мы должны использовать пакет `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"]
```

Файлы и шаблоны

Вместо того, чтобы вводить нашу разметку HTML в операторы return, мы можем использовать `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)
```

Это будет использовать наш файл шаблона `about-us.html` . Чтобы наше приложение могло найти этот файл, мы должны организовать наш каталог в следующем формате:

```
- application.py
/templates
    - about-us.html
    - login-form.html
/static
    /styles
        - about-style.css
        - login-style.css
    /scripts
        - about-script.js
        - login-script.js
```

Самое главное, ссылки на эти файлы в HTML должны выглядеть так:

```
<link rel="stylesheet" type="text/css", href="{{url_for('static', filename='styles/about-
style.css')}}">
```

который направит приложение искать `about-style.css` в папке стилей под статической папкой. Тот же формат пути применяется ко всем ссылкам на изображения, стили, сценарии или файлы.

Jinja Templating

Подобно Meteor.js, Flask хорошо сочетается с услугами по настройке интерфейсов. Flask использует по умолчанию Jinja Templating. Шаблоны позволяют использовать небольшие фрагменты кода в файле HTML, такие как условные обозначения или циклы.

Когда мы создаем шаблон, любые параметры за пределами имени файла шаблона передаются в службу шаблонов HTML. Следующий маршрут будет передавать имя пользователя и дату присоединения (от функции в другом месте) в 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)
```

Когда этот шаблон визуализируется, он может использовать переменные, переданные ему из функции `render_template()` . Вот содержимое `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>
```

Для разных интерпретаций используются следующие разделители:

- `{% ... %}` обозначает оператор
- `{{ ... }}` обозначает выражение, в котором выводится шаблон
- `{# ... #}` обозначает комментарий (не входит в вывод шаблона)
- `{# ... ##` подразумевает, что остальная часть строки должна интерпретироваться как оператор

Объект запроса

Объект `request` предоставляет информацию о запросе, который был отправлен на маршрут. Чтобы использовать этот объект, он должен быть импортирован из модуля фляжки:

```
from flask import request
```

Параметры URL

В предыдущих примерах использовались `request.method` и `request.form` , однако мы также можем использовать свойство `request.args` для получения словаря ключей / значений в параметрах 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"
```

Чтобы правильно аутентифицироваться в этом контексте, понадобится следующий URL (заменив имя пользователя любым именем пользователя:

`www.example.com/api/users/guido-van-rossum?key=pa55w0Rd`

Загрузка файлов

Если загрузка файла была частью представленной формы в запросе POST, файлы можно обрабатывать с использованием объекта `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
```

Печенье

Запрос также может включать файлы cookie в словаре, аналогичном параметрам 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")
```

Прочитайте колба онлайн: https://riptutorial.com/ru/python/topic/8682/колба

глава 84: Комментарии и документация

Синтаксис

- # Это однострочный комментарий
- print ("") # Это встроенный комментарий
- «»»
  Это
  многострочный комментарий
  «»»

замечания

Разработчики должны следовать руководящим принципам PEP257 - Docstring . В некоторых случаях руководства по стилям (например, руководство по стилю Google Style) или документация, предоставляющая сторонние стороны (например, Sphinx), могут содержать дополнительные соглашения для докстрон.

Examples

Однострочные, встроенные и многострочные комментарии

Комментарии используются для объяснения кода, когда сам базовый код не ясен.

Python игнорирует комментарии и поэтому не выполняет код там, или повышает синтаксические ошибки для простых английских предложений.

Однострочные комментарии начинаются с символа хеша (`#`) и заканчиваются концом строки.

- Комментарий одиночной строки:

```
# This is a single line comment in Python
```

- Встроенный комментарий:

```
print("Hello World")  # This line prints "Hello World"
```

- Комментарии, охватывающие несколько строк, имеют `"""` или `'''` на обоих концах. Это то же самое, что и многострочная строка, но они могут использоваться как комментарии:

```
"""
This type of comment spans multiple lines.
These are mostly used for documentation of functions, classes and modules.
"""
```

Программный доступ к docstrings

Docstrings - в отличие от обычных комментариев - сохраняются как атрибут функции, которую они документируют, что означает, что вы можете получить к ним доступ программно.

Примерная функция

```
def func():
    """This is a function that does nothing at all"""
    return
```

Доступ к docstring можно получить с `__doc__` атрибута `__doc__` :

```
print(func.__doc__)
```

> Это функция, которая ничего не делает

```
help(func)
```

> Справка по функции `func` в модуле `__main__` :
>
> `func()`
>
> Это функция, которая ничего не делает

Другая примерная функция

`function.__doc__` - это только фактическая docstring как строка, а `help` функция предоставляет общую информацию о функции, включая docstring. Вот более полезный пример:

```
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)
```

> Справка по функции `greet` в модуле `__main__` :

```
greet(name, greeting='Hello')
```

Печать приветствие пользователя `name`
Дополнительное `greeting` параметра может изменить то, с чем они приветствуются.

Преимущества docstrings над регулярными комментариями

Просто не вводить никакую docstring или регулярный комментарий в функции делает ее намного менее полезной.

```
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__)
```

Никто

```
help(greet)
```

Справка по функции приветствия в модуле **main** :

```
greet(name, greeting='Hello')
```

Напишите документацию с помощью docstrings

Docstring - многострочный комментарий, используемый для документирования модулей, классов, функций и методов. Он должен быть первым выражением описываемого компонента.

```
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.
    """
```

Значение docstring может быть доступно в программе и, например, используется командой `help` .

Соглашения о синтаксисе

PEP 257

PEP 257 определяет стандарт синтаксиса для комментариев docstring. Он в основном позволяет использовать два типа:

- Однострочные Docstrings:

Согласно PEP 257, они должны использоваться с короткими и простыми функциями. Все помещается в одну строку, например:

```
def hello():
    """Say hello to your friends."""
    print("Hello my friends!")
```

Докштрина заканчивается периодом, глагол должен быть в императивной форме.

- Многострочные Docstrings:

Многострочная docstring должна использоваться для более длинных, более сложных функций, модулей или классов.

```
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)
```

Они начинаются с краткой сводки (эквивалентной содержимому однострочной docstring), которая может быть в той же строке, что и кавычки или на следующей строке, дает дополнительные детали, параметры списка и возвращаемые значения.

Примечание. PEP 257 определяет, какая информация должна быть указана в docstring, она не определяет, в каком формате она должна быть указана. Именно по этой причине другие стороны и инструменты синтаксического анализа документации указали свои собственные стандарты для документации, некоторые из которых перечислены ниже и в этом вопросе .

сфинкс

Sphinx - это инструмент для создания документации на основе HTML для проектов Python на основе docstrings. Используемый язык разметки - reStructuredText . Они определяют свои собственные стандарты для документации, у pythonhosted.org есть очень хорошее описание . Формат Sphinx, например, используется IDE pyCharm .

Функция будет задокументирована так, используя формат 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
```

Руководство по стилю Google Python

Google опубликовал руководство по стилю Google Python, которое определяет соглашения о кодировании для Python, включая комментарии к документации. По сравнению с Sphinx / reST многие говорят, что документация в соответствии с рекомендациями Google лучше читается человеком.

На упомянутой выше странице pythonhosted.org также приводятся некоторые примеры хорошей документации в соответствии с Руководством по стилю Google.

Используя плагин Napoleon , Sphinx также может анализировать документацию в формате, соответствующем Руководству по стилю Google.

Функция будет документирована так, как это описано в формате Руководства по стилю 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
```

Прочитайте Комментарии и документация онлайн: https://riptutorial.com/ru/python/topic/4144/

комментарии-и-документация

глава 85: Контекстные менеджеры ("with" Statement)

Вступление

Хотя контекстные менеджеры Python широко используются, мало кто понимает цель их использования. Эти операторы, обычно используемые при чтении и записи файлов, помогают приложению сохранять память системы и улучшать управление ресурсами, гарантируя, что определенные ресурсы используются только для определенных процессов. В этом разделе объясняется и демонстрируется использование контекстных менеджеров Python.

Синтаксис

- с «context_manager» (как «псевдоним») (, «context_manager» (как «псевдоним»)?) *:

замечания

Контекстные менеджеры определены в PEP 343 . Они предназначены для использования в качестве более сжатого механизма управления ресурсами, чем `try ... finally` конструирует. Формальное определение состоит в следующем.

> В этом PEP менеджеры контекста предоставляют `__enter__()` и `__exit__()` , которые вызывают при входе в тело оператора exit и выходят из него.

Затем далее определяется оператор `with` следующим образом.

```
with EXPR as VAR:
    BLOCK
```

Перевод вышеуказанного заявления:

```
    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

Введение в контекстные менеджеры и оператор with

Диспетчер контекста - это объект, который уведомляется, когда *начинается* и *заканчивается* контекст (блок кода). Вы обычно используете один `with` инструкцией с. Он заботится об уведомлении.

Например, файловые объекты являются менеджерами контекста. Когда контекст заканчивается, объект файла закрывается автоматически:

```
open_file = open(filename)
with open_file:
    file_contents = open_file.read()

# the open_file object has automatically been closed.
```

Вышеприведенный пример обычно упрощается с помощью ключевого слова `as` :

```
with open(filename) as open_file:
    file_contents = open_file.read()

# the open_file object has automatically been closed.
```

Все, что заканчивает выполнение блока, вызывает вызов метода диспетчера контекста. Это включает исключения и может быть полезно, когда ошибка приводит к преждевременному выходу из открытого файла или соединения. Выход из сценария без должного закрытия файлов / соединений - плохая идея, которая может привести к потере данных или другим проблемам. Используя диспетчер контекста, вы можете гарантировать, что меры предосторожности всегда принимаются, чтобы предотвратить повреждение или потерю таким образом. Эта функция была добавлена в Python 2.5.

Назначение целевой

Многие менеджеры контекста возвращают объект при вводе. Вы можете назначить этот объект на новое имя в `with` заявлением.

Например, использование соединения с базой данных в операторе `with` может дать вам объект-курсор:

```
with database_connection as cursor:
```

```
    cursor.execute(sql_query)
```

Объекты файлов возвращаются сами собой, это позволяет как открывать объект файла, так и использовать его в качестве менеджера контекста в одном выражении:

```
with open(filename) as open_file:
    file_contents = open_file.read()
```

Написание собственного менеджера контекста

Диспетчер контекста - это любой объект, который реализует два магических метода `__enter__()` и `__exit__()` (хотя он может также реализовать другие методы):

```
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
```

Если контекст выходит с исключением, информация о том , что исключение будет передан как тройной `exc_type` , `exc_value` , `traceback` (это те же переменные , как возвращаемое `sys.exc_info()` функции). Если контекст завершается нормально, все три из этих аргументов будут `None` .

Если возникает исключение и передается методу `__exit__` , метод может возвращать `True` , чтобы подавить исключение, или исключение будет повторно поднято в конце функции `__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
```

Обратите внимание, что во втором примере, даже если исключение встречается в середине тела оператора-оператора, обработчик `__exit__` все равно выполняется, прежде чем исключение распространится во внешнюю область.

Если вам нужен только `__exit__` , вы можете вернуть экземпляр менеджера контекста:

```
class MyContextManager:
    def __enter__(self):
        return self

    def __exit__(self):
        print('something')
```

Написание собственного контекстного менеджера с использованием синтаксиса генератора

Также можно написать менеджер контекста, используя синтаксис генератора, благодаря декоратору `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))
```

производит:

```
Enter
Right in the middle with cm = 3
Exit
```

Декоратор упрощает задачу написания диспетчера контекстов путем преобразования генератора в один. Все до выражения yield становится методом `__enter__` , `__enter__` значение становится значением, возвращаемым генератором (которое может быть привязано к переменной в инструкции with), и все после выражения yield становится методом `__exit__` .

Если исключение должно обрабатываться менеджером контекста, в `try..except..finally` может быть записано `try..except..finally` и любое исключение, созданное в блоке `with` -block, будет обрабатываться этим блоком исключений.

```
@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)
```

Это дает:

```
Enter
Dividing by cm = 0
Caught error
Cleaning up
Exit
```

Несколько менеджеров контекста

Вы можете одновременно открыть несколько менеджеров контента:

```
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')
```

Он имеет тот же эффект, что и вложенные контекстные менеджеры:

```
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')
```

Управление ресурсами

```
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()
```

Метод `__init__()` устанавливает объект, в этом случае настраивая имя файла и режим для открытия файла. `__enter__()` открывает и возвращает файл, а `__exit__()` просто закрывает

его.

Используя эти магические методы (`__enter__` , `__exit__`) позволяет реализовать объекты , которые могут быть легко использованы `with` с утверждением.

Используйте класс File:

```
for _ in range(10000):
    with File('foo.txt', 'w') as f:
        f.write('foo')
```

Прочитайте Контекстные менеджеры ("with" Statement) онлайн:
https://riptutorial.com/ru/python/topic/928/контекстные-менеджеры---with--statement-

глава 86: Копирование данных

Examples

Выполнение мелкой копии

Неглубокая копия является копией коллекции без выполнения копии ее элементов.

```
>>> import copy
>>> c = [[1,2]]
>>> d = copy.copy(c)
>>> c is d
False
>>> c[0] is d[0]
True
```

Выполнение глубокой копии

Если у вас есть вложенные списки, желательно также клонировать вложенные списки. Это действие называется глубокой копией.

```
>>> import copy
>>> c = [[1,2]]
>>> d = copy.deepcopy(c)
>>> c is d
False
>>> c[0] is d[0]
False
```

Выполнение мелкой копии списка

Вы можете создавать мелкие копии списков, используя срезы.

```
>>> l1 = [1,2,3]
>>> l2 = l1[:]     # Perform the shallow copy.
>>> l2
[1,2,3]
>>> l1 is l2
False
```

Скопировать словарь

Объект словаря имеет `copy` метода. Он выполняет мелкую копию словаря.

```
>>> d1 = {1:[]}
>>> d2 = d1.copy()
>>> d1 is d2
False
```

```
>>> d1[1] is d2[1]
True
```

Скопируйте набор

Наборы также имеют метод `copy` . Вы можете использовать этот метод для выполнения мелкой копии.

```
>>> s1 = {()}
>>> s2 = s1.copy()
>>> s1 is s2
False
>>> s2.add(3)
>>> s1
{[]}
>>> s2
{3,[]}
```

Прочитайте Копирование данных онлайн: https://riptutorial.com/ru/python/topic/920/копирование-данных

глава 87: Кортеж

Вступление

Кортеж - это неизменный список значений. Кортежи являются одним из самых простых и наиболее распространенных типов коллекций Python и могут быть созданы с помощью оператора запятой (`value = 1, 2, 3`).

Синтаксис

- (1, a, "hello") # a должна быть переменной
- () # пустой кортеж
- (1,) # 1-элементный кортеж. `(1)` не является кортежем.
- 1, 2, 3 # 3-элементный кортеж (1, 2, 3)

замечания

Скобки нужны только для пустых кортежей или при использовании в вызове функции.

Кортеж представляет собой последовательность значений. Значения могут быть любого типа, и они индексируются целыми числами, поэтому в этом случае кортежи очень похожи на списки. Важным отличием является то, что кортежи неизменяемы и хешируются, поэтому их можно использовать в наборах и картах

Examples

Индексирующие кортежи

```
x = (1, 2, 3)
x[0]  # 1
x[1]  # 2
x[2]  # 3
x[3]  # IndexError: tuple index out of range
```

Индексирование с отрицательными номерами начинается с последнего элемента как -1:

```
x[-1] # 3
x[-2] # 2
x[-3] # 1
x[-4] # IndexError: tuple index out of range
```

Индексирование диапазона элементов

```
print(x[:-1])   # (1, 2)
print(x[-1:])   # (3,)
print(x[1:3])   # (2, 3)
```

Кортежи неизменяемы

Одно из главных различий между `list` s и `tuple` в Python состоит в том, что кортежи неизменяемы, то есть нельзя добавлять или изменять элементы после инициализации кортежа. Например:

```
>>> 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
```

Аналогично, кортежи не имеют `.append` и `.extend` в `list` . Использование `+=` возможно, но оно меняет привязку переменной, а не сам кортеж:

```
>>> t = (1, 2)
>>> q = t
>>> t += (3, 4)
>>> t
(1, 2, 3, 4)
>>> q
(1, 2)
```

Будьте осторожны при размещении изменяемых объектов, таких как `lists` , внутри кортежей. Это может привести к очень запутывающим результатам при их изменении. Например:

```
>>> t = (1, 2, 3, [1, 2, 3])
(1, 2, 3, [1, 2, 3])
>>> t[3] += [4, 5]
```

Оба повысят ошибку и изменят содержимое списка в кортеже:

```
TypeError: 'tuple' object does not support item assignment
>>> t
(1, 2, 3, [1, 2, 3, 4, 5])
```

Вы можете использовать оператор `+=` для добавления к кортежу - это работает, создавая новый кортеж с новым добавленным вами элементом и присваивая его текущей переменной; старый кортеж не изменен, а заменен!

Это позволяет избежать конвертирования в список и из него, но это медленный процесс и является плохой практикой, особенно если вы собираетесь добавлять несколько раз.

Кортеж - элементно-хешируемый и равносильный

```
hash( (1, 2) )  # ok
hash( ([], {"hello"})  # not ok, since lists and sets are not hashabe
```

Таким образом, кортеж может быть помещен внутри `set` или в ключ в `dict` только если каждый из его элементов может.

```
{ (1, 2) } #  ok
{ ([], {"hello"}) ) # not ok
```

Кортеж

Синтаксически кортеж представляет собой список значений, разделенных запятыми:

```
t = 'a', 'b', 'c', 'd', 'e'
```

Хотя это и не обязательно, обычно заключают кортежи в круглые скобки:

```
t = ('a', 'b', 'c', 'd', 'e')
```

Создайте пустой кортеж с круглыми скобками:

```
t0 = ()
type(t0)            # <type 'tuple'>
```

Чтобы создать кортеж с одним элементом, вы должны включить конечную запятую:

```
t1 = 'a',
type(t1)              # <type 'tuple'>
```

Обратите внимание, что одно значение в круглых скобках не является кортежем:

```
t2 = ('a')
type(t2)              # <type 'str'>
```

Для создания одноэлементного кортежа необходимо иметь конечную запятую.

```
t2 = ('a',)
type(t2)              # <type 'tuple'>
```

Обратите внимание, что для одноточечных кортежей рекомендуется использовать круглые скобки (см. PEP8 в концевых запятых). Кроме того, после запятой нет пробела (см. PEP8 на пробелах)

```
t2 = ('a',)           # PEP8-compliant
```

```
t2 = 'a',              # this notation is not recommended by PEP8
t2 = ('a', )           # this notation is not recommended by PEP8
```

Другой способ создания кортежа - встроенный `tuple` функции.

```
t = tuple('lupins')
print(t)               # ('l', 'u', 'p', 'i', 'n', 's')
t = tuple(range(3))
print(t)               # (0, 1, 2)
```

Эти примеры основаны на материале из книги **Think Python** Аллена Б. Дауни .

Упаковка и распаковка кортежей

Кортежи в Python являются значениями, разделенными запятыми. Закрывающиеся круглые скобки для ввода кортежей являются необязательными, поэтому два назначения

```
a = 1, 2, 3   # a is the tuple (1, 2, 3)
```

а также

```
a = (1, 2, 3) # a is the tuple (1, 2, 3)
```

эквивалентны. Назначение `a = 1, 2, 3` также называется *упаковкой,* поскольку оно объединяет значения в кортеж.

Обратите внимание, что однозначный кортеж также является кортежем. Чтобы сообщить Python, что переменная является кортежем, а не одним значением, вы можете использовать конечную запятую

```
a = 1  # a is the value 1
a = 1, # a is the tuple (1,)
```

Также необходима запятая, если вы используете круглые скобки

```
a = (1,) # a is the tuple (1,)
a = (1)  # a is the value 1 and not a tuple
```

Чтобы распаковать значения из кортежа и использовать несколько назначений

```
# unpacking AKA multiple assignment
x, y, z = (1, 2, 3)
# x == 1
# y == 2
# z == 3
```

Символ _ можно использовать как одноразовое имя переменной, если нужны только некоторые элементы кортежа, действующие в качестве заполнителя:

```
a = 1, 2, 3, 4
_, x, y, _ = a
# x == 2
# y == 3
```

Одиночные элементы:

```
x, = 1,  # x is the value 1
x  = 1,  # x is the tuple (1,)
```

В Python 3 целевая переменная с префиксом `*` может использоваться как переменная *catch-all* (см. Раздел «Распаковка итераций»):

Python 3.x 3.0

```
first, *more, last = (1, 2, 3, 4, 5)
# first == 1
# more == [2, 3, 4]
# last == 5
```

Обратные элементы

Обратные элементы в кортеже

```
colors = "red", "green", "blue"
rev = colors[::-1]
# rev: ("blue", "green", "red")
colors = rev
# colors: ("blue", "green", "red")
```

Или, используя обратный (обратный, дает итерабельность, которая преобразуется в кортеж):

```
rev = tuple(reversed(colors))
# rev: ("blue", "green", "red")
colors = rev
# colors: ("blue", "green", "red")
```

Встроенные функции Tuple

Кортежи поддерживают следующие встроенные функции

сравнение

Если элементы одного типа, python выполняет сравнение и возвращает результат. Если элементы разных типов, он проверяет, являются ли они числами.

- Если числа, выполните сравнение.

- Если какой-либо элемент является числом, то возвращается другой элемент.
- В противном случае типы сортируются по алфавиту.

Если мы дойдем до конца одного из списков, более длинный список «больше». Если оба списка одинаковы, он возвращает 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
```

Длина кортежа

Функция `len` возвращает общую длину кортежа

```
len(tuple1)
Out: 5
```

Макс кортежа

Функция `max` возвращает элемент из кортежа с максимальным значением

```
max(tuple1)
Out: 'e'

max(tuple2)
Out: '3'
```

Мин кортежа

Функция min возвращает элемент из кортежа с минимальным значением

```
min(tuple1)
Out: 'a'

min(tuple2)
Out: '1'
```

Преобразование списка в кортеж

Встроенный `tuple` функции преобразует список в кортеж.

```
list = [1,2,3,4,5]
tuple(list)
Out: (1, 2, 3, 4, 5)
```

Конкатенация кортежей

Использовать `+` для объединения двух кортежей

```
tuple1 + tuple2
Out: ('a', 'b', 'c', 'd', 'e', '1', '2', '3')
```

Прочитайте Кортеж онлайн: https://riptutorial.com/ru/python/topic/927/кортеж

глава 88: логирование

Examples

Введение в ведение журнала Python

Этот модуль определяет функции и классы, которые реализуют гибкую систему регистрации событий для приложений и библиотек.

Ключевое преимущество наличия API протоколирования, предоставляемого стандартным библиотечным модулем, состоит в том, что все модули Python могут участвовать в ведении журнала, поэтому ваш журнал приложений может включать ваши собственные сообщения, интегрированные с сообщениями сторонних модулей.

Итак, начнем:

Пример конфигурации непосредственно в коде

```
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')
```

Пример вывода:

```
2016-07-26 18:53:55,332 root         DEBUG    this is a debug test
```

Пример конфигурации через файл INI

Предположим, что файл имеет имя logging_config.ini. Более подробная информация о формате файла находится в разделе конфигурации ведения журнала учебного журнала .

```
[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
```

Затем используйте `logging.config.fileConfig()` в коде:

```
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')
```

Пример конфигурации через словарь

Начиная с Python 2.7, вы можете использовать словарь с подробными сведениями о конфигурации. PEP 391 содержит список обязательных и необязательных элементов в словаре конфигурации.

```
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')
```

Исключение регистрации

Если вы хотите регистрировать исключения, вы можете и должны использовать метод

`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
```

Не передавать исключение в качестве аргумента:

Поскольку `logging.exception(msg)` ожидает аргумент `msg` , это общая ошибка, чтобы передать исключение в вызов регистрации, например:

```
>>> 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
```

Хотя может показаться, что сначала это правильно, это на самом деле проблематично из-за причины того, как исключения и различные кодировки работают вместе в модуле регистрации:

```
>>> 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
```

Пытаясь зарегистрировать исключение, содержащее символы unicode, этот путь потерпит неудачу . Он скроет стек из первоначального исключения, переопределив его новым, который `logging.exception(e)` во время форматирования вашего `logging.exception(e)` .

Очевидно, что в вашем собственном коде вам может быть известно о кодировании в

исключениях. Однако сторонние библиотеки могут обрабатывать это по-другому.

Правильное использование:

Если вместо исключения вы просто передаете сообщение и пусть python делает свою магию, он будет работать:

```
>>> 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
```

Как вы можете видеть, мы фактически не используем `e` в этом случае, вызов `logging.exception(...)` магически форматирует последнее исключение.

Исключения журналов с уровнями журнала не ERROR

Если вы хотите регистрировать исключение с другим уровнем журнала, чем ERROR, вы можете использовать аргумент `exc_info` для регистраторов по умолчанию:

```
logging.debug('exception occurred', exc_info=1)
logging.info('exception occurred', exc_info=1)
logging.warning('exception occurred', exc_info=1)
```

Доступ к сообщению об исключении

Имейте в виду, что библиотеки там могут генерировать исключения с сообщениями как любые юникодные или (utf-8, если вам повезет) байт-строки. Если вам действительно нужен доступ к тексту исключения, единственным надежным способом, который будет всегда работать, является использование форматирования `repr(e)` или форматирования строки `%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
```

Прочитайте логирование онлайн: https://riptutorial.com/ru/python/topic/4081/логирование

глава 89: Манипулирование XML

замечания

Не все элементы XML-входа будут представлены как элементы дерева синтаксического анализа. В настоящее время этот модуль пропускает любые комментарии XML, инструкции по обработке и объявления типа документа во входе. Тем не менее деревья, построенные с использованием API этого модуля, а не синтаксический анализ из текста XML, могут содержать комментарии и инструкции по обработке; они будут включены при генерации вывода XML.

Examples

Открытие и чтение с использованием ElementTree

Импортируйте объект ElementTree, откройте соответствующий .xml-файл и получите корневой тег:

```
import xml.etree.ElementTree as ET
tree = ET.parse("yourXMLfile.xml")
root = tree.getroot()
```

Есть несколько способов поиска по дереву. Сначала по итерации:

```
for child in root:
    print(child.tag, child.attrib)
```

В противном случае вы можете ссылаться на определенные местоположения, такие как список:

```
print(root[0][1].text)
```

Для поиска определенных тегов по имени используйте `.find` или `.findall` :

```
print(root.findall("myTag"))
print(root[0].find("myOtherTag"))
```

Изменение файла XML

Импортировать модуль Element Tree и открыть xml-файл, получить элемент xml

```
import xml.etree.ElementTree as ET
tree = ET.parse('sample.xml')
root=tree.getroot()
```

```
element = root[0] #get first child of root element
```

Элемент объекта можно манипулировать, изменяя его поля, добавляя и изменяя атрибуты, добавляя и удаляя детей

```
element.set('attribute_name', 'attribute_value') #set the attribute to xml element
element.text="string_text"
```

Если вы хотите удалить элемент, используйте метод Element.remove ()

```
root.remove(element)
```

Метод ElementTree.write (), используемый для вывода объекта xml в xml-файлы.

```
tree.write('output.xml')
```

Создание и сборка XML-документов

Модуль импорта элемента

```
import xml.etree.ElementTree as ET
```

Функция Element () используется для создания элементов XML

```
p=ET.Element('parent')
```

Функция SubElement (), используемая для создания подэлементов в элементе give

```
c = ET.SubElement(p, 'child1')
```

Функция dump () используется для сброса xml-элементов.

```
ET.dump(p)
# Output will be like this
#<parent><child1 /></parent>
```

Если вы хотите сохранить файл, создайте дерево xml с помощью функции ElementTree () и сохраните в файл метод write ()

```
tree = ET.ElementTree(p)
tree.write("output.xml")
```

Функция Comment () используется для вставки комментариев в XML-файл.

```
comment = ET.Comment('user comment')
p.append(comment) #this comment will be appended to parent element
```

Открытие и чтение больших XML-файлов с использованием iterparse (инкрементный синтаксический анализ)

Иногда мы не хотим загружать весь XML-файл, чтобы получить нужную нам информацию. В этих случаях полезно использовать возможность поэтапной загрузки соответствующих разделов, а затем удалить их, когда мы закончим. С помощью функции iterparse вы можете редактировать дерево элементов, которое хранится при разборе XML.

Импортируйте объект ElementTree:

```
import xml.etree.ElementTree as ET
```

Откройте XML-файл и переберите все элементы:

```
for event, elem in ET.iterparse("yourXMLfile.xml"):
    ... do something ...
```

Кроме того, мы можем искать только определенные события, такие как начальные и конечные теги или пространства имен. Если этот параметр опущен (как указано выше), возвращаются только события «end»:

```
events=("start", "end", "start-ns", "end-ns")
for event, elem in ET.iterparse("yourXMLfile.xml", events=events):
    ... do something ...
```

Вот полный пример, показывающий, как очистить элементы от дерева в памяти, когда мы закончим с ними:

```
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 ...
```

Поиск XML с помощью XPath

Начиная с версии 2.7 `ElementTree` имеет лучшую поддержку запросов XPath. XPath - это синтаксис, позволяющий вам перемещаться по XML, например, SQL используется для поиска через базу данных. Обе функции `find` и `findall` поддерживают XPath. Ниже приведен пример xml ниже.

```
 <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>
```

Поиск всех книг:

```
import xml.etree.cElementTree as ET
tree = ET.parse('sample.xml')
tree.findall('Books/Book')
```

Поиск книги с названием = «Цвет магии»:

```
tree.find("Books/Book[Title='The Colour of Magic']")
# always use '' in the right side of the comparison
```

Поиск книги с id = 5:

```
tree.find("Books/Book[@id='5']")
# searches with xml attributes must have '@' before the name
```

Поиск второй книги:

```
tree.find("Books/Book[2]")
# indexes starts at 1, not 0
```

Поиск последней книги:

```
tree.find("Books/Book[last()]")
# 'last' is the only xpath function allowed in ElementTree
```

Поиск всех авторов:

```
tree.findall(".//Author")
#searches with // must use a relative path
```

Прочитайте Манипулирование XML онлайн: https://riptutorial.com/ru/python/topic/479/манипулирование-xml

глава 90: Массивы

Вступление

«Массивы» на Python - это не массивы в обычных языках программирования, таких как C и Java, но ближе к спискам. Список может быть набором однородных или гетерогенных элементов и может содержать ints, строки или другие списки.

параметры

параметр	подробности
`b`	Представляет знаковое целое число размером 1 байт
`B`	Представляет целое число без знака размера 1 байт
`c`	Представляет символ размера 1 байт
`u`	Представляет символ unicode размера 2 байта
`h`	Представляет знаковое целое число размером 2 байта
`H`	Представляет целое число без знака размером 2 байта
`i`	Представляет знаковое целое число размером 2 байта
`I`	Представляет целое число без знака размером 2 байта
`w`	Представляет символ unicode размером 4 байта
`l`	Представляет знаковое целое число размером 4 байта
`L`	Представляет целое число без знака 4 байта
`f`	Представляет с плавающей точкой размер 4 байта
`d`	Представляет с плавающей точкой размер 8 байтов

Examples

Основное введение в массивы

Массив - это структура данных, в которой хранятся значения одного и того же типа

данных. В Python это основное различие между массивами и списками.

Хотя списки python могут содержать значения, соответствующие различным типам данных, массивы в python могут содержать только значения, соответствующие одному типу данных. В этом уроке мы будем понимать массивы Python с несколькими примерами.

Если вы новичок в Python, начните с статьи Python Introduction.

Чтобы использовать массивы на языке python, вам нужно импортировать стандартный модуль `array` . Это связано с тем, что массив не является фундаментальным типом данных, например, строками, целыми числами и т. Д. Вот как вы можете импортировать модуль `array` в python:

```
from array import *
```

Когда вы импортируете модуль `array` , вы можете объявить массив. Вот как вы это делаете:

```
arrayIdentifierName = array(typecode, [Initializers])
```

В объявлении выше `arrayIdentifierName` - это имя массива, `typecode` позволяет python знать тип массива, а `Initializers` - это значения, с которыми инициализируется массив.

Typecodes - это коды, которые используются для определения типа значений массива или типа массива. Таблица в разделе параметров показывает возможные значения, которые можно использовать при объявлении массива и его типа.

Вот пример реального мира объявления массива python:

```
my_array = array('i',[1,2,3,4])
```

В приведенном выше примере используется тип typecode: `i` . Этот typecode представляет собой целое число со знаком, размер которого равен 2 байтам.

Вот простой пример массива, содержащего 5 целых чисел

```
from array import *
my_array = array('i', [1,2,3,4,5])
for i in my_array:
    print(i)
# 1
# 2
# 3
# 4
# 5
```

Доступ к отдельным элементам через индексы

Доступ к отдельным элементам осуществляется через индексы. Массивы Python ноль-

индексируются. Вот пример:

```
my_array = array('i', [1,2,3,4,5])
print(my_array[1])
# 2
print(my_array[2])
# 3
print(my_array[0])
# 1
```

Добавить любое значение в массив с помощью метода append ()

```
my_array = array('i', [1,2,3,4,5])
my_array.append(6)
# array('i', [1, 2, 3, 4, 5, 6])
```

Обратите внимание, что значение `6` было добавлено к существующим значениям массива.

Вставить значение в массив с помощью метода insert ()

Мы можем использовать метод `insert()` для вставки значения в любой индекс массива. Вот пример:

```
my_array = array('i', [1,2,3,4,5])
my_array.insert(0,0)
#array('i', [0, 1, 2, 3, 4, 5])
```

В приведенном выше примере значение 0 было вставлено в индекс 0. Обратите внимание, что первым аргументом является индекс, а вторым аргументом является значение.

Расширить массив python с помощью метода extend ()

Массив python может быть расширен более чем одним значением с использованием метода `extend()` . Вот пример:

```
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])
```

Мы видим, что массив my_array был расширен значениями `my_extnd_array` .

Добавить элементы из списка в массив, используя метод fromlist ()

Вот пример:

```
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])
```

Таким образом, мы видим, что значения 11, 12 и 13 были добавлены из списка `c` в `my_array` .

Удалите любой элемент массива с помощью метода remove ()

Вот пример:

```
my_array = array('i', [1,2,3,4,5])
my_array.remove(4)
# array('i', [1, 2, 3, 5])
```

Мы видим, что элемент 4 был удален из массива.

Удалить последний элемент массива с помощью метода pop ()

`pop` удаляет последний элемент из массива. Вот пример:

```
my_array = array('i', [1,2,3,4,5])
my_array.pop()
# array('i', [1, 2, 3, 4])
```

Итак, мы видим, что последний элемент (`5`) выскочил из массива.

Извлеките любой элемент через его индекс, используя метод index ()

`index()` возвращает первый индекс соответствующего значения. Помните, что массивы индексируются нулями.

```
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
```

Обратите внимание, что в этом втором примере был возвращен только один индекс, хотя значение существует дважды в массиве

Обратный массив python с использованием метода reverse ()

Метод `reverse()` делает то, что имя говорит, что он будет делать - инвертирует массив. Вот пример:

```
my_array = array('i', [1,2,3,4,5])
my_array.reverse()
# array('i', [5, 4, 3, 2, 1])
```

Получить информацию о буфере массива с помощью метода buffer_info ()

Этот метод предоставляет вам начальный адрес буфера массива в памяти и количество элементов в массиве. Вот пример:

```
my_array = array('i', [1,2,3,4,5])
my_array.buffer_info()
(33881712, 5)
```

Проверьте количество вхождений элемента с помощью метода count ()

`count()` вернет количество раз, и элемент появится в массиве. В следующем примере мы видим, что значение `3` встречается дважды.

```
my_array = array('i', [1,2,3,3,5])
my_array.count(3)
# 2
```

Преобразование массива в строку с помощью метода tostring ()

`tostring()` преобразует массив в строку.

```
my_char_array = array('c', ['g','e','e','k'])
# array('c', 'geek')
print(my_char_array.tostring())
# geek
```

Преобразование массива в список python с одинаковыми элементами с использованием метода tolist ()

Когда вам нужен объект `list` Python, вы можете использовать метод `tolist()` для преобразования вашего массива в список.

```
my_array = array('i', [1,2,3,4,5])
c = my_array.tolist()
# [1, 2, 3, 4, 5]
```

Добавить строку в массив char, используя метод fromstring ()

Вы можете добавить строку в массив символов, используя `fromstring()`

```
my_char_array = array('c', ['g','e','e','k'])
my_char_array.fromstring("stuff")
print(my_char_array)
#array('c', 'geekstuff')
```

Прочитайте Массивы онлайн: https://riptutorial.com/ru/python/topic/4866/массивы

глава 91: Математический модуль

Examples

Округление: круглое, напольное, потолочное, trunc

В дополнение к встроенной `round` функции, то `math` модуль обеспечивает `floor` , `ceil` и `trunc` функции.

```
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` и `round` всегда возвращают `float` .

```
round(1.3)  # 1.0
```

`round` всегда разрывает связи с нуля.

```
round(0.5)  # 1.0
round(1.5)  # 2.0
```

Python 3.x 3.0

`floor` , `ceil` и `trunc` всегда возвращают значение `Integral` , а `round` возвращает `Integral` значение, если `trunc` с одним аргументом.

```
round(1.3)       # 1
round(1.33, 1)   # 1.3
```

`round` перерывы связывают с ближайшим четным числом. Это исправляет смещение в сторону больших чисел при выполнении большого количества вычислений.

```
round(0.5)  # 0
round(1.5)  # 2
```

Предупреждение!

Как и в любом представлении с плавающей запятой, некоторые фракции *не могут быть представлены точно* . Это может привести к неожиданному поведению округления.

```
round(2.675, 2)  # 2.67, not 2.68!
```

Предупреждение о полу, trunc и целочисленном делении отрицательных чисел

Python (и C ++ и Java) округляются от нуля для отрицательных чисел. Рассматривать:

```
>>> math.floor(-1.7)
-2.0
>>> -5 // 2
-3
```

Логарифмы

`math.log(x)` дает естественный (базовый `e`) логарифм `x` .

```
math.log(math.e)  # 1.0
math.log(1)       # 0.0
math.log(100)     # 4.605170185988092
```

`math.log` может потерять точность с числами, близкими к 1, из-за ограничений чисел с плавающей запятой. Чтобы точно подсчитать логарифмы, близкие к 1, используйте `math.log1p` , который оценивает естественный логарифм 1 плюс аргумент:

```
math.log(1 + 1e-20)  # 0.0
math.log1p(1e-20)    # 1e-20
```

`math.log10` может использоваться для базы бревна 10:

```
math.log10(10)  # 1.0
```

Python 2.x 2.3.0

При использовании с двумя аргументами `math.log(x, base)` дает логарифм `x` в данной `base` (т.е. `log(x) / log(base)` .

```
math.log(100, 10) # 2.0
math.log(27, 3)   # 3.0
math.log(1, 10)   # 0.0
```

Копирование знаков

В Python 2.6 и выше `math.copysign(x, y)` возвращает `x` со знаком `y` . Возвращаемое значение всегда является `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
```

тригонометрия

Вычисление длины гипотенузы

```
math.hypot(2, 4) # Just a shorthand for SquareRoot(2**2 + 4**2)
# Out: 4.47213595499958
```

Преобразование градусов в / из радианов

Все `math` функции ожидают **радианов,** поэтому вам нужно преобразовать градусы в радианы:

```
math.radians(45)              # Convert 45 degrees to radians
# Out: 0.7853981633974483
```

Все результаты обратных тригонометрических функций возвращают результат в радианах, поэтому вам может потребоваться преобразовать его обратно в градусы:

```
math.degrees(math.asin(1))    # Convert the result of asin to degrees
# Out: 90.0
```

Синус, косинус, касательная и обратная функции

```
# 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"
```

Помимо `math.atan` существует также функция с двумя аргументами `math.atan2` , которая вычисляет правильный квадрант и избегает ловушек деления на ноль:

```
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"
```

Гиперболический синус, косинус и касательная

```
# 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
```

Константы

`math` модули включают в себя две часто используемые математические константы.

- `math.pi` - Математическая константа pi
- `math.e` - Математическая константа *e* (основа натурального логарифма)

```
>>> from math import pi, e
>>> pi
3.141592653589793
>>> e
2.718281828459045
>>>
```

Python 3.5 и выше имеют константы бесконечности и NaN («не число»). Более старый синтаксис передачи строки в `float()` прежнему работает.

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
```

Воображаемые числа

Воображаемые числа в Python представлены «j» или «J», заканчивающими целевой номер.

```
1j         # Equivalent to the square root of -1.
1j * 1j    # = (-1+0j)
```

Бесконечность и NaN («не число»)

Во всех версиях Python мы можем представить бесконечность и NaN («не число») следующим образом:

```
pos_inf = float('inf')      # positive infinity
neg_inf = float('-inf')     # negative infinity
not_a_num = float('nan')    # NaN ("not a number")
```

В Python 3.5 и выше мы также можем использовать определенные константы `math.inf` и `math.nan` :

Python 3.x 3.5

```
pos_inf = math.inf
neg_inf = -math.inf
```

```
not_a_num = math.nan
```

Строковые представления отображаются как `inf` и `-inf` и `nan` :

```
pos_inf, neg_inf, not_a_num
# Out: (inf, -inf, nan)
```

Мы можем проверить либо положительную, либо отрицательную бесконечность с `isinf` метода `isinf` :

```
math.isinf(pos_inf)
# Out: True

math.isinf(neg_inf)
# Out: True
```

Мы можем точно проверить положительную бесконечность или отрицательную бесконечность путем прямого сравнения:

```
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 и выше также позволяет проверять конечность:

Python 3.x 3.2

```
math.isfinite(pos_inf)
# Out: False

math.isfinite(0.0)
# Out: True
```

Операторы сравнения работают как ожидалось для положительной и отрицательной бесконечности:

```
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
```

Но если арифметическое выражение создает значение, большее максимального, которое может быть представлено как `float` , оно станет бесконечным:

```
pos_inf == sys.float_info.max * 1.0000001
# Out: True

neg_inf == -sys.float_info.max * 1.0000001
# Out: True
```

Однако деление на ноль не дает результата бесконечности (или, при необходимости, отрицательной бесконечности), а скорее вызывает исключение `ZeroDivisionError` .

```
try:
    x = 1.0 / 0.0
    print(x)
except ZeroDivisionError:
    print("Division by zero")

# Out: Division by zero
```

Арифметические операции на бесконечности дают бесконечные результаты или иногда 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 никогда не сравнится ни с чем, даже с самим собой. Мы можем протестировать его с `isnan` метода `isnan` :

```
not_a_num == not_a_num
# Out: False

math.isnan(not_a_num)
Out: True
```

NaN всегда сравнивается как «не равно», но не меньше или больше:

```
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
```

Арифметические операции на NaN всегда дают NaN. Это включает умножение на -1: нет отрицательного NaN.

```
5.0 * not_a_num
# Out: nan

float('-nan')
# Out: nan
```

Python 3.x 3.5

```
-math.nan
# Out: nan
```

Существует одно тонкое различие между старыми версиями `float` NaN и бесконечности и константами `math` библиотеки 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 для более быстрого возведения в степень

Использование модуля timeit из командной строки:

```
> 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
```

Встроенный `**` оператор часто пригодится, но если производительность имеет значение, используйте math.pow. Не забудьте отметить, однако, что pow возвращает float, даже если аргументы целые:

```
> from math import pow
> pow(5,5)
3125.0
```

Сложные числа и модуль cmath

Модуль `cmath` похож на `math` модуль, но определяет функции для сложной плоскости.

Прежде всего, комплексные числа являются числовым типом, который является частью языка Python, а не предоставлен классом библиотеки. Таким образом, нам не нужно `import cmath` для обычных арифметических выражений.

Заметим, что мы используем `j` (или `J`), а не `i` .

```
z = 1 + 3j
```

Мы должны использовать `1j` так как `j` будет именем переменной, а не числовым литералом.

```
1j * 1j
Out: (-1+0j)

1j ** 1j
# Out: (0.20787957635076193+0j)     # "i to the i"  ==  math.e ** -(math.pi/2)
```

Мы имеем `real` часть и `imag` (мнимую) часть, а также комплексное `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
```

Встроенные функции `abs` и `complex` также являются частью самого языка и не требуют импорта:

```
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)
```

`complex` функция может принимать строку, но она не может иметь пробелов:

```
complex('1+1j')
# Out: (1+1j)

complex('1 + 1j')
# Exception: ValueError: complex() arg is a malformed string
```

Но для большинства функций нам нужен модуль, например `sqrt` :

```
import cmath
```

```
cmath.sqrt(-1)
# Out: 1j
```

Естественно, поведение `sqrt` различно для сложных чисел и действительных чисел. В некоммерческой `math` квадратный корень отрицательного числа вызывает исключение:

```
import math

math.sqrt(-1)
# Exception: ValueError: math domain error
```

Для преобразования в полярные координаты и из них предусмотрены функции:

```
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)
```

Математическое поле комплексного анализа выходит за рамки этого примера, но многие функции в комплексной плоскости имеют «ветвь разреза», обычно вдоль вещественной оси или мнимой оси. Большинство современных платформ поддерживают «подписанный ноль», как указано в IEEE 754, что обеспечивает непрерывность этих функций по обеим сторонам среза. Следующий пример из документации Python:

```
cmath.phase(complex(-1.0, 0.0))
# Out: 3.141592653589793

cmath.phase(complex(-1.0, -0.0))
# Out: -3.141592653589793
```

Модуль `cmath` также предоставляет множество функций с прямыми аналогами из `math` модуля.

В дополнение к `sqrt` существуют сложные версии `exp` , `log` , `log10` , тригонометрические функции и их обратные (`sin` , `cos` , `tan` , `asin` , `acos` , `atan`) и гиперболические функции и их обратные (`sinh` , `cosh` , `tanh` , `asinh` , `acosh` , `atanh`). Обратите внимание, однако, нет сложного аналога `math.atan2` , `math.atan2` формы арктангенса.

```
cmath.log(1+1j)
# Out: (0.34657359027997264+0.7853981633974483j)

cmath.exp(1j * cmath.pi)
# Out: (-1+1.2246467991473532e-16j)   # e to the i pi == -1, within rounding error
```

Предусмотрены константы `pi` и `e` . Обратите внимание, что это `float` и не `complex` .

```
type(cmath.pi)
# Out: <class 'float'>
```

Модуль `cmath` также предоставляет сложные версии `isinf` и (для Python `isfinite`) `isfinite` . См. « Бесконечность и NaN ». Комплексное число считается бесконечным, если его действительная часть или ее мнимая часть бесконечна.

```
cmath.isinf(complex(float('inf'), 0.0))
# Out: True
```

Аналогично, модуль `cmath` предоставляет сложную версию `isnan` . См. « Бесконечность и NaN ». Сложное число считается «не числом», если либо его действительная часть, либо ее мнимая часть «не является числом».

```
cmath.isnan(0.0, float('nan'))
# Out: True
```

Обратите внимание, что нет `cmath` аналога `math.inf` и `math.nan` (от Python 3.5 и выше)

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'
```

В Python 3.5 и выше существует метод `isclose` как в `cmath` и в `math` модулях.

Python 3.x 3.5

```
z = cmath.rect(*cmath.polar(1+1j))

z
# Out: (1.0000000000000002+1.0000000000000002j)

cmath.isclose(z, 1+1j)
# True
```

Прочитайте Математический модуль онлайн: https://riptutorial.com/ru/python/topic/230/математический-модуль

глава 92: Метаклассы

Вступление

Метаклассы позволяют вам глубоко модифицировать поведение классов Python (с точки зрения их определения, создания экземпляров, доступа и т. Д.) Путем замены метакласса `type` который новые классы используют по умолчанию.

замечания

При проектировании вашей архитектуры учитывайте, что многие вещи, которые могут выполняться с помощью метаклассов, также могут быть выполнены с использованием более простой семантики:

- Традиционного наследования часто более чем достаточно.
- Декораторы классов могут сочетать функциональность с классами по специальному подходу.
- Python 3.6 представляет `__init_subclass__()` который позволяет классу участвовать в создании своего подкласса.

Examples

Основные метаклассы

Когда `type` вызывается с тремя аргументами, он ведет себя как класс (meta) и создает новый экземпляр, т. Е. он создает новый класс / тип.

```
Dummy = type('OtherDummy', (), dict(x=1))
Dummy.__class__              # <type 'type'>
Dummy().__class__.__class__  # <type 'type'>
```

`type` подкласса можно создать для пользовательского метакласса.

```
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
```

Теперь у нас есть новый `mytype` который можно использовать для создания классов таким же образом, как и `type` .

```
MyDummy = mytype('MyDummy', (), dict(x=2))
MyDummy.__class__              # <class '__main__.mytype'>
MyDummy().__class__.__class__  # <class '__main__.mytype'>
MyDummy.__custom_attribute__   # 2
```

Когда мы создаем новый класс с использованием ключевого слова `class` метаклас по умолчанию выбирается на основе базовых классов.

```
>>> class Foo(object):
...     pass

>>> type(Foo)
type
```

В приведенном выше примере единственным базовым классом является `object` поэтому наш метакласс будет типом `object` , который является `type` . Можно переопределить значение по умолчанию, однако это зависит от того, используем ли мы Python 2 или Python 3:

Python 2.x 2.7

Для определения метакласса можно использовать специальный атрибут `__metaclass__` .

```
class MyDummy(object):
    __metaclass__ = mytype
type(MyDummy)  # <class '__main__.mytype'>
```

Python 3.x 3.0

Специальный `metaclass` аргумент ключевого слова определяет метакласс.

```
class MyDummy(metaclass=mytype):
    pass
type(MyDummy)  # <class '__main__.mytype'>
```

Любые аргументы ключевых слов (кроме `metaclass`) в объявлении класса будут переданы в метакласс. Таким образом, `class MyDummy(metaclass=mytype, x=2)` передаст `x=2` в качестве аргумента ключевого слова в конструктор `mytype` .

Прочтите это подробное описание метаклассов python для более подробной информации.

Синглтоны, использующие метаклассы

Синглтон - это шаблон, который ограничивает экземпляр класса одним экземпляром / объектом. Для получения дополнительной информации о шаблонах проектирования синглтона python см. Здесь .

```
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
```

Использование метакласса

Синтаксис метакласса

Python 2.x 2.7

```
class MyClass(object):
    __metaclass__ = SomeMetaclass
```

Python 3.x 3.0

```
class MyClass(metaclass=SomeMetaclass):
    pass
```

Совместимость с Python 2 и 3 с `six`

```
import six

class MyClass(six.with_metaclass(SomeMetaclass)):
    pass
```

Пользовательские функции с метаклассами

Функциональность в метаклассах может быть изменена так, что всякий раз, когда создается класс, строка выводится на стандартный вывод или генерируется исключение. Этот метакласс отобразит имя создаваемого класса.

```
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
```

Вы можете использовать метакласс следующим образом:

```
class Spam(metaclass=VerboseMetaclass):
    def eggs(self):
        print("[insert example string here]")
s = Spam()
s.eggs()
```

Стандартный выход будет:

```
Creating class Spam
[insert example string here]
```

Введение в метаклассы

Что такое метакласс?

В Python все является объектом: целые числа, строки, списки, даже функции и классы сами по себе являются объектами. И каждый объект является экземпляром класса.

Чтобы проверить класс объекта x, можно вызвать `type(x)` , поэтому:

```
>>> type(5)
<type 'int'>
>>> type(str)
<type 'type'>
>>> type([1, 2, 3])
<type 'list'>

>>> class C(object):
...     pass
...
>>> type(C)
<type 'type'>
```

Большинство классов в python являются экземплярами `type` . сам `type` также является классом. Такие классы, экземпляры которых также являются классами, называются метаклассами.

Самый простой метакласс

OK, так что уже один метаклассом в Python: `type` . Можем ли мы создать еще один?

```
class SimplestMetaclass(type):
    pass
```

```
class MyClass(object):
    __metaclass__ = SimplestMetaclass
```

Это не добавляет каких-либо функциональных возможностей, но это новый метакласс, см., Что MyClass теперь является экземпляром SimplestMetaclass:

```
>>> type(MyClass)
<class '__main__.SimplestMetaclass'>
```

Метакласс, который делает что-то

Метакласс, который делает что-то обычно, переопределяет `type` `__new__` , чтобы изменить некоторые свойства создаваемого класса, прежде чем вызывать исходный `__new__` который создает класс:

```
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)
```

Метакласс по умолчанию

Возможно, вы слышали, что все в Python является объектом. Это правда, и все объекты имеют класс:

```
>>> type(1)
int
```

Литерал 1 является экземпляром `int` . Давайте объявим класс:

```
>>> class Foo(object):
...    pass
...
```

Теперь давайте создадим экземпляр:

```
>>> bar = Foo()
```

Что такое класс `bar` ?

```
>>> type(bar)
Foo
```

Ницца, `bar` - это пример `Foo` . Но каков класс самого `Foo` ?

```
>>> type(Foo)
type
```

Хорошо, сам `Foo` является экземпляром `type` . Как насчет `type` ?

```
>>> type(type)
type
```

Итак, что такое метакласс? Пока давайте притвориться, что это просто причудливое имя для класса класса. Takeaways:

- Все это объект в Python, поэтому все имеет класс
- Класс класса называется метаклассом
- Метакласс по умолчанию - это `type` , и на сегодняшний день он является наиболее распространенным метаклассом

Но почему вы должны знать о метаклассах? Ну, сам Python довольно «взломан», и концепция метакласса важна, если вы занимаетесь продвинутыми вещами, такими как мета-программирование, или если вы хотите контролировать, как инициализируются ваши классы.

Прочитайте Метаклассы онлайн: https://riptutorial.com/ru/python/topic/286/метаклассы

глава 93: Многомерные массивы

Examples

Списки в списках

Хорошим способом визуализации массива 2d является список списков. Что-то вроде этого:

```
lst=[[1,2,3],[4,5,6],[7,8,9]]
```

здесь внешний список `lst` имеет три вещи в нем. каждая из этих вещей - другой список: первый из них: `[1,2,3]` , второй - `[4,5,6]` а третий: `[7,8,9]` . Вы можете получить доступ к этим спискам так же, как и к другому элементу списка, например:

```
print (lst[0])
#output: [1, 2, 3]

print (lst[1])
#output: [4, 5, 6]

print (lst[2])
#output: [7, 8, 9]
```

Затем вы можете получить доступ к различным элементам в каждом из этих списков таким же образом:

```
print (lst[0][0])
#output: 1

print (lst[0][1])
#output: 2
```

Здесь первое число внутри скобок `[]` означает получение списка в этой позиции. В приведенном выше примере мы использовали число `0` чтобы получить список в 0-й позиции, который равен `[1,2,3]` . Второй набор `[]` скобок означает, что элемент находится в этой позиции из внутреннего списка. В этом случае мы использовали как `0` и `1` -ю позицию в списке, который мы получили, - это номер `1` а в 1-й позиции - `2`

Вы также можете установить значения внутри этих списков так же:

```
lst[0]=[10,11,12]
```

Теперь список `[[10,11,12],[4,5,6],[7,8,9]]` . В этом примере мы изменили весь первый список на совершенно новый список.

```
lst[1][2]=15
```

Теперь список `[[10,11,12],[4,5,15],[7,8,9]]` . В этом примере мы изменили один элемент внутри одного из внутренних списков. Сначала мы вошли в список в позиции 1 и изменили элемент внутри него в позиции 2, которому было 6, теперь это 15.

Списки в списках в списках в ...

Это может быть расширено. Вот трехмерный массив:

```
[[[111,112,113],[121,122,123],[131,132,133]],[[211,212,213],[221,222,223],[231,232,233]],[[311,312,313
```

Как очевидно, это становится немного трудным для чтения. Используйте обратную косую черту, чтобы разбить различные размеры:

```
[[[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]]]
```

Вложенные списки, подобные этому, можно расширить до произвольно больших размеров.

Доступ аналогичен 2D-массивам:

```
print(myarray)
print(myarray[1])
print(myarray[2][1])
print(myarray[1][0][2])
etc.
```

И редактирование тоже похоже:

```
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.
```

Прочитайте Многомерные массивы онлайн: https://riptutorial.com/ru/python/topic/8186/многомерные-массивы

глава 94: Многопоточность

Вступление

Темы позволяют программам Python обрабатывать сразу несколько функций, а не выполнять последовательность команд по отдельности. В этом разделе объясняются принципы работы с потоками и демонстрируется его использование.

Examples

Основы многопоточности

Используя `threading` модуль, новый поток выполнения может быть начат путем создания нового `threading.Thread` и присвоения ему функции для выполнения:

```
import threading

def foo():
  print "Hello threading!"

my_thread = threading.Thread(target=foo)
```

`target` параметр ссылается на функцию (или вызываемый объект), которая должна быть запущена. Нить не будет `start` до тех пор, пока не `start` вызов объекта `Thread` .

Запуск темы

```
my_thread.start() # prints 'Hello threading!'
```

Теперь, когда `my_thread` запущен и завершен, вызов `start` снова приведет к `RuntimeError` . Если вы хотите запустить свой поток в качестве демона, передав `daemon=True` kwarg или установив `my_thread.daemon` в `True` перед вызовом `start()` , ваш `Thread` будет запускаться тихо в фоновом режиме в качестве демона.

Присоединение к теме

В случаях, когда вы разделяете одну большую работу на несколько небольших и хотите запускать их одновременно, но перед тем, как продолжить, нужно дождаться, пока все они закончатся, `Thread.join()` - это метод, который вы ищете.

Например, скажем, вы хотите загрузить несколько страниц веб-сайта и скомпилировать их на одну страницу. Вы сделали бы это:

```
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)
```

Более пристальный взгляд на то, как работают функции `join()` можно найти здесь .

Создание пользовательского класса темы

Используя класс `threading.Thread` мы можем подклассифицировать новый пользовательский класс Thread. мы должны переопределить метод `run` в подклассе.

```
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.")
```

Общение между потоками

В коде есть несколько потоков, и вам нужно безопасно общаться между ними.

Вы можете использовать `Queue` из библиотеки `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()
```

Создание потоков производителей и потребителей с общей очередью

```
q = Queue()
t1 = Thread(target=consumer, args=(q,))
t2 = Thread(target=producer, args=(q,))
t1.start()
t2.start()
```

Создание рабочего пула

Использование `threading` и `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)
```

Использование `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, 3-е издание, Дэвид Бэйсли и Брайан K. Джонс (O'Reilly). Copyright 2013 Дэвид Бэйсли и Брайан Джонс, 978-1-449-34037-7.

Расширенное использование многопоточных

Этот раздел будет содержать некоторые из самых передовых примеров, реализованных с использованием многопоточности.

Расширенный принтер (регистратор)

Поток, который печатает все, принимается и изменяет выходные данные в соответствии с шириной терминала. Приятная часть заключается в том, что и «уже написанный» выход изменяется при изменении ширины терминала.

```
#!/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)
```

Стопорная нить с петлей 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!"
```

Исходя из этого вопроса .

Прочитайте Многопоточность онлайн: https://riptutorial.com/ru/python/topic/544/многопоточность

глава 95: многопроцессорная обработка

Examples

Выполнение двух простых процессов

Простым примером использования нескольких процессов будет два процесса (рабочие), которые выполняются отдельно. В следующем примере запускаются два процесса:

- `countUp()` подсчитывает 1 раз, каждую секунду.
- `countDown()` подсчитывает 1 вниз, каждую секунду.

```
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()
```

Вывод выглядит следующим образом:

```
Up:     0
Down:     3
Up:     1
Up:     2
Down:     2
Up:     3
Down:     1
```

```
Down:     0
```

Использование пула и карты

```
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` - это класс, который управляет несколькими `Workers` (процессами) за кулисами и позволяет вам, программисту, использовать.

`Pool(5)` создает новый пул с 5 процессами, а `pool.map` работает так же, как карта, но использует несколько процессов (количество, определенное при создании пула).

Аналогичные результаты могут быть достигнуты с помощью `map_async` , `apply` и `apply_async` которые можно найти в документации .

Прочитайте многопроцессорная обработка онлайн: https://riptutorial.com/ru/python/topic/3601/многопроцессорная-обработка

глава 96: Модуль base64

Вступление

Базовая кодировка 64 представляет собой общую схему кодирования двоичного кода в формате строки ASCII с использованием radix 64. Модуль base64 является частью стандартной библиотеки, что означает, что он устанавливает вместе с Python. Понимание байтов и строк имеет решающее значение для этой темы и может быть рассмотрено здесь . В этом разделе объясняется, как использовать различные функции и числовые базы модуля base64.

Синтаксис

- base64.b64encode (s, altchars = None)
- base64.b64decode (s, altchars = None, validate = False)
- base64.standard_b64encode (ы)
- base64.standard_b64decode (ы)
- base64.urlsafe_b64encode (ы)
- base64.urlsafe_b64decode (ы)
- base64.b32encode (ы)
- base64.b32decode (ы)
- base64.b16encode (ы)
- base64.b16decode (ы)
- 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 (б)

параметры

параметр	Описание
`base64.b64encode(s, altchars=None)`	
s	Байт-подобный объект
altchars	Байт-подобный объект длиной 2+ символов для замены символов «+» и «=» при создании алфавита Base64. Дополнительные символы игнорируются.
`base64.b64decode(s, altchars=None, validate=False)`	

параметр	Описание
s	Байт-подобный объект
altchars	Байт-подобный объект длиной 2+ символов для замены символов «+» и «=» при создании алфавита Base64. Дополнительные символы игнорируются.
утверждать	Если valide - True, символы, не находящиеся в нормальном алфавите Base64 или альтернативном алфавите, **не будут** отбрасываться до проверки дополнения
`base64.standard_b64encode(s)`	
s	Байт-подобный объект
`base64.standard_b64decode(s)`	
s	Байт-подобный объект
`base64.urlsafe_b64encode(s)`	
s	Байт-подобный объект
`base64.urlsafe_b64decode(s)`	
s	Байт-подобный объект
`b32encode(s)`	
s	Байт-подобный объект
`b32decode(s)`	
s	Байт-подобный объект
`base64.b16encode(s)`	
s	Байт-подобный объект
`base64.b16decode(s)`	
s	Байт-подобный объект
`base64.a85encode(b, foldspaces=False, wrapcol=0, pad=False, adobe=False)`	
б	Байт-подобный объект
foldspaces	Если foldspaces - True, символ 'y' будет использоваться вместо 4 последовательных

параметр	Описание
	пробелов.
wrapcol	Число символов перед новой строкой (0 не означает новых строк)
подушечка	Если pad является True, байты заполняются до кратного 4 перед кодированием
саман	Если adobe is True, закодированный секвенсор с рамкой '<~' и '' ~>, используемый в продуктах Adobe
`base64.a85decode(b, foldspaces=False, adobe=False, ignorechars=b'\t\n\r\v')`	
б	Байт-подобный объект
foldspaces	Если foldspaces - True, символ 'y' будет использоваться вместо 4 последовательных пробелов.
саман	Если adobe is True, закодированный секвенсор с рамкой '<~' и '' ~>, используемый в продуктах Adobe
ignorechars	Байт-подобный объект символов для игнорирования в процессе кодирования
`base64.b85encode(b, pad=False)`	
б	Байт-подобный объект
подушечка	Если pad является True, байты заполняются до кратного 4 перед кодированием
`base64.b85decode(b)`	
б	Байт-подобный объект

замечания

До выхода Python 3.4 функции кодирования и декодирования base64 работали только с `bytes` или типами `bytearray` . Теперь эти функции принимают любой байтоподобный объект .

Examples

Кодировка и декодирование Base64

Чтобы включить модуль base64 в ваш скрипт, вы должны сначала импортировать его:

```
import base64
```

Для кодирования и декодирования base64 оба требуют байтового объекта . Чтобы получить нашу строку в байтах, мы должны закодировать ее, используя встроенную функцию кодирования Python. Чаще всего используется `UTF-8` , однако полный список этих стандартных кодировок (включая языки с разными символами) можно найти здесь, в официальной документации на Python. Ниже приведен пример кодирования строки в байтах:

```
s = "Hello World!"
b = s.encode("UTF-8")
```

Результатом последней строки будет:

```
b'Hello World!'
```

Префикс `b` используется для обозначения значения байтового объекта.

Чтобы Base64 закодировать эти байты, мы используем `base64.b64encode()` :

```
import base64
s = "Hello World!"
b = s.encode("UTF-8")
e = base64.b64encode(b)
print(e)
```

Этот код выводит следующее:

```
b'SGVsbG8gV29ybGQh'
```

который все еще находится в объекте bytes. Чтобы получить строку из этих байтов, мы можем использовать метод `decode()` Python с `UTF-8` :

```
import base64
s = "Hello World!"
b = s.encode("UTF-8")
e = base64.b64encode(b)
s1 = e.decode("UTF-8")
print(s1)
```

Результатом будет:

```
SGVsbG8gV29ybGQh
```

Если мы хотим кодировать строку и затем декодировать, мы могли бы использовать метод `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)
```

Как вы и ожидали, выход будет исходной строкой:

```
Base64 Encoded: SGVsbG8gV29ybGQh
Hello World!
```

Кодирование и декодирование Base32

Модуль base64 также включает функции кодирования и декодирования для Base32. Эти функции очень похожи на функции 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)
```

Это приведет к следующему результату:

```
Base32 Encoded: JBSWY3DPEBLW64TMMQQQ====
Hello World!
```

Основание кодирования и декодирования16

Модуль base64 также включает функции кодирования и декодирования для Base16. Базу 16 чаще всего называют **шестнадцатеричной** . Эти функции очень похожи на функции Base64 и 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)
```

Это приведет к следующему результату:

```
Base16 Encoded: 48656C6C6F20576F726C6421
Hello World!
```

Кодирование и декодирование ASCII85

Adobe создала собственную кодировку под названием **ASCII85,** которая похожа на Base85, но имеет свои отличия. Эта кодировка часто используется в файлах Adobe PDF. Эти функции были выпущены в Python версии 3.4. В противном случае функции `base64.a85encode()` и `base64.a85encode()` аналогичны функциям предыдущего:

```
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)
```

Это обеспечивает следующее:

```
ASCII85 Encoded: 87cURD]i,"Ebo80
Hello World!
```

Кодировка и декодирование Base85

Подобно функциям Base64, Base32 и Base16, функции кодирования и декодирования `base64.b85encode()` - `base64.b85encode()` и `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)
```

который выводит следующее:

```
Base85 Encoded: NM&qnZy;B1a%^NF
Hello World!
```

Прочитайте Модуль base64 онлайн: https://riptutorial.com/ru/python/topic/8678/модуль-base64

глава 97: Модуль Deque

Синтаксис

- dq = deque () # Создает пустой deque
- dq = deque (iterable) # Создает deque с некоторыми элементами
- dq.append (объект) # Добавляет объект справа от deque
- dq.appendleft (object) # Добавляет объект слева от deque
- dq.pop () -> object # Удаляет и возвращает правый объект
- dq.popleft () -> object # Удаляет и возвращает левый самый объект
- dq.extend (iterable) # Добавляет некоторые элементы справа от deque
- dq.extendleft (iterable) # Добавляет некоторые элементы слева от deque

параметры

параметр	подробности
`iterable`	Создает deque с исходными элементами, скопированными из другого итерабельного.
`maxlen`	Ограничивает, насколько большой может быть дека, выталкивая старые элементы как новые.

замечания

Этот класс полезен, когда вам нужен объект, похожий на список, который позволяет быстро добавлять и удалять операции с обеих сторон (имя `deque` означает « *двойная очередь* »).

Предоставленные методы действительно очень похожи, за исключением того, что некоторые из них, такие как `pop` , `append` или `extend` могут быть добавлены `left` . Структура данных `deque` должна быть предпочтительнее списка, если нужно часто вставлять и удалять элементы на обоих концах, потому что это позволяет делать это в постоянное время O (1).

Examples

Использование базового deque

Основными методами, которые полезны для этого класса, являются `popleft` и `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])
```

ограничение размера deque

Используйте параметр `maxlen` , создавая deque для ограничения размера 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)
```

Доступные методы в deque

Создание пустого deque:

```
dl = deque()  # deque([]) creating empty deque
```

Создание deque с некоторыми элементами:

```
dl = deque([1, 2, 3, 4])  # deque([1, 2, 3, 4])
```

Добавление элемента в deque:

```
dl.append(5)  # deque([1, 2, 3, 4, 5])
```

Добавление элемента левой стороны deque:

```
dl.appendleft(0)  # deque([0, 1, 2, 3, 4, 5])
```

Добавление списка элементов в deque:

```
dl.extend([6, 7])  # deque([0, 1, 2, 3, 4, 5, 6, 7])
```

Добавление списка элементов с левой стороны:

```
dl.extendleft([-2, -1])  # deque([-1, -2, 0, 1, 2, 3, 4, 5, 6, 7])
```

Использование `.pop()` естественно удалит элемент с правой стороны:

```
dl.pop()  # 7 => deque([-1, -2, 0, 1, 2, 3, 4, 5, 6])
```

Использование `.popleft()` для удаления элемента с левой стороны:

```
dl.popleft()  # -1 deque([-2, 0, 1, 2, 3, 4, 5, 6])
```

Удалить элемент по его значению:

```
dl.remove(1)  # deque([-2, 0, 2, 3, 4, 5, 6])
```

Обратный порядок элементов в deque:

```
dl.reverse()  # deque([6, 5, 4, 3, 2, 0, -2])
```

Поиск в ширину

Deque - единственная структура данных Python с быстрыми операциями очереди . (Note `queue.Queue` обычно не подходит, поскольку он предназначен для связи между потоками.) Основной пример использования очереди - это поиск по ширине .

```
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
```

Скажем, у нас есть простой ориентированный граф:

```
graph = {1:[2,3], 2:[4], 3:[4,5], 4:[3,5], 5:[]}
```

Теперь мы можем найти расстояния от некоторой исходной позиции:

```
>>> bfs(graph, 1)
{1: 0, 2: 1, 3: 1, 4: 2, 5: 2}

>>> bfs(graph, 3)
{3: 0, 4: 1, 5: 1}
```

Прочитайте Модуль Deque онлайн: https://riptutorial.com/ru/python/topic/1976/модуль-deque

глава 98: Модуль dis

Examples

Константы в этом модуле

```
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', '...
```

Что такое байт-код Python?

Python - это гибридный интерпретатор. При запуске программы он сначала собирает его в *байт-код,* который затем может быть запущен в интерпретаторе Python (также называемом *виртуальной машиной Python*). Модуль `dis` в стандартной библиотеке можно использовать, чтобы сделать байт-код Python доступным для человека, разобрав классы, методы, функции и объекты кода.

```
>>> 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
```

Интерпретатор Python основан на стеках и использует систему «first-in last-out».

Каждый код операции (код операции) на языке ассемблера Python (байт-код) принимает фиксированное количество элементов из стека и возвращает фиксированное количество элементов в стек. Если в стеке недостаточно элементов для кода операции, интерпретатор Python будет аварийно завершен, возможно, без сообщения об ошибке.

Разборка модулей

Чтобы разобрать модуль Python, сначала это нужно преобразовать в `.pyc` файл (скомпилированный Python). Для этого запустите

```
python -m compileall <file>.py
```

Затем в интерпретаторе запустите

```
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
```

Это скомпилирует модуль Python и выведет инструкции байт-кода с помощью `dis` . Модуль никогда не импортируется, поэтому он безопасен для использования с ненадежным кодом.

Прочитайте Модуль dis онлайн: https://riptutorial.com/ru/python/topic/1763/модуль-dis

глава 99: Модуль Functools

Examples

частичный

`partial` функция создает приложение частичной функции из другой функции. Он используется для *привязки* значений к некоторым аргументам функции (или аргументам по ключевым словам) и создает *вызываемый* без уже определенных аргументов.

```
>>> from functools import partial
>>> unhex = partial(int, base=16)
>>> unhex.__doc__ = 'Convert base16 string to int'
>>> unhex('ca11ab1e')
3390155550
```

`partial()` , как следует из названия, допускает частичную оценку функции. Давайте посмотрим на следующий пример:

```
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
```

Когда `g` создается, `f` , который принимает четыре аргумента (`a, b, c, x`), также частично оценивается для первых трех аргументов, `a, b, c,` Оценка `f` завершается , когда `g` называется, `g(2)` , который проходит четвертый аргумент `f` .

Один из способов думать о `partial` - это регистр сдвига; нажав один аргумент в то время на некоторую функцию. `partial` подходит для случаев, когда данные поступают как поток, и мы не можем передавать более одного аргумента.

total_ordering

Когда мы хотим создать упорядочиваемый класс, обычно нам нужно определить методы `__eq()__` , `__lt__()` , `__le__()` , `__gt__()` и `__ge__()` .

Декоратор `total_ordering` , применяемый к классу, допускает определение `__eq__()` и только одно из `__lt__()` , `__le__()` , `__gt__()` и `__ge__()` и все еще допускает все операции упорядочения в классе.

```
@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))
```

Декоратор использует состав предоставленных методов и алгебраических операций для получения других методов сравнения. Например, если мы определили `__lt__()` и `__eq__()` и хотим вывести `__gt__()` , мы можем просто проверить `not __lt__() and not __eq__()` .

Примечание . Функция `total_ordering` доступна только с Python 2.7.

уменьшить

В Python 3.x функция `reduce` уже объясненная здесь , была удалена из встроенных модулей и теперь должна быть импортирована из `functools` .

```
from functools import reduce
def factorial(n):
    return reduce(lambda a, b: (a*b), range(1, n+1))
```

lru_cache

Декоратор `@lru_cache` можно использовать для `@lru_cache` дорогостоящей, вычислительно-интенсивной функции с использованием кеша, используемого в последнее время . Это позволяет запоминать вызовы функций, так что будущие вызовы с одинаковыми параметрами могут мгновенно возвращаться, а не быть перепрограммированными.

