Kostenloses eBook

LERNEN

Python Language

Free unaffiliated eBook created from
Stack Overflow contributors.

#python

Inhaltsverzeichnis

Über

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.

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

Kapitel 1: Erste Schritte mit Python Language

Bemerkungen

Python ist eine weit verbreitete Programmiersprache. Es ist:

- **High-Level** : Python automatisiert Low-Level-Vorgänge wie die Speicherverwaltung. Der Programmierer hat etwas weniger Kontrolle, hat aber viele Vorteile, einschließlich Lesbarkeit des Codes und minimale Code-Ausdrücke.
- **Allzweck** : Python kann in allen Kontexten und Umgebungen verwendet werden. Ein Beispiel für eine nicht-allgemeinsprachliche Sprache ist PHP: PHP wurde speziell als serverseitige Skriptsprache für die Webentwicklung entwickelt. Im Gegensatz dazu *kann* Python für die serverseitige Web-Entwicklung verwendet werden, sondern auch Desktop - Anwendungen für den Aufbau.
- **Dynamisch geschrieben** : Jede Variable in Python kann auf jeden Datentyp verweisen. Ein einzelner Ausdruck kann zu unterschiedlichen Zeitpunkten Daten verschiedener Typen auswerten. Daher ist folgender Code möglich:

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

- **Stark typisiert** : Während der Programmausführung dürfen Sie nichts tun, was mit dem Datentyp, mit dem Sie arbeiten, nicht kompatibel ist. Beispielsweise gibt es keine versteckten Konvertierungen von Zeichenfolgen in Zahlen. Eine aus Ziffern bestehende Zeichenfolge wird niemals als Zahl behandelt, wenn Sie sie nicht explizit konvertieren:

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

- **Anfängerfreundlich :)** : Pythons Syntax und Struktur sind sehr intuitiv. Es ist auf hohem Niveau und enthält Konstrukte, die das Schreiben klarer Programme in kleinem und großem Maßstab ermöglichen. Python unterstützt mehrere Programmierparadigmen, einschließlich objektorientierter, imperativer und funktionaler Programmierung oder prozeduraler Stile. Es verfügt über eine umfangreiche, umfassende Standardbibliothek und viele einfach zu installierende Bibliotheken von Drittanbietern.

Seine Gestaltungsprinzipien werden im *Zen of Python beschrieben* .

Derzeit gibt es zwei Hauptveröffentlichungszweige von Python, die einige signifikante

Unterschiede aufweisen. Python 2.x ist die Legacy-Version, obwohl sie immer noch weit verbreitet ist. Python 3.x führt eine Reihe rückwärtskompatibler Änderungen durch, um die Duplizierung von Funktionen zu reduzieren. In diesem Artikel erfahren Sie, wie Sie entscheiden können, welche Version für Sie am besten geeignet ist .

Die offizielle Python-Dokumentation ist auch eine umfassende und nützliche Ressource, die Dokumentation für alle Python-Versionen sowie Tutorials enthält, die Ihnen den Einstieg erleichtern.

Es gibt eine offizielle Implementierung der von Python.org bereitgestellten Sprache, die im Allgemeinen als CPython bezeichnet wird, und mehrere alternative Implementierungen der Sprache auf anderen Laufzeitplattformen. Dazu gehören IronPython (auf dem .NET Python ausgeführt wird), Jython (auf der Java-Laufzeitumgebung) und PyPy (das Python in einer Teilmenge von sich selbst implementiert).

Versionen

Python 3.x

Ausführung	Veröffentlichungsdatum
[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

Ausführung	Veröffentlichungsdatum
2,7	2010-07-03
2.6	2008-10-02
2,5	2006-09-19

Ausführung	Veröffentlichungsdatum
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

Fertig machen

Python ist eine weit verbreitete Programmiersprache für allgemeine Zwecke, die von Guido van Rossum entwickelt und erstmals 1991 veröffentlicht wurde. Python verfügt über ein dynamisches Typsystem und eine automatische Speicherverwaltung und unterstützt mehrere Programmierparadigmen, einschließlich objektorientierter, funktionale Programmierung und prozedurale Stile. Es verfügt über eine große und umfassende Standardbibliothek.

Derzeit sind zwei Hauptversionen von Python aktiv:

- Python 3.x ist die aktuelle Version und wird derzeit weiterentwickelt.
- Python 2.x ist die ältere Version und wird bis 2020 nur Sicherheitsupdates erhalten. Es werden keine neuen Funktionen implementiert. Beachten Sie, dass viele Projekte weiterhin Python 2 verwenden, obwohl die Migration zu Python 3 einfacher wird.