```
@lru_cache(maxsize=None)  # Boundless cache
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

>>> fibonacci(15)
```

В приведенном выше примере значение `fibonacci(3)` вычисляется только один раз, тогда как если у `fibonacci` не было кеша LRU, то `fibonacci(3)` были бы вычислены в 230 раз. Следовательно, `@lru_cache` особенно `@lru_cache` для рекурсивных функций или динамического программирования, где дорогостоящую функцию можно вызывать несколько раз с одинаковыми точными параметрами.

`@lru_cache` имеет два аргумента

- `maxsize` : количество вызовов для сохранения. Когда количество уникальных вызовов

превышает `maxsize` , кеш LRU удалит наименее недавно используемые вызовы.

- `typed` (добавлен в 3.3): Флаг для определения , если эквивалентные аргументы различных типов принадлежат к разным записям кэша (т.е. если `3.0` и `3` считаются разными аргументами)

Мы также видим статистику кеша:

```
>>> fib.cache_info()
CacheInfo(hits=13, misses=16, maxsize=None, currsize=16)
```

ПРИМЕЧАНИЕ . Поскольку `@lru_cache` использует словари для кэширования результатов, все параметры для этой функции должны быть хешируемыми для работы кеша.

Официальные документы Python для `@lru_cache` . `@lru_cache` добавлен в 3.2.

cmp_to_key

Python изменил методы сортировки, чтобы принять ключевую функцию. Эти функции принимают значение и возвращают ключ, который используется для сортировки массивов.

Старые функции сравнения, используемые для принятия двух значений и возврата -1, 0 или +1, если первый аргумент мал, равен или больше второго аргумента соответственно. Это несовместимо с новой ключевой функцией.

Вот где `functools.cmp_to_key` входит:

```
>>> import functools
>>> import locale
>>> sorted(["A", "S", "F", "D"], key=functools.cmp_to_key(locale.strcoll))
['A', 'D', 'F', 'S']
```

Пример, взятый и адаптированный из документации стандартной библиотеки Python .

Прочитайте Модуль Functools онлайн: https://riptutorial.com/ru/python/topic/2492/модуль-functools

глава 100: Модуль Itertools

Синтаксис

- `import itertools`

Examples

Группировка элементов из итерируемого объекта с использованием функции

Начните с итерабельного, который нужно сгруппировать

```
lst = [("a", 5, 6), ("b", 2, 4), ("a", 2, 5), ("c", 2, 6)]
```

Генерировать сгруппированный генератор, группируя второй элемент в каждом кортеже:

```
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)]
```

Сгруппированы только группы последовательных элементов. Возможно, вам придется сортировать по тому же ключу до вызова groupby For Eg, (последний элемент изменен)

```
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)]
```

Группа, возвращаемая groupby, является итератором, который будет недействителен перед следующей итерацией. Например, следующее не будет работать, если вы хотите, чтобы группы сортировались по ключу. Группа 5 пуста ниже, потому что, когда выбирается группа 2, она делает недействительными 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 []
```

Чтобы правильно сортировать, создайте список из итератора перед сортировкой

```
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)]
```

Возьмите кусочек генератора

Itertools «islice» позволяет обрезать генератор:

```
results = fetch_paged_results()  # returns a generator
limit = 20  # Only want the first 20 results
for data in itertools.islice(results, limit):
    print(data)
```

Обычно вы не можете нарезать генератор:

```
def gen():
    n = 0
    while n < 20:
        n += 1
        yield n

for part in gen()[:3]:
    print(part)
```

Дам

```
Traceback (most recent call last):
  File "gen.py", line 6, in <module>
    for part in gen()[:3]:
TypeError: 'generator' object is not subscriptable
```

Однако это работает:

```
import itertools

def gen():
    n = 0
    while n < 20:
        n += 1
        yield n

for part in itertools.islice(gen(), 3):
    print(part)
```

Обратите внимание, что, как обычный срез, вы также можете использовать аргументы `start` , `stop` и `step` :

```
itertools.islice(iterable, 1, 30, 3)
```

itertools.product

Эта функция позволяет вам перебирать декартово произведение списка итераций.

Например,

```
for x, y in itertools.product(xrange(10), xrange(10)):
    print x, y
```

эквивалентно

```
for x in xrange(10):
    for y in xrange(10):
        print x, y
```

Как и все функции python, которые принимают переменное количество аргументов, мы можем передать список itertools.product для распаковки с помощью оператора *.

Таким образом,

```
its = [xrange(10)] * 2
for x,y in itertools.product(*its):
    print x, y
```

дает те же результаты, что и в предыдущих примерах.

```
>>> 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

Вступление:

Эта простая функция порождает бесконечные ряды чисел. Например...

```
for number in itertools.count():
    if number > 20:
        break
    print(number)
```

Обратите внимание, что мы должны сломать или печатать навсегда!

Выход:

```
0
1
2
3
4
5
6
7
8
9
10
```

Аргументы:

`count()` принимает два аргумента: `start` и `step` :

```
for number in itertools.count(start=10, step=4):
    print(number)
    if number > 20:
        break
```

Выход:

```
10
14
18
22
```

itertools.takewhile

itertools.takewhile позволяет вам брать элементы из последовательности, пока условие сначала не станет `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)
```

Это выводит `[0, 2, 4, 12, 18]` .

Обратите внимание, что первое число, которое нарушает предикат (то есть: функция, возвращающая логическое значение) `is_even` is, `13` . Как только `takewhile` встречает значение, которое производит `False` для данного предиката, оно вырывается.

Результат, **полученный с** помощью `takewhile` , аналогичен **выходному** `takewhile` генерируемому кодом ниже.

```
def takewhile(predicate, iterable):
    for x in iterable:
        if predicate(x):
            yield x
        else:
            break
```

Примечание . Конкатенация результатов, полученных путем `takewhile` и `dropwhile` создает оригинальную итерабельность.

```
result = list(itertools.takewhile(is_even, lst)) + list(itertools.dropwhile(is_even, lst))
```

itertools.dropwhile

itertools.dropwhile позволяет вам брать элементы из последовательности после того, как условие сначала станет `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)
```

Это выдает `[13, 14, 22, 23, 44]` .

(*Этот пример такой же, как в примере для `takewhile` но с использованием `dropwhile` .*)

Обратите внимание, что первое число, которое нарушает предикат (то есть: функция, возвращающая логическое значение) `is_even` is, `13` . Все элементы до этого отбрасываются.

Результат, **созданный** `dropwhile` , аналогичен **выходному** `dropwhile` генерируемому кодом

ниже.

```
def dropwhile(predicate, iterable):
    iterable = iter(iterable)
    for x in iterable:
        if not predicate(x):
            yield x
            break
    for x in iterable:
        yield x
```

Конкатенация результатов, полученных путем `takewhile` и `dropwhile` дает исходный результат.

```
result = list(itertools.takewhile(is_even, lst)) + list(itertools.dropwhile(is_even, lst))
```

Закрепление двух итераторов до тех пор, пока они не будут исчерпаны

Подобно встроенной функции `zip()` , `itertools.zip_longest` будет продолжать итерирование за пределами более короткого из двух итераций.

```
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)
```

Необязательный аргумент `fillvalue` может быть передан (по умолчанию `''`) следующим образом:

```
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)
```

В Python 2.6 и 2.7 эта функция называется `itertools.izip_longest` .

Метод комбинаций в модуле Itertools

`itertools.combinations` вернет генератор последовательности *k*- комбинации списка.

Другими словами: он вернет генератор кортежей из всех возможных k-мерных комбинаций входного списка.

Например:

Если у вас есть список:

```
a = [1,2,3,4,5]
b = list(itertools.combinations(a, 2))
```

```
print b
```

Выход:

```
[(1, 2), (1, 3), (1, 4), (1, 5), (2, 3), (2, 4), (2, 5), (3, 4), (3, 5), (4, 5)]
```

Выше выход генератора преобразуется в список кортежей из всех возможных комбинаций *пары* -wise списка входных `a`

Вы также можете найти все 3 комбинации:

```
a = [1,2,3,4,5]
b = list(itertools.combinations(a, 3))
print b
```

Выход:

```
[(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)]
```

Объединение нескольких итераторов вместе

Используйте `itertools.chain` чтобы создать один генератор, который будет давать значения из нескольких генераторов в последовательности.

```
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))
```

Результаты в:

```
'1 2 3 4 x y z'
```

В качестве альтернативного конструктора вы можете использовать classmethod `chain.from_iterable` который в качестве единственного параметра принимает итерабельность итераций. Чтобы получить тот же результат, что и выше:

```
' '.join(chain.from_iterable([a,b])
```

Хотя `chain` может принимать произвольное количество аргументов, `chain.from_iterable` - единственный способ связать *бесконечное* число итераций.

itertools.repeat

Повторите что-нибудь n раз:

```
>>> import itertools
>>> for i in itertools.repeat('over-and-over', 3):
...    print(i)
over-and-over
over-and-over
over-and-over
```

Получите накопленную сумму чисел в итерируемой

Python 3.x 3.2

`accumulate` доходность совокупную сумму (или произведение) чисел.

```
>>> 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]
```

Цикл через элементы в итераторе

`cycle` - бесконечный итератор.

```
>>> import itertools as it
>>> it.cycle('ABCD')
A B C D A B C D A B C D ...
```

Поэтому следите за тем, чтобы использовать границы, чтобы избежать бесконечного цикла. Пример:

```
>>> # 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` возвращает генератор с последовательными перестановками r-длины элементов в истребителе.

```
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)]
```

если в списке `a` есть повторяющиеся элементы, в результате перестановки будут иметь

повторяющиеся элементы, вы можете использовать `set` для получения уникальных перестановок:

```
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)}
```

Прочитайте Модуль Itertools онлайн: https://riptutorial.com/ru/python/topic/1564/модуль-itertools

глава 101: Модуль JSON

замечания

Для полной документации, включая функциональность, зависящую от версии, ознакомьтесь с официальной документацией .

Типы

Значения по умолчанию

`json` модуль будет обрабатывать кодирование и декодирование следующих типов по умолчанию:

Типы де-сериализации:

JSON	питон
объект	ДИКТ
массив	список
строка	ул
число (int)	ИНТ
номер (реальный)	поплавок
true, false	Правда, ложь
ноль	Никто

Модуль `json` также понимает `NaN` , `Infinity` и `-Infinity` как их соответствующие значения float, которые находятся вне спецификации JSON.

Типы сериализации:

питон	JSON
ДИКТ	объект
список, кортеж	массив

питон	JSON
ул	строка
int, float, (int / float) -перечисленный Enums	число
Правда	правда
Ложь	ложный
Никто	ноль

Чтобы запретить кодирование `NaN` , `Infinity` и `-Infinity` вы должны закодировать с `allow_nan=False` . Это приведет к повышению значения `ValueError` если вы попытаетесь закодировать эти значения.

Пользовательская (де-) сериализация

Существуют различные крючки, которые позволяют обрабатывать данные, которые должны быть представлены по-разному. Использование `functools.partial` позволяет вам частично применить соответствующие параметры к этим функциям для удобства.

Сериализация:

Вы можете предоставить функцию, которая работает с объектами до их сериализации следующим образом:

```
# 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)
```

Десериализация:

Существуют различные крючки, которые обрабатываются функциями json, такими как object_hook и parse_float. Для исчерпывающего списка вашей версии python см. Здесь .

```
# 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)
```

Дальнейшая настройка (дезактивация):

Модуль `json` также позволяет расширять / заменять `json.JSONEncoder` и `json.JSONDecoder` для обработки разных типов. Крюки, описанные выше, могут быть добавлены как значения по умолчанию, создав эквивалентный именованный метод. Чтобы использовать их, просто передайте класс как параметр `cls` в соответствующую функцию. Использование `functools.partial` позволяет частично применить параметр cls к этим функциям для удобства, например

```
# 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

Создание JSON из Python dict

```
import json
d = {
    'foo': 'bar',
    'alice': 1,
    'wonderland': [1, 2, 3]
}
json.dumps(d)
```

Вышеприведенный фрагмент возвращает следующее:

```
'{"wonderland": [1, 2, 3], "foo": "bar", "alice": 1}'
```

Создание Python dict из JSON

```
import json
s = '{"wonderland": [1, 2, 3], "foo": "bar", "alice": 1}'
json.loads(s)
```

Вышеприведенный фрагмент возвращает следующее:

```
{u'alice': 1, u'foo': u'bar', u'wonderland': [1, 2, 3]}
```

Хранение данных в файле

Следующий фрагмент кодирует данные, хранящиеся в `d` в JSON, и сохраняет их в файле (замените `filename` на фактическое имя файла).

```
import json

d = {
    'foo': 'bar',
    'alice': 1,
    'wonderland': [1, 2, 3]
}

with open(filename, 'w') as f:
    json.dump(d, f)
```

Извлечение данных из файла

Следующий фрагмент открывает JSON-кодированный файл (заменяет `filename` фактическим именем файла) и возвращает объект, который хранится в файле.

```
import json

with open(filename, 'r') as f:
    d = json.load(f)
```

`load` vs` load`, `dump` vs` dumps`

Модуль `json` содержит функции как для чтения, так и для записи в строки Unicode, а также для чтения и записи в файлы и из них. Они дифференцируются по завершению `s` в имени функции. В этих примерах мы используем объект StringIO, но те же функции применимы к любому файлоподобному объекту.

Здесь мы используем строковые функции:

```
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": []}
```

И здесь мы используем файловые функции:

```
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": []}
```

Как вы можете видеть, основное отличие заключается в том, что при распаковке json-данных вы должны передать дескриптор файла функции, а не захватывать возвращаемое значение. Также стоит отметить, что перед чтением или письмом вы должны искать начало файла, чтобы избежать повреждения данных. При открытии файла курсор помещается в позицию `0` , так что ниже также будет работать:

```
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": []}
```

Имея оба способа работы с json-данными, вы можете идиоматически и эффективно работать с форматами, которые основываются на json, например `pyspark` -per-line `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')
```

Вызов `json.tool` из командной строки для вывода корректного вывода JSON

Учитывая некоторый JSON-файл «foo.json», например:

```
{"foo": {"bar": {"baz": 1}}}
```

мы можем вызвать модуль непосредственно из командной строки (передав имя файла в качестве аргумента), чтобы напечатать его:

```
$ python -m json.tool foo.json
{
    "foo": {
        "bar": {
            "baz": 1
        }
    }
}
```

Модуль также будет принимать входные данные от STDOUT, поэтому (в Bash) мы одинаково можем:

```
$ cat foo.json | python -m json.tool
```

Форматирование вывода JSON

Допустим, у нас есть следующие данные:

```
>>> data = {"cats": [{"name": "Tubbs", "color": "white"}, {"name": "Pepper", "color":
"black"}]}
```

Просто демпинг это, поскольку JSON не делает ничего особенного здесь:

```
>>> print(json.dumps(data))
{"cats": [{"name": "Tubbs", "color": "white"}, {"name": "Pepper", "color": "black"}]}
```

Настройка отступа для получения более красивого результата

Если мы хотим печатать, мы можем установить размер `indent` :

```
>>> print(json.dumps(data, indent=2))
{
  "cats": [
    {
```

```
      "name": "Tubbs",
      "color": "white"
    },
    {
      "name": "Pepper",
      "color": "black"
    }
  ]
}
```

Сортировка клавиш в алфавитном порядке для получения согласованного вывода

По умолчанию порядок ключей на выходе не определен. Мы можем получить их в алфавитном порядке, чтобы мы всегда получали одинаковый результат:

```
>>> print(json.dumps(data, sort_keys=True))
{"cats": [{"color": "white", "name": "Tubbs"}, {"color": "black", "name": "Pepper"}]}
```

Как избавиться от пробелов, чтобы получить компактный выход

Возможно, нам захочется избавиться от ненужных пробелов, которые выполняются путем установки разделительных строк, отличных от значений по умолчанию `', '` и `': '` :

```
>>>print(json.dumps(data, separators=(',', ':')))
{"cats":[{"name":"Tubbs","color":"white"},{"name":"Pepper","color":"black"}]}
```

JSON-кодирование пользовательских объектов

Если мы просто попробуем следующее:

```
import json
from datetime import datetime
data = {'datetime': datetime(2016, 9, 26, 4, 44, 0)}
print(json.dumps(data))
```

мы получаем сообщение об ошибке: `TypeError: datetime.datetime(2016, 9, 26, 4, 44) is not JSON serializable` .

Чтобы иметь возможность сериализовать объект datetime должным образом, нам нужно

написать собственный код для его преобразования:

```
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)
```

а затем используйте этот класс энкодера вместо `json.dumps` :

```
encoder = DatetimeJSONEncoder()
print(encoder.encode(data))
# prints {"datetime": "2016-09-26T04:44:00"}
```

Прочитайте Модуль JSON онлайн: https://riptutorial.com/ru/python/topic/272/модуль-json

глава 102: Модуль os

Вступление

Этот модуль обеспечивает переносимый способ использования функциональных возможностей, зависящих от операционной системы.

Синтаксис

- import os

параметры

параметр	подробности
Дорожка	Путь к файлу. Сепаратор пути может быть определен `os.path.sep` .
Режим	Желаемое разрешение в восьмеричном (например, `0700`)

Examples

Создать каталог

```
os.mkdir('newdir')
```

Если вам нужно указать разрешения, вы можете использовать необязательный аргумент `mode` :

```
os.mkdir('newdir', mode=0700)
```

Получить текущий каталог

Используйте `os.getcwd()` :

```
print(os.getcwd())
```

Определите имя операционной системы

Модуль `os` предоставляет интерфейс для определения того, какой тип операционной системы работает в настоящий момент.

```
os.name
```

В Python 3 это может вернуть одно из следующих:

- `posix`
- `nt`
- `ce`
- `java`

Более подробную информацию можно получить из `sys.platform`

Удалить каталог

Удалите каталог по `path` :

```
os.rmdir(path)
```

Вы не должны использовать `os.remove()` для удаления каталога. Эта функция предназначена для *файлов,* и использование ее в каталогах приведет к `OSError`

Следуйте символической ссылке (POSIX)

Иногда вам нужно определить цель символической ссылки. `os.readlink` сделает следующее:

```
print(os.readlink(path_to_symlink))
```

Изменение разрешений на файл

```
os.chmod(path, mode)
```

где `mode` - это требуемое разрешение, в восьмеричном.

makedirs - создание рекурсивного каталога

Для локального каталога со следующим содержимым:

```
 └── dir1
     ├── subdir1
     └── subdir2
```

Мы хотим создать тот же subdir1, subdir2 под новым каталогом dir2, который еще не существует.

```
import os

os.makedirs("./dir2/subdir1")
os.makedirs("./dir2/subdir2")
```

Выполнение этого результата

```
├── dir1
│   ├── subdir1
│   └── subdir2
└── dir2
    ├── subdir1
    └── subdir2
```

dir2 создается только в первый раз, когда это необходимо, для создания subdir1.

Если бы мы использовали **os.mkdir** вместо этого, у нас было бы исключение, потому что dir2 еще не существовало бы.

```
    os.mkdir("./dir2/subdir1")
OSError: [Errno 2] No such file or directory: './dir2/subdir1'
```

os.makedirs не понравится, если целевой каталог уже существует. Если мы снова запустим его:

```
OSError: [Errno 17] File exists: './dir2/subdir1'
```

Однако это можно легко устранить, поймав исключение и проверив, что каталог создан.

```
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
```

Прочитайте Модуль os онлайн: https://riptutorial.com/ru/python/topic/4127/модуль-os

глава 103: модуль pyautogui

Вступление

pyautogui - это модуль, используемый для управления мышью и клавиатурой. Этот модуль в основном используется для автоматизации задач нажатия клавиш и клавиатуры. Для мыши координаты экрана (0,0) начинаются с верхнего левого угла. Если вы вышли из-под контроля, затем быстро переместите курсор мыши влево-вверх, он возьмет управление с помощью мыши и клавиатуры с Python и вернет его вам.

Examples

Функции мыши

Это некоторые из полезных функций мыши для управления мышью.

```
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.
```

Функции клавиатуры

Это некоторые полезные функции клавиатуры для автоматизации нажатия клавиши.

```
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 и распознавание изображений

Эта функция поможет вам сделать снимок экрана, а также сопоставить изображение с частью экрана.

```
.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.
```

Прочитайте модуль pyautogui онлайн: https://riptutorial.com/ru/python/topic/9432/модуль-pyautogui

глава 104: Модуль Sqlite3

Examples

Sqlite3 - не требует отдельного процесса сервера.

Модуль sqlite3 был написан Герхардом Хэринг. Чтобы использовать модуль, вы должны сначала создать объект Connection, который представляет базу данных. Здесь данные будут сохранены в файле example.db:

```
import sqlite3
conn = sqlite3.connect('example.db')
```

Вы также можете указать специальное имя: memory: создать базу данных в ОЗУ. После того, как у вас есть соединение, вы можете создать объект Cursor и вызвать его метод execute () для выполнения команд 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()
```

Получение значений из базы данных и обработки ошибок

Получение значений из базы данных SQLite3.

Печатать значения строк, возвращенные выбранным запросом

```
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
```

Чтобы получить один метод сопоставления fetchone ()

```
print c.fetchone()
```

Для нескольких строк используется метод fetchall ()

```
a=c.fetchall() #which is similar to list(cursor) method used previously
for row in a:
    print row
```

Обработка ошибок может быть выполнена с использованием встроенной функции sqlite3.Error

```
try:
    #SQL Code
except sqlite3.Error as e:
    print "An error occurred:", e.args[0]
```

Прочитайте Модуль Sqlite3 онлайн: https://riptutorial.com/ru/python/topic/7754/модуль-sqlite3

глава 105: Модуль Webbrowser

Вступление

Согласно стандартной документации Python, модуль webbrowser предоставляет интерфейс высокого уровня, позволяющий пользователям просматривать веб-документы. В этом разделе объясняется и демонстрируется правильное использование модуля webbrowser.

Синтаксис

- `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)`

параметры

параметр	подробности
`webbrowser.open()`	
URL	URL-адрес, который нужно открыть в веб-браузере
новый	0 открывает URL-адрес на существующей вкладке, 1 открывается в новом окне, 2 открывается на новой вкладке
AutoRaise	если установлено значение Истина, окно будет перемещено поверх других окон
`webbrowser.open_new()`	
URL	URL-адрес, который нужно открыть в веб-браузере
`webbrowser.open_new_tab()`	
URL	URL-адрес, который нужно открыть в веб-браузере
`webbrowser.get()`	
с помощью	браузер использовать
`webbrowser.register()`	
URL	имя браузера
конструктор	путь к исполняемому браузеру (справка)

параметр	подробности
пример	Экземпляр веб-браузера, возвращаемый методом `webbrowser.get()`

замечания

В следующей таблице перечислены предопределенные типы браузеров. Левый столбец - это имена, которые можно передать в метод `webbrowser.get()` а в правом столбце перечислены имена классов для каждого типа браузера.

Тип Название	Название класса
`'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

Открытие URL-адреса с помощью браузера по умолчанию

Чтобы просто открыть URL-адрес, используйте метод `webbrowser.open()` :

```
import webbrowser
webbrowser.open("http://stackoverflow.com")
```

Если окно браузера открыто, метод откроет новую вкладку с указанным URL-адресом. Если окно не открыто, метод откроет браузер операционной системы по умолчанию и переместится к URL-адресу в параметре. Открытый метод поддерживает следующие параметры:

- `url` - URL-адрес, который нужно открыть в веб-браузере (строка) **[обязательно]**
- `new` - 0 открывается в текущей вкладке, 1 открывает новое окно, 2 открывает новую вкладку (целое число) **[по умолчанию 0]**
- `autoraise` - если установлено значение True, окно будет перемещено поверх окон других приложений (Boolean) **[по умолчанию False]**

Обратите внимание, что `new` и `autoraise` аргументы редко работают, так как большинство современных браузеров отказываются от этих коммитов.

Webbrowser также может попытаться открыть URL-адреса в новых окнах с `open_new` метода `open_new` :

```
import webbrowser
webbrowser.open_new("http://stackoverflow.com")
```

Этот метод обычно игнорируется современными браузерами, и URL-адрес обычно открывается на новой вкладке. Открытие новой вкладки может быть проверено модулем с `open_new_tab` метода `open_new_tab` :

```
import webbrowser
webbrowser.open_new_tab("http://stackoverflow.com")
```

Открытие URL-адреса с помощью разных браузеров

Модуль webbrowser также поддерживает различные браузеры, используя методы `register()` и `get()` . Метод get используется для создания контроллера браузера с использованием пути конкретного исполняемого файла, а метод register используется для присоединения этих исполняемых файлов к предустановленным типам браузера для будущего использования, обычно при использовании нескольких типов браузеров.

```
import webbrowser
```

```
ff_path = webbrowser.get("C:/Program Files/Mozilla Firefox/firefox.exe")
ff = webbrowser.get(ff_path)
ff.open("http://stackoverflow.com/")
```

Регистрация типа браузера:

```
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/")
```

Прочитайте Модуль Webbrowser онлайн: https://riptutorial.com/ru/python/topic/8676/модуль-webbrowser

глава 106: Модуль коллекций

Вступление

Встроенный `collections` предоставляет несколько специализированных гибких типов коллекций, которые являются высокопроизводительными и предоставляют альтернативы общим типам коллекции `dict` , `list` , `tuple` и `set` . Модуль также определяет абстрактные базовые классы, описывающие различные типы функций сбора (такие как `MutableSet` и `ItemsView`).

замечания

В модуле **коллекций** доступны три других типа, а именно:

1. UserDict
2. UserList
3. UserString

Каждый из них действует как обертка вокруг связанного объекта, например, *UserDict* действует как обертка вокруг объекта *dict* . В каждом случае класс имитирует свой именованный тип. Содержимое экземпляра хранится в объекте обычного типа, доступном через атрибут data экземпляра обертки. В каждом из этих трех случаев потребность в этих типах была частично вытеснена возможностью подкласса непосредственно из основного типа; однако с классом-оболочкой легче работать, поскольку базовый тип доступен как атрибут.

Examples

collections.Counter

Счетчик - это подкласс класса dict, который позволяет легко подсчитывать объекты. Он имеет полезные методы для работы с частотами объектов, которые вы считаете.

```
import collections
counts = collections.Counter([1,2,3])
```

приведенный выше код создает объект, счетчик, который имеет частоты всех элементов, переданных конструктору. Этот пример имеет значение `Counter({1: 1, 2: 1, 3: 1})`

Примеры конструктора

Счетчик писем

```
>>> 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})
```

Счетчик слов

```
>>> 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})
```

Рецепты

```
>>> c = collections.Counter({'a': 4, 'b': 2, 'c': -2, 'd': 0})
```

Получить количество отдельных элементов

```
>>> c['a']
4
```

Установить количество отдельных элементов

```
>>> c['c'] = -3
>>> c
Counter({'a': 4, 'b': 2, 'd': 0, 'c': -3})
```

Получить общее количество элементов в счетчике (4 + 2 + 0 - 3)

```
>>> sum(c.itervalues())  # negative numbers are counted!
3
```

Получить элементы (сохраняются только те, у которых есть положительный счетчик)

```
>>> list(c.elements())
['a', 'a', 'a', 'a', 'b', 'b']
```

Удалить ключи с 0 или отрицательным значением

```
>>> c - collections.Counter()
Counter({'a': 4, 'b': 2})
```

Удалить все

```
>>> c.clear()
>>> c
Counter()
```

Добавить удалить отдельные элементы

```
>>> 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) возвращает подкласс `dict` который имеет значение по умолчанию для отсутствующих ключей. Аргумент должен быть функцией, которая возвращает значение по умолчанию при вызове без аргументов. Если ничего не прошло, по умолчанию используется значение `None` .

```
>>> state_capitals = collections.defaultdict(str)
>>> state_capitals
defaultdict(<class 'str'>, {})
```

возвращает ссылку на defaultdict, который будет создавать строковый объект с его методом default_factory.

Типичным использованием `defaultdict` является использование одного из встроенных типов, таких как `str` , `int` , `list` или `dict` как default_factory, поскольку они возвращают пустые типы при вызове без аргументов:

```
>>> str()
''
>>> int()
0
>>> list
[]
```

Вызов defaultdict с ключом, который не существует, не вызывает ошибки, как в обычном словаре.

```
>>> state_capitals['Alaska']
''
>>> state_capitals
defaultdict(<class 'str'>, {'Alaska': ''})
```

Другой пример: `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})
```

Обычные словарные методы работают со стандартным словарем

```
>>> state_capitals['Alabama'] = 'Montgomery'
>>> state_capitals
defaultdict(<class 'str'>, {'Alabama': 'Montgomery', 'Alaska': ''})
```

Использование `list` как default_factory приведет к созданию списка для каждого нового ключа.

```
>>> 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

Порядок ключей в словарях Python произволен: они не регулируются порядком, в котором вы их добавляете.

Например:

```
>>> 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}
```
```

(Вышеуказанный произвольный порядок означает, что вы можете получить разные результаты с приведенным выше кодом, как показано здесь.)

Порядок, в котором появляются клавиши, - это порядок, в котором они будут повторяться, например, используя цикл `for` .

Класс `collections.OrderedDict` предоставляет объекты словаря, которые сохраняют порядок ключей. `OrderedDict` s может быть создан, как показано ниже, с серией упорядоченных элементов (здесь, список пар ключей-значений кортежа):

```
>>> 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)])
```

Или мы можем создать пустой `OrderedDict` а затем добавить элементы:

```
>>> o = OrderedDict()
>>> o['key1'] = "value1"
>>> o['key2'] = "value2"
>>> print(o)
OrderedDict([('key1', 'value1'), ('key2', 'value2')])
```

Итерация через `OrderedDict` позволяет использовать ключ в том порядке, в котором они были добавлены.

Что произойдет, если мы присвоим новое значение существующему ключу?

```
>>> d['foo'] = 4
>>> print(d)
OrderedDict([('foo', 4), ('bar', 6), ('baz', 7), ('foobar', 8)])
```

Ключ сохраняет свое первоначальное место в `OrderedDict` .

collections.namedtuple

Определите новый тип `Person` использующий `namedtuple` следующим образом:

```
Person = namedtuple('Person', ['age', 'height', 'name'])
```

Второй аргумент - это список атрибутов, которые будет иметь кортеж. Вы можете также перечислить эти атрибуты как строки, разделенные запятыми:

```
Person = namedtuple('Person', 'age, height, name')
```

или же

```
Person = namedtuple('Person', 'age height name')
```

После определения именованный кортеж может быть создан путем вызова объекта с необходимыми параметрами, например:

```
dave = Person(30, 178, 'Dave')
```

Именованные аргументы также могут использоваться:

```
jack = Person(age=30, height=178, name='Jack S.')
```

Теперь вы можете получить доступ к атрибутам namedtuple:

```
print(jack.age)  # 30
print(jack.name)  # 'Jack S.'
```

Первым аргументом конструктора namedtuple (в нашем примере `'Person'`) является `typename` . Типично использовать одно и то же слово для конструктора и typename, но они могут быть разными:

```
Human = namedtuple('Person',  'age, height, name')
dave = Human(30, 178, 'Dave')
print(dave)  # yields: Person(age=30, height=178, name='Dave')
```

collections.deque

Возвращает новый объект `deque` инициализированный слева направо (с использованием append ()) с данными из итерабельного. Если iterable не указан, новый `deque` пуст.

Deques - это обобщение стеков и очередей (название произносится как «колода» и не подходит для «очереди с двойным концом»). Deques поддерживает поточно-безопасную, эффективную по объему память добавляет и выскакивает с обеих сторон `deque` примерно с той же производительностью O (1) в любом направлении.

Хотя объекты списка поддерживают аналогичные операции, они оптимизированы для операций с быстрой фиксированной длиной и несут затраты на перемещение памяти O (n) для операций pop (0) и insert (0, v), которые изменяют как размер, так и положение базового представления данных ,

Новое в версии 2.4.

Если `maxlen` не указан или `None` , то уровни могут выражаться до произвольной длины. В противном случае `deque` ограничена указанной максимальной длиной. После того, как `deque` ограниченной длины заполнено, когда новые предметы добавляются, соответствующее количество предметов отбрасывается с противоположного конца. Ограниченные ограничения длины обеспечивают функциональность, подобную хвостовому фильтру в Unix. Они также полезны для отслеживания транзакций и других пулов данных, которые представляют интерес только для самых последних видов деятельности.

Изменено в версии 2.6: Добавлен параметр 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'])
```

Источник: https://docs.python.org/2/library/collections.html

collections.ChainMap

`ChainMap` является новым в **версии 3.3**

Возвращает новый объект `ChainMap` учетом количества `maps` . Этот объект группирует несколько диктов или других сопоставлений вместе для создания единого обновляемого представления.

`ChainMap` s полезны для управления вложенными контекстами и наложениями. Пример в

мире python содержится в реализации класса `Context` в движке шаблонов Django. Это полезно для быстрого связывания нескольких отображений, чтобы результат можно рассматривать как единое целое. Это часто намного быстрее, чем создание нового словаря и выполнение нескольких вызовов `update()` .

Каждый раз, когда есть цепочка значений поиска, может быть случай для `ChainMap` . Пример включает в себя как пользовательские значения, так и словарь значений по умолчанию. Другим примером являются карты параметров `POST` и `GET` найденные в Интернете, например Django или Flask. Благодаря использованию `ChainMap` возвращается комбинированный вид двух разных словарей.

Список параметров `maps` упорядочен по первому запросу до последнего поиска. Поисковые запросы последовательно просматривают базовые сопоставления до тех пор, пока не будет найден ключ. Напротив, записи, обновления и удаления работают только при первом сопоставлении.

```
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)
```

Обратите внимание, что влияние порядка, по которому значение определяется первым в последующем поиске

```
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
```

Прочитайте Модуль коллекций онлайн: https://riptutorial.com/ru/python/topic/498/модуль-коллекций

глава 107: Модуль локали

замечания

Документы Python 2: [https://docs.python.org/2/library/locale.html#locale.currency][1]

Examples

Форматирование валюты в долларах США с использованием модуля локали

```
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'
```

Прочитайте Модуль локали онлайн: https://riptutorial.com/ru/python/topic/1783/модуль-локали

глава 108: Модуль очереди

Вступление

Модуль Queue реализует многопроцессорные очереди с несколькими потребителями. Это особенно полезно при программировании с резьбой, когда информация должна быть безопасно заменена между несколькими потоками. Существует три типа очередей, предоставляемых модулем очереди, которые следующие: 1. Очередь 2. LifoQueue 3. Приоритет PriorityQueue, который может быть достигнут: 1. Полный (переполнение очереди) 2. Пустой (нижний поток очереди)

Examples

Простой пример

```
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))
```

Выход:

```
('key', 1)
('key', 2)
('key', 3)
('key', 4)
('key', 5)
('key', 6)
('key', 7)
('key', 8)
('key', 9)
```

Прочитайте Модуль очереди онлайн: https://riptutorial.com/ru/python/topic/8339/модуль-очереди

глава 109: Написание расширений

Examples

Hello World с расширением C

Следующий исходный файл C (который мы будем называть `hello.c` для демонстрационных целей) создает модуль расширения с именем `hello` который содержит одну функцию `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
```

Чтобы скомпилировать файл с помощью `gcc` компилятора, выполните следующую команду в своем любимом терминале:

```
gcc /path/to/your/file/hello.c -o /path/to/your/file/hello
```

Чтобы выполнить функцию `greet()` которую мы написали ранее, создайте файл в том же

каталоге и назовите его `hello.py`

```
import hello           # imports the compiled library
hello.greet("Hello!") # runs the greet() function with "Hello!" as an argument
```

Передача открытого файла в C Расширения

Передайте открытый файловый объект из кода расширения Python в C.

Вы можете преобразовать файл в дескриптор целочисленного файла, используя функцию `PyObject_AsFileDescriptor` :

```
PyObject *fobj;
int fd = PyObject_AsFileDescriptor(fobj);
if (fd < 0){
    return NULL;
}
```

Чтобы преобразовать дескриптор целочисленного файла обратно в объект python, используйте `PyFile_FromFd` .

```
int fd; /* Existing file descriptor */
PyObject *fobj = PyFile_FromFd(fd, "filename","r",-1,NULL,NULL,NULL,1);
```

C Расширение Использование c ++ и Boost

Это основной пример *расширения* C с использованием C ++ и Boost .

Код C ++

Код на C ++ помещен в 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)
        ;
}
```

Чтобы скомпилировать это в модуль python, вам понадобятся заголовки python и библиотеки boost. Этот пример был сделан на Ubuntu 12.04 с использованием python 3.4 и gcc. Boost поддерживается на многих платформах. В случае Ubuntu необходимые пакеты были установлены с использованием:

```
sudo apt-get install gcc libboost-dev libpython3.4-dev
```

Компиляция исходного файла в .so-файл, который впоследствии может быть импортирован как модуль, если он находится на пути python:

```
gcc -shared -o hello.so -fPIC -I/usr/include/python3.4 hello.cpp -lboost_python-py34 -
lboost_system -l:libpython3.4m.so
```

Код python в файле example.py:

```
import hello

print(hello.get_hello())

h = hello.Hello("World hello!")
print(h.as_list(3))
```

Затем `python3 example.py` даст следующий результат:

```
Hello world!
['World hello!', 'World hello!', 'World hello!']
```

Прочитайте Написание расширений онлайн: https://riptutorial.com/ru/python/topic/557/написание-расширений

глава 110: начать с GZip

Вступление

Этот модуль обеспечивает простой интерфейс для сжатия и распаковки файлов, как и программы GNU gzip и gunzip.

Сжатие данных обеспечивается модулем zlib.

Модуль gzip предоставляет класс GzipFile, который моделируется после Файлового объекта Python. Класс GzipFile читает и записывает файлы в формате gzip, автоматически сжимая или распаковывая данные, чтобы он выглядел как обычный файловый объект.

Examples

Чтение и запись ZIP-файлов 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)
```

Сохраните его как 1gzip_write.py1.Run его через терминал.

```
$ python gzip_write.py

application/x-gzip; charset=binary
example.txt.gz contains 68 bytes of compressed data
```

Прочитайте начать с GZip онлайн: https://riptutorial.com/ru/python/topic/8993/начать-с-gzip

глава 111: Неизменяемые типы данных (int, float, str, кортеж и frozensets)

Examples

Отдельные символы строк не могут быть назначены

```
foo = "bar"
foo[0] = "c" # Error
```

Неизменяемое значение переменной не может быть изменено после их создания.

Отдельные члены Tuple не могут быть назначены

```
foo = ("bar", 1, "Hello!",)
foo[1] = 2 # ERROR!!
```

Вторая строка вернет ошибку, так как члены-кортежи после создания не назначаются. Из-за неизменности кортежа.

Frozenset являются неизменными и не могут быть назначены

```
foo = frozenset(["bar", 1, "Hello!"])
foo[2] = 7 # ERROR
foo.add(3) # ERROR
```

Вторая строка будет возвращать ошибку, поскольку члены frozenset после создания не могут быть назначены. Третья строка вернет ошибку, поскольку фризонсеты не поддерживают функции, которые могут манипулировать членами.

Прочитайте Неизменяемые типы данных (int, float, str, кортеж и frozensets) онлайн: https://riptutorial.com/ru/python/topic/4806/неизменяемые-типы-данных--int--float--str--кортеж-и-frozensets-

глава 112: Неофициальные реализации Python

Examples

IronPython

Реализация с открытым исходным кодом для .NET и Mono, написанная на C #, лицензированная под лицензией Apache 2.0. Он полагается на DLR (Dynamic Language Runtime). Он поддерживает только версию 2.7, версия 3 в настоящее время разрабатывается.

Различия с CPython:

- Тесная интеграция с .NET Framework.
- Строки по умолчанию Unicode.
- Не поддерживает расширения для CPython, написанные на C.
- Не страдает от Global Interpreter Lock.
- Производительность обычно ниже, хотя это зависит от тестов.

Привет, мир

```
print "Hello World!"
```

Вы также можете использовать функции .NET:

```
import clr
from System import Console
Console.WriteLine("Hello World!")
```

внешняя ссылка

- Официальный веб-сайт
- Репозиторий GitHub

Jython

Реализация Open-source для JVM, написанная на Java, лицензированная в соответствии с лицензией Python Software Foundation. Он поддерживает только версию 2.7, версия 3 в настоящее время разрабатывается.

Различия с CPython:

- Тесная интеграция с JVM.
- Строки - это Юникод.
- Не поддерживает расширения для CPython, написанные на C.
- Не страдает от Global Interpreter Lock.
- Производительность обычно ниже, хотя это зависит от тестов.

Привет, мир

```
print "Hello World!"
```

Вы также можете использовать функции Java:

```
from java.lang import System
System.out.println("Hello World!")
```

внешняя ссылка

- Официальный веб-сайт
- Меркуриальный репозиторий

Transcrypt

Transcrypt - это инструмент для предварительного компиляции довольно обширного подмножества Python в компактный, читаемый Javascript. Он имеет следующие характеристики:

- Позволяет классическое программирование ОО с множественным наследованием с использованием чистого синтаксиса Python, проанализированного собственным парсером CPython
- Бесшовная интеграция со вселенной высококачественных веб-ориентированных библиотек JavaScript, а не настольных Python-приложений
- Иерархическая система на основе URL-адресов, позволяющая распределять модули через PyPi
- Простая связь между источником Python и сгенерированным кодом JavaScript для легкой отладки
- Многоуровневые исходные коды и необязательная аннотация целевого кода с исходными ссылками
- Компактные загрузки, скорее, чем kB, а не MB
- Оптимизированный код JavaScript, используя memoization (кеширование вызовов), чтобы опционально обходить цепочку поиска прототипа

- Перегрузка оператора может быть включена и выключена локально, чтобы облегчить считываемую численную математику

Размер и скорость кода

Опыт показал, что 650 kB исходного кода Python приблизительно переводится в том же количестве исходного кода JavaScript. Скорость соответствует скорости рукописного JavaScript и может превзойти ее, если включена перезапись вызова.

Интеграция с 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>
```

Интеграция с JavaScript и 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 ()
```

Интеграция с другими библиотеками JavaScript

Transcrypt может использоваться в сочетании с любой библиотекой JavaScript без специальных мер или синтаксиса. В примерах документации приведены для ао react.js, riot.js, fabric.js и node.js.

Связь между кодом Python и JavaScript

питон

```
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');
```

внешняя ссылка

- Официальный сайт: http://www.transcrypt.org/
- Репозиторий: https://github.com/JdeH/Transcrypt

Прочитайте Неофициальные реализации Python онлайн:
https://riptutorial.com/ru/python/topic/5225/неофициальные-реализации-python

глава 113: Несовместимость, перемещающаяся с Python 2 на Python 3

Вступление

В отличие от большинства языков, Python поддерживает две основные версии. С 2008 года, когда был выпущен Python 3, многие сделали переход, в то время как многие из них этого не сделали. Чтобы понять оба варианта, в этом разделе рассматриваются важные различия между Python 2 и Python 3.

замечания

В настоящее время существуют две поддерживаемые версии Python: 2.7 (Python 2) и 3.6 (Python 3). Кроме того, версии 3.3 и 3.4 получают обновления безопасности в исходном формате.

Python 2.7 совместим с предыдущими версиями Python и может запускать код Python из версий 1.x и 2.x Python без изменений. Он широко доступен с обширной коллекцией пакетов. Он также считается устаревшим разработчиком CPython и получает только защиту и исправление ошибок. Разработчики CPython намерены отказаться от этой версии языка в 2020 году .

Согласно Python Enhancement Proposal 373 , запланированные будущие выпуски Python 2 после 25 июня 2016 года не будут исправлены, но исправления ошибок и обновления безопасности будут поддерживаться до 2020 года. (В нем не указано, какой точной датой в 2020 году станет дата заката Python 2.)

Python 3 намеренно нарушил совместимость в обратном направлении, чтобы решить проблемы, с которыми языковые разработчики сталкивались с ядром языка. Python 3 получает новые разработки и новые функции. Это версия языка, которую разработчики языка намерены продвигать вперед.

За время между первоначальным выпуском Python 3.0 и текущей версией некоторые функции Python 3 были перенесены в Python 2.6, а другие части Python 3 были расширены, чтобы иметь синтаксис, совместимый с Python 2. Поэтому можно писать Python, который будет работать как на Python 2, так и на Python 3, используя будущие импорты и специальные модули (например, **шесть**).

Будущий импорт должен быть в начале вашего модуля:

```
from __future__ import print_function
# other imports and instructions go after __future__
```

```
print('Hello world')
```

Для получения дополнительной информации о модуле `__future__` см. Соответствующую страницу в документации Python .

Инструмент 2to3 - это программа Python, которая преобразует код Python 2.x в код Python 3.x, см. Также документацию Python .

В пакете 6 предусмотрены утилиты для совместимости с Python 2/3:

- унифицированный доступ к переименованным библиотекам
- переменные для типов string / unicode
- функции для метода, который был удален или был переименован

Ссылка на различия между Python 2 и Python 3 приведена здесь .

Examples

Операция печати и функции печати

В Python 2 `print` - это утверждение:

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)"
```

В Python 3 функция `print()` - это функция с аргументами ключевого слова для общего использования:

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)"
```

Функция печати имеет следующие параметры:

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

`sep` - это то, что отделяет объекты, которые вы передаете, для печати. Например:

```
print('foo', 'bar', sep='~') # out: foo~bar
print('foo', 'bar', sep='.') # out: foo.bar
```

`end` - это то, что следует за завершением инструкции print. Например:

```
print('foo', 'bar', end='!') # out: foo bar!
```

Повторная печать после завершающего оператора печати, не относящегося к новой строке, *будет* напечатана в той же строке:

```
print('foo', end='~')
print('bar')
# out: foo~bar
```

Примечание. Для будущей совместимости *функция* `print` также доступна в Python 2.6; однако его нельзя использовать, если разбор *инструкции* `print` не отключен

```
from __future__ import print_function
```

Эта функция имеет тот же формат, что и Python 3, за исключением того, что ему не хватает параметра `flush` .

См. PEP 3105 для обоснования.

Строки: байты против Unicode

Python 2.x 2.7

В Python 2 есть два варианта строки: те из байтов с типом (`str`) и те, что сделаны из текста с типом (`unicode`).

В Python 2 объект типа `str` всегда является байтовой последовательностью, но обычно используется как для текстовых, так и для двоичных данных.

Строковый литерал интерпретируется как байтовая строка.

```
s = 'Cafe'    # type(s) == str
```

Есть два исключения: вы можете явно определить *литерал Unicode (text)* , префиксного литерала с помощью `u` :

```
s = u'Café'   # type(s) == unicode
b = 'Lorem ipsum'  # type(b) == str
```

Кроме того, вы можете указать, что строковые литералы всего модуля должны создавать литералы Unicode (text):

```
from __future__ import unicode_literals

s = 'Café'    # type(s) == unicode
b = 'Lorem ipsum'  # type(b) == unicode
```

Чтобы проверить, является ли ваша переменная строкой (либо Unicode, либо байтовая строка), вы можете использовать:

```
isinstance(s, basestring)
```

Python 3.x 3.0

В Python 3 тип `str` - это текстовый тип Unicode.

```
s = 'Cafe'           # type(s) == str
s = 'Café'           # type(s) == str (note the accented trailing e)
```

Кроме того, Python 3 добавил объект `bytes` , подходящий для двоичных «blobs» или записи в независимые от кодирования файлы. Чтобы создать объект байтов, вы можете префикс `b` в строковый литерал или вызвать метод `encode` строки:

```
# Or, if you really need a byte string:
s = b'Cafe'          # type(s) == bytes
s = 'Café'.encode()  # type(s) == bytes
```

Чтобы проверить, является ли значение строкой, используйте:

```
isinstance(s, str)
```

Python 3.x 3.3

Также возможно префикс строковых литералов с префиксом `u` чтобы упростить совместимость между базами данных Python 2 и Python 3. Поскольку в Python 3 все строки по умолчанию Unicode, добавление строкового литерала с `u` имеет никакого эффекта:

```
u'Cafe' == 'Cafe'
```

Python 2 в сырце Unicode строки префикс `ur` не поддерживаются, однако:

```
>>> ur'Café'
  File "<stdin>", line 1
    ur'Café'
           ^
SyntaxError: invalid syntax
```

Обратите внимание, что вы должны `encode` объект Python 3 text (`str`), чтобы преобразовать его в представление `bytes` этого текста. Кодировка по умолчанию этого метода - UTF-8 .

Вы можете использовать `decode` чтобы задать объект `bytes` для текста Unicode, который он представляет:

```
>>> b.decode()
'Café'
```

Python 2.x 2.6

Хотя тип `bytes` существует как в Python 2, так и в 3, тип `unicode` существует только в Python 2. Чтобы использовать неявные строки Unicode Python 3 в Python 2, добавьте следующее в начало файла кода:

```
from __future__ import unicode_literals
print(repr("hi"))
# u'hi'
```

Python 3.x 3.0

Другое важное различие заключается в том, что индексирование байтов в Python 3 приводит к выводу `int` так:

```
b"abc"[0] == 97
```

В то время как разрезание в размере одного результата приводит к объекту длиной 1 байт:

```
b"abc"[0:1] == b"a"
```

Кроме того, Python 3 исправляет некоторые необычные действия с помощью unicode, т. Е. Обратные байтовые строки в Python 2. Например, разрешена следующая проблема :

```
# -*- 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
```

Целостный отдел

Стандартный **символ разделения** (`/`) работает по-разному в Python 3 и Python 2 при применении к целым числам.

При делении целого на другое целое число в Python 3 операция деления `x / y` представляет собой **истинное деление** (использует метод `__truediv__`) и производит результат с плавающей запятой. Между тем, та же операция в Python 2 представляет собой **классическое разделение,** которое округляет результат до отрицательной бесконечности (также известный как *слово*).

Например:

Код	Выход Python 2	Выход Python 3
`3 / 2`	1	1,5
`2 / 3`	0	0,6666666666666666
`-3 / 2`	-2	-1,5

Поведение округления к нулю было устарело в Python 2.2 , но осталось в Python 2.7 для обратной совместимости и было удалено в Python 3.

Примечание. Чтобы получить результат с *плавающей точкой* в Python 2 (без округления по полу), мы можем указать один из операндов с десятичной точкой. Вышеприведенный пример `2/3` который дает `0` в Python 2, должен использоваться как `2 / 3.0` или `2.0 / 3` или `2.0/3.0` для получения `0.6666666666666666`

Код	Выход Python 2	Выход Python 3
`3.0 / 2.0`	1,5	1,5
`2 / 3.0`	0,6666666666666666	0,6666666666666666
`-3.0 / 2`	-1,5	-1,5

Существует также **оператор деления пола** (`//`), который работает одинаково в обеих версиях: он округляется до ближайшего целого. (хотя float возвращается при использовании с float). В обеих версиях оператор `//` сопоставляется с `__floordiv__` .

Код	Выход Python 2	Выход Python 3
`3 // 2`	1	1
`2 // 3`	0	0
`-3 // 2`	-2	-2

Код	Выход Python 2	Выход Python 3
`3.0 // 2.0`	1,0	1,0
`2.0 // 3`	0.0	0.0
`-3 // 2.0`	-2,0	-2,0

В явном виде можно обеспечить прямое разделение или разделение полов с использованием собственных функций в модуле `operator` :

```
from operator import truediv, floordiv
assert truediv(10, 8) == 1.25            # equivalent to `/` in Python 3
assert floordiv(10, 8) == 1              # equivalent to `//`
```

Хотя ясное и ясное, использование операторских функций для каждого деления может быть утомительным. Частое изменение поведения оператора `/` часто бывает предпочтительным. Общей практикой является устранение типичного поведения деления путем добавления `from __future__ import division` в качестве первого оператора в каждом модуле:

```
# needs to be the first statement in a module
from __future__ import division
```

Код	Выход Python 2	Выход Python 3
`3 / 2`	1,5	1,5
`2 / 3`	0,6666666666666666	0,6666666666666666
`-3 / 2`	-1,5	-1,5

`from __future__ import division` гарантирует, что оператор `/` представляет истинное деление и только внутри модулей, которые содержат импорт `__future__` , поэтому нет никаких веских причин не включать его во все новые модули.

Примечание . Некоторые другие языки программирования используют *округление до нуля* (усечение), а не *округление до отрицательной бесконечности,* как это делает Python (т.е. на этих языках `-3 / 2 == -1`). Такое поведение может вызвать путаницу при переносе или сравнении кода.

Примечание по операндам float . В качестве альтернативы `from __future__ import division` можно использовать обычный символ разделения `/` и гарантировать, что хотя бы один из операндов является float: `3 / 2.0 == 1.5` . Однако это можно считать плохой практикой. Слишком просто написать `average = sum(items) / len(items)` и забыть использовать один из аргументов для float. Более того, такие случаи могут часто уклоняться от уведомления во

время тестирования, например, если вы тестируете массив, содержащий `float` s, но получаете массив `int` s в процессе производства. Кроме того, если тот же код используется в Python 3, программы, ожидающие, что 3/2 `3 / 2 == 1` будет True, будут работать неправильно.

См. PEP 238 для более подробного обоснования того, почему оператор разделения был изменен в Python 3 и почему следует избегать разделения в старом стиле.

См. Раздел « *Простая математика»* для получения дополнительной информации о разделении.

Сокращение больше не является встроенным

В Python 2 `reduce` доступно либо как встроенная функция, либо из пакета `functools` (начиная с версии 2.6), тогда как в Python 3 `reduce` доступно только из `functools` . Однако синтаксис `reduce` как в Python2, так и в Python3 одинаковый и `reduce(function_to_reduce, list_to_reduce)` .

В качестве примера рассмотрим сокращение списка до одного значения путем деления каждого из соседних чисел. Здесь мы используем функцию `truediv` из библиотеки `operator` .

В Python 2.x это просто:

Python 2.x 2.3

```
>>> my_list = [1, 2, 3, 4, 5]
>>> import operator
>>> reduce(operator.truediv, my_list)
0.008333333333333333
```

В Python 3.x пример становится немного сложнее:

Python 3.x 3.0

```
>>> my_list = [1, 2, 3, 4, 5]
>>> import operator, functools
>>> functools.reduce(operator.truediv, my_list)
0.008333333333333333
```

Мы также можем использовать `from functools import reduce` чтобы избежать `reduce` вызова с помощью имени пространства имен.

Различия между диапазонами и функциями xrange

В Python 2 функция `range` возвращает список, тогда как `xrange` создает специальный объект `xrange` , который является неизменной последовательностью, которая, в отличие от других встроенных типов последовательностей, не поддерживает срез и не имеет ни методов `index` ни `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
```

В Python 3 `xrange` был расширен до последовательности `range` , что теперь создает объект `range` . Нет типа `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
```

Кроме того, поскольку Python 3.2, `range` также поддерживает нарезку, `index` и `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
```

Преимущество использования специального типа последовательности вместо списка состоит в том, что интерпретатору не нужно выделять память для списка и заполнять его:

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(100000000000000000))
# Out: xrange(100000000000000000)
```

Поскольку последнее поведение обычно желательно, первое было удалено в Python 3.

Если вы все еще хотите иметь список в Python 3, вы можете просто использовать конструктор `list()` для объекта `range` :

Python 3.x 3.0

```
print(list(range(1, 10)))
# Out: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Совместимость

Чтобы поддерживать совместимость между версиями Python 2.x и Python 3.x, вы можете использовать `builtins` модуль из `future` внешнего пакета для достижения как *совместимости с переходом, так* и *обратной совместимости* :

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
```

`range` в `future` библиотеке поддерживает нарезку, `index` и `count` во всех версиях Python, как и встроенный метод на Python 3.2+.

Распаковка итераций

Python 3.x 3.0

В Python 3 вы можете распаковать итерацию, не зная точного количества элементов в ней и даже иметь переменную, удерживающую конец итерабельного. Для этого вы предоставляете переменную, которая может собирать список значений. Это делается путем размещения звездочки перед именем. Например, распаковка `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
```

Примечание . При использовании синтаксиса `*variable` `variable` всегда будет списком, даже если исходный тип не был списком. Он может содержать ноль или более элементов в зависимости от количества элементов в исходном списке.

```
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
```

Аналогично, распаковка `str` :

```
begin, *tail = "Hello"
print(begin)
# Out: 'H'
print(tail)
# Out: ['e', 'l', 'l', 'o']
```

Пример распаковки `date` ; _ Используется в этом примере в качестве переменной (холостой мы заинтересованы только в `year` стоимости):

```
person = ('John', 'Doe', (10, 16, 2016))
*_, (*_, year_of_birth) = person
print(year_of_birth)
# Out: 2016
```

Стоит отметить, что, поскольку `*` ест переменное количество элементов, вы не можете иметь два `*` s *для одного и того же итерабельного* в задании - он не будет знать, сколько элементов *входит* в первую распаковку, и сколько во втором :

```
*head, *tail = [1, 2]
# Out: SyntaxError: two starred expressions in assignment
```

Python 3.x 3.5

До сих пор мы обсуждали распаковку в заданиях. `*` и `**` были расширены в Python 3.5 . Теперь в одном выражении можно иметь несколько операций распаковки:

```
{*range(4), 4, *(5, 6, 7)}
# Out: {0, 1, 2, 3, 4, 5, 6, 7}
```

Python 2.x 2.0

Также можно распаковать итерируемый аргумент функции:

```
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

Распаковка словаря использует две соседние звезды `**` (PEP 448):

```
tail = {'y': 2, 'z': 3}
{'x': 1, **tail}
 # Out: {'x': 1, 'y': 2, 'z': 3}
```

Это позволяет как переопределять старые значения, так и объединять словари.

```
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 удаляет распаковку в функции. Следовательно, в Python 3 не работает следующее

```
# 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
```

См. PEP 3113 для подробного обоснования.

Поднятие и обработка исключений

Это синтаксис Python 2, обратите внимание на запятые `,` на `raise` и `except` строк:

Python 2.x 2.3

```
try:
    raise IOError, "input/output error"
except IOError, exc:
    print exc
```

В Python 3, `,` синтаксис отбрасывается и заменяется скобкой и в `as` ключевого слова:

```
try:
    raise IOError("input/output error")
except IOError as exc:
    print(exc)
```

Для обратной совместимости синтаксис Python 3 также доступен в Python 2.6, поэтому он должен использоваться для всего нового кода, который не должен быть совместим с предыдущими версиями.

Python 3.x 3.0

Python 3 также добавляет цепочку исключений , в которой вы можете указать, что *причиной* этого исключения является другое исключение. Например

```
try:
    file = open('database.db')
except FileNotFoundError as e:
    raise DatabaseError('Cannot open {}') from e
```

Исключение, возникающее в инструкции `except` имеет тип `DatabaseError` , но исходное исключение помечено как атрибут `__cause__` этого исключения. Когда отображается трассировка, исходное исключение также будет отображаться в 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')
```

Если вы выбрали `except` блок *без* явной цепочки:

```
try:
    file = open('database.db')
except FileNotFoundError as e:
    raise DatabaseError('Cannot open {}')
```

След

```
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

Ни один из них не поддерживается в Python 2.x; исходное исключение и его трассировка будут потеряны, если в исключающем блоке будет создано другое исключение. Следующий код может использоваться для совместимости:

```
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

Чтобы «забыть» ранее заброшенное исключение, используйте `raise from None`

```
try:
    file = open('database.db')
except FileNotFoundError as e:
    raise DatabaseError('Cannot open {}') from None
```

Теперь трассировка будет просто

```
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
DatabaseError('Cannot open database.db')
```

Или, чтобы сделать его совместимым с Python 2 и 3, вы можете использовать шесть пакетов следующим образом:

```
import six
try:
    file = open('database.db')
except FileNotFoundError as e:
    six.raise_from(DatabaseError('Cannot open {}'), None)
```

.next () для итераторов, переименованных

В Python 2 итератор может быть пройден с помощью метода, называемого `next` на самом итераторе:

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
```

В Python 3 метод `.next` был переименован в `.__next__` , признав свою «магическую» роль, поэтому вызов `.next` повысит `AttributeError` . Правильный способ доступа к этой функциональности как в Python 2, так и в Python 3 - это вызов `next` *функции* с итератором в качестве аргумента.

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
```

Этот код переносится в разных версиях от версии 2.6 до последних версий.

Сравнение различных типов

Python 2.x 2.3

Можно сравнить объекты разных типов. Результаты являются произвольными, но непротиворечивыми. Они упорядочены так, что `None` меньше, чем что-либо еще, числовые типы меньше, чем нечисловые типы, а все остальное упорядочено лексикографически по типу. Таким образом, `int` меньше, чем `str` а `tuple` больше, чем `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
```

Первоначально это было сделано, поэтому список смешанных типов можно было отсортировать, и объекты будут сгруппированы по типу:

```
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

Исключение возникает при сравнении разных (нечисловых) типов:

```
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()
```

Чтобы отсортировать смешанные списки в Python 3 по типам и добиться совместимости между версиями, вы должны предоставить ключ к отсортированной функции:

```
>>> 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}]
```

Использование `str` в качестве `key` функции временно преобразует каждый элемент в строку только для целей сравнения. Затем он видит строковое представление, начиная с `[` , `'` , `{` или `0-9` и он может сортировать эти (и все следующие символы).

Вход пользователя

В Python 2 пользовательский ввод принимается с использованием функции `raw_input` ,

Python 2.x 2.3

```
user_input = raw_input()
```

В то время как в Python 3 пользовательский ввод принимается с использованием функции `input` .

Python 3.x 3.0

```
user_input = input()
```

В Python 2 `input` функция принимает вход и *интерпретирует* его. Хотя это может быть полезно, оно имеет несколько соображений безопасности и было удалено в Python 3. Для доступа к той же функциональности можно использовать `eval(input())` .

Чтобы сохранить сценарий переносимым по двум версиям, вы можете поместить код ниже в верхней части своего сценария Python:

```
try:
    input = raw_input
except NameError:
    pass
```

Изменения в словаре

В Python 3 многие из методов словаря отличаются поведением от Python 2, и многие также были удалены: `has_key` , `iter*` и `view*` исчезли. Вместо `d.has_key(key)` , который давно устарел, теперь нужно использовать `key in d` .

В Python 2 словарные `keys` методов, `values` и `items` возвращают списки. В Python 3 вместо

этого они возвращают объекты *вида* ; объекты представления не являются итераторами, и они отличаются от них двумя способами, а именно:

- они имеют размер (на них можно использовать функцию `len`)
- они могут повторяться много раз

Кроме того, как итераторы, изменения в словаре отражаются в объектах представления.

Python 2.7 поддерживает эти методы с Python 3; они доступны как `viewkeys` , `viewvalues` и `viewitems` . Чтобы преобразовать код Python 2 в код Python 3, соответствующие формы:

- `d.keys()` , `d.values()` и `d.items()` Python 2 должны быть изменены в `list(d.keys())` , `list(d.values())` и `list(d.items())`
- `d.iterkeys()` , `d.itervalues()` и `d.iteritems()` должны быть изменены на `iter(d.keys())` или даже лучше, `iter(d)` ; `iter(d.values())` и `iter(d.items())` соответственно
- и, наконец, методы Python 2.7 вызовы `d.viewkeys()` , `d.viewvalues()` и `d.viewitems()` могут быть заменены на `d.keys()` , `d.values()` и `d.items()` .

Портирование кода Python 2, который *выполняет итерации* по словарным клавишам, значениям или элементам при их мутации, иногда бывает сложным. Рассматривать:

```
d = {'a': 0, 'b': 1, 'c': 2, '!': 3}
for key in d.keys():
    if key.isalpha():
        del d[key]
```

Код выглядит так, как если бы он работал аналогично в Python 3, но там метод `keys` возвращает объект вида, а не список, и если словарь изменяет размер при повторении, код Python 3 будет сбой с `RuntimeError: dictionary changed size during iteration` . Разумеется, решение должно правильно записываться `for key in list(d)` .

Аналогично, объекты просмотра ведут себя иначе, чем итераторы: нельзя использовать `next()` для них, и нельзя *возобновить* итерацию; он вместо этого перезапустится; если код Python 2 передает возвращаемое значение `d.iterkeys()` , `d.itervalues()` или `d.iteritems()` в метод, который ожидает итератор вместо *итерабельного* , то это должно быть `iter(d)` , `iter(d.values())` или `iter(d.items())` в Python 3.

Оператор exec является функцией в Python 3

В Python 2 `exec` - это оператор со специальным синтаксисом: `exec code [in globals[, locals]].` В Python 3 `exec` теперь есть функция: `exec(code, [, globals[, locals]])` , а синтаксис Python 2 поднимет `SyntaxError` .

Поскольку `print` была изменена из оператора в функцию, был `__future__` импорт `__future__` . Тем не менее, нет `from __future__ import exec_function` , поскольку он не нужен: оператор

exec в Python 2 также может использоваться с синтаксисом, который выглядит точно так же, как вызов функции `exec` в Python 3. Таким образом, вы можете изменить утверждения

Python 2.x 2.3

```
exec 'code'
exec 'code' in global_vars
exec 'code' in global_vars, local_vars
```

к формам

Python 3.x 3.0

```
exec('code')
exec('code', global_vars)
exec('code', global_vars, local_vars)
```

и последние формы гарантированно работают одинаково как в Python 2, так и в Python 3.

Ошибка функции hasattr в Python 2

В Python 2, когда свойство `hasattr` ошибку, `hasattr` игнорирует это свойство, возвращая `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
```

Эта ошибка исправлена в Python3. Поэтому, если вы используете Python 2, используйте

```
try:
    a.get
except AttributeError:
    print("no get property!")
```

или вместо этого использовать `getattr`

```
p = getattr(a, "get", None)
if p is not None:
    print(p)
else:
    print("no get property!")
```

Переименованные модули

Несколько модулей в стандартной библиотеке были переименованы:

Старое название	Новое имя
_winreg	WinREG
ConfigParser	ConfigParser
copy_reg	copyreg
Очередь	очередь
SocketServer	SocketServer
_markupbase	markupbase
магнезии	reprlib
test.test_support	test.support
Tkinter	Tkinter
tkFileDialog	tkinter.filedialog
urllib / urllib2	urllib, urllib.parse, urllib.error, urllib.response, urllib.request, urllib.robotparser

Некоторые модули даже были преобразованы из файлов в библиотеки. Возьмите tkinter и urllib сверху в качестве примера.

Совместимость

Поддерживая совместимость между версиями Python 2.x и 3.x, вы можете использовать `future` внешний пакет, чтобы включить импорт стандартных пакетов стандартного пакета с именами Python 3.x в версиях Python 2.x.

Октальные константы

В Python 2 восьмой литерал можно определить как

```
>>> 0755  # only Python 2
```

Чтобы обеспечить кросс-совместимость, используйте

```
0o755  # both Python 2 and Python 3
```

Все классы являются «классами нового стиля» в Python 3.

В Python `3.x` все классы являются классами *нового стиля* ; при определении нового класса python неявно делает его наследуемым от `object` . Таким образом, указание `object` в определении `class` является полностью необязательным:

Python 3.x 3.0

```
class X: pass
class Y(object): pass
```

Оба этих класса теперь содержат `object` в своем `mro` (порядок разрешения метода):

Python 3.x 3.0

```
>>> X.__mro__
(__main__.X, object)

>>> Y.__mro__
(__main__.Y, object)
```

В классах Python `2.x` по умолчанию используются классы старого стиля; они не неявно наследуют от `object` . Это заставляет семантику классов различаться в зависимости от того, если мы явно добавляем `object` в качестве базового `class` :

Python 2.x 2.3

```
class X: pass
class Y(object): pass
```

В этом случае, если мы попытаемся напечатать `__mro__` of `Y` , `__mro__` аналогичный вывод, как в случае с Python `3.x` :

Python 2.x 2.3

```
>>> Y.__mro__
(<class '__main__.Y'>, <type 'object'>)
```

Это происходит потому, что мы явно наследовали `Y` от объекта при его определении: `class Y(object): pass` . Для класса `X` который *не* наследуется от объекта, атрибут `__mro__` не

существует, попытка доступа к нему приводит к `__mro__` `AttributeError` .

Чтобы **обеспечить совместимость** между обеими версиями Python, классы могут быть определены с `object` в качестве базового класса:

```
class mycls(object):
    """I am fully compatible with Python 2/3"""
```

В качестве альтернативы, если переменная `__metaclass__` задана для `type` в глобальной области видимости, все последующие классы в данном модуле являются неявно новыми, не требуя явно наследовать от `object` :

```
__metaclass__ = type

class mycls:
    """I am also fully compatible with Python 2/3"""
```

Удаленные операторы <> и ``, синонимичные с! = И repr ()

В Python 2 `<>` является синонимом для `!=` ; Аналогично, `` `foo` `` является синонимом для `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
```

кодирование / декодирование в hex больше недоступно

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'
```

Однако, как было предложено сообщением об ошибке, вы можете использовать модуль `codecs` для достижения того же результата:

```
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'
```

Обратите внимание, что `codecs.encode` возвращает объект `bytes` . Чтобы получить объект `str` просто `decode` в ASCII:

```
codecs.encode(b'\x1d\xea\xdb\xee\xff', 'hex').decode('ascii')
# Out: '1deadbeeff'
```

Функция cmp удалена в Python 3

В Python 3 была удалена встроенная функция `cmp` вместе со специальным методом `__cmp__` .

Из документации:

> Функция `cmp()` следует рассматривать как ушедшую, а специальный метод `__cmp__()` больше не поддерживается. Используйте `__lt__()` для сортировки `__eq__()` с `__hash__()` и другими богатыми сравнениями по мере необходимости. (Если вам действительно нужна функция `cmp()` , вы можете использовать выражение `(a > b) - (a < b)` как эквивалент для `cmp(a, b)` .)

Более того, все встроенные функции, которые принимают параметр `cmp` теперь принимают только параметр ключевого `key` слова.

В `functools` модуле есть также полезная функция `cmp_to_key(func)` , которая позволяет конвертировать из `cmp` функции -style к `key` -style функции:

> Преобразуйте функцию сравнения старого стиля в ключевую функцию. Используется с инструментами, которые принимают ключевые функции (такие как `sorted()` , `min()` , `max()` , `heapq.nlargest()` , `heapq.nsmallest()` , `itertools.groupby()`). Эта функция в основном используется в качестве инструмента перехода для программ, преобразованных из Python 2, которые поддерживают использование функций сравнения.

Исключенные переменные в понимании списка

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!'
```

Как видно из примера, в Python 2 значение `x` просочилось: оно замаскировало `hello world!` и распечатал `U` , так как это было последним значением `x` когда цикл закончился.

Тем не менее, в Python 3 `x` печатает первоначально определенный `hello world!` , так как локальная переменная из понимания списка не маскирует переменные из окружающей области.

Кроме того, ни одно из выражений генератора (доступно в Python начиная с версии 2.5), а также словарные или установочные представления (которые были переданы Python 2.7 из Python 3) в Python 2.

Обратите внимание, что в обоих Python 2 и Python 3 переменные будут просачиваться в окружающую область при использовании цикла for:

```
x = 'hello world!'
vowels = []
for x in 'AEIOU':
    vowels.append(x)
print(x)
# Out: 'U'
```

карта()

`map()` является встроенным, который полезен для применения функции к элементам итерации. В Python 2 `map` возвращает список. В Python 3 `map` возвращает *объект карты* , который является генератором.

```
# 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']
```

В Python 2 вы можете передать `None` чтобы служить функцией идентификации. Это больше не работает в 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
```

Более того, при передаче более одного итерабельного аргумента в Python 2, накладки `map` короче итераций с `None` (аналогично `itertools.izip_longest`). В Python 3 итерация останавливается после кратчайшего повторения.

В 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)]
```

В 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)]
```

Примечание . Вместо `map` рассмотрите использование списков, совместимых с Python 2/3. Замена `map(str, [1, 2, 3, 4, 5])` :

```
>>> [str(i) for i in [1, 2, 3, 4, 5]]
['1', '2', '3', '4', '5']
```

filter (), map () и zip () возвращают итераторы вместо последовательностей

Python 2.x 2.7

В `filter` Python 2, встроенные функции `map` и `zip` возвращают последовательность. `map` и `zip` всегда возвращают список, а с `filter` тип возврата зависит от типа заданного параметра:

```
>>> 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

В Python 3 `filter` , `map` и `zip` обратный итератор вместо:

```
>>> 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)]
```

Поскольку Python 2 `itertools.izip` эквивалентен Python 3, `zip` `izip` был удален на Python 3.

Абсолютный / относительный импорт

В Python 3 PEP 404 изменяет способ работы импорта с Python 2. *Имплицитный относительный* импорт больше не разрешен в пакетах и `from ... import *` импорт разрешен только в модульном уровне кода.

Чтобы достичь поведения Python 3 в Python 2:

- функция абсолютного импорта может быть включена с `from __future__ import absolute_import`
- *явный относительный* импорт поощряется вместо *имплицитного относительного* импорта

Для пояснения в Python 2 модуль может импортировать содержимое другого модуля, расположенного в том же каталоге, что и ниже:

```
import foo
```

Обратите внимание, что местоположение `foo` неоднозначно из оператора импорта. Этот тип неявного относительного импорта, таким образом, обескуражен в пользу явного относительного импорта , который выглядит следующим образом:

```
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
```

Точка `.` позволяет явное объявление местоположения модуля в дереве каталогов.

Подробнее о относительном импорте

Рассмотрим некоторый пользовательский пакет, называемый `shapes` . Структура каталогов выглядит следующим образом:

```
shapes
├── __init__.py
```

```
|
├── circle.py
|
├── square.py
|
└── triangle.py
```

`circle.py` , `square.py` и `triangle.py` все import `util.py` в качестве модуля. Как они будут ссылаться на модуль на одном уровне?

```
from . import util # use util.PI, util.sq(x), etc
```

ИЛИ ЖЕ

```
from .util import * #use PI, sq(x), etc to call functions
```

`.` используется для относительного импорта на уровне одного уровня.

Теперь рассмотрим альтернативный макет модуля `shapes` :

```
shapes
├── __init__.py
|
├── circle
│   ├── __init__.py
│   └── circle.py
|
├── square
│   ├── __init__.py
│   └── square.py
|
├── triangle
│   ├── __init__.py
│   ├── triangle.py
|
└── util.py
```

Теперь, как эти 3 класса ссылаются на util.py?

```
from .. import util # use util.PI, util.sq(x), etc
```

ИЛИ ЖЕ

```
from ..util import * # use PI, sq(x), etc to call functions
```

The `..` используется для относительного импорта родительского уровня. Добавить еще `.` s с количеством уровней между родительским и дочерним.

Файловый ввод-вывод

`file` больше не является встроенным именем в 3.x (`open` still works).

Внутренние данные ввода-вывода файлов были перенесены в стандартный библиотечный модуль `io` , который также является новым домом для `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'
```

Режим файла (text vs binary) теперь определяет тип данных, полученных при чтении файла (и типа, необходимого для записи):

```
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
```

Кодировка для текстовых файлов по умолчанию соответствует тому, что возвращается `locale.getpreferredencoding(False)` . Чтобы явно указать кодировку, используйте параметр ключевого слова для `encoding` :

```
with open('old_japanese_poetry.txt', 'shift_jis') as text:
    haiku = text.read()
```

Функция round () для циклического размыкания и возврата

раунд ()

В Python 2, используя `round()` для числа, равного близкому к двум целым числам, вернется один из самых удаленных от 0. Например:

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
```

Однако в Python 3 `round()` вернет четное целое число (например , *округление банкиров*). Например:

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
```

Функция round () следует стратегии от *половины до четного округления* , которая будет округлять половинные числа до ближайшего четного целого числа (например, `round(2.5)` теперь возвращает 2, а не 3.0).

Согласно ссылке в Википедии , это также известно как *беспристрастное округление* , *округленное округление* , *округление статистики* , *голландское округление* , *гауссово округление* или *нечетное округление* .

Половина и округление - это часть стандарта IEEE 754, а также режим округления по умолчанию в Microsoft .NET.

Эта стратегия округления имеет тенденцию уменьшать общую ошибку округления. Так как в среднем количество округленных чисел совпадает с количеством округленных чисел, ошибки округления сокращаются. Другие методы округления вместо этого имеют тенденцию иметь отклонение вверх или вниз в средней ошибке.

круглый () тип возврата

Функция `round()` возвращает тип `float` в Python 2.7

Python 2.x 2.7

```
round(4.8)
# 5.0
```

Начиная с Python 3.0, если второй аргумент (число цифр) опущен, он возвращает `int` .

Python 3.x 3.0

```
round(4.8)
# 5
```

Верно, ложно и нет

В Python 2, `True` , `False` и `None` есть встроенные константы. Это означает, что можно переназначить их.

Python 2.x 2.0

```
True, False = False, True
True   # False
False  # True
```

Вы не можете сделать это с помощью `None` с Python 2.4.

Python 2.x 2.4

```
None = None  # SyntaxError: cannot assign to None
```

В Python 3, `True` , `False` и `None` теперь используются ключевые слова.

Python 3.x 3.0

```
True, False = False, True  # SyntaxError: can't assign to keyword

None = None  # SyntaxError: can't assign to keyword
```

Возвращаемое значение при записи в файл-объект

В Python 2 запись непосредственно в дескриптор файла возвращает `None` :

Python 2.x 2.3

```
hi = sys.stdout.write('hello world\n')
# Out: hello world
type(hi)
# Out: <type 'NoneType'>
```

В Python 3 запись в дескриптор возвращает количество символов, записанных при записи текста, и количество байтов, записанных при написании байтов:

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

В Python 2 любое целое число, большее, чем C `ssize_t` , будет преобразовано в `long` тип данных, обозначенный суффиксом `L` в литерале. Например, в 32-битной сборке Python:

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
```

Однако в Python 3 был удален `long` тип данных; независимо от того, насколько велика целое число, это будет `int` .

Python 3.x 3.0

```
2**1024
# Output:
1797693134862315907729305190789024733617976978942306572734300811577326758055009631327084773224075360211

print(-(2**1024))
# Output: -
1797693134862315907729305190789024733617976978942306572734300811577326758055009631327084773224075360211

type(2**1024)
# Output: <class 'int'>
```

Класс Boolean Value

Python 2.x 2.7

В Python 2, если вы хотите самостоятельно определить логическое значение класса, вам необходимо реализовать метод `__nonzero__` в вашем классе. Значение по умолчанию равно True.

```
class MyClass:
    def __nonzero__(self):
        return False

my_instance = MyClass()
print bool(MyClass)       # True
print bool(my_instance)   # False
```

Python 3.x 3.0

В Python 3, `__bool__` используется вместо `__nonzero__`

```
class MyClass:
    def __bool__(self):
        return False

my_instance = MyClass()
print(bool(MyClass))       # True
print(bool(my_instance))   # False
```

Прочитайте Несовместимость, перемещающаяся с Python 2 на Python 3 онлайн: https://riptutorial.com/ru/python/topic/809/несовместимость--перемещающаяся-с-python-2-на-python-3

глава 114: Область переменных и привязка

Синтаксис

- глобальные a, b, c
- нелокальный a, b
- x = что-то # связывает x
- (x, y) = что-то # связывает x и y
- x + = что-то # связывает x. Аналогично для всех остальных "op ="
- del x # binds x
- для x в чем-то: # binds x
- с чем-то как x: # binds x
- кроме Exception как ex: # связывает ex внутренний блок

Examples

Глобальные переменные

В Python переменные внутри функций считаются локальными тогда и только тогда, когда они появляются в левой части оператора присваивания или в каком-либо другом случае привязки; в противном случае такая привязка просматривается в приложениях, вплоть до глобальной области. Это верно, даже если оператор присваивания никогда не выполняется.

```
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
```

Обычно присваивание внутри области видимости будет теневой для любых внешних переменных с тем же именем:

```
x = 'Hi'

def change_local_x():
    x = 'Bye'
    print(x)
change_local_x()  # prints Bye
print(x)  # prints Hi
```

Объявление `global` имени означает, что для остальной части области все назначения имени будут выполняться на верхнем уровне модуля:

```
x = 'Hi'

def change_global_x():
    global x
    x = 'Bye'
    print(x)

change_global_x()  # prints Bye
print(x)  # prints Bye
```

`global` ключевое слово означает, что назначения будут выполняться на верхнем уровне модуля, а не на верхнем уровне программы. Другие модули по-прежнему нуждаются в обычном пунктирном доступе к переменным в модуле.

Подводя итог: чтобы узнать, является ли переменная `x` локальной для функции, вы должны прочитать *всю* функцию:

1. если вы нашли `global x` , то `x` является **глобальной** переменной
2. Если вы нашли `nonlocal x` , то `x` принадлежит закрывающей функции и не является ни локальной, ни глобальной
3. Если вы нашли `x = 5` или `for x in range(3)` или какой-либо другой привязке, то `x` является **локальной** переменной
4. В противном случае `x` относится к некоторой охватывающей области (область действия, глобальная область или встроенные функции)

Локальные переменные

Если имя *связано* внутри функции, оно по умолчанию доступно только внутри функции:

```
def foo():
    a = 5
    print(a) # ok

print(a) #  NameError: name 'a' is not defined
```

Конструкции потока управления не влияют на область действия (за исключением `except`),

но доступ к переменной, которая еще не была назначена, является ошибкой:

```
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
```

Обычными операциями связывания являются назначения, `for` циклов и расширенные назначения, такие как `a += 5`

Нелокальные переменные

Python 3.x 3.0

Python 3 добавил новое ключевое слово, называемое **нелокальным** . Нелокальное ключевое слово добавляет переопределение области к внутренней области. Вы можете прочитать все об этом в PEP 3104 . Это лучше всего иллюстрируется несколькими примерами кода. Одним из наиболее распространенных примеров является создание функции, которая может увеличиваться:

```
def counter():
    num = 0
    def incrementer():
        num += 1
        return num
    return incrementer
```

Если вы попытаетесь запустить этот код, вы получите **UnboundLocalError,** потому что переменная **num** ссылается прежде, чем она будет назначена в самой внутренней функции. Давайте добавим нелокальное в микс:

```
def counter():
    num = 0
    def incrementer():
        nonlocal num
        num += 1
        return num
    return incrementer

c = counter()
c() # = 1
c() # = 2
c() # = 3
```

В основном `nonlocal` позволяет вам назначать переменные во внешней области, но не глобальную область. Таким образом, вы не можете использовать `nonlocal` в нашей функции

`counter` потому что тогда он попытается присвоить глобальную область. Попробуйте, и вы быстро получите `SyntaxError` . Вместо этого вы должны использовать `nonlocal` во вложенной функции.

(Обратите внимание, что функциональность, представленная здесь, лучше реализована с использованием генераторов.)

Привязка

```
x = 5
x += 7
for x in iterable: pass
```

Каждое из вышеуказанных утверждений является *вложением привязки* - `x` становится привязанным к объекту, обозначенному символом `5` . Если это утверждение появляется внутри функции, то `x` будет функция локального по умолчанию. См. Раздел «Синтаксис» для списка операторов привязки.

Функции пропускают область класса при поиске имен

Классы имеют локальную область видимости во время определения, но функции внутри класса не используют эту область при поиске имен. Поскольку lambdas - это функции, а понимание реализуется с использованием области функций, это может привести к неожиданному поведению.

```
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
```

Пользователи, незнакомые с тем, как работает эта область действия, могут ожидать `b` , `c` и `e` для печати `class` .

От PEP 227 :

> Имена в классе не доступны. Имена разрешаются в самой внутренней области приложения. Если определение класса происходит в цепочке вложенных областей, процесс разрешения пропускает определения классов.

Из документации Python по именованию и привязке :

> Объем имен, определенных в блоке класса, ограничен блоком класса; он не распространяется на кодовые блоки методов - это включает в себя выражения и выражения генератора, поскольку они реализованы с использованием области функций. Это означает, что следующее:
>
> ```
> class A:
>     a = 42
>     b = list(a + i for i in range(10))
> ```

В этом примере используются ссылки из этого ответа Martijn Pieters, в котором содержится более подробный анализ этого поведения.

Команда del

Эта команда имеет несколько взаимосвязанных, но разных форм.

del v

Если `v` - переменная, команда `del v` удаляет переменную из ее области. Например:

```
x = 5
print(x) # out: 5
del x
print(x) # NameError: name 'f' is not defined
```

> Обратите внимание, что `del` - это *событие привязки* , а это означает, что если явно не указано иначе (используя `nonlocal` или `global`), `del v` сделает `v` локальным в текущей области. Если вы хотите удалить `v` во внешней области, используйте `nonlocal v` или `global v` в той же области действия инструкции `del v` .

Во всем следующем, намерение команды является поведением по умолчанию, но не выполняется на языке. Класс может быть написан таким образом, что это недействительно.

del v.name

Эта команда вызывает вызов `v.__delattr__(name)` .

Цель состоит в том, чтобы сделать `name` атрибута недоступным. Например:

```
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]`

Эта команда вызывает вызов `v.__delitem__(item)` .

Цель состоит в том, что `item` не будет принадлежать отображению, реализуемому объектом `v` . Например:

```
x = {'a': 1, 'b': 2}
del x['a']
print(x) #  out: {'b': 2}
print(x['a']) # error: KeyError: 'a'
```

`del v[a:b]`

Это фактически вызывает `v.__delslice__(a, b)` .

Цель аналогична описанной выше, но с фрагментами - диапазонами элементов вместо одного элемента. Например:

```
x = [0, 1, 2, 3, 4]
del x[1:3]
print(x) #  out: [0, 3, 4]
```

См. Также Сбор мусора # Команда del .

Локальная и глобальная область

Каковы местные и глобальные масштабы?

Все переменные Python, доступные в некоторой точке кода, находятся либо в *локальной области,* либо в *глобальной области* .

Объяснение заключается в том, что локальная область включает все переменные, определенные в текущей функции, а глобальная область включает переменные, определенные за пределами текущей функции.

```
foo = 1  # global

def func():
    bar = 2  # local
    print(foo)  # prints variable foo from global scope
    print(bar)  # prints variable bar from local scope
```

Можно проверить, какие переменные находятся в этой области. Встроенные функции `locals()` и `globals()` возвращают целые области в качестве словарей.

```
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
```

Что происходит с именами?

```
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
```

Чтобы изменить глобальную переменную, используйте ключевое слово `global` :

```
foo = 1

def func():
    global foo
    foo = 2  # this modifies the global foo, rather than creating a local variable
```

Объем определяется для всего тела функции!

Это означает, что переменная никогда не будет глобальной для половины функции, а затем локальной или наоборот.

```
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)
```

Аналогичным образом, 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
```

Функции внутри функций

Внутри функций может быть много уровней функций, но внутри любой одной функции есть только одна локальная область для этой функции и глобальной области. Нет промежуточных областей.

```
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` или `nonlocal` (только для Python 3)

Оба этих ключевых слова используются для получения доступа на запись к переменным,

которые не являются локальными для текущих функций.

`global` ключевое слово объявляет, что имя должно рассматриваться как глобальная переменная.

```
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
```

С другой стороны, `nonlocal` (см. Nonlocal Variables), доступные в Python 3, берут *локальную* переменную из охватывающей области в локальную область текущей функции.

Из документации Python по `nonlocal` :

> Нелокальный оператор заставляет перечисленные идентификаторы ссылаться на ранее связанные переменные в ближайшей охватывающей области, исключая глобальные переменные.

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!
```

Прочитайте Область переменных и привязка онлайн: https://riptutorial.com/ru/python/topic/263/область-переменных-и-привязка

глава 115: Общие проблемы

Вступление

Python - это язык, который должен быть понятным и читаемым без каких-либо двусмысленностей и неожиданного поведения. К сожалению, эти цели не достижимы во всех случаях, поэтому Python имеет несколько угловых случаев, где он может делать что-то другое, чем ожидалось.

В этом разделе будут показаны некоторые проблемы, которые могут возникнуть при написании кода Python.

Examples

Изменение последовательности, которую вы повторяете

Цикл `for` выполняет итерацию по последовательности, поэтому **изменение этой последовательности внутри цикла может привести к неожиданным результатам** (особенно при добавлении или удалении элементов):

```
alist = [0, 1, 2]
for index, value in enumerate(alist):
    alist.pop(index)
print(alist)
# Out: [1]
```

Примечание: `list.pop()` используется для удаления элементов из списка.

Второй элемент не был удален, так как итерация идет по индексам по порядку. Вышеупомянутый цикл повторяется дважды, со следующими результатами:

```
# 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]
```

Эта проблема возникает из-за того, что индексы изменяются, итерации в направлении возрастания индекса. Чтобы избежать этой проблемы, вы можете **перебирать петлю в обратном направлении** :

```
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]
```

Итерацией по циклу, начинающимся с конца, поскольку элементы удаляются (или добавляются), это не влияет на индексы элементов ранее в списке. Таким образом, этот пример будет корректно удалять все элементы, которые даже с `alist` .

Аналогичная проблема возникает при **вставке или добавлении элементов в список, который вы повторяете** , что может привести к бесконечному циклу:

```
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]
```

Без условия `break` цикл будет вставлять `'a'` пока компьютер не исчерпывает память, и программе разрешено продолжить. В такой ситуации обычно предпочтительнее создавать новый список и добавлять элементы в новый список по мере того, как вы просматриваете исходный список.

При использовании цикла `for` **вы не можете изменять элементы списка с помощью переменной-заполнителя** :

```
alist = [1,2,3,4]
for item in alist:
    if item % 2 == 0:
        item = 'even'
print(alist)
# Out: [1,2,3,4]
```

В приведенном выше примере **изменение `item` ничего не меняет в исходном списке** . Вам нужно использовать индекс списка (`alist[2]`), и `enumerate()` работает хорошо для этого:

```
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']
```

В `while` **цикл** может быть лучшим выбором в некоторых случаях:

Если вы собираетесь **удалить все элементы** в списке:

```
zlist = [0, 1, 2]
while zlist:
    print(zlist[0])
    zlist.pop(0)
print('After: zlist =', zlist)

# Out: 0
#      1
#      2
# After: zlist = []
```

Хотя простой сброс `zlist` приведет к `zlist` же результату;

```
zlist = []
```

Вышеприведенный пример также можно объединить с `len()` для остановки после определенной точки или для удаления всех элементов, кроме `x` , в списке:

```
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]
```

Или **прокручивать список при удалении элементов, удовлетворяющих определенному условию** (в этом случае удаление всех четных элементов):

```
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]
```

Обратите внимание, что вы не увеличиваете `i` после удаления элемента. `zlist[i]` элемент в `zlist[i]` , индекс следующего элемента уменьшился на единицу, поэтому, проверив `zlist[i]` с тем же значением для `i` на следующей итерации, вы будете правильно проверять следующий элемент в списке ,

---

Непосредственный способ подумать об удалении нежелательных элементов из списка - **добавить нужные элементы в новый список** . Следующий пример является альтернативой последнего в `while` , например петли:

```
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]
```

Здесь мы собираем желаемые результаты в новый список. Затем мы можем переназначить временный список исходной переменной.

С этой тенденцией мышления вы можете вызывать одну из самых элегантных и мощных функций Python, **список понятий** , которые устраняют временные списки и расходятся с ранее обсуждавшейся идеологией мувирования списка / индекса.

```
zlist = [1,2,3,4,5]
[item for item in zlist if item % 2 != 0]
# Out: [1, 3, 5]
```

Mutable аргумент по умолчанию

```
def foo(li=[]):
    li.append(1)
    print(li)

foo([2])
# Out: [2, 1]
foo([3])
# Out: [3, 1]
```

Этот код ведет себя так, как ожидалось, но что, если мы не передадим аргумент?

```
foo()
# Out: [1] As expected...

foo()
# Out: [1, 1]  Not as expected...
```

Это объясняется тем, что аргументы по умолчанию и методы вычисляются во время **определения,** а не времени выполнения. Поэтому у нас только один экземпляр списка `li` .

Способ обойти это использовать только неизменные типы для аргументов по умолчанию:

```
def foo(li=None):
    if not li:
```

```
        li = []
    li.append(1)
    print(li)

foo()
# Out: [1]

foo()
# Out: [1]
```

В то время как улучшение, и хотя, `if not li` правильно оценивает значение `False` , многие другие объекты также работают, например, последовательности нулевой длины. Следующие примеры аргументов могут привести к непредвиденным результатам:

```
x = []
foo(li=x)
# Out: [1]

foo(li="")
# Out: [1]

foo(li=0)
# Out: [1]
```

Идиоматический подход заключается в прямом проверке аргумента против объекта `None` :

```
def foo(li=None):
    if li is None:
        li = []
    li.append(1)
    print(li)

foo()
# Out: [1]
```

Перемножение списков и общие ссылки

Рассмотрим случай создания вложенной структуры списка путем умножения:

```
li = [[]] * 3
print(li)
# Out: [[], [], []]
```

На первый взгляд мы думаем, что у нас есть список из 3 разных вложенных списков. Попробуем добавить `1` к первому:

```
li[0].append(1)
print(li)
# Out: [[1], [1], [1]]
```

`1` добавлен ко всем спискам в `li` .

Причина в том, что `[[]] * 3` не создает `list` из 3 разных `list` s. Скорее, он создает `list` содержащий 3 ссылки на один и тот же объект `list` . Таким образом, когда мы добавляем к `li[0]` изменение видимо во всех подэлементах `li` . Это эквивалентно:

```
li = []
element = [[]]
li = element + element + element
print(li)
# Out: [[], [], []]
element.append(1)
print(li)
# Out: [[1], [1], [1]]
```

Это может быть дополнительно подтверждено, если мы печатаем адреса памяти содержащегося `list` с помощью `id` :

```
li = [[]] * 3
print([id(inner_list) for inner_list in li])
# Out: [6830760, 6830760, 6830760]
```

Решением является создание внутренних списков с помощью цикла:

```
li = [[] for _ in range(3)]
```

Вместо того, чтобы создавать один `list` и затем делать 3 ссылки на него, мы теперь создаем 3 разных отдельных списка. Это, опять же, можно проверить, используя функцию `id` :

```
print([id(inner_list) for inner_list in li])
# Out: [6331048, 6331528, 6331488]
```

Вы также можете это сделать. Это приводит к созданию нового пустого списка в каждом `append` .

```
>>> li = []
>>> li.append([])
>>> li.append([])
>>> li.append([])
>>> for k in li: print(id(k))
...
4315469256
4315564552
4315564808
```

Не используйте индекс для перебора последовательности.

Не рекомендуется **:**

```
for i in range(len(tab)):
    print(tab[i])
```

Сделайте :

```
for elem in tab:
    print(elem)
```

`for` будет автоматизировать большинство операций итерации для вас.

Используйте перечисление, если вам действительно нужен как индекс, так и элемент .

```
for i, elem in enumerate(tab):
     print((i, elem))
```

Будьте осторожны при использовании «==» для проверки True или 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
```

Не проверяйте, можете ли вы, просто сделайте это и обработайте ошибку

Питонисты обычно говорят: «Легче просить прощения, чем разрешения».

Не рекомендуется **:**

```
if os.path.isfile(file_path):
    file = open(file_path)
else:
    # do something
```

Делать:

```
try:
    file = open(file_path)
except OSError as e:
    # do something
```

Или даже лучше с `Python 2.6+` :

```
with open(file_path) as file:
```

Это намного лучше, потому что это гораздо более общий. Вы можете применить `try/except` почти ничего. Вам не нужно заботиться о том, что делать, чтобы предотвратить это, просто заботьтесь об ошибке, которую вы рискуете.

Не проверяйте тип

Python динамически типизирован, поэтому проверка на тип позволяет вам потерять гибкость. Вместо этого используйте утиную печать , проверяя поведение. Если вы ожидаете строку в функции, используйте `str()` для преобразования любого объекта в строку. Если вы ожидаете список, используйте `list()` для преобразования любого итерабельного в список.

Не рекомендуется **:**

```
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)
```

Делать:

```
def foo(name) :
    print(str(name).lower())

def bar(listing) :
    l = list(listing)
    l.extend((1, 2, 3))
    return ", ".join(l)
```

Используя последний способ, `foo` примет любой объект. `bar` будет принимать строки, кортежи, наборы, списки и многое другое. Дешевые сухие.

Не смешивайте пространства и вкладки

Использовать *объект в* качестве первого родителя

Это сложно, но это укусит вас по мере роста вашей программы. В `Python 2.x` есть старые и

новые классы. Старые, старые, старые. Им не хватает некоторых функций, и они могут иметь неудобное поведение с наследованием. Чтобы быть полезным, любой из ваших классов должен быть «нового стиля». Для этого сделайте его наследуемым от `object` .

Не рекомендуется **:**

```
class Father:
    pass

class Child(Father):
    pass
```

Делать:

```
class Father(object):
    pass


class Child(Father):
    pass
```

В `Python 3.x` все классы являются новым стилем, поэтому вам не нужно это делать.

Не инициализировать атрибуты класса вне метода init

Люди, приезжающие из других языков, находят это заманчивым, потому что это то, что вы делаете на Java или PHP. Вы пишете имя класса, затем указываете свои атрибуты и даете им значение по умолчанию. Кажется, что это работает на Python, однако это не работает так, как вы думаете. Это будет определять атрибуты класса (статические атрибуты), а затем, когда вы попытаетесь получить атрибут объекта, он даст вам свое значение, если оно не пусто. В этом случае он вернет атрибуты класса. Это подразумевает две большие опасности:

- Если атрибут класса изменяется, то начальное значение изменяется.
- Если вы установите изменяемый объект в качестве значения по умолчанию, вы получите тот же объект, общий для экземпляров.

Не нужно (если вы не хотите статического):

```
class Car(object):
    color = "red"
    wheels = [Wheel(), Wheel(), Wheel(), Wheel()]
```

Делать :

```
class Car(object):
    def __init__(self):
        self.color = "red"
        self.wheels = [Wheel(), Wheel(), Wheel(), Wheel()]
```

Целое и строковое тождество

Python использует внутреннее кэширование для целого ряда целых чисел, чтобы уменьшить ненужные накладные расходы из их повторного создания.

По сути, это может привести к запутанному поведению при сравнении целых тождеств:

```
>>> -8 is (-7 - 1)
False
>>> -3 is (-2 - 1)
True
```

и, используя другой пример:

```
>>> (255 + 1) is (255 + 1)
True
>>> (256 + 1) is (256 + 1)
False
```

Чего ждать?

Мы можем видеть , что операция идентичности `is` доходностью `True` для некоторых целых чисел (`-3` , `256`) , но не для других (`-8` , `257`).

Чтобы быть более конкретным, целые числа в диапазоне `[-5, 256]` внутренне кэшируются во время запуска интерпретатора и создаются только один раз. Таким образом , они **идентичны** и сравнивая их идентичность с `is` дают `True` ; целые числа вне этого диапазона (обычно) создаются «на лету», а их идентификаторы сравниваются с `False` .

Это обычная ошибка, так как это общий диапазон для тестирования, но достаточно часто код не работает в более позднем этапе (или хуже - производство) без видимых причин после полной работы в разработке.

Решение состоит в том, чтобы **всегда сравнивать значения, используя** оператор **равенства** (`==`), а **не** тождественный (`is`) оператор.

Python также сохраняет ссылки на часто используемые строки , и может привести к аналогичным странному поведению при сравнении идентичности (т.е. используя `is`) строк.

```
>>> 'python' is 'py' + 'thon'
True
```

Обычно используется строка `'python'` , поэтому у Python есть один объект, который использует все ссылки на строку `'python'` .

Для необычных строк сравнение идентификатора не выполняется, даже когда строки равны.

```
>>> '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
```

Итак, как и правило для целых чисел, **всегда сравнивайте строковые значения с помощью** оператора **равенства** (`==`), а **не** идентификатора (`is`).

Доступ к атрибутам int литералов

Возможно, вы слышали, что все в Python - это объект, даже литералы. Это означает, например, что `7` - объект, а значит, он имеет атрибуты. Например, одним из этих атрибутов является `bit_length` . Он возвращает количество бит, необходимое для представления значения, на которое оно вызвано.

```
x = 7
x.bit_length()
# Out: 3
```

Увидев вышеприведенный код, вы можете интуитивно подумать, что будет работать `7.bit_length()` , но только для того, чтобы узнать, что он вызывает `SyntaxError` . Зачем? потому что интерпретатор должен различать доступ к атрибуту и плавающее число (например, `7.2` или `7.bit_length()`). Этого не может быть, и поэтому возникает исключение.

Существует несколько способов доступа к атрибутам `int` литералов:

```
# parenthesis
(7).bit_length()
# a space
7 .bit_length()
```

Использование двух точек (например, `7..bit_length()`) не работает в этом случае, потому что это создает литерал с `float` а float не имеют `bit_length()` .

Эта проблема не существует при доступе к атрибутам `float` литералов, поскольку интерпертер «умный» достаточно, чтобы знать, что литерал с `float` не может содержать два `.` , например:

```
7.2.as_integer_ratio()
# Out: (8106479329266893, 1125899906842624)
```

Цепочка или оператор

При тестировании любого из нескольких сравнений сравнений:

```
if a == 3 or b == 3 or c == 3:
```

заманчиво сокращать это до

```
if a or b or c == 3: # Wrong
```

Это не верно; оператор `or` имеет более низкий приоритет, чем `==` , поэтому выражение будет оцениваться как `if (a) or (b) or (c == 3):` Правильный способ явно проверяет все условия:

```
if a == 3 or b == 3 or c == 3:  # Right Way
```

Альтернативно, встроенная функция `any()` может использоваться вместо цепочки `or` операторов:

```
if any([a == 3, b == 3, c == 3]): # Right
```

Или, чтобы сделать его более эффективным:

```
if any(x == 3 for x in (a, b, c)): # Right
```

Или, чтобы сделать его короче:

```
if 3 in (a, b, c): # Right
```

Здесь мы используем оператор `in` для проверки, присутствует ли значение в кортеже, содержащем значения, которые мы хотим сравнить.

Аналогично, неправильно писать

```
if a == 1 or 2 or 3:
```

который должен быть записан как

```
if a in (1, 2, 3):
```

sys.argv [0] - это имя исполняемого файла

Первый элемент `sys.argv[0]` - это имя исполняемого файла python. Остальные элементы являются аргументами сценария.

```
# 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']
```

Словари неупорядочены

Вы можете ожидать, что словарь Python будет отсортирован по таким ключам, как, например, C ++ `std::map` , но это не так:

```
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 не имеет встроенного класса, который автоматически сортирует свои элементы по ключу.

Однако, если сортировка не является обязательной, и вы просто хотите, чтобы ваш словарь запоминал порядок вставки его пар ключ / значение, вы можете использовать `collections.OrderedDict` :

```
from collections import OrderedDict

oDict = OrderedDict([('first', 1), ('second', 2), ('third', 3)])

print([k for k in oDict])
# Out: ['first', 'second', 'third']
```

Имейте в виду, что инициализация `OrderedDict` со стандартным словарем не будет сортировать в любом случае словарь для вас. Все, что делает эта структура, - это *сохранить* порядок вставки ключа.

Реализация словарей была изменена в Python 3.6 для улучшения потребления памяти. Побочным эффектом этой новой реализации является то, что она также сохраняет порядок аргументов ключевого слова, переданных функции:

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
```

Предостережение : остерегайтесь того, что *« сохраняющий порядок аспект этой новой реализации рассматривается как деталь реализации, и на него нельзя положиться »* , поскольку это может измениться в будущем.

Глобальный блокиратор перехвата (GIL) и блокирующие потоки

Много написано о GIL Python . Иногда это может вызвать путаницу при работе с многопоточными (не путать с многопроцессорными) приложениями.

Вот пример:

```
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")
```

Вы ожидали увидеть, что `Calculating...` распечатано сразу после начала потока, мы хотели, чтобы вычисления произошли в новом потоке! Но на самом деле, вы видите, что он печатается после завершения расчета. Это связано с тем, что новый поток использует функцию C (`math.factorial`), которая будет блокировать GIL во время ее выполнения.

Есть пара способов обойти это. Первый - реализовать вашу факториальную функцию в родном Python. Это позволит основному потоку захватить управление, находясь внутри вашего цикла. Недостатком является то, что это решение будет **намного** медленнее, поскольку мы больше не используем функцию 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
```

Вы также можете `sleep` течение определенного периода времени перед началом выполнения. Примечание: это фактически не позволит вашей программе прервать вычисление, происходящее внутри функции C, но это позволит вашему основному потоку продолжить работу после появления, что вы можете ожидать.

```
def calc_fact(num):
    sleep(0.001)
```

```
    math.factorial(num)
```

Переменная утечка в списках и для циклов

Рассмотрим следующее понимание списка

Python 2.x 2.7

```
i = 0
a = [i for i in range(3)]
print(i) # Outputs 2
```

Это происходит только в Python 2 из-за того, что понимание списка «утечки» переменной управления циклом в окружающую область (источник). Такое поведение может привести к труднодоступным ошибкам и ***было исправлено в Python 3*** .

Python 3.x 3.0

```
i = 0
a = [i for i in range(3)]
print(i) # Outputs 0
```

Аналогично, для циклов нет частной области для их переменной итерации

```
i = 0
for i in range(3):
    pass
print(i) # Outputs 2
```

Этот тип поведения происходит как в Python 2, так и в Python 3.

Чтобы избежать проблем с утечкой переменных, используйте новые переменные в контекстах списков и, если необходимо, для циклов.

Многократный возврат

Функция xyz возвращает два значения a и b:

```
def xyz():
  return a, b
```

Код, вызывающий xyz, сохраняет результат в одну переменную, предполагая, что xyz возвращает только одно значение:

```
t = xyz()
```

Значение `t` на самом деле является кортежем (a, b), поэтому любое действие по `t` предполагающее, что он не является кортежем, может сильно **затухать** в коде с

неожиданной **ошибкой в** кортежах.

> TypeError: тип кортежа не определяет ... метод

Исправить будет:

```
a, b = xyz()
```

Начинающим будет сложно найти причину этого сообщения, только прочитав сообщение об ошибке кортежа!

Питонические ключи JSON

```
my_var = 'bla';
api_key = 'key';
...lots of code here...
params = {"language": "en", my_var: api_key}
```

Если вы используете JavaScript, оценка переменных в словарях Python будет не такой, как вы ожидаете. Этот оператор в JavaScript приведет к объекту `params` следующим образом:

```
{
    "language": "en",
    "my_var": "key"
}
```

В Python, однако, это приведет к следующему словарю:

```
{
    "language": "en",
    "bla": "key"
}
```

`my_var` оценивается и его значение используется как ключ.

Прочитайте Общие проблемы онлайн: https://riptutorial.com/ru/python/topic/3553/общие-проблемы

глава 116: Объекты недвижимости

замечания

Примечание . В Python 2 убедитесь, что ваш класс наследует объект (что делает его классом нового стиля), чтобы все свойства свойств были доступны.

Examples

Использование декоратора @property

`@property` может использоваться для определения методов в классе, которые действуют как атрибуты. Одним из примеров, когда это может быть полезно, является раскрытие информации, которая может потребовать первоначального (дорогого) поиска и простого извлечения после этого.

Для некоторого модуля `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
```

затем

```
>>> 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
```

Использование декоратора @property для свойств чтения и записи

Если вы хотите использовать `@property` для реализации настраиваемого поведения для настройки и получения, используйте этот шаблон:

```
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:])
```

Чтобы использовать это:

```
>>> 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
```

Переопределение только getter, setter или deterter объекта свойства

Когда вы наследуете класс с свойством, вы можете предоставить новую реализацию для одной или нескольких функций `getter` , `setter` или `deleter` , ссылаясь на объект свойства *родительского класса* :

```
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)
```

Вы также можете добавить сеттер или удалить, если раньше не было базового класса.

Использование свойств без декораторов

Хотя использование синтаксиса декоратора (с помощью @) удобно, оно также немного скрывает. Вы можете использовать свойства напрямую, без декораторов. Следующий пример Python 3.x показывает это:

```
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)
```

Прочитайте Объекты недвижимости онлайн: https://riptutorial.com/ru/python/topic/2050/объекты-недвижимости

глава 117: Операторский модуль

Examples

Операторы как альтернатива инфиксному оператору

Для каждого инфиксного оператора, например `+` существует `operator` функция (`operator.add` для `+`):

```
1 + 1
# Output: 2
from operator import add
add(1, 1)
# Output: 2
```

хотя основная документация говорится , что для арифметических операторов только числовой ввод разрешено можно:

```
from operator import mul
mul('a', 10)
# Output: 'aaaaaaaaaa'
mul([3], 3)
# Output: [3, 3, 3]
```

См. Также: отображение из операции в операторную функцию в официальной документации Python .

Methodcaller

Вместо этой `lambda` функции, которая вызывает метод явно:

```
alist = ['wolf', 'sheep', 'duck']
list(filter(lambda x: x.startswith('d'), alist))     # Keep only elements that start with 'd'
# Output: ['duck']
```

можно использовать оператор-функцию, которая делает то же самое:

```
from operator import methodcaller
list(filter(methodcaller('startswith', 'd'), alist)) # Does the same but is faster.
# Output: ['duck']
```

Itemgetter

Группировка пар ключ-значение словаря по значению с помощью `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}}
```

который эквивалентен (но быстрее) для `lambda` функции следующим образом:

```
dict((i, dict(v)) for i, v in groupby(adict.items(), lambda x: x[1]))
```

Или отсортировать список кортежей вторым элементом сначала первым элементом как вторичным:

```
alist_of_tuples = [(5,2), (1,3), (2,2)]
sorted(alist_of_tuples, key=itemgetter(1,0))
# Output: [(2, 2), (5, 2), (1, 3)]
```

Прочитайте Операторский модуль онлайн: https://riptutorial.com/ru/python/topic/257/операторский-модуль

глава 118: Определение функций со списком аргументов

Examples

Функция и вызов

Списки как аргументы - это еще одна переменная:

```
def func(myList):
    for item in myList:
        print(item)
```

и может быть передан в самом вызове функции:

```
func([1,2,3,5,7])

1
2
3
5
7
```

Или как переменная:

```
aList = ['a','b','c','d']
func(aList)

a
b
c
d
```

Прочитайте Определение функций со списком аргументов онлайн: https://riptutorial.com/ru/python/topic/7744/определение-функций-со-списком-аргументов

глава 119: Оптимизация производительности

замечания

При попытке улучшить производительность скрипта Python, прежде всего, вы должны найти узкое место вашего сценария и отметить, что никакая оптимизация не может компенсировать плохой выбор структур данных или недостаток в разработке вашего алгоритма. Определение узких мест производительности может быть выполнено путем профилирования вашего скрипта. Во-вторых, не пытайтесь оптимизировать слишком рано в процессе кодирования за счет удобочитаемости / дизайна / качества. Дональд Кнут сделал следующее заявление об оптимизации:

> «Мы должны забыть о небольшой эффективности, скажем, около 97% времени: преждевременная оптимизация - корень всего зла. Но мы не должны упускать наши возможности в этих критических 3% ».

Examples

Профилирование кода

Прежде всего, вы должны найти узкое место вашего сценария и отметить, что никакая оптимизация не может компенсировать плохой выбор структуры данных или недостаток в дизайне вашего алгоритма. Во-вторых, не пытайтесь оптимизировать слишком рано в процессе кодирования за счет удобочитаемости / дизайна / качества. Дональд Кнут сделал следующее заявление об оптимизации:

> «Мы должны забыть о небольшой эффективности, скажем, примерно в 97% случаев: преждевременная оптимизация - это корень всего зла, но мы не должны упускать наши возможности в этих критических 3%

Для `cProfile` вашего кода у вас есть несколько инструментов: `cProfile` (или более медленный `profile`) из стандартной библиотеки, `line_profiler` и `timeit` . Каждый из них служит другой цели.

`cProfile` - это детерминированный профилировщик: отслеживаются вызов функции, возврат функции и события исключения, а также точные тайминги для интервалов между этими событиями (до 0,001 с). Документация библиотеки ([https://docs.python.org/2/library/profile.html][1]) предоставляет нам простой пример использования

```
import cProfile
def f(x):
    return "42!"
cProfile.run('f(12)')
```

Или, если вы предпочитаете обертывать части вашего существующего кода:

```
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()
```

Это создаст результаты, похожие на таблицу ниже, где вы сможете быстро увидеть, где ваша программа проводит большую часть своего времени, и определить функции для оптимизации.

```
         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}
```

Модуль `line_profiler` ([https://github.com/rkern/line_profiler][1]) полезен для линейного анализа вашего кода. Очевидно, что это невозможно для длинных скриптов, но нацелено на фрагменты. Дополнительную информацию см. В документации. Самый простой способ начать - использовать скрипт kernprof, как описано на странице пакета, обратите внимание, что вам нужно будет вручную указать функцию (функции) для профиля.

```
$ kernprof -l script_to_profile.py
```

kernprof создаст экземпляр LineProfiler и вставляет его в пространство имен `__builtins__` с профилем имени. Он был написан для использования в качестве декоратора, поэтому в вашем скрипте вы украшаете функции, которые хотите профилировать с помощью `@profile` .

```
@profile
def slow_function(a, b, c):
    ...
```

Поведение kernprof по умолчанию заключается в том, чтобы поместить результаты в двоичный файл `script_to_profile.py.lprof` . Вы можете сказать kernprof немедленно просмотреть отформатированные результаты на терминале с опцией [-v / - view]. В

противном случае вы можете просмотреть результаты позже так:

```
$ python -m line_profiler script_to_profile.py.lprof
```

Наконец, `timeit` предоставляет простой способ протестировать один лайнер или небольшое выражение как из командной строки, так и из оболочки python. Этот модуль ответит на вопрос, например: быстрее ли выполнять понимание списка или использовать встроенный `list()` при преобразовании набора в список. Найдите ключевое слово `setup` или `-s` чтобы добавить установочный код.

```
>>> import timeit
>>> timeit.timeit('"-".join(str(n) for n in range(100))', number=10000)
0.8187260627746582
```

от терминала

```
$ python -m timeit '"-".join(str(n) for n in range(100))'
10000 loops, best of 3: 40.3 usec per loop
```

Прочитайте Оптимизация производительности онлайн:
https://riptutorial.com/ru/python/topic/5889/оптимизация-производительности

глава 120: Оптическое распознавание символов

Вступление

Оптическое распознавание символов преобразует изображения текста в фактический текст. В этих примерах найдите способы использования OCR в python.

Examples

PyTesseract

PyTesseract - это пакет python для разработки для OCR.

Использование PyTesseract довольно просто:

```
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 является открытым исходным кодом и может быть найден здесь .

PyOCR

Другим модулем для использования является `PyOCR` , исходный код которого приведен здесь .

Также прост в использовании и имеет больше возможностей, чем `PyTesseract` .

Для инициализации:

```
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.
```

И некоторые примеры использования:

```
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
```

Прочитайте Оптическое распознавание символов онлайн:
https://riptutorial.com/ru/python/topic/9302/оптическое-распознавание-символов

глава 121: Основные входные и выходные данные

Examples

Использование input () и raw_input ()

Python 2.x 2.3

`raw_input` будет ждать ввода пользователем текста, а затем возвращает результат в виде строки.

```
foo = raw_input("Put a message here that asks the user for input")
```

В приведенном выше примере `foo` сохранит все входящие данные, предоставленные пользователем.

Python 3.x 3.0

`input` будет ждать `input` пользователем текста, а затем возвращает результат в виде строки.

```
foo = input("Put a message here that asks the user for input")
```

В приведенном выше примере `foo` сохранит все входящие данные, предоставленные пользователем.

Использование функции печати

Python 3.x 3.0

В Python 3 функция печати имеет форму функции:

```
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

В Python 2 изначально был напечатан оператор, как показано ниже.

```
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.
```

Примечание: использование `from __future__ import print_function` в Python 2 позволит пользователям использовать функцию `print()` же, как и код Python 3. Это доступно только в Python 2.6 и выше.

Функция для запроса пользователю номера

```
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)
```

И использовать его:

```
user_number = input_number("input a number: ", "that's not a number!")
```

Или, если вы не хотите «сообщение об ошибке»:

```
user_number = input_number("input a number: ")
```

Печать строки без новой строки в конце

Python 2.x 2.3

В Python 2.x, чтобы продолжить строку с `print` , завершите инструкцию `print` запятой. Он автоматически добавит пробел.

```
print "Hello,",
print "World!"
# Hello, World!
```

Python 3.x 3.0

В Python 3.x функция `print` имеет необязательный `end` параметр, который он печатает в конце данной строки. По умолчанию это символ новой строки, что эквивалентно этому:

```
print("Hello, ", end="\n")
print("World!")
# Hello,
# World!
```

Но вы могли бы передать другие строки

```
print("Hello, ", end="")
print("World!")
# Hello, World!

print("Hello, ", end="<br>")
print("World!")
# Hello, <br>World!

print("Hello, ", end="BREAK")
print("World!")
# Hello, BREAKWorld!
```

Если вы хотите получить больше контроля над выходом, вы можете использовать `sys.stdout.write` :

```
import sys

sys.stdout.write("Hello, ")
sys.stdout.write("World!")
# Hello, World!
```

Читать с stdin

Программы Python могут считывать из unix-конвейеров . Вот простой пример, как читать из `stdin` :

```
import sys

for line in sys.stdin:
    print(line)
```

Имейте в `sys.stdin` что `sys.stdin` - это поток. Это означает, что for-loop прекратится только после завершения потока.

Теперь вы можете передать вывод другой программы в свою программу python следующим образом:

```
$ cat myfile | python myprogram.py
```

В этом примере `cat myfile` может быть любой командой unix, которая выводит на `stdout` .

В качестве альтернативы, использование модуля fileinput может оказаться полезным:

```
import fileinput
```

```
for line in fileinput.input():
    process(line)
```

Вход из файла

Входные данные также можно считывать из файлов. Файлы могут быть открыты с помощью встроенной функции `open` . Использование а `with <command> as <name>` синтаксис `with <command> as <name>` (называемый «Контекст-менеджер») делает использование `open` и получение дескриптора для файла очень просто:

```
with open('somefile.txt', 'r') as fileobj:
    # write code here using fileobj
```

Это гарантирует, что при выходе кода из блока файл автоматически закрывается.

Файлы можно открывать в разных режимах. В приведенном выше примере файл открывается как доступный только для чтения. Чтобы открыть существующий файл для чтения, используйте только `r` . Если вы хотите прочитать этот файл в байтах, используйте `rb` . Чтобы добавить данные в существующий файл, используйте `a` . Используйте `w` для создания файла или перезаписывания любых существующих файлов с тем же именем. Вы можете использовать `r+` для открытия файла для чтения и записи. Первый аргумент `open()` - имя файла, второй - режим. Если режим оставлен пустым, по умолчанию будет установлено значение `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']
```

Если размер файла крошечный, можно прочитать все содержимое файла в памяти. Если файл очень большой, часто лучше читать строки за строкой или кусками и обрабатывать ввод в том же цикле. Для этого:

```
with open('shoppinglist.txt', 'r') as fileobj:
```

```
    # this method reads line by line:
    lines = []
    for line in fileobj:
        lines.append(line.strip())
```

При чтении файлов имейте в виду персонажи разрыва строки операционной системы. Хотя `for line in fileobj` автоматически удаляет их, всегда безопасно вызывать `strip()` на прочитанных строках, как показано выше.

Открытые файлы (`fileobj` в приведенных выше примерах) всегда указывают на конкретное место в файле. Когда они сначала открываются, дескриптор файла указывает на самое начало файла, который является позицией `0` . Дескриптор файла может отображать его текущее положение с `tell` :

```
fileobj = open('shoppinglist.txt', 'r')
pos = fileobj.tell()
print('We are at %u.' % pos) # We are at 0.
```

После прочтения всего содержимого позиция обработчика файла будет указана в конце файла:

```
content = fileobj.read()
end = fileobj.tell()
print('This file was %u characters long.' % end)
# This file was 22 characters long.
fileobj.close()
```

Позиция обработчика файла может быть настроена на все, что необходимо:

```
fileobj = open('shoppinglist.txt', 'r')
fileobj.seek(7)
pos = fileobj.tell()
print('We are at character #%u.' % pos)
```

Вы также можете читать любую длину из содержимого файла во время данного вызова. Для этого передайте аргумент для `read()` . Когда `read()` вызывается без аргумента, он будет считываться до конца файла. Если вы передадите аргумент, он будет читать это количество байтов или символов в зависимости от режима (`rb` и `r` соответственно):

```
# 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()
```

Чтобы продемонстрировать разницу между символами и байтами:

```
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'>
```

Прочитайте Основные входные и выходные данные онлайн:
https://riptutorial.com/ru/python/topic/266/основные-входные-и-выходные-данные

глава 122: Основные проклятия с Python

замечания

Curses - это базовый модуль обработки терминала (или символьного отображения) из Python. Это можно использовать для создания пользовательских интерфейсов на терминале или TUI.

Это порт python более популярной библиотеки C-библиотеки ncurses,

Examples

Основной пример вызова

```
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()
```

Вспомогательная функция wrapper ().

Хотя основной вызов выше достаточно прост, пакет curses предоставляет

вспомогательную функцию `wrapper(func, ...)` . Пример ниже содержит эквивалент выше:

```
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)
```

Здесь обертка будет инициализировать проклятия, создать `stdscr` , WindowObject и передать как stdscr, так и любые дополнительные аргументы для `func` . Когда `func` возвращает, `wrapper` будет восстанавливать терминал до выхода программы.

Прочитайте Основные проклятия с Python онлайн: https://riptutorial.com/ru/python/topic/5851/основные-проклятия-с-python

глава 123: откладывать в долгий ящик

Вступление

Shelve - это модуль python, используемый для хранения объектов в файле. Модуль полки реализует постоянное хранилище для произвольных объектов Python, которые можно мариновать, используя словарь-подобный API. Модуль полки может использоваться как простая постоянная память для объектов Python, когда реляционная база данных переполнена. Доступ к полкам осуществляется с помощью клавиш, как и со словарем. Значения подсчитываются и записываются в базу данных, созданную и управляемую anydbm.

замечания

Примечание. Не полагайтесь на закрывающуюся полку автоматически; всегда вызывайте `close()` явно, когда вам это больше не нужно, или используйте `shelve.open()` в качестве менеджера контекста:

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

Предупреждение:

Поскольку модуль `shelve` опирается на `pickle` , небезопасно загружать полку из ненадежного источника. Как и при рассоле, загрузка полки может выполнять произвольный код.

ограничения

1 . Выбор того, какой пакет базы данных будет использоваться (например, dbm.ndbm или dbm.gnu), зависит от того, какой интерфейс доступен. Поэтому небезопасно открывать базу данных напрямую с помощью dbm. База данных также (к сожалению) зависит от ограничений dbm, если она используется - это означает, что (маринованное представление) объекты, хранящиеся в базе данных, должны быть довольно маленькими, и в редких случаях столкновения с ключами могут привести к тому, что база данных будет отказаться от обновлений.

2. Модуль полки не поддерживает одновременный доступ на чтение и запись к закрытым объектам. (Несколько одновременных доступов для чтения безопасны.) Когда у программы есть открытая полка для записи, никакая другая программа не должна открывать ее для

чтения или записи. Для решения этой проблемы можно использовать блокировку файлов Unix, но это отличается от версий Unix и требует знаний о используемой реализации базы данных.

Examples

Пример кода для полки

Чтобы отложить объект, сначала импортируйте модуль, а затем назначьте значение объекта следующим образом:

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

Чтобы суммировать интерфейс (ключ - строка, данные - произвольный объект):

```
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
```

Создание нового шельфа

Самый простой способ использования полки - это класс **DbfilenameShelf** . Он использует

anydbm для хранения данных. Вы можете использовать класс напрямую или просто вызвать **shelve.open ()** :

```
import shelve

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

Чтобы снова получить доступ к данным, откройте полку и используйте ее как словарь:

```
    import shelve

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

print existing
```

Если вы запускаете оба примера сценариев, вы должны увидеть:

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

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

Модуль **dbm** не поддерживает одновременное использование нескольких приложений, записываемых в одну и ту же базу данных. Если вы знаете, что ваш клиент не будет изменять полку, вы можете сообщить полке, чтобы открыть базу данных только для чтения.

```
import shelve

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

print existing
```

Если ваша программа пытается изменить базу данных, пока она открыта только для чтения, генерируется исключение ошибки доступа. Тип исключения зависит от модуля базы данных, выбранного anydbm при создании базы данных.

Write-назад

Полки не отслеживают изменения изменчивых объектов по умолчанию. Это означает, что если вы измените содержимое элемента, хранящегося на полке, вы должны явно обновить полку, сохранив элемент снова.

```
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()
```

В этом примере словарь в 'key1' не сохраняется снова, поэтому, когда полка снова открывается, изменения не сохранились.

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

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

Чтобы автоматически улавливать изменения изменчивых объектов, хранящихся на полке, откройте полку с включенной обратной записью. Флаг обратной записи заставляет полку запоминать все объекты, извлеченные из базы данных, используя кеш в памяти. Каждый объект кэша также записывается обратно в базу данных, когда полка закрыта.

```
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()
```

Хотя это уменьшает вероятность ошибки программиста и может сделать сохранение объекта более прозрачным, использование режима обратной записи может быть нежелательным в любой ситуации. Кэш потребляет дополнительную память, пока полка открыта, и приостановка записи каждого кэшированного объекта обратно в базу данных,

когда она закрыта, может занять дополнительное время. Поскольку нет способа узнать, были ли изменены кешированные объекты, все они записаны обратно. Если ваше приложение читает данные больше, чем пишет, обратная связь добавит больше накладных расходов, чем вы могли бы захотеть.

```
$ 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'}
```

Прочитайте откладывать в долгий ящик онлайн: https://riptutorial.com/ru/python/topic/10629/откладывать-в-долгий-ящик

глава 124: отладка

Examples

Отладчик Python: сквозная отладка с помощью _pdb_

Стандартная библиотека Python включает в себя интерактивную библиотеку отладки под названием *pdb* . *pdb* обладает широкими возможностями, наиболее часто используемым является способность «пройти» программу.

Для немедленного ввода в пошаговую отладку используйте:

```
python -m pdb <my_file.py>
```

Это запустит отладчик в первой строке программы.

Обычно вам нужно настроить таргетинг на определенный раздел кода для отладки. Для этого мы импортируем библиотеку pdb и используем *set_trace (),* чтобы прервать поток этого проблемного кода примера.

```
import pdb

def divide(a, b):
    pdb.set_trace()
    return a/b
    # What's wrong with this? Hint: 2 != 3

print divide(1, 2)
```

Запуск этой программы запустит интерактивный отладчик.

```
python foo.py
> ~/scratch/foo.py(5)divide()
-> return a/b
(Pdb)
```

Часто эта команда используется в одной строке, поэтому ее можно прокомментировать одним символом #

```
 import pdf; pdb.set_trace()
```

В *командной* строке *(Pdb)* могут быть введены команды. Этими командами могут быть команды отладчика или python. Для печати переменных мы можем использовать *p* из отладчика или *печати* python.

```
(Pdb) p a
```

```
1
(Pdb) print a
1
```

Чтобы просмотреть список всех локальных переменных, используйте

```
locals
```

встроенная функция

Это хорошие команды отладчика, чтобы знать:

```
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)
```

Отладчик также может оценивать python в интерактивном режиме:

```
-> 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]
```

Замечания:

Если любое из ваших имен переменных совпадает с командами отладчика, используйте восклицательный знак « **!** 'перед тем, как var явно ссылается на переменную, а не на команду отладчика. Например, часто бывает так, что вы используете имя переменной « **c** » для счетчика, и вы можете распечатать его в отладчике. простая команда « **c** » продолжит выполнение до следующей точки останова. Вместо этого используйте ' **! C** ', чтобы

напечатать значение переменной следующим образом:

```
(Pdb) !c
4
```

Через IPython и ipdb

Если IPython (или Jupyter) установлены, отладчик можно вызвать, используя:

```
import ipdb
ipdb.set_trace()
```

По достижении кода код выхода и выхода:

```
 /home/usr/ook.py(3)<module>()
      1 import ipdb
      2 ipdb.set_trace()
----> 3 print("Hello world!")

ipdb>
```

Очевидно, это означает, что нужно редактировать код. Существует более простой способ:

```
from IPython.core import ultratb
sys.excepthook = ultratb.FormattedTB(mode='Verbose',
                                     color_scheme='Linux',
                                     call_pdb=1)
```

Это приведет к вызову отладчика, если возникло непокрытое исключение.

Удаленный отладчик

Иногда вам нужно отлаживать код python, который выполняется другим процессом, и в этом случае `rpdb` .

> rpdb - это оболочка вокруг pdb, которая перенаправляет stdin и stdout в обработчик сокета. По умолчанию он открывает отладчик на порту 4444

Использование:

```
# In the Python file you want to debug.
import rpdb
rpdb.set_trace()
```

И тогда вам нужно запустить это в терминале, чтобы подключиться к этому процессу.

```
# Call in a terminal to see the output
$ nc 127.0.0.1 4444
```

И вы получите pdb promt

```
> /home/usr/ook.py(3)<module>()
-> print("Hello world!")
(Pdb)
```

Прочитайте отладка онлайн: https://riptutorial.com/ru/python/topic/2077/отладка

глава 125: Параллельное вычисление

замечания

Из-за GIL (блокировка глобального интерпретатора) только один экземпляр интерпретатора python выполняется в одном процессе. В общем, использование многопоточности только улучшает привязку IO, а не связанные с ЦП. Модуль `multiprocessing` рекомендуется, если вы хотите параллельно выполнять задачи, связанные с ЦП.

GIL применяется к CPython, самой популярной реализации Python, а также PyPy. Другие реализации, такие как Jython и IronPython, не имеют GIL .

Examples

Использование многопроцессорного модуля для параллелизации задач

```
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]
```

Поскольку выполнение каждого вызова `fib` происходит параллельно, время выполнения полного примера составляет **1,8 × быстрее,** чем если бы последовательное выполнение на двоичном процессоре.

Python 2.2+

Использование скриптов Parent и Children для параллельного выполнения кода

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()
```

Это полезно для параллельных независимых запросов HTTP-запроса / ответа или выбора / вставки базы данных. Аргументы командной строки могут быть предоставлены и **скрипту child.py** . Синхронизация между сценариями может быть достигнута всеми скриптами, регулярно проверяющими отдельный сервер (например, экземпляр Redis).

Использование С-расширения для параллелизации задач

Идея здесь состоит в том, чтобы переместить интенсивные вычислительные задания на C (используя специальные макросы), независимо от Python, и иметь C-код для выпуска GIL во время работы.

```
#include "Python.h"
...
PyObject *pyfunc(PyObject *self, PyObject *args) {
    ...
    Py_BEGIN_ALLOW_THREADS
    // Threaded C code
    ...
    Py_END_ALLOW_THREADS
    ...
}
```

Использование модуля PyPar для параллелизации

PyPar - это библиотека, которая использует интерфейс передачи сообщений (MPI) для обеспечения параллелизма в Python. Простой пример в PyPar (см. Https://github.com/daleroberts/pypar) выглядит следующим образом:

```
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()
```

Прочитайте Параллельное вычисление онлайн: https://riptutorial.com/ru/python/topic/542/параллельное-вычисление

глава 126: Параллельность Python

замечания

Разработчики Python следили за тем, чтобы API между `threading` и `multiprocessing` был схожим, так что для обоих программистов проще переключаться между двумя вариантами.

Examples

Модуль резьбонарезания

```
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()
```

В некоторых реализациях Python, таких как CPython, истинный параллелизм не достигается с помощью потоков из-за использования так называемого GIL или **G** lobal **I** nterpreter **L** ock.

Вот отличный обзор параллелизма Python:

Параллельный подход Python Дэвида Бизли (YouTube)

Многопроцессорный модуль

```
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()
```

Здесь каждая функция выполняется в новом процессе. Поскольку новый экземпляр Python VM запускает код, `GIL` не существует, и вы выполняете параллелизм на нескольких ядрах.

Метод `Process.start` запускает этот новый процесс и запускает функцию, переданную в `target` аргументе аргументами `args` . Метод `Process.join` ждет завершения выполнения процессов `p1` и `p2` .

Новые процессы запускаются по-разному в зависимости от версии python и формы табло, на которой работает код, *например* :

- Для создания нового процесса Windows использует `spawn` .
- В Unix-системах и версии раньше 3.3 процессы создаются с использованием `fork` . Обратите внимание, что этот метод не учитывает использование вилки POSIX и, таким образом, приводит к неожиданному поведению, особенно при взаимодействии с другими многопроцессорными библиотеками.
- С системой unix и версией 3.4+ вы можете запускать новые процессы с помощью `fork` , `forkserver` или `spawn` с использованием `multiprocessing.set_start_method` в начале вашей программы. `forkserver` и `spawn` медленнее, чем forking, но избегают некоторых неожиданных действий.

Использование вилки POSIX :

> После вилки в многопоточной программе, ребенок может безопасно вызывать только функции, защищенные от асинхронного сигнала, до тех пор, пока он не вызовет execve.
> (см.)

Используя fork, новый процесс будет запущен с тем же самым состоянием для всех текущих мьютексов, но будет запущен только `MainThread` . Это небезопасно, так как это может привести к условиям гонки, *например* :

- Если вы используете `Lock` в `MainThread` и передаете его другому потоку, который, предположительно, заблокирует его в какой-то момент. Если `fork` одновременно, новый процесс начинается с заблокированной блокировки, которая никогда не будет выпущена, поскольку второй поток не существует в этом новом процессе.

Собственно, такое поведение не должно происходить в чистом питоне, так как `multiprocessing` обрабатывает его правильно, но если вы взаимодействуете с другой библиотекой, такое поведение может происходить, что приводит к сбою вашей системы (например, с помощью numpy / speed на macOS).

Передача данных между процессами многопроцессорности

Поскольку данные чувствительны при взаимодействии между двумя потоками (думаю, одновременное чтение и одновременная запись могут конфликтовать друг с другом, вызывая условия гонки), был создан набор уникальных объектов, чтобы облегчить передачу данных между потоками. Любая действительно атомная операция может использоваться между потоками, но всегда безопасно придерживаться очереди.

```
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
```

Большинство людей полагают, что при использовании очереди всегда ставить данные очереди в try: except: block вместо использования пустого. Тем не менее, для приложений, где не имеет значения, если вы пропустите цикл сканирования (данные могут быть помещены в очередь, в то время как они перебрасывают состояния из `queue.Empty==True` в `queue.Empty==False`), как правило, лучше размещать чтение и записывать доступ в том, что я называю блоком Iftry, потому что оператор «if» технически более эффективен, чем перехват исключения.

```
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
```

Прочитайте Параллельность Python онлайн: https://riptutorial.com/ru/python/topic/3357/параллельность-python

глава 127: перегрузка

Examples

Магические / Dunder методы

Магия (также называемая dunder как аббревиатура для double-underscore) методов в Python служит аналогичной цели для перегрузки операторов на других языках. Они позволяют классу определять его поведение, когда он используется как операнд в унарных или двоичных операторных выражениях. Они также служат в качестве реализаций, называемых некоторыми встроенными функциями.

Рассмотрим эту реализацию двумерных векторов.

```
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)
```

Теперь можно естественным образом использовать экземпляры класса `Vector` в различных выражениях.

```
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
```

Типы контейнеров и последовательности

Можно эмулировать типы контейнеров, которые поддерживают доступ к значениям с помощью ключа или индекса.

Рассмотрим эту наивную реализацию редкого списка, в котором хранятся только ненулевые элементы для сохранения памяти.

```
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
```

Затем мы можем использовать `sparselist` , как обычный `list` .

```
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
```

Типы вызовов

```
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
```

Обработка нереализованного поведения

Если ваш класс не реализует определенный перегруженный оператор для предоставленных типов аргументов, он должен `return NotImplemented` (**обратите внимание,** что это специальная константа , не такая же, как `NotImplementedError`). Это позволит Python вернуться к другим методам, чтобы заставить работу работать:

> Когда `NotImplemented` возвращается, интерпретатор затем попытается `NotImplemented` отраженную операцию на другом типе или другом резервном копировании в зависимости от оператора. Если все предпринятые операции возвращают `NotImplemented` , интерпретатор поднимет соответствующее исключение.

Например, если `x + y` , если `x.__add__(y)` возвращает unimplemented, вместо этого выполняется `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__
```

Поскольку это *отраженный* метод, мы должны реализовать `__add__` **и** `__radd__` чтобы получить ожидаемое поведение во всех случаях; к счастью, поскольку они оба делают то же самое в этом простом примере, мы можем воспользоваться ярлыком.

В использовании:

```
>>> 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
```

Перегрузка оператора

Ниже приведены операторы, которые могут быть перегружены в классах, а также требуемые определения методов и пример оператора, используемого в выражении.

NB Использование `other` в качестве имени переменной не является обязательным, но считается нормой.

оператор	метод	выражение
`+` Дополнение	`__add__(self, other)`	`a1 + a2`
`-` Вычитание	`__sub__(self, other)`	`a1 - a2`
`*` Умножение	`__mul__(self, other)`	`a1 * a2`
`@` Матричное умножение	`__matmul__(self, other)`	`a1 @ a2` (*Python 3.5*)
`/` Отдел	`__div__(self, other)`	`a1 / a2` (*только для Python 2*)
`/` Отдел	`__truediv__(self, other)`	`a1 / a2` (*Python 3*)
`//` Отдел этажей	`__floordiv__(self, other)`	`a1 // a2`
`%` Modulo / Remainder	`__mod__(self, other)`	`a1 % a2`

оператор	метод	выражение
** Мощность	__pow__(self, other[, modulo])	a1 ** a2
<< Побитовый сдвиг влево	__lshift__(self, other)	a1 << a2
>> Побитовый правый сдвиг	__rshift__(self, other)	a1 >> a2
& Побитовое И	__and__(self, other)	a1 & a2
^ Побитовое XOR	__xor__(self, other)	a1 ^ a2
| (Побитовое ИЛИ)	__or__(self, other)	a1 | a2
- Отрицание (арифметика)	__neg__(self)	-a1
+ Положительный	__pos__(self)	+a1
~ Побитовое НЕ	__invert__(self)	~a1
< Меньше, чем	__lt__(self, other)	a1 < a2
<= Меньше или равно	__le__(self, other)	a1 <= a2
== Равно	__eq__(self, other)	a1 == a2
!= Не равно	__ne__(self, other)	a1 != a2
> Больше, чем	__gt__(self, other)	a1 > a2
>= Больше или равно	__ge__(self, other)	a1 >= a2
[index] Оператор индекса	__getitem__(self, index)	a1[index]
in операторе In	__contains__(self, other)	a2 in a1
(*args, ...) Вызов	__call__(self, *args, **kwargs)	a1(*args, **kwargs)

Необязательный параметр `modulo` для `__pow__` используется только встроенной функцией `pow` .

Каждый из методов, соответствующих *двоичному* оператору, имеет соответствующий «правый» метод, начинающийся с `__r` , например `__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
```

а также соответствующую версию `__i` , начиная с `__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
```

Поскольку в этих методах нет ничего особенного, многие другие части языка, части стандартной библиотеки и даже сторонние модули самостоятельно добавляют магические методы, например методы для приведения объекта к типу или проверки свойств объекта. Например, встроенная функция `str()` вызывает метод `__str__` объекта, если он существует. Некоторые из этих видов использования перечислены ниже.

функция	метод	выражение
Кастинг для `int`	`__int__(self)`	`int(a1)`
Абсолютная функция	`__abs__(self)`	`abs(a1)`
Кастинг на `str`	`__str__(self)`	`str(a1)`
Кастинг для `unicode`	`__unicode__(self)`	`unicode(a1)` (только для Python 2)
Строковое представление	`__repr__(self)`	`repr(a1)`
Кастинг для `bool`	`__nonzero__(self)`	`bool(a1)`
Форматирование строк	`__format__(self, formatstr)`	`"Hi {:abc}".format(a1)`
хеширования	`__hash__(self)`	`hash(a1)`

функция	метод	выражение
длина	`__len__(self)`	`len(a1)`
Перевернутый	`__reversed__(self)`	`reversed(a1)`
Этаж	`__floor__(self)`	`math.floor(a1)`
потолок	`__ceil__(self)`	`math.ceil(a1)`

Существуют также специальные методы `__enter__` и `__exit__` для менеджеров контекста и многие другие.

Прочитайте перегрузка онлайн: https://riptutorial.com/ru/python/topic/2063/перегрузка

глава 128: Переопределение метода

Examples

Основной метод переопределения

Ниже приведен пример базового переопределения в Python (для ясности и совместимости с Python 2 и 3, используя новый класс стиля и `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
```

Когда класс `Child` создан, он наследует методы класса `Parent` . Это означает, что любые методы, которые имеет родительский класс, также будет иметь дочерний класс. В этом примере `introduce` определено для класса `Child` потому что оно определено для `Parent` , несмотря на то, что оно явно не определено в определении класса `Child` .

В этом примере переопределение происходит, когда `Child` определяет свой собственный метод `print_name` . Если этот метод не был объявлен, то `c.print_name()` напечатал бы `"Parent"` . Однако `Child` переопределил определение `Parent` `print_name` , и теперь, после вызова `c.print_name()` , печатается слово `"Child"` .

Прочитайте Переопределение метода онлайн: https://riptutorial.com/ru/python/topic/3131/переопределение-метода

глава 129: Перечисление списков

Вступление

Перечисление списков в Python - это сжатые, синтаксические конструкции. Они могут использоваться для создания списков из других списков путем применения функций к каждому элементу в списке. В следующем разделе объясняется и демонстрируется использование этих выражений.

Синтаксис

- [x + 1 для x в (1, 2, 3)] # перечисление, дает [2, 3, 4]
- (x + 1 для x в (1, 2, 3)) # генераторное выражение, даст 2, затем 3, затем 4
- [x для x в (1, 2, 3), если x% 2 == 0] # понимание списка с фильтром, дает [2]
- [x + 1, если x% 2 == 0 else x для x в (1, 2, 3)] # список с тройным
- [x + 1, если x% 2 == 0 else x для x в диапазоне (-3,4), если x> 0] # список с тройной и фильтрацией
- {x for x in (1, 2, 2, 3)} # set assrehension, дает {1, 2, 3}
- {k: v для k, v в [('a', 1), ('b', 2)]} # dict понимает, дает {'a': 1, 'b': 2} (python 2.7+ и Только 3.0+)
- [x + y для x в [1, 2] для y в [10, 20]] # Вложенные петли дают [11, 21, 12, 22]
- [x + y для x в [1, 2, 3], если x> 2 для y в [3, 4, 5]] # Условие проверено на 1-ом для цикла
- [x + y для x в [1, 2, 3] для y в [3, 4, 5], если x> 2] # Условие проверено на 2-ом для цикла
- [x для x в xrange (10), если x% 2 == 0] # Условие проверено, если зацикленные числа являются нечетными числами

замечания

Понимание - синтаксические конструкции, которые определяют структуры данных или выражения, уникальные для конкретного языка. Правильное использование понятий переосмысливает их в легко понятых выражениях. В качестве выражений они могут использоваться:

- в правой части заданий
- как аргументы для вызова функций
- в теле лямбда-функции
- как самостоятельные заявления. (Например: `[print(x) for x in range(10)]`)

Examples

Список рекомендаций

[Понимание списка](#) создает новый `list` , применяя выражение к каждому элементу [итерабельного](#) . Самая основная форма:

```
[ <expression> for <element> in <iterable> ]
```

Также есть необязательное условие «если»:

```
[ <expression> for <element> in <iterable> if <condition> ]
```

Каждый `<element>` в `<iterable>` подключается к `<expression>` если (необязательно) `<condition>` значение true . Все результаты возвращаются сразу в новом списке. Выражения генератора оцениваются лениво, но списки понимают всю целостность итератора, потребляющую память пропорционально длине итератора.

Чтобы создать `list` целых чисел в квадрате:

```
squares = [x * x for x in (1, 2, 3, 4)]
# squares: [1, 4, 9, 16]
```

Выражение `for` для каждого значения, в свою очередь, устанавливает `x (1, 2, 3, 4)` . Результат выражения `x * x` добавляется во внутренний `list` . Внутренний `list` присваивается `squares` переменных по завершении.

Помимо увеличения скорости (как поясняется здесь), понимание списка примерно эквивалентно следующему for-loop:

```
squares = []
for x in (1, 2, 3, 4):
    squares.append(x * x)
# squares: [1, 4, 9, 16]
```

Выражение, применяемое к каждому элементу, может быть как можно более сложным:

```
# 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']
```

`else` может использоваться в конструкциях понимания List, но будьте осторожны относительно синтаксиса. Предложения if / else должны использоваться до цикла `for` , а не после:

```
# 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']
```

Обратите внимание, что это использует другую конструкцию языка, условное выражение , которое само по себе не является частью синтаксиса понимания . Принимая во внимание, что `if` после того, как `for…in` *является* частью понимания списков и используется для *фильтрации* элементов из исходного итерабельного.

Двойная итерация

Порядок двойной итерации `[... for x in ... for y in ...]` является либо естественным, либо контринтуитивным. Правило заключается в том , чтобы следовать эквивалент `for` цикла:

```
def foo(i):
    return i, i + 0.5

for i in range(3):
    for x in foo(i):
        yield str(x)
```

Это становится:

```
[str(x)
    for i in range(3)
        for x in foo(i)
]
```

Это можно сжать в одну строку как `[str(x) for i in range(3) for x in foo(i)]`

Мутация на месте и другие побочные эффекты

Перед использованием списка понимания, понять разницу между функциями , вызываемыми для их побочных эффектов *(Mutating,* или *в месте* функции) , которые , как правило , не возвращают `None` , и функции , которые возвращают интересное значение.

Многие функции (особенно *чистые* функции) просто берут объект и возвращают некоторый объект. Функция *in-place* изменяет существующий объект, который называется *побочным эффектом* . Другие примеры включают операции ввода и вывода, такие как печать.

`list.sort()` сортирует список *на месте* (что означает, что он изменяет исходный список) и возвращает значение `None` . Поэтому он не будет работать, как ожидалось, в понимании списка:

```
[x.sort() for x in [[2, 1], [4, 3], [0, 1]]]
# [None, None, None]
```

Вместо этого `sorted()` возвращает отсортированный `list` а не сортировку на месте:

```
[sorted(x) for x in [[2, 1], [4, 3], [0, 1]]]
# [[1, 2], [3, 4], [0, 1]]
```

Возможно использование понятий для побочных эффектов, таких как функции ввода-вывода или функции на месте. Однако цикл for обычно более читабельен. Хотя это работает в Python 3:

```
[print(x) for x in (1, 2, 3)]
```

Вместо этого используйте:

```
for x in (1, 2, 3):
    print(x)
```

В некоторых ситуациях, побочные функции эффекта подходят для списка понимания. `random.randrange()` имеет побочный эффект изменения состояния генератора случайных чисел, но он также возвращает интересное значение. Кроме того, `next()` можно вызывать на итераторе.

Следующий генератор случайных величин не является чистым, но имеет смысл, когда случайный генератор сбрасывается каждый раз, когда выражение оценивается:

```
from random import randrange
[randrange(1, 7) for _ in range(10)]
# [2, 3, 2, 1, 1, 5, 2, 4, 3, 5]
```

Пробелы в списках

Более сложные проверки списков могут достигать нежелательной длины или становиться менее читаемыми. Хотя это менее распространено в примерах, можно разбить понимание списка на несколько строк следующим образом:

```
[
    x for x
    in 'foo'
    if x not in 'bar'
]
```

Словарь

Понимание словаря аналогично пониманию списка, за исключением того, что он создает объект словаря вместо списка.

Основной пример:

Python 2.x 2.7

```
{x: x * x for x in (1, 2, 3, 4)}
# Out: {1: 1, 2: 4, 3: 9, 4: 16}
```

что является еще одним способом написания:

```
dict((x, x * x) for x in (1, 2, 3, 4))
# Out: {1: 1, 2: 4, 3: 9, 4: 16}
```

Как и в случае со списком, мы можем использовать условное утверждение внутри понимания dict, чтобы создать только элементы dict, удовлетворяющие некоторому критерию.

Python 2.x 2.7

```
{name: len(name) for name in ('Stack', 'Overflow', 'Exchange') if len(name) > 6}
# Out: {'Exchange': 8, 'Overflow': 8}
```

Или, переписанный с использованием выражения генератора.

```
dict((name, len(name)) for name in ('Stack', 'Overflow', 'Exchange') if len(name) > 6)
# Out: {'Exchange': 8, 'Overflow': 8}
```

Начиная со словаря и использования словарного знака в качестве фильтра пары ключ-значение

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}
```

Клавиша переключения и значение словаря (инвертированный словарь)

Если у вас есть *указатель,* содержащий простые значения *хеширования* (дублирующиеся значения могут иметь неожиданные результаты):

```
my_dict = {1: 'a', 2: 'b', 3: 'c'}
```

и вы хотели поменять клавиши и значения, вы можете использовать несколько подходов в зависимости от стиля кодирования:

- `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

Если ваш словарь большой, подумайте об *импорте* *itertools* и используйте `izip` или `imap` .

Слияние словарей

Объедините словари и, возможно, переопределите старые значения с вложенным пониманием словаря.

```
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}
```

Тем не менее, распаковка словарей (PEP 448) может быть предпочтительной.

Python 3.x 3.5

```
{**dict1, **dict2}
# Out: {'w': 1, 'x': 2, 'y': 2, 'z': 2}
```

Примечание : в Python 3.0 были добавлены словарные словари и были добавлены в версии 2.0+, в отличие от списков, которые были добавлены в версии 2.0. Версии <2.7 могут

использовать выражения генератора и встроенный `dict()` для имитации поведения понимания словаря.

Выражения генератора

Выражения генератора очень похожи на списки. Основное различие заключается в том, что он не создает сразу полный набор результатов; он создает объект-генератор, который затем может быть повторен.

Например, см. Разницу в следующем коде:

```
# 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>
```

Это два очень разных объекта:

- представление списка возвращает объект `list` тогда как генераторное понимание возвращает `generator` .
- объекты `generator` не могут быть проиндексированы и используют `next` функцию для упорядочивания элементов.

Примечание . Мы используем `xrange` так как он также создает объект-генератор. Если мы будем использовать диапазон, будет создан список. Кроме того, `xrange` существует только в более поздней версии python 2. В python 3 `range` просто возвращает генератор. Дополнительные сведения см. В разделе « *Различия между диапазонами и* примерами *функций 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

> ПРИМЕЧАНИЕ. Функция `g.next()` должна быть заменена `next(g)` и `xrange` с `range` поскольку `Iterator.next()` и `xrange()` не существуют в Python 3.