Sie können beide Versionen von Python hier herunterladen und installieren. Siehe Python 3 vs. Python 2 für einen Vergleich. Darüber hinaus bieten einige Drittanbieter neu verpackte Versionen von Python an, die häufig verwendete Bibliotheken und andere Funktionen hinzufügen, um das Einrichten allgemeiner Anwendungsfälle wie Mathematik, Datenanalyse oder wissenschaftliche Verwendung zu erleichtern. Siehe die Liste auf der offiziellen Website .

Überprüfen Sie, ob Python installiert ist

Um zu bestätigen, dass Python korrekt installiert wurde, können Sie dies überprüfen, indem Sie den folgenden Befehl in Ihrem bevorzugten Terminal ausführen (Wenn Sie ein Windows-Betriebssystem verwenden, müssen Sie der Umgebungsvariablen den Pfad von Python hinzufügen, bevor Sie ihn in der Eingabeaufforderung verwenden):

```
$ python --version
```

Python 3.x 3.0

Wenn Sie *Python 3* installiert haben und dies Ihre Standardversion ist (weitere Informationen finden Sie unter **Fehlerbehebung**), sollten Sie etwa Folgendes sehen:

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

Python 2.x 2.7

Wenn Sie *Python 2* installiert haben und dies Ihre Standardversion ist (weitere Informationen finden Sie unter **Fehlerbehebung**), sollten Sie etwa Folgendes sehen:

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

Wenn Sie Python 3 installiert haben, aber `$ python --version` eine Python 2-Version ausgibt, haben Sie auch Python 2 installiert. Dies ist häufig auf MacOS und vielen Linux-Distributionen der Fall. Verwenden `$ python3` stattdessen `$ python3` , um den Python 3-Interpreter explizit zu verwenden.

Hallo, World in Python mit IDLE

IDLE ist ein einfacher Editor für Python, der im Lieferumfang von Python enthalten ist.

So erstellen Sie ein Hello, World-Programm in IDLE

- Öffnen Sie IDLE auf Ihrem System Ihrer Wahl.
  - In älteren Windows-Versionen finden Sie es unter `All Programs` im Windows-Menü.
  - Suchen Sie in Windows 8+ nach `IDLE` oder suchen Sie nach den Apps, die in Ihrem System vorhanden sind.
  - Auf Unix-basierten (einschließlich Mac) Systemen können Sie es von der Shell aus öffnen, indem Sie `$ idle python_file.py` .
- Es wird eine Shell mit Optionen oben geöffnet.

In der Schale werden drei rechtwinklige Klammern angezeigt:

```
>>>
```

Schreiben Sie nun den folgenden Code in die Eingabeaufforderung:

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

Drücken Sie die `Eingabetaste` .

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

Hallo Welt Python-Datei

Erstellen Sie eine neue Datei `hello.py` , die die folgende Zeile enthält:

Python 3.x 3.0

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

Python 2.x 2.6

Sie können die Python 3- `print` in Python 2 mit der folgenden `import` :

```
from __future__ import print_function
```

Python 2 verfügt über eine Reihe von Funktionalitäten, die optional mit dem Modul `__future__` aus Python 3 `__future__` werden können, wie hier beschrieben .

Python 2.x 2.7

Wenn Sie Python 2 verwenden, können Sie auch die folgende Zeile eingeben. Beachten Sie, dass dies in Python 3 nicht gültig ist und daher nicht empfohlen wird, da es die Kompatibilität von Cross-Versionscode reduziert.

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

Navigieren Sie in Ihrem Terminal zu dem Verzeichnis, in dem sich die Datei `hello.py` .

`python hello.py` und `python hello.py` die `Eingabetaste` .

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

Sie sollten `Hello, World` auf der Konsole anzeigen.

Sie können auch `hello.py` durch den Pfad zu Ihrer Datei `hello.py` . Wenn sich die Datei beispielsweise in Ihrem Home-Verzeichnis befindet und Ihr Benutzer unter Linux "Benutzer" ist, können Sie `python /home/user/hello.py` .

Starten Sie eine interaktive Python-Shell

Durch Ausführen (Ausführen) des `python` Befehls in Ihrem Terminal wird eine interaktive Python-Shell angezeigt. Dies wird auch als Python-Interpreter oder REPL (für "Read Evaluate Print Loop") bezeichnet.

```
$ 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
>>>
```

Wenn Sie Python 3 von Ihrem Terminal aus ausführen `python3` , führen Sie den Befehl `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
>>>
```

Alternativ können Sie die interaktive Eingabeaufforderung starten und die Datei mit `python -i <file.py>` .

Führen Sie in der Befehlszeile Folgendes aus:

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

Es gibt mehrere Möglichkeiten, die Python-Shell zu schließen:

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

oder

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

Alternativ schließen Sie `STRG + D` die Shell und setzen Sie wieder in die Befehlszeile Ihres Terminals.

Wenn Sie einen Befehl abbrechen möchten, befinden Sie sich mitten in der Eingabe und kehren zu einer Eingabeaufforderung zum Reinigen zurück, während Sie sich in der Interpreter-Shell befinden, und `drücken Sie STRG + C`.

Testen Sie eine interaktive Python-Shell online .

Andere Online-Shells

Verschiedene Websites bieten Online-Zugriff auf Python-Shells.

Online-Shells können für folgende Zwecke nützlich sein:

- Führen Sie ein kleines Code-Snippet von einer Maschine aus, auf der Python-Installation fehlt (Smartphones, Tablets usw.).
- Lernen oder lehren Sie grundlegende Python.
- Lösen Sie Online-Richterprobleme.

Beispiele:

> Haftungsausschluss: Dokumentationsautor (en) sind nicht mit den unten aufgeführten Ressourcen verbunden.

- https://www.python.org/shell/ - Die auf der offiziellen Python-Website gehostete Online-Python-Shell.
- https://ideone.com/ - Wird im Internet häufig verwendet, um das Verhalten von Codeausschnitten darzustellen.
- https://repl.it/languages/python3 - Leistungsfähiger und einfacher Online-Compiler, IDE und Interpreter. Code programmieren, kompilieren und Code in Python ausführen.
- https://www.tutorialspoint.com/execute_python_online.php - Eine voll funktionsfähige UNIX-Shell und ein benutzerfreundlicher Projektexplorer.
- http://rextester.com/l/python3_online_compiler - Einfache und benutzerfreundliche IDE, die die Ausführungszeit anzeigt

Befehle als String ausführen

Python kann beliebigen Code als String in der Shell übergeben werden:

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

Dies kann hilfreich sein, wenn Sie die Ergebnisse von Skripts in der Shell miteinander verketten.

Muscheln und jenseits

Paketverwaltung - Das von PyPA empfohlene Werkzeug zum Installieren von Python-Paketen ist PIP . Zur Installation führen Sie auf Ihrer Kommandozeile `pip install <the package name>` . Zum Beispiel `pip install numpy` . (Hinweis: Unter Windows müssen Sie den PATH-Umgebungsvariablen pip hinzufügen. Um dies zu vermeiden, verwenden Sie `python -m pip install <the package name>`)

Shells - Bisher haben wir verschiedene Möglichkeiten zur Ausführung von Code mit Pythons nativer interaktiver Shell beschrieben. Shells verwenden Pythons Interpretationskraft, um mit Code in Echtzeit zu experimentieren. Zu den alternativen Shells gehören IDLE - eine vorgefertigte GUI, IPython -, die für die Erweiterung der interaktiven Erfahrung usw. bekannt ist.

Programme - Zur Langzeitspeicherung können Sie Inhalte in .py-Dateien speichern und als Skripte oder Programme mit externen Tools wie Shell, IDEs (wie PyCharm), Jupyter-Notebooks usw. bearbeiten / ausführen. Benutzer mit Zwischenspeicher können diese Tools verwenden. Die hier besprochenen Methoden reichen jedoch für den Einstieg aus.

Mit dem Python-Tutor können Sie schrittweise durch den Python-Code navigieren, um zu visualisieren, wie das Programm abläuft, und hilft Ihnen zu verstehen, wo Ihr Programm fehlerhaft ist.

PEP8 definiert Richtlinien zum Formatieren von Python-Code. Das Formatieren von Code ist wichtig, damit Sie schnell lesen können, was der Code bewirkt.

Variablen anlegen und Werte zuweisen

Um eine Variable in Python zu erstellen, müssen Sie nur den Variablennamen angeben und dieser einen Wert zuweisen.

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

Python verwendet `=` , um Variablen Werte zuzuweisen. Es ist nicht erforderlich, eine Variable im Voraus zu deklarieren (oder ihr einen Datentyp zuzuweisen). Wenn Sie einer Variablen einen Wert zuweisen, deklariert und initialisiert sie die Variable mit diesem Wert. Es gibt keine Möglichkeit, eine Variable zu deklarieren, ohne ihr einen Anfangswert zuzuweisen.

```
# 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
```

Die variable Zuweisung funktioniert von links nach rechts. Im Folgenden erhalten Sie einen Syntaxfehler.

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

Sie können die Schlüsselwörter von python nicht als gültigen Variablennamen verwenden. Sie können die Liste der Keywords anzeigen:

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

Regeln für die Benennung von Variablen:

1. Variablennamen müssen mit einem Buchstaben oder einem Unterstrich beginnen.

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

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

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

2. Der Rest Ihres Variablennamens kann aus Buchstaben, Zahlen und Unterstrichen bestehen.

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

3. Namen sind case sensitive.

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

Obwohl bei der Deklaration einer Variablen in Python kein Datentyp angegeben werden muss, während der erforderliche Speicherbereich für die Variable reserviert wird, wählt der Python-Interpreter automatisch den am besten geeigneten integrierten Typ dafür aus:

```
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'>
```

Jetzt kennen Sie die Grundlagen der Zuweisung. Lassen Sie uns diese Feinheiten über die Zuweisung in Python aus dem Weg räumen.

Wenn Sie `=` für eine Zuweisungsoperation verwenden, ist links von `=` ein ***Name*** für das ***Objekt*** auf der rechten Seite. Schließlich , was `=` tut , ist die ***Referenz*** des Objekts auf der linken Seite auf der rechten Seite auf den ***Namen*** zuweisen.

Das ist:

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

Wenn wir also aus vielen Zuordnungsbeispielen oben `pi = 3.14` auswählen, ist `pi` **ein** Name (nicht **der** Name, da ein Objekt mehrere Namen haben kann) für das Objekt `3.14` . Wenn Sie unten etwas nicht verstehen, kommen Sie zu diesem Punkt zurück und lesen Sie das noch einmal! Sie können dies auch zum besseren Verständnis betrachten.

Sie können mehreren Variablen in einer Zeile mehrere Werte zuweisen. Beachten Sie, dass auf der rechten und linken Seite des Operators `=` die gleiche Anzahl von Argumenten vorhanden sein muss:

```
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
```

Der Fehler im letzten Beispiel kann vermieden werden, indem der gleichen Anzahl von beliebigen Variablen verbleibende Werte zugewiesen werden. Diese Dummy-Variable kann einen beliebigen Namen haben, es ist jedoch üblich, den Unterstrich (_) zum Zuweisen unerwünschter Werte zu verwenden:

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

Beachten Sie, dass die Anzahl von _ und die Anzahl der verbleibenden Werte gleich sein müssen.

Andernfalls wird "zu viele Werte zum Auspacken des Fehlers" wie oben ausgegeben:

```
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)
```

Sie können auch mehreren Variablen gleichzeitig einen einzelnen Wert zuweisen.

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

Wenn Sie eine solche kaskadierende Zuweisung verwenden, ist es wichtig zu beachten, dass sich alle drei Variablen `a` , `b` und `c` auf *dasselbe Objekt* im Speicher beziehen, ein `int` Objekt mit dem Wert 1. Mit anderen Worten sind `a` , `b` und `c` drei verschiedene Namen auf dasselbe int Objekt gegeben. Wenn Sie einem Objekt anschließend ein anderes Objekt zuweisen, werden die anderen Objekte nicht wie erwartet geändert:

```
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.
```

Das Obige gilt auch für veränderliche Typen (wie `list` , `dict` usw.), ebenso wie für unveränderliche Typen (wie `int` , `string` , `tuple` usw.):

```
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]
```

So weit, ist es gut. Bei der *Änderung* des Objekts *sieht* die Sache etwas anders aus (im Gegensatz zur *Zuweisung* des Namens zu einem anderen Objekt, wie oben beschrieben), wenn die kaskadierende Zuordnung für veränderliche Typen verwendet wird. Schauen Sie unten, und Sie werden es aus erster Hand sehen:

```
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
```

Verschachtelte Listen sind auch in Python gültig. Das bedeutet, dass eine Liste eine andere Liste als Element enthalten kann.

```
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
```

Schließlich müssen Variablen in Python nicht den Typ beibehalten, für den sie zuerst definiert wurden. Sie können einfach `=` , um einer Variablen einen neuen Wert zuzuweisen, selbst wenn dieser Wert einen anderen Typ hat.

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

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

Wenn dies Sie stört, denken Sie darüber nach, dass das, was links von `=` ist, nur ein Name für ein Objekt ist. Zuerst Sie den Anruf `int` mit dem Wert 2 Objekt `a` , dann ändern Sie Ihre Meinung und entscheiden , den Namen zu geben , `a` zu einem `string` - Objekt, Wert ‚Neuer Wert' mit. Einfach, richtig?

Benutzereingabe

Interaktive Eingabe

Um vom Benutzer Eingaben zu erhalten, verwenden Sie die `input` (**Hinweis** : In Python 2.x heißt die Funktion stattdessen `raw_input` , obwohl Python 2.x eine eigene `input` , die sich völlig unterscheidet):

Python 2.x 2.3

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

> **Sicherheitshinweis** Verwenden Sie `input()` in Python2 - der eingegebene Text wird wie ein Python-Ausdruck (entspricht `eval(input())` in Python3) ausgewertet, der leicht zu einer Sicherheitsanfälligkeit werden kann. In diesem Artikel finden Sie weitere Informationen zu den Risiken bei der Verwendung dieser Funktion.

Python 3.x 3.0

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

Der Rest dieses Beispiels verwendet die Python 3-Syntax.

Die Funktion nimmt ein Zeichenfolgenargument an, das es als Eingabeaufforderung anzeigt und eine Zeichenfolge zurückgibt. Der obige Code bietet eine Eingabeaufforderung, die auf die Eingabe des Benutzers wartet.

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

Wenn der Benutzer „Bob“ und Hits eingeben, der Variable `name` wird mit dem String zugewiesen werden `"Bob"` :

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

Beachten Sie, dass die `input` immer vom Typ `str` ist. `str` ist wichtig, wenn der Benutzer Zahlen eingeben soll. Daher müssen Sie den `str` konvertieren, bevor Sie ihn als Zahl verwenden `str` :