Хотя оба они могут быть повторены аналогичным образом:

```
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
"""
```

Случаи применения

Выражения генератора оцениваются лениво, что означает, что они генерируют и возвращают каждое значение только при повторении генератора. Это часто бывает полезно при повторении с помощью больших наборов данных, избегая необходимости создания дубликата набора данных в памяти:

```
for square in (x**2 for x in range(1000000)):
    #do something
```

Другой распространенный вариант использования - избегать повторения по целому итерабельному, если это не является необходимым. В этом примере элемент извлекается из удаленного API с каждой итерацией `get_objects()` . Тысячи объектов могут существовать, их необходимо извлекать один за другим, и нам нужно знать только, существует ли объект, соответствующий шаблону. Используя выражение генератора, когда мы сталкиваемся с объектом, соответствующим шаблону.

```
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
```

Установить понимание

Установление понимания аналогично пониманию списка и словаря , но оно создает набор , который представляет собой неупорядоченный набор уникальных элементов.

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'])
```

Демо-версия

Имейте в виду, что наборы неупорядочены. Это означает, что порядок результатов в наборе может отличаться от порядка, представленного в приведенных выше примерах.

Примечание . Установить понимание доступно, поскольку python 2.7+, в отличие от системных списков, которые были добавлены в версии 2.0. В Python 2.2 до Python 2.6 функция `set()` может использоваться с выражением-генератором для получения того же результата:

Python 2.x 2.2

```
set(x for x in range(5))
# Out: {0, 1, 2, 3, 4}
```

Избегайте повторяющихся и дорогостоящих операций с использованием условной оговорки

Рассмотрим нижеприведенное понимание:

```
>>> 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, ...]
```

Это приводит к двум вызовам `f(x)` для 1000 значений `x` : один вызов для генерации значения, а другой для проверки условия `if` . Если `f(x)` является особенно дорогостоящей операцией, это может иметь значительные последствия для производительности. Хуже того, если вызов `f()` имеет побочные эффекты, он может иметь неожиданные результаты.

Вместо этого вы должны оценивать дорогостоящую операцию только один раз для каждого значения `x` , создавая промежуточное итерируемое (выражение генератора) следующим образом:

```
>>> [v for v in (f(x) for x in range(1000)) if v > 10]
[16, 25, 36, ...]
```

Или, используя встроенный эквивалент карты :

```
>>> [v for v in map(f, range(1000)) if v > 10]
[16, 25, 36, ...]
```

Другим способом, который может привести к более читаемому коду, является поместить частичный результат (`v` в предыдущем примере) в итерируемый (например, список или кортеж), а затем перебрать его. Поскольку `v` будет единственным элементом в итерабельном, результат состоит в том, что теперь мы имеем ссылку на вывод нашей медленной функции, вычисленной только один раз:

```
>>> [v for x in range(1000) for v in [f(x)] if v > 10]
```

```
[16, 25, 36, ...]
```

Однако на практике логика кода может быть более сложной, и важно сохранить ее читабельным. В общем случае для сложного однострочного слоя рекомендуется отдельная функция генератора :

```
>>> 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, ...]
```

Еще один способ предотвратить вычисление `f(x)` несколько раз - использовать декоратор `@functools.lru_cache()` (Python 3.2+) на `f(x)` . Таким образом, поскольку вывод `f` для входа `x` уже был вычислен один раз, второй вызов функции для первоначального понимания списка будет таким же быстрым, как поиск словаря. Этот подход использует memoization для повышения эффективности, что сопоставимо с использованием выражений генератора.

Скажем, вы должны сгладить список

```
l = [[1, 2, 3], [4, 5, 6], [7], [8, 9]]
```

Некоторые из методов могут быть следующими:

```
reduce(lambda x, y: x+y, l)

sum(l, [])

list(itertools.chain(*l))
```

Однако понимание списков обеспечит наилучшую временную сложность.

```
[item for sublist in l for item in sublist]
```

Ярлыки, основанные на + (включая подразумеваемое использование в сумме), являются, по необходимости, O (L ^ 2), когда есть L sublists - поскольку промежуточный список результатов продолжает увеличиваться, на каждом шаге появляется новый объект промежуточного результата выделено, и все элементы предыдущего промежуточного результата должны быть скопированы (а также несколько новых добавленных в конце). Таким образом (для простоты и без фактической потери общности) скажем, что у вас есть L подсписок из I предметов каждый: первые предметы I копируются взад и вперед L-1 раз, второй I - L-2 раза и т. Д .; общее количество копий I умножает сумму x для x от 1 до L, т. е. I * (L ** 2) / 2.

Понимание списка просто генерирует один список, один раз и копирует каждый элемент (от его первоначального места жительства до списка результатов) также ровно один раз.

Понимание, связанное с кортежами

Предложение `for` для определения списка может содержать более одной переменной:

```
[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]
```

Это так же , как регулярные `for` петель:

```
for x, y in [(1,2), (3,4), (5,6)]:
    print(x+y)
# 3
# 7
# 11
```

Обратите внимание, однако, если выражение, которое начинает понимать, является кортежем, тогда оно должно быть заключено в скобки:

```
[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)]
```

Подсчет событий с использованием понимания

Когда мы хотим подсчитать количество элементов в итерабельном, которые удовлетворяют некоторому условию, мы можем использовать понимание для создания идиоматического синтаксиса:

```
# 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
```

Основную концепцию можно обобщить следующим образом:

1. Итерации по элементам в `range(1000)` .
2. Сцепить все необходимое , `if` условия.
3. Используйте 1 как *выражение,* чтобы вернуть 1 для каждого элемента, который соответствует условиям.

4. Суммируйте все `1` с для определения количества элементов, соответствующих условиям.

Примечание . Здесь мы не собираем `1` с в списке (обратите внимание на отсутствие квадратных скобок), но мы передаем их непосредственно `sum` функции, суммирующей их. Это называется *выражением генератора* , которое аналогично пониманию.

Изменение типов в списке

Количественные данные часто считываются как строки, которые перед обработкой необходимо преобразовать в числовые типы. Типы всех элементов списка могут быть преобразованы либо с помощью функции « Пояснение списка», либо с помощью функции `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]
```

Прочитайте Перечисление списков онлайн: https://riptutorial.com/ru/python/topic/196/перечисление-списков

глава 130: Плагины и расширения

Examples

Примеси

В объектно-ориентированном языке программирования mixin - это класс, который содержит методы для использования другими классами, не будучи родительским классом этих других классов. Как другие классы получают доступ к методам mixin, зависит от языка.

Он обеспечивает механизм множественного наследования, позволяя нескольким классам использовать общую функциональность, но без сложной семантики множественного наследования. Миксины полезны, когда программист хочет разделить функциональность между разными классами. Вместо повторения одного и того же кода снова и снова, общая функциональность может быть просто сгруппирована в mixin и затем унаследована в каждый класс, который ее требует.

Когда мы используем несколько микстинов, важно отметить порядок миксинов. вот простой пример:

```
class Mixin1(object):
    def test(self):
        print "Mixin1"

class Mixin2(object):
    def test(self):
        print "Mixin2"

class MyClass(Mixin1, Mixin2):
    pass
```

В этом примере мы вызываем `MyClass` и метод `test` ,

```
>>> obj = MyClass()
>>> obj.test()
Mixin1
```

Результат должен быть Mixin1, потому что порядок слева направо. Это может показать неожиданные результаты, когда суперклассы добавляют с ним. Таким образом, обратный порядок более хорош именно так:

```
class MyClass(Mixin2, Mixin1):
    pass
```

Результат будет:

```
>>> obj = MyClass()
>>> obj.test()
Mixin2
```

Микшины могут использоваться для определения пользовательских плагинов.

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.
```

Плагины с настраиваемыми классами

В Python 3.6 PEP 487 добавил специальный метод `__init_subclass__` , который упрощает и расширяет настройку класса без использования метаклассов . Следовательно, эта функция позволяет создавать простые плагины . Здесь мы демонстрируем эту функцию путем изменения предыдущего примера :

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.")
```

Результаты:

```
PluginA().test()
# Base.
# Plugin A.

PluginB().test()
# Base.
# Plugin B.

Base.plugins
# [__main__.PluginA, __main__.PluginB]
```

Прочитайте Плагины и расширения онлайн: https://riptutorial.com/ru/python/topic/4724/плагины-и-расширения

глава 131: Побитовые операторы

Вступление

Побитовые операции изменяют двоичные строки на уровне бит. Эти операции невероятно просты и напрямую поддерживаются процессором. Эти несколько операций необходимы при работе с драйверами устройств, низкоуровневой графикой, криптографией и сетевыми коммуникациями. В этом разделе приведены полезные сведения и примеры побитовых операторов Python.

Синтаксис

- x << y # Побитовый сдвиг влево
- x >> y # Побитовый правый сдвиг
- x & y # Побитовое И
- x | y # Побитовое ИЛИ
- ~ x # Побитовое NOT
- x ^ y # Побитовое XOR

Examples

Побитовое И

Оператор `&` будет выполнять двоичный код **AND** , где бит копируется, если он существует в **обоих** операндах. Это означает:

```
# 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
```

Побитовое ИЛИ

`|` оператор будет выполнять двоичный «или», где бит копируется, если он существует в любом из операндов. Это означает:

```
# 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
```

Побитовое XOR (Исключительное ИЛИ)

Оператор `^` будет выполнять двоичный **XOR,** в котором двоичный код `1` копируется тогда и только тогда, когда он является значением точно **одного** операнда. Другой способ заявить, что результат равен `1` только если операнды разные. Примеры включают:

```
# 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
```

Побитовый сдвиг влево

Оператор `<<` выполнит поразрядный «сдвиг влево», где значение левого операнда будет перемещено влево на число бит, заданное правым операндом.

```
# 2 = 0b10
2 << 2
# Out: 8
# 8 = 0b1000

bin(2 << 2)
# Out: 0b1000
```

Выполнение сдвига левого бита `1` эквивалентно умножению на `2` :

```
7 << 1
# Out: 14
```

Выполнение сдвига левого бита `n` эквивалентно умножению на `2**n` :

```
3 << 4
# Out: 48
```

Побитовый правый сдвиг

Оператор `>>` выполнит поразрядный «сдвиг вправо», где значение левого операнда будет перемещаться вправо на количество бит, заданное правым операндом.

```
# 8 = 0b1000
8 >> 2
# Out: 2
# 2 = 0b10

bin(8 >> 2)
# Out: 0b10
```

Выполнение смещения правого бита `1` эквивалентно целочисленному делению на `2` :

```
36 >> 1
# Out: 18

15 >> 1
# Out: 7
```

Выполнение правого битового сдвига `n` эквивалентно целочисленному делению на `2**n` :

```
48 >> 4
# Out: 3

59 >> 3
# Out: 7
```

Побитовое НЕ

Оператор `~` перевернет все биты числа. Поскольку компьютеры используют подписанные представления чисел, в первую очередь, обозначение дополнений двух для кодирования отрицательных двоичных чисел, где отрицательные числа записываются с ведущим (1) вместо начального нуля (0).

Это означает, что если вы использовали 8 бит для представления чисел ваших двух дополнений, вы будете обрабатывать шаблоны от `0000 0000` до `0111 1111` для представления чисел от 0 до 127 и зарезервировать `1xxx xxxx` для представления отрицательных чисел.

Восьмибитовые номера с двумя номерами дополнений

Биты	Беззнаковое значение	Значение второго уровня
0000 0000	0	0
0000 0001	1	1
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

По сути, это означает, что в то время как `1010 0110` имеет неподписанное значение 166 (получено путем добавления `(128 * 1) + (64 * 0) + (32 * 1) + (16 * 0) + (8 * 0) + (4 * 1) + (2 * 1) + (1 * 0)`), он имеет значение с двумя дополнениями -90 (полученное добавлением `(128 * 1) - (64 * 0) - (32 * 1) - (16 * 0) - (8 * 0) - (4 * 1) - (2 * 1) - (1 * 0)` и дополняя значение).

Таким образом, отрицательные числа варьируются до -128 (`1000 0000`). Нуль (0) представляется как `0000 0000` , а минус один (-1) как `1111 1111` .

В общем, это означает, что `~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
```

Обратите внимание , что общий эффект этой операции при применении к положительным числам можно суммировать:

```
~n -> -|n+1|
```

И затем, когда применяется к отрицательным числам, соответствующий эффект:

```
~-n -> |n-1|
```

Следующие примеры иллюстрируют это последнее правило ...

```
# -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
```

Операции на месте

Все побитовые операторы (кроме `~`) имеют свои собственные версии

```
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
```

Прочитайте Побитовые операторы онлайн: https://riptutorial.com/ru/python/topic/730/побитовые-операторы

глава 132: Повышение пользовательских ошибок / исключений

Вступление

У Python есть много встроенных исключений, которые заставляют вашу программу выводить ошибку, когда что-то в ней идет не так.

Однако иногда вам может понадобиться создавать пользовательские исключения, которые служат вашей цели.

В Python пользователи могут определять такие исключения, создавая новый класс. Этот класс исключений должен быть получен прямо или косвенно из класса Exception. Большинство встроенных исключений также получены из этого класса.

Examples

Пользовательское исключение

Здесь мы создали пользовательское исключение, называемое CustomError, которое получено из класса Exception. Это новое исключение может быть поднято, как и другие исключения, с помощью оператора raise с дополнительным сообщением об ошибке.

```
class CustomError(Exception):
       pass

x = 1

if x == 1:
    raise CustomError('This is custom error')
```

Выход:

```
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
```

Поймать обычное исключение

В этом примере показано, как поймать пользовательское исключение

```
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))
```

Выход:

```
Catched CustomError :Can you catch me ?
```

Прочитайте Повышение пользовательских ошибок / исключений онлайн: https://riptutorial.com/ru/python/topic/10882/повышение-пользовательских-ошибок---исключений

глава 133: Подключение Python к SQL Server

Examples

Подключение к серверу, создание таблицы, данных запроса

Установите пакет:

```
$ 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()
```

Вы можете что-либо сделать, если ваша работа связана с выражениями SQL, просто передайте эти выражения методу выполнения (операции CRUD).

Для справки , вызова хранимой процедуры, обработки ошибок или проверки большего числа примеров: pymssql.org

Прочитайте Подключение Python к SQL Server онлайн: https://riptutorial.com/ru/python/topic/7985/подключение-python-к-sql-server

глава 134: Подключение Secure Shell в Python

параметры

параметр	использование
имя хоста	Этот параметр указывает хосту, для которого необходимо установить соединение
имя пользователя	имя пользователя, необходимое для доступа к хосту
порт	хост-порт
пароль	пароль для учетной записи

Examples

Подключение 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
```

Прочитайте Подключение Secure Shell в Python онлайн:
https://riptutorial.com/ru/python/topic/5709/подключение-secure-shell-в-python

глава 135: Подкоманды CLI с точным выводом справки

Вступление

Различные способы создания подкоманд, таких как `hg` или `svn` с точным интерфейсом командной строки и вывод справки, как показано в разделе «Примечания».

Аргументы командной строки Parsing охватывают более широкую тему анализа аргументов.

замечания

Различные способы создания подкоманд, таких как `hg` или `svn` с интерфейсом командной строки, показанным в справочном сообщении:

```
usage: sub <command>

commands:

  status -  show status
  list   -  print list
```

Examples

Родной способ (без библиотек)

```
"""
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())
```

Выход без аргументов:

```
usage: sub <command>

commands:

  status -  show status
  list   -  print list
```

Плюсы:

- нет депов
- каждый должен уметь читать, что
- полный контроль над форматированием справки

argparse (формат справки по умолчанию)

```
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())
```

Выход без аргументов:

```
usage: sub {status,list} ...

positional arguments:
  {status,list}
    status       show status
    list         print list
```

Плюсы:

- поставляется с Python

- синтаксический разбор включен

argparse (формат форматирования помощи)

Расширенная версия http://www.riptutorial.com/python/example/25282/argparse--default-help-formatter-, которая фиксировала выход справки.

```
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())
```

Выход без аргументов:

```
usage: sub <command>

commands:

  status -   show status
  list   -   print list
```

Прочитайте Подкоманды CLI с точным выводом справки онлайн: https://riptutorial.com/ru/python/topic/7701/подкоманды-cli-с-точным-выводом-справки

глава 136: подсчет

Examples

Подсчет всех происшествий всех элементов в истребителе: 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
```

`collections.Counter` могут использоваться для любых итераций и подсчитывают каждое событие для каждого элемента.

Примечание . Одно исключение - если задан какой-либо `dict` или другой `collections.Mapping` Подобный класс сопоставляется, то он не будет считать их, а создает счетчик с такими значениями:

```
Counter({"e": 2})
# Out: Counter({"e": 2})

Counter({"e": "e"})        # warning Counter does not verify the values are int
# Out: Counter({"e": "e"})
```

Получение наиболее общей ценности (-s): collections.Counter.most_common ()

Подсчет *ключей* `Mapping` невозможен с помощью `collections.Counter` но мы можем подсчитать *значения* :

```
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})
```

Наиболее распространенные элементы доступны большинству `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)]
```

Подсчет появления одного элемента в последовательности: list.count () и 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
```

Подсчет появления подстроки в строке: str.count ()

```
astring = 'thisisashorttext'
astring.count('t')
# Out: 4
```

Это работает даже для подстрок длиннее одного символа:

```
astring.count('th')
# Out: 1
astring.count('is')
# Out: 2
astring.count('text')
# Out: 1
```

что было бы невозможно с `collections.Counter` который учитывает только одиночные символы:

```
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})
```

Подсчет значений в массиве numpy

Чтобы подсчитать количество значений в массиве numpy. Это будет работать:

```
>>> import numpy as np
>>> a=np.array([0,3,4,3,5,4,7])
>>> print np.sum(a==3)
2
```

Логика заключается в том, что булевский оператор создает массив, где все вхождения запрошенных значений равны 1, а все остальные равны нулю. Таким образом, суммирование дает количество случаев. Это работает для массивов любой формы или типа.

Есть два метода, которые я использую для подсчета вхождения всех уникальных значений в numpy. Уникальный и двухуровневый. Уникальный автоматически сглаживает многомерные массивы, а bincount работает только с 1d массивами, содержащими только положительные целые числа.

```
>>> 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]
```

Если ваши данные являются массивами numpy, то, как правило, намного быстрее использовать методы numpy, а затем преобразовывать ваши данные в общие методы.

Прочитайте подсчет онлайн: https://riptutorial.com/ru/python/topic/476/подсчет

глава 137: подушка

Examples

Чтение файла изображения

```
from PIL import Image

im = Image.open("Image.bmp")
```

Преобразование файлов в 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)
```

Прочитайте подушка онлайн: https://riptutorial.com/ru/python/topic/6841/подушка

глава 138: поиск

замечания

Все алгоритмы поиска на итерациях, содержащие `n` элементов, имеют сложность `O(n)` . Только специализированные алгоритмы, такие как `bisect.bisect_left()` могут быть быстрее с сложностью `O(log(n))` .

Examples

Получение индекса для строк: str.index (), str.rindex () и str.find (), str.rfind ()

`String` также имеет `index` метод, но также более продвинутые параметры и дополнительную `str.find` . Для обоих из них существует дополнительный *обратный* метод.

```
astring = 'Hello on StackOverflow'
astring.index('o')  # 4
astring.rindex('o') # 20

astring.find('o')   # 4
astring.rfind('o')  # 20
```

Разница между `index` / `rindex` и `find` / `rfind` заключается в том, что происходит, если подстрока не найдена в строке:

```
astring.index('q') # ValueError: substring not found
astring.find('q')  # -1
```

Все эти методы позволяют начинать и заканчивать индекс:

```
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: подстрока не найдена

```
astring.rindex('o', 20) # 20
astring.rindex('o', 19) # 20 - still from left to right

astring.rindex('o', 4, 7) # 6
```

Поиск элемента

Все встроенные в коллекции в Python реализовать способ проверить членство элемента с использованием `in` .

Список

```
alist = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
5 in alist   # True
10 in alist  # False
```

Кортеж

```
atuple = ('0', '1', '2', '3', '4')
4 in atuple    # False
'4' in atuple  # True
```

строка

```
astring = 'i am a string'
'a' in astring   # True
'am' in astring  # True
'I' in astring   # False
```

Задавать

```
aset = {(10, 10), (20, 20), (30, 30)}
(10, 10) in aset  # True
10 in aset        # False
```

Dict

`dict` немного особенный: нормальный `in` проверяет только *ключи.* Если вы хотите искать в *значениях,* вам нужно указать его. То же самое, если вы хотите найти пары *ключ-значение* .

```
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
```

Получение списка индексов и кортежей: list.index (), tuple.index ()

`list` и `tuple` имеют `index` метод для получения позиции элемента:

```
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 нет в списке

Но возвращает только позицию первого найденного элемента:

```
atuple = (10, 16, 26, 5, 2, 19, 105, 26)
atuple.index(26)   # 2
atuple[2]          # 26
atuple[7]          # 26 - is also 26!
```

Поиск ключа (ов) для значения в dict

`dict` не имеет встроенного метода для поиска значения или ключа, поскольку *словари* неупорядочены. Вы можете создать функцию, которая получает ключ (или ключи) для указанного значения:

```
def getKeysForValue(dictionary, value):
    foundkeys = []
    for keys in dictionary:
        if dictionary[key] == value:
            foundkeys.append(key)
    return foundkeys
```

Это также можно записать в виде эквивалентного списка:

```
def getKeysForValueComp(dictionary, value):
    return [key for key in dictionary if dictionary[key] == value]
```

Если вам нужен только один найденный ключ:

```
def getOneKeyForValue(dictionary, value):
    return next(key for key in dictionary if dictionary[key] == value)
```

Первые две функции возвращают `list` всех `keys` , имеющих указанное значение:

```
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) # []
```

Другой возвращает только один ключ:

```
getOneKeyForValue(adict, 10)   # 'c'  - depending on the circumstances this could also be 'a'
getOneKeyForValue(adict, 20)   # 'b'
```

и вызвать `StopIteration` - `Exception` если значение не указано в `dict` :

```
getOneKeyForValue(adict, 25)
```

StopIteration

Получение индекса для отсортированных последовательностей: bisect.bisect_left ()

Сортированные последовательности позволяют использовать более быстрые алгоритмы поиска: `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

Для очень больших **отсортированных последовательностей** коэффициент усиления может быть довольно высоким. В случае первого поиска примерно в 500 раз быстрее:

```
%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
```

Хотя это немного медленнее, если элемент является одним из первых:

```
%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
```

Поиск вложенных последовательностей

Поиск во вложенных последовательностях, таких как `list` `tuple` требует подхода, такого как поиск ключей для значений в `dict` но нуждается в настраиваемых функциях.

Индекс самой внешней последовательности, если значение было найдено в последовательности:

```
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
```

или индекс внешней и внутренней последовательности:

```
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
```

В общем случае (*не всегда*) использование `next` и **генераторное выражение** с условиями поиска первого вхождения искомого значения является наиболее эффективным подходом.

Поиск в пользовательских классах: __contains__ и __iter__

Для того, чтобы разрешить использование `in` пользовательских классах класса должен либо предоставить магический метод `__contains__` или, если это невозможно, в `__iter__` -метод.

Предположим, у вас есть класс, содержащий `list` `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))
```

Использование тестирования членства возможно при использовании `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__.
```

даже после удаления метода `__contains__` :

```
del ListList.__contains__
5 in a     # True
# Prints: Using __iter__.
```

Примечание: зацикливание `in` (как `for i in a`) всегда будет использовать `__iter__` даже если класс реализует `__contains__` метод.

Прочитайте поиск онлайн: https://riptutorial.com/ru/python/topic/350/поиск

глава 139: Полиморфизм

Examples

Основной полиморфизм

Полиморфизм - это способность выполнять действие над объектом независимо от его типа. Обычно это достигается путем создания базового класса и наличия двух или более подклассов, которые реализуют методы с одной и той же сигнатурой. Любые другие функции или методы, которые манипулируют этими объектами, могут вызывать одни и те же методы независимо от того, к какому типу объекта он работает, не требуя сначала проверки типа. В объектно-ориентированной терминологии, когда класс X расширяет класс Y, тогда Y называется суперклассом или базовым классом, а X называется подклассом или производным классом.

```
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)
```

Мы должны увидеть этот результат:

> Никто
> 4
> 6,0

Что происходит без полиморфизма?

Без полиморфизма может потребоваться проверка типа перед выполнением действия над объектом для определения правильного метода вызова. Следующий **пример счетчика** выполняет ту же задачу, что и предыдущий код, но без использования полиморфизма `get_area()` должна выполнять больше работы.

```
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)
```

Мы должны увидеть этот результат:

> 4
> 6,0

Важная заметка
Обратите внимание, что классы, используемые в примере счетчика, являются классами «нового стиля» и неявно наследуются от класса объекта, если используется Python 3. Полиморфизм будет работать как в Python 2.x, так и 3.x, но код встречного примера полиморфизма вызовет исключение, если он запущен в интерпретаторе Python 2.x, потому что type (input_obj). **имя** будет возвращать «экземпляр» вместо имени класса, если они явно не наследуются от объекта, в результате чего область никогда не назначается.

Утиная печать

Полиморфизм без наследования в виде утиной печати, доступный в Python из-за его системы динамического набора. Это означает, что до тех пор, пока классы содержат одни и те же методы, интерпретатор Python не различает их, так как только проверка вызовов происходит во время выполнения.

```
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)
```

Выход:

> Quaaaaaack!
> Утка имеет белые и серые перья.
> Человек имитирует утку.
> Человек берет перо с земли и показывает его.

Прочитайте Полиморфизм онлайн: https://riptutorial.com/ru/python/topic/5100/полиморфизм

глава 140: Пользовательские методы

Examples

Создание пользовательских объектов метода

Пользовательские объекты метода могут создаваться при получении атрибута класса (возможно, через экземпляр этого класса), если этот атрибут представляет собой определяемый пользователем объект функции, несвязанный определяемый пользователем объект метода или объект метода класса.

```
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'>>
```

Когда атрибут является определяемым пользователем объектом метода, новый объект метода создается только в том случае, если класс, из которого он извлекается, является таким же, как или производным классом класса, хранящегося в исходном объекте метода; в противном случае исходный объект метода используется как есть.

```
# 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
```

Пример черепахи

Ниже приведен пример использования пользовательской функции, которая с легкостью называется множественным (∞) раза в сценарии.

```
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
```

Прочитайте Пользовательские методы онлайн: https://riptutorial.com/ru/python/topic/3965/пользовательские-методы

глава 141: Последовательная связь Python (pyserial)

Синтаксис

- ser.read (размер = 1)
- ser.readline ()
- ser.write ()

параметры

параметр	подробности
порт	Имя устройства eg / dev / ttyUSB0 на GNU / Linux или COM3 в Windows.
бод	baudrate type: int default: 9600 стандартных значений: 50, 75, 110, 134, 150, 200, 300, 600, 1200, 1800, 2400, 4800, 9600, 19200, 38400, 57600, 115200

замечания

Для получения дополнительной информации проверьте документацию pyserial

Examples

Инициализировать последовательное устройство

```
import serial
#Serial takes these two parameters: serial device and baudrate
ser = serial.Serial('/dev/ttyUSB0', 9600)
```

Чтение из последовательного порта

Инициализировать последовательное устройство

```
import serial
#Serial takes two parameters: serial device and baudrate
ser = serial.Serial('/dev/ttyUSB0', 9600)
```

читать один байт с последовательного устройства

```
data = ser.read()
```

читать заданное количество байтов с последовательного устройства

```
data = ser.read(size=5)
```

для чтения одной строки с последовательного устройства.

```
data = ser.readline()
```

для чтения данных с последовательного устройства, пока над ним что-то написано.

```
#for python2.7
data = ser.read(ser.inWaiting())

#for python3
ser.read(ser.inWaiting)
```

Проверьте, какие последовательные порты доступны на вашем компьютере

Чтобы получить список доступных последовательных портов, используйте

```
python -m serial.tools.list_ports
```

в командной строке или

```
from serial.tools import list_ports
list_ports.comports()  # Outputs list of available serial ports
```

из оболочки Python.

Прочитайте Последовательная связь Python (pyserial) онлайн: https://riptutorial.com/ru/python/topic/5744/последовательная-связь-python--pyserial-

глава 142: Построение графика с помощью Matplotlib

Вступление

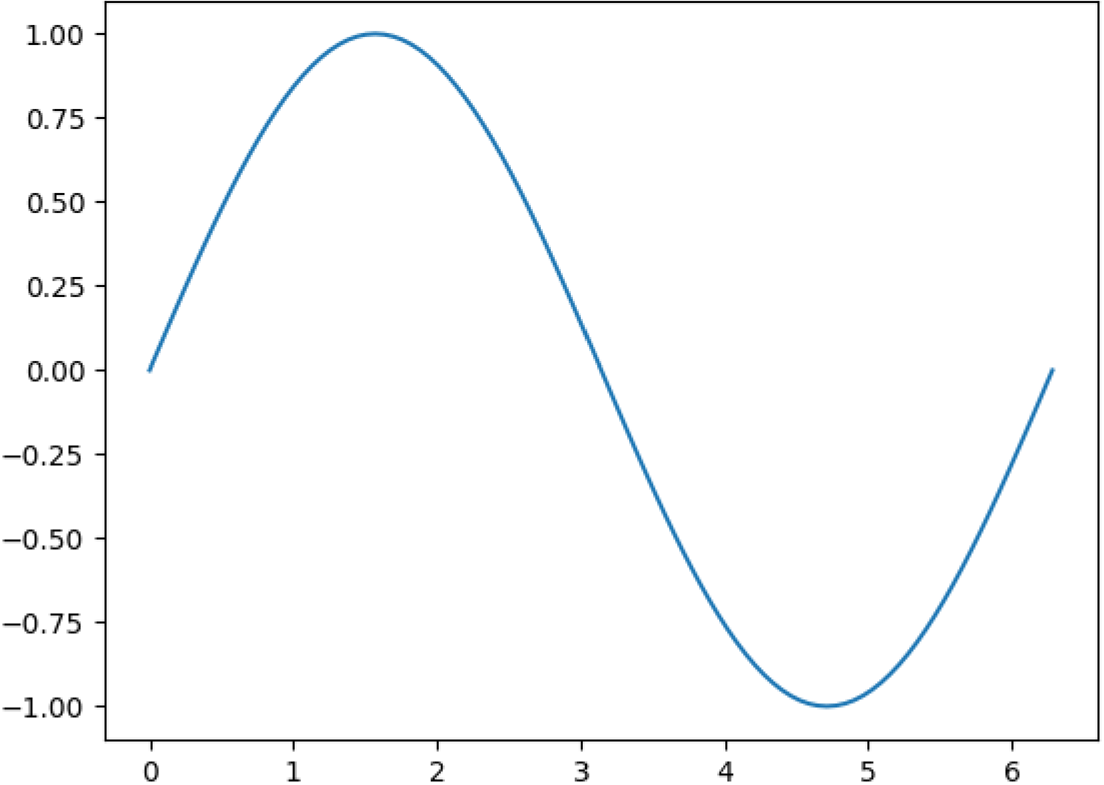
Matplotlib (https://matplotlib.org/) - это библиотека для 2D-построения на основе NumPy. Вот несколько основных примеров. Дополнительные примеры можно найти в официальной документации (https://matplotlib.org/2.0.2/gallery.html и https://matplotlib.org/2.0.2/examples/index.html), а также в http: //www.riptutorial.com/topic/881

Examples

Простой участок в Matplotlib

В этом примере показано, как создать простую синусовую кривую с использованием **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()
```

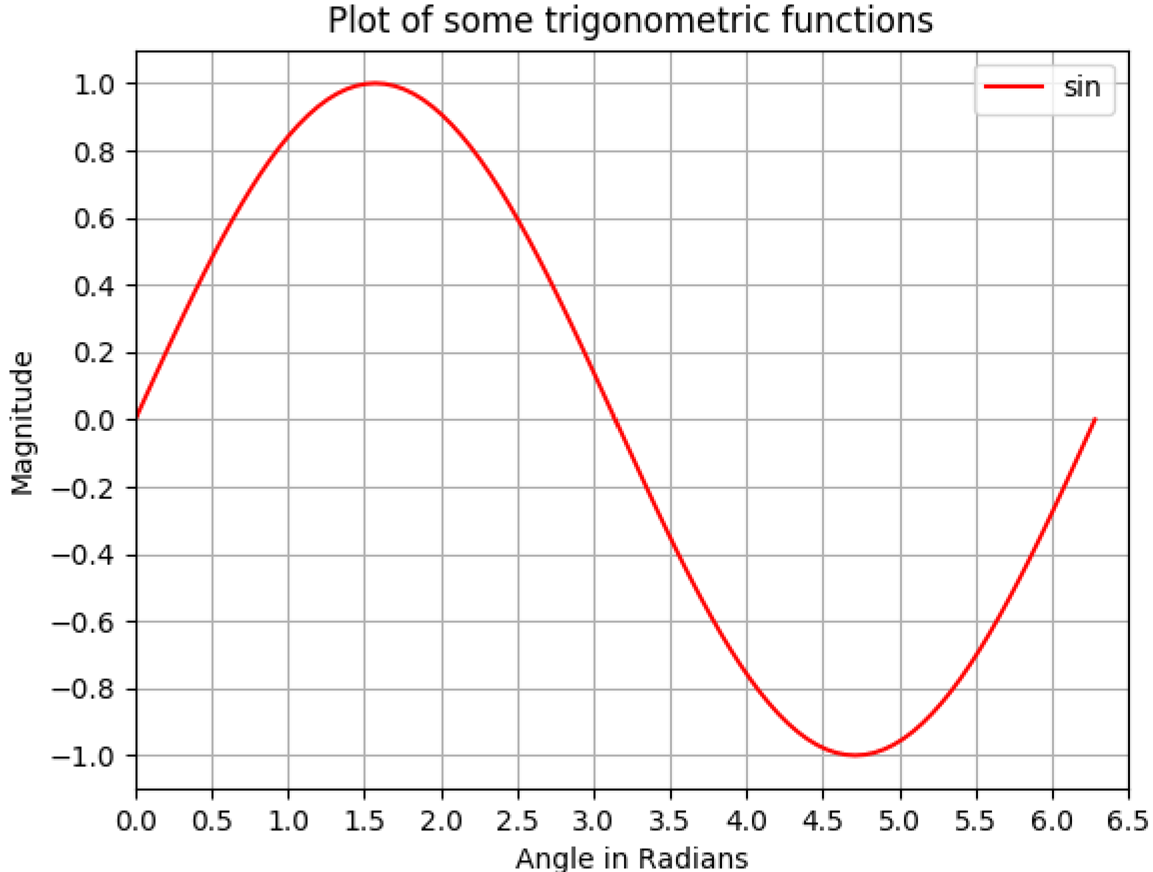


Добавление большего количества функций в простой график: метки осей, название, тики оси, сетка и легенда

В этом примере мы берем график кривой синуса и добавляем к нему дополнительные функции; а именно названия, осевые метки, название, тики оси, сетку и легенду.

```
# 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()
```

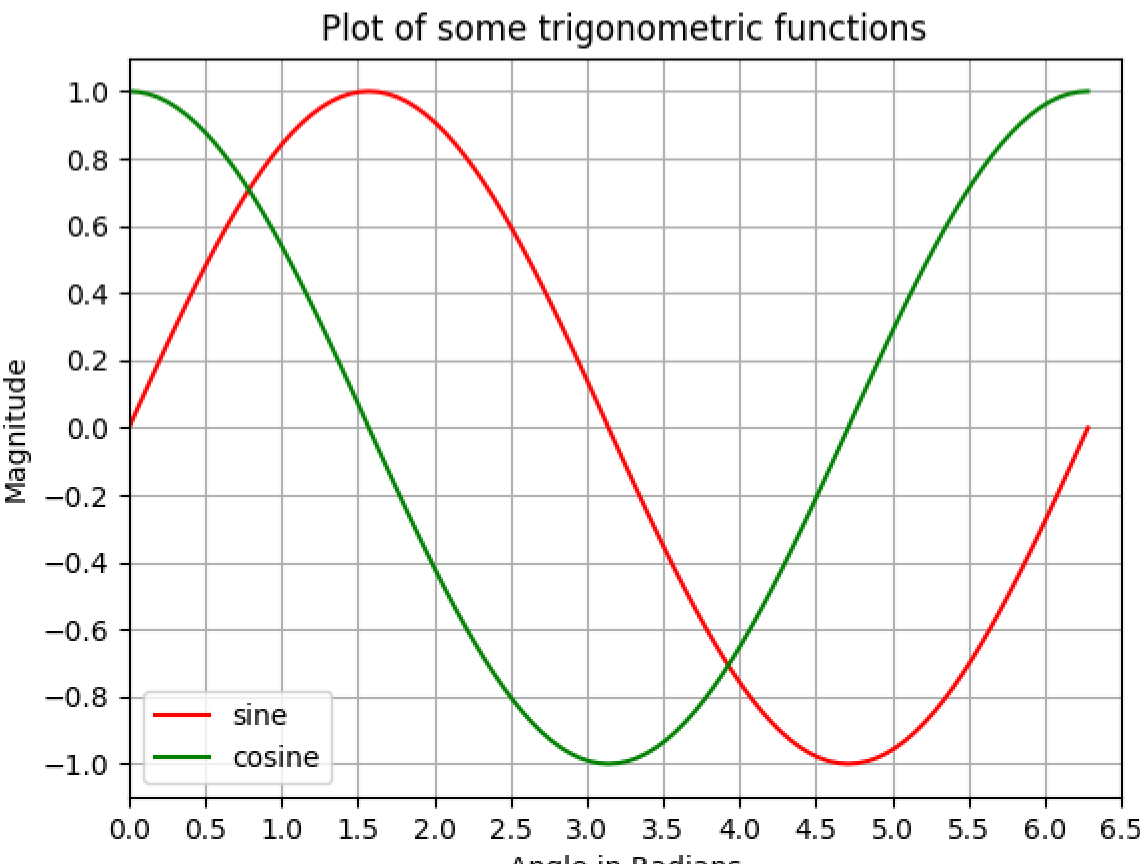


Создание нескольких графиков на одном рисунке с помощью наложения, аналогичного MATLAB

В этом примере кривая синуса и кривая косинуса изображены на том же рисунке, накладывая графики друг на друга.

```
# 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()
```

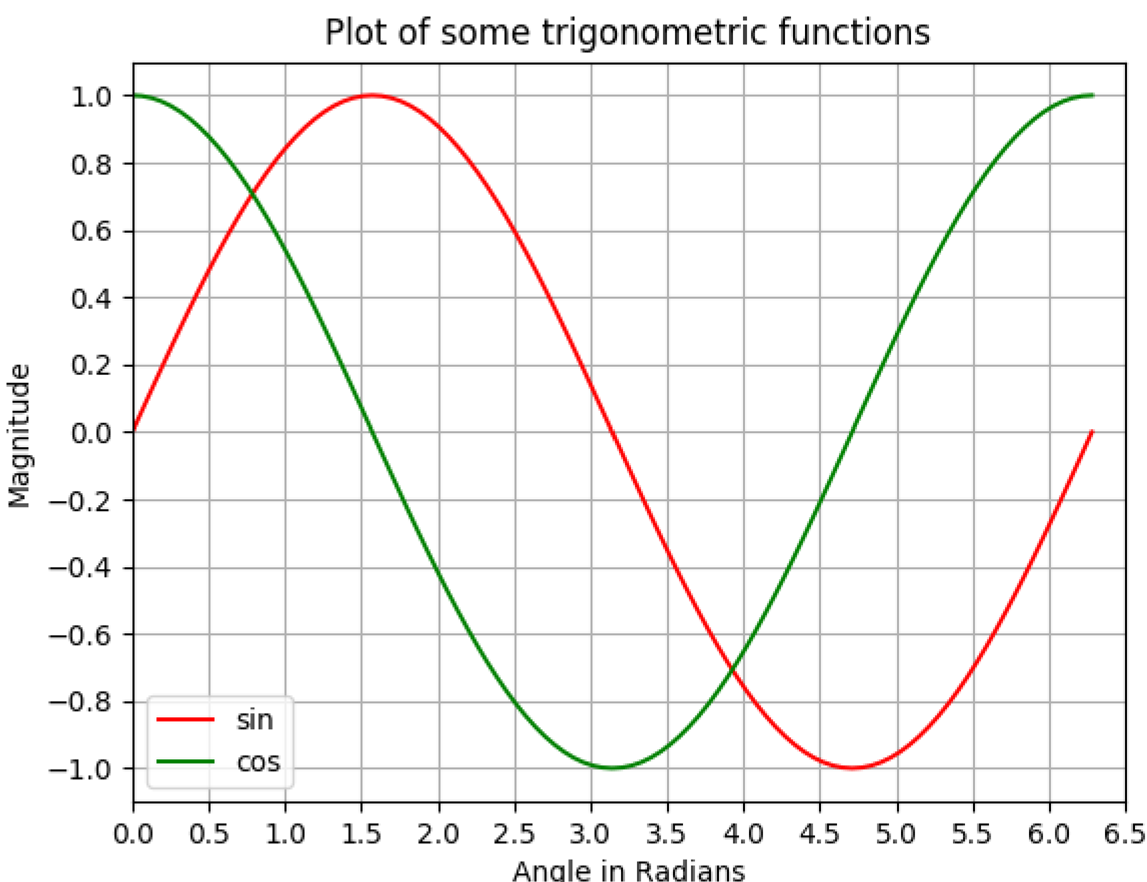


Создание нескольких сюжетов на одном рисунке с использованием наложения сюжетов с отдельными командами сюжета

Как и в предыдущем примере, здесь кривая синуса и косинуса изображается на том же рисунке с использованием отдельных команд графика. Это больше Pythonic и может использоваться для получения отдельных дескрипторов для каждого сюжета.

```
# 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()
```

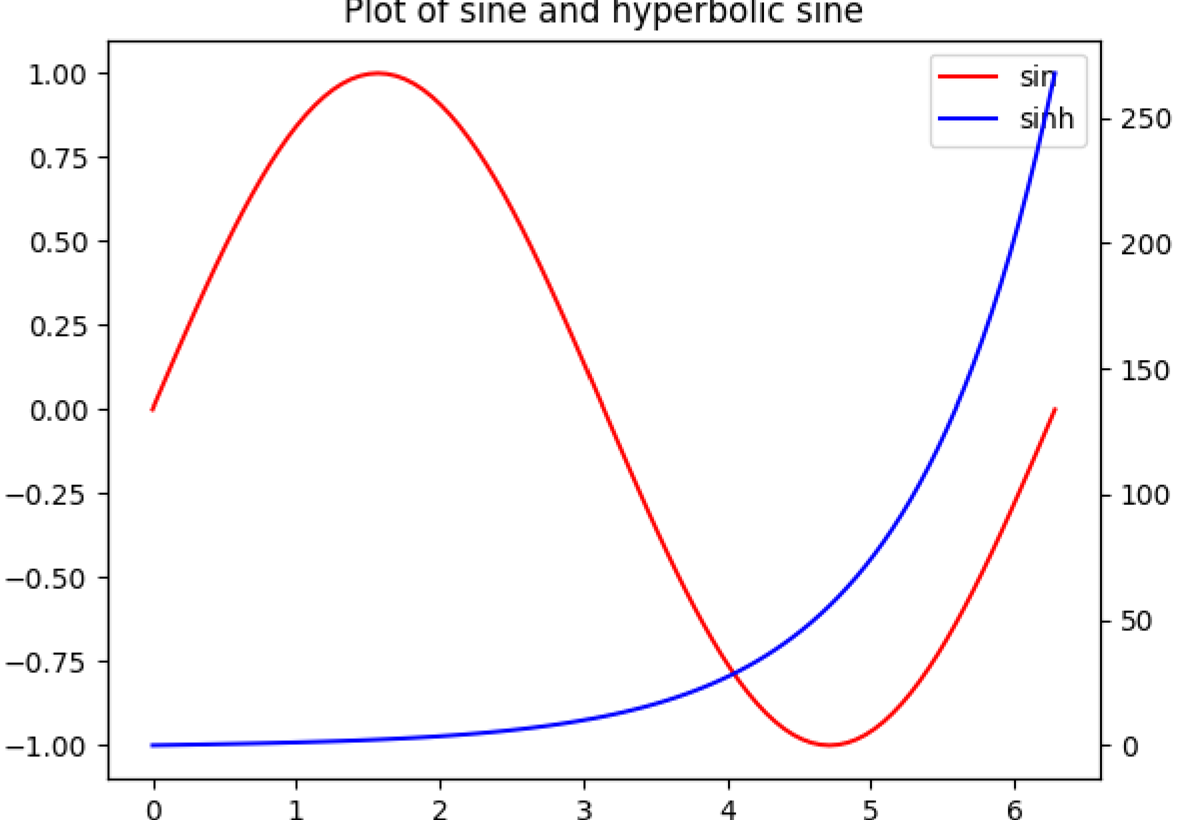


Графики с общей осью X, но с другой осью Y: использование doublex ()

В этом примере мы построим кривую синуса и гиперболическую синусовую кривую на том же графике с общей осью x, имеющей разную ось y. Это достигается с помощью команды **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()
```

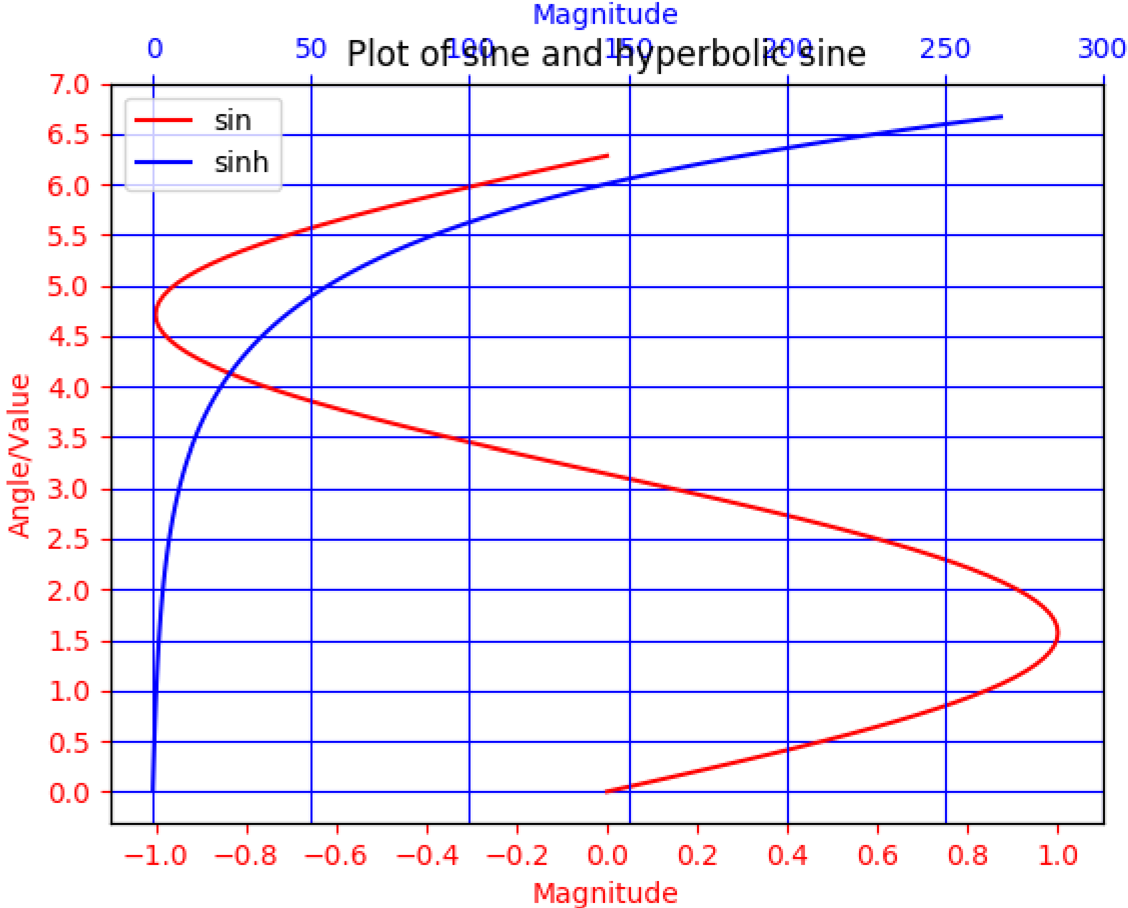


Графики с общей осью Y и другой осью X с использованием twiny ()

В этом примере график с кривыми, имеющими общую ось y, но с другой осью x, **показан** с использованием **метода twiny ()** . Кроме того, к сюжете добавляются некоторые дополнительные функции, такие как название, легенда, метки, сетки, тики и цвета осей.

```
# 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()
```



Прочитайте Построение графика с помощью Matplotlib онлайн: https://riptutorial.com/ru/python/topic/10264/построение-графика-с-помощью-matplotlib

глава 143: Примеси

Синтаксис

- class *ClassName* (*MainClass* , *Mixin1* , *Mixin2* , ...): # Используется для объявления класса с именем *ClassName* , основным (первым) классом *MainClass* и mixins *Mixin1* , *Mixin2* и т. д.
- class *ClassName* (*Mixin1* , *MainClass* , *Mixin2* , ...): # «Основной» класс не должен быть первым классом; нет никакой разницы между ним и mixin

замечания

Добавление mixin в класс очень похоже на добавление суперкласса, потому что это в значительной степени именно это. Объект класса с mixin *Foo* также будет экземпляром *Foo* , а `isinstance(instance, Foo)` вернет true

Examples

Mixin

Mixin - это набор свойств и методов, которые могут использоваться в разных классах, которые *не* относятся к базовому классу. В языках объектно-ориентированного программирования вы обычно используете *наследование* для предоставления одинаковым функциям объектов разных классов; если набор объектов обладает некоторой способностью, вы помещаете эту способность в базовый класс, на который оба объекта *наследуются* .

> Например, скажем, у вас есть классы `Car` , `Boat` и `Plane` . Объекты из всех этих классов имеют возможность путешествовать, поэтому они получают функцию `travel` . В этом сценарии все они перемещаются одним и тем же основным способом; путем получения маршрута и перемещения по нему. Чтобы реализовать эту функцию, вы можете получить все классы из `Vehicle` и поместить функцию в этот общий класс:

```
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):
   ...
```

С помощью этого кода вы можете позвонить в `travel` по машине (`car.travel("Montana")`), лодку (`boat.travel("Hawaii")`) и самолет (`plane.travel("France")`)

Однако, что, если у вас есть функциональность, недоступная базовому классу? Скажем, например, вы хотите дать `Car` радио и возможность использовать его для воспроизведения песни на радиостанции с помощью `play_song_on_station` , но у вас также есть `Clock` которые также могут использовать радио. `Car` и `Clock` могут делиться базовым классом (`Machine`). Однако не все машины могут воспроизводить песни; `Boat` и `Plane` не могут (по крайней мере, в этом примере). Итак, как вы это делаете без дублирования кода? Вы можете использовать mixin. В Python предоставление класса mixin так же просто, как добавление его в список подклассов, например

```
class Foo(main_super, mixin): ...
```

`Foo` наследует все свойства и методы `main_super` , а также свойства `mixin` .

Итак, чтобы дать классу `Car` и часы возможность использовать радио, вы можете переопределить `Car` из последнего примера и написать это:

```
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):
   ...
```

Теперь вы можете вызвать `car.play_song_on_station(98.7)` и `clock.play_song_on_station(101.3)` , но не что-то вроде `boat.play_song_on_station(100.5)`

Важная вещь с mixins заключается в том, что они позволяют добавлять функциональные возможности к различным объектам, которые не разделяют «основной» подкласс с этой функциональностью, но тем не менее все равно используют код для него. Без миксинов

делать что-то вроде приведенного выше примера было бы намного сложнее и / или потребовало бы повторения.

Переопределение методов в миксинах

Mixins - это своего рода класс, который используется для «смешения» дополнительных свойств и методов в классе. Это обычно прекрасно, потому что много раз классы mixin не переопределяют друг друга или методы базового класса. Но если вы переопределяете методы или свойства в ваших микшинах, это может привести к неожиданным результатам, поскольку в Python иерархия классов определяется справа налево.

Например, возьмите следующие классы

```
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
```

В этом случае класс Mixin2 является базовым классом, расширенным Mixin1 и, наконец, BaseClass. Таким образом, если мы выполним следующий фрагмент кода:

```
>>> x = MyClass()
>>> x.test()
Base
```

Мы видим, что результат возвращается из класса Base. Это может привести к непредвиденным ошибкам в логике вашего кода, и их необходимо учитывать и учитывать

Прочитайте Примеси онлайн: https://riptutorial.com/ru/python/topic/4359/примеси

глава 144: Приоритет оператора

Вступление

Операторы Python имеют заданный **порядок приоритета** , который определяет, какие операторы сначала оцениваются в потенциально неоднозначном выражении. Например, в выражении 3 * 2 + 7 первое 3 умножается на 2, а затем результат добавляется к 7, что дает 13. Выражение не оценивается наоборот, потому что * имеет более высокий приоритет, чем +.

Ниже приведен список операторов по приоритету и краткое описание того, что они (как правило) делают.

замечания

Из документации Python:

> Следующая таблица суммирует приоритеты операторов в Python, начиная с наименьшего приоритета (наименьшая привязка) до наивысшего приоритета (самая привязка). Операторы в том же поле имеют одинаковый приоритет. Если синтаксис явно не указан, операторы являются двоичными. Операторы в одной группе блоков слева направо (за исключением сравнений, включая тесты, все из которых имеют одинаковый приоритет и цепочку слева направо и экспоненциальность, которые группируются справа налево).

оператор	Описание
лямбда	Лямбда-выражение
если еще	Условное выражение
или же	Логическое ИЛИ
а также	Логическое И
не х	Boolean NOT
in, not in, is, is, is, <, <=,>,> =, <>,! =, ==	Сравнение, включая тесты на членство и тесты на идентичность
|	Побитовое ИЛИ
^	Побитовое XOR

оператор	Описание
&	Побитовое И
<<, >>	Сдвиги
+, -	Сложение и вычитание
*, /, //,%	Умножение, деление, остаток [8]
+ x, -x, ~ x	Положительный, отрицательный, побитовый NOT
**	Экспоненциальность [9]
x [index], x [index: index], x (аргументы ...), x.attribute	Подписка, нарезка, вызов, ссылка на атрибут
(выражения ...), [выражения ...], {ключ: значение ...}, выражения ...	Отображение привязки или кортежа, отображение списка, отображение словаря, преобразование строк

Examples

Простые примеры приоритетов операторов в python.

Python следует правилу PEMDAS. PEMDAS - это скобки, экспоненты, умножения и разделения, а также добавление и вычитание.

Пример:

```
>>> 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
```

Дополнительно: математические правила сохраняются, но не всегда :

```
>>> 300 / 300 * 200
200.0
>>> 300 * 200 / 300
200.0
>>> 1e300 / 1e300 * 1e200
1e+200
>>> 1e300 * 1e200 / 1e300
inf
```

Прочитайте Приоритет оператора онлайн: https://riptutorial.com/ru/python/topic/5040/приоритет-оператора

глава 145: Проверка наличия и разрешения пути

параметры

параметр	подробности
os.F_OK	Значение для передачи в качестве параметра режима доступа () для проверки существования пути.
os.R_OK	Значение для включения в параметр режима доступа () для проверки читаемости пути.
os.W_OK	Значение для включения в параметр режима доступа () для проверки возможности записи пути.
os.X_OK	Значение для включения в параметр режима доступа (), чтобы определить, может ли путь быть выполнен.

Examples

Выполнять проверки с использованием os.access

`os.access` - намного лучшее решение для проверки наличия каталога, и оно доступно для чтения и записи.

```
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)
```

также возможно перенести все проверки вместе

```
os.access(path, os.F_OK & os.R_OK & os.W_OK & os.E_OK)
```

Все приведенное выше возвращает `True` если доступ разрешен и `False` если не разрешено. Они доступны в unix и windows.

Прочитайте Проверка наличия и разрешения пути онлайн: https://riptutorial.com/ru/python/topic/1262/проверка-наличия-и-разрешения-пути

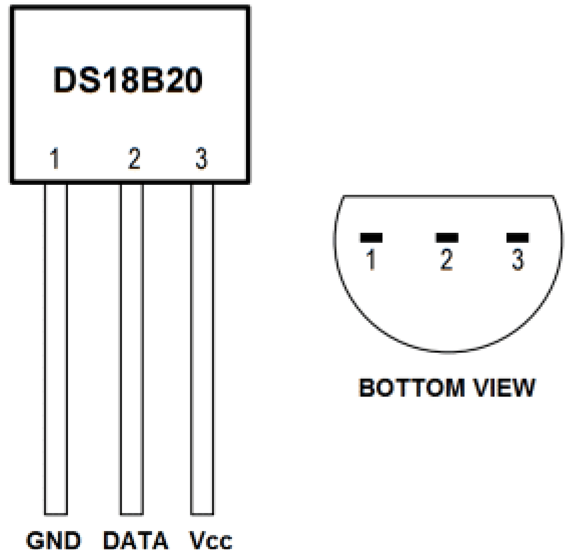
глава 146: Программирование IoT с использованием Python и малины PI

Examples

Пример - датчик температуры

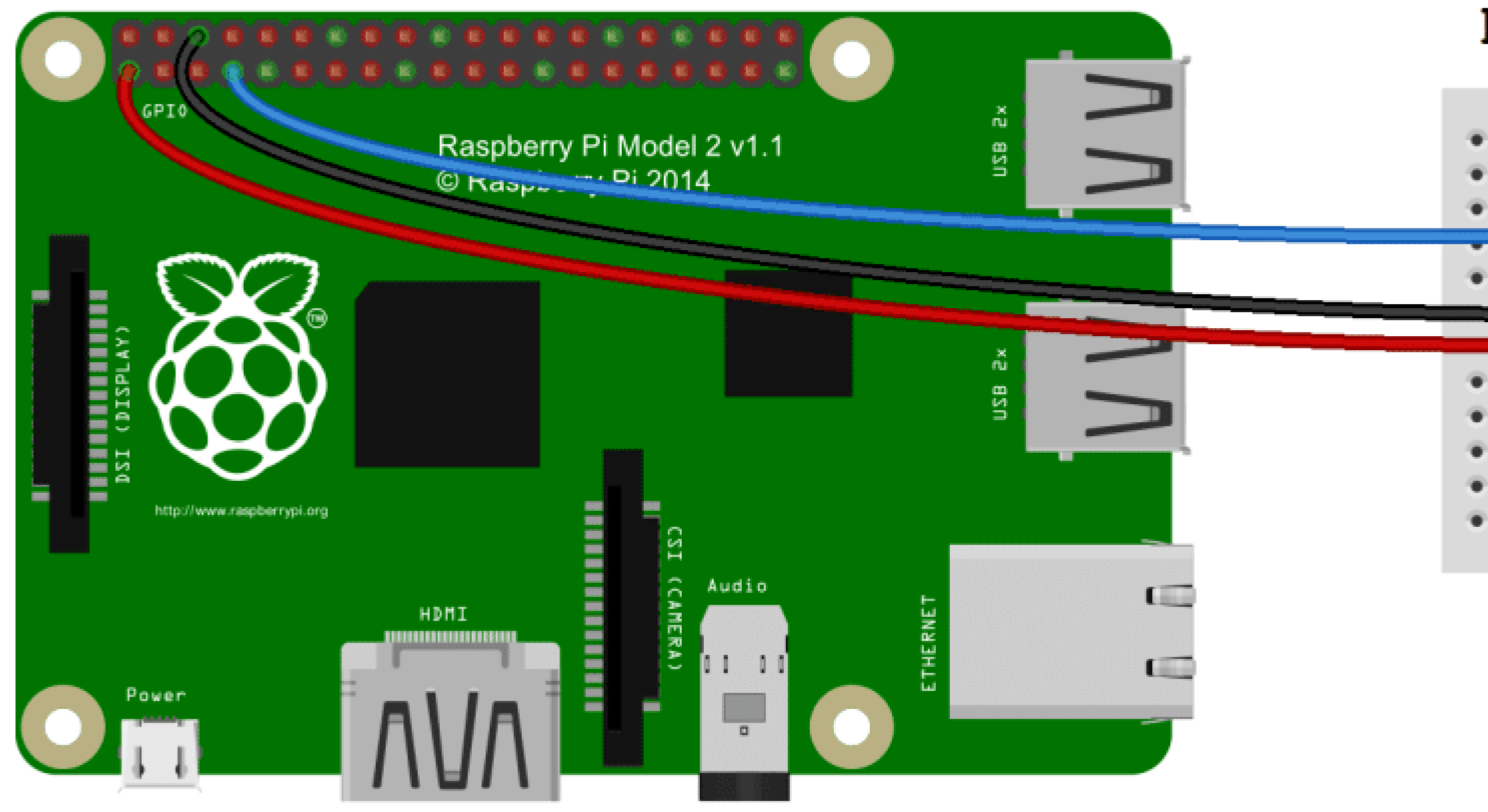
Взаимодействие DS18B20 с малиновым pi

Соединение DS18B20 с малиновой пи



Вы можете видеть, что есть три терминала

1. Vcc
2. Gnd
3. Данные (один проводной протокол)



R1 - сопротивление 4.7 кОм для вытягивания уровня напряжения

1. **Vcc** должен быть подключен к любому из контактов 5v или 3.3v из малины pi (PIN: 01, 02, 04, 17).
2. **Gnd** должен быть подключен к любому из Gnd контактов из малины pi (PIN: 06, 09, 14, 20, 25).
3. **DATA** необходимо подключить к (PIN: 07)

Включение однопроводного интерфейса со стороны RPi

4. Войдите в Raspberry pi с помощью шпатлевки или любого другого терминала linux / unix.

5. После входа в систему, откройте файл /boot/config.txt в своем любимом браузере.

   nano /boot/config.txt

6. Теперь добавьте эту строку `dtoverlay=w1-gpio` в конец файла.

7. Теперь перезагрузите перезагрузку малины pi `sudo reboot` .

8. Войдите в Raspberry pi и запустите `sudo modprobe g1-gpio`

9. Затем запустите `sudo modprobe w1-therm`

10. Теперь перейдите в каталог / sys / bus / w1 / devices `cd /sys/bus/w1/devices`

11. Теперь вы обнаружите виртуальный каталог, созданный вашим температурным датчиком, начиная с 28 - ********.

12. Перейти к этому каталогу `cd 28-********`

13. Теперь есть имя файла **w1-slave** , этот файл содержит температуру и другую информацию, такую как CRC. `cat w1-slave` .

Теперь напишите модуль в python, чтобы прочитать температуру

```
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)
```

Выше модуля python будет печатать температуру vs адрес в течение бесконечного времени. Параметр RATE определяется для изменения или настройки частоты запроса температуры от датчика.

Фиксированная диаграмма GPIO

1. [https://www.element14.com/community/servlet/JiveServlet/previewBody/73950-102-11-339300/pi3_gpio.png][3]

Прочитайте Программирование IoT с использованием Python и малины PI онлайн: https://riptutorial.com/ru/python/topic/10735/программирование-iot-с-использованием-python-и-малины-pi

глава 147: Простые математические операторы

Вступление

Python сам по себе использует общие математические операторы, включая целочисленное и плавающее деление, умножение, возведение в степень, добавление и вычитание. Математический модуль (входит во все стандартные версии Python) предлагает расширенные функции, такие как тригонометрические функции, корневые операции, логарифмы и многие другие.

замечания

Численные типы и их метаклассы

Модуль `numbers` содержит абстрактные метаклассы для числовых типов:

подклассы	numbers.Number	numbers.Integral	numbers.Rational	numbers.Real	nur
BOOL	✓	✓	✓	✓	✓
ИНТ	✓	✓	✓	✓	✓
fractions.Fraction	✓	-	✓	✓	✓
поплавок	✓	-	-	✓	✓
сложный	✓	-	-	-	✓
decimal.Decimal	✓	-	-	-	-

Examples

прибавление

```
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
```

Возможные комбинации (встроенные типы):

- `int` и `int` (дает `int`)
- `int` и `float` (дает `float`)
- `int` и `complex` (дает `complex`)
- `float` и `float` (дает `float`)
- `float` и `complex` (дает `complex`)
- `complex` и `complex` (дает `complex`)

Примечание: оператор `+` используется для конкатенации строк, списков и кортежей:

```
"first string " + "second string"    # = 'first string second string'

[1, 2, 3] + [4, 5, 6]                # = [1, 2, 3, 4, 5, 6]
```

Вычитание

```
a, b = 1, 2

# Using the "-" operator:
b - a                   # = 1


import operator         # contains 2 argument arithmetic functions
operator.sub(b, a)      # = 1
```

Возможные комбинации (встроенные типы):

- `int` и `int` (дает `int`)
- `int` и `float` (дает `float`)
- `int` и `complex` (дает `complex`)
- `float` и `float` (дает `float`)
- `float` и `complex` (дает `complex`)
- `complex` и `complex` (дает `complex`)

умножение

```
a, b = 2, 3
```

```
a * b                   # = 6

import operator
operator.mul(a, b)      # = 6
```

Возможные комбинации (встроенные типы):

- `int` и `int` (дает `int`)
- `int` и `float` (дает `float`)
- `int` и `complex` (дает `complex`)
- `float` и `float` (дает `float`)
- `float` и `complex` (дает `complex`)
- `complex` и `complex` (дает `complex`)

Примечание. Оператор `*` также используется для повторной конкатенации строк, списков и кортежей:

```
3 * 'ab'  # = 'ababab'
3 * ('a', 'b')  # = ('a', 'b', 'a', 'b', 'a', 'b')
```

разделение

Python выполняет целочисленное деление, когда оба операнда являются целыми числами. Поведение операторов разделения Python изменилось с Python 2.x и 3.x (см. Также Integer Division).

```
a, b, c, d, e = 3, 2, 2.0, -3, 10
```

Python 2.x 2.7

В Python 2 результат оператора «/» зависит от типа числителя и знаменателя.

```
a / b                   # = 1

a / c                   # = 1.5

d / b                   # = -2

b / a                   # = 0

d / e                   # = -1
```

Обратите внимание, что поскольку `a` и `b` являются `int` s, результатом является `int` .

Результат всегда округляется (перекрывается).

Поскольку `c` является float, результатом `a / c` является `float` .

Вы также можете использовать операторский модуль:

```
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

Что делать, если вы хотите иметь плавающее подразделение:

Рекомендуемые:

```
from __future__ import division # applies Python 3 style division to the entire module
a / b                  # = 1.5
a // b                 # = 1
```

Хорошо (если вы не хотите обращаться ко всему модулю):

```
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
```

Не рекомендуется (может вызвать TypeError, например, если аргумент сложный):

```
float(a) / b           # = 1.5
a / float(b)           # = 1.5
```

Python 2.x 2.2

Оператор «//» в Python 2 блокирует разделение независимо от типа.

```
a // b                # = 1
a // c                # = 1.0
```

Python 3.x 3.0

В Python 3 оператор `/` выполняет «истинное» разделение независимо от типов. Оператор `//` выполняет разделение по полу и поддерживает тип.

```
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
```

Возможные комбинации (встроенные типы):

- `int` и `int` (дает `int` в Python 2 и `float` в Python 3)
- `int` и `float` (дает `float`)
- `int` и `complex` (дает `complex`)
- `float` и `float` (дает `float`)
- `float` и `complex` (дает `complex`)
- `complex` и `complex` (дает `complex`)

См. PEP 238 для получения дополнительной информации.

возведения

```
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
```

Другое различие между встроенным `pow` и `math.pow` заключается в том, что встроенная `math.pow` `pow` может принимать три аргумента:

```
a, b, c = 2, 3, 2

pow(2, 3, 2)           # 0, calculates (2 ** 3) % 2, but as per Python docs,
                       #    does so more efficiently
```

Специальные функции

Функция `math.sqrt(x)` вычисляет квадратный корень из `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)
```

Чтобы вычислить другие корни, такие как корень куба, поднимите число на обратную степень корня. Это можно сделать с помощью любой из экспоненциальных функций или оператора.

```
import math
x = 8
math.pow(x, 1/3) # evaluates to 2.0
```

```
x**(1/3) # evaluates to 2.0
```

Функция `math.exp(x)` вычисляет `e ** x` .

```
math.exp(0)  # 1.0
math.exp(1)  # 2.718281828459045 (e)
```

Функция `math.expm1(x)` вычисляет `e ** x - 1` . Когда `x` мало, это дает значительно лучшую точность, чем `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      # 1.000000500000166666708333341666...
```

Логарифмы

По умолчанию функция `math.log` вычисляет логарифм числа, основание e. Вы можете опционально указать базу в качестве второго аргумента.

```
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)
```

Специальные вариации функции `math.log` существуют для разных оснований.

```
# 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)
```

Операции на месте

Для приложений обычно необходимо иметь такой код:

```
a = a + 1
```

или же

```
a = a * 2
```

Существует эффективный ярлык для этих операций:

```
a += 1
# and
a *= 2
```

Любой математический оператор может использоваться до символа '=' для выполнения операции inplace:

- `-=` уменьшить значение переменной на месте
- `+=` приращение переменной на месте
- `*=` умножить переменную на место
- `/=` разделить переменную на месте
- `//=` пол разделяет переменную на месте # Python 3
- `%=` возвращает модуль переменной на месте
- `**=` повышение мощности

Остальные на месте операторы существуют для побитовых операторов (`^` , `|` т.д.)

Тригонометрические функции

```
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
```

> Заметим, что `math.hypot(x, y)` также является длиной вектора (или евклидова расстояния) от начала координат `(0, 0)` до точки `(x, y)` .
>
> Чтобы вычислить евклидово расстояние между двумя точками `(x1, y1)` & `(x2, y2)` вы можете использовать `math.hypot` следующим образом
>
> ```
> math.hypot(x2-x1, y2-y1)
> ```

Для преобразования из радианов -> градусов и градусов -> радиан соответственно используют `math.degrees` и `math.radians`

```
math.degrees(a)
# Out: 57.29577951308232

math.radians(57.29577951308232)
# Out: 1.0
```

модуль

Как и во многих других языках, Python использует оператор `%` для вычисления модуля.

```
3 % 4     # 3
10 % 2    # 0
6 % 4     # 2
```

Или с помощью `operator` модуля:

```
import operator

operator.mod(3 , 4)     # 3
operator.mod(10 , 2)    # 0
operator.mod(6 , 4)     # 2
```

Вы также можете использовать отрицательные числа.

```
-9 % 7     # 5
9 % -7     # -5
-9 % -7    # -2
```

Если вам нужно найти результат целочисленного деления и модуля, вы можете использовать функцию `divmod` как ярлык:

```
quotient, remainder = divmod(9, 4)
# quotient = 2, remainder = 1 as 4 * 2 + 1 == 9
```

Прочитайте Простые математические операторы онлайн:
https://riptutorial.com/ru/python/topic/298/простые-математические-операторы

глава 148: профилирование

Examples

%% timeit и% timeit в IPython

Профилирование строки конкатенации:

```
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
```

Профилирование циклов по итерациям и спискам:

```
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
```

функция timeit ()

Профилирование повторения элементов в массиве

```
>>> import timeit
>>> timeit.timeit('list(itertools.repeat("a", 100))', 'import itertools', number = 10000000)
10.997665435877963
>>> timeit.timeit('["a"]*100', number = 10000000)
7.118789926862576
```

команда timeit

Профилирование конкатенации чисел

```
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 в командной строке

Исходный код с директивой @profile перед функцией, которую мы хотим профилировать:

```
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()
```

Использование команды kernprof для расчета профилирования по строкам

```
$ 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)])
```

Запрос страницы почти всегда медленнее, чем любой расчет, основанный на информации на странице.

Использование cProfile (Preferred Profiler)

Python включает профилировщик cProfile. Обычно это предпочтительнее использования timeit.

Он разбивает весь ваш скрипт, и для каждого метода в вашем скрипте он сообщает вам:

- `ncalls` : количество раз, когда был вызван метод
- `tottime` : общее время, затрачиваемое на заданную функцию (исключая время, сделанное при вызовах `tottime`)
- `percall` : Время, затраченное на звонок. Или отношение tottime, деленное на ncalls
- `cumtime` : совокупное время, потраченное на эту и все подфункции (от вызова до выхода). Этот показатель является точным даже для рекурсивных функций.

- `percall` : является частным временем cumtime, деленным на примитивные вызовы
- `filename:lineno(function)` : предоставляет соответствующие данные каждой функции

CProfiler можно легко вызвать в командной строке, используя:

```
$ python -m cProfile main.py
```

Чтобы отсортировать возвращенный список профилированных методов к времени, указанному в методе:

```
$ python -m cProfile -s time main.py
```

Прочитайте профилирование онлайн: https://riptutorial.com/ru/python/topic/3818/профилирование

глава 149: Процессы и потоки

Вступление

Большинство программ выполняются по строкам, за один раз выполняется только один процесс. Нитки позволяют нескольким процессам течь независимо друг от друга. Threading с несколькими процессорами позволяет программам запускать несколько процессов одновременно. В этом разделе описывается реализация и использование потоков в Python.

Examples

Глобальная блокировка переводчика

Частота многопоточности Python часто страдает из-за **блокировки Global Interpreter** . Короче говоря, хотя вы можете иметь несколько потоков в программе Python, только одна команда байт-кода может выполняться параллельно в любой момент времени, независимо от количества процессоров.

Таким образом, многопоточность в случаях, когда операции блокируются внешними событиями, такими как доступ к сети, может быть весьма эффективной:

```
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
```

Обратите внимание: несмотря на то, что для каждого `process` потребовалось 2 секунды для выполнения, четыре процесса вместе могли эффективно работать параллельно, занимая 2 секунды.

Однако многопоточность в случаях, когда интенсивные вычисления выполняются в коде Python - например, при большом количестве вычислений - не приводит к значительному улучшению и даже может быть медленнее, чем параллельная работа:

```
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
```

В последнем случае многопроцессорность может быть эффективной, поскольку несколько процессов могут, разумеется, выполнять несколько команд одновременно:

```
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
```

Работа в нескольких потоках

Используйте `threading.Thread` для запуска функции в другом потоке.

```
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
```

Работа в нескольких процессах

Используйте `multiprocessing.Process` для запуска функции в другом процессе. Интерфейс похож на `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
```

Совместное использование между потоками

Поскольку все потоки работают в одном и том же процессе, все потоки имеют доступ к тем же данным.

Однако одновременный доступ к общим данным должен быть защищен блокировкой во избежание проблем синхронизации.

```
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}
```

Совместное использование состояний между процессами

Код, выполняющийся в разных процессах, по умолчанию не использует одни и те же данные. Тем не менее, модуль `multiprocessing` содержит примитивы, которые помогают совместно использовать значения для нескольких процессов.

```
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
```

Прочитайте Процессы и потоки онлайн: https://riptutorial.com/ru/python/topic/4110/процессы-и-потоки

глава 150: Работа над глобальным блокировщиком перевода (GIL)

замечания

Почему существует GIL?

GIL работает в CPython с момента создания потоков Python в 1992 году. Он разработан для обеспечения безопасности потоков при работе с кодом Python. Устные переводчики Python, написанные с помощью GIL, предотвращают одновременное выполнение нескольких собственных потоков от байт-кодов Python. Это облегчает для плагинов обеспечение того, чтобы их код был потокобезопасным: просто заблокируйте GIL, и только ваш активный поток может работать, поэтому ваш код автоматически потокобезопасен.

Краткая версия: GIL гарантирует, что независимо от того, сколько процессоров и потоков у вас есть, *только один поток интерпретатора python будет запускаться за один раз.*

Это имеет много преимуществ в удобстве использования, но также имеет множество негативных преимуществ.

Обратите внимание, что GIL не является требованием языка Python. Следовательно, вы не можете получить доступ к GIL напрямую из стандартного кода python. Не все реализации Python используют GIL.

Интерпретаторы, которые имеют GIL: CPython, PyPy, Cython (но вы можете отключить GIL с `nogil`)

Переводчики, не имеющие GIL: Jython, IronPython

Подробная информация о том, как работает GIL:

Когда поток работает, он блокирует GIL. Когда поток хочет запустить, он запрашивает GIL и ждет, пока он не будет доступен. В CPython, перед версией 3.2, текущий поток проверял бы после некоторого количества инструкций python, чтобы узнать, нужен ли другой код блокировке (то есть, он освободил блокировку, а затем запросил ее снова). Этот метод, как правило, вызывал головокружение потоков, главным образом потому, что поток, который освободил блокировку, приобрел его снова, прежде чем ожидающие потоки имели возможность проснуться. Начиная с 3.2, потоки, которые хотят, чтобы GIL ожидал

блокировки в течение некоторого времени, и после этого времени, они устанавливают общую переменную, которая заставляет текущую нить давать. Однако это может привести к значительному увеличению времени выполнения. См. Ссылки ниже от dabeaz.com (в разделе ссылок) для более подробной информации.

CPython автоматически освобождает GIL, когда поток выполняет операцию ввода-вывода. Библиотеки обработки изображений и операции хруста с номером номера освобождают GIL перед выполнением их обработки.

Преимущества GIL

Для переводчиков, которые используют GIL, GIL является системным. Он используется для сохранения состояния приложения. Преимущества включают:

- Сбор мусора - подсчет ссылок на потоки должен быть изменен, когда GIL заблокирован. *В CPython вся коллекция garbarge привязана к GIL.* Это большой; см. статью wiki python.org о GIL (см. ниже в ссылках) для получения подробной информации о том, что должно быть функционально, если вы хотите удалить GIL.
- Простота для программистов, занимающихся GIL - блокировка, все упрощена, но легко кодируется
- Упрощает импорт модулей с других языков

Последствия GIL

GIL только позволяет одному потоку запускать код python за раз в интерпретаторе python. Это означает, что многопоточность процессов, выполняющих строгий код python, просто не работает. При использовании потоков против GIL у вас, вероятно, будет хуже производительность с потоками, чем при запуске в одном потоке.

Рекомендации:

https://wiki.python.org/moin/GlobalInterpreterLock - краткое изложение того, что он делает, мелкие детали всех преимуществ

http://programmers.stackexchange.com/questions/186889/why-was-python-written-with-the-gil - четко написанное резюме

http://www.dabeaz.com/python/UnderstandingGIL.pdf - как работает GIL и почему он замедляется на нескольких ядрах

http://www.dabeaz.com/GIL/gilvis/index.html - визуализация данных, показывающих, как GIL блокирует потоки

http://jeffknupp.com/blog/2012/03/31/pythons-hardest-problem/ - простая для понимания история проблемы GIL

https://jeffknupp.com/blog/2013/06/30/pythons-hardest-problem-revisited/ - подробная информация о способах работы с ограничениями GIL

Examples

Multiprocessing.Pool

Простой ответ, когда вы спрашиваете, как использовать потоки в Python, это: «Не используйте. Вместо этого используйте процессы». Многопроцессорный модуль позволяет создавать процессы с похожим синтаксисом для создания потоков, но я предпочитаю использовать их удобный объект Pool.

Используя код, который Дэвид Бэйзли впервые использовал для выявления опасностей потоков против GIL , мы перепишем его с помощью многопроцессорной обработки. Пул :

Код Дэвида Бэзли, который показал проблемы с резьбой 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
```

Переписано с использованием многопроцессорности.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)
```

Вместо создания потоков это создает новые процессы. Поскольку каждый процесс является его собственным интерпретатором, не происходит столкновений GIL. multiprocessing.Pool откроет столько процессов, сколько на нем будет ядер, хотя в приведенном выше примере ему потребуется только два. В реальном сценарии вы хотите создать свой список, чтобы иметь как минимум столько же, сколько на вашем компьютере. Пул будет запускать функцию, которую вы укажете ей для запуска с каждым аргументом, вплоть до количества процессов, которые он создает. Когда функция завершится, все остальные функции в списке будут запущены в этом процессе.

Я обнаружил, что даже используя оператор `with` , если вы не закрываете и не присоединяетесь к пулу, процессы продолжают существовать. Чтобы очистить ресурсы, я всегда закрываю и присоединяюсь к моим пулам.

Cython nogil:

Cython - альтернативный интерпретатор python. Он использует GIL, но позволяет отключить его. См. Их документацию

В качестве примера, используя код, который Дэвид Бэйсли впервые использовал, чтобы показать опасности потоков против GIL , мы перепишем его с помощью nogil:

Код Дэвида Бэзли, который показал проблемы с резьбой 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
```

Переписано с использованием nogil (ТОЛЬКО РАБОТАЕТ В ЦИТОНЕ):

```
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
```

Это так просто, если вы используете cython. Обратите внимание, что в документации указано, что вы не должны изменять какие-либо объекты python:

> Код в теле оператора не должен каким-либо образом манипулировать объектами Python и не должен называть ничего, что манипулирует объектами Python, без предварительного повторного приобретения GIL. Cython в настоящее время не проверяет это.

Прочитайте Работа над глобальным блокировщиком перевода (GIL) онлайн: https://riptutorial.com/ru/python/topic/4061/работа-над-глобальным-блокировщиком-перевода--gil-

глава 151: Работа с ZIP-архивами

Синтаксис

- импорт zipfile
- *класс* zipfile. **ZipFile** (*файл, режим = 'r', сжатие = ZIP_STORED, allowZip64 = True*)

замечания

Если вы попытаетесь открыть файл, который не является ZIP-файлом, `zipfile.BadZipFile` исключение `zipfile.BadZipFile` .

В Python 2.7 это было написано `zipfile.BadZipfile` , и это старое имя сохраняется вместе с новым в Python 3.2+

Examples

Открытие почтовых файлов

Чтобы начать, импортируйте `zipfile` модуль и задайте имя файла.

```
import zipfile
filename = 'zipfile.zip'
```

Работа с zip-архивами очень похожа на работу с файлами , вы создаете объект, открывая zip-файл, который позволяет вам работать над ним, прежде чем снова закрыть файл.

```
zip = zipfile.ZipFile(filename)
print(zip)
# <zipfile.ZipFile object at 0x0000000002E51A90>
zip.close()
```

В Python 2.7 и Python 3 версии выше 3.2, можно использовать `with` менеджером контекста. Мы открываем файл в режиме «читать», а затем печатаем список имен файлов:

```
with zipfile.ZipFile(filename, 'r') as z:
    print(zip)
    # <zipfile.ZipFile object at 0x0000000002E51A90>
```

Изучение содержимого Zipfile

Есть несколько способов проверить содержимое zipfile. Вы можете использовать `printdir` чтобы просто получить различную информацию, отправленную на `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.
```

Мы также можем получить список имен файлов с помощью метода `namelist` . Здесь мы просто печатаем список:

```
with zipfile.ZipFile(filename) as zip:
    print(zip.namelist())

# Out: ['pyexpat.pyd', 'python.exe', 'python3.dll', 'python35.dll', ... etc. ...]
```

Вместо `namelist` мы можем вызвать метод `infolist` , который возвращает список объектов `ZipInfo` , которые содержат дополнительную информацию о каждом файле, например `ZipInfo` метку и размер файла:

```
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
```

Извлечение содержимого zip-файла в каталог

Извлечь все содержимое файла zip-файла

```
import zipfile
with zipfile.ZipFile('zipfile.zip','r') as zfile:
    zfile.extractall('path')
```

Если вы хотите, чтобы отдельные файлы использовали метод extract, он принимает список имен и путь в качестве входного параметра

```
import zipfile
f=open('zipfile.zip','rb')
zfile=zipfile.ZipFile(f)
for cont in zfile.namelist():
    zfile.extract(cont,path)
```

Создание новых архивов

Чтобы создать новый архив, откройте zipfile с режимом записи.

```
import zipfile
new_arch=zipfile.ZipFile("filename.zip",mode="w")
```

Чтобы добавить файлы в этот архив, используйте метод 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()
```

Если вы хотите записать строку байтов в архив, вы можете использовать метод writestr ().

```
str_bytes="string buffer"
new_arch.writestr('filename_string_in_archive.txt',str_bytes)
new_arch.close()
```

Прочитайте Работа с ZIP-архивами онлайн: https://riptutorial.com/ru/python/topic/3728/работа-с-zip-архивами

глава 152: Разбор аргументов командной строки

Вступление

Большинство инструментов командной строки полагаются на аргументы, переданные программе после ее выполнения. Вместо запроса ввода данных эти программы ожидают, что данные или определенные флаги (которые становятся логическими) будут установлены. Это позволяет как пользовательским, так и другим программам запускать файл Python, передавая его по мере его запуска. В этом разделе объясняется и демонстрируется реализация и использование аргументов командной строки в Python.

Examples

Привет, мир в argparse

Следующая программа приветствует пользователя. Он принимает один позиционный аргумент, имя пользователя и также может быть передан приветствие.

```
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!
```

Для получения дополнительной информации, пожалуйста, прочитайте документацию argparse .

Основной пример с docopt

docopt преобразует аргумент аргумента командной строки на голове. Вместо анализа аргументов вы просто **пишете строку использования** для своей программы, а docopt **анализирует строку использования** и использует ее для извлечения аргументов командной строки.

```
"""
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)
```

Примеры прогонов:

```
$ 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'}
```

Установка взаимно исключающих аргументов с помощью argparse

Если вы хотите, чтобы два или более аргумента были взаимоисключающими. Вы можете

использовать функцию `argparse.ArgumentParser.add_mutually_exclusive_group()` . В приведенном ниже примере может существовать либо foo, либо bar, но не оба одновременно.

```
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
```

Если вы попытаетесь запустить скрипт, указав оба аргумента `--foo` и `--bar` , скрипт будет жаловаться на следующее сообщение.

```
error: argument -b/--bar: not allowed with argument -f/--foo
```

Использование аргументов командной строки с argv

Всякий раз, когда скрипт Python вызывается из командной строки, пользователь может предоставить дополнительные **аргументы командной строки,** которые будут переданы скрипту. Эти аргументы будут доступны для программиста от переменной системы `sys.argv` («ARGV» является традиционным названием используется в большинстве языков программирования, и это означает «**ARG** ument **v** Эктор»).

По соглашению, первым элементом в списке `sys.argv` является имя самого скрипта Python, а остальные элементы - токены, переданные пользователем при вызове скрипта.

```
# 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']
```

Вот еще один пример использования `argv` . Сначала мы отключаем исходный элемент sys.argv, потому что он содержит имя скрипта. Затем мы объединим остальные аргументы в одном предложении и, наконец, напечатаем это предложение, добавив имя текущего пользователя в систему (чтобы он эмулировал программу чата).

```
import getpass
import sys
```

```
words = sys.argv[1:]
sentence = " ".join(words)
print("[%s] %s" % (getpass.getuser(), sentence))
```

Алгоритм, обычно используемый при «ручном» анализе нескольких непозиционных аргументов, заключается в `sys.argv` списку `sys.argv` . Один из способов - перечислить список и поместить каждый его элемент:

```
# 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()
```

Сообщение об ошибке парсерного анализатора с помощью argparse

Вы можете создавать сообщения об ошибках парсера в соответствии с потребностями вашего скрипта. Это через функцию `argparse.ArgumentParser.error` . В приведенном ниже примере показано, как сценарий печатает использование и сообщение об ошибке для `stderr` когда `--foo` но не `--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
```

Предположим, что имя вашего скрипта - sample.py, и мы запускаем: `python sample.py --foo ds_in_fridge`

Сценарий будет жаловаться на следующее:

```
usage: sample.py [-h] [-f FOO] [-b BAR]
sample.py: error: --foo requires --bar. You did not specify bar.
```

Концептуальная группировка аргументов с помощью

argparse.add_argument_group ()

Когда вы создаете argparse ArgumentParser () и запускаете свою программу с помощью «-h», вы получаете автоматическое сообщение об использовании, объясняющее, с какими аргументами вы можете управлять своим программным обеспечением. По умолчанию позиционные аргументы и условные аргументы разделяются на две категории, например, вот небольшой скрипт (example.py) и вывод при запуске `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
```

Есть несколько ситуаций, когда вы хотите разделить свои аргументы на дополнительные концептуальные разделы, чтобы помочь вашему пользователю. Например, вы можете иметь все параметры ввода в одной группе и все параметры форматирования вывода в другом. Вышеприведенный пример можно настроить для разделения аргументов `--foo_*` от аргументов `--bar_*` .

```
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()
```

Который производит этот вывод, когда запускается `python example.py -h` :

```
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
```

Расширенный пример с docopt и docopt_dispatch

Как docopt, с [docopt_dispatch] вы ремесло ваш `--help` в `__doc__` переменной вашей точки входа модуля. Там вы вызываете `dispatch` с doc-строкой в качестве аргумента, чтобы она могла запускать парсер.

Это делается вместо того, чтобы вручную обрабатывать аргументы (которые обычно заканчиваются в высокой циклической структуре if / else), вы оставляете его для отправки, предоставляя только то, как вы хотите обрабатывать множество аргументов.

Это то, на что работает `dispatch.on` : вы даете ему аргумент или последовательность аргументов, которые должны запускать функцию, и эта функция будет выполняться с соответствующими значениями в качестве параметров.

```
"""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__)
```

Прочитайте Разбор аргументов командной строки онлайн:
https://riptutorial.com/ru/python/topic/1382/разбор-аргументов-командной-строки

глава 153: развертывание

Examples

Загрузка пакета Conda

Перед началом работы вы должны:

Anaconda, установленная на вашей учетной записи системы на Binstar Если вы не используете Anaconda 1.6+, установите клиент командной строки binstar :

```
$ conda install binstar
$ conda update binstar
```

Если вы не используете Anaconda, Binstar также доступен на pypi:

```
$ pip install binstar
```

Теперь мы можем войти в систему:

```
$ binstar login
```

Проверьте свой логин с помощью команды whoami:

```
$ binstar whoami
```

Мы собираемся загрузить пакет с простой функцией «hello world». Чтобы следовать за ним, получив мой демонстрационный пакет репо от Github:

```
$ git clone https://github.com/<NAME>/<Package>
```

Это небольшой каталог, который выглядит так:

```
package/
        setup.py
        test_package/
           __init__.py
           hello.py
           bld.bat
           build.sh
           meta.yaml
```

`Setup.py` является стандартным файлом сборки python, а `hello.py` имеет нашу единственную функцию hello_world ().

`bld.bat` , `build.sh` и `meta.yaml` скрипты и метаданные для `Conda` пакета. Вы можете прочитать

страницу сборки Conda для получения дополнительной информации об этих трех файлах и их целях.

Теперь мы создаем пакет, запустив:

```
$ conda build test_package/
```

Это все, что требуется для создания пакета Conda.

Последним шагом является загрузка в binstar путем копирования и вставки последней строки печати после запуска команды test_package / command. В моей системе команда:

```
$ binstar upload /home/xavier/anaconda/conda-bld/linux-64/test_package-0.1.0-py27_0.tar.bz2
```

Поскольку вы впервые создаете пакет и выпускаете, вам будет предложено заполнить некоторые текстовые поля, которые можно альтернативно сделать через веб-приложение.

Вы увидите `done` распечатан , чтобы подтвердить , что вы успешно загрузили пакет Конда в Binstar.

Прочитайте развертывание онлайн: https://riptutorial.com/ru/python/topic/4064/развертывание

глава 154: Разница между модулем и пакетом

замечания

Можно поместить пакет Python в ZIP-файл и использовать его таким образом, если вы добавите эти строки в начало вашего скрипта:

```
import sys
sys.path.append("package.zip")
```

Examples

Модули

Модуль представляет собой один файл Python, который можно импортировать. Использование модуля выглядит так:

`module.py`

```
def hi():
    print("Hello world!")
```

`my_script.py`

```
import module
module.hi()
```

в переводчике

```
>>> from module import hi
>>> hi()
# Hello world!
```

пакеты

Пакет состоит из нескольких файлов (или модулей) Python и может содержать даже библиотеки, написанные на C или C ++. Вместо того, чтобы быть единственным файлом, это целая структура папок, которая может выглядеть так:

Папка `package`

- `__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!")
```

Все пакеты Python должны содержать файл `__init__.py` . Когда вы импортируете пакет в свой скрипт (`import package`), будет запущен скрипт `__init__.py` , предоставляющий вам доступ ко всем функциям пакета. В этом случае он позволяет использовать функции `package.hi` и `package.woof` .

Прочитайте Разница между модулем и пакетом онлайн: https://riptutorial.com/ru/python/topic/3142/разница-между-модулем-и-пакетом

глава 155: Распаковка файлов

Вступление

Для извлечения или распаковки файла tarball, ZIP или gzip предоставляются соответственно файлы tarfile, zipfile и gzip Python. Модуль tarfile Python предоставляет `TarFile.extractall(path=".", members=None)` для извлечения из файла tarball. Zipfile-модуль Python предоставляет `ZipFile.extractall([path[, members[, pwd]]])` для извлечения или распаковки ZIP-архивов. Наконец, модуль gzip Python предоставляет класс GzipFile для распаковки.

Examples

Использование Python ZipFile.extractall () для распаковки ZIP-файла

```
file_unzip = 'filename.zip'
unzip = zipfile.ZipFile(file_unzip, 'r')
unzip.extractall()
unzip.close()
```

Использование Python TarFile.extractall () для распаковки tarball

```
file_untar = 'filename.tar.gz'
untar = tarfile.TarFile(file_untar)
untar.extractall()
untar.close()
```

Прочитайте Распаковка файлов онлайн: https://riptutorial.com/ru/python/topic/9505/распаковка-файлов

глава 156: распределение

Examples

py2app

Чтобы использовать фреймворк py2app, вы должны установить его в первую очередь. Сделайте это, открыв терминал и введя следующую команду:

```
sudo easy_install -U py2app
```

Вы также можете `pip` установить пакеты , как:

```
pip install py2app
```

Затем создайте установочный файл для своего скрипта python:

```
py2applet --make-setup MyApplication.py
```

Измените настройки файла настроек по своему усмотрению, это значение по умолчанию:

```
"""
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'],
)
```

Чтобы добавить файл значка (этот файл должен иметь расширение .icns) или включить изображения в приложение в качестве ссылки, измените свои параметры, как показано:

```
DATA_FILES = ['myInsertedImage.jpg']
OPTIONS = {'argv_emulation': True, 'iconfile': 'myCoolIcon.icns'}
```

Наконец, введите это в терминал:

```
python setup.py py2app
```

Скрипт должен запускаться, и вы найдете свое законченное приложение в папке dist.

Используйте следующие параметры для дополнительной настройки:

```
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

Установите cx_Freeze отсюда

Разархивируйте папку и запустите эти команды из этого каталога:

```
python setup.py build
sudo python setup.py install
```

Создайте новый каталог для вашего скрипта python и создайте файл **«setup.py»** в том же каталоге со следующим содержимым:

```
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)])
```

Теперь запустите setup.py с терминала:

```
python setup.py bdist_mac
```

ПРИМЕЧАНИЕ. На El Capitan это должно быть запущено как root с отключенным режимом SIP.

Прочитайте распределение онлайн: https://riptutorial.com/ru/python/topic/2026/распределение

глава 157: Регулярные выражения (регулярное выражение)

Вступление

Python предоставляет регулярные выражения через модуль `re` .

Регулярные выражения - это комбинации символов, которые интерпретируются как правила для подстановки подстрок. Например, выражение `'amount\D+\d+'` будет соответствовать любой строке, состоящей из `amount` слова плюс целое число, разделенное одной или несколькими цифрами, например: `amount=100` , `amount is 3` , `amount is equal to: 33` и т. Д.

Синтаксис

- **Прямые регулярные выражения**
- re.match (pattern, string, flag = 0) # Out: шаблон соответствия в начале строки или None
- re.search (pattern, string, flag = 0) # Out: шаблон соответствия внутри строки или None
- re.findall (pattern, string, flag = 0) # Out: список всех совпадений шаблона в строке или []
- re.finditer (pattern, string, flag = 0) # Out: то же, что и re.findall, но возвращает объект итератора
- re.sub (pattern, replacement, string, flag = 0) # Out: строка с заменой (строка или функция) вместо шаблона
- **Предварительно скомпилированные регулярные выражения**
- precompiled_pattern = re.compile (pattern, flag = 0)
- precompiled_pattern.match (строка) # Out: совпадение в начале строки или None
- precompiled_pattern.search (строка) # Out: соответствие в любом месте строки или None
- precompiled_pattern.findall (строка) # Out: список всех соответствующих подстрок
- precompiled_pattern.sub (строка / шаблон / функция, строка) # Out: заменена строка

Examples

Соответствие началу строки

Первый аргумент `re.match()` - это регулярное выражение, второе - строка, которая соответствует:

```
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'
```

Вы можете заметить, что переменная шаблона представляет собой строку с префиксом `r` , которая указывает, что строка является *строковым литералом* .

Строковый литерал имеет немного иной синтаксис, чем строковый литерал, а именно обратная косая черта `\` в строковом литерале означает «просто обратную косую черту», и нет необходимости в удвоении задержек, чтобы избежать «escape-последовательностей», таких как новые строки (`\n`) , tabs (`\t`), backspaces (`\`), form-feeds (`\r`) и т. д. В обычных строковых литералах каждая обратная косая черта должна быть удвоена, чтобы не быть принятой за начало escape-последовательности.

Следовательно, `r"\n"` представляет собой строку из двух символов: `\` и `n` . В шаблонах регулярных выражений также используются обратные косые черты, например, `\d` относится к любому символу цифры. Мы можем избежать двойного выхода из наших строк (`"\\d"`), используя необработанные строки (`r"\d"`).

Например:

```
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'
```

Согласование выполняется только с начала строки. Если вы хотите `re.search` любом месте, используйте `re.search` :

```
match = re.match(r"(123)", "a123zzb")

match is None
```

```
# Out: True

match = re.search(r"(123)", "a123zzb")

match.group()
# Out: '123'
```

поиск

```
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.'
```

Поиск выполняется в любом месте строки в отличие от `re.match` . Вы также можете использовать `re.findall` .

Вы также можете выполнить поиск в начале строки (используйте `^`),

```
match = re.search(r"^123", "123zzb")
match.group(0)
# Out: '123'

match = re.search(r"^123", "a123zzb")
match is None
# Out: True
```

в конце строки (используйте `$`),

```
match = re.search(r"123$", "zzb123")
match.group(0)
# Out: '123'

match = re.search(r"123$", "123zzb")
match is None
# Out: True
```

или оба (используйте как `^` и `$`):

```
match = re.search(r"^123$", "123")
match.group(0)
# Out: '123'
```

группирование

Группировка выполняется с помощью круглых скобок. Calling `group()` возвращает строку,

сформированную из соответствующих подгрупп в скобках.

```
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'
```

Аргументы также могут быть предоставлены `group()` для извлечения определенной подгруппы.

Из документов :

> Если есть один аргумент, результатом будет одиночная строка; если есть несколько аргументов, результатом является кортеж с одним элементом для каждого аргумента.

С другой стороны, вызывающие `groups()` возвращают список кортежей, содержащих подгруппы.

```
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')
```

Именованные группы

```
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'
```

Создает группу захвата, на которую можно ссылаться как по имени, так и по индексу.

Нехватывающие группы

Использование `(?:)` создает группу, но группа не захватывается. Это означает, что вы можете использовать его как группу, но это не будет загрязнять ваше «групповое пространство».

```
re.match(r'(\d+)(\+(\d+))?', '11+22').groups()
# Out: ('11', '+22', '22')

re.match(r'(\d+)(?:\+(\d+))?', '11+22').groups()
# Out: ('11', '22')
```

Этот пример соответствует `11+22` или `11` , но не `11+` . Это связано с тем, что знак `+` и второй член сгруппированы. С другой стороны, знак `+` не фиксируется.

Экранирование специальных символов

Специальные символы (например, скобки символьного класса `[` и `]` ниже) не соответствуют буквально:

```
match = re.search(r'[b]', 'a[b]c')
match.group()
# Out: 'b'
```

Удерживая специальные символы, их можно сопоставить буквально:

```
match = re.search(r'\[b\]', 'a[b]c')
match.group()
# Out: '[b]'
```

Для `re.escape()` можно использовать функцию `re.escape()` :

```
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'
```

Функция `re.escape()` все специальные символы, поэтому полезно, если вы составляете регулярное выражение, основанное на пользовательском вводе:

```
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.!']
```

Замена

Замены могут быть выполнены в строках с использованием `re.sub` .

Замена строк

```
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'
```

Использование ссылок на группы

Замены с небольшим количеством групп могут быть сделаны следующим образом:

```
re.sub(r"t([0-9])([0-9])", r"t\2\1", "t13 t19 t81 t25")
# Out: 't31 t91 t18 t52'
```

Однако, если вы делаете идентификатор группы, такой как «10», это не работает : `\10` читается как «Идентификационный номер 1, за которым следует 0». Поэтому вы должны быть более конкретными и использовать обозначение `\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'
```

Использование функции замены

```
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'
```

Найти все неперекрывающиеся соответствия

```
re.findall(r"[0-9]{2,3}", "some 1 text 12 is 945 here 4445588899")
# Out: ['12', '945', '444', '558', '889']
```

Обратите внимание, что `r` до `"[0-9]{2,3}"` говорит python интерпретировать строку as-is; как «сырую» строку.

Вы также можете использовать `re.finditer()` который работает так же, как и `re.findall()` но возвращает итератор с объектами `SRE_Match` вместо списка строк:

```
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
'''
```

Предварительно скомпилированные шаблоны

```
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
```

Компиляция шаблона позволяет повторно использовать его в программе. Однако обратите внимание, что Python кэширует недавно используемые выражения (docs , SO ответ), поэтому *«программы, которые используют только несколько регулярных выражений одновременно, не должны беспокоиться о компиляции регулярных выражений»* .

```
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
```

Его можно использовать с re.match ().

Проверка допустимых символов

Если вы хотите проверить, что строка содержит только определенный набор символов, в этом случае az, AZ и 0-9, вы можете сделать это так,

```
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'
```

Вы также можете адаптировать строку выражения от `[^a-zA-Z0-9.]` `[^a-z0-9.]` , Чтобы, например, запретить прописные буквы.

Частичный кредит: http://stackoverflow.com/a/1325265/2697955

Разбиение строки с использованием регулярных выражений

Вы также можете использовать регулярные выражения для разделения строки. Например,

```
import re
data = re.split(r'\s+', 'James 94 Samantha 417 Scarlett 74')
print( data )
# Output: ['James', '94', 'Samantha', '417', 'Scarlett', '74']
```

Флаги

Для некоторых особых случаев нам нужно изменить поведение регулярного выражения, это делается с использованием флагов. Флаги могут быть установлены двумя способами, с помощью ключевого слова `flags` или непосредственно в выражении.

Ключевое слово Flags

Ниже пример для `re.search` но он работает для большинства функций модуля `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'
```

Общие флаги

Флаг	Краткое описание
`re.IGNORECASE` , `re.I`	Заставляет шаблон игнорировать случай
`re.DOTALL` , `re.S`	Делает `.` соответствовать всем, включая символы новой строки
`re.MULTILINE` , `re.M`	Делает `^` совпадающим начало строки и `$` конец строки
`re.DEBUG`	Включает отладочную информацию

Полный список всех доступных флагов проверяет документы

Встроенные флаги

Из документов :

> `(?iLmsux)` (одна или несколько букв из набора «i», «L», «m», «s», «u», «x».)
>
> Группа соответствует пустой строке; буквы устанавливают соответствующие флаги: re.I (игнорировать регистр), re.L (зависит от локали), re.M (многострочный), re.S (точка соответствует всем), re.U (зависит от Юникода) и re.X (verbose), для всего регулярного выражения. Это полезно, если вы хотите включить флаги как часть обычного выражения, вместо передачи аргумента флага функции re.compile ().
>
> Обратите внимание, что флаг (? X) изменяет способ анализа выражения. Он должен использоваться сначала в строке выражения или после одного или нескольких символов пробелов. Если перед флагом есть символы без пробелов, результаты не определены.

Итерация по совпадениям с использованием `re.finditer`

Вы можете использовать `re.finditer` для `re.finditer` всех совпадений в строке. Это дает вам (по сравнению с дополнительной информацией `re.findall` , такой как информация о местоположении соответствия в строке (индексы):

```
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))
```

Результат:

```
Match "an" found at: [5,7]
Match "an" found at: [20,22]
Match "ant" found at: [23,26]
```

Соответствовать выражению только в определенных местах

Часто вы хотите сопоставить выражение только в *определенных* местах (оставив их нетронутыми в других, то есть). Рассмотрим следующее предложение:

```
An apple a day keeps the doctor away (I eat an apple everyday).
```

Здесь «яблоко» происходит дважды, что можно решить с помощью так называемых *контрольных глаголов backtracking,* которые поддерживаются новым модулем `regex` . Идея такова:

```
forget_this | or this | and this as well | (but keep this)
```

С нашим примером яблока это будет:

```
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
```

Это соответствует «яблоку» только тогда, когда ее можно найти за пределами круглых скобок.

Вот как это работает:

- При просмотре **слева направо** двигатель регулярного выражения потребляет все влево, `(*SKIP)` действует как «всегда-истинное утверждение». Впоследствии он корректно выходит из строя `(*FAIL)` и возвращается.
- Теперь он доходит до точки `(*SKIP)` **справа налево** (ака, в то время как возвращается), где запрещено идти дальше влево. Вместо этого двигателю предлагается выбросить что-либо влево и перейти к точке, где был вызван `(*SKIP)` .

Прочитайте Регулярные выражения (регулярное выражение) онлайн:
https://riptutorial.com/ru/python/topic/632/регулярные-выражения--регулярное-выражение-

глава 158: Рекурсия

замечания

Рекурсия требует остановки `stopCondition` для выхода из рекурсии.

Первоначальная переменная должна быть передана рекурсивной функции, поэтому она будет сохранена.

Examples

Сумма чисел от 1 до n

Если бы я хотел узнать сумму чисел от `1` до `n` где `n` - натуральное число, я могу сделать `1 + 2 + 3 + 4 + ... + (several hours later) + n` . В качестве альтернативы я мог бы написать цикл `for` :

```
n = 0
for i in range (1, n+1):
    n += i
```

Или я мог бы использовать технику, известную как рекурсия:

```
def recursion(n):
    if n == 1:
        return 1
    return n + recursion(n - 1)
```

Рекурсия имеет преимущества по сравнению с вышеприведенными двумя методами. Рекурсия занимает меньше времени, чем запись `1 + 2 + 3` для суммы от 1 до 3. Для `recursion(4)` рекурсия может использоваться для обратной работы:

Функциональные вызовы: (4 -> 4 + 3 -> 4 + 3 + 2 -> 4 + 3 + 2 + 1 -> 10)

В то время `for` цикл `for` работает строго вперед: (1 -> 1 + 2 -> 1 + 2 + 3 -> 1 + 2 + 3 + 4 -> 10). Иногда рекурсивное решение проще, чем итеративное решение. Это очевидно при реализации обращения к связанному списку.

Что, как и когда рекурсия

Рекурсия происходит, когда вызов функции вызывает повторение той же функции до того, как завершение вызова функции функции завершается. Например, рассмотрим известное математическое выражение `x!` (т.е. факториальной операции). Факториальная операция определена для всех неотрицательных целых чисел следующим образом:

- Если число равно 0, то ответ равен 1.
- В противном случае ответ заключается в том, что число раз факториал одного меньше, чем это число.

В Python наивная реализация факториальной операции может быть определена как функция следующим образом:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

Иногда функции рекурсии трудно схватить, поэтому давайте поэтапно пройдемся. Рассмотрим выражение `factorial(3)` . Это и *все* вызовы функций создают новую **среду** . Среда в основном представляет собой таблицу, которая сопоставляет идентификаторы (например, `n` , `factorial` , `print` и т. Д.) С их соответствующими значениями. В любой момент времени вы можете получить доступ к текущей среде с помощью `locals()` . В первом вызове функции единственная локальная переменная, которая определяется, равна `n = 3` . Поэтому печать `locals()` будет показывать `{'n': 3}` . Так как `n == 3` , возвращаемое значение становится `n * factorial(n - 1)` .

На этом следующем этапе ситуация может немного запутаться. Глядя на наше новое выражение, мы уже знаем, что такое `n` . Однако мы еще не знаем, что такое `factorial(n - 1)` . Во-первых, `n - 1` оценивается до `2` . Затем `2` передается `factorial` как значение для `n` . Поскольку это новый вызов функции, создается вторая среда для хранения этого нового `n` . Пусть *A* - первое окружение, а *B* - второе окружение. *A* все еще существует и равен `{'n': 3}` , однако *B* (что равно `{'n': 2}`) является текущей средой. Глядя на тело функции, возвращаемое значение, опять же, `n * factorial(n - 1)` . Не оценивая это выражение, заменим его на исходное выражение return. Делая это, мы мысленно отбрасываем *B* , поэтому не забудьте заменить `n` соответственно (т.е. ссылки на *B* `n` заменены на `n - 1` который использует *A* `n`). Теперь исходное обратное выражение становится `n * ((n - 1) * factorial((n - 1) - 1))` . Сделайте секунду, чтобы понять, почему это так.

Теперь давайте оценим `factorial((n - 1) - 1))` . Так как *A* `n == 3` , мы переходим `1` к `factorial` . Поэтому мы создаем новую среду *C,* которая равна `{'n': 1}` . Опять же, возвращаемое значение `n * factorial(n - 1)` . Итак, заменим `factorial((n - 1) - 1))` выражения «original» return аналогично тому, как раньше мы скорректировали исходное выражение return. « Оригинальное» выражение теперь `n * ((n - 1) * ((n - 2) * factorial((n - 2) - 1)))` .

Почти сделано. Теперь нам нужно оценить `factorial((n - 2) - 1)` . На этот раз мы пройдем `0` . Следовательно, это оценивается в `1` . Теперь давайте проведем нашу последнюю замену. «Оригинальное» выражение возврата теперь `n * ((n - 1) * ((n - 2) * 1))` . Напомнив, что исходное выражение возврата оценивается под *A* , выражение становится `3 * ((3 - 1) * ((3 - 2) * 1))` . Это, конечно, оценивается до 6. Чтобы подтвердить, что это правильный

ответ, напомните, что `3! == 3 * 2 * 1 == 6` . Прежде чем читать дальше, убедитесь, что вы полностью понимаете концепцию среды и то, как они применяются к рекурсии.

Утверждение, `if n == 0: return 1` , называется базовым. Это потому, что он не рекурсии. Требуется базовый корпус. Без этого вы столкнетесь с бесконечной рекурсией. С учетом сказанного, если у вас есть хотя бы один базовый случай, у вас может быть столько случаев, сколько вы хотите. Например, мы могли бы эквивалентно записать `factorial` следующим образом:

```
def factorial(n):
    if n == 0:
        return 1
    elif n == 1:
        return 1
    else:
        return n * factorial(n - 1)
```

У вас может также быть несколько случаев рекурсии, но мы не будем вдаваться в это, потому что это относительно редко, и часто трудно мысленно обрабатывать.

Вы также можете иметь «параллельные» рекурсивные вызовы функций. Например, рассмотрим последовательность Фибоначчи, которая определяется следующим образом:

- Если число равно 0, то ответ равен 0.
- Если число равно 1, то ответ равен 1.
- В противном случае ответ представляет собой сумму двух предыдущих чисел Фибоначчи.

Мы можем определить это следующим образом:

```
def fib(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib(n - 2) + fib(n - 1)
```

Я не буду проходить эту функцию так же тщательно, как и с `factorial(3)` , но окончательное значение возврата `fib(5)` эквивалентно следующему (*синтаксически* недействительному) выражению:

```
(
  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)
    )
  )
)
```

Это становится `(1 + (0 + 1)) + ((0 + 1) + (1 + (0 + 1)))` который, конечно, оценивается до `5` .

Теперь давайте рассмотрим еще несколько терминов словаря:

- **Хвост вызова** - это просто **вызов** рекурсивной функции, который является последней операцией, которая должна быть выполнена перед возвратом значения. Чтобы быть ясным, `return foo(n - 1)` является хвостовым вызовом, но `return foo(n - 1) + 1` не является (поскольку добавление является последней операцией).
- **Оптимизация звонков** (TCO) - это способ автоматического сокращения рекурсии в рекурсивных функциях.
- **Устранение хвостового вызова** (TCE) - это сокращение хвостового вызова до выражения, которое может быть оценено без рекурсии. TCE - тип TCO.

Оптимизация звонков может быть полезной по нескольким причинам:

- Интерпретатор может минимизировать объем памяти, занятой средами. Поскольку ни один компьютер не имеет неограниченной памяти, чрезмерные рекурсивные вызовы функций приведут к **переполнению стека** .
- Интерпретатор может уменьшить количество переключателей **кадров стека** .

Python не имеет формы TCO, реализованной по ряду причин . Поэтому для ограничения этого ограничения необходимо использовать другие методы. Метод выбора зависит от варианта использования. С некоторой интуицией определения `factorial` и `fib` можно относительно легко преобразовать в итеративный код следующим образом:

```
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
```

Обычно это самый эффективный способ ручного устранения рекурсии, но для более сложных функций может оказаться довольно сложным.

Другим полезным инструментом является декодер lru_cache Python, который можно использовать для уменьшения количества избыточных вычислений.

Теперь у вас есть идея о том, как избежать рекурсии в Python, но когда вы *должны* использовать рекурсию? Ответ «не часто». Все рекурсивные функции могут быть реализованы итеративно. Это просто вопрос, как это сделать. Однако есть редкие случаи, когда рекурсия в порядке. Рекурсия распространена в Python, когда ожидаемые входы не вызовут значительного количества вызовов рекурсивных функций.

Если рекурсия - это тема, которая вас интересует, я умоляю вас изучить функциональные языки, такие как Scheme или Haskell. На таких языках рекурсия гораздо более полезна.

Обратите внимание, что приведенный выше пример последовательности Фибоначчи, хотя он хорошо показывает, как применять определение в python и позже использовать кэш lru, имеет неэффективное время работы, так как он делает 2 рекурсивных вызова для каждого не базового случая. Количество вызовов функции растет экспоненциально до `n` . Скорее неинтуитивно более эффективная реализация будет использовать линейную рекурсию:

```
def fib(n):
    if n <= 1:
        return (n,0)
    else:
        (a, b) = fib(n - 1)
        return (a + b, a)
```

Но у этого есть вопрос о возвращении *пары* чисел. Это подчеркивает, что некоторые функции действительно не сильно выигрывают от рекурсии.

Исследование деревьев с рекурсией

Скажем, у нас есть следующее дерево:

```
root
- A
  - AA
  - AB
- B
  - BA
  - BB
    - BBA
```

Теперь, если мы хотим перечислить все имена элементов, мы могли бы сделать это с помощью простого цикла for. Мы предполагаем, что существует функция `get_name()` чтобы вернуть строку имени узла, функцию `get_children()` чтобы вернуть список всех `get_children()` данного узла в дереве, а также функцию `get_root()` для получить корневой узел.

```
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
```

Это работает хорошо и быстро, но что, если суб-узлы получили собственные узлы? И эти узлы могут иметь больше суб-узлов ... Что делать, если вы заранее не знаете, сколько их будет? Метод решения этой проблемы - использование рекурсии.

```
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
```

Возможно, вы не хотите печатать, но возвращаете плоский список всех имен узлов. Это можно сделать, передав список в качестве параметра.

```
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']
```

Увеличение максимальной глубины рекурсии

Существует предел глубине возможной рекурсии, которая зависит от реализации Python. Когда предел достигнут, возникает исключение RuntimeError:

```
RuntimeError: Maximum Recursion Depth Exceeded
```

Вот пример программы, которая вызовет эту ошибку:

```
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!
```

Можно изменить предел глубины рекурсии, используя

```
sys.setrecursionlimit(limit)
```

Вы можете проверить, какие текущие параметры лимита выполняются:

```
sys.getrecursionlimit()
```

Выполняя тот же метод выше с нашим новым пределом, получим

```
sys.setrecursionlimit(2000)
cursing(0)
# Out: I recursed 1997 times!
```

Из Python 3.5 исключение представляет собой RecursionError, который получен из RuntimeError.

Рекурсия хвоста - плохая практика

Когда единственная вещь, возвращаемая функцией, является рекурсивным вызовом, она называется хвостовой рекурсией.

Вот пример обратного отсчета с использованием хвостовой рекурсии:

```
def countdown(n):
    if n == 0:
        print "Blastoff!"
    else:
        print n
        countdown(n-1)
```

Любые вычисления, которые могут быть сделаны с использованием итерации, также могут быть выполнены с использованием рекурсии. Вот версия find_max, написанная с использованием хвостовой рекурсии:

```
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)
```

Рекурсия хвоста считается плохой практикой в Python, поскольку компилятор Python не

обрабатывает оптимизацию для хвостовых рекурсивных вызовов. Рекурсивное решение в таких случаях использует больше системных ресурсов, чем эквивалентное итеративное решение.

Оптимизация регенерации хвоста посредством интроспекции стека

По умолчанию рекурсивный стек Python не может превышать 1000 кадров. Это можно изменить, установив `sys.setrecursionlimit(15000)` который быстрее, однако этот метод потребляет больше памяти. Вместо этого мы также можем решить проблему рекурсии хвоста, используя интроспекцию стека.

```
#!/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
```

Чтобы оптимизировать рекурсивные функции, мы можем использовать декоратор `@tail_call_optimized` для вызова нашей функции. Вот несколько примеров общей рекурсии с использованием декоратора, описанного выше:

Факториальный пример:

```
@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.
```

Пример Фибоначчи:

```
@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.
```

Прочитайте Рекурсия онлайн: https://riptutorial.com/ru/python/topic/1716/рекурсия

глава 159: Розетки

Вступление

Многие языки программирования используют сокеты для связи между процессами или между устройствами. В этом разделе объясняется правильное использование модуля сокетов в Python для облегчения отправки и приема данных по общим сетевым протоколам.

параметры

параметр	Описание
socket.AF_UNIX	Разъем UNIX
socket.AF_INET	IPv4
socket.AF_INET6	IPv6
socket.SOCK_STREAM	TCP
socket.SOCK_DGRAM	UDP

Examples

Отправка данных через UDP

UDP - протокол без установления соединения. Сообщения другим процессам или компьютерам отправляются без установления какого-либо соединения. Нет автоматического подтверждения, если ваше сообщение получено. UDP обычно используется в чувствительных к задержкам приложениях или в приложениях, отправляющих широковещательные широковещательные передачи.

Следующий код отправляет сообщение процессу, прослушивающему локальный порт 6667 с использованием UDP

Обратите внимание, что нет необходимости «закрывать» сокет после отправки, потому что UDP является без установления соединения .

```
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))
```

Получение данных через UDP

UDP - протокол без установления соединения. Это означает, что одноранговые сообщения, отправляющие сообщения, не требуют установления соединения перед отправкой сообщений. `socket.recvfrom` образом, `socket.recvfrom` возвращает кортеж (`msg` [сообщение, полученное сокетом], `addr` [адрес отправителя])

UDP-сервер, использующий исключительно модуль `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))
```

Ниже приведена альтернативная реализация с использованием `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()
```

По умолчанию блок `sockets` . Это означает, что выполнение скрипта будет ждать, пока сокет получит данные.

Отправка данных через TCP

Отправка данных через Интернет стала возможной благодаря использованию нескольких модулей. Модуль сокетов обеспечивает низкоуровневый доступ к операциям операционной системы, отвечающим за отправку или получение данных от других компьютеров или процессов.

Следующий код отправляет байтовую строку `b'Hello'` на TCP-сервер, прослушивающий порт 6667 на локальном хосте и закрывает соединение по завершении:

```
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()
```

Выход сокета блокируется по умолчанию, это означает, что программа будет ждать в соединениях и отправлять вызовы до тех пор, пока действие не будет завершено. Для подключения это означает, что сервер фактически принимает соединение. Для отправки это означает только, что операционная система имеет достаточное пространство для хранения очереди для отправки данных позже.

Розетки всегда должны быть закрыты после использования.

Многопоточный сервер TCP Socket

При запуске без аргументов эта программа запускает сервер сокетов TCP, который прослушивает подключения к `127.0.0.1` на порту `5000` . Сервер обрабатывает каждое соединение в отдельном потоке.

При запуске с аргументом `-c` эта программа подключается к серверу, считывает список клиентов и распечатывает его. Список клиентов переносится как строка JSON. Имя клиента может быть указано путем передачи аргумента `-n` . Передавая разные имена, можно наблюдать влияние на список клиентов.

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()
```

Выход сервера

```
$ python client_list.py
Starting server...
```

Выход клиента

```
$ python client_list.py -c -n name1
{
    "name1": {
        "address": "127.0.0.1",
        "port": 62210,
        "name": "name1"
    }
}
```

Буферы приема ограничены 1024 байтами. Если строковое представление JSON списка клиентов превышает этот размер, оно будет усечено. Это приведет к возникновению следующего исключения:

```
ValueError: Unterminated string starting at: line 1 column 1023 (char 1022)
```

Необработанные сокеты в Linux

Сначала вы отключите автоматическое контрольное суммирование вашей сетевой карты:

```
sudo ethtool -K eth1 tx off
```

Затем отправьте свой пакет, используя гнездо 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)
```

Прочитайте Розетки онлайн: https://riptutorial.com/ru/python/topic/1530/розетки

глава 160: Связанные списки

Вступление

Связанный список представляет собой набор узлов, каждый из которых состоит из ссылки и значения. Узлы объединяются в последовательность, используя их ссылки. Связанные списки могут использоваться для реализации более сложных структур данных, таких как списки, стеки, очереди и ассоциативные массивы.

Examples

Пример с одним связанным списком

В этом примере реализуется связанный список со многими из тех же методов, что и встроенный объект списка.

```
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()
```

Функции использования похожи на функции встроенного списка.

```
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
```

Прочитайте Связанные списки онлайн: https://riptutorial.com/ru/python/topic/9299/связанные-списки

глава 161: Сериализация данных

Синтаксис

- unpickled_string = pickle.loads (строка)
- unpickled_string = pickle.load (file_object)
- pickled_string = pickle.dumps ([('', 'cmplx'), {('object',): None}], pickle.HIGHEST_PROTOCOL)
- pickle.dump (('', 'cmplx'), {('object',): None}], file_object, pickle.HIGHEST_PROTOCOL)
- unjsoned_string = json.loads (строка)
- 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)

параметры

параметр	подробности
`protocol`	Используя `pickle` или `cPickle` , это метод, который объекты сериализуются / несериализуются. Вероятно, вы захотите использовать `pickle.HIGHEST_PROTOCOL` здесь, что означает новейший метод.

замечания

Зачем использовать JSON?

- Поддержка перекрестных языков
- Человек читаемый
- В отличие от рассола, у него нет опасности запуска произвольного кода

Почему бы не использовать JSON?

- Не поддерживает типы данных Pythonic
- Ключи в словарях не должны быть отличными от строковых типов данных.

Почему рассол?

- Отличный способ для сериализации Pythonic (кортежи, функции, классы)
- Ключи в словарях могут быть любого типа данных.

Почему не рассол?

- Отсутствует поддержка перекрестного языка

- Это не безопасно для загрузки произвольных данных

Examples

Сериализация с использованием JSON

JSON - это кросс-язык, широко используемый метод сериализации данных

Поддерживаемые типы данных: *int* , *float* , *boolean* , *string* , *list* и *dict* . Смотрите -> JSON Wiki для получения дополнительной информации

Вот пример, демонстрирующий **основное** использование **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)
```

Дополнительную информацию о JSON см. В JSON-модуле .

Сериализация с использованием Pickle

Вот пример, демонстрирующий **основное** использование **рассола** : -

```
# 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)
```

См. Pickle для получения более подробной информации о Pickle.

ПРЕДУПРЕЖДЕНИЕ . Официальная документация на рассол дает четкое представление о том, что гарантий безопасности нет. Не загружайте данные, которым вы не доверяете.

Прочитайте Сериализация данных онлайн: https://riptutorial.com/ru/python/topic/3347/сериализация-данных

глава 162: Сериализация данных сортировки

Синтаксис

- pickle.dump (объект, файл, протокол) # Для сериализации объекта
- pickle.load (file) # Чтобы де-сериализовать объект
- pickle.dumps (объект, протокол) # Сериализация объекта в байтах
- pickle.loads (buffer) # Для де-сериализации объекта из байтов

параметры

параметр	подробности
объект	Объект, который должен быть сохранен
файл	Открытый файл, который будет содержать объект
протокол	Протокол, используемый для травления объекта (необязательный параметр)
буфер	Объект bytes, который содержит сериализованный объект

замечания

Распылительные типы

Следующие объекты являются picklable.

- `None` , `True` и `False`
- числа (всех типов)
- строки (всех типов)
- `tuple` s, `list` s, `set` s и `dict` s, содержащий только сортируемые объекты
- функции, определенные на верхнем уровне модуля
- встроенные функции
- классы, которые определены на верхнем уровне модуля
  - экземпляры таких классов, чей `__dict__` или результат вызова `__getstate__()`

являются picklable (подробности см. в официальных документах).

На основе официальной документации Python .

`pickle` и безопасность

Модуль рассола **не** является **безопасным** . Он не должен использоваться при получении сериализованных данных от ненадежной стороны, например, через Интернет.

Examples

Использование Pickle для сериализации и десериализации объекта

Модуль `pickle` реализует алгоритм превращения произвольного объекта Python в ряд байтов. Этот процесс также называется **сериализацией** объекта. Затем поток байтов, представляющий объект, может быть передан или сохранен, а затем реконструирован для создания нового объекта с теми же характеристиками.

Для простейшего кода мы используем функции `dump()` и `load()` .

Сериализация объекта

```
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)
```

Для десериализации объекта

```
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)
```

Использование объектов рассола и байтов

Также возможно сериализовать и дезаминировать из байтовых объектов, используя функцию `dumps` и `loads` , которые эквивалентны `dump` и `load` .

```
serialized_data = pickle.dumps(data, pickle.HIGHEST_PROTOCOL)
# type(serialized_data) is bytes

deserialized_data = pickle.loads(serialized_data)
# deserialized_data == data
```

Настроить маринованные данные

Некоторые данные нельзя травить. Другие данные не следует мариновать по другим причинам.

То, что будет маринован, может быть определено в методе `__getstate__` . Этот метод должен возвращать то, что может быть сорвано.

На противоположной стороне `__setstate__` : он получит то, что создал `__getstate__` и должен инициализировать объект.

```
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
```

Теперь это можно сделать:

```
>>> 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
```

Реализация здесь отображает список с одним значением: `[self.important_data]` . Это был всего лишь пример, `__getstate__` мог бы вернуть все, что можно выбрать, если `__setstate__` знает, как сделать oppoisite. Хорошей альтернативой является словарь всех значений: `{'important_data': self.important_data}` .

Конструктор не называется! Обратите внимание, что в предыдущем примере экземпляр `a2` был создан в `pickle.loads` без вызова `A.__init__` , поэтому `A.__setstate__` должен был инициализировать все, что `__init__` инициализировалось бы, если бы оно было `A.__setstate__` .

Прочитайте Сериализация данных сортировки онлайн: https://riptutorial.com/ru/python/topic/2606/сериализация-данных-сортировки

глава 163: Сеть Python

замечания

(Очень) базовый пример сокета клиента Python

Examples

Простейший пример клиент-сервера сокета Python

Серверная сторона:

```
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
```

Сторона клиента:

```
import socket

clientsocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
clientsocket.connect(('localhost', 8089))
clientsocket.send('hello')
```

Сначала запустите SocketServer.py и убедитесь, что сервер готов слушать / получать sth. Затем клиент отправляет информацию на сервер; После того, как сервер получил sth, он завершается

Создание простого Http-сервера

Чтобы обмениваться файлами или размещать простые веб-сайты (http и javascript) в вашей локальной сети, вы можете использовать встроенный модуль SimpleHTTPServer Python. Python должен находиться в переменной Path. Перейдите в папку, где находятся ваши файлы, и введите:

Для `python 2` :

```
$ python -m SimpleHTTPServer <portnumber>
```

Для `python 3` :

```
$ python3 -m http.server <portnumber>
```

Если номер порта не указан, то `8000` - это порт по умолчанию. Таким образом, выход будет:

> Обслуживание HTTP на сервере 0.0.0.0 8000 ...

Вы можете получить доступ к своим файлам через любое устройство, подключенное к локальной сети, набрав `http://hostipaddress:8000/` .

`hostipaddress` - ваш локальный IP-адрес, который, вероятно, начинается с `192.168.xx`

Чтобы закончить модуль, просто нажмите `ctrl+c`.

Создание TCP-сервера

Вы можете создать TCP-сервер, используя библиотеку `socketserver` . Вот простой эхо-сервер.

Серверная сторона

```
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()
```

Сторона клиента

```
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` упрощает создание простых TCP-серверов. Однако вы должны знать, что по умолчанию серверы однопоточные и могут обслуживать только одного клиента за раз. Если вы хотите обрабатывать несколько клиентов, вместо этого `ThreadingTCPServer` экземпляр `ThreadingTCPServer` .

```
from socketserver import ThreadingTCPServer
...
if __name__ == '__main__':
    server = ThreadingTCPServer(('', 5000), EchoHandler)
    server.serve_forever()
```

Создание UDP-сервера

UDP-сервер легко создается с использованием библиотеки `socketserver` .

простой сервер времени:

```
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()
```

Тестирование:

```
>>> 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))
```

Запустите Simple HttpServer в потоке и откройте браузер.

Полезно, если ваша программа выводит веб-страницы на этом пути.

```
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))
```

Прочитайте Сеть Python онлайн: https://riptutorial.com/ru/python/topic/1309/сеть-python

глава 164: системный

Вступление

Модуль **sys** обеспечивает доступ к функциям и значениям, относящимся к среде выполнения программы, например параметрам командной строки в `sys.argv` или функции `sys.exit()` чтобы завершить текущий процесс из любой точки потока программы.

В то время как он полностью разделен на модуль, он фактически встроен и как таковой всегда будет доступен при обычных обстоятельствах.

Синтаксис

- Импортируйте модуль sys и сделайте его доступным в текущем пространстве имен:

  ```
  import sys
  ```

- Импортируйте определенную функцию из модуля sys непосредственно в текущее пространство имен:

  ```
  from sys import exit
  ```

замечания

Подробнее о всех членах модуля **sys** см. В официальной документации .

Examples

Аргументы командной строки

```
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.
```

Обратите внимание, что в больших и более полированных программах вы должны использовать такие модули, как щелчок, чтобы обрабатывать аргументы командной строки, а не делать это самостоятельно.

Имя скрипта

```
# 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
```

Стандартный поток ошибок

```
# 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)
```

Прекращение процесса преждевременно и возврат кода выхода

```
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()
```

Прочитайте системный онлайн: https://riptutorial.com/ru/python/topic/9847/системный

глава 165: Скорость программы Python

Examples

нотация

Основная идея

Нотация, используемая при описании скорости вашей программы Python, называется нотой Big-O. Допустим, у вас есть функция:

```
def list_check(to_check, the_list):
    for item in the_list:
        if to_check == item:
          return True
    return False
```

Это простая функция для проверки того, находится ли элемент в списке. Чтобы описать сложность этой функции, вы скажете O (n). Это означает «порядок n», поскольку функция O известна как функция Order.

O (n) - обычно n - количество элементов в контейнере

O (k) - обычно k - значение параметра или количество элементов в параметре

Список операций

Операции: средний случай (предполагает, что параметры генерируются случайным образом)

Добавить: O (1)

Копировать: O (n)

Del slice: O (n)

Удалить элемент: O (n)

Вставка: O (n)

Получить элемент: O (1)

Установить элемент: O (1)

Итерация: O (n)

Получить срез: O (k)

Установить срез: O (n + k)

Расширение: O (k)

Сортировка: O (n log n)

Умножить: O (nk)

x в s: O (n)

min (s), max (s): O (n)

Получить длину: O (1)

Операции Deque

Дека - это двойная очередь.

```
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)
```

Операции: средний случай (предполагает, что параметры генерируются случайным образом)

Добавить: O (1)

Appendleft: O (1)

Копировать: O (n)

Расширение: O (k)

Extendleft: O (k)

Поп: O (1)

Popleft: O (1)

Удалить: O (n)

Повернуть: O (k)

Установить операции

Операция: средний случай (предполагает произвольные параметры): худший случай

x в s: O (1)

Разность s - t: O (len (s))

Пересечение s & t: O (мин (len (s), len (t))): O (len (s) * len (t)

Множественное пересечение s1 & s2 & s3 & ... & sn:: (n-1) * O (l) где l 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))

Симметрическая разность s ^ t: O (len (s)): O (len (s) * len (t))

Union s | t: O (len (s) + len (t))

Алгоритмические обозначения ...

Существуют определенные принципы, которые применяются к оптимизации на любом компьютерном языке, а Python не является исключением. **Не оптимизируйте, когда вы идете** : пишите свою программу без учета возможных оптимизаций, вместо этого концентрируясь на том, чтобы код был чистым, правильным и понятным. Если он слишком большой или слишком медленный, когда вы закончите, вы можете его оптимизировать.

Помните правило 80/20 : во многих полях вы можете получить 80% результата с 20% усилий (также называемое правилом 90/10 - это зависит от того, с кем вы говорите). Всякий раз, когда вы собираетесь оптимизировать код, используйте профилирование, чтобы узнать, где это 80% времени выполнения, поэтому вы знаете, где сосредоточить свои усилия.

Всегда выполняйте тесты «до» и «после» : как еще вы узнаете, что ваши оптимизации действительно повлияли? Если ваш оптимизированный код окажется немного быстрее или меньше оригинальной версии, отмените изменения и вернитесь к исходному, четкому коду.

Используйте правильные алгоритмы и структуры данных: не используйте алгоритм

сортировки пузырьков O (n2), чтобы сортировать тысячу элементов, когда есть доступная операционная система O (n log n). Точно так же не храните тысячу элементов в массиве, который требует поиска O (n), если вы можете использовать двоичное дерево O (log n) или хэш-таблицу O (1) Python.

Для более подробной информации см. Ссылку ниже ... Python Speed Up

Следующие 3 асимптотические обозначения в основном используются для представления временной сложности алгоритмов.

1. **Обозначение** : тета-нотация ограничивает функции сверху и снизу, поэтому она определяет точное асимптотическое поведение. Простым способом получить обозначение Тэта выражения является падение младших членов и игнорирование ведущих констант. Например, рассмотрим следующее выражение. 3n3 + 6n2 + 6000 = Θ (n3) Отбрасывание членов младшего порядка всегда отлично, потому что всегда будет n0, после которого Θ (n3) имеет более высокие значения, чем Θn2), независимо от используемых констант. Для данной функции g (n) обозначим Θ (g (n)) следующее множество функций. Θ (g (n)) = {f (n): существуют положительные константы c1, c2 и n0 такие, что 0 <= c1 *g (n) <= f (n) <= c2* g (n) для всех n> = n0} Вышеприведенное определение означает, что если f (n) является тетой g (n), то значение f (n) всегда находится между c1 *g (n) и c2* g (n) при больших значениях n (n> = n0). Определение theta также требует, чтобы f (n) была неотрицательной при значениях n, больших n0.

2. **Big O Notation** : **нота** Big O определяет верхнюю границу алгоритма, она ограничивает функцию только сверху. Например, рассмотрим случай сортировки вставки. В худшем случае требуется линейное время в лучшем случае и квадратичное время. Мы можем с уверенностью сказать, что временная сложность сортировки Insertion равна O (n ^ 2). Заметим, что O (n ^ 2) также охватывает линейное время. Если мы используем обозначение ation для представления временной сложности сортировки Insertion, мы должны использовать два утверждения для лучших и худших случаев:

1. Худшая временная сложность Сортировка Вставки - Θ (n ^ 2).
2. Наилучшая временная сложность Сортировка вставки - Θ (n).

Обозначение Big O полезно, когда у нас есть только верхняя граница по временной сложности алгоритма. Много раз мы легко находим верхнюю границу, просто просматривая алгоритм. O (g (n)) = {f (n): существуют положительные константы c и n0 такие, что 0 <= f (n) <= cg (n) для всех n> = n0}

3. **Ω Обозначение** : так же, как примечание Big O обеспечивает асимптотическую верхнюю границу функции, обозначение Ω дает асимптотическую нижнюю границу. Ω Обозначение <может быть полезно, когда мы имеем нижнюю границу по временной сложности алгоритма. Как обсуждалось в предыдущем сообщении, наилучшая производительность алгоритма обычно не полезна, нотация Omega является

наименее используемой нотацией среди всех трех. Для данной функции g (n) обозначим через Ω (g (n)) множество функций. Ω (g (n)) = {f (n): существуют положительные константы c и n0 такие, что 0 <= cg (n) <= f (n) для всех n> = n0}. Давайте рассмотрим тот же пример сортировки Insertion. Сложность времени сортировки вставки может быть записана как Ω (n), но это не очень полезная информация о сортировке вставки, поскольку нас обычно интересуют наихудший случай, а иногда и средний случай.

Прочитайте Скорость программы Python онлайн: https://riptutorial.com/ru/python/topic/9185/скорость-программы-python

глава 166: Скрытые функции

Examples

Перегрузка оператора

Все в Python - это объект. Каждый объект имеет некоторые специальные внутренние методы, которые он использует для взаимодействия с другими объектами. Как правило, эти методы следуют `__action__` об именах `__action__` . В совокупности это называется моделью данных Python .

Вы можете перегрузить *любой* из этих методов. Это обычно используется при перегрузке оператора в Python. Ниже приведен пример перегрузки оператора с использованием модели данных Python. Класс `Vector` создает простой вектор двух переменных. Мы добавим соответствующую поддержку математических операций двух векторов с использованием перегрузки операторов.

```
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)>
```

Приведенный выше пример демонстрирует перегрузку основных числовых операторов. Полный список можно найти здесь .

Прочитайте Скрытые функции онлайн: https://riptutorial.com/ru/python/topic/946/скрытые-функции

глава 167: Сложная математика

Синтаксис

- cmath.rect (AbsoluteValue, Phase)

Examples

Усовершенствованная сложная арифметика

Модуль `cmath` включает дополнительные функции для использования комплексных чисел.

```
import cmath
```

Этот модуль может рассчитать фазу комплексного числа в радианах:

```
z = 2+3j # A complex number
cmath.phase(z) # 0.982793723247329
```

Он позволяет преобразовать между декартовыми (прямоугольными) и полярными представлениями комплексных чисел:

```
cmath.polar(z) # (3.605551275463989, 0.982793723247329)
cmath.rect(2, cmath.pi/2) # (0+2j)
```

Модуль содержит сложную версию

- Экспоненциальные и логарифмические функции (как обычно, `log` - натуральный логарифм, `log10` - десятичный логарифм):

  ```
  cmath.exp(z) # (-7.315110094901103+1.0427436562359045j)
  cmath.log(z) # (1.2824746787307684+0.982793723247329j)
  cmath.log10(-100) # (2+1.3643763538418412j)
  ```

- Квадратные корни:

  ```
  cmath.sqrt(z) # (1.6741492280355401+0.8959774761298381j)
  ```

- Тригонометрические функции и их обратные:

  ```
  cmath.sin(z)  # (9.15449914691143-4.168906959966565j)
  cmath.cos(z)  # (-4.189625690968807-9.109227893755337j)
  cmath.tan(z)  # (-0.003764025641504249+1.00323862735361j)
  cmath.asin(z) # (0.5706527843210994+1.9833870299165355j)
  cmath.acos(z) # (1.0001435424737972-1.9833870299165355j)
  ```

```
cmath.atan(z) # (1.4099210495965755+0.22907268296853878j)
cmath.sin(z)**2 + cmath.cos(z)**2 # (1+0j)
```

- Гиперболические функции и их обратные:

```
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
```

Основная сложная арифметика

Python имеет встроенную поддержку комплексной арифметики. Мнимая единица обозначается через `j` :

```
z = 2+3j # A complex number
w = 1-7j # Another complex number
```

Сложные числа могут быть суммированы, вычтены, умножены, делены и экспоненциальны:

```
z + w # (3-4j)
z - w # (1+10j)
z * w # (23-11j)
z / w # (-0.38+0.34j)
z**3  # (-46+9j)
```

Python также может извлекать действительную и мнимую части комплексных чисел и вычислять их абсолютное значение и сопрягать:

```
z.real # 2.0
z.imag # 3.0
abs(z) # 3.605551275463989
z.conjugate() # (2-3j)
```

Прочитайте Сложная математика онлайн: https://riptutorial.com/ru/python/topic/1142/сложная-математика

глава 168: Случайный модуль

Синтаксис

- random.seed (a = None, version = 2) (версия доступна только для python 3.x)
- random.getstate ()
- random.setstate (состояние)
- random.randint (a, b)
- random.randrange (остановка)
- random.randrange (начало, остановка, шаг = 1)
- random.choice (сл)
- random.shuffle (x, random = random.random)
- random.sample (population, k)

Examples

Случайные и последовательности: перетасовка, выбор и выборка

```
import random
```

перетасовать ()

Вы можете использовать `random.shuffle()` для смешивания / рандомизации элементов в **изменяемой и индексируемой** последовательности. Например, `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!
```

выбор()

Принимает случайный элемент из произвольной **последовательности** :

```
print(random.choice(laughs))
# Out: He                  # Output may vary!
```

образец()

Подобно `choice` он принимает случайные элементы из произвольной **последовательности,** но вы можете указать, сколько:

```
#                   |--sequence--|--number--|
print(random.sample(    laughs   ,      1    ))  # Take one element
# Out: ['Ho']                    # Output may vary!
```

он не будет принимать один и тот же элемент дважды:

```
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: выборка больше, чем население

Создание случайных целых чисел и поплавков: randint, randrange, random и равномерное

```
import random
```

randint ()

Возвращает случайное целое число между `x` и `y` (включительно):

```
random.randint(x, y)
```

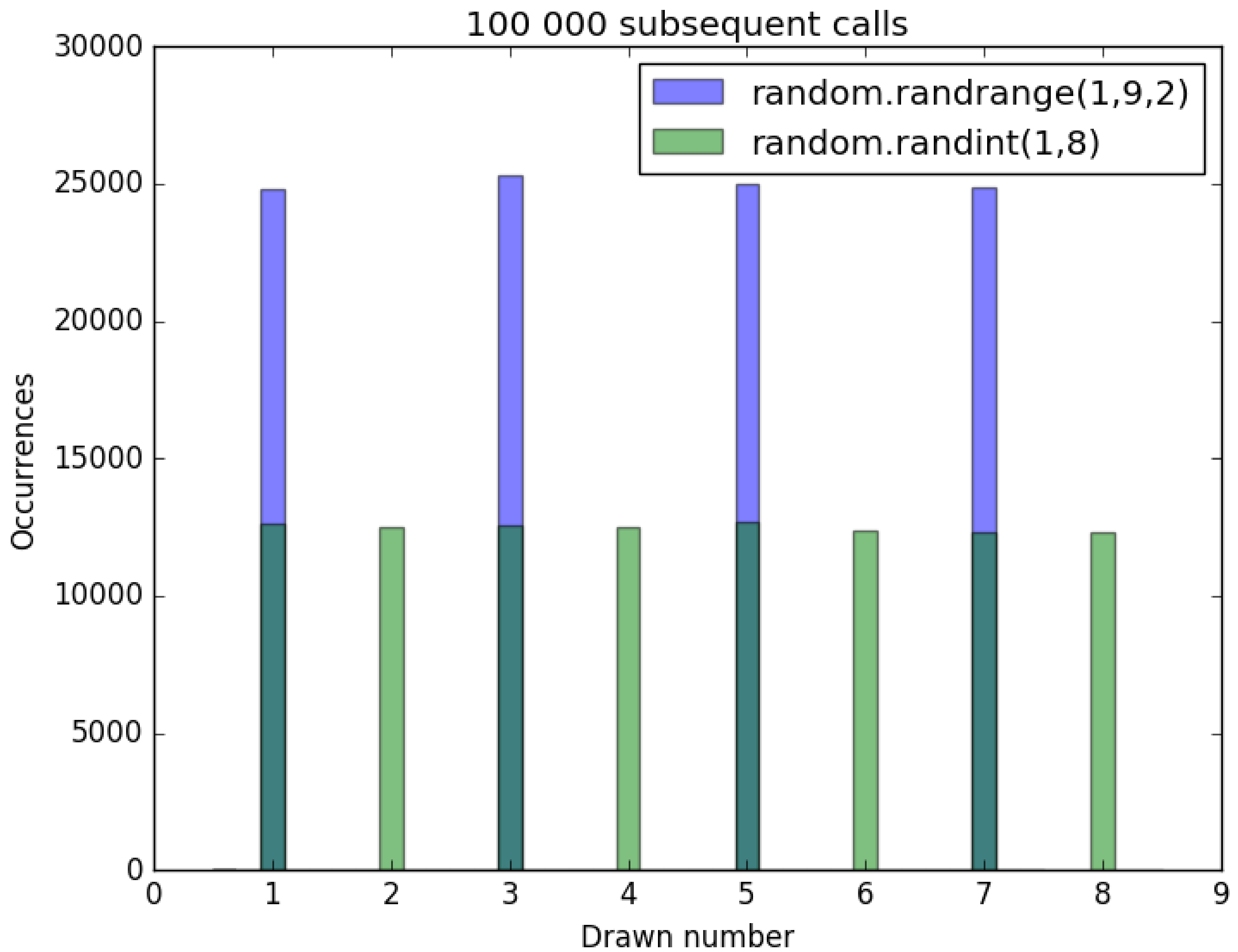
Например, получение случайного числа от `1` до `8` :

```
random.randint(1, 8) # Out: 8
```

randrange ()

`random.randrange` имеет тот же синтаксис, что и `range` и в отличие от `random.randint` , последнее значение **не** включает:

```
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)
```



случайный

Возвращает случайное число с плавающей запятой между 0 и 1:

```
random.random() # Out: 0.66486093215306317
```

единообразный

Возвращает случайное число с плавающей запятой между `x` и `y` (включительно):

```
random.uniform(1, 8) # Out: 3.726062641730108
```

Воспроизводимые случайные числа: семена и состояние

Установка определенного семени создаст фиксированную последовательность случайных чисел:

```
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
```

Сброс семени снова приведет к повторному повторению последовательности:

```
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
```

Поскольку семя фиксируется, эти результаты всегда равны `9` и `4` . Если для определенных чисел не требуется только то, что значения будут одинаковыми, можно также просто использовать `getstate` и `setstate` для восстановления в предыдущем состоянии:

```
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
```

Чтобы псевдо-рандомизировать последовательность снова, вы `seed` `None` :

```
random.seed(None)
```

Или вызовите метод `seed` без аргументов:

```
random.seed()
```

Создание криптографически защищенных случайных чисел

По умолчанию случайный модуль Python использует Merrenne Twister `PRNG` для генерации случайных чисел, которые, хотя и подходят в таких областях, как симуляции, не соответствуют требованиям безопасности в более сложных условиях.

Чтобы создать криптографически безопасное псевдослучайное число, можно использовать `SystemRandom` который, используя `os.urandom` , может действовать как криптографически защищенный генератор псевдослучайных чисел, CPRNG .

Самый простой способ его использования - инициализация класса `SystemRandom` . Предоставленные методы аналогичны тем, которые экспортируются случайным модулем.

```
from random import SystemRandom
secure_rand_gen = SystemRandom()
```

Чтобы создать случайную последовательность из 10 `int` s в диапазоне `[0, 20]` , можно просто вызвать `randrange()` :

```
print([secure_rand_gen.randrange(10) for i in range(10)])
# [9, 6, 9, 2, 2, 3, 8, 0, 9, 9]
```

Чтобы создать случайное целое число в заданном диапазоне, можно использовать `randint` :

```
print(secure_rand_gen.randint(0, 20))
# 5
```

и, соответственно, для всех других методов. Интерфейс точно такой же, единственным изменением является генератор базового числа.

Вы также можете использовать `os.urandom` непосредственно для получения криптографически защищенных случайных байтов.

Создание случайного пароля пользователя

Чтобы создать случайный пароль пользователя, мы можем использовать символы, представленные в `string` модуле. В частности, `punctuation` символов пунктуации, `ascii_letters` для букв и `digits` для цифр:

```
from string import punctuation, ascii_letters, digits
```

Затем мы можем объединить все эти символы в имени с именами `symbols` :

```
symbols = ascii_letters + digits + punctuation
```

Удалите любой из них, чтобы создать пул символов с меньшим количеством элементов.

После этого мы можем использовать `random.SystemRandom` для генерации пароля. Для пароля длиной 10:

```
secure_random = random.SystemRandom()
password = "".join(secure_random.choice(symbols) for i in range(10))
print(password)  # '^@g;J?]M6e'
```

Обратите внимание, что другие процедуры, сделанные немедленно доступными `random` модулем - например, `random.choice` , `random.randint` и т. Д. - *непригодны* для

криптографических целей.

За шторами эти подпрограммы используют Mersenne Twister PRNG , который не удовлетворяет требованиям CSPRNG . Таким образом, в частности, вы не должны использовать какой-либо из них для создания паролей, которые вы планируете использовать. Всегда используйте экземпляр `SystemRandom` как показано выше.

Python 3.x 3.6

Начиная с Python 3.6 доступен модуль `secrets` , который предоставляет криптографически безопасную функциональность.

Цитируя официальную документацию , чтобы генерировать *«десятизначный буквенно-цифровой пароль с хотя бы одним строчным символом, по крайней мере одним символом верхнего регистра и не менее трех цифр»,* вы могли бы:

```
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
```

Случайное двоичное решение

```
import random

probability = 0.3

if random.random() < probability:
    print("Decision with probability 0.3")
else:
    print("Decision with probability 0.7")
```

Прочитайте Случайный модуль онлайн: https://riptutorial.com/ru/python/topic/239/случайный-модуль

глава 169: События, отправленные сервером Python

Вступление

Server Sent Events (SSE) - однонаправленное соединение между сервером и клиентом (обычно это веб-браузер), который позволяет серверу «нажимать» информацию клиенту. Это очень похоже на websockets и длительный опрос. Основное различие между SSE и websockets заключается в том, что SSE является однонаправленным, только сервер может отправлять информацию клиенту, где, как и в случае с websockets, обе могут отправлять информацию друг другу. SSE обычно считается гораздо более простым в использовании / внедрении, чем в websockets.