```
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
```

Hinweis: Es wird empfohlen, `try` / `except` Blöcke zu verwenden, um Ausnahmen bei Benutzereingaben abzufangen . Wenn zum Beispiel Ihr Code einen `raw_input` in ein `int` umwandeln möchte und der Benutzer nichts zu schreiben hat, ist dies nicht der `ValueError` . Dies führt zu einem `ValueError` .

IDLE - Python-GUI

IDLE ist die integrierte Entwicklungs- und Lernumgebung von Python und ist eine Alternative zur Befehlszeile. Wie der Name schon sagt, ist IDLE sehr nützlich, um neuen Code zu entwickeln oder Python zu lernen. Unter Windows wird dieser mit dem Python-Interpreter ausgeliefert. Bei anderen Betriebssystemen müssen Sie ihn möglicherweise über Ihren Paketmanager installieren.

Die Hauptziele von IDLE sind:

- Texteditor für mehrere Fenster mit Syntaxhervorhebung, Autovervollständigung und intelligentem Einzug
- Python-Shell mit Syntaxhervorhebung
- Integrierter Debugger mit schrittweisen, dauerhaften Haltepunkten und Sichtbarkeit der Aufrufstapel
- Automatische Einrückung (nützlich für Anfänger, die etwas über Pythons Einzug erfahren)
- Speichern Sie das Python-Programm als .py-Dateien, führen Sie sie aus und bearbeiten Sie sie später mit IDLE.

Drücken Sie unter IDLE die Taste `F5` oder `run Python Shell` , um einen Interpreter zu starten. Die Verwendung von IDLE kann für neue Benutzer eine bessere Lernerfahrung sein, da Code beim Schreiben des Benutzers interpretiert wird.

Beachten Sie, dass es viele Alternativen gibt, siehe zum Beispiel diese Diskussion oder diese

Liste .

Fehlerbehebung

- **Windows**

  Unter Windows ist der Standardbefehl `python` . Wenn Sie den Fehler `"'python' is not recognized"` , liegt die wahrscheinlichste Ursache darin, dass sich der Ort von Python nicht in der Umgebungsvariablen `PATH` Ihres Systems befindet. Sie können auf diesen zugreifen, indem Sie mit der rechten Maustaste auf 'Arbeitsplatz' klicken und 'Eigenschaften' auswählen oder über 'Systemsteuerung' zu 'System' navigieren. Klicken Sie auf "Erweiterte Systemeinstellungen" und dann auf "Umgebungsvariablen ...". Bearbeiten Sie die `PATH` Variable, um das Verzeichnis Ihrer Python-Installation sowie den `C:\Python27;C:\Python27\Scripts` (normalerweise `C:\Python27;C:\Python27\Scripts`) `C:\Python27;C:\Python27\Scripts` . Dies erfordert Administratorrechte und erfordert möglicherweise einen Neustart.

  Wenn Sie mehrere Versionen von Python auf demselben Computer verwenden, können Sie eine der `python.exe` Dateien umbenennen. Wenn Sie beispielsweise eine Version `python27.exe` `python27` , wird `python27` zum Python-Befehl für diese Version.

  Sie können auch den Python Launcher für Windows verwenden, der im Installationsprogramm verfügbar ist und standardmäßig enthalten ist. Sie können die Version von Python auswählen, indem Sie `py -[xy]` anstelle von `python[xy]` . Sie können die neueste Version von Python 2 verwenden, indem Sie Skripts mit `py -2` und die neueste Version von Python 3 `py -3` indem Sie Skripts mit `py -3` .

- **Debian / Ubuntu / MacOS**

  In diesem Abschnitt wird davon ausgegangen, dass die Position der ausführbaren `python` Datei der Umgebungsvariablen `PATH` hinzugefügt wurde.

  Wenn Sie Debian / Ubuntu / MacOS verwenden, öffnen Sie das Terminal und geben Sie `python` für Python 2.x oder `python3` für Python 3.x ein.

  Geben Sie `which python` zu sehen, welcher Python-Interpreter verwendet wird.

- **Arch Linux**

  Der Standard-Python für Arch Linux (und seine Nachkommen) ist Python 3. Verwenden Sie daher `python` oder `python3` für Python 3.x und `python2` für Python 2.x.

- **Andere Systeme**

  Python 3 ist manchmal an `python` statt an `python3` gebunden. Um Python 2 auf diesen Systemen, auf denen es installiert ist, zu verwenden, können Sie `python2` .

Datentypen

Eingebaute Typen

Booleaner

`bool` : Ein boolescher Wert von " `True` oder " `False` . Logische Operationen wie `and` , `or` , `not` auf booleans durchgeführt werden.

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

In Python 2.x und Python 3.x ist ein Boolean auch ein `int` . Der `bool` Typ ist eine Unterklasse des `int` Typs, und `True` und `False` sind die einzigen Instanzen:

```
issubclass(bool, int) # True

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

Wenn boolesche Werte in arithmetischen Operationen verwendet werden, werden ihre ganzzahligen Werte (`1` und `0` für `True` und `False`) verwendet, um ein ganzzahliges Ergebnis zurückzugeben:

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

Zahlen

- `int` : Ganzzahl

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

  Ganzzahlen in Python sind von beliebiger Größe.

  Hinweis: In älteren Versionen von Python war ein `long` Typ verfügbar, der sich von `int` . Die beiden wurden vereinheitlicht.

- `float` : Fließkommazahl; Die Genauigkeit hängt von der Implementierung und der Systemarchitektur ab. Bei CPython entspricht der `float` Datentyp einem C-Double.

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

- `complex`

: Komplexe Zahlen

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

Die Operatoren `<` , `<=` , `>` und `>=` `TypeError` eine `TypeError` Exception aus, wenn ein Operand eine komplexe Zahl ist.

Zeichenketten

Python 3.x 3.0

- `str` : eine **Unicode-Zeichenfolge** . Die Art von `'hello'`
- `bytes` : eine **Bytefolge** . Die Art von `b'hello'`

Python 2.x 2.7

- `str` : eine **Bytefolge** . Die Art von `'hello'`
- `bytes` : synonym für `str`
- `unicode` : eine **Unicode-Zeichenfolge** . Die Art von `u'hello'`

Sequenzen und Sammlungen

Python unterscheidet zwischen geordneten Sequenzen und ungeordneten Sammlungen (wie `set` und `dict`).

- Zeichenfolgen (`str` , `bytes` , `unicode`) sind Sequenzen
- `reversed` : Eine umgekehrte Reihenfolge von `str` mit `reversed` Funktion

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

- `tuple` : Eine geordnete Sammlung von `n` Werten eines beliebigen Typs (`n >= 0`).

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

  Unterstützt die Indizierung; unveränderlich; hashable, wenn alle seine Mitglieder hashable sind

- `list` : Eine geordnete Sammlung von `n` Werten (`n >= 0`)

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

  Nicht hashbar; veränderlich.

- `set` : Eine ungeordnete Sammlung eindeutiger Werte. Elemente müssen hashbar sein .

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

- `dict` : Eine ungeordnete Sammlung eindeutiger Schlüssel-Wert-Paare. Schlüssel müssen hashbar sein .

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

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

Ein Objekt ist hashbar, wenn es einen Hashwert hat, der sich während seiner Lebensdauer niemals ändert (es benötigt eine `__hash__()` Methode) und kann mit anderen Objekten verglichen werden (es benötigt eine `__eq__()` Methode). Hashfähige Objekte, die Gleichheit vergleichen, müssen denselben Hashwert haben.

Eingebaute Konstanten

In Verbindung mit den integrierten Datentypen gibt es eine kleine Anzahl von eingebauten Konstanten im integrierten Namespace:

- `True` : Der wahre Wert des eingebauten `bool`
- `False` : Der False-Wert des integrierten Typs `bool`
- `None` : Ein Einzelobjekt, das signalisiert, dass ein Wert nicht vorhanden ist.
- `Ellipsis` oder `...` : Wird in Core Python3 + überall und in Python2.7 + als Teil der Array-Notation verwendet. `numpy` und verwandte Pakete verwenden dies als Referenz für "alles einschließen" in Arrays.
- `NotImplemented` : Ein Singleton, der Python `NotImplemented` , dass eine spezielle Methode die bestimmten Argumente nicht unterstützt, und Python versucht Alternativen, falls verfügbar.

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

Python 3.x 3.0

`None` hat keine natürliche Reihenfolge. Die Verwendung von Sortiervergleichsoperatoren (`<` , `<=` , `>=` , `>`) wird nicht mehr unterstützt und führt zu einem `TypeError` .

Python 2.x 2.7

`None` ist immer kleiner als eine beliebige Zahl (`None < -32` zu `True` ausgewertet).

Testen des Variablentyps

In Python können wir den Datentyp eines Objekts überprüfen Sie die integrierte Funktion mit `type` .