Examples

Фляжка 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

В этом примере используется асинхронная SSE-библиотека: 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()
```

Прочитайте События, отправленные сервером Python онлайн:
https://riptutorial.com/ru/python/topic/9100/события--отправленные-сервером-python

глава 170: Создание виртуальной среды с помощью virtualenvwrapper в окнах

Examples

Виртуальная среда с virtualenvwrapper для окон

Предположим, вам нужно работать над тремя различными проектами: проект A, проект B и проект C. Проект A и проект B нуждаются в python 3 и некоторых требуемых библиотеках. Но для проекта C вам нужны python 2.7 и зависимые библиотеки.

Поэтому лучше всего отделить эти среды проекта. Для создания отдельной виртуальной среды python необходимо выполнить следующие шаги:

Шаг 1. Установите pip с помощью этой команды: `python -m pip install -U pip`

Шаг 2. Затем установите пакет «virtualenvwrapper-win» с помощью команды (команда может быть выполнена оболочкой Windows Power):

`pip install virtualenvwrapper-win`

Шаг 3. Создайте новую среду virtualenv, используя команду: `mkvirtualenv python_3.5`

Шаг 4: Активируйте среду, используя команду:

```
workon < environment name>
```

Основные команды для 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.
```

Прочитайте Создание виртуальной среды с помощью virtualenvwrapper в окнах онлайн: https://riptutorial.com/ru/python/topic/9984/создание-виртуальной-среды-с-помощью-virtualenvwrapper-в-окнах

глава 171: Создание пакетов Python

замечания

Проект pypa sample содержит полный, легко изменяемый шаблон `setup.py` который демонстрирует широкий набор возможностей, которые могут предложить инструменты настройки.

Examples

Вступление

Для каждого пакета требуется файл `setup.py` который описывает пакет.

Рассмотрим следующую структуру каталогов для простого пакета:

```
+-- package_name
|       |
|       +-- __init__.py
|
+-- setup.py
```

`__init__.py` содержит только строку `def foo(): return 100` .

Следующий `setup.py` определит пакет:

```
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 отлично подходит для тестирования пакетов, не изменяя другие среды 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
```

Загрузка в PyPI

Как только ваш `setup.py` полностью работоспособен (см. Введение), очень легко загрузить пакет в PyPI .

Настройка файла .pypirc

Этот файл хранит логины и пароли для аутентификации ваших учетных записей. Он обычно хранится в вашем домашнем каталоге.

```
# .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
```

Безопаснее использовать `twine` для загрузки пакетов, поэтому убедитесь, что они установлены.

```
$ pip install twine
```

Зарегистрировать и загрузить в testpypi (необязательно)

Примечание . PyPI не позволяет перезаписывать загруженные пакеты , поэтому разумно сначала протестировать развертывание на специализированном тестовом сервере, например testpypi. Этот вариант будет обсуждаться. Перед загрузкой рассмотрите схему управления версиями для вашего пакета, например, для управления версиями календаря или семантического управления версиями .

Войдите в систему или создайте новую учетную запись на testpypi . Регистрация необходима только в первый раз, хотя регистрация более одного раза не является вредной.

```
$ python setup.py register -r pypitest
```

В корневой папке вашего пакета:

```
$ twine upload dist/* -r pypitest
```

Теперь ваш пакет должен быть доступен через вашу учетную запись.

тестирование

Сделайте тестовую виртуальную среду. Попробуйте `pip install` пакет пакета с помощью testpypi или 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
```

В случае успеха ваш пакет является наименее доступным. Вы можете рассмотреть возможность тестирования своего API также до окончательной загрузки в PyPI. Если пакет не прошел во время тестирования, не беспокойтесь. Вы все равно можете его исправить, повторно загрузить в testpypi и снова проверить.

Регистрация и загрузка в PyPI

Убедитесь, что `twine` установлен:

```
$ pip install twine
```

Войдите в систему или создайте новую учетную запись в PyPI .

```
$ python setup.py register -r pypi
$ twine upload dist/*
```

Это оно! Ваш пакет теперь в прямом эфире .

Если вы обнаружите ошибку, просто загрузите новую версию своего пакета.

Документация

Не забудьте указать хотя бы какую-то документацию для вашего пакета. PyPi принимает в качестве языка форматирования по умолчанию reStructuredText .

Прочти меня

Если в вашем пакете нет большой документации, `README.rst` то, что может помочь другим пользователям в файле `README.rst` . Когда файл готов, нужно еще один, чтобы показать PyPi, чтобы показать его.

Создайте файл `setup.cfg` и вставьте в него эти две строки:

```
[metadata]
description-file = README.rst
```

Обратите внимание: если вы попытаетесь поместить файл Markdown в свой пакет, PyPi прочитает его как чистый текстовый файл без какого-либо форматирования.

лицензирование

Часто бывает более чем желательно разместить файл `LICENSE.txt` в вашем пакете с одной из лицензий OpenSource, чтобы сообщить пользователям, могут ли они использовать ваш пакет, например, в коммерческих проектах, или если ваш код можно использовать с их лицензией.

В более читаемом виде некоторые лицензии объясняются в TL; DR .

Выполнение пакета

Если ваш пакет является не только библиотекой, но и частью кода, который может быть использован либо как витрина, либо как отдельное приложение, когда ваш пакет

установлен, поместите этот фрагмент кода в файл `__main__.py` .

Поместите `__main__.py` в папку `package_name` . Таким образом вы сможете запустить его прямо с консоли:

```
python -m package_name
```

Если файл `__main__.py` отсутствует, пакет не будет запущен с этой командой, и эта ошибка будет напечатана:

> python: нет модуля с именем `package_name.__main__;` 'package_name' - это пакет и не может быть выполнен непосредственно.

Прочитайте Создание пакетов Python онлайн: https://riptutorial.com/ru/python/topic/1381/создание-пакетов-python

глава 172: Создание службы Windows с использованием Python

Вступление

Безглавые процессы (без интерфейса) в Windows называются службами. Их можно контролировать (запускать, останавливать и т. Д.), Используя стандартные средства управления Windows, такие как консоль командной строки, панель Powershell или вкладка «Службы» в диспетчере задач. Хорошим примером может быть приложение, которое предоставляет сетевые службы, такие как веб-приложение, или, возможно, приложение резервного копирования, которое выполняет различные фоновые архивные задачи. Существует несколько способов создания и установки приложения Python как службы в Windows.

Examples

Сценарий Python, который может выполняться как служба

Модули, используемые в этом примере, являются частью pywin32 (расширения для Python для Windows). В зависимости от того, как вы установили Python, вам может потребоваться установить это отдельно.

```
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)
```

Это просто шаблон. Ваш код приложения, возможно, ссылающийся на отдельный скрипт, будет находиться в функции main ().

Вам также потребуется установить это как услугу. Лучшим решением для этого на данный момент является использование Non-sucking Service Manager . Это позволяет вам установить службу и предоставить графический интерфейс для настройки командной строки, выполняемой службой. Для Python вы можете это сделать, что создает сервис за один раз:

```
nssm install MyServiceName c:\python27\python.exe c:\temp\myscript.py
```

Где my_script.py - это скрипт шаблона выше, модифицированный для вызова вашего сценария приложения или кода в функции main (). Обратите внимание, что служба не запускает скрипт Python напрямую, он запускает интерпретатор Python и передает ему основной сценарий в командной строке.

Кроме того, вы можете использовать инструменты, предлагаемые в наборе ресурсов Windows Server Resource для вашей версии операционной системы, чтобы создать службу.

Запуск веб-приложения Flask в качестве сервиса

Это вариант типичного примера. Вам просто нужно импортировать скрипт приложения и вызвать его метод `run()` в функции `main()` службы. В этом случае мы также используем модуль многопроцессорности из-за проблемы с доступом к `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)
```

Адаптировано с http://stackoverflow.com/a/25130524/318488

Прочитайте Создание службы Windows с использованием Python онлайн: https://riptutorial.com/ru/python/topic/9065/создание-службы-windows-с-использованием-python

глава 173: Сокеты и шифрование / расшифровка сообщений между клиентом и сервером

Вступление

Криптография используется в целях безопасности. Существует не так много примеров шифрования / дешифрования в Python, использующих CREATE CREATE IDEA. **Цель этой документации:**

Расширение и внедрение схемы цифровой подписи RSA в связи между станциями. Использование Hash для целостности сообщения, то есть SHA-1. Создайте простой протокол передачи ключей. Шифровать ключ с помощью кода IDEA. Режим блочного шифрования - режим счетчика

замечания

Используемый язык: Python 2.7 (ссылка для загрузки: https://www.python.org/downloads/)

Используемая библиотека:

* **PyCrypto** (ссылка для загрузки: https://pypi.python.org/pypi/pycrypto)

* **PyCryptoPlus** (ссылка для скачивания: https://github.com/doegox/python-cryptoplus)

Установка библиотеки:

PyCrypto: разархивируйте файл. Перейдите в каталог и откройте терминал для linux (alt + ctrl + t) и CMD (сдвиньте + правый щелчок + выберите командную строку, открытую здесь) для окон. После этого напишите python setup.py install (Make Sure Python Environment правильно настроена в ОС Windows)

PyCryptoPlus: такая же, как и последняя библиотека.

Выполнение задач: задача разделяется на две части. Один из них - процесс рукопожатия, а другой - процесс коммуникации. Настройка гнезда:

- Как создание открытых и закрытых ключей, а также хэширование открытого ключа, нам нужно настроить сокет сейчас. Для настройки сокета нам нужно импортировать другой модуль со «импортным сокетом» и подключить (для клиента) или связать (для сервера) IP-адрес и порт с получением сокета от пользователя.

----------Сторона клиента----------

```
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))
```

---------- Серверная сторона ---------

```
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» позволит нам использовать функцию accept () и основные принципы обмена сообщениями. Вместо этого мы можем использовать «socket.AF_INET, socket.SOCK_DGRAM», но в то же время нам придется использовать setblocking (значение).

Процесс рукопожатия:

- (КЛИЕНТ) Первой задачей является создание открытого и закрытого ключа. Чтобы создать закрытый и открытый ключ, нам нужно импортировать некоторые модули. Это: из Crypto import Random и из Crypto.PublicKey import RSA. Чтобы создать ключи, нам нужно написать несколько простых строк кодов:

```
random_generator = Random.new().read
        key = RSA.generate(1024,random_generator)
        public = key.publickey().exportKey()
```

random_generator получен из модуля « **из Crypto import Random** ». Ключ получен из « **из Crypto.PublicKey import RSA** », который создаст закрытый ключ размером 1024, генерируя случайные символы. Публикация экспортирует открытый ключ из ранее сгенерированного закрытого ключа.

- (CLIENT). После создания открытого и закрытого ключа мы должны использовать хэш для открытого ключа для отправки на сервер с использованием хэша SHA-1. Чтобы использовать хэш SHA-1, нам нужно импортировать другой модуль, написав «import hashlib». Для хэш-ключа мы пишем две строки кода:

```
hash_object = hashlib.sha1(public)
hex_digest = hash_object.hexdigest()
```

Здесь hash_object и hex_digest - наша переменная. После этого клиент отправит hex_digest и public на сервер, и сервер проверит их, сравнив хэш с клиентом и новый хэш открытого ключа. Если новый хеш и хэш от клиента совпадают, он перейдет к следующей процедуре.

Поскольку публикация, отправленная от клиента, имеет форму строки, она не сможет использоваться как ключ на стороне сервера. Чтобы предотвратить это и преобразовать открытый ключ строки в открытый ключ rsa, нам нужно написать `server_public_key = RSA.importKey(getpbk)` , здесь getpbk - это открытый ключ от клиента.

- (SERVER) Следующий шаг - создать ключ сеанса. Здесь я использовал модуль «os» для создания случайного ключа «key = os.urandom (16)», который даст нам 16-битный длинный ключ, после чего я зашифровал этот ключ в «AES.MODE_CTR» и снова хешу с 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()
```

Таким образом, en_digest будет нашим ключом сеанса.

- (SERVER) Для окончательной части процесса рукопожатия необходимо зашифровать открытый ключ, полученный от клиента, и ключ сеанса, созданный на стороне сервера.

```
 #encrypting session key and public key
 E = server_public_key.encrypt(encrypto,16)
```

После шифрования сервер отправит ключ клиенту в виде строки.

- (CLIENT) После получения зашифрованной строки (открытого и сеансового ключа) с сервера клиент расшифрует их с помощью Private Key, который был создан ранее вместе с открытым ключом. Поскольку зашифрованный (открытый и сеансовый ключ) был в форме строки, теперь мы должны вернуть его в качестве ключа, используя eval (). Если дешифрование завершено, процесс рукопожатия завершается также, когда обе стороны подтверждают, что используют одни и те же ключи. Чтобы расшифровать:

```
 en = eval(msg)
 decrypt = key.decrypt(en)
 # hashing sha1
 en_object = hashlib.sha1(decrypt) en_digest = en_object.hexdigest()
```

Я использовал SHA-1 здесь, чтобы он был доступен для чтения на выходе.

Процесс коммуникации:

Для коммуникационного процесса мы должны использовать ключ сеанса с обеих сторон в качестве ключа для кодирования IDEA MODE_CTR. Обе стороны будут шифровать и дешифровать сообщения с помощью IDEA.MODE_CTR, используя ключ сеанса.

- (Шифрование) Для шифрования IDEA нам нужен ключ размером 16 бит и счетчик, который должен быть вызван. Счетчик является обязательным в MODE_CTR. Ключ сеанса, который мы зашифровали и хешировал, теперь имеет размер 40, который будет превышать предельный ключ шифрования IDEA. Следовательно, нам нужно уменьшить размер ключа сеанса. Для сокращения мы можем использовать обычный питон, встроенный в строку функций [значение: значение]. Если значение может быть любым значением в соответствии с выбором пользователя. В нашем случае я сделал « key [: 16]», где от ключа будет от 0 до 16 значений. Это преобразование можно было бы сделать многими способами, такими как клавиша [1:17] или клавиша [16:]. Следующая часть - создать новую функцию шифрования IDEA, написав IDEA.new (), которая будет принимать 3 аргумента для обработки. Первым аргументом будет KEY, вторым аргументом будет режим шифрования IDEA (в нашем случае IDEA.MODE_CTR), а третьим аргументом будет счетчик = который является обязательной вызываемой функцией. Счетчик = будет содержать размер строки, который будет возвращен функцией. Чтобы определить счетчик =, мы должны использовать разумные значения. В этом случае я использовал размер KEY, определяя лямбда. Вместо использования лямбда мы могли бы использовать Counter.Util, который генерирует случайное значение для counter =. Чтобы использовать Counter.Util, нам нужно импортировать модуль счетчика из crypto. Следовательно, код будет:

  ```
  ideaEncrypt = IDEA.new(key, IDEA.MODE_CTR, counter=lambda : key)
  ```

Определив «ideaEncrypt» как нашу переменную шифрования IDEA, мы можем использовать встроенную функцию шифрования для шифрования любого сообщения.

```
eMsg = ideaEncrypt.encrypt(whole)
#converting the encrypted message to HEXADECIMAL to readable eMsg =
eMsg.encode("hex").upper()
```

В этом сегменте кода целое является зашифрованным сообщением, а eMsg является зашифрованным сообщением. После шифрования сообщения я преобразовал его в HEXADECIMAL, чтобы сделать readable и upper () - встроенной функцией, чтобы сделать символы в верхнем регистре. После этого это зашифрованное сообщение будет отправлено на противоположную станцию для дешифрования.

- **(Дешифрование)**

Чтобы расшифровать зашифрованные сообщения, нам нужно будет создать другую переменную шифрования, используя те же аргументы и тот же ключ, но на этот раз переменная расшифрует зашифрованные сообщения. Код для этого же, как и в последний раз. Однако перед расшифровкой сообщений нам нужно декодировать сообщение из шестнадцатеричного числа, поскольку в нашей части шифрования мы закодировали зашифрованное сообщение в шестнадцатеричном виде, чтобы сделать чтение.

Следовательно, весь код будет:

```
decoded = newmess.decode("hex")
ideaDecrypt = IDEA.new(key, IDEA.MODE_CTR, counter=lambda: key)
dMsg = ideaDecrypt.decrypt(decoded)
```

Эти процессы будут выполняться как на стороне сервера, так и на стороне клиента для шифрования и дешифрования.

Examples

Реализация на стороне сервера

```
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")
```

Реализация клиентской стороны

```
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()
```

Прочитайте Сокеты и шифрование / расшифровка сообщений между клиентом и сервером онлайн: https://riptutorial.com/ru/python/topic/8710/сокеты-и-шифрование---расшифровка-сообщений-между-клиентом-и-сервером

глава 174: Сообщение Python Requests

Вступление

Документация для модуля запросов Python в контексте метода HTTP POST и его соответствующей функции Запросы

Examples

Простой пост

```
from requests import post

foo = post('http://httpbin.org/post', data = {'key':'value'})
```

Будет выполнять простую операцию HTTP POST. Опубликованные данные могут быть самыми большими форматами, однако наиболее важными являются пары ключевых значений.

Заголовки

Заголовки можно просмотреть:

```
print(foo.headers)
```

Пример ответа:

```
{'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'}
```

Заголовки также могут быть подготовлены до публикации:

```
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'})
```

кодирование

Кодирование можно установить и просмотреть так же:

```
 print(foo.encoding)

'utf-8'

foo.encoding = 'ISO-8859-1'
```

Проверка SSL

Запросы по умолчанию проверяют SSL-сертификаты доменов. Это можно переопределить:

```
foo = post('http://httpbin.org/post', data = {'key':'value'}, verify=False)
```

Перенаправление

Любое перенаправление будет выполняться (например, http до https), это также можно изменить:

```
foo = post('http://httpbin.org/post', data = {'key':'value'}, allow_redirects=False)
```

Если операция отправки была перенаправлена, к ней можно получить доступ:

```
print(foo.url)
```

Можно просмотреть полную историю переадресаций:

```
print(foo.history)
```

Формированные кодированные данные

```
from requests import post

payload = {'key1' : 'value1',
           'key2' : 'value2'
           }

foo = post('http://httpbin.org/post', data=payload)
```

Чтобы передать кодированные данные формы с последующей операцией, данные должны быть структурированы в виде словаря и представлены в качестве параметра данных.

Если данные не хотят быть закодированы в форме, просто передайте строку или целое число в параметр данных.

Поставка словаря в параметр json для запросов для автоматического форматирования

данных:

```
from requests import post

payload = {'key1' : 'value1', 'key2' : 'value2'}

foo = post('http://httpbin.org/post', json=payload)
```

Файл загружен

С модулем Requests его необходимо только предоставить дескриптор файла, а не содержимое, полученное с помощью `.read()` :

```
from requests import post

files = {'file' : open('data.txt', 'rb')}

foo = post('http://http.org/post', files=files)
```

Также можно указать имя файла, content_type и заголовки:

```
files = {'file': ('report.xls', open('report.xls', 'rb'), 'application/vnd.ms-excel',
{'Expires': '0'})}

foo = requests.post('http://httpbin.org/post', files=files)
```

Строки также могут быть отправлены в виде файла, поскольку они поставляются в качестве параметра `files` .

Несколько файлов

Несколько файлов могут быть поставлены так же, как один файл:

```
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)
```

Ответы

Коды ответов могут быть просмотрены после операции:

```
from requests import post

foo = post('http://httpbin.org/post', data={'data' : 'value'})
print(foo.status_code)
```

Возвращенные данные

Доступ к возвращаемым данным:

```
foo = post('http://httpbin.org/post', data={'data' : 'value'})
print(foo.text)
```

Сырьевые ответы

В тех случаях, когда вам нужно получить доступ к основному объекту urllib3 response.HTTPResponse, это можно сделать следующим образом:

```
foo = post('http://httpbin.org/post', data={'data' : 'value'})
res = foo.raw

print(res.read())
```

Аутентификация

Простая HTTP-аутентификация

Простая HTTP-аутентификация может быть достигнута следующим образом:

```
from requests import post

foo = post('http://natas0.natas.labs.overthewire.org', auth=('natas0', 'natas0'))
```

Это технически короткая рука для следующего:

```
from requests import post
from requests.auth import HTTPBasicAuth

foo = post('http://natas0.natas.labs.overthewire.org', auth=HTTPBasicAuth('natas0', 'natas0'))
```

Проверка подлинности HTTP-дайджеста

HTTP-дайджест Аутентификация выполняется очень схожим образом, для запросов задается другой объект:

```
from requests import post
from requests.auth import HTTPDigestAuth

foo = post('http://natas0.natas.labs.overthewire.org', auth=HTTPDigestAuth('natas0',
'natas0'))
```

Пользовательская аутентификация

В некоторых случаях встроенные механизмы аутентификации могут быть недостаточными, представьте себе этот пример:

Сервер настроен на прием аутентификации, если отправитель имеет правильную строку

пользовательского агента, определенное значение заголовка и предоставляет правильные учетные данные через базовую проверку подлинности HTTP. Для этого необходимо подготовить собственный класс проверки подлинности, подклассифицируя AuthBase, который является базой для реализации аутентификации запросов:

```
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
```

Затем это можно использовать со следующим кодом:

```
foo = get('http://test.com/admin', auth=CustomAuth('SecretHeader', 'CustomUserAgent', 'user',
'password' ))
```

Доверенные

Каждая операция POST запроса может быть настроена на использование сетевых прокси

Прокси 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)
```

HTTP Basic Authentication может быть предоставлена таким образом:

```
proxies = {'http': 'http://user:pass@192.168.0.128:312'}
foo = requests.post('http://httpbin.org/post', proxies=proxies)
```

Прокси SOCKS

Для использования прокси-серверов socks требуются `requests[socks]` сторонних зависимостей `requests[socks]` , когда установленные прокси-серверы используются очень похоже на HTTPBasicAuth:

```
proxies = {
'http': 'socks5://user:pass@host:port',
'https': 'socks5://user:pass@host:port'
}

foo = requests.post('http://httpbin.org/post', proxies=proxies)
```

Прочитайте Сообщение Python Requests онлайн: https://riptutorial.com/ru/python/topic/10021/сообщение-python-requests

глава 175: Сортировка списка (выбор частей списков)

Синтаксис

- a [начало: конец] # элемента начинается с конца-1
- [начало:] # элемента начинаются с остальной части массива
- a [: end] # элемента с начала до конца-1
- a [начало: конец: шаг] # начать через не прошлый конец, шаг
- a [:] # копия всего массива
- источник

замечания

- `lst[::-1]` дает вам обратную копию списка
- `start` или `end` может быть отрицательным числом, что означает, что он отсчитывается от конца массива, а не от начала. Так:

```
a[-1]    # last item in the array
a[-2:]   # last two items in the array
a[:-2]   # everything except the last two items
```

(источник)

Examples

Используя третий аргумент «шаг»

```
lst = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']

lst[::2]
# Output: ['a', 'c', 'e', 'g']

lst[::3]
# Output: ['a', 'd', 'g']
```

Выбор подсписок из списка

```
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']
```

Изменение списка с нарезкой

```
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]
```

Смена списка с помощью нарезки

```
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]
```

Прочитайте Сортировка списка (выбор частей списков) онлайн:
https://riptutorial.com/ru/python/topic/1494/сортировка-списка--выбор-частей-списков-

глава 176: Сортировка, минимальная и максимальная

Examples

Получение минимального или максимального значения нескольких значений

```
min(7,2,1,5)
# Output: 1

max(7,2,1,5)
# Output: 7
```

Использование ключевого аргумента

Возможно обнаружение минимума / максимума последовательности последовательностей:

```
list_of_tuples = [(0, 10), (1, 15), (2, 8)]
min(list_of_tuples)
# Output: (0, 10)
```

но если вы хотите сортировать по определенному элементу в каждой последовательности, используйте `key` -argument:

```
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)]
```

По умолчанию Аргумент max, мин.

Вы не можете передать пустую последовательность в `max` или `min` :

```
min([])
```

> ValueError: min () arg - пустая последовательность

Однако с Python 3 вы можете передать аргумент по `default` со значением, которое будет возвращено, если последовательность пуста, вместо того, чтобы создавать исключение:

```
max([], default=42)
# Output: 42
max([], default=0)
# Output: 0
```

Специальный случай: словари

Получение минимума или максимума или `sorted` зависит от итераций по объекту. В случае `dict` , итерация выполняется только по клавишам:

```
adict = {'a': 3, 'b': 5, 'c': 1}
min(adict)
# Output: 'a'
max(adict)
# Output: 'c'
sorted(adict)
# Output: ['a', 'b', 'c']
```

Чтобы сохранить структуру словаря, вам необходимо выполнить итерацию по `.items()` :

```
min(adict.items())
# Output: ('a', 3)
max(adict.items())
# Output: ('c', 1)
sorted(adict.items())
# Output: [('a', 3), ('b', 5), ('c', 1)]
```

Для `sorted` вы можете создать `OrderedDict` чтобы сохранить сортировку, имея структуру типа `dict` :

```
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
```

По значению

Опять же, это возможно с использованием `key` аргумента:

```
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)]
```

Получение упорядоченной последовательности

Используя **одну** последовательность:

```
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']
```

Результат - это всегда новый `list` ; исходные данные остаются неизменными.

Минимальная и максимальная последовательности

Получение минимума последовательности (итерабельности) эквивалентно доступу к первому элементу `sorted` последовательности:

```
min([2, 7, 5])
# Output: 2
sorted([2, 7, 5])[0]
# Output: 2
```

Максимум немного сложнее, потому что `sorted` сохраняет порядок, а `max` возвращает первое найденное значение. В случае отсутствия дубликатов максимальный размер совпадает с последним элементом отсортированного возврата:

```
max([2, 7, 5])
# Output: 7
sorted([2, 7, 5])[-1]
# Output: 7
```

Но нет, если есть несколько элементов, которые оцениваются как имеющие максимальное значение:

```
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
```

Разрешены любые итеративные элементы, поддерживающие операции `<` или `>` .

Сделать заказные классы упорядоченными

`min` , `max` и `sorted` все объекты должны быть упорядочены. Чтобы быть правильно упорядоченным, класс должен определить все 6 методов `__lt__` , `__gt__` , `__ge__` , `__le__` , `__ne__` и `__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
```

Хотя реализация всех этих методов будет казаться ненужной, опускание некоторых из них сделает ваш код подверженным ошибкам .

Примеры:

```
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` с помощью `reverse=True` также использует `__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)]
```

Но `sorted` может использовать `__gt__` вместо этого, если значение по умолчанию не реализовано:

```
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)
```

Методы сортировки повысят значение `TypeError` если не `__lt__` ни `__lt__` ни `__gt__` :

```
del IntegerContainer.__gt__    # The IntegerContainer no longer implements "greater then"

res = min(alist)
```

> ТипError: unorderable types: IntegerContainer () <IntegerContainer ()

`functools.total_ordering` decorator можно использовать, чтобы упростить работу над этими богатыми методами сравнения. Если вы украшаете свой класс `total_ordering` , вам нужно реализовать `__eq__` , `__ne__` и только один из `__lt__` , `__le__` , `__ge__` или `__gt__` , а декоратор заполнит остальные:

```
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
```

Обратите внимание, как теперь `>` (*больше*) вызывает вызов *меньше, чем* метод, а в некоторых случаях даже метод `__eq__` . Это также означает, что если скорость имеет большое значение, вы должны реализовать каждый богатый метод сравнения самостоятельно.

Извлечение N наибольших или N наименьших элементов из итерабельного

Для того, чтобы найти некоторое количество (более одного) из больших или мельчайших значений итератора, вы можете использовать `nlargest` и `nsmallest` из `heapq` модуля:

```
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]
```

Это намного эффективнее, чем сортировка всей итерации, а затем нарезка с конца или начала. Внутренне эти функции используют структуру данных очереди приоритетов двоичной кучи , которая очень эффективна для этого варианта использования.

Подобно `min` , `max` и `sorted` , эти функции принимают необязательный `key` аргумент `key` слова, который должен быть функцией, которая при задании элемента возвращает свой ключ сортировки.

Вот программа, которая извлекает 1000 длиннейших строк из файла:

```
import heapq
with open(filename) as f:
    longest_lines = heapq.nlargest(1000, f, key=len)
```

Здесь мы открываем файл и передаем дескриптор файла `f` в `nlargest` . Итерирование файла дает каждую строку файла как отдельную строку; `nlargest` затем передает каждый элемент (или строку) передается функции `len` чтобы определить его ключ сортировки. `len` , заданная строкой, возвращает длину строки в символах.

Это нужно только для хранения списка из 1000 крупнейших линий, что может быть противопоставлено

```
longest_lines = sorted(f, key=len)[1000:]
```

который должен содержать *весь файл в памяти* .

Прочитайте Сортировка, минимальная и максимальная онлайн: https://riptutorial.com/ru/python/topic/252/сортировка--минимальная-и-максимальная

глава 177: Специальная переменная __name__

Вступление

Специальная переменная `__name__` используется для проверки того, был ли файл импортирован как модуль или нет, а также для идентификации объекта функции, класса, модуля их атрибутом `__name__` .

замечания

Специальная переменная `__name__` Python устанавливается на имя содержащего модуля. На верхнем уровне (например, в интерактивном интерпретаторе или в основном файле) устанавливается значение `'__main__'` . Это можно использовать для запуска блока операторов, если модуль запускается напрямую, а не импортируется.

Связанный специальный атрибут `obj.__name__` находится в классах, импортированных модулях и функциях *(включая методы)* и дает имя объекта при его определении.

Examples

__name__ == '__main__'

Пользовательская переменная `__name__` не задается пользователем. Он в основном используется для проверки того, выполняется ли модуль самостоятельно или выполняется, потому что был выполнен `import` . Чтобы ваш модуль не запускал определенные части своего кода при импорте, проверьте `if __name__ == '__main__'` .

Пусть **module_1.py** - всего одна строка:

```
import module2.py
```

И посмотрим, что произойдет, в зависимости от **модуля2.py**

Ситуация 1

module2.py

```
print('hello')
```

Запуск **module1.py** будет печатать `hello`

Запуск **модуля2.py** будет печатать `hello`

Ситуация 2

module2.py

```
if __name__ == '__main__':
    print('hello')
```

Запуск **module1.py** ничего не напечатает
Запуск **модуля2.py** будет печатать `hello`

function_class_or_module .__ name__

Специальный атрибут `__name__` функции, класса или модуля - это строка, содержащая ее имя.

```
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
```

Атрибут `__name__` не является, однако, именем переменной, которая ссылается на класс, метод или функцию, скорее это имя, данное ему, когда определено.

```
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
```

Это можно использовать, среди прочего, для отладки:

```
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
```

Использование при регистрации

При настройке встроенных функций `logging` общий шаблон заключается в создании регистратора с `__name__` текущего модуля:

```
logger = logging.getLogger(__name__)
```

Это означает, что полное имя модуля появится в журналах, что упростит просмотр сообщений.

Прочитайте Специальная переменная __name__ онлайн: https://riptutorial.com/ru/python/topic/1223/специальная-переменная---name--

глава 178: Список

Вступление

Список Python - это общая структура данных, широко используемая в программах Python. Они находятся на других языках, часто называемых *динамическими массивами* . Они оба *изменяемы* и имеют тип данных *последовательности,* который позволяет их *индексировать* и *нарезать* . Список может содержать различные типы объектов, включая другие объекты списка.

Синтаксис

- [значение, значение, ...]
- список ([итерация])

замечания

`list` является конкретным типом итерации, но он не единственный, который существует в Python. Иногда лучше использовать `set` , `tuple` или `dictionary`

`list` - это имя, указанное в Python для динамических массивов (подобно `vector<void*>` из C ++ или Java `ArrayList<Object>`). Это не связанный список.

Доступ к элементам осуществляется в постоянное время и очень быстро. Добавление элементов в конец списка является амортизированным постоянным временем, но время от времени оно может включать в себя распределение и копирование всего `list` .

Перечисления списков относятся к спискам.

Examples

Доступ к значениям списка

Списки Python ноль-индексируются и действуют как массивы на других языках.

```
lst = [1, 2, 3, 4]
lst[0]  # 1
lst[1]  # 2
```

Попытка доступа к индексу за пределами списка приведет к созданию `IndexError` .

```
lst[4]  # IndexError: list index out of range
```

Отрицательные индексы интерпретируются как считанные с *конца* списка.

```
lst[-1]  # 4
lst[-2]  # 3
lst[-5]  # IndexError: list index out of range
```

Это функционально эквивалентно

```
lst[len(lst)-1]  # 4
```

Списки позволяют использовать *нотацию среза* как `lst[start:end:step]` . Вывод нотации среза - это новый список, содержащий элементы от `start` индекса до `end-1` . Если параметры опущены, `start` умолчанию начало списка, от `end` до конца списка и `step` 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
```

Имея это в виду, вы можете распечатать отмененную версию списка, позвонив

```
lst[::-1]    # [4, 3, 2, 1]
```

При использовании длины шагов отрицательных сумм начальный индекс должен быть больше, чем конечный индекс, иначе результат будет пустым.

```
lst[3:1:-1] # [4, 3]
```

Использование отрицательных индексов шагов эквивалентно следующему коду:

```
reversed(lst)[0:2] # 0 = 1 -1
                   # 2 = 3 -1
```

Используемые индексы на 1 меньше, чем те, которые используются при отрицательной индексации, и меняются на противоположные.

Расширенный нарезка

Когда списки `__getitem__()` метод `__getitem__()` объекта списка с объектом `slice` . Python имеет встроенный метод среза для создания объектов среза. Мы можем использовать это для *хранения* фрагмента и повторного использования позже,

```
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'
```

Это может быть очень `__getitem__` предоставляя функции среза для наших объектов, переопределяя `__getitem__` в нашем классе.

Методы списка и поддерживаемые операторы

Начиная с данного списка `a` :

```
a = [1, 2, 3, 4, 5]
```

1. `append(value)` - добавляет новый элемент в конец списка.

   ```
   # 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"]
   ```

   Обратите внимание, что метод `append()` добавляет только один новый элемент в конец списка. Если вы добавляете список в другой список, список, который вы добавляете, становится единственным элементом в конце первого списка.

   ```
   # 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)` - расширяет список, добавляя элементы из другого перечисляемого.

   ```
   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]
```

Списки также могут быть объединены с оператором `+` . Обратите внимание, что это не изменяет ни один из исходных списков:

```
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])` - получает индекс первого вхождения входного значения. Если входное значение отсутствует в списке, возникает исключение `ValueError` . Если предоставляется второй аргумент, поиск начинается с указанного индекса.

```
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)` - вставляет `value` непосредственно перед указанным `index` . Таким образом, после вставки новый элемент занимает `index` позиции.

```
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])` - удаляет и возвращает элемент по `index` . Без аргумента он удаляет и возвращает последний элемент списка.

```
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)` - удаляет первое вхождение указанного значения. Если предоставленное значение не может быть найдено, повышается значение `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()` - отменяет список на месте и возвращает `None` .

```
a.reverse()
# a: [8, 7, 6, 5, 4, 3, 2, 1]
```

Существуют также другие способы обращения вспять списка .

8. `count(value)` - подсчитывает количество вхождений некоторого значения в списке.

```
a.count(7)
# Returns: 2
```

9. `sort()` - сортирует список в числовом и лексикографическом порядке и возвращает `None` .

```
a.sort()
# a = [1, 2, 3, 4, 5, 6, 7, 8]
# Sorts the list in numerical order
```

Списки также могут быть отменены при сортировке с использованием флага `reverse=True` в методе `sort()` .

```
a.sort(reverse=True)
# a = [8, 7, 6, 5, 4, 3, 2, 1]
```

Если вы хотите сортировать по атрибутам элементов, вы можете использовать `key` аргумент `key` слова:

```
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]
```

В случае списка dicts понятие одно и то же:

```
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]
```

Сортировать по sub dict:

```
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]
```

Лучший способ сортировки с помощью `attrgetter` и `itemgetter`

Списки также могут быть отсортированы с помощью `attrgetter` и `itemgetter` из модуля оператора. Это может улучшить читаемость и повторное использование. Вот несколько примеров,

```
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` также может быть присвоен индекс. Это полезно, если вы хотите сортировать по

индексам кортежа.

```
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)]
```

Используйте `attrgetter` если вы хотите сортировать по атрибутам объекта,

```
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()` - удаляет все элементы из списка

```
a.clear()
# a = []
```

11. **Репликация** - умножение существующего списка на целое число приведет к созданию большего списка, состоящего из множества копий оригинала. Это может быть полезно, например, для инициализации списка:

```
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]
```

Позаботьтесь об этом, если ваш список содержит ссылки на объекты (например, список списков), см. « Общие ошибки» - умножение списка и общие ссылки .

12. **Удаление элемента** - можно удалить несколько элементов в списке, используя ключевое слово `del` и фрагмент:

```
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. **копирование**

Назначение по умолчанию «=» присваивает ссылку исходного списка на новое имя. То есть исходное имя и новое имя указывают на один и тот же объект списка. Изменения, внесенные через любой из них, будут отражены в другом. Это часто не

то, что вы намеревались.

```
b = a
a.append(6)
# b: [1, 2, 3, 4, 5, 6]
```

Если вы хотите создать копию списка, у вас есть варианты ниже.

Вы можете отрезать его:

```
new_list = old_list[:]
```

Вы можете использовать встроенную функцию list ():

```
new_list = list(old_list)
```

Вы можете использовать generic copy.copy ():

```
import copy
new_list = copy.copy(old_list) #inserts references to the objects found in the original.
```

Это немного медленнее, чем list (), потому что сначала нужно выяснить тип данных old_list.

Если список содержит объекты и вы хотите их скопировать, используйте общий copy.deepcopy ():

```
import copy
new_list = copy.deepcopy(old_list) #inserts copies of the objects found in the original.
```

Очевидно, самый медленный и самый необходимый для памяти способ, но иногда неизбежный.

Python 3.x 3.0

`copy()` - возвращает неполную копию списка

```
    aa = a.copy()
    # aa = [1, 2, 3, 4, 5]
```

Длина списка

Используйте `len()` чтобы получить одномерную длину списка.

```
len(['one', 'two'])  # returns 2

len(['one', [2, 3], 'four'])  # returns 3, not 4
```

`len()` также работает с строками, словарями и другими структурами данных, подобными спискам.

Обратите внимание: `len()` - это встроенная функция, а не метод объекта списка.

Также обратите внимание, что стоимость `len()` равна `O(1)` , то есть потребуется столько же времени, чтобы получить длину списка независимо от его длины.

Итерирование по списку

Python поддерживает использование цикла `for` непосредственно в списке:

```
my_list = ['foo', 'bar', 'baz']
for item in my_list:
    print(item)

# Output: foo
# Output: bar
# Output: baz
```

Вы также можете получить позицию каждого элемента одновременно:

```
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
```

Другой способ итерации списка на основе значения индекса:

```
for i in range(0,len(my_list)):
    print(my_list[i])
#output:
>>>
foo
bar
baz
```

Обратите внимание, что изменение элементов в списке при повторном запуске может иметь неожиданные результаты:

```
for item in my_list:
    if item == 'foo':
        del my_list[0]
    print(item)

# Output: foo
# Output: baz
```

В этом последнем примере мы удалили первый элемент на первой итерации, но это

привело к тому, что `bar` был пропущен.

Проверка наличия элемента в списке

Python упрощает проверку наличия элемента в списке. Просто используйте оператор `in` .

```
lst = ['test', 'twest', 'tweast', 'treast']

'test' in lst
# Out: True

'toast' in lst
# Out: False
```

> Примечание: оператор `in` на множестве асимптотически быстрее, чем в списках. Если вам нужно использовать его много раз в потенциально больших списках, вы можете захотеть преобразовать свой `list` в `set` и проверить наличие элементов в `set` .

```
slst = set(lst)
'test' in slst
# Out: True
```

Элементы реверсивного списка

Вы можете использовать `reversed` функцию, которая возвращает итератор в обратный список:

```
In [3]: rev = reversed(numbers)

In [4]: rev
Out[4]: [9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Обратите внимание, что список «числа» остается неизменным для этой операции и остается в том же порядке, что и был изначально.

Вы можете использовать `reverse` метод .

Вы также можете отменить список (на самом деле получить копию, исходный список не затронут), используя синтаксис разрезания, установив третий аргумент (шаг) как -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]
```

Проверка наличия пустого списка

Пустота списка связана с логическим `False` , поэтому вам не нужно проверять `len(lst) == 0` ,

но только `lst` или `not lst`

```
lst = []
if not lst:
    print("list is empty")

# Output: list is empty
```

Списки конкатенации и объединения

1. **Самый простой способ объединить** `list1` и `list2` :

   ```
   merged = list1 + list2
   ```

2. **`zip` возвращает список кортежей** , где i-й кортеж содержит i-й элемент из каждой из последовательностей аргументов или итераций:

   ```
   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
   ```

   Если списки имеют разную длину, тогда результат будет содержать только столько элементов, сколько кратчайший:

   ```
   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
   ```

   Для заполнения списков неравной длины до самой длинной с помощью `None` s используйте `itertools.zip_longest` (`itertools.izip_longest` в 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. **Вставка в определенные значения индекса:**

```
alist = [123, 'xyz', 'zara', 'abc']
alist.insert(3, [2009])
print("Final List :", alist)
```

Выход:

```
Final List : [123, 'xyz', 'zara', 2009, 'abc']
```

Каждый и все

Вы можете использовать `all()` чтобы определить, соответствуют ли все значения в итерабельной True

```
nums = [1, 1, 0, 1]
all(nums)
# False
chars = ['a', 'b', 'c', 'd']
all(chars)
# True
```

Аналогично, `any()` определяет, имеет ли одно или несколько значений в итерабельной значение True

```
nums = [1, 1, 0, 1]
any(nums)
# True
vals = [None, None, None, False]
any(vals)
# False
```

Хотя в этом примере используется список, важно отметить, что эти встроенные работы работают с любым итерабельным, включая генераторы.

```
vals = [1, 2, 3, 4]
any(val > 12 for val in vals)
# False
any((val * 2) > 6 for val in vals)
# True
```

Удалить повторяющиеся значения в списке

Удаление повторяющихся значений в списке может быть выполнено путем преобразования списка в `set` (это неупорядоченный набор отдельных объектов). Если структура данных `list` необходима, то набор можно преобразовать обратно в список, используя `list()` функций `list()` :

```
names = ["aixk", "duke", "edik", "tofp", "duke"]
list(set(names))
# Out: ['duke', 'tofp', 'aixk', 'edik']
```

Обратите внимание, что путем преобразования списка в набор исходное упорядочение теряется.

Для сохранения порядка списка можно использовать `OrderedDict`

```
import collections
>>> collections.OrderedDict.fromkeys(names).keys()
# Out: ['aixk', 'duke', 'edik', 'tofp']
```

Доступ к значениям во вложенном списке

Начиная с трехмерного списка:

```
alist = [[[1,2],[3,4]], [[5,6,7],[8,9,10], [12, 13, 14]]]
```

Доступ к элементам в списке:

```
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
```

Выполнение операций поддержки:

```
alist[0][0].append(11)
print(alist[0][0][2])
#11
#Appends 11 to the end of the first list in the first list
```

Использование вложенных циклов для печати списка:

```
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]
```

Обратите внимание, что эта операция может использоваться в понимании списка или даже в качестве генератора для повышения эффективности, например:

```
[col for row in alist for col in row]
#[[1, 2, 11], [3, 4], [5, 6, 7], [8, 9, 10], [12, 13, 14]]
```

Не все элементы во внешних списках должны быть самими списками:

```
alist[1].insert(2, 15)
#Inserts 15 into the third position in the second list
```

Другой способ использования вложенных циклов. Другой способ лучше, но я должен был использовать это иногда:

```
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]
```

Использование срезов во вложенном списке:

```
print(alist[1][1:])
#[[8, 9, 10], 15, [12, 13, 14]]
#Slices still work
```

Окончательный список:

```
print(alist)
#[[[1, 2, 11], [3, 4]], [[5, 6, 7], [8, 9, 10], 15, [12, 13, 14]]]
```

Сравнение списков

Можно сравнивать списки и другие последовательности с лексикографическим использованием операторов сравнения. Оба операнда должны быть одного типа.

```
[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
```

Если один из списков содержится в начале другого, выигрывает самый короткий список.

```
[1, 10] < [1, 10, 100]
# True
```

Инициализация списка для фиксированного количества элементов

Для **неизменяемых** элементов (например, `None` , строковые литералы и т. Д.):

```
my_list = [None] * 10
my_list = ['test'] * 10
```

Для **изменяемых** элементов **одна** и та же конструкция приведет к тому, что все элементы списка относятся к одному и тому же объекту, например, для набора:

```
>>> 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}]
```

Вместо этого, чтобы инициализировать список с фиксированным числом **различных изменяемых** объектов, используйте:

```
my_list=[{1} for _ in range(10)]
```

Прочитайте Список онлайн: https://riptutorial.com/ru/python/topic/209/список

глава 179: Список деструктурирования (ака упаковка и распаковка)

Examples

Назначение деструктуризации

В назначениях вы можете разделить Iterable на значения, используя синтаксис «распаковка»:

Разрушение как значения

```
a, b = (1, 2)
print(a)
# Prints: 1
print(b)
# Prints: 2
```

Если вы попытаетесь распаковать больше длины итерации, вы получите сообщение об ошибке:

```
a, b, c = [1]
# Raises: ValueError: not enough values to unpack (expected 3, got 1)
```

Python 3.x 3.0

Разрушение как список

Вы можете распаковать список неизвестной длины, используя следующий синтаксис:

```
head, *tail = [1, 2, 3, 4, 5]
```

Здесь мы извлекаем первое значение как скаляр, а остальные значения - в виде списка:

```
print(head)
# Prints: 1
print(tail)
# Prints: [2, 3, 4, 5]
```

Это эквивалентно:

```
l = [1, 2, 3, 4, 5]
head = l[0]
tail = l[1:]
```

Он также работает с несколькими элементами или элементами, составляющими конец списка:

```
a, b, *other, z = [1, 2, 3, 4, 5]
print(a, b, z, other)
# Prints: 1 2 5 [3, 4]
```

Игнорирование значений в назначениях деструктурирования

Если вас интересует только данное значение, вы можете использовать _ чтобы указать, что вас это не интересует. Примечание: это все равно будет установлено _ , большинство людей не используют его как переменную.

```
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

Игнорирование списков в назначениях деструктурирования

Наконец, вы можете игнорировать многие значения, используя синтаксис `*_` в присваивании:

```
a, *_ = [1, 2, 3, 4, 5]
print(a)
# Prints: 1
```

что не очень интересно, поскольку вместо этого вы можете использовать индексирование в списке. Там, где это приятно, нужно сохранить первые и последние значения в одном задании:

```
a, *_, b = [1, 2, 3, 4, 5]
print(a, b)
# Prints: 1 5
```

или извлечь сразу несколько значений:

```
a, _, b, _, c, *_ = [1, 2, 3, 4, 5, 6]
```

```
print(a, b, c)
# Prints: 1 3 5
```

Аргументы функции упаковки

В функциях вы можете определить ряд обязательных аргументов:

```
def fun1(arg1, arg2, arg3):
    return (arg1,arg2,arg3)
```

который сделает функцию вызываемой только тогда, когда будут указаны три аргумента:

```
fun1(1, 2, 3)
```

и вы можете определить аргументы как необязательные, используя значения по умолчанию:

```
def fun2(arg1='a', arg2='b', arg3='c'):
    return (arg1,arg2,arg3)
```

поэтому вы можете вызвать функцию различными способами, например:

```
fun2(1)              → (1,b,c)
fun2(1, 2)           → (1,2,c)
fun2(arg2=2, arg3=3) → (a,2,3)
...
```

Но вы также можете использовать синтаксис деструктурирования для *упаковки* аргументов, поэтому вы можете назначать переменные, используя `list` или `dict` .

Упаковка списка аргументов

У вас есть список значений

```
l = [1,2,3]
```

Вы можете вызвать функцию со списком значений в качестве аргумента, используя синтаксис `*` :

```
fun1(*l)
# Returns: (1,2,3)
fun1(*['w', 't', 'f'])
# Returns: ('w','t','f')
```

Но если вы не предоставляете список, длина которого соответствует количеству аргументов:

```
fun1(*['oops'])
# Raises: TypeError: fun1() missing 2 required positional arguments: 'arg2' and 'arg3'
```

Параметр ключевого слова для упаковки

Теперь вы можете также упаковать аргументы с помощью словаря. Вы можете использовать оператор `**` чтобы сообщить Python о распаковке `dict` качестве значений параметра:

```
d = {
  'arg1': 1,
  'arg2': 2,
  'arg3': 3
}
fun1(**d)
# Returns: (1, 2, 3)
```

когда функция имеет только позиционные аргументы (те, которые не имеют значений по умолчанию), вам необходимо, чтобы словарь содержал все ожидаемые параметры и не имел дополнительного параметра, или вы получите сообщение об ошибке:

```
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'
```

Для функций, которые имеют необязательные аргументы, вы можете скомпоновать аргументы в качестве словаря:

```
fun2(**d)
# Returns: (1, 2, 3)
```

Но там вы можете опустить значения, поскольку они будут заменены значениями по умолчанию:

```
fun2(**{'arg2': 2})
# Returns: ('a', 2, 'c')
```

И так же, как и раньше, вы не можете давать дополнительные значения, которые не являются существующими параметрами:

```
fun2(**{'arg1':1, 'arg2':2, 'arg3':3, 'arg4':4})
# Raises: TypeError: fun2() got an unexpected keyword argument 'arg4'
```

В реальном мире функции могут иметь как позиционные, так и необязательные аргументы, и они работают одинаково:

```
def fun3(arg1, arg2='b', arg3='c')
    return (arg1, arg2, arg3)
```

вы можете вызвать функцию только с помощью итерации:

```
fun3(*[1])
# Returns: (1, 'b', 'c')
fun3(*[1,2,3])
# Returns: (1, 2, 3)
```

или просто с помощью словаря:

```
fun3(**{'arg1':1})
# Returns: (1, 'b', 'c')
fun3(**{'arg1':1, 'arg2':2, 'arg3':3})
# Returns: (1, 2, 3)
```

или вы можете использовать оба метода в одном и том же вызове:

```
fun3(*[1,2], **{'arg3':3})
# Returns: (1,2,3)
```

Помните, что вы не можете предоставить несколько значений для одного и того же аргумента:

```
fun3(*[1,2], **{'arg2':42, 'arg3':3})
# Raises: TypeError: fun3() got multiple values for argument 'arg2'
```

Распаковка аргументов функции

Если вы хотите создать функцию, которая может принимать любое количество аргументов, а не использовать позицию или имя аргумента при компиляции, это возможно и вот как это сделать:

```
def fun1(*args, **kwargs):
    print(args, kwargs)
```

Параметры `*args` и `**kwargs` являются специальными параметрами, которые устанавливаются в `tuple` и `dict` соответственно:

```
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}
```

Если вы посмотрите на достаточно код Python, вы быстро обнаружите, что он широко используется при передаче аргументов другой функции. Например, если вы хотите

расширить класс строк:

```
class MyString(str):
    def __init__(self, *args, **kwarg):
        print('Constructing MyString')
        super(MyString, self).__init__(*args, **kwarg)
```

Прочитайте Список деструктурирования (ака упаковка и распаковка) онлайн: https://riptutorial.com/ru/python/topic/4282/список-деструктурирования--ака-упаковка-и-распаковка-

глава 180: Список рекомендаций

Вступление

Понимание списка является синтаксическим инструментом для создания списков естественным и сжатым способом, как показано в следующем коде, чтобы составить список квадратов чисел от 1 до 10: `[i ** 2 for i in range(1,11)]` Манекен `i` из существующего `range` списка используется для создания нового шаблона элемента. Он используется там, где цикл for необходим в менее выразительных языках.

Синтаксис

- [i для i в диапазоне (10)] # основное понимание списка
- [i for i in xrange (10)] # базовое понимание списка с объектом-генератором в python 2.x
- [i для i в диапазоне (20), если i% 2 == 0] # с фильтром
- [x + y для x в [1, 2, 3] для y в [3, 4, 5]] # вложенных циклах
- [i, если i> 6 else 0 для i в диапазоне (10)] # тройное выражение
- [i, если i> 4 else 0 для i в диапазоне (20), если i% 2 == 0] # с фильтром и тройным выражением
- [[x + y для x в [1, 2, 3]] для y в [3, 4, 5]] # понимание вложенного списка

замечания

Перечисления списков были описаны в PEP 202 и представлены в Python 2.0.

Examples

Условные списки

Учитывая список понимание вы можете добавить один или несколько , `if` условия для фильтрации значений.

```
[<expression> for <element> in <iterable> if <condition>]
```

Для каждого `<element>` в `<iterable>` ; если `<condition>` значение `True` , добавьте `<expression>` (обычно функцию `<element>`) в возвращаемый список.

Например, это можно использовать для извлечения только четных чисел из последовательности целых чисел:

```
[x for x in range(10) if x % 2 == 0]
# Out: [0, 2, 4, 6, 8]
```

Демо-версия

Вышеприведенный код эквивалентен:

```
even_numbers = []
for x in range(10):
    if x % 2 == 0:
        even_numbers.append(x)

print(even_numbers)
# Out: [0, 2, 4, 6, 8]
```

Кроме того, усвоение условного списка формы `[e for x in y if c]` (где `e` и `c` - выражения в терминах `x`) эквивалентно `list(filter(lambda x: c, map(lambda x: e, y)))` .

Несмотря на тот же результат, обратите внимание на то, что первый пример почти в 2 раза быстрее, чем последний. Для тех, кто любопытен, это хорошее объяснение причины.

Обратите внимание, что это сильно отличается от `... if ... else ...` условного выражения (иногда называемого тройным выражением), которое вы можете использовать для части `<expression>` list>. Рассмотрим следующий пример:

```
[x if x % 2 == 0 else None for x in range(10)]
# Out: [0, None, 2, None, 4, None, 6, None, 8, None]
```

Демо-версия

Здесь условное выражение не является фильтром, а скорее оператором, определяющим значение, которое должно использоваться для элементов списка:

```
<value-if-condition-is-true> if <condition> else <value-if-condition-is-false>
```

Это становится более очевидным, если вы объедините его с другими операторами:

```
[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]
```

Демо-версия

Если вы используете Python 2.7, `xrange` может быть лучше, чем `range` по нескольким причинам, как описано в документации `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]
```

Вышеприведенный код эквивалентен:

```
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]
```

Можно комбинировать тройные выражения и `if` условия. Термальный оператор работает с отфильтрованным результатом:

```
[x if x > 2 else '*' for x in range(10) if x % 2 == 0]
# Out: ['*', '*', 4, 6, 8]
```

То же самое не могло быть достигнуто только тройным оператором:

```
[x if (x > 2 and x % 2 == 0) else '*' for x in range(10)]
# Out:['*', '*', '*', '*', 4, '*', 6, '*', 8, '*']
```

См. Также: Фильтры , которые часто предоставляют достаточную альтернативу условным спискам.

Список рекомендаций с вложенными циклами

List Comprehensions может использовать вложенные `for` циклов. Вы можете закодировать любое количество вложенных циклов для внутри списка понимания, и каждый `for` цикла может иметь дополнительный связанный , `if` тест. При этом порядок следования `for` построений такой же порядок , как при написании серии вложенных `for` заявлений. Общая структура списков выглядит следующим образом:

```
[ expression for target1 in iterable1 [if condition1]
             for target2 in iterable2 [if condition2]...
             for targetN in iterableN [if conditionN] ]
```

Например, следующий код выравнивает список списков, используя несколько `for` операторов:

```
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]
```

может быть эквивалентно записано как понимание списка с несколькими `for` конструкций:

```
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]
```

Демо-версия

Как в развернутой форме, так и в понимании списка, первый цикл (первый для оператора) является первым.

В дополнение к тому, чтобы быть более компактным, вложенное понимание также значительно быстрее.

```
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
```

Накладные расходы для вызова функции выше составляют около *140 нс* .

Inline, `if` s вложен аналогично и может возникать в любой позиции после первого `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]
```

Демо-версия

Однако для удобства чтения вам следует использовать традиционные *for-loops* . Это особенно верно, когда вложенность более двух уровней глубокая, и / или логика понимания слишком сложна. множественное понимание списка вложенных циклов может быть подвержено ошибкам или дает неожиданный результат.

Фильтр рефакторинга и отображение списка для составления списка

Функции `filter` или `map` часто следует заменять списками . Гвидо Ван Россум описывает это

хорошо в открытом письме в 2005 году :

> `filter(P, S)` почти всегда записывается как `[x for x in S if P(x)]` , и это имеет огромное преимущество в том, что наиболее распространенные способы использования включают предикаты, которые являются сравнениями, например `x==42` , и определяют лямбда для этого требует гораздо больше усилий для читателя (плюс лямбда медленнее, чем понимание списка). Тем более, что для `map(F, S)` которое становится `[F(x) for x in S]` . Конечно, во многих случаях вы могли бы использовать выражения генератора.

Следующие строки кода считаются « *не pythonic* » и будут вызывать ошибки во многих python-линтерах.

```
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
```

Принимая то, что мы узнали из предыдущей цитаты, мы можем разбить эти выражения `filter` и `map` на их эквивалентные *списки* ; также удаляя *лямбда-* функции из каждого, делая код более читаемым в процессе.

```
# 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)]
```

Читаемость становится еще более очевидной при работе с функциями цепочки. В тех случаях, когда из-за удобочитаемости результаты одной карты или функции фильтра должны быть переданы в результате к следующему; с простыми случаями, их можно заменить на единое понимание списка. Кроме того, из понимания списка мы можем легко понять, каков результат нашего процесса, где есть большая познавательная нагрузка при рассуждении о цепочке процесса Map & Filter.

```
# 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]
```

Рефакторинг - краткая справочная информация

- **карта**

```
map(F, S) == [F(x) for x in S]
```

- **Фильтр**

```
filter(P, S) == [x for x in S if P(x)]
```

где `F` и `P` - функции, которые соответственно преобразуют входные значения и возвращают `bool`

Вложенные списки

Вложенные проверки списков, в отличие от контекстов списков с вложенными циклами, - это понимание List в понимании списка. Исходным выражением может быть любое произвольное выражение, включая другое понимание списка.

```
#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]]
```

Вложенный пример эквивалентен

```
l = []
for y in [3, 4, 5]:
    temp = []
    for x in [1, 2, 3]:
        temp.append(x + y)
    l.append(temp)
```

Один пример, когда вложенное понимание может быть использовано для переноса матрицы.

```
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]]
```

Как вложенные `for` циклов, нет ограничений на то, как глубокие понимания могут быть вложены.

```
[[[i + j + k for k in 'cd'] for j in 'ab'] for i in '12']
# Out: [[['1ac', '1ad'], ['1bc', '1bd']], [['2ac', '2ad'], ['2bc', '2bd']]]
```

Итерация двух или более списков одновременно в понимании списка

Для повторения более двух списков одновременно в *понимании списка* можно использовать `zip()` как:

```
>>> 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 ...
```

Прочитайте Список рекомендаций онлайн: https://riptutorial.com/ru/python/topic/5265/список-рекомендаций

глава 181: Сравнения

Синтаксис

- ! = - не равно
- == - Является равным
- `>` - больше, чем
- `<` - меньше, чем
- `>=` - больше или равно
- `<=` - меньше или равно
- `is` - test, если объекты являются одним и тем же объектом
- is not = test, если объекты не являются одним и тем же объектом

параметры

параметр	подробности
Икс	Первый элемент для сравнения
Y	Второй предмет для сравнения

Examples

Больше или меньше

```
x > y
x < y
```

Эти операторы сравнивают два типа значений, они меньше и чем операторы. Для чисел это просто сравнивает числовые значения, чтобы увидеть, что больше:

```
12 > 4
# True
12 < 4
# False
1 < 4
# True
```

Для строк они будут сравнивать лексикографически, что аналогично алфавитному порядку, но не совсем одинаково.

```
"alpha" < "beta"
# True
"gamma" > "beta"
# True
"gamma" < "OMEGA"
# False
```

В этих сравнениях строчные буквы считаются «большими» в верхнем регистре, поэтому `"gamma" < "OMEGA"` является ложной. Если бы они были заглавными, он бы возвращал ожидаемый результат в алфавитном порядке:

```
"GAMMA" < "OMEGA"
# True
```

Каждый тип определяет его вычисления с операторами `<` и `>` разному, поэтому вам следует исследовать, что означают операторы с заданным типом, прежде чем использовать его.

Не равен

```
x != y
```

Это возвращает значение `True` если `x` и `y` не равны и в противном случае возвращает значение `False` .

```
12 != 1
# True
12 != '12'
# True
'12' != '12'
# False
```

Равно

```
x == y
```

Это выражение оценивает, являются ли `x` и `y` одним и тем же значением и возвращает результат в виде логического значения. Как правило, оба типа и значения должны совпадать, поэтому int `12` не совпадает с строкой `'12'` .

```
12 == 12
# True
12 == 1
# False
'12' == '12'
# True
'spam' == 'spam'
```

```
# True
'spam' == 'spam '
# False
'12' == 12
# False
```

Обратите внимание, что каждый тип должен определять функцию, которая будет использоваться для оценки того, являются ли два значения одинаковыми. Для встроенных типов эти функции ведут себя так, как вы ожидали, и просто оцениваете вещи на основе того же значения. Однако пользовательские типы могут определять тестирование равенства как угодно, включая всегда возвращающее `True` или всегда возвращающее `False` .

Сравнение цепей

Вы можете сравнить несколько элементов с несколькими операторами сравнения с целым сравнением. Например

```
x > y > z
```

это всего лишь короткая форма:

```
x > y and y > z
```

Это будет оцениваться как `True` только если оба сравнения равны `True` .

Общий вид

```
a OP b OP c OP d ...
```

Где `OP` представляет собой одну из нескольких операций сравнения, которую вы можете использовать, а буквы представляют собой произвольные допустимые выражения.

> Заметим, что `0 != 1 != 0` значение `True` , хотя `0 != 0` - `False` . В отличие от общей математической нотации, в которой `x != y != z` означает, что `x` , `y` и `z` имеют разные значения. Операции Chaining `==` имеют естественное значение в большинстве случаев, так как равенство обычно является транзитивным.

Стиль

Теоретического ограничения на количество элементов и операций сравнения, которые вы используете, пока у вас есть правильный синтаксис:

```
1 > -1 < 2 > 0.5 < 100 != 24
```

Приведенное выше возвращает значение `True` если каждое сравнение возвращает `True` .

Однако использование свернутого цепочки не является хорошим стилем. Хорошая цепочка будет «направленной», не более сложной, чем

```
1 > x > -4 > y != 8
```

Побочные эффекты

Как только одно сравнение возвращает `False` , выражение немедленно оценивает значение `False` , пропуская все оставшиеся сравнения.

Заметим, что выражение `exp` в `a > exp > b` будет оцениваться только один раз, тогда как в случае

```
a > exp and exp > b
```

`exp` будет вычисляться дважды, если `a > exp` истинно.

Сравнение `is` vs` == `

Типичная ошибка является запутанными операторы сравнения равенства `is` и `==` .

`a == b` сравнивает значения `a` и `b` .

`a is b` будет сравнивать *тождества* `a` и `b` .

Проиллюстрировать:

```
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 [!!]
```

В принципе, `is` можно считать сокращением для `id(a) == id(b)` .

Помимо этого, существуют причуды среды выполнения, которые еще больше усложняют ситуацию. Короткие строки и маленькие целые числа возвращают `True` по сравнению с `is` , из-за того, что машина Python пытается использовать меньше памяти для идентичных объектов.

```
a = 'short'
b = 'short'
c = 5
d = 5
a is b # True
c is d # True
```

Но более длинные строки и большие целые числа будут храниться отдельно.

```
a = 'not so short'
b = 'not so short'
c = 1000
d = 1000
a is b # False
c is d # False
```

Вы должны использовать `is` , чтобы проверить на `None` :

```
if myvar is not None:
    # not None
    pass
if myvar is None:
    # None
    pass
```

Использование `is` заключается в проверке «дозорного» (т.е. уникального объекта).

```
sentinel = object()
def myfunc(var=sentinel):
    if var is sentinel:
        # value wasn’t provided
        pass
    else:
        # value was provided
        pass
```

Сравнение объектов

Чтобы сравнить равенство пользовательских классов, вы можете переопределить `==` и `!=` `__eq__` `__ne__` методов `__eq__` и `__ne__` . Вы также можете переопределить `__lt__` (`<`), `__le__` (`<=`), `__gt__` (`>`) и `__ge__` (`>`). Обратите внимание, что вам нужно только переопределить два метода сравнения, и Python может обрабатывать остальные (`==` то же самое, что и `not <` а `not >` и т. Д.).

```
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
```

Обратите внимание, что это простое сравнение предполагает, что `other` (объект, сравниваемый с) является одним и тем же типом объекта. По сравнению с другим типом будет выдана ошибка:

```
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'
```

Проверка `isinstance()` или аналогичного `isinstance()` поможет предотвратить это (при желании).

Common Gotcha: Python не применяет типизацию

Во многих других языках, если вы запускаете следующий (пример Java)

```
if("asgdsrf" == 0) {
    //do stuff
}
```

... вы получите сообщение об ошибке. Вы не можете просто сравнивать строки с целыми числами. В Python это совершенно законное утверждение - это просто разрешит `False` .

Обычным способом является следующее:

```
myVariable = "1"
if 1 == myVariable:
    #do stuff
```

Это сравнение будет оцениваться `False` без ошибки, каждый раз, потенциально скрывая ошибку или нарушая условное выражение.

Прочитайте Сравнения онлайн: https://riptutorial.com/ru/python/topic/248/сравнения

глава 182: стек

Вступление

Стек представляет собой контейнер объектов, которые вставляются и удаляются в соответствии с принципом последнего выхода (LIFO). В стеках сбрасывания разрешены только две операции: **нажмите элемент в стек и вытащите элемент из стека** . Стек представляет собой структуру данных с ограниченным доступом - **элементы могут быть добавлены и удалены из стека только сверху** . Вот структурное определение стека: стек либо пуст, либо состоит из вершины, а остальное - стека.

Синтаксис

- stack = [] # Создать стек
- stack.append (object) # Добавить объект в начало стека
- stack.pop () -> object # Возвращает самый верхний объект из стека, а также удаляет его
- list [-1] -> object # Заглянуть в самый верхний объект, не удаляя его

замечания

Из Википедии :

> В информатике *стек* представляет собой абстрактный тип данных, который служит в качестве набора элементов с двумя основными операциями: *push* , который добавляет элемент в коллекцию и *pop* , который удаляет последний добавленный элемент, который еще не был удален.

В связи с тем , как их элементы , доступ, стеки также известны как *Last-In, First-Out (LIFO) суммируется.*

В Python можно использовать списки как стеки с `append()` как push и `pop()` качестве поп-операций. Обе операции выполняются в постоянное время O (1).

Структура данных `deque` Python также может использоваться как стек. По сравнению с списками, `deque` s позволяет выполнять операции push и pop с постоянной сложностью по времени с обоих концов.

Examples

Создание класса Stack с помощью объекта списка

Используя объект `list` вы можете создать полностью функциональный общий стек со вспомогательными методами, такими как просмотр и проверка, если стек пуст. Ознакомьтесь с официальными документами python для использования `list` как `Stack` здесь .

```
#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
```

Пример:

```
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())
```

Выход:

```
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
```

Параллельные партисы

Стеки часто используются для синтаксического анализа. Простая задача синтаксического анализа состоит в проверке соответствия строк круглых скобок.

Например, строка `([])` соответствует, потому что внешние и внутренние скобки образуют пары. `()<>)` не соответствует, потому что последний `)` не имеет партнера. `([)]` также не соответствует, потому что пары должны быть либо полностью внутри, либо вне других пар.

```
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()
```

Прочитайте стек онлайн: https://riptutorial.com/ru/python/topic/3807/стек

глава 183: Стойкость к Python

Синтаксис

- pickle.dump (obj, file, protocol = None, *, fix_imports = True)
- pickle.load (файл, *, fix_imports = True, encoding = "ASCII", errors = "strict")

параметры

параметр	подробности
OBJ	маринованное представление obj в файл файла открытого файла
протокол	целое число, сообщает pickler использовать данный протокол, `0` -ASCII, `1` старый двоичный формат
файл	Аргумент файла должен иметь метод write () `wb` для метода *дампа* и для загрузки метода read () `rb`

Examples

Стойкость к Python

Объекты, такие как числа, списки, словари, вложенные структуры и объекты экземпляра класса, хранятся в памяти вашего компьютера и теряются, как только заканчивается скрипт.

pickle хранит данные настойчиво в отдельном файле.

маринованное представление объекта всегда является байтовым объектом во всех случаях, поэтому нужно открывать файлы в `wb` для хранения данных и `rb` для загрузки данных из рассола.

данные могут быть не такими, например,

```
data={'a':'some_value',
     'b':[9,4,7],
     'c':['some_str','another_str','spam','ham'],
     'd':{'key':'nested_dictionary'},
     }
```

Хранить данные

```
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
```

Загрузить данные

```
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']}
```

Следующие типы можно мариновать

1. Нет, True и False
2. целые числа, числа с плавающей запятой, комплексные числа
3. строки, байты, bytearrays
4. кортежи, списки, наборы и словари, содержащие только сортируемые объекты
5. функции, определенные на верхнем уровне модуля (с использованием def, а не лямбда)
6. встроенные функции, определенные на верхнем уровне модуля
7. классы, которые определены на верхнем уровне модуля
8. экземпляры таких классов, чей **dict** или результат вызова **getstate** ()

Функция утилиты для сохранения и загрузки

Сохранение данных в файл и из него

```
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'
```

Прочитайте Стойкость к Python онлайн: https://riptutorial.com/ru/python/topic/7810/стойкость-

к-python

глава 184: Строковые методы

Синтаксис

- str.capitalize () -> str
- str.casefold () -> str [только для Python> 3.3]
- str.center (width [, fillchar]) -> str
- str.count (sub [, start [, end]]) -> int
- str.decode (encoding = "utf-8" [, errors]) -> unicode [только в Python 2.x]
- str.encode (encoding = "utf-8", errors = "strict") -> bytes
- str.endswith (суффикс [, start [, end]]) -> 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
- static str.maketrans (x [, y [, z]])
- str.partition (sep) -> (head, sep, tail)
- 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) -> (head, sep, tail)
- str.rsplit (sep = None, maxsplit = -1) -> список строк
- str.rstrip ([chars]) -> str
- str.split (sep = None, maxsplit = -1) -> список строк
- str.splitlines ([keepends]) -> список строк
- str.startswith (prefix [, start [, end]]) -> book
- str.strip ([chars]) -> str

- str.swapcase () -> str
- str.title () -> str
- str.translate (table) -> str
- str.upper () -> str
- str.zfill (width) -> str

замечания

Строковые объекты неизменяемы, что означает, что они не могут быть изменены на месте, как может выглядеть список. Из-за этого методы встроенного типа `str` всегда возвращают **новый** объект `str` , который содержит результат вызова метода.

Examples

Изменение капитализации строки

Строковый тип Python предоставляет множество функций, которые действуют на капитализацию строки. Они включают :

- `str.casefold`
- `str.upper`
- `str.lower`
- `str.capitalize`
- `str.title`
- `str.swapcase`

С unicode-строками (по умолчанию в Python 3) эти операции **не** являются 1: 1 сопоставлениями или обратимыми. Большинство из этих операций предназначены для показа, а не для нормализации.

Python 3.x 3.3

`str.casefold()`

`str.casefold` создает строчную строку, подходящую для сравнения без `str.casefold` регистра. Это более агрессивно, чем `str.lower` и может изменять строки, которые уже находятся в нижнем регистре, или увеличивать длину строк и не предназначены для отображения.

```
"XßΣ".casefold()
# 'xssσ'

"XßΣ".lower()
# 'xßς'
```

Преобразования, происходящие в casefolding, определяются консорциумом Unicode в файле CaseFolding.txt на их веб-сайте.

`str.upper()`

`str.upper` берет каждый символ в строке и преобразует его в свой верхний регистр, например:

```
"This is a 'string'.".upper()
# "THIS IS A 'STRING'."
```

`str.lower()`

`str.lower` делает обратное; он принимает каждый символ в строке и преобразует его в его нижний регистр:

```
"This IS a 'string'.".lower()
# "this is a 'string'."
```

`str.capitalize()`

`str.capitalize` возвращает `str.capitalize` версию строки, т. е. делает первый символ верхним регистром, а остальные ниже:

```
"this Is A 'String'.".capitalize() # Capitalizes the first character and lowercases all others
# "This is a 'string'."
```

`str.title()`

`str.title` возвращает версию строки с заголовком, то есть каждая буква в начале слова выполнена в верхнем регистре, а все остальные - в нижнем регистре:

```
"this Is a 'String'".title()
# "This Is A 'String'"
```

`str.swapcase()`

`str.swapcase` возвращает новый строковый объект, в котором все символы нижнего регистра заменяются на верхний регистр, а все символы верхнего регистра - ниже:

```
"this iS A STRiNG".swapcase() #Swaps case of each character
# "THIS Is a strIng"
```

Использование методов класса `str`

Стоит отметить, что эти методы можно назвать либо строковыми объектами (как показано выше), либо как метод класса класса `str` (с явным вызовом `str.upper` и т. Д.),

```
str.upper("This is a 'string'")
# "THIS IS A 'STRING'"
```

Это наиболее полезно при применении одного из этих методов ко многим строкам сразу, скажем, к функции *`map`* .

```
map(str.upper,["These","are","some","'strings'"])
# ['THESE', 'ARE', 'SOME', "'STRINGS'"]
```

Разделить строку на основе разделителя на список строк

`str.split(sep=None, maxsplit=-1)`

`str.split` принимает строку и возвращает список подстрок исходной строки. Поведение отличается в зависимости от того, предоставлен или исключен аргумент `sep` .

Если `sep` не предоставлен или `None` , то расщепление происходит везде, где есть пробелы. Однако ведущее и завершающее пробелы игнорируются, а несколько последовательных символов пробелов обрабатываются так же, как один символ пробела:

```
>>> "This is a sentence.".split()
['This', 'is', 'a', 'sentence.']

>>> " This is    a sentence.  ".split()
['This', 'is', 'a', 'sentence.']

>>> "            ".split()
[]
```

Параметр `sep` может использоваться для определения строки разделителя. Исходная строка разделяется на строку разделителя, и сам разделитель отбрасывается. Несколько последовательных разделителей *не* обрабатываются так же, как одно вхождение, а скорее создают пустые строки.

```
>>> "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.']
```

По умолчанию используется разделение на *каждое* вхождение разделителя, однако параметр `maxsplit` ограничивает количество разбиений, которые происходят. Значение по

умолчанию `-1` означает лимит:

```
>>> "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») отличается от `str.split` («left split»), когда `maxsplit` . Разделение начинается в конце строки, а не в начале:

```
>>> "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', '.']
```

Примечание . Python указывает максимальное количество выполняемых *разбиений* , в то время как большинство других языков программирования определяют максимальное количество созданных *подстрок* . Это может создать путаницу при переносе или сравнении кода.

Замените все вхождения одной подстроки на другую подстроку

У `str` типа Python также есть метод для замещения вхождения одной подстроки другой подстрокой в заданной строке. Для более требовательных случаев можно использовать `re.sub` .

`str.replace(old, new[, count])` :

`str.replace` принимает два аргумента `old` и `new` содержащие `old` подстроку, которая должна быть заменена `new` подстрокой. Необязательный аргумент `count` указывает количество замен:

Например, чтобы заменить `'foo'` на `'spam'` в следующей строке, мы можем вызвать `str.replace` со `old = 'foo'` и `new = 'spam'` :

```
>>> "Make sure to foo your sentence.".replace('foo', 'spam')
"Make sure to spam your sentence."
```

Если данная строка содержит несколько примеров, которые соответствуют `old` аргументу, **все** вхождения заменяются значением, указанным в `new` :

```
>>> "It can foo multiple examples of foo if you want.".replace('foo', 'spam')
"It can spam multiple examples of spam if you want."
```

если, конечно, мы не даем значение для `count` . В этом случае `count` экземпляров собираются заменить:

```
>>> """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 и f-strings: форматирование значений в строку

Python предоставляет функции интерполяции и форматирования строк через функцию `str.format` , введенную в версии 2.6 и f-строки, введенные в версии 3.6.

Учитывая следующие переменные:

```
i = 10
f = 1.5
s = "foo"
l = ['a', 1, 2]
d = {'a': 1, 2: 'foo'}
```

Следующие утверждения эквивалентны

```
"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}"
```

Для справки, Python также поддерживает квалификаторы C-стиля для форматирования строк. Приведенные ниже примеры эквивалентны приведенным выше, но версии `str.format` являются предпочтительными из-за преимуществ гибкости, согласованности обозначений и расширяемости:

```
"%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)
```

Скобки, используемые для интерполяции в `str.format` также могут быть пронумерованы для уменьшения дублирования при форматировании строк. Например, следующие эквиваленты:

```
"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")
```

Хотя официальная документация на python, как обычно, достаточно тщательна, pyformat.info имеет большой набор примеров с подробными объяснениями.

Кроме того, символы `{` и `}` могут быть экранированы с помощью двойных скобок:

```
"{'a': 5, 'b': 6}"
```

```
>>> "{{'{}': {}, '{}': {}}}".format("a", 5, "b", 6)

>>> f"{{'{'a'}': {5}, '{'b'}': {6}}"
```

См. Форматирование строк для получения дополнительной информации. `str.format()` был предложен в PEP 3101 и f-строках в PEP 498 .

Подсчет количества раз, когда подстрока появляется в строке

Один метод доступен для подсчета количества вхождений подстроки в другой строке `str.count` .

`str.count(sub[, start[, end]])`

`str.count` возвращает `int` указывающий количество неперекрывающихся вхождений подстроки `sub` в другой строке. Необязательные аргументы `start` и `end` указывают начало и конец, в котором будет выполняться поиск. По умолчанию `start = 0` и `end = len(str)` означает, что вся строка будет искать:

```
>>> s = "She sells seashells by the seashore."
>>> s.count("sh")
2
>>> s.count("se")
3
>>> s.count("sea")
2
>>> s.count("seashells")
```

```
1
```

Указав другое значение для `start` , `end` мы можем получить более локализованный поиск и подсчет, например, если `start` равно `13` вызов:

```
>>> s.count("sea", start)
1
```

эквивалентно:

```
>>> t = s[start:]
>>> t.count("sea")
1
```

Проверьте начальные и конечные символы строки

Чтобы проверить начало и окончание заданной строки в Python, можно использовать методы `str.startswith()` и `str.endswith()` .

`str.startswith(prefix[, start[, end]])`

Как следует из названия, `str.startswith` используется для проверки того, начинается ли заданная строка с заданными символами в `prefix` .

```
>>> s = "This is a test string"
>>> s.startswith("T")
True
>>> s.startswith("Thi")
True
>>> s.startswith("thi")
False
```

Необязательные аргументы `start` и `end` указывают начальную и конечную точки, с которых начнется и закончится тестирование. В следующем примере, указав начальное значение `2` наша строка будет искать с позиции `2` и затем:

```
>>> s.startswith("is", 2)
True
```

Это дает `True` поскольку `s[2] == 'i'` и `s[3] == 's'` .

Вы также можете использовать `tuple` чтобы проверить, начинается ли он с любого из набора строк

```
>>> s.startswith(('This', 'That'))
True
>>> s.startswith(('ab', 'bc'))
False
```

---

str.endswith(prefix[, start[, end]])

`str.endswith` точно подобен `str.startswith` с той лишь разницей, что он ищет конечные символы, а не начальные символы. Например, чтобы проверить, заканчивается ли строка с полной остановкой, можно написать:

```
>>> s = "this ends in a full stop."
>>> s.endswith('.')
True
>>> s.endswith('!')
False
```

так как при `startswith` в качестве конечной последовательности может использоваться более одного символа:

```
>>> s.endswith('stop.')
True
>>> s.endswith('Stop.')
False
```

Вы также можете использовать `tuple` чтобы проверить, заканчивается ли он любым набором строк

```
>>> s.endswith(('.', 'something'))
True
>>> s.endswith(('ab', 'bc'))
False
```

Тестирование строки, состоящей из

Тип `str` Python также имеет ряд методов, которые можно использовать для оценки содержимого строки. Это `str.isalpha` , `str.isdigit` , `str.isalnum` , `str.isspace` . Капитализацию можно протестировать с помощью `str.isupper` , `str.islower` и `str.istitle` .

---

str.isalpha

`str.isalpha` принимает аргументов и возвращает `True` если все символы в данной строке являются алфавитными, например:

```
>>> "Hello World".isalpha()  # contains a space
False
>>> "Hello2World".isalpha()  # contains a number
False
>>> "HelloWorld!".isalpha()  # contains punctuation
False
>>> "HelloWorld".isalpha()
True
```

В качестве краевого случая пустая строка оценивается как `False` при использовании с

`"".isalpha()` .

`str.isupper` **,** **`str.islower`** **,** **`str.istitle`**

Эти методы проверяют капитализацию в заданной строке.

`str.isupper` - это метод, возвращающий `True` если все символы в заданной строке имеют верхний регистр и `False` противном случае.

```
>>> "HeLLO WORLD".isupper()
False
>>> "HELLO WORLD".isupper()
True
>>> "".isupper()
False
```

И наоборот, `str.islower` - это метод, который возвращает `True` если все символы в заданной строке имеют строчные буквы и `False` противном случае.

```
>>> "Hello world".islower()
False
>>> "hello world".islower()
True
>>> "".islower()
False
```

`str.istitle` возвращает значение `True` если заданная строка имеет название; то есть каждое слово начинается с символа верхнего регистра, за которым следуют строчные буквы.

```
>>> "hello world".istitle()
False
>>> "Hello world".istitle()
False
>>> "Hello World".istitle()
True
>>> "".istitle()
False
```

`str.isdecimal` **,** **`str.isdigit`** **,** **`str.isnumeric`**

`str.isdecimal` возвращает, является ли строка последовательностью десятичных цифр, подходящей для представления десятичного числа.

`str.isdigit` содержит цифры не в форме, подходящей для представления десятичного числа, например надстрочных цифр.

`str.isnumeric` включает любые значения чисел, даже если не цифры, такие как значения

вне диапазона 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` в Python 3, `str` в Python 2), поддерживает только `isdigit` , которая проверяет только основные ASCII-цифры.

Как и в случае `str.isalpha` , пустая строка оценивается как `False` .

`str.isalnum`

Это комбинация `str.isalpha` и `str.isnumeric` , в частности, она принимает значение `True` если все символы в данной строке являются **буквенно-цифровыми** , то есть они состоят из буквенных *или* числовых символов:

```
>>> "Hello2World".isalnum()
True
>>> "HelloWorld".isalnum()
True
>>> "2016".isalnum()
True
>>> "Hello World".isalnum()  # contains whitespace
False
```

`str.isspace`

Вычисляет значение `True` если строка содержит только пробельные символы.

```
>>> "\t\r\n".isspace()
True
>>> " ".isspace()
True
```

Иногда строка выглядит «пустой», но мы не знаем, связано ли это с тем, что она содержит просто пробельные символы или вообще не имеет характера

```
>>> "".isspace()
False
```

Чтобы покрыть этот случай, нам нужен дополнительный тест

```
>>> my_str = ''
>>> my_str.isspace()
```

```
False
>>> my_str.isspace() or not my_str
True
```

Но самый короткий способ проверить, является ли строка пустой или просто содержит пробельные символы, - это использовать `strip` (без аргументов она удаляет все ведущие и завершающие пробельные символы)

```
>>> not my_str.strip()
True
```

str.translate: перевод символов в строку

Python поддерживает метод `translate` по типу `str` который позволяет вам указать таблицу переводов (используется для замены), а также любые символы, которые необходимо удалить в процессе.

```
str.translate(table[, deletechars])
```

параметр	Описание
`table`	Это таблица поиска, которая определяет отображение от одного символа к другому.
`deletechars`	Список символов, которые нужно удалить из строки.

Метод `maketrans` (`str.maketrans` в Python 3 и `string.maketrans` в Python 2) позволяет сгенерировать таблицу переводов.

```
>>> translation_table = str.maketrans("aeiou", "12345")
>>> my_string = "This is a string!"
>>> translated = my_string.translate(translation_table)
'Th3s 3s 1 str3ng!'
```

Метод `translate` возвращает строку, которая представляет собой переведенную копию исходной строки.

Вы можете установить аргумент `table` `None` если вам нужно только удалить символы.

```
>>> 'this syntax is very useful'.translate(None, 'aeiou')
'ths syntx s vry sfl'
```

Удаление ненужных ведущих / завершающих символов из строки

`str.strip` три метода, которые позволяют `str.strip` ведущие и завершающие символы из

строки: `str.strip` , `str.rstrip` и `str.lstrip` . Все три метода имеют одну и ту же подпись, и все три возвращают новый строковый объект с удалением ненужных символов.

`str.strip([chars])`

`str.strip` действует на заданной строки и удаляет (полоски) или каких - либо ведущих задних символов , содержащихся в аргументе `chars` ; если `chars` не указаны или `None` , все символы пробела удаляются по умолчанию. Например:

```
>>> "    a line with leading and trailing space     ".strip()
'a line with leading and trailing space'
```

Если заданы `chars` , все символы, содержащиеся в нем, удаляются из строки, которая возвращается. Например:

```
>>> ">>> a Python prompt".strip('> ')  # strips '>' character and space character
'a Python prompt'
```

`str.rstrip([chars])` и `str.lstrip([chars])`

Эти методы имеют сходную семантику и аргументы с `str.strip()` , их различие лежит в том направлении, с которого они начинаются. `str.rstrip()` начинается с конца строки, а `str.lstrip()` разбивается с начала строки.

Например, используя `str.rstrip` :

```
>>> "     spacious string      ".rstrip()
'     spacious string'
```

Хотя, используя `str.lstrip` :

```
>>> "     spacious string      ".rstrip()
'spacious string      '
```

Сравнительные строки, нечувствительные к регистру

Сравнение строки в нечувствительном к регистру образом кажется чем-то тривиальным, но это не так. В этом разделе рассматриваются только строки unicode (по умолчанию в Python 3). Обратите внимание, что Python 2 может иметь небольшие недостатки относительно Python 3 - более поздняя обработка юникода намного более полная.

Первое, что следует отметить, - это то, что конверсии case-remove в unicode не являются тривиальными. Существует текст, для которого `text.lower() != text.upper().lower()` , например `"ß"` :

```
>>> "ß".lower()
'ß'

>>> "ß".upper().lower()
'ss'
```

Но скажем, вы хотели без "BUSSE" сравнений "BUSSE" и "Buße" . Черт, вы, вероятно, также хотите сравнить "BUSSE" и "BU�E" равными - это более новая форма капитала. Рекомендуемым способом является использование 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.
"""
```

Не используйте только lower . Если casefold недоступен, выполнение .upper().lower() помогает (но только несколько).

Тогда вы должны рассмотреть акценты. Если ваш рендерер шрифта хорош, вы, вероятно, думаете "ê" == "ê " - но это не так:

```
>>> "ê" == "ê "
False
```

Это потому, что они на самом деле

```
>>> 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']
```

Самый простой способ справиться с этим - unicodedata.normalize . Вероятно, вы хотите использовать нормализацию **NFKD** , но не стесняйтесь проверить документацию. Тогда

```
>>> unicodedata.normalize("NFKD", "ê") == unicodedata.normalize("NFKD", "ê ")
True
```

Чтобы закончить, здесь это выражается в функциях:

```
import unicodedata
```

```
def normalize_caseless(text):
    return unicodedata.normalize("NFKD", text.casefold())

def caseless_equal(left, right):
    return normalize_caseless(left) == normalize_caseless(right)
```

Присоедините список строк в одну строку

Строку можно использовать в качестве разделителя для объединения списка строк в одну строку с использованием метода `join()` . Например, вы можете создать строку, в которой каждый элемент в списке разделяется пробелом.

```
>>> " ".join(["once","upon","a","time"])
"once upon a time"
```

В следующем примере разделяются строковые элементы с тремя дефисами.

```
>>> "---".join(["once", "upon", "a", "time"])
"once---upon---a---time"
```

Полезные константы модуля String

`string` модуль Python предоставляет константы для операций, связанных с строкой. Чтобы использовать их, импортируйте `string` модуль:

```
>>> import string
```

`string.ascii_letters` :

Конкатенация `ascii_lowercase` и `ascii_uppercase` :

```
>>> string.ascii_letters
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

`string.ascii_lowercase` :

Содержит все символы ASCII нижнего регистра:

```
>>> string.ascii_lowercase
'abcdefghijklmnopqrstuvwxyz'
```

`string.ascii_uppercase` :

Содержит все символы ASCII верхнего регистра:

```
>>> string.ascii_uppercase
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

`string.digits` :

Содержит все десятичные знаковые символы:

```
>>> string.digits
'0123456789'
```

`string.hexdigits` :

Содержит все символы шестнадцатеричной цифры:

```
>>> string.hexdigits
'0123456789abcdefABCDEF'
```

`string.octaldigits` :

Содержит все восьмеричные цифры:

```
>>> string.octaldigits
'01234567'
```

`string.punctuation` :

Содержит все символы, которые считаются пунктуацией в локали `C` :

```
>>> string.punctuation
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

`string.whitespace` :

Содержит все символы ASCII, считанные пробелами:

```
>>> string.whitespace
' \t\n\r\x0b\x0c'
```

В режиме сценария `print(string.whitespace)` будет печатать фактические символы, используйте `str` для получения строки, возвращенной выше.

`string.printable` :

Содержит все символы, которые считаются пригодными для печати; комбинация `string.digits` , `string.ascii_letters` , `string.punctuation` и `string.whitespace` .

```
>>> string.printable
'0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&\'()*+,-
./:;<=>?@[\\]^_`{|}~ \t\n\r\x0b\x0c'
```

Обращение строки

Строка может быть отменена с помощью встроенной функции reverse `reversed()` , которая берет строку и возвращает итератор в обратном порядке.

```
>>> reversed('hello')
<reversed object at 0x0000000000000000>
>>> [char for char in reversed('hello')]
['o', 'l', 'l', 'e', 'h']
```

`reversed()` может быть завернута в вызов `''.join()` чтобы сделать строку из итератора.

```
>>> ''.join(reversed('hello'))
'olleh'
```

Хотя использование `reversed()` может быть более читаемым для непосвященных пользователей Python, использование расширенного среза с шагом `-1` выполняется быстрее и более кратким. Здесь попробуйте реализовать его как функцию:

```
>>> def reversed_string(main_string):
...     return main_string[::-1]
...
>>> reversed_string('hello')
'olleh'
```

Обозначить строки

Python предоставляет функции для оправдания строк, позволяя добавлять текст для упрощения выравнивания различных строк.

Ниже приведен пример `str.ljust` и `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` и `rjust` очень похожи. Оба имеют параметр `width` и необязательный параметр `fillchar` . Любая строка, созданная этими функциями, по крайней мере до тех пор, пока параметр `width` который был передан в функцию. Если строка длиннее `width` , она не усекается. Аргумент `fillchar` , который по умолчанию имеет символ пробела `' '` должен быть одним символом, а не многострочной.

`ljust` функция подушечки конца строки она называется на с `fillchar` до тех пор, пока `width` длиной символов. Функция `rjust` устанавливает начало строки аналогичным образом. Следовательно, `l` и `r` в именах этих функций относятся к стороне, в которой исходная строка, а *не* *`fillchar`* , *`fillchar`* в выходную строку.

Преобразование между данными str или байтов и символами Unicode

Содержимое файлов и сетевых сообщений может представлять кодированные символы. Их часто нужно преобразовать в unicode для правильного отображения.

В Python 2 вам может потребоваться преобразовать данные str в символы Unicode. Значение по умолчанию (`''` , `""` и т. Д.) Является строкой ASCII с любыми значениями вне диапазона ASCII, отображаемыми как экранированные значения. Строки Unicode - `u''` (или `u""` и т. Д.).

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
```

В Python 3 вам может понадобиться преобразовать массивы байтов (называемые «байтовым литералом») в строки символов Unicode. По умолчанию используется строка

Юникода, а литералы bytestring теперь должны быть введены как `b''` , `b""` и т. `isinstance(some_val, byte)` вернет `True` в `isinstance(some_val, byte)` , предполагая, что `some_val` будет строкой, которая может быть закодирована как байты.

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.
```

Содержит строку

Python чрезвычайно интуитивно проверяет, содержит ли строка заданную подстроку. Просто используйте оператор `in` :

```
>>> "foo" in "foo.baz.bar"
True
```

Примечание: тестирование пустой строки всегда приведет к `True` :

```
>>> "" in "test"
True
```

Прочитайте Строковые методы онлайн: https://riptutorial.com/ru/python/topic/278/строковые-методы

глава 185: Строковые представления экземпляров класса: методы __str__ и __ repr__

замечания

Заметка об использовании обоих методов

Когда оба метода реализованы, довольно часто возникает метод `__str__` который возвращает удобное для человека представление (например, «Ace of Spaces»), а `__repr__` возвращает представление, `__repr__` на `eval` .

Фактически, документы Python для `repr()` отмечают именно это:

> Для многих типов эта функция пытается вернуть строку, которая даст объект с тем же значением при передаче в eval (), иначе представление представляет собой строку, заключенную в угловые скобки, которая содержит имя типа объекта вместе с дополнительной информацией, которая часто включает имя и адрес объекта.

Это означает, что `__str__` может быть реализован, чтобы вернуть что-то вроде «Ace of Spaces», как показано ранее, `__repr__` может быть реализовано вместо возврата `Card('Spades', 1)`

Эта строка может быть передана непосредственно в `eval` в виде «кругового путешествия»:

```
object -> string -> object
```

Примером реализации такого метода может быть:

```
def __repr__(self):
    return "Card(%s, %d)" % (self.suit, self.pips)
```

Заметки

[1] Этот вывод специфичен для реализации. Строка отображается из cpython.

[2] Возможно, вы уже видели результат этого разделения `str()` / `repr()` и не знали его. Когда строки, содержащие специальные символы, такие как обратные косые черты, преобразуются в строки через `str()` обратные косые черты появляются как есть (они

появляются один раз). Когда они преобразуются в строки с помощью функции `repr()` (например, в качестве элементов отображаемого списка), обратные косы сбрасываются и, таким образом, отображаются дважды.

Examples

мотивация

Таким образом, вы только что создали свой первый класс в Python, аккуратном небольшом классе, который инкапсулирует игровую карту:

```
class Card:
    def __init__(self, suit, pips):
        self.suit = suit
        self.pips = pips
```

В другом месте вашего кода вы создаете несколько экземпляров этого класса:

```
ace_of_spades = Card('Spades', 1)
four_of_clubs = Card('Clubs',  4)
six_of_hearts = Card('Hearts', 6)
```

Вы даже создали список карточек, чтобы представить «руку»:

```
my_hand = [ace_of_spades, four_of_clubs, six_of_hearts]
```

Теперь, во время отладки, вы хотите увидеть, как выглядит ваша рука, поэтому вы делаете то, что приходит естественно и пишите:

```
print(my_hand)
```

Но то, что вы возвращаете, - это куча тарабарщин:

```
[<__main__.Card instance at 0x0000000002533788>,
 <__main__.Card instance at 0x00000000025B95C8>,
 <__main__.Card instance at 0x00000000025FF508>]
```

Смущенный, вы пытаетесь просто распечатать одну карту:

```
print(ace_of_spades)
```

И снова вы получаете этот странный результат:

```
<__main__.Card instance at 0x0000000002533788>
```

Не бойся. Мы собираемся это исправить.

Во-первых, однако, важно понять, что здесь происходит. Когда вы написали `print(ace_of_spades)` вы сказали Python, что хотите распечатать информацию о экземпляре `Card` ваш код вызывает `ace_of_spades` . И, честно говоря, так оно и было.

Этот вывод состоит из двух важных бит: `type` объекта и `id` объекта. Только вторая часть (шестнадцатеричное число) достаточно, чтобы однозначно идентифицировать объект во время вызова `print` . [1]

Что на самом деле произошло, так это то, что вы попросили Python «положить в слова» суть этого объекта, а затем отобразить его для вас. Более явная версия того же механизма может быть:

```
string_of_card = str(ace_of_spades)
print(string_of_card)
```

В первой строке вы пытаетесь превратить экземпляр своей `Card` в строку, а во втором - ее.

Эта проблема

Проблема, с которой вы сталкиваетесь, возникает из-за того, что, хотя вы сказали Python обо всем, что вам нужно знать о классе `Card` для *создания* карт, вы *не* сказали ему, как вы хотели, чтобы экземпляры `Card` были преобразованы в строки.

И поскольку он не знал, когда вы (неявно) написали `str(ace_of_spades)` , он дал вам то, что вы видели, общее представление экземпляра `Card` .

Решение (часть 1)

Но *мы можем* сказать Python, как мы хотим, чтобы экземпляры наших пользовательских классов были преобразованы в строки. И способ, которым мы это делаем, - это `__str__` «dunder» (для двойного подчеркивания) или «волшебный» метод.

Всякий раз, когда вы указываете Python на создание строки из экземпляра класса, он будет искать метод `__str__` в классе и вызывать его.

Рассмотрим следующую обновленную версию нашего класса `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)
```

Здесь мы теперь определили метод `__str__` на нашем классе `Card` который после простого поиска словаря для лицевых карточек **возвращает** строку, форматированную, однако мы решаем.

(Обратите внимание, что здесь `str(ace_of_spades)` , чтобы подчеркнуть важность возврата строки, а не просто ее печать. Печать может показаться `str(ace_of_spades)` , но тогда вы будете распечатывать карту, когда вы сделали что-то вроде `str(ace_of_spades)` , даже не имея вызова функции печати в вашей основной программе. Поэтому, чтобы быть понятным, убедитесь, что `__str__` возвращает строку.).

Метод `__str__` - это метод, поэтому первый аргумент будет `self` и он не должен принимать и не принимать дополнительные аргументы.

Возвращаясь к нашей проблеме отображения карты более удобным образом, если мы снова запустим:

```
ace_of_spades = Card('Spades', 1)
print(ace_of_spades)
```

Мы увидим, что наш результат намного лучше:

```
Ace of Spades
```

Настолько замечательно, что мы закончили, да?

Ну, просто чтобы покрыть наши базы, давайте дважды проверем, что мы решили первую проблему, с которой мы столкнулись, распечатав список экземпляров `Card` , `hand` .

Поэтому мы перепроверем следующий код:

```
my_hand = [ace_of_spades, four_of_clubs, six_of_hearts]
print(my_hand)
```

И, к нашему удивлению, мы снова получаем эти смешные шестнадцатеричные коды:

```
[<__main__.Card instance at 0x00000000026F95C8>,
 <__main__.Card instance at 0x000000000273F4C8>,
 <__main__.Card instance at 0x0000000002732E08>]
```

В чем дело? Мы сказали Python, что мы хотим, чтобы наши экземпляры `Card` отображались, почему они, похоже, не забыли?

Решение (часть 2)

Ну, закулисный механизм немного отличается, когда Python хочет получить строковое представление элементов в списке. Оказывается, Python не заботится о `__str__` для этой цели.

Вместо этого, он ищет другой метод, `__repr__` , и если *это* не найдено, он возвращается на «шестнадцатеричном вещь». [2]

Значит, вы говорите, что я должен сделать два метода, чтобы сделать то же самое? Один из них, когда я хочу `print` свою карточку сам по себе, а другой, когда он находится в каком-то контейнере?

Нет, но сначала давайте посмотрим, каким будет наш класс , если бы мы реализовали `__str__` и `__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)
```

Здесь реализация двух методов `__str__` и `__repr__` точно такая же, за исключением того, что для различения двух методов `(S)` добавляется к строкам, возвращаемым `__str__` и `(R)` добавляется к строкам, возвращаемым `__repr__` .

Обратите внимание, что как и наш метод `__str__` , `__repr__` принимает аргументов и **возвращает** строку.

Теперь мы можем видеть, какой метод отвечает за каждый случай:

```
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)
```

Как было `__str__` метод `__str__` вызывался, когда мы передавали экземпляр нашей `Card` для `print` и `__repr__` метод `__repr__` когда мы передавали *список наших экземпляров* для `print` .

На этом этапе стоит отметить, что так же, как мы можем явно создать строку из экземпляра пользовательского класса, используя `str()` как это было ранее, мы также можем явно создать **строковое представление** нашего класса со встроенной функцией, называемой `repr()` .

Например:

```
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)
```

И дополнительно, если они определены, мы *могли бы* напрямую вызвать методы (хотя это кажется немного неясным и ненужным):

```
print(four_of_clubs.__str__())     # 4 of Clubs (S)

print(four_of_clubs.__repr__())    # 4 of Clubs (R)
```

Об этих дублированных функциях ...

Разработчики Python поняли, что в случае, если вы хотите, чтобы идентичные строки были возвращены из `str()` и `repr()` вам, возможно, придется использовать функционально-дублирующие методы - что-то не нравится.

Поэтому вместо этого существует механизм для устранения необходимости в этом. Один из них я протащил тебя до этого момента. Оказывается, если класс реализует метод `__repr__` *но не* метод `__str__` , и вы передаете экземпляр этого класса в `str()` (неявно или явно), Python будет `__repr__` от вашей реализации `__repr__` и использовать это.

Итак, чтобы быть понятным, рассмотрите следующую версию класса `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)
```

Обратите внимание , эта версия *только* реализует `__repr__` метод. Тем не менее, вызовы

`str()` приводят к удобной версии:

```
print(six_of_hearts)            # 6 of Hearts  (implicit conversion)
print(str(six_of_hearts))       # 6 of Hearts  (explicit conversion)
```

как и вызовы `repr()` :

```
print([six_of_hearts])          #[6 of Hearts] (implicit conversion)
print(repr(six_of_hearts))      # 6 of Hearts  (explicit conversion)
```

Резюме

Чтобы вы могли расширять возможности своих экземпляров класса, чтобы «показать себя» в удобных для пользователя целях, вам нужно рассмотреть возможность внедрения, по крайней мере, метода `__repr__` вашего класса. Если память используется, во время разговора Раймонд Хеттингер сказал, что обеспечение классов реализует `__repr__` - это одна из первых вещей, которую он ищет при выполнении обзоров кода Python, и теперь должно быть понятно, почему. Объем информации, которую вы *могли* бы добавить к отладкам, отчетам о сбоях или файлам журнала с помощью простого метода, является огромным по сравнению с ничтожной и часто менее полезной (тип, идентификатор) информацией, которая предоставляется по умолчанию.

Если вам нужны *разные* представления для того, когда, например, внутри контейнера, вы захотите реализовать методы `__repr__` и `__str__` . (Подробнее о том, как вы можете использовать эти два метода по-разному ниже).

Оба метода реализованы, стиль eval-round-trip __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)
```

Прочитайте Строковые представления экземпляров класса: методы __str__ и __repr__ онлайн: https://riptutorial.com/ru/python/topic/4845/строковые-представления-экземпляров-класса--методы---str---и---repr--

глава 186: Сходства в синтаксисе, Различия в значении: Python и JavaScript

Вступление

Иногда бывает, что два языка имеют разные значения в одном и том же или подобном синтаксическом выражении. Когда оба языка представляют интерес для программиста, уточнение этих точек бифуркации помогает лучше понять оба языка в их основах и тонкостях.

Examples

`in` со списками

```
2 in [2, 3]
```

В Python это оценивается как True, но в JavaScript - false. Это происходит потому, что в Python в проверках, если значение содержится в списке, поэтому 2 находится в [2, 3] в качестве первого элемента. В JavaScript в используется с объектами и проверяется, содержит ли объект свойство с именем, выраженным значением. Таким образом, JavaScript рассматривает [2, 3] как объект или карту ключа-значения следующим образом:

```
{'0': 2, '1': 3}
```

и проверяет, есть ли у него свойство или ключ «2». Целое число 2 тихо преобразуется в строку «2».

Прочитайте Сходства в синтаксисе, Различия в значении: Python и JavaScript онлайн: https://riptutorial.com/ru/python/topic/10766/сходства-в-синтаксисе--различия-в-значении--python-и-javascript

глава 187: Тестирование устройства

замечания

Для Python существует несколько модулей тестирования. В этом разделе документации описывается базовый модуль `unittest` . Другие инструменты тестирования включают `py.test` и `nosetests` . В этой документации по тестированию python сравниваются некоторые из этих инструментов без углубления.

Examples

Исключения для тестирования

Программы выдают ошибки, если, например, задан неправильный ввод. Из-за этого необходимо убедиться, что при выдаче фактического неправильного ввода возникает ошибка. Из-за этого нам нужно проверить точное исключение, для этого примера мы воспользуемся следующим исключением:

```
class WrongInputException(Exception):
    pass
```

Это исключение возникает, когда вводится неправильный ввод, в следующем контексте, где мы всегда ожидаем число как ввод текста.

```
def convert2number(random_input):
    try:
        my_input = int(random_input)
    except ValueError:
        raise WrongInputException("Expected an integer!")
    return my_input
```

Чтобы проверить, было ли `assertRaises` исключение, мы используем `assertRaises` для проверки этого исключения. `assertRaises` можно использовать двумя способами:

1. Использование обычного вызова функции. Первый аргумент принимает тип исключения, второй - вызываемый (обычно функция), а остальные аргументы передаются этому вызываемому.
2. Использование предложения `with` , предоставляющее только тип исключения для этой функции. У этого есть преимущество, что больше кода может быть выполнено, но его следует использовать с осторожностью, поскольку несколько функций могут использовать одно и то же исключение, которое может быть проблематичным. Пример: с self.assertRaises (WrongInputException): convert2number ("not number")

Это было реализовано в следующем тестовом примере:

```
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()
```

Также может потребоваться проверка исключения, которое не должно было быть выбрано. Тем не менее, тест будет автоматически терпеть неудачу при возникновении исключения и, следовательно, может вообще не понадобиться. Просто чтобы показать параметры, второй метод тестирования показывает случай, как можно проверить, не исключено ли исключение. В принципе, это ловушка исключения, а затем отказ от теста с использованием метода `fail` .

Смысловые функции с unittest.mock.create_autospec

Один из способов издеваться над функцией - использовать функцию `create_autospec` , которая будет издеваться над объектом в соответствии со своими спецификациями. С помощью функций мы можем использовать это, чтобы гарантировать, что они вызываются надлежащим образом.

С функцией `multiply` в `custom_math.py` :

```
def multiply(a, b):
    return a * b
```

И функция `multiples_of` в `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
```

Мы можем протестировать `multiples_of` одиночку, издеваясь `multiply` . В приведенном ниже примере используется стандартная библиотека Python unittest, но это может быть использовано и с другими платформами тестирования, например, pytest или нос:

```
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
```

Настройка тестирования и отключение в пределах unittest.TestCase

Иногда мы хотим подготовить контекст для каждого теста, который будет запущен. Метод `setUp` выполняется до каждого теста в классе. `tearDown` запускается в конце каждого теста. Эти методы являются необязательными. Помните, что TestCases часто используются в совместном множественном наследовании, поэтому вы должны быть осторожны, чтобы всегда вызывать `super` в этих методах, чтобы также `tearDown` методы `setUp` и `tearDown` базового класса. Базовая реализация `TestCase` предоставляет пустые `setUp` и `tearDown` чтобы их можно было вызывать без привлечения исключений:

```
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()
```

Обратите внимание, что в python2.7 + существует также метод `addCleanup` который регистрирует функции, которые будут вызываться после запуска теста. В отличие от `tearDown` который только `setUp` если `setUp` преуспевает, функции, зарегистрированные через `addCleanup` будут вызываться даже в случае необработанного исключения в `setUp` . В качестве конкретного примера этот метод часто можно увидеть, удаляя различные издевательства, которые были зарегистрированы во время теста:

```
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)
```

Другим преимуществом регистрации очистки таким образом является то, что он позволяет программисту поместить код очистки рядом с установочным кодом, и он защитит вас в том случае, если субкласс забывает называть `super` в `tearDown` .

Утверждение об исключениях

Вы можете проверить, что функция генерирует исключение со встроенным unittest с помощью двух разных методов.

Использование диспетчера контекстов

```
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)
```

Это запустит код внутри диспетчера контекста, и если он преуспеет, он не пройдет тест, потому что исключение не было создано. Если код вызывает исключение правильного типа, тест будет продолжен.

Вы также можете получить содержимое поднятого исключения, если хотите выполнить против него дополнительные утверждения.

```
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')
```

Предоставляя вызываемую функцию

```
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)
```

Исключение для проверки должно быть первым параметром, а вызываемая функция должна быть передана как второй параметр. Любые другие указанные параметры будут переданы непосредственно вызываемой функции, что позволит вам указать параметры, которые инициируют исключение.

Выбор утверждений в рамках Unittests

В то время как у Python есть `assert` , структура тестирования модулей Python имеет лучшие утверждения, специализированные для тестов: они более информативны при сбоях и не зависят от режима отладки исполнения.

Возможно, самым простым утверждением является `assertTrue` , которое можно использовать следующим образом:

```
import unittest

class SimplisticTest(unittest.TestCase):
    def test_basic(self):
        self.assertTrue(1 + 1 == 2)
```

Это будет нормально работать, но вместо этой строки

```
        self.assertTrue(1 + 1 == 3)
```

не удастся.

Утверждение `assertTrue` скорее всего, является самым общим утверждением, так как все проверенное может быть отлито как некоторое логическое условие, но часто есть лучшие альтернативы. При тестировании на равенство, как и выше, лучше писать

```
        self.assertEqual(1 + 1, 3)
```

Когда первое не удается, сообщение

```
======================================================================

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
```

но когда последнее не выполняется, сообщение

```
======================================================================

FAIL: test (__main__.TruthTest)

----------------------------------------------------------------------

Traceback (most recent call last):

  File "stuff.py", line 6, in test

    self.assertEqual(1 + 1, 3)
AssertionError: 2 != 3
```

который более информативен (он фактически оценил результат левой стороны).

Вы можете найти список утверждений в стандартной документации . В общем, это хорошая идея, чтобы выбрать утверждение, которое наиболее точно соответствует условию. Таким образом, как показано выше, для утверждения, что `1 + 1 == 2` лучше использовать `assertEqual` чем `assertTrue` . Аналогично, для утверждения, что `a is None` , лучше использовать `assertIsNone` чем `assertEqual` .

Заметим также, что утверждения имеют отрицательные формы. Таким образом, `assertEqual` имеет свой отрицательный аналог `assertNotEqual` , а `assertIsNone` имеет свой отрицательный аналог `assertIsNotNone` . Опять же, используя, при необходимости, отрицательные копии, приведет к более ясным сообщениям об ошибках.

Единичные тесты с помощью pytest

установка pytest:

```
pip install pytest
```

подготовка тестов:

```
mkdir tests
touch tests/test_docker.py
```

Функции для тестирования в `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)
```

Тест импортирует `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
```

издеваясь над файлом, подобным объекту в `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
```

Патч обезьяны с pytest в `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
```

Примеры тестов должны начинаться с префикса `test_` в файле `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])
```

запуск тестов по одному за раз:

```
py.test -k test_docker_install tests
py.test -k test_copy_file_to_docker tests
py.test -k test_docker_exec_something tests
```

запуск всех тестов в папке `tests` :

```
py.test -k test_ tests
```

Прочитайте Тестирование устройства онлайн: https://riptutorial.com/ru/python/topic/631/тестирование-устройства

глава 188: Тип подсказки

Синтаксис

- typing.Callable [[int, str], None] -> def func (a: int, b: str) -> None
- 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 или int
- typing.Sequence [int] -> [1, 2, 3] или (1, 2, 3)
- typing.Any -> Любой тип
- typing.Union [int, str] -> 1 или "1"
- T = typing.TypeVar ('T') -> Общий тип

замечания

Тип Hinting, как указано в `PEP 484` , является формализованным решением для статического указания типа значения для кода Python. Появляясь рядом с модулем `typing` , подсказки типа предлагают пользователям Python возможность комментировать их код, тем самым помогая контролерам типов, а косвенно, документируя их код с дополнительной информацией.

Examples

Общие типы

Тип `typing.TypeVar` - это фабрика общего типа. Основная задача состоит в том, чтобы служить параметром / заполнителем для общих аннотаций функции / класса / метода:

```
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]
```

Добавление типов к функции

Возьмем пример функции, которая получает два аргумента и возвращает значение, обозначающее их сумму:

```
def two_sum(a, b):
```

```
    return a + b
```

`two_sum` этот код, нельзя смело и без сомнения указать тип аргументов функции `two_sum` . Он работает как при `int` значений `int` :

```
print(two_sum(2, 1))  # result: 3
```

и со строками:

```
print(two_sum("a", "b"))  # result: "ab"
```

и с другими значениями, такими как `list` s, `tuple` s et cetera.

Из-за этой динамической природы типов python, где многие применимы для данной операции, любая проверка типов не сможет обоснованно утверждать, должен ли разрешаться вызов этой функции или нет.

Чтобы помочь нашему контролеру типов, мы можем теперь предоставить ему подсказки типов в определении функции, указывающем тип, который мы разрешаем.

Чтобы указать, что мы хотим только разрешить типы `int` мы можем изменить определение функции следующим образом:

```
def two_sum(a: int, b: int):
    return a + b
```

Аннотации следуют за именем аргумента и разделяются символом `:`

Точно так же, чтобы указать только `str` типа разрешены, мы бы изменить нашу функцию , чтобы определить его:

```
def two_sum(a: str, b: str):
    return a + b
```

Помимо указания типа аргументов, можно также указать возвращаемое значение вызова функции. Это делается добавлением символа `->` за которым следует тип после закрывающей круглой скобки в списке аргументов, *но* до `:` в конце объявления функции:

```
def two_sum(a: int, b: int) -> int:
    return a + b
```

Теперь мы указали, что возвращаемое значение при вызове `two_sum` должно иметь тип `int` . Аналогичным образом мы можем определить соответствующие значения для `str` , `float` , `list` , `set` и других.

Хотя подсказки типов в основном используются контролерами типов и IDE, иногда вам

может потребоваться их получить. Это можно сделать с `__annotations__` специального атрибута `__annotations__` :

```
two_sum.__annotations__
# {'a': <class 'int'>, 'b': <class 'int'>, 'return': <class 'int'>}
```

Участники и методы класса

```
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))
```

Требуется прямая ссылка текущего класса, так как аннотации оцениваются при определении функции. Прямые ссылки также могут использоваться при обращении к классу, которые приведут к циклическому импорту, если импортируются.

Переменные и атрибуты

Переменные аннотируются с использованием комментариев:

```
x = 3  # type: int
x = negate(x)
x = 'a type-checker might catch this error'
```

Python 3.x 3.6

Начиная с Python 3.6, есть также новый синтаксис для переменных аннотаций . Вышеприведенный код может использовать форму

```
x: int = 3
```

В отличие от комментариев, также можно просто добавить подсказку типа к переменной, которая ранее не была объявлена, без установки для нее значения:

```
y: int
```

Кроме того, если они используются в модуле или уровне класса, подсказки типа могут быть

получены с помощью `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'>}, {})
```

Кроме того, к ним можно получить доступ с помощью специальной переменной или атрибута `__annotations__` :

```
x: int
print(__annotations__)
# {'x': <class 'int'>}

class C:
    s: str
print(C.__annotations__)
# {'s': <class 'str'>}
```

NamedTuple

Создание именованного набора с подсказками типа выполняется с помощью функции NamedTuple из модуля `typing` :

```
import typing
Point = typing.NamedTuple('Point', [('x', int), ('y', int)])
```

Обратите внимание, что имя результирующего типа является первым аргументом функции, но оно должно быть назначено переменной с тем же именем, чтобы облегчить работу контролеров типов.

Введите подсказки для аргументов ключевых слов

```
def hello_world(greeting: str = 'Hello'):
    print(greeting + ' world!')
```

Обратите внимание на пробелы вокруг знака равенства, а не на то, как обычно формулируются аргументы ключевых слов.

Прочитайте Тип подсказки онлайн: https://riptutorial.com/ru/python/topic/1766/тип-подсказки

глава 189: Типы данных Python

Вступление

Типы данных - это не что иное, как переменная, которую вы использовали для резервирования некоторого пространства в памяти. Переменные Python не нуждаются в явном объявлении для резервирования пространства памяти. Объявление присваивается автоматически, когда вы присваиваете значение переменной.

Examples

Тип данных чисел

Числа имеют четыре типа в Python. Int, float, complex и long.

```
int_num = 10    #int value
float_num = 10.2    #float value
complex_num = 3.14j    #complex value
long_num = 1234567L    #long value
```

Строковый тип данных

Строка определяется как непрерывный набор символов, представленных в кавычках. Python позволяет использовать пары одиночных или двойных кавычек. Строки - это неизменный тип данных последовательности, т. Е. Каждый раз, когда вы вносите какие-либо изменения в строку, создается совершенно новый строковый объект.

```
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
```

Тип данных списка

Список содержит элементы, разделенные запятыми и заключенные в квадратные скобки []. Списки почти аналогичны массивам в C. Одно отличие состоит в том, что все элементы, принадлежащие списку, могут быть разных типов данных.

```
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']
```

Тип данных Tuple

Списки заключены в скобки [], и их элементы и размер могут быть изменены, в то время как кортежи заключены в круглые скобки () и не могут быть обновлены. Кортежи неизменяемы.

```
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.
```

Тип данных словаря

Словарь состоит из пар ключ-значение. Он заключен в фигурные скобки {}, и значения могут быть назначены и доступны с помощью квадратных скобок [].

```
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']
```

Установка типов данных

Наборы представляют собой неупорядоченные коллекции уникальных объектов, существует два типа набора:

1. Наборы - они изменяемы, и новые элементы могут быть добавлены после определения наборов

```
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. Замороженные наборы - они неизменяемы, и новые элементы не могут быть добавлены после его определения.

```
b = frozenset('asdfagsa')
print(b)
> frozenset({'f', 'g', 'd', 'a', 's'})
cities = frozenset(["Frankfurt", "Basel","Freiburg"])
print(cities)
```

```
> frozenset({'Frankfurt', 'Basel', 'Freiburg'})
```

Прочитайте Типы данных Python онлайн: https://riptutorial.com/ru/python/topic/9366/типы-данных-python

глава 190: толковый словарь

Синтаксис

- mydict = {}
- mydict [k] = значение
- value = mydict [k]
- value = mydict.get (k)
- value = mydict.get (k, "default_value")

параметры

параметр	подробности
ключ	Желаемый ключ для поиска
значение	Значение для установки или возврата

замечания

Полезные элементы, которые нужно запомнить при создании словаря:

- Каждый ключ должен быть **уникальным** (иначе он будет отменен)
- Каждый ключ должен быть **хешируемым** (может использовать `hash` функцию для хеширования, иначе `TypeError` будет выброшен)
- Для ключей нет особого порядка.

Examples

Доступ к значениям словаря

```
dictionary = {"Hello": 1234, "World": 5678}
print(dictionary["Hello"])
```

Вышеприведенный код напечатает `1234` .

Строка `"Hello"` в этом примере называется *ключом* . Он используется для поиска значения в `dict` , помещая ключ в квадратные скобки.

Число `1234` видно после соответствующего двоеточия в определении `dict` . Это называется *значением,* которое `"Hello"` *отображает* в этом `dict` .

Поиск такого значения с помощью ключа, который не существует, приведет к `KeyError` исключения `KeyError` , прекратив выполнение, если он не сфотографирован. Если мы хотим получить доступ к значению без риска использования `KeyError` , мы можем использовать метод `dictionary.get` . По умолчанию, если ключ не существует, метод возвращает `None` . Мы можем передать ему второе значение для возврата вместо `None` в случае неудачного поиска.

```
w = dictionary.get("whatever")
x = dictionary.get("whatever", "nuh-uh")
```

В этом примере `w` получит значение `None` а `x` получит значение `"nuh-uh"` .

Конструктор dict ()

Конструктор `dict()` может использоваться для создания словарей из аргументов ключевого слова или из одного итерабельного из пар ключ-значение или из одного словаря и аргументов ключевого слова.

```
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}
```

Избегание исключений KeyError

Одной из распространенных ошибок при использовании словарей является доступ к несуществующему ключу. Обычно это `KeyError` исключению `KeyError`

```
mydict = {}
mydict['not there']
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'not there'
```

Один из способов избежать ключевых ошибок - использовать метод `dict.get` , который позволяет указать значение по умолчанию для возврата в случае отсутствующего ключа.

```
value = mydict.get(key, default_value)
```

Что возвращает `mydict[key]` если он существует, но в противном случае возвращает значение `default_value` . Обратите внимание, что это не добавляет `key` к `mydict` . Так что если вы хотите сохранить эту ключевую пару значений, вы должны использовать `mydict.setdefault(key, default_value)` , который *не* хранить пару ключей значения.

```
mydict = {}
```

```
print(mydict)
# {}
print(mydict.get("foo", "bar"))
# bar
print(mydict)
# {}
print(mydict.setdefault("foo", "bar"))
# bar
print(mydict)
# {'foo': 'bar'}
```

Альтернативный способ решения этой проблемы - уловка исключения

```
try:
    value = mydict[key]
except KeyError:
    value = default_value
```

Вы также можете проверить, находится ли ключ `in` словаре.

```
if key in mydict:
    value = mydict[key]
else:
    value = default_value
```

Однако обратите внимание, что в многопоточных средах ключ может быть удален из словаря после проверки, создавая условие гонки, в котором все еще может быть выбрано исключение.

Другой вариант - использовать подкласс dict, collection.defaultdict, который имеет свойство default_factory для создания новых записей в dict при задании new_key.

Доступ к ключам и значениям

При работе со словарями часто необходимо получить доступ ко всем ключам и значениям в словаре, либо в цикле `for` , либо в понимании списка, либо просто в виде простого списка.

Учитывая словарь, например:

```
mydict = {
    'a': '1',
    'b': '2'
}
```

Вы можете получить список ключей с помощью метода `keys()` :

```
print(mydict.keys())
# Python2: ['a', 'b']
# Python3: dict_keys(['b', 'a'])
```

Если вместо этого вы хотите получить список значений, используйте метод `values()` :

```
print(mydict.values())
# Python2: ['1', '2']
# Python3: dict_values(['2', '1'])
```

Если вы хотите работать как с ключом, так и с его соответствующим значением, вы можете использовать метод `items()` :

```
print(mydict.items())
# Python2: [('a', '1'), ('b', '2')]
# Python3: dict_items([('b', '2'), ('a', '1')])
```

ПРИМЕЧАНИЕ. Поскольку `dict` unsorted, `keys()` , `values()` и `items()` имеют порядка сортировки. Используйте `sort()` , `OrderedDict sorted()` или `OrderedDict` если вам `OrderedDict` порядок, возвращаемый этими методами.

Python 2/3 Difference: в Python 3 эти методы возвращают специальные итеративные объекты, а не списки, и эквивалент методов Python 2 `iterkeys()` , `itervalues()` и `iteritems()` . Эти объекты могут использоваться как списки по большей части, хотя есть некоторые отличия. Подробнее см. PEP 3106 .

Введение в словарь

Словарь - это пример *хранилища значений ключей,* также известный как *Mapping* в Python. Он позволяет хранить и извлекать элементы, ссылаясь на ключ. Поскольку словари ссылаются на ключ, у них очень быстрый поиск. Поскольку они в основном используются для ссылки на элементы по ключу, они не сортируются.

создание диктата

Словари могут быть инициированы по-разному:

литеральный синтаксис

```
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}
```

понимание диктата

```
d = {k:v for k,v in [('key', 'value',)]}
```

Смотри также: Понимание

встроенный класс: `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)
```

изменение диктата

Чтобы добавить элементы в словарь, просто создайте новый ключ со значением:

```
d['newkey'] = 42
```

Также можно добавить `list` и `dictionary` качестве значения:

```
d['new_list'] = [1, 2, 3]
d['new_dict'] = {'nested_dict': 1}
```

Чтобы удалить элемент, удалите ключ из словаря:

```
del d['newkey']
```

Словарь со значениями по умолчанию

Доступно в стандартной библиотеке как `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'
```

[*] В качестве альтернативы, если вы должны использовать встроенный класс `dict` , `using`

`dict.setdefault()` позволит вам создавать по умолчанию всякий раз, когда вы получаете доступ к ключу, который раньше не существовал:

```
>>> d = {}
{}
>>> d.setdefault('Another_key', []).append("This worked!")
>>> d
{'Another_key': ['This worked!']}
```

Имейте в виду, что если у вас есть много значений для добавления, `dict.setdefault()` создаст новый экземпляр начального значения (в этом примере a `[]`) при каждом его вызове - что может создать ненужные рабочие нагрузки.

[*] *Python Cookbook, 3-е издание, Дэвид Бэйсли и Брайан К. Джонс (O'Reilly). Copyright 2013 Дэвид Бэйсли и Брайан Джонс, 978-1-449-34037-7.*

Создание упорядоченного словаря

Вы можете создать упорядоченный словарь, который будет следовать определенному порядку при итерации по клавишам в словаре.

Используйте `OrderedDict` из модуля `collections` . Это всегда будет возвращать элементы словаря в исходном порядке вставки при повторении.

```
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])
```

Распаковка словарей с использованием оператора **

Вы можете использовать оператор распараллеливания аргументов аргументов `**` для доставки пар ключ-значение в словаре в аргументы функции. Упрощенный пример из официальной документации :

```
>>>
>>> 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 !
```

С Python 3.5 вы также можете использовать этот синтаксис для слияния произвольного количества объектов `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'}
```

Как показывает этот пример, дубликаты ключей сопоставляются с их последним значением (например, «Клиффорд» переопределяет «Немо»).

Слияние словарей

Рассмотрим следующие словари:

```
>>> 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'}
```

Как показывает этот пример, дубликаты ключей сопоставляются с их последним значением (например, «Клиффорд» переопределяет «Немо»).

Python 3.3+

```
>>> from collections import ChainMap
>>> dict(ChainMap(fish, dog))
{'hands': 'fins', 'color': 'red', 'special': 'gills', 'name': 'Nemo'}
```

При таком методе главное значение имеет приоритет для данного ключа, а не последнего («Клиффорд» выбрасывается в пользу «Немо»).

Python 2.x, 3.x

```
>>> from itertools import chain
>>> dict(chain(fish.items(), dog.items()))
{'hands': 'paws', 'color': 'red', 'name': 'Clifford', 'special': 'gills'}
```

Это использует последнее значение, как с использованием метода `**` для объединения («Клиффорд» переопределяет «Немо»).

```
>>> fish.update(dog)
>>> fish
{'color': 'red', 'hands': 'paws', 'name': 'Clifford', 'special': 'gills'}
```

`dict.update` использует последний dict для перезаписывания предыдущего.

Задняя запятая

Подобно спискам и кортежам, вы можете включить запятую в ваш словарь.

```
role = {"By day": "A typical programmer",
        "By night": "Still a typical programmer", }
```

PEP 8 диктует, что вы должны оставить пробел между конечной запятой и закрывающей скобкой.

Все комбинации значений словаря

```
options = {
    "x": ["a", "b"],
    "y": [10, 20, 30]
}
```

Для словаря, такого как показанный выше, где есть список, представляющий набор значений для изучения соответствующего ключа. Предположим, вы хотите изучить `"x"="a"` с `"y"=10` , затем `"x"="a"` с `"y"=10` и т. Д., Пока вы не изучите все возможные комбинации.

Вы можете создать список, который возвращает все такие комбинации значений, используя следующий код.

```
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
```

Это дает нам следующий список, хранящийся в `combinations` переменных:

```
[{'x': 'a', 'y': 10},
 {'x': 'b', 'y': 10},
 {'x': 'a', 'y': 20},
 {'x': 'b', 'y': 20},
 {'x': 'a', 'y': 30},
 {'x': 'b', 'y': 30}]
```

Итерация по словарю

Если вы используете словарь как итератор (например, в инструкции `for`), он перемещает **ключи** словаря. Например:

```
d = {'a': 1, 'b': 2, 'c':3}
for key in d:
    print(key, d[key])
# c 3
# b 2
# a 1
```

То же самое верно при использовании в понимании

```
print([key for key in d])
# ['c', 'b', 'a']
```

Python 3.x 3.0

Метод `items()` может использоваться для циклического переключения между **ключом** и **значением** одновременно:

```
for key, value in d.items():
    print(key, value)
# c 3
# b 2
# a 1
```

Хотя метод `values()` может использоваться для перебора только значений, как и следовало ожидать:

```
for key, value in d.values():
    print(key, value)
    # 3
    # 2
    # 1
```

Python 2.x 2.2

Здесь, методы `keys()` , `values()` и `items()` возвращают списки, и есть три дополнительных метода `iterkeys()` `itervalues()` и `iteritems()` для возврата итераторов.

Создание словаря

Правила создания словаря:

- Каждый ключ должен быть **уникальным** (иначе он будет отменен)
- Каждый ключ должен быть **хешируемым** (может использовать `hash` функцию для хеширования, иначе `TypeError` будет выброшен)
- Для ключей нет особого порядка.

```
# 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}
```

Примеры словарей

Словари сопоставляют ключи со значениями.

```
car = {}
car["wheels"] = 4
car["color"] = "Red"
car["model"] = "Corvette"
```

Значения словаря могут быть доступны по их ключам.

```
print "Little " + car["color"] + " " + car["model"] + "!"
# This would print out "Little Red Corvette!"
```

Словари также могут быть созданы в стиле JSON:

```
car = {"wheels": 4, "color": "Red", "model": "Corvette"}
```

Значения словаря можно повторить:

```
for key in car:
  print key + ": " + car[key]

# wheels: 4
# color: Red
# model: Corvette
```

Прочитайте толковый словарь онлайн: https://riptutorial.com/ru/python/topic/396/толковый-словарь

глава 191: Узел узлового списка

Examples

Напишите простой узел Linked List в python

Связанный список:

- пустой список, представленный None, или
- узел, который содержит грузовой объект и ссылку на связанный список.

```
#! /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")
```

Прочитайте Узел узлового списка онлайн: https://riptutorial.com/ru/python/topic/6916/узел-узлового-списка

глава 192: уменьшить

Синтаксис

- reduce (function, iterable [, initializer])

параметры

параметр	подробности
функция	функция, которая используется для уменьшения итерации (требуется два аргумента). (*только для позиции*)
итерируемый	iterable, который будет уменьшен. (*только для позиции*)
инициализатор	начальная стоимость сокращения. (*необязательно* , *только для позиции*)

замечания

`reduce` может быть не всегда наиболее эффективной функцией. Для некоторых типов существуют эквивалентные функции или методы:

- `sum()` для суммы последовательности, содержащей *сменные* элементы (не строки):

```
sum([1,2,3])                                 # = 6
```

- `str.join` для конкатенации строк:

```
''.join(['Hello', ',', ' World'])            # = 'Hello, World'
```

- `next` вместе с генератором может быть вариантом короткого замыкания по сравнению с `reduce` :

```
# First falsy item:
next((i for i in [100, [], 20, 0] if not i)) # = []
```

Examples

обзор

```
# 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` уменьшает итерацию, повторно применяя функцию к следующему элементу `iterable` и кумулятивного результата.

```
def add(s1, s2):
    return s1 + s2

asequence = [1, 2, 3]

reduce(add, asequence)  # equivalent to: add(add(1,2),3)
# Out: 6
```

В этом примере мы определили нашу собственную функцию `add` . Однако Python имеет стандартную эквивалентную функцию в модуле `operator` :

```
import operator
reduce(operator.add, asequence)
# Out: 6
```

`reduce` также можно передать начальное значение:

```
reduce(add, asequence, 10)
# Out: 16
```

Использование сокращения

```
def multiply(s1, s2):
    print('{arg1} * {arg2} = {res}'.format(arg1=s1,
                                           arg2=s2,
                                           res=s1*s2))
    return s1 * s2

asequence = [1, 2, 3]
```

При `initializer` функция запускается путем применения ее к инициализатору и первому итерируемому элементу:

```
cumprod = reduce(multiply, asequence, 5)
# Out: 5 * 1 = 5
#      5 * 2 = 10
#      10 * 3 = 30
print(cumprod)
# Out: 30
```

Без параметра `initializer` `reduce` начинается с применения функции к первым двум элементам списка:

```
cumprod = reduce(multiply, asequence)
# Out: 1 * 2 = 2
#      2 * 3 = 6
print(cumprod)
# Out: 6
```

Накопительный продукт

```
import operator
reduce(operator.mul, [10, 5, -3])
# Out: -150
```

Вариант без короткого замыкания любого / всего

`reduce` не приведет к завершению итерации до того, как `iterable` будет завершен полностью, поэтому он может быть использован для создания функции короткого замыкания `any()` или `all()` :

```
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
```

Первый элемент правды / ложности последовательности (или последний элемент, если их нет)

```
# 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
```

Вместо создания `lambda` функции обычно рекомендуется создать именованную функцию:

```
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])                  # = []
```

Прочитайте уменьшить онлайн: https://riptutorial.com/ru/python/topic/328/уменьшить

глава 193: Файлы и папки I / O

Вступление

Когда речь идет о хранении, чтении или передаче данных, работа с файлами операционной системы необходима и проста с помощью Python. В отличие от других языков, где ввод и вывод файлов требует сложного чтения и записи объектов, Python упрощает процесс, требуя только команд для открытия, чтения / записи и закрытия файла. В этом разделе объясняется, как Python может взаимодействовать с файлами в операционной системе.

Синтаксис

- file_object = open (имя_файла [, access_mode] [, буферизация])

параметры

параметр	подробности
имя файла	путь к вашему файлу или, если файл находится в рабочем каталоге, имя файла вашего файла
access_mode	строковое значение, определяющее способ открытия файла
буферизация	целочисленное значение, используемое для необязательной буферизации строк

замечания

Избегайте кросс-платформенного кодирования Ад

При использовании встроенного `open()` Python лучше всего всегда передавать аргумент `encoding` , если вы намерены использовать ваш код для кросс-платформенного. Причина этого заключается в том, что стандартная кодировка системы отличается от платформы к платформе.

Хотя `linux` системы действительно используют `utf-8` по умолчанию, это ***не*** обязательно верно для MAC и Windows.

Чтобы проверить кодировку по умолчанию, попробуйте следующее:

```
import sys
sys.getdefaultencoding()
```

от любого интерпретатора python.

Следовательно, разумно всегда выделять кодировку, чтобы убедиться, что строки, с которыми вы работаете, закодированы как то, что вы думаете, что они обеспечивают, кросс-платформенная совместимость.

```
with open('somefile.txt', 'r', encoding='UTF-8') as f:
    for line in f:
        print(line)
```

Examples

Режимы файлов

Существуют разные режимы, с помощью которых вы можете открыть файл, заданный параметром `mode` . Они включают:

- `'r'` - режим чтения. По умолчанию. Это позволяет вам читать только файл, а не изменять его. При использовании этого режима файл должен существовать.
- `'w'` - режим записи. Он создаст новый файл, если он не существует, иначе он удалит файл и позволит вам записать его.
- `'a'` - добавить режим. Он будет записывать данные в конец файла. Он не стирает файл, и файл должен существовать для этого режима.
- `'rb'` - режим чтения в двоичном формате. Это похоже на `r` за исключением того, что чтение принудительно в двоичном режиме. Это также выбор по умолчанию.
- `'r+'` - режим чтения и режим записи одновременно. Это позволяет одновременно читать и записывать файлы без использования `r` и `w` .
- `'rb+'` - режим чтения и записи в двоичном формате. То же, что и `r+` за исключением данных в двоичном
- `'wb'` - режим записи в двоичном формате. То же, что и `w` за исключением данных, находится в двоичном формате.
- `'w+'` - режим записи и чтения. Точно так же, как и `r+` но если файл не существует, создается новый. В противном случае файл будет перезаписан.
- `'wb+'` - режим записи и чтения в двоичном режиме. То же, что и `w+` но данные

находятся в двоичном формате.

- `'ab'` - добавление в двоичном режиме. Как и `a` исключением того, что данные находятся в двоичном формате.
- `'a+'` - режим добавления и чтения. Подобно `w+` поскольку он создаст новый файл, если файл не существует. В противном случае указатель файла находится в конце файла, если он существует.
- `'ab+'` - режим добавления и чтения в двоичном формате. То же, что `a+` за исключением того, что данные находятся в двоичном формате.

```
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)
```

	р	г +	вес	ш +		а +
Читать	✔	✔	✘	✔	✘	✔
Написать	✘	✔	✔	✔	✔	✔
Создает файл	✘	✘	✔	✔	✔	✔
Стирает файл	✘	✘	✔	✔	✘	✘
Исходное положение	Начните	Начните	Начните	Начните	Конец	Конец

Python 3 добавил новый режим для `exclusive creation` чтобы вы случайно не усекали или не перезаписывали и не добавляли существующий файл.

- `'x'` - открыт для исключительного создания, поднимет `FileExistsError` если файл уже существует
- `'xb'` - открыт для режима создания уникального создания в двоичном формате. То же, что и `x` кроме данных, находится в двоичном формате.
- `'x+'` - режим чтения и записи. Подобно `w+` поскольку он создаст новый файл, если файл не существует. В противном случае будет увеличено значение `FileExistsError` .
- `'xb+'` - режим записи и чтения. Точно так же, как `x+` но данные двоичные

	Икс	х +
Читать	✘	✔
Написать	✔	✔

	Икс	x +
Создает файл	✔	✔
Стирает файл	✘	✘
Исходное положение	Начните	Начните

Позвольте написать файл открытого кода более питоническим образом:

Python 3.x 3.3

```
try:
    with open("fname", "r") as fout:
        # Work with your open file
except FileExistsError:
    # Your error handling goes here
```

В Python 2 вы бы сделали что-то вроде

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
```

Чтение файла по очереди

Самый простой способ перебора файлов по строкам:

```
with open('myfile.txt', 'r') as fp:
    for line in fp:
        print(line)
```

`readline()` позволяет более детально управлять последовательной итерацией. Пример ниже эквивалентен приведенному выше:

```
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)
```

Использование итератора for loop и readline () вместе считается плохой практикой.

Чаще всего метод `readlines()` используется для хранения итерируемой коллекции строк файла:

```
with open("myfile.txt", "r") as fp:
    lines = fp.readlines()
for i in range(len(lines)):
    print("Line " + str(i) + ": " + line)
```

Это напечатает следующее:

> Строка 0: привет
>
> Линия 1: мир

Получение полного содержимого файла

Предпочтительный способ ввода / вывода файла заключается в использовании `with` ключевым словом. Это гарантирует, что дескриптор файла будет закрыт после завершения чтения или записи.

```
with open('myfile.txt') as in_file:
    content = in_file.read()

print(content)
```

или, чтобы справиться с закрытием файла вручную, вы можете отказаться `with` и просто позвонить `close` себя:

```
in_file = open('myfile.txt', 'r')
content = in_file.read()
print(content)
in_file.close()
```

Имейте в виду, что без использования инструкции `with` вы можете случайно открыть файл в случае возникновения непредвиденного исключения:

```
in_file = open('myfile.txt', 'r')
raise Exception("oops")
in_file.close()  # This will never be called
```

Запись в файл

```
with open('myfile.txt', 'w') as f:
    f.write("Line 1")
    f.write("Line 2")
    f.write("Line 3")
    f.write("Line 4")
```

Если вы откроете `myfile.txt` , вы увидите, что его содержимое:

Линия 1Line 2Line 3Line 4

Python автоматически не добавляет разрывы строк, вам нужно сделать это вручную:

```
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")
```

Строка 1
Строка 2
Строка 3
Строка 4

Не используйте `os.linesep` в качестве ограничителя строк при записи файлов, открытых в текстовом режиме (по умолчанию); Вместо этого используйте `\n` .

Если вы хотите указать кодировку, вы просто добавляете параметр `encoding` в `open` функцию:

```
with open('my_file.txt', 'w', encoding='utf-8') as f:
    f.write('utf-8 text')
```

Также можно использовать оператор печати для записи в файл. Механика отличается от Python 2 и Python 3, но концепция та же самая, что и вы можете сделать вывод, который бы вышел на экран и отправить его в файл.

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
```

В Python 2 вы бы сделали что-то вроде

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
```

В отличие от функции записи функция печати автоматически добавляет разрывы строк.

Копирование содержимого одного файла в другой файл

```
with open(input_file, 'r') as in_file, open(output_file, 'w') as out_file:
    for line in in_file:
        out_file.write(line)
```

- Использование `shutil` модуля:

```
import shutil
shutil.copyfile(src, dst)
```

Проверьте, существует ли файл или путь

Используйте стиль кодирования EAFP и `try` открыть его.

```
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
```

Это также позволит избежать условий гонки, если другой процесс удаляет файл между проверкой и когда он используется. Это состояние гонки может произойти в следующих случаях:

- Использование модуля `os` :

  ```
  import os
  os.path.isfile('/path/to/some/file.txt')
  ```

Python 3.x 3.4

- Использование `pathlib` :

  ```
  import pathlib
  path = pathlib.Path('/path/to/some/file.txt')
  if path.is_file():
      ...
  ```

Чтобы проверить, существует ли данный путь или нет, вы можете выполнить описанную выше процедуру EAFP или явно проверить путь:

```
import os
path = "/home/myFiles/directory1"
```

```
if os.path.exists(path):
    ## Do stuff
```

Скопировать дерево каталогов

```
import shutil
source='//192.168.1.2/Daily Reports'
destination='D:\\Reports\\Today'
shutil.copytree(source, destination)
```

Целевой каталог уже **не должен существовать** .

Итерировать файлы (рекурсивно)

Чтобы итератировать все файлы, в том числе в подкаталогах, используйте os.walk:

```
import os
for root, folders, files in os.walk(root_dir):
    for filename in files:
        print root, filename
```

root_dir может быть "." для запуска из текущего каталога или любого другого пути для начала.

Python 3.x 3.5

Если вы также хотите получить информацию о файле, вы можете использовать более эффективный метод os.scandir следующим образом:

```
for entry in os.scandir(path):
   if not entry.name.startswith('.') and entry.is_file():
       print(entry.name)
```

Чтение файла между диапазоном строк

Итак, давайте предположим, что вы хотите итерации только между некоторыми конкретными строками файла

Вы можете использовать `itertools` для этого

```
import itertools

with open('myfile.txt', 'r') as f:
    for line in itertools.islice(f, 12, 30):
        # do something here
```

Это будет читать строки с 13 по 20, так как при индексации python начинается с 0. Так что строка номер 1 индексируется как 0

Также можно прочитать некоторые дополнительные строки, используя здесь `next()` ключевое слово.

И когда вы используете файл-объект как итеративный, не используйте здесь инструкцию `readline()` поскольку два метода перемещения файла не должны смешиваться

Случайный доступ к файлам с помощью mmap

Использование модуля `mmap` позволяет пользователю случайным образом обращаться к местоположениям в файле путем сопоставления файла в память. Это альтернатива использованию обычных файловых операций.

```
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()
```

Замена текста в файле

```
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='')
```

Проверка того, что файл пуст

```
>>> import os
>>> os.stat(path_to_file).st_size == 0
```

или же

```
>>> import os
>>> os.path.getsize(path_to_file) > 0
```

Тем не менее, оба будут генерировать исключение, если файл не существует. Чтобы избежать такой ошибки, сделайте следующее:

```
import os
def is_empty_file(fpath):
    return os.path.isfile(fpath) and os.path.getsize(fpath) > 0
```

который вернет значение `bool` .

Прочитайте Файлы и папки I / O онлайн: https://riptutorial.com/ru/python/topic/267/файлы-и-папки-i---o

глава 194: Фильтр

Синтаксис

- фильтр (функция, итерация)
- itertools.ifilter (функция, iterable)
- future_builtins.filter (функция, итерация)
- itertools.ifilterfalse (функция, iterable)
- itertools.filterfalse (функция, iterable)

параметры

параметр	подробности
функция	*вызывающий,* который определяет условие или `None` затем использует функцию идентификации для фильтрации (*только* для *позиции*)
итерируемый	iterable, который будет отфильтрован (*только для позиции*)

замечания

В большинстве случаев выражение понимания или генератора является более читаемым, более мощным и более эффективным, чем `filter()` или `ifilter()` .

Examples

Основное использование фильтра

Для `filter` исключает элементы последовательности, основанные на некоторых критериях:

```
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>
```

Фильтр без функции

Если параметр функции `None` , тогда будет использоваться функция идентификации:

```
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
```

Фильтровать как проверку короткого замыкания

`filter` (Python 3.x) и `ifilter` (Python 2.x) возвращает генератор , таким образом они могут быть очень удобно при создании теста короткого замыкания , как `or` или `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
```

Чтобы найти первый элемент, который меньше 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)
```

`next` функция дает следующий (в данном случае первый) элемент и, следовательно, является причиной его короткого замыкания.

Дополнительная функция: filterfalse, ifilterfalse

В `itertools` есть дополнительная функция для `filter` :

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
```

который работает точно так же, как `filter` *генератора,* но сохраняет только элементы, которые являются `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)
```

Прочитайте Фильтр онлайн: https://riptutorial.com/ru/python/topic/201/фильтр

глава 195: Форматирование даты

Examples

Время между двумя датами

```
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
```

Синхронизация строки с объектом datetime

Использует коды стандартного формата 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)
```

Вывод объекта datetime в строку

Использует коды стандартного формата 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
```

Прочитайте Форматирование даты онлайн: https://riptutorial.com/ru/python/topic/7284/форматирование-даты

глава 196: Форматирование строк

Вступление

Когда хранятся и преобразуются данные для людей, чтобы увидеть, форматирование строк может стать очень важным. Python предлагает широкий спектр методов форматирования строк, которые описаны в этом разделе.

Синтаксис

- "{}". format (42) ==> "42"
- "{0}". Format (42) ==> "42"
- "{0: .2f}". Format (42) ==> "42.00"
- Формат "{0: .0f}". (42.1234) ==> "42"
- "{answer}". format (no_answer = 41, answer = 42) ==> "42"
- "{answer: .2f}". format (no_answer = 41, answer = 42) ==> "42.00"
- "{[key]}". format ({'key': 'value'}) ==> "value"
- "{[1]}". Format (['zero', 'one', 'two']) ==> "one"
- "{answer} = {answer}". format (answer = 42) ==> "42 = 42"
- '.join ([' stack ',' overflow ']) ==> "переполнение стека"

замечания

- Следует проверить PyFormat.info за очень тщательное и нежное введение / объяснение того, как это работает.

Examples

Основы форматирования строк

```
foo = 1
bar = 'bar'
baz = 3.14
```

Вы можете использовать `str.format` для форматирования вывода. Парные пары заменяются аргументами в том порядке, в котором передаются аргументы:

```
print('{}, {} and {}'.format(foo, bar, baz))
# Out: "1, bar and 3.14"
```

Индексы также могут быть указаны внутри скобок. Номера соответствуют индексам аргументов, переданных функции `str.format` (на основе 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
```

Можно также использовать именованные аргументы:

```
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."
```

Атрибуты объекта могут ссылаться при передаче в `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"
```

Также могут использоваться словарные ключи:

```
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"
```

То же самое относится к индексам списка и кортежей:

```
my_list = ['zero', 'one', 'two']
print("2nd element is: {0[2]}".format(my_list))  # "0" is optional
# Out: "2nd element is: two"
```

> Примечание. В дополнение к `str.format` , Python также предоставляет modulo operator `%` также известный как *оператор форматирования строк* или *интерполяции* (см. PEP 3101) - для форматирования строк. `str.format` является преемником `%` и он обеспечивает большую гибкость, например, упрощая выполнение нескольких замещений.

Помимо индексов аргументов, вы также можете включить *спецификацию формата* внутри фигурных скобок. Это выражение , которое следует , и особые правила должны предшествовать двоеточие (`:`). Подробное описание спецификации формата см. В документах . Примером спецификации формата является директива выравнивания `:~^20` (`^` обозначает выравнивание по центру, общая ширина 20, заполнение символом ~):

```
'{:~^20}'.format('centered')
# Out: '~~~~~~centered~~~~~~'
```

`format` позволяет поведение невозможно с `%` , например повторение аргументов:

```
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
```

Поскольку `format` является функцией, его можно использовать в качестве аргумента в других функциях:

```
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
```

Выравнивание и отступы

Python 2.x 2.6

Метод `format()` может использоваться для изменения выравнивания строки. Вы должны сделать это с помощью выражения формата формы `:[fill_char][align_operator][width]` где `align_operator` является одним из:

- `<` заставляет поле выравниваться по левому краю в пределах `width` .
- `>` принудительно выравнивает поле по `width` .
- `^` заставляет поле центрировать по `width` .
- `=` заставляет прокладку помещаться после знака (только для числовых типов).

`fill_char` (если опущен по умолчанию - это пробел) - это символ, используемый для заполнения.

```
'{:~<9s}, World'.format('Hello')
# 'Hello~~~~, World'

'{:~>9s}, World'.format('Hello')
# '~~~~Hello, World'

'{:~^9s}'.format('Hello')
# '~~Hello~~'

'{:0=6d}'.format(-123)
# '-00123'
```

Примечание: вы можете добиться тех же результатов, используя строковые функции `ljust()` , `rjust()` , `center()` , `zfill()` , однако эти функции устарели с версии 2.5.

Форматирование литералов (f-строка)

Строки с литеральным форматированием были введены в PEP 498 (Python3.6 и выше), что позволяет вам добавить `f` к началу строкового литерала, чтобы эффективно применять `.format` к нему со всеми переменными в текущей области.

```
>>> foo = 'bar'
>>> f'Foo is {foo}'
'Foo is bar'
```

Это также работает с более продвинутыми строками формата, включая выравнивание и точечную нотацию.