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

In Bedingungsanweisungen ist es möglich, den Datentyp mit einer `isinstance` zu testen. Es wird jedoch normalerweise nicht empfohlen, sich auf den Typ der Variablen zu verlassen.

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

Informationen zu den Unterschieden zwischen `type()` und `isinstance()` Sie unter: Unterschiede zwischen isinstance und type in Python

So testen Sie, ob etwas von `NoneType` :

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

Konvertierung zwischen Datentypen

Sie können eine explizite Datentypkonvertierung durchführen.

Zum Beispiel ist '123' vom Typ `str` und kann mit `int` Funktion in eine Ganzzahl umgewandelt werden.

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

Die Konvertierung von einem Float-String wie '123.456' kann mit der `float` Funktion erfolgen.

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

Sie können auch Sequenz- oder Sammlungstypen konvertieren

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

Expliziter Zeichenfolgentyp bei der Definition von Literalen

Mit einem Buchstaben vor den Anführungszeichen können Sie feststellen, welche Art von Zeichenfolge Sie definieren möchten.

- `b'foo bar'` : Ergebnisse `bytes` in Python 3 `str` in Python 2
- `u'foo bar'` : Ergebnisse `str` in Python 3, `unicode` in Python 2
- `'foo bar'` : Ergebnisse `str`
- `r'foo bar'` : führt zu einer so genannten rohen Zeichenfolge, bei der das `r'foo bar'` Sonderzeichen nicht erforderlich ist. Während der Eingabe wird alles wörtlich genommen

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

Veränderliche und unveränderliche Datentypen

Ein Objekt wird als *veränderlich bezeichnet,* wenn es geändert werden kann. Wenn Sie beispielsweise eine Liste an eine Funktion übergeben, kann die Liste geändert werden:

```
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
```

Ein Objekt wird als *unveränderlich bezeichnet,* wenn es in keiner Weise geändert werden kann. Zum Beispiel sind Ganzzahlen unveränderlich, da sie nicht geändert werden können:

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

Beachten Sie, dass **Variablen** selbst veränderbar sind, sodass wir die *Variable* `x` neu zuweisen können. Dies ändert jedoch nicht das Objekt, auf das `x` zuvor gezeigt hat. Es wurde nur `x` auf ein neues Objekt gesetzt.

Datentypen, deren Instanzen veränderlich sind, werden als *veränderliche Datentypen* bezeichnet und für unveränderliche Objekte und Datentypen gleichermaßen.

Beispiele für unveränderliche Datentypen:

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

Beispiele für veränderliche Datentypen:

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

Eingebaute Module und Funktionen

Ein Modul ist eine Datei, die Python-Definitionen und Anweisungen enthält. Funktion ist ein Stück Code, der einige Logik ausführt.

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

Um die eingebaute Funktion in Python zu überprüfen, können wir `dir()`. Wenn Sie ohne Argument aufgerufen werden, geben Sie die Namen im aktuellen Bereich zurück. Andernfalls wird eine alphabetische Liste von Namen zurückgegeben, die (einige) das Attribut des angegebenen Objekts sowie die von ihm erreichbaren Attribute enthalten.

```
>>> 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'
]
```

Um die Funktionalität einer beliebigen Funktion kennen, können wir in Funktion nutzen gebaut `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.
```

Eingebaute Module enthalten zusätzliche Funktionalitäten. Um beispielsweise Quadratwurzeln einer Zahl zu erhalten, müssen wir ein `math` hinzufügen.

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

Um alle Funktionen in einem Modul zu kennen, können wir die Funktionsliste einer Variablen zuweisen und dann die Variable drucken.

```
>>> 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']
```

Es scheint, dass `__doc__` nützlich ist, um beispielsweise in Funktionen einige Dokumentation bereitzustellen

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

Neben Funktionen kann Dokumentation auch in Modulen bereitgestellt werden. Wenn Sie also eine Datei namens `helloWorld.py` wie `helloWorld.py` :

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

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

Sie können auf die Dokumentstrings folgendermaßen zugreifen:

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

- Für jeden benutzerdefinierten Typ können seine Attribute, die Attribute seiner Klasse und

rekursiv die Attribute der Basisklassen seiner Klasse mit dir () abgerufen werden.