```
>>> f'{foo:^7s}'
'  bar  '
```

Примечание: `f''` не обозначает конкретный тип типа `b''` для `bytes` или `u''` для `unicode` в python2. Формирование немедленно применяется, что приводит к нормальному перемешиванию.

Строки формата также могут быть *вложенными* :

```
>>> price = 478.23
>>> f"{f'${price:0.2f}':*>20s}"
'*************$478.23'
```

Выражения в f-строке оцениваются в порядке слева направо. Это можно обнаружить только в том случае, если выражения имеют побочные эффекты:

```
>>> 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]
```

Форматирование строк с датой

Любой класс может настроить свой собственный синтаксис форматирования строк с помощью метода `__format__` . Тип стандартной библиотеки Python, который позволяет это

использовать, - это тип `datetime` , где можно использовать коды форматирования в `str.format` `strftime` непосредственно в `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.'
```

Полный список списков форматов даты и времени можно найти в официальной документации .

Формат с использованием Getitem и Getattr

Любая структура данных, поддерживающая `__getitem__` может иметь `__getitem__` структуру вложенных структур:

```
person = {'first': 'Arthur', 'last': 'Dent'}
'{p[first]} {p[last]}'.format(p=person)
# 'Arthur Dent'
```

Доступ к объектным атрибутам можно получить с помощью `getattr()` :

```
class Person(object):
    first = 'Zaphod'
    last = 'Beeblebrox'

'{p.first} {p.last}'.format(p=Person())
# 'Zaphod Beeblebrox'
```

Форматирование поплавка

```
>>> '{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'
```

То же самое относится и к другим способам ссылок:

```
>>> '{:.3f}'.format(42.12345)
'42.123'

>>> '{answer:.3f}'.format(answer=42.12345)
```

```
'42.123'
```

Числа с плавающей точкой также могут быть отформатированы в научной нотации или в процентах:

```
>>> '{0:.3e}'.format(42.12345)
'4.212e+01'

>>> '{0:.0%}'.format(42.12345)
'4212%'
```

Вы также можете комбинировать обозначения `{0}` и `{name}` . Это особенно полезно, если вы хотите округлить все переменные до заданного числа десятичных знаков *с 1 объявлением* :

```
>>> 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'
```

Форматирование числовых значений

Метод `.format()` может интерпретировать число в разных форматах, например:

```
>>> '{:c}'.format(65)    # Unicode character
'A'

>>> '{:d}'.format(0x0a)  # base 10
'10'

>>> '{:n}'.format(0x0a)  # base 10 using current locale for separators
'10'
```

Форматировать целые числа в разные базы (hex, oct, binary)

```
>>> '{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'
```

Используйте форматирование для преобразования кортежа RGB в шестнадцатеричную строку цвета:

```
>>> r, g, b = (1.0, 0.4, 0.0)
>>> '#{:02X}{:02X}{:02X}'.format(int(255 * r), int(255 * g), int(255 * b))
'#FF6600'
```

Только целые числа могут быть преобразованы:

```
>>> '{: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'
```

Пользовательское форматирование для класса

Замечания:

> Все ниже применимо к методу `str.format` , а также к функции `format` . В тексте ниже они взаимозаменяемы.

Для каждого значения, которое передается функции `format` , Python ищет метод `__format__` для этого аргумента. Поэтому Ваш собственный пользовательский класс может иметь свой собственный `__format__` метод определения , как `format` функция будет отображать и форматировать свой класс , и это атрибуты.

Это отличается от метода `__str__` , так как в методе `__format__` вы можете учитывать язык форматирования, включая выравнивание, ширину поля и т. Д. И даже (если хотите) реализовать свои собственные спецификаторы формата и собственные расширения языка форматирования. 1

```
object.__format__(self, format_spec)
```

Например :

```
# 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.
```

Замечания:

> Если у вашего пользовательского класса нет настраиваемого метода `__format__` и экземпляр класса передается в функцию `format` , **Python2** всегда будет использовать возвращаемое значение метода `__str__` или метод `__repr__` чтобы определить, что печатать (и если они не существуют, то по умолчанию `repr` будет использоваться), и вам нужно будет использовать `s` спецификатор формата в формат этого. С **Python3** , чтобы передать ваш пользовательский класс в функцию `format` , вам нужно будет определить метод `__format__` в вашем пользовательском классе.

Вложенное форматирование

Некоторые форматы могут принимать дополнительные параметры, такие как ширина форматированной строки или выравнивание:

```
>>> '{:.>10}'.format('foo')
'.......foo'
```

Они также могут предоставляться как параметры для `format` , вложенные больше `{}` внутри `{}` :

```
>>> '{:.>{}}'.format('foo', 10)
'.......foo'
'{:{}{}{}}'.format('foo', '*', '^', 15)
'******foo******'
```

В последнем примере строка форматирования `'{:{}{}{}}'` изменена на `'{:*^15}'` (т.е. «центр и пэд с * на общую длину 15»), прежде чем применять его к фактическая строка `'foo'` будет отформатирована таким образом.

Это может быть полезно в случаях, когда параметры не известны заранее, для случаев, когда выравнивание табличных данных:

```
>>> data = ["a", "bbbbbbb", "ccc"]
>>> m = max(map(len, data))
```

```
>>> for d in data:
...     print('{:>{}}'.format(d, m))
      a
bbbbbbb
    ccc
```

Заполнение и усечение строк, комбинированных

Предположим, вы хотите печатать переменные в столбце из 3 символов.

Примечание: удвоение `{` и `}` ускользает от них.

```
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))
```

Выход:

```
pad
{:3}             :1  :

truncate
{:.3}            :555:

combined
{:>3.3}          :  1:
{:3.3}           :1  :
{:3.3}           :333:
{:3.3}           :555:
```

Именованные заполнители

Строки формата могут содержать именованные заполнители, которые интерполируются с использованием аргументов ключевого слова для `format` .

Использование словаря (Python 2.x)

```
>>> data = {'first': 'Hodor', 'last': 'Hodor!'}
>>> '{first} {last}'.format(**data)
'Hodor Hodor!'
```

Использование словаря (Python 3.2+)

```
>>> '{first} {last}'.format_map(data)
'Hodor Hodor!'
```

`str.format_map` позволяет использовать словари, не распаковывая их сначала. Также класс `data` (который может быть нестандартным) используется вместо нового заполненного `dict` .

Без словаря:

```
>>> '{first} {last}'.format(first='Hodor', last='Hodor!')
'Hodor Hodor!'
```

Прочитайте Форматирование строк онлайн: https://riptutorial.com/ru/python/topic/1019/форматирование-строк

глава 197: функции

Вступление

Функции в Python предоставляют организованный, многоразовый и модульный код для выполнения ряда конкретных действий. Функции упрощают процесс кодирования, предотвращают избыточную логику и упрощают выполнение кода. В этом разделе описывается декларация и использование функций в Python.

Python имеет множество *встроенных функций,* таких как `print()` , `input()` , `len()` . Помимо встроенных модулей вы также можете создавать свои собственные функции для выполнения более конкретных заданий - это называемые *пользовательскими функциями* .

Синтаксис

- def *function_name* (*arg1, ... argN, * args, kw1, kw2 = default, ..., ** kwargs*): *statements*
- lambda *arg1, ... argN, * args, kw1, kw2 = default, ..., ** kwargs* : *выражение*

параметры

параметр	подробности
arg1 , ..., *argN*	Обычные аргументы
* *apr*	Без названия позиционные аргументы
kw1 , ..., *kwN*	Ключ-только аргументы
** *kwargs*	Остальные аргументы ключевого слова

замечания

5 основных вещей, которые вы можете выполнять с функциями:

- Назначение функций переменным

```
def f():
  print(20)
y = f
y()
# Output: 20
```

- Определить функции внутри других функций (вложенные функции)

```
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
```

- Функции могут возвращать другие функции

```
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
```

- Функции могут передаваться как параметры для других функций

```
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
```

- Внутренние функции имеют доступ к охватывающей области (Closure)

```
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!
```

Дополнительные ресурсы

- Подробнее о функциях и декораторах: https://www.thecodeship.com/patterns/guide-to-python-function-decorators/

Examples

Определение и вызов простых функций

Использование инструкции `def` является наиболее распространенным способом определения функции в python. Этот оператор представляет собой так называемый *составной оператор single clause* со следующим синтаксисом:

```
def function_name(parameters):
    statement(s)
```

`function_name` известно как *идентификатор* функции. Поскольку определение функции является исполняемым оператором, его выполнение *связывает* имя функции с функциональным объектом, который может быть вызван позже при использовании идентификатора.

`parameters` - это необязательный список идентификаторов, которые привязаны к значениям, указанным в качестве аргументов при вызове функции. Функция может иметь произвольное число аргументов, разделенных запятыми.

`statement(s)` - также известный как *тело функции* - это непустая последовательность операторов, выполняемых каждый раз при вызове функции. Это означает, что тело функции не может быть пустым, как и любой *отложенный блок* .

Вот пример простого определения функции, целью которого является печать `Hello` каждый раз, когда он вызывается:

```
def greet():
    print("Hello")
```

Теперь давайте назовем определенную функцию `greet()` :

```
greet()
# Out: Hello
```

Это еще один пример определения функции, который принимает один единственный аргумент и отображает переданное значение при каждом вызове функции:

```
def greet_two(greeting):
    print(greeting)
```

После этого `greet_two()` должна вызываться с аргументом:

```
greet_two("Howdy")
# Out: Howdy
```

Также вы можете указать значение по умолчанию для этого аргумента функции:

```
def greet_two(greeting="Howdy"):
    print(greeting)
```

Теперь вы можете вызвать функцию без значения:

```
greet_two()
# Out: Howdy
```

Вы заметите, что в отличие от многих других языков вам не нужно явно объявлять возвращаемый тип функции. Функции Python могут возвращать значения любого типа с

помощью ключевого слова `return` . Одна функция может возвращать любое количество различных типов!

```
def many_types(x):
    if x < 0:
        return "Hello!"
    else:
        return 0

print(many_types(1))
print(many_types(-1))

# Output:
0
Hello!
```

Пока это правильно обрабатывается вызывающим, это вполне допустимый код Python.

Функция, которая достигает конца выполнения без оператора return, всегда будет возвращать `None` :

```
def do_nothing():
    pass

print(do_nothing())
# Out: None
```

Как упоминалось ранее, определение функции должно иметь тело функции, непустую последовательность операторов. Поэтому инструкция `pass` используется как тело функции, которая является пустой операцией - когда она выполняется, ничего не происходит. Он делает то, что это значит, он пропускает. Он полезен в качестве заполнителя, если оператор требуется синтаксически, но код не должен выполняться.

Возвращаемые значения из функций

Функции могут `return` значение, которое вы можете использовать напрямую:

```
def give_me_five():
    return 5

print(give_me_five())  # Print the returned value
# Out: 5
```

или сохранить значение для последующего использования:

```
num = give_me_five()
print(num)             # Print the saved returned value
# Out: 5
```

или использовать значение для любых операций:

```
print(give_me_five() + 10)
# Out: 15
```

Если в функции встречается `return` функция будет немедленно выведена, и последующие операции не будут оцениваться:

```
def give_me_another_five():
    return 5
    print('This statement will not be printed. Ever.')

print(give_me_another_five())
# Out: 5
```

Вы также можете `return` несколько значений (в виде кортежа):

```
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
```

Функция *без* оператора `return` неявно возвращает `None` . Аналогично функция с оператором `return` , но никакое возвращаемое значение или переменная не возвращает `None` .

Определение функции с аргументами

Аргументы определяются в круглых скобках после имени функции:

```
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)
```

Имя функции и ее список аргументов называются *сигнатурой* функции. Каждый именованный аргумент является фактически локальной переменной функции.

При вызове функции дайте значения для аргументов, указав их в порядке

```
divide(10, 2)
# output: 5
```

или указать их в любом порядке, используя имена из определения функции:

```
divide(divisor=2, dividend=10)
# output: 5
```

Определение функции с необязательными аргументами

Необязательные аргументы могут быть определены путем присвоения (using `=`) значения по умолчанию для имени-аргумента:

```
def make(action='nothing'):
    return action
```

Вызов этой функции возможен тремя различными способами:

```
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
```

Предупреждение

Уменяемые типы (`list` , `dict` , `set` и т. Д.) Следует относиться с осторожностью, когда они заданы как атрибут по **умолчанию** . Любая мутация аргумента по умолчанию изменяет его навсегда. См. Раздел Определение функции с необязательными изменяемыми аргументами .

Определение функции с несколькими аргументами

Можно дать функцию столько аргументов, сколько нужно, единственными фиксированными правилами являются то, что каждое имя аргумента должно быть уникальным, а необязательные аргументы должны быть после необязательных:

```
def func(value1, value2, optionalvalue=10):
    return '{0} {1} {2}'.format(value1, value2, optionalvalue1)
```

При вызове функции вы можете либо указать каждое ключевое слово без имени, но затем порядок:

```
print(func(1, 'a', 100))
# Out: 1 a 100

print(func('abc', 14))
# abc 14 10
```

Или объедините аргументы с именем и без. Тогда те, у кого есть имя, должны следовать за ними, но порядок имен с именем не имеет значения:

```
print(func('This', optionalvalue='StackOverflow Documentation', value2='is'))
# Out: This is StackOverflow Documentation
```

Определение функции с произвольным числом аргументов

Произвольное число позиционных аргументов:

Определение функции, способной принимать произвольное количество аргументов, может быть выполнено путем префикса одного из аргументов с помощью `*`

```
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
```

Вы **не можете** предоставить значение по умолчанию для `args` , например `func(*args=[1, 2, 3])` повысит синтаксическую ошибку (даже не компилируется).

Вы **не можете** предоставить их по имени при вызове функции, например `func(*args=[1, 2, 3])` приведет к созданию `TypeError` .

Но если у вас уже есть аргументы в массиве (или любой другой `Iterable`), вы **можете** вызвать свою функцию следующим образом: `func(*my_stuff)` .

К этим аргументам (`*args`) можно обращаться по индексу, например `args[0]` вернет первый аргумент

Произвольное количество аргументов ключевого слова

Вы можете взять произвольное количество аргументов с именем, определив аргумент в определении с **двумя** `*` перед ним:

```
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
```

Вы **не можете** предоставить их **без** имен, например `func(1, 2, 3)` поднимет `TypeError` .

`kwargs` - простой родной словарь python. Например, `args['value1']` даст значение для аргумента `value1` . Обязательно проверьте заранее, что есть такой аргумент или `KeyError` будет поднят.

Предупреждение

Вы можете смешивать их с другими необязательными и необходимыми аргументами, но порядок в определении имеет значение.

Первыми должны быть аргументы **positional / keyword** . (Необходимые аргументы).
Затем идут **произвольные** аргументы `*arg` . (Необязательный).
Затем следуют аргументы **только для ключевого слова** . (Необходимые).
Наконец, возникает **произвольное ключевое слово** `**kwargs` . (Необязательный).

```
#       |-positional-|-optional-|---keyword-only--|-optional-|
def func(arg1, arg2=10 , *args, kwarg1, kwarg2=2, **kwargs):
     pass
```

- `arg1` , в противном случае возникает `TypeError` . Он может быть задан как позиционный (`func(10)`) или аргумент ключевого слова (`func(arg1=10)`).
- `kwarg1` также должен быть указан, но он может быть предоставлен только в качестве ключевого слова: `func(kwarg1=10)` .
- `arg2` и `kwarg2` являются необязательными. Если значение должно быть изменено, применяются те же правила, что и для `arg1` (либо позиционный, либо ключевое слово), либо `kwarg1` (только ключевое слово).
- `*args`

ловит дополнительные позиционные параметры. Но учтите, что `arg1` и `arg2` должны быть представлены как позиционные аргументы для передачи аргументов в `*args` : `func(1, 1, 1, 1)` .

- `**kwargs` ловит все дополнительные параметры ключевых слов. В этом случае любой параметр, который не является `arg1` , `arg2` , `kwarg1` или `kwarg2` . Например: `func(kwarg3=10)` .
- В Python 3 вы можете использовать `*` самостоятельно, чтобы указать, что все последующие аргументы должны быть указаны как ключевые слова. Например, функция `math.isclose` в Python 3.5 и выше определяется с помощью `def math.isclose (a, b, *, rel_tol=1e-09, abs_tol=0.0)` , что означает, что первые два аргумента могут быть поставлены позиционно, но необязательный третий и четвертый параметры могут поставляться только в качестве аргументов ключевого слова.

Python 2.x не поддерживает параметры только для ключевого слова. Такое поведение можно эмулировать с помощью `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 ...
```

Примечание о присвоении имен

Условность наименования необязательных позиционных аргументов `args` и необязательных аргументы ключевых слов `kwargs` просто условность вы **можете** использовать любые имена вам нравится , **но** это полезно следовать соглашению , чтобы другие знали , что вы делаете, *или даже сам потом* так пожалуйста.

Примечание о уникальности

Любая функция может быть определена без **одного или одного** `*args` и **ни одного, ни одного** `**kwargs` но не с более чем одним из них. Также `*args` **должен** быть последним позиционным аргументом, а `**kwargs` должен быть последним параметром. Попытка использовать более одного из них **приведет** к исключению ошибки синтаксиса.

Примечание о функциях вложенности с необязательными аргументами

Можно использовать такие функции, и обычным соглашением является удаление элементов, которые уже обработал код, **но** если вы передаете параметры, вам необходимо

передать необязательные позиционные аргументы с префиксом `*` и необязательными ключевыми словами args с префиксом `**` , иначе args передаются как список или кортеж, а kwargs - как один словарь. например:

```
def fn(**kwargs):
    print(kwargs)
    f1(**kwargs)

def f1(**kwargs):
    print(len(kwargs))

fn(a=1, b=2)
# Out:
# {'a': 1, 'b': 2}
# 2
```

Определение функции с необязательными изменяемыми аргументами

Существует проблема при использовании **необязательных аргументов** с **изменяемым типом по умолчанию** (описанным в разделе «Определение функции с необязательными аргументами»), что потенциально может привести к неожиданному поведению.

объяснение

Эта проблема возникает из-за того, что аргументы по умолчанию функции инициализируются **один раз** , в момент, когда функция *определена* , а **не** (как и многие другие языки) при *вызове* функции. Значения по умолчанию хранятся внутри переменной-члена `__defaults__` объекта функции.

```
def f(a, b=42, c=[]):
    pass

print(f.__defaults__)
# Out: (42, [])
```

Для **неизменяемых** типов (см. Прохождение аргументации и изменчивость) это не проблема, потому что нет способа изменить эту переменную; его можно только переназначить, оставив исходное значение без изменений. Следовательно, последующие гарантированно имеют одинаковое значение по умолчанию. Однако для **изменяемого** типа исходное значение может мутировать, выполняя вызовы его различных функций-членов. Поэтому для последовательных вызовов функции не гарантируется первоначальное значение по умолчанию.

```
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]
```

> **Примечание.** Некоторые IDE, такие как PyCharm, выдают предупреждение, если в качестве атрибута по умолчанию указан изменяемый тип.

Решение

Если вы хотите, чтобы аргумент по умолчанию всегда был тем, который вы указываете в определении функции, тогда решение **всегда** должно использовать неизменяемый тип в качестве аргумента по умолчанию.

Общей идиомой для достижения этого, когда изменяемый тип необходим по умолчанию, заключается в том, чтобы использовать `None` (immutable) в качестве аргумента по умолчанию, а затем присваивать фактическое значение по умолчанию переменной аргумента, если оно равно `None` .

```
def append(elem, to=None):
    if to is None:
        to = []

    to.append(elem)
    return to
```

Функции Lambda (Inline / Anonymous)

Ключевое слово `lambda` создает встроенную функцию, содержащую одно выражение. Значение этого выражения - это то, что функция возвращает при вызове.

Рассмотрим функцию:

```
def greeting():
    return "Hello"
```

который, когда он называется:

```
print(greeting())
```

печатает:

```
Hello
```

Это можно записать в виде лямбда-функции следующим образом:

```
greet_me = lambda: "Hello"
```

> См. Примечание в нижней части этого раздела о назначении лямбда для переменных. Как правило, не делайте этого.

Это создает встроенную функцию с именем `greet_me` которое возвращает `Hello` . Обратите внимание, что вы не записываете `return` при создании функции с лямбдой. Значение после `:` автоматически возвращается.

После присвоения переменной она может использоваться как обычная функция:

```
print(greet_me())
```

печатает:

```
Hello
```

`lambda` s также может принимать аргументы:

```
strip_and_upper_case = lambda s: s.strip().upper()

strip_and_upper_case("  Hello   ")
```

возвращает строку:

```
HELLO
```

Они также могут принимать произвольное количество аргументов / аргументов ключевого слова, таких как обычные функции.

```
greeting = lambda x, *args, **kwargs: print(x, args, kwargs)
greeting('hello', 'world', world='world')
```

печатает:

```
hello ('world',) {'world': 'world'}
```

`lambda` s обычно используются для коротких функций, которые удобно определять в том месте, где они вызываются (обычно с `sorted` , `filter` и `map`).

Например, эта строка сортирует список строк, игнорируя их случай и игнорируя пробелы в начале и в конце:

```
sorted( [" foo ", "    bAR", "BaZ    "], key=lambda s: s.strip().upper())
# Out:
# ['    bAR', 'BaZ    ', ' foo ']
```

Список сортировки просто игнорирует пробелы:

```
sorted( [" foo ", "    bAR", "BaZ    "], key=lambda s: s.strip())
# Out:
# ['BaZ    ', '    bAR', ' foo ']
```

Примеры с `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']
```

Примеры с числовыми списками:

```
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]
```

Можно вызвать другие функции (с / без аргументов) изнутри лямбда-функции.

```
def foo(msg):
    print(msg)

greet = lambda x = "hello world": foo(x)
greet()
```

печатает:

```
hello world
```

Это полезно, потому что `lambda` может содержать только одно выражение и с помощью вспомогательной функции можно запускать несколько операторов.

НОТА

Имейте в виду, что PEP-8 (официальное руководство по стилю Python) не рекомендует назначать lambdas для переменных (как это было в первых двух примерах):

> Всегда используйте инструкцию def вместо оператора присваивания, который привязывает лямбда-выражение непосредственно к идентификатору.
>
> Да:
>
> ```
> def f(x): return 2*x
> ```
>
> Нет:
>
> ```
> f = lambda x: 2*x
> ```
>
> Первая форма означает, что имя результирующего функционального объекта является специально `f` вместо общего `<lambda>` . Это более полезно для отслеживания и представления строк в целом. Использование оператора присваивания исключает единственное преимущество, которое лямбда-выражение может предлагать в явном выражении `def` (т. Е. Что оно может быть встроено в большее выражение).

Прохождение и изменчивость аргумента

Во-первых, некоторая терминология:

- **аргумент (*фактический* параметр):** фактическая переменная передается функции;
- **параметр (*формальный* параметр):** принимающая переменная, которая используется в функции.

В Python аргументы передаются *присваиванием* (в отличие от других языков, где аргументы могут передаваться по значению / ссылке / указателю).

- Мутирование параметра будет мутировать аргумент (если тип аргумента изменен).

```
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
```

- Переназначение параметра не приведет к переназначению аргумента.

```
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]
```

В Python мы фактически не присваиваем значения переменным, вместо этого *связываем* (т.е. присваиваем, присоединяем) переменные (считающиеся *именами*) к объектам.

- **Неизменяемость:** целые числа, строки, кортежи и т. Д. Все операции делают копии.
- **Mutable:** списки, словари, наборы и т. Д. Операции могут или не могут мутировать.

```
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]
```

закрытие

Замыкания в Python создаются вызовами функций. Здесь вызов `makeInc` создает привязку для `x` которую ссылается внутри функции `inc` . Каждый вызов `makeInc` создает новый экземпляр этой функции, но каждый экземпляр имеет ссылку на другую привязку `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
```

Обратите внимание, что при регулярном закрытии закрытая функция полностью наследует все переменные из среды окружения, в этой конструкции закрытая функция имеет только доступ для чтения к унаследованным переменным, но не может назначать им

```
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 предлагает `nonlocal` оператор (нелокальные переменные) для реализации полного закрытия с вложенными функциями.

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
```

Рекурсивные функции

Рекурсивная функция - это функция, которая вызывает себя в своем определении. Например, математическая функция, факториал, определяемая `factorial(n) = n*(n-1)*(n-2)*...*3*2*1` . может быть запрограммирован как

```
def factorial(n):
    #n here should be an integer
    if n == 0:
        return 1
    else:
        return n*factorial(n-1)
```

выходы здесь:

```
factorial(0)
#out 1
factorial(1)
#out 1
factorial(2)
#out 2
factorial(3)
#out 6
```

как и ожидалось. Обратите внимание, что эта функция рекурсивна, потому что второй

`return factorial(n-1)` , где функция вызывает себя в своем определении.

Некоторые рекурсивные функции могут быть реализованы с использованием лямбда , факториальная функция, использующая лямбда, будет примерно такой:

```
factorial = lambda n: 1 if n == 0 else n*factorial(n-1)
```

Функция выводит то же, что и выше.

Предел рекурсии

Существует предел глубине возможной рекурсии, которая зависит от реализации Python. Когда предел достигнут, возникает исключение 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!
```

Можно изменить предел глубины рекурсии, используя `sys.setrecursionlimit(limit)` и проверить этот предел на `sys.getrecursionlimit()` .

```
sys.setrecursionlimit(2000)
cursing(0)
# Out: I recursed 1997 times!
```

Из Python 3.5 исключение представляет собой `RecursionError` , который получен из `RuntimeError` .

Вложенные функции

Функции в python являются первоклассными объектами. Они могут быть определены в любой области

```
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
```

Функции захвата их охватывающей области могут передаваться как любой другой вид объекта

```
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
```

Итерируемая и распаковка словарей

Функции позволяют указывать эти типы параметров: позиционный, именованный, переменный позиционный, аргументы ключевых слов (kwargs). Вот ясное и краткое использование каждого типа.

```
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'
```

Принуждение использования именованных параметров

Все параметры, указанные после первой звездочки в сигнатуре функции, являются ключевыми словами.

```
def f(*a, b):
    pass

f(1, 2, 3)
# TypeError: f() missing 1 required keyword-only argument: 'b'
```

В Python 3 можно поместить единственную звездочку в подпись функции, чтобы гарантировать, что остальные аргументы могут передаваться только с использованием аргументов ключевого слова.

```
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
```

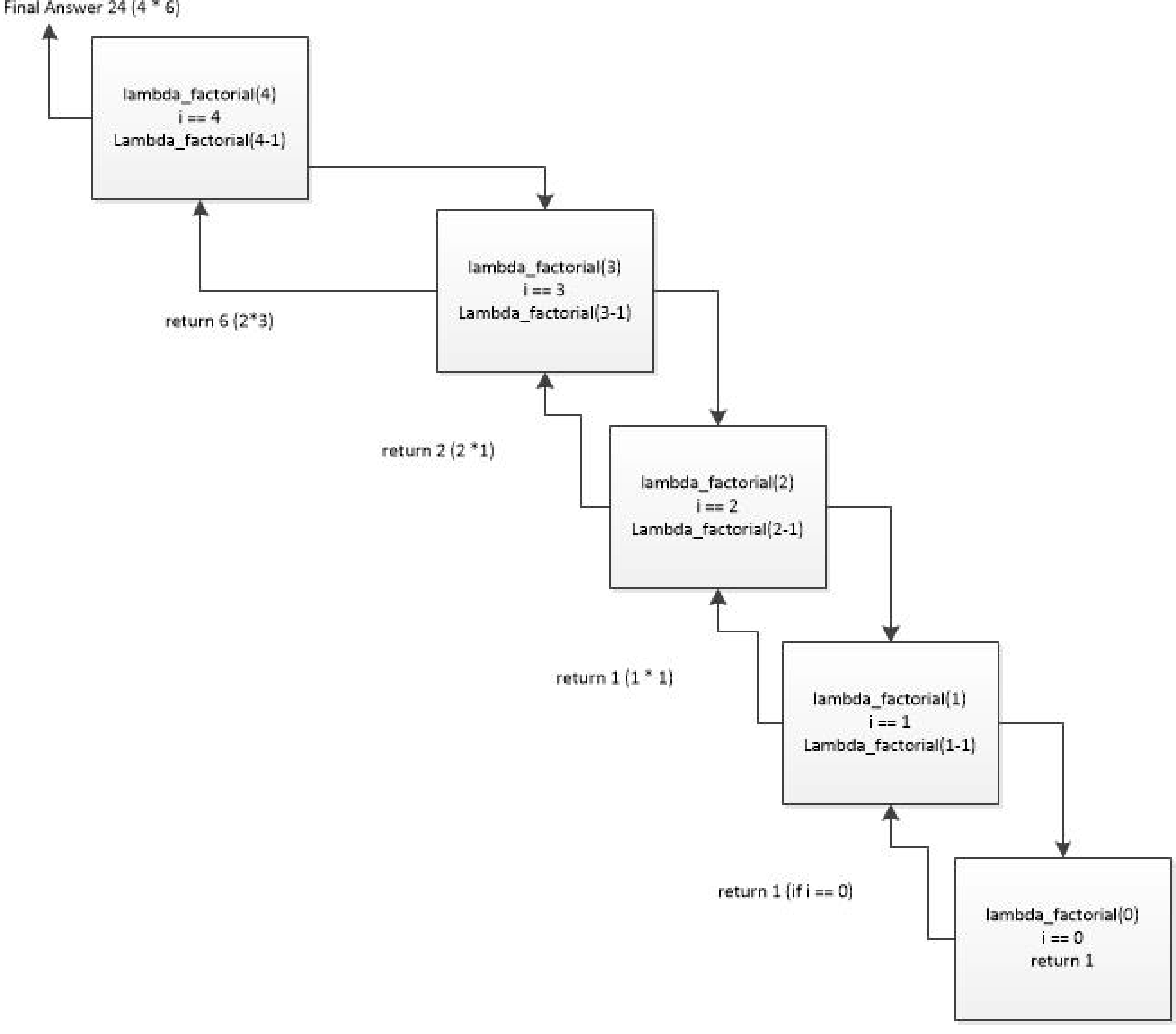
Рекурсивная Лямбда с использованием назначенной переменной

Один из способов создания рекурсивных лямбда-функций заключается в назначении функции переменной и последующем ее привязке к самой функции. Общим примером этого является рекурсивный расчет факториала числа - например, как показано в следующем коде:

```
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
```

Описание кода

Лямбда-функция через назначение переменной передается значение (4), которое оно оценивает и возвращает 1, если оно равно 0, или возвращает текущее значение (`i`) * другое вычисление с помощью лямбда-функции значения - 1 (`i-1`). Это продолжается до тех пор, пока переданное значение не уменьшится до 0 (`return 1`). Процесс, который можно визуализировать как:



Прочитайте функции онлайн: https://riptutorial.com/ru/python/topic/228/функции

глава 198: Функциональное программирование на Python

Вступление

Функциональное программирование разлагает проблему на набор функций. В идеале, функции принимают только входные данные и производят выходные данные и не имеют никакого внутреннего состояния, которое влияет на выход, созданный для данного входа. Ниже приведены функциональные методы, общие для многих языков: например, лямбда, карта, сокращение.

Examples

Лямбда-функция

Анонимная, встроенная функция, определенная с помощью лямбда. Параметры лямбда определены слева от двоеточия. Тело функции определено справа от двоеточия. Результат работы тела функции (неявно) возвращен.

```
s=lambda x:x*x
s(2)    =>4
```

Функция карты

Карта принимает функцию и набор элементов. Он создает новую пустую коллекцию, запускает функцию для каждого элемента в исходной коллекции и вставляет каждое возвращаемое значение в новую коллекцию. Он возвращает новую коллекцию.

Это простая карта, которая берет список имен и возвращает список длин этих имен:

```
name_lengths = map(len, ["Mary", "Isla", "Sam"])
print(name_lengths)    =>[4, 4, 3]
```

Уменьшить функцию

Уменьшение принимает функцию и набор элементов. Он возвращает значение, которое создается путем объединения элементов.

Это простое сокращение. Он возвращает сумму всех элементов в коллекции.

```
total = reduce(lambda a, x: a + x, [0, 1, 2, 3, 4])
print(total)    =>10
```

Функция фильтра

Фильтр принимает функцию и коллекцию. Он возвращает коллекцию каждого элемента, для которого функция вернула True.

```
arr=[1,2,3,4,5,6]
[i for i in filter(lambda x:x>4,arr)]    # outputs[5,6]
```

Прочитайте Функциональное программирование на Python онлайн: https://riptutorial.com/ru/python/topic/9552/функциональное-программирование-на-python

глава 199: Функция карты

Синтаксис

- map (function, iterable [, * Additional_iterables])
- future_builtins.map (function, iterable [, * Additional_iterables])
- itertools.imap (функция, iterable [, * дополнительные_имя)]

параметры

параметр	подробности
функция	функция для отображения (должна принимать столько параметров, сколько есть итераций) (*только для позиции*)
итерируемый	функция применяется к каждому элементу итерабельного (*только для позиции*)
* additional_iterables	см. iterable, но сколько угодно (*необязательно* , *только для позиции*)

замечания

Все, что можно сделать с помощью `map` также можно сделать с `comprehensions` :

```
list(map(abs, [-1,-2,-3]))    # [1, 2, 3]
[abs(i) for i in [-1,-2,-3]]  # [1, 2, 3]
```

Хотя вам понадобится `zip` если у вас много итераций:

```
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]
```

Учет списков эффективен и может быть быстрее, чем `map` во многих случаях, поэтому проверяйте время обоих подходов, если скорость важна для вас.

Examples

Основное использование карты, itertools.imap и future_builtins.map

Функция map является самой простой среди встроенных модулей Python, используемых для функционального программирования. `map()` применяет указанную функцию к каждому элементу в итерируемой:

```
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>
```

3-совместимая `map` Python включена в модуль `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>
```

В качестве альтернативы, в Python 2 можно использовать `imap` из `itertools` для получения генератора

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>
```

Результат может быть явно преобразован в `list` чтобы удалить различия между Python 2 и 3:

```
list(map(len, names))
# Out: [4, 5, 6]
```

`map()` можно заменить эквивалентным *пониманием списка* или *выражением генератора* :

```
[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>
```

Отображение каждого значения в итерабельном

Например, вы можете принять абсолютное значение каждого элемента:

```
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]
```

Анонимная функция также поддерживает отображение списка:

```
map(lambda x:x*2, [1, 2, 3, 4, 5])
# Out: [2, 4, 6, 8, 10]
```

или преобразование десятичных значений в проценты:

```
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]
```

или конвертации долларов в евро (с учетом обменного курса):

```
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` - удобный способ исправить параметры функций, чтобы их можно было использовать с `map` вместо использования `lambda` или создания настраиваемых функций.

Отображение значений разных итераций

Например, вычисляя среднее значение каждого `i` элемента из множества итераций:

```
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]
```

Существуют разные требования, если более чем один итерабельный передается на `map` зависимости от версии python:

- Функция должна принимать столько параметров, сколько есть итераций:

  ```
  def median_of_three(a, b, c):
  ```

```
    return sorted((a, b, c))[1]

list(map(median_of_three, measurement1, measurement2))
```

> TypeError: median_of_three () отсутствует 1 требуемый позиционный аргумент: 'c'

```
list(map(median_of_three, measurement1, measurement2, measurement3, measurement3))
```

> TypeError: median_of_three () принимает 3 позиционных аргумента, но 4 даны

Python 2.x 2.0.1

- `map` : отображение повторяется до тех пор, пока один итерабель еще не полностью поглощен, но принимает значение `None` из полностью потребляемых итераций:

```
import operator

measurement1 = [100, 111, 99, 97]
measurement2 = [102, 117]

# Calculate difference between elements
list(map(operator.sub, measurement1, measurement2))
```

> TypeError: неподдерживаемый тип операндов для -: 'int' и 'NoneType'

- `itertools.imap` и `future_builtins.map` : Отображение останавливается, как только один итеративный останавливается:

```
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

- Отображение останавливается, как только один истребитель останавливается:

```
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]
```

Транспонирование с помощью карты: использование параметра «Нет» в качестве аргумента функции (только для 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: объект «NoneType» не может быть вызван

Но есть обходное решение для получения аналогичных результатов:

```
def conv_to_list(*args):
    return list(args)

list(map(conv_to_list, *image))
# Out: [[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

Серия и параллельное сопоставление

map () - это встроенная функция, что означает, что она доступна повсюду, без необходимости использовать оператор import. Он доступен везде, как print () Если вы посмотрите пример 5, вы увидите, что мне пришлось использовать оператор импорта, прежде чем я смогу использовать довольно печатную (import pprint). Таким образом, pprint

не является встроенной функцией

Отображение серии

В этом случае каждый аргумент итерабельности передается в качестве аргумента функции отображения в порядке возрастания. Это возникает, когда у нас есть только одна итерабельность для отображения, а для функции отображения требуется один аргумент.

Пример 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
```

приводит к

```
['fly is an insect', 'ant is an insect', 'beetle is an insect', 'cankerworm is an insect']
```

Пример 2.

```
print(list(map(len, insects))) # the len function is executed each item in the insect list
```

приводит к

```
[3, 3, 6, 10]
```

Параллельное отображение

В этом случае каждый аргумент функции сопоставления выталкивается из всех итераций (по одному от каждого итеративного) параллельно. Таким образом, количество предоставленных итераций должно соответствовать количеству аргументов, требуемых функцией.

```
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)
```

Пример 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)))
```

приводит к

```
TypeError: len() takes exactly one argument (4 given)
```

Пример 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)))
```

приводит к

```
TypeError: animals() missing 3 required positional arguments: 'x', 'y', and 'z'
```

Пример 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)))
```

приводит к

```
 ['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']
```

Прочитайте Функция карты онлайн: https://riptutorial.com/ru/python/topic/333/функция-карты

глава 200: Функция печати

Examples

Основы печати

В Python 3 и выше `print` - это функция, а не ключевое слово.

```
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
```

Вы также можете передать несколько параметров для `print` :

```
print(foo, bar, baz)
# out: 1 bar 3.14
```

Другой способ `print` нескольких параметров - использовать `+`

```
print(str(foo) + " " + bar + " " + str(baz))
# out: 1 bar 3.14
```

Однако вы должны быть осторожны при использовании `+` для печати нескольких параметров, так как тип параметров должен быть одинаковым. Попытка напечатать приведенный выше пример без приведения в `string` сначала приведет к ошибке, потому что она попытается добавить номер `1` в строку `"bar"` и добавить это к числу `3.14` .

```
# Wrong:
# type:int  str  float
print(foo + bar + baz)
# will result in an error
```

Это связано с тем, что сначала будет оцениваться содержимое `print` :

```
print(4 + 5)
# out: 9
print("4" + "5")
# out: 45
print([4] + [5])
```

```
# out: [4, 5]
```

В противном случае использование `+` может быть очень полезно для пользователя для чтения вывода переменных. В приведенном ниже примере вывод очень прост для чтения!

В приведенном ниже сценарии это

```
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))
```

Вы можете запретить автоматическую `print` функции печати с помощью параметра `end` :

```
print("this has no newline at the end of it... ", end="")
print("see?")
# out: this has no newline at the end of it... see?
```

Если вы хотите записать файл, вы можете передать его как `file` параметров:

```
with open('my_file.txt', 'w+') as my_file:
    print("this goes to the file!", file=my_file)
```

> это идет к файлу!

Параметры печати

Вы можете делать больше, чем просто печатать текст. `print` также есть несколько параметров, которые помогут вам.

Аргумент `sep` : поместите строку между аргументами.

Вам нужно распечатать список слов, разделенных запятой или какой-либо другой строкой?

```
>>> print('apples','bannas', 'cherries', sep=', ')
apple, bannas, cherries
>>> print('apple','banna', 'cherries', sep=', ')
apple, banna, cherries
>>>
```

Аргумент `end` : используйте что-то другое, кроме новой строки в конце

Без аргумента `end` все функции `print()` записывают строку, а затем переходят к началу следующей строки. Вы можете изменить его, чтобы ничего не делать (используйте пустую строку '') или двойной интервал между абзацами, используя две новые строки.

```
>>> print("<a", end=''); print(" class='jidn'" if 1 else "", end=''); print("/>")
<a class='jidn'/>
```

```
>>> print("paragraph1", end="\n\n"); print("paragraph2")
paragraph1

paragraph2
>>>
```

`file` аргумента: отправьте вывод в другое место, кроме sys.stdout.

Теперь вы можете отправить свой текст в stdout, файл или StringIO и не заботятся о том, что вам дано. Если он кланяется как файл, он работает как файл.

```
>>> 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

>>>
```

Существует четвертый параметр `flush` который будет принудительно очищать поток.

Прочитайте Функция печати онлайн: https://riptutorial.com/ru/python/topic/1360/функция-печати

глава 201: Частичные функции

Вступление

Поскольку вы, вероятно, знаете, пришли ли вы из школы ООП, специализируясь на абстрактном классе и используете его, вам следует помнить при написании кода.

Что делать, если вы можете определить абстрактную функцию и специализироваться на ее создании? Думает, что это своего рода *функция Inheritance,* где вы связываете определенные параметры, чтобы сделать их надежными для конкретного сценария.

Синтаксис

- partial (function, ** params_you_want_fix)

параметры

Param	подробности
Икс	номер, который должен быть поднят
Y	экспонента
повышение	функция, которая будет специализированной

замечания

Как указано в документе Python, *functools.partial* :

> Верните новый частичный объект, который при вызове будет вести себя как func, вызываемый с аргументами аргументов positional аргументов и ключевых слов. Если к вызову добавлено больше аргументов, они добавляются к args. Если предоставляются дополнительные аргументы ключевых слов, они расширяют и переопределяют ключевые слова.

Проверьте эту ссылку, чтобы узнать, как *частично* можно реализовать.

Examples

Повысить мощность

Предположим, мы хотим поднять *x* на число *y* .

Вы должны написать это как:

```
def raise_power(x, y):
    return x**y
```

Что делать, если ваше значение *y* может принимать конечный набор значений?

Предположим, что *y* может быть одним из *[3,4,5],* и, допустим, вы не хотите предлагать конечным пользователям возможность использовать такую функцию, поскольку она очень интенсивно вычисляется. Фактически, вы должны проверить, если при условии, что *y* принимает действительное значение и переписывает вашу функцию как:

```
def raise(x, y):
    if y in (3,4,5):
        return x**y
    raise NumberNotInRangeException("You should provide a valid exponent")
```

Беспорядочный? Давайте используем абстрактную форму и специализируемся на всех трех случаях: давайте их **частично** реализовать.

```
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)
```

Что здесь происходит? Мы зафиксировали *y-* параметры и определили три различные функции.

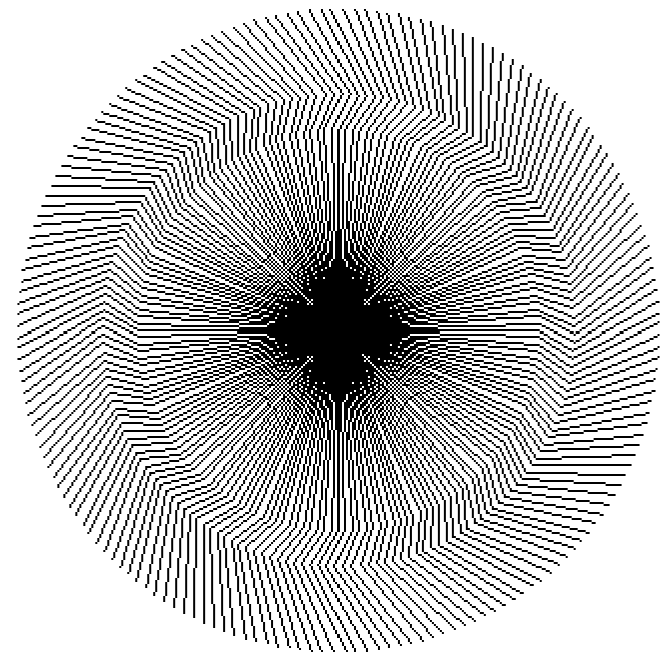
Не нужно использовать абстрактную функцию, определенную выше (вы можете сделать ее *закрытой*), но вы можете использовать **частично прикладные** функции для решения вопроса о повышении числа до фиксированного значения.

Прочитайте Частичные функции онлайн: https://riptutorial.com/ru/python/topic/9383/частичные-функции

глава 202: Черепашья графика

Examples

Ninja Twist (черепаховая графика)



Здесь черепаховая графика ниндзя 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()
```

Прочитайте Черепашья графика онлайн: https://riptutorial.com/ru/python/topic/7915/черепашья-графика

глава 203: Чтение и запись CSV

Examples

Запись файла TSV

питон

```
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'])
```

Выходной файл

```
$ cat /tmp/output.tsv

name    field
Dijkstra    Computer Science
Shelah    Math
Aumann    Economic Sciences
```

Использование панд

Напишите CSV-файл из файла `dict` или `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")
```

Прочтите CSV-файл как `DataFrame` и преобразуйте его в `dict` :

```
df = pd.read_csv("data.csv")
d = df.to_dict()
```

Прочитайте Чтение и запись CSV онлайн: https://riptutorial.com/ru/python/topic/2116/чтение-и-запись-csv

глава 204: Шаблоны в python

Examples

Простая программа вывода данных с использованием шаблона

```
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))
```

Выход:

```
Simon bought 2 candy for 8 dollar
```

Шаблоны поддерживают замену на основе $ вместо замены на основе%. **Заменить** (сопоставление, ключевые слова) выполняет замену шаблона, возвращая новую строку.

Отображение - любой объект, похожий на словарь, с ключами, которые соответствуют шаблонам-заполнителям. В этом примере цена и количество являются заполнителями. Аргументы ключевого слова также могут использоваться в качестве заполнителей. Заполнители из ключевых слов имеют приоритет, если они присутствуют.

Изменение разделителя

Вы можете изменить разделитель «$» на любой другой. Следующий пример:

```
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))
```

Вы можете прочитать здесь документы

Прочитайте Шаблоны в python онлайн: https://riptutorial.com/ru/python/topic/6029/шаблоны-в-python

глава 205: Шаблоны проектирования

Вступление

Шаблон проектирования является общим решением общей проблемы в разработке программного обеспечения. Этот раздел документации специально предназначен для предоставления примеров общих шаблонов проектирования в Python.

Examples

Шаблон стратегии

Этот шаблон дизайна называется Strategy Pattern. Он используется для определения семейства алгоритмов, инкапсулирует каждый из них и делает их взаимозаменяемыми. Шаблон разработки стратегии позволяет алгоритму варьироваться независимо от клиентов, которые его используют.

Например, животные могут «ходить» разными способами. Хождение можно рассматривать как стратегию, реализуемую различными видами животных:

```
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))
```

Выполнение этого примера приведет к следующему результату:

```
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
```

Обратите внимание, что в таких языках, как C ++ или Java, этот шаблон реализуется с использованием абстрактного класса или интерфейса для определения стратегии аа. В Python имеет смысл просто определять некоторые функции извне, которые могут динамически добавляться к классу с использованием `types.MethodType` .

Введение в шаблоны проектирования и шаблон Singleton

Шаблоны проектирования предоставляют решения `commonly occurring problems` в разработке программного обеспечения. Шаблоны дизайна были впервые введены `GoF(Gang of Four)` где они описывали общие шаблоны как проблемы, возникающие снова и снова, и решения этих проблем.

Шаблоны проектирования имеют четыре основных элемента:

1. `The pattern name` - это дескриптор, который мы можем использовать для описания проблемы проектирования, ее решений и последствий одним или двумя словами.
2. `The problem` когда применять шаблон.
3. `The solution` описывает элементы, которые составляют дизайн, их отношения, обязанности и сотрудничество.
4. `The consequences` являются результаты и компромиссы применения шаблона.

Преимущества шаблонов проектирования:

1. Они многократно используются в нескольких проектах.
2. Архитектурный уровень проблем можно решить
3. Они проверены временем и хорошо зарекомендовали себя, это опыт разработчиков и архитекторов
4. Они имеют надежность и зависимость

Шаблоны проектирования можно разделить на три категории:

1. Шаблон создания
2. Структурный рисунок
3. Поведенческая картина

`Creational Pattern` - они связаны с тем, как объект может быть создан, и они изолируют детали создания объекта.

`Structural Pattern` - они создают структуру классов и объектов, чтобы они могли сочинять для достижения больших результатов.

`Behavioral Pattern` - они связаны с взаимодействием между объектами и ответственностью объектов.

Синглтон :

Это тип `creational pattern` который обеспечивает механизм для того, чтобы иметь только один и один объект данного типа и обеспечивает глобальную точку доступа.

например, Singleton может использоваться в операциях базы данных, где мы хотим, чтобы объект базы данных поддерживал согласованность данных.

Реализация

Мы можем реализовать Singleton Pattern в Python, создав только один экземпляр класса Singleton и снова обслуживая тот же объект.

```
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)
```

Выход:

```
('Object created', <__main__.Singleton object at 0x10a7cc310>)
('Object2 created', <__main__.Singleton object at 0x10a7cc310>)
```

Обратите внимание, что в таких языках, как C ++ или Java, этот шаблон реализуется путем создания частного конструктора и создания статического метода, который выполняет инициализацию объекта. Таким образом, один объект создается при первом вызове, и класс возвращает тот же объект после этого. Но в Python у нас нет никакого способа создать частные конструкторы.

Заводской шаблон

Заводская модель также является `Creational pattern` . Термин `factory` означает, что класс отвечает за создание объектов других типов. Существует класс, который действует как фабрика, которая имеет связанные с ней объекты и методы. Клиент создает объект, вызывая методы с определенными параметрами, и фабрика создает объект желаемого типа и возвращает его клиенту.

```
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)
```

Выход:

```
Which music you want to play Mp3 or Ogg"Ogg"
Playing .ogg music!
```

`MusicFactory` - это фабричный класс, который создает либо объект типа `Mp3` либо `Ogg` зависимости от выбора, который предоставляет пользователь.

полномочие

Объект прокси часто используется для обеспечения защищенного доступа к другому объекту, внутренняя бизнес-логика которого мы не хотим загрязнять с требованиями безопасности.

Предположим, мы хотели бы гарантировать, что только пользователь определенных разрешений может получить доступ к ресурсу.

Определение прокси: (он гарантирует, что только пользователи, которые действительно могут видеть оговорки, смогут пользоваться услугой customer_service)

```
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)
```

ВЫГОДЫ

- **мы избегаем каких-либо изменений в `ReservationService` при изменении ограничений доступа.**
- **мы не смешиваем данные, связанные с бизнесом (`date_from` , `date_to` , `reservations_count`) с не связанными с доменом понятиями (разрешениями пользователя) в сервисе.**
- **Потребитель (`StatsService`) также свободен от связанной с разрешениями логики**

ПРЕДОСТЕРЕЖЕНИЯ

- Интерфейс прокси всегда точно такой же, как и объект, который он скрывает, поэтому пользователь, который потребляет услугу, завернутый прокси-сервером, даже не знал о наличии прокси-сервера.

Прочитайте Шаблоны проектирования онлайн: https://riptutorial.com/ru/python/topic/8056/шаблоны-проектирования

глава 206: Юникод и байты

Синтаксис

- str.encode (кодирование, errors = 'strict')
- bytes.decode (encoding, errors = 'strict')
- open (имя файла, режим, кодирование = нет)

параметры

параметр	подробности
кодирование	Используемая кодировка, например `'ascii'` , `'utf8'` и т. Д. ...
ошибки	Режим ошибок, например, `'replace'` чтобы заменить плохие символы вопросительными знаками, `'ignore'` чтобы игнорировать плохие символы и т. Д. ...

Examples

основы

В Python 3 `str` - тип строк с поддержкой unicode, а `bytes` - это тип для последовательностей необработанных байтов.

```
type("f") == type(u"f")  # True, <class 'str'>
type(b"f")               # <class 'bytes'>
```

В Python 2 случайная строка представляла собой последовательность необработанных байтов по умолчанию, а строка юникода - каждая строка с префиксом «u».

```
type("f") == type(b"f")  # True, <type 'str'>
type(u"f")               # <type 'unicode'>
```

Юникод в байтах

Строки Unicode могут быть преобразованы в байты с `.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

в py2 стандартная консольная кодировка - это `sys.getdefaultencoding() == 'ascii'` а не `utf-8` как в py3, поэтому ее печать, как и в предыдущем примере, напрямую не возможна.

```
>>> 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
```

Если кодировка не может обрабатывать строку, создается «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)
```

Байты для unicode

Байты могут быть преобразованы в строки unicode с `.decode(encoding)` .

Последовательность байтов может быть преобразована только в строку юникода через соответствующую кодировку!

```
>>> b'\xc2\xa313.55'.decode('utf8')
'£13.55'
```

Если кодировка не может обрабатывать строку, создается `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
```

Обработка ошибок кодирования / декодирования

`.encode` и `.decode` имеют режим ошибок.

Значение по умолчанию - `'strict'` , что приводит к возникновению исключений при ошибке. Другие режимы более прощающие.

кодирование

```
>>> "£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'
```

расшифровка

```
>>> 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'
```

боевой дух

Из сказанного ясно, что очень важно сохранить ваши кодировки прямо при работе с unicode и байтами.

Файловый ввод-вывод

Файлы, открытые в недвоичном режиме (например, `'r'` или `'w'`), обрабатывают строки. Кодировка deafult - `'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")
```

Файлы, открытые в двоичном режиме (например, `'rb'` или `'wb'`), обрабатываются байтами. Аргумент кодирования не может быть указан, поскольку кодировка отсутствует.

```
open(fn, mode='wb')  # open file for writing bytes

# ERROR: cannot write string when bytes is expected:
open(fn, mode='wb').write("hi")
```

Прочитайте Юникод и байты онлайн: https://riptutorial.com/ru/python/topic/1216/юникод-и-байты

кредиты

S. No	Главы	Contributors
1	Начало работы с языком Python	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, CLDSEED, 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, noʇʎdʎzɐɹC, 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, Sнаđошƒа, 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 и ** 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	ArcPy	Midavalo, PolyGeo, Zhanping Shi
4	ChemPy - пакет python	Biswa_9937
5	Conditionals	Andy Hayden, BusyAnt, Chris Larson, deepakkt, Delgan, Elazar, evuez, Ffisegydd, Geeklhem, Hannes Karppila, James, Kevin Brown, krato, Max Feng, noɥʇʎdʎzɐɹƆ, rajah9, rrao, SashaZd, Simplans, Slayther, Soumendra Kumar Sahoo, Thomas Gerot, Trimax, Valentin Lorentz, Vinzee, wwii, xgord, Zack
6	ConfigParser	Chinmay Hegde, Dunatotatos
7	ctypes	Or East
8	Enum	Andy, Elazar, evuez, Martijn Pieters, techydesigner
9	hashlib	Mark Omo, xiaoyi
10	Heapq	ettanany
11	HTML-анализ	alecxe, talhasch
12	ijson	Prem Narain
13	kivy - кроссплатформенная платформа Python для разработки NUI	dhimanta
14	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
15	Mutable vs Immutable (и Hashable) в Python	Cilyan
16	Neo4j и Cypher используют Py2Neo	Wingston Sharon
17	os.path	Claudiu, Fábio Perez, girish946, Jmills, Szabolcs Dombi, VJ Magar
18	Pandas Transform: операции надготовкой по группам и конкатенация результатов	Dee
19	pip: Менеджер пакетов PyPI	Andy, Arpit Solanki, Community, InitializeSahib, JNat, Mahdi, Majid, Matt Giltaji, Nathaniel Ford, Rápli András, SerialDev, Simplans, Steve Barnes, StuxCrystal, tlo
20	PostgreSQL	Alessandro Trinca Tornidor, RamenChef, Stephen Leppik, user2027202827
21	py.test	Andy, Claudiu, Ffisegydd, Kinifwyne, Matt Giltaji
22	pyaudio	Biswa_9937
23	Pygame	Anthony Pham, Aryaman Arora, Pavan Nath
24	Pyglet	Comrade SparklePony, Stephen Leppik
25	PyInstaller - Распространение кода Python	ChaoticTwist, Eric, mnoronha
26	Python Anti-Patterns	Alessandro Trinca Tornidor, Annonymous, eenblam, Mahmoud Hashemi, RamenChef, Stephen Leppik
27	Python HTTP Server	Arpit Solanki, J F, jmunsch, Justin Chadwell, Mark, MervS, orvi, quantummind, Raghav, RamenChef, Sachin Kalkur, Simplans, techydesigner
28	Python Lex-Yacc	CLDSEED

29	Python и Excel	bee-sting, Chinmay Hegde, GiantsLoveDeathMetal, hackvan, Majid, talhasch, user2314737, Will
30	setup.py	Adam Brenecki, amblina, JNat, ravigadila, strpeter, user2027202827, Y0da
31	tempfile NamedTemporaryFile	Alessandro Trinca Tornidor, amblina, Kevin Brown, Stephen Leppik
32	Tkinter	Dartmouth, rlee827, Thomas Gerot, TidB
33	Unicode	wim
34	URLLIB	Amitay Stern, ravigadila, sth, Will
35	WebSockets	2Cubed, Stephen Leppik, Tyler Gubala
36	Абстрактное синтаксическое дерево	Teepeemm
37	Абстрактные базовые классы (abc)	Akshat Mahajan, Alessandro Trinca Tornidor, JGreenwell, Kevin Brown, Mattew Whitt, mkrieger1, SashaZd, Stephen Leppik
38	Альтернативы для переключения оператора с других языков	davidism, J F, zmo, Валерий Павлов
39	Асинхронный модуль	2Cubed, Alessandro Trinca Tornidor, Cimbali, hiro protagonist, obust, pylang, RamenChef, Seth M. Larson, Simplans, Stephen Leppik, Udi
40	аудио	blueberryfields, Comrade SparklePony, frankyjuang, jmunsch, orvi, qwertyuip9, Stephen Leppik, Thomas Gerot
41	Безопасность и криптография	adeora, ArtOfCode, BSL-5, Kevin Brown, matsjoyce, SuperBiasedMan, Thomas Gerot, Wladimir Palant, wrwrwr
42	Библиотека подпроцессов	Adam Matan, Andrew Schade, Brendan Abel, jfs, jmunsch, Riccardo Petraglia
43	Булевы операторы	boboquack, Brett Cannon, Dair, Ffisegydd, John Zwinck, Severiano Jaramillo Quintanar, Steven Maude
44	Введение в RabbitMQ с использованием AMQPStorm	eandersson

45	Ввод, подмножество и вывод внешних файлов данных с использованием Pandas	Mark Miller
46	вдавливание	Alessandro Trinca Tornidor, depperm, J F, JGreenwell, Matt Giltaji, Pasha, RamenChef, Stephen Leppik
47	Веб-скребок с Python	alecxe, Amitay Stern, jmunsch, mrtuovinen, Ni., RamenChef, Saiful Azad, Saqib Shamsi, Simplans, Steven Maude, sth, sytech, talhasch, Thomas Gerot
48	Визуализация данных с помощью Python	Aquib Javed Khan, Arun, ChaoticTwist, cledoux, Ffisegydd, ifma
49	Виртуальная среда Python - virtualenv	Vikash Kumar Jain
50	виртуальная среда с virtualenvwrapper	Sirajus Salayhin
51	Виртуальные среды	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
52	Возведение	Anthony Pham, intboolstring, jtbandes, Luke Taylor, MSeifert, Pasha, supersam654
53	Вывоз мусора	bogdanciobanu, Claudiu, Conrad.Dean, Elazar, FazeL, J F, James Elderfield, lukess, muddyfish, Sam Whited, SiggyF, Stephen Leppik, SuperBiasedMan, Xavier Combelle
54	Вызовите Python с C #	Julij Jegorov
55	Выполнение динамического кода с помощью `exec` и` eval`	Antti Haapala, Ilja Everilä
56	Генераторы	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
57	граф-инструмент	xiaoyi
58	группа по()	Parousia, Thomas Gerot
59	Дата и время	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
60	Двоичные данные	Eleftheria, evuez, mnoronha
61	Декораторы	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
62	дескриптор	bbayles, cizixs, Nemo, pylang, SuperBiasedMan
63	Джанго	code_geek, orvi
64	Доступ к атрибутам	Elazar, SashaZd, SuperBiasedMan
65	Доступ к базе данных	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
66	Доступ к исходному коду и байт-коду Python	muddyfish, StuxCrystal, user2314737
67	Задавать	Andrzej Pronobis, Andy Hayden, Bahrom, Cimbali, Cody Piersall, Conrad.Dean, Elazar, evuez, J F, James, Or East,

		pylang, RahulHP, RamenChef, Simplans, user2314737
68	Запись в CSV из строки или списка	Hriddhi Dey, Thomas Crowley
69	Заявление о прохождении	Anaphory
70	Идиомы	Benjamin Hodgson, Elazar, Faiz Halde, J F, Lee Netherton, loading..., Mister Mister
71	Импорт модулей	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
72	Индексация и нарезка	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
73	Инструмент 2to3	Alessandro Trinca Tornidor, Dartmouth, Firix, Kevin Brown, Naga2Raja, Stephen Leppik
74	Интерпретатор (консоль командной строки)	Aaron Christiansen, David, Elazar, Peter Shinners, ppperry
75	Интерфейс шлюза веб-сервера (WSGI)	David Heyman, Kevin Brown, Preston, techydesigner
76	Исключения	Adrian Antunez, Alessandro Trinca Tornidor, Alfe, Andy, Benjamin Hodgson, Brian Rodriguez, BusyAnt, Claudiu, driax, Elazar, flazzarini, ghostarbeiter, Ilia Barahovski, J F, Marco Pashkov, muddyfish, noɥʇʎԀʎzɐɹƆ, Paul Weaver, Rahul Nair, RamenChef, Shawn Mehan, Shiven, Shkelqim Memolla, Simplans, Slickytail, Stephen Leppik, Sudip Bhandari, SuperBiasedMan, user2314737
77	Исключения из Содружества	Juan T, TemporalWolf

78	Использование модуля «pip»: диспетчер пакетов PyPI	Zydnar
79	Использование петель внутри функций	naren
80	Итераторы и итераторы	4444, Conrad.Dean, demonplus, Ilia Barahovski, Pythonista
81	Классы	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
82	Кодовые блоки, кадры выполнения и пространства имен	Jeremy, Mohammed Salman
83	колба	Stephen Leppik, Thomas Gerot
84	Комментарии и документация	Ani Menon, FunkySayu, MattCorr, SuperBiasedMan, TuringTux
85	Контекстные менеджеры ("with" Statement)	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
86	Копирование данных	hashcode55, StuxCrystal
87	Кортеж	Anthony Pham, Antoine Bolvy, BusyAnt, Community, Elazar, James, Jim Fasarakis Hilliard, Joab Mendes, Majid,

		Md.Sifatul Islam, Mechanic, mezzode, nlsdfnbch, noɥʇʎd ʎzɐɹƆ, Selcuk, Simplans, textshell, tobias_k, Tony Suffolk 66, user2314737
88	логирование	Gal Dreiman, Jörn Hees, sxnwlfkk
89	Манипулирование XML	4444, Brad Larson, Chinmay Hegde, Francisco Guimaraes, greuze, heyhey2k, Rob Murray
90	Массивы	Andy, Pavan Nath, RamenChef, Vin
91	Математический модуль	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
92	Метаклассы	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
93	Многомерные массивы	boboquack, Buzz, rrao
94	Многопоточность	Alu, cldseed, juggernaut, Kevin Brown, Kristof, mattgathu, Nabeel Ahmed, nlsdfnbch, Rahul, Rahul Nair, Riccardo Petraglia, Thomas Gerot, Will, Yogendra Sharma
95	многопроцессорная обработка	Alon Alexander, Nander Speerstra, unutbu, Vinzee, Will
96	Модуль base64	Thomas Gerot
97	Модуль Deque	Anthony Pham, BusyAnt, matsjoyce, ravigadila, Simplans, Thomas Ahle, user2314737
98	Модуль dis	muddyfish, user2314737
99	Модуль Functools	Alessandro Trinca Tornidor, enrico.bacis, flamenco, RamenChef, Shrey Gupta, Simplans, Stephen Leppik, StuxCrystal
100	Модуль 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

101	Модуль JSON	Indradhanush Gupta, Leo, Martijn Pieters, pzp, theheadofabroom, Underyx, Wolfgang
102	Модуль os	Andy, Christian Ternus, JelmerS, JL Peyret, mnoronha, Vinzee
103	модуль pyautogui	Damien, Rednivrug
104	Модуль Sqlite3	Chinmay Hegde, Simplans
105	Модуль Webbrowser	Thomas Gerot
106	Модуль коллекций	asmeurer, Community, Elazar, jmunsch, kon psych, Marco Pashkov, MSeifert, RamenChef, Shawn Mehan, Simplans, Steven Maude, Symmitchry, void, XCoder Real
107	Модуль локали	Will, XonAether
108	Модуль очереди	Prem Narain
109	Написание расширений	Dartmouth, J F, mattgathu, Nathan Osman, techydesigner, ygram
110	начать с GZip	orvi
111	Неизменяемые типы данных (int, float, str, кортеж и frozensets)	Alessandro Trinca Tornidor, FazeL, Ganesh K, RamenChef, Stephen Leppik
112	Неофициальные реализации Python	Jacques de Hooge, Squidward
113	Несовместимость, перемещающаяся с Python 2 на 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, CLDSEED, 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, noɥʇʎԀʎzɐɹƆ, 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
114	Область переменных и привязка	Anthony Pham, davidism, Elazar, Esteis, Mike Driscoll, SuperBiasedMan, user2314737, zvone
115	Общие проблемы	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
116	Объекты недвижимости	Alessandro Trinca Tornidor, Darth Shadow, DhiaTN, J F, Jacques de Hooge, Leo, Martijn Pieters, mnoronha, Priya, RamenChef, Stephen Leppik
117	Операторский модуль	MSeifert
118	Определение функций со списком аргументов	zenlc2000
119	Оптимизация	A. Ciclet, RamenChef, user2314737

	производительности	
120	Оптическое распознавание символов	rassar
121	Основные входные и выходные данные	Doraemon, GoatsWearHats, J F, JNat, Marco Pashkov, Mark Miller, Martijn Pieters, Nathaniel Ford, Nicolás, pcurry, pzp, SashaZd, SuperBiasedMan, Vilmar
122	Основные проклятия с Python	4444, Guy, kollery, Vinzee
123	откладывать в долгий ящик	Biswa_9937
124	отладка	Aldo, B8vrede, joel3000, Sardathrion, Sardorbek Imomaliev, Vlad Bezden
125	Параллельное вычисление	Akshat Mahajan, Dair, Franck Dernoncourt, J F, Mahdi, nlsdfnbch, Ryan Smith, Vinzee, Xavier Combelle
126	Параллельность Python	David Heyman, Faiz Halde, Iván Rodríguez Torres, J F, Thomas Moreau, Tyler Gubala
127	перегрузка	Andy Hayden, Darth Shadow, ericmarkmartin, Ffisegydd, Igor Raush, Jonas S, jonrsharpe, L3viathan, Majid, RamenChef, Simplans, Valentin Lorentz
128	Переопределение метода	DeepSpace, James
129	Перечисление списков	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, noɥʇʎԀʎzɐɹƆ, 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
130	Плагины и расширения	2Cubed, proprefenetre, pylang, rrao, Simon Hibbs, Simplans
131	Побитовые операторы	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
132	Повышение пользовательских ошибок / исключений	naren
133	Подключение Python к SQL Server	metmirr
134	Подключение Secure Shell в Python	mnoronha, Shijo
135	Подкоманды CLI с точным выводом справки	Alessandro Trinca Tornidor, anatoly techtonik, Darth Shadow
136	подсчет	Andy Hayden, MSeifert, Peter Mølgaard Pallesen, pylang
137	подушка	Razik
138	поиск	Dan Sanderson, Igor Raush, MSeifert
139	Полиморфизм	Benedict Bunting, DeepSpace, depperm, Simplans,

		skrrgwasme, Vinzee
140	Пользовательские методы	Alessandro Trinca Tornidor, Beall619, mnoronha, RamenChef, Stephen Leppik, Sun Qingyao
141	Последовательная связь Python (pyserial)	Alessandro Trinca Tornidor, Ani Menon, girish946, mnoronha, Saranjith, user2314737
142	Построение графика с помощью Matplotlib	Arun, user2314737
143	Примеси	Doc, Rahul Nair, SashaZd
144	Приоритет оператора	HoverHell, JGreenwell, MathSquared, SashaZd, Shreyash S Sarnayak
145	Проверка наличия и разрешения пути	Esteis, Marlon Abeykoon, mnoronha, PYPL
146	Программирование IoT с использованием Python и малины PI	dhimanta
147	Простые математические операторы	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
148	профилирование	J F, keiv.fly, SashaZd
149	Процессы и потоки	Claudiu, Thomas Gerot
150	Работа над глобальным блокировщиком перевода (GIL)	Scott Mermelstein
151	Работа с ZIP-архивами	Chinmay Hegde, ghostarbeiter, Jeffrey Lin, SuperBiasedMan
152	Разбор аргументов командной строки	amblina, Braiam, Claudiu, cledoux, Elazar, Gerard Roche, krato, loading..., Marco Pashkov, Or Duan, Pasha, RamenChef, rfkortekaas, Simplans, Thomas Gerot, Topperfalkon, zmo, zondo

153	развертывание	Gal Dreiman, lancnorden, Wayne Werner
154	Разница между модулем и пакетом	DeepSpace, Simplans, tjohnson
155	Распаковка файлов	andrew
156	распределение	Alessandro Trinca Tornidor, JGreenwell, metahost, Pigman168, RamenChef, Stephen Leppik
157	Регулярные выражения (регулярное выражение)	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
158	Рекурсия	Bastian, japborst, JGreenwell, Jossie Calderon, mbomb007, SashaZd, Tyler Crompton
159	Розетки	David Cullen, Dev, MattCorr, nlsdfnbch, Rob H, StuxCrystal, textshell, Thomas Gerot, Will
160	Связанные списки	Nemo
161	Сериализация данных	Devesh Saini, Infinity, rfkortekaas
162	Сериализация данных сортировки	J F, Majid, Or East, RahulHP, rfkortekaas, zvone
163	Сеть Python	atayenel, ChaoticTwist, David, Geeklhem, mattgathu, mnoronha, thsecmaniac
164	системный	blubberdiblub
165	Скорость программы Python	ADITYA, Antonio, Elodin, Neil A., Vinzee
166	Скрытые функции	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
167	Сложная математика	Adeel Ansari, Bosoneando, bpachev

168	Случайный модуль	Alex Gaynor, Andrzej Pronobis, Anthony Pham, Community, David Robinson, Delgan, giucal, Jim Fasarakis Hilliard, michaelrbock, MSeifert, Nobilis, ppperry, RamenChef, Simplans, SuperBiasedMan
169	События, отправленные сервером Python	Nick Humrich
170	Создание виртуальной среды с помощью virtualenvwrapper в окнах	Sirajus Salayhin
171	Создание пакетов Python	Claudiu, KeyWeeUsr, Marco Pashkov, pylang, SuperBiasedMan, Thtu
172	Создание службы Windows с использованием Python	Simon Hibbs
173	Сокеты и шифрование / расшифровка сообщений между клиентом и сервером	Mohammad Julfikar
174	Сообщение Python Requests	Ken Y-N, RandomHash
175	Сортировка списка (выбор частей списков)	Greg, JakeD
176	Сортировка, минимальная и максимальная	Antti Haapala, APerson, GoatsWearHats, Mirec Miskuf, MSeifert, RamenChef, Simplans, Valentin Lorentz
177	Специальная переменная __ name__	Annonymous, BusyAnt, Christian Ternus, jonrsharpe, Lutz Prechelt, Steven Elliott
178	Список	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, noɥʇʎԀʎzɐɹƆ, 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
179	Список деструктурирования (ака упаковка и распаковка)	J F, sth, zmo
180	Список рекомендаций	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, noɥʇʎԀʎzɐɹƆ, pktangyue, Pyth0nicPenguin, Rahul Nair, Riccardo Petraglia, SashaZd, shrishinde, Simplans, Slayther, sudo bangbang, theheadofabroom, then0rTh, Tim McNamara, Udi, Valentin Lorentz, Veedrac, Zags
181	Сравнения	Anthony Pham, Ares, Elazar, J F, MSeifert, Shawn Mehan, SuperBiasedMan, Will, Xavier Combelle
182	стек	ADITYA, boboquack, Chromium, cjds, depperm, Hannes Karppila, JGreenwell, Jonatan, kdopen, OliPro007, orvi, SashaZd, Sнаđошƒа, textshell, Thomas Ahle, user2314737
183	Стойкость к Python	RamenChef, user2728397
184	Строковые методы	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
185	Строковые представления экземпляров класса: методы __str__ и __repr__	Alessandro Trinca Tornidor, jedwards, JelmerS, RamenChef, Stephen Leppik
186	Сходства в синтаксисе, Различия в значении: Python и JavaScript	user2683246
187	Тестирование устройства	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
188	Тип подсказки	alecxe, Annonymous, Antti Haapala, Elazar, Jim Fasarakis Hilliard, Jonatan, RamenChef, Seth M. Larson, Simplans, Stephen Leppik
189	Типы данных Python	Gavin, lorenzofeliz, Pike D., Rednivrug
190	толковый словарь	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, noɥʇʎԀʎzɐɹƆ, ronrest, Roy Iacob, Shawn Mehan, Simplans, SuperBiasedMan, TehTris, Valentin Lorentz, viveksyngh, Xavier Combelle
191	Узел узлового списка	orvi
192	уменьшить	APerson, Igor Raush, Martijn Pieters, MSeifert
193	Файлы и папки 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

194	Фильтр	APerson, cfi, J Atkin, MSeifert, rajah9, SuperBiasedMan
195	Форматирование даты	surfthecity
196	Форматирование строк	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
197	функции	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., noɥʇʎԀʎzɐɹƆ, 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
198	Функциональное программирование на Python	Imran Bughio, mvis89, Rednivrug
199	Функция карты	APerson, cfi, Igor Raush, Jon Ericson, Karl Knechtel, Marco Pashkov, MSeifert, noɥʇʎԀʎzɐɹƆ, Parousia, Simplans, SuperBiasedMan, tlama, user2314737
200	Функция печати	Beall619, Frustrated, Justin, Leon Z., lukewrites, SuperBiasedMan, Valentin Lorentz

201	Частичные функции	FrankBr
202	Черепашья графика	Luca Van Oort, Stephen Leppik
203	Чтение и запись CSV	Adam Matan, Franck Dernoncourt, Martin Valgur, mnoronha, ravigadila, Setu
204	Шаблоны в python	4444, Alessandro Trinca Tornidor, Fred Barclay, RamenChef, Ricardo, Stephen Leppik
205	Шаблоны проектирования	Charul, denvaar, djaszczurowski
206	Юникод и байты	Claudiu, KeyWeeUsr