```
>>> 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__']
```

Jeder Datentyp kann einfach mit einer eingebauten Funktion namens `str` in String konvertiert werden. Diese Funktion wird standardmäßig aufgerufen, wenn ein Datentyp an den `print`

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

Einrückung blockieren

Python verwendet Einrückungen zum Definieren von Steuerungs- und Schleifenkonstrukten. Dies trägt zur Lesbarkeit von Python bei, erfordert jedoch, dass der Programmierer der Verwendung von Leerzeichen große Aufmerksamkeit widmet. Die Fehlkalibrierung des Editors kann daher zu Code führen, der sich unerwartet verhält.

Python verwendet das Doppelpunkt - Zeichen (`:`) und Vertiefung zu zeigen , wo Codeblocks beginnen und enden (Wenn Sie aus einer anderen Sprache kommen, nicht zu verwechseln diese mit irgendwie mit dem zusammenhängt ternären Operator). Das heißt, Blöcke in Python, wie z. B. Funktionen, Schleifen, `if` Klauseln und andere Konstrukte, haben keine Endekennungen. Alle Blöcke beginnen mit einem Doppelpunkt und enthalten dann die eingerückten Zeilen darunter.

Zum Beispiel:

```
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
```

oder

```
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
```

Blöcke, die genau eine einzeilige Anweisung enthalten, können in dieselbe Zeile eingefügt werden, obwohl dieses Formular im Allgemeinen nicht als guter Stil gilt:

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

Der Versuch, dies mit mehr als einer einzelnen Anweisung zu tun, funktioniert *nicht* :

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

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

Ein leerer Block verursacht einen `IndentationError` . Verwenden Sie `pass` (ein Befehl, der nichts tut), wenn Sie einen Block ohne Inhalt haben:

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

Leerzeichen vs. Tabs

Kurz gesagt: Verwenden Sie **immer** 4 Felder für die Einrückung.

Die ausschließliche Verwendung von Registerkarten ist möglich, PEP 8 , der Style-Guide für Python-Code, besagt, dass Leerzeichen bevorzugt werden.

Python 3.x 3.0

In Python 3 ist das Mischen von Tabs und Leerzeichen für die Einrückung nicht zulässig. In diesem Fall wird ein Fehler bei der Kompilierung generiert: `Inconsistent use of tabs and spaces in indentation` und das Programm wird nicht ausgeführt.

Python 2.x 2.7

Python 2 erlaubt das Mischen von Tabs und Leerzeichen beim Einrücken. Dies wird dringend empfohlen. Das Tabulatorzeichen schließt den vorherigen Einzug mit einem Vielfachen von **8** Leerzeichen ab . Da Editoren üblicherweise so konfiguriert sind, dass Registerkarten als mehrere von **vier** Leerzeichen angezeigt werden, kann dies zu geringfügigen Fehlern führen.

Zitieren von PEP 8 :

> Wenn Sie den Python 2-Befehlszeileninterpreter mit der Option `-t` aufrufen, werden Warnungen über Code ausgegeben, der Tabulatoren und Leerzeichen illegal mischt. Bei Verwendung von `-tt` diese Warnungen zu Fehlern. Diese Optionen werden dringend empfohlen!

Viele Editoren haben "Tabs to Spaces" -Konfiguration. Wenn Sie den Editor konfiguriert, sollte man zwischen den Tab - *Zeichen* (‚\ t‘) und der `Tab` - Taste unterscheiden.

- Die *Tabulatorzeichen* sollten zu zeigen , 8 Räume so konfiguriert werden, um die Sprachsemantik entsprechen - zumindest in den Fällen , wenn (versehentliche) gemischte Einrückung möglich ist. Editoren können das Tabulatorzeichen auch automatisch in Leerzeichen umwandeln.
- Es kann jedoch hilfreich sein, den Editor so zu konfigurieren, dass durch Drücken der `Tabulatortaste` 4 Leerzeichen anstelle eines Tabulatorzeichens eingefügt werden.

Python Quellcode mit einer Mischung aus tabs geschrieben und Räumen oder mit Nicht-Standard - Zahl der Einrückung Räume kann Pep8-konforme Verwendung gemacht werden autopep8 . (Eine weniger leistungsfähige Alternative ist bei den meisten Python-Installationen enthalten: reindent.py.)

Auflistungsarten

Es gibt eine Reihe von Auflistungstypen in Python. Während Typen wie `int` und `str` einen einzelnen Wert enthalten, enthalten Auflistungstypen mehrere Werte.

Listen

Der `list` ist wahrscheinlich der am häufigsten verwendete Auflistungstyp in Python. Trotz ihres Namens ähnelt eine Liste eher einem Array in anderen Sprachen, meistens JavaScript. In Python ist eine Liste lediglich eine geordnete Sammlung gültiger Python-Werte. Eine Liste kann erstellt werden, indem Werte, die durch Kommas getrennt sind, in eckige Klammern gesetzt werden:

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

Eine Liste kann leer sein:

```
empty_list = []
```

Die Elemente einer Liste sind nicht auf einen einzelnen Datentyp beschränkt. Dies ist sinnvoll, wenn Python eine dynamische Sprache ist:

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

Eine Liste kann eine andere Liste als Element enthalten:

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

Auf die Elemente einer Liste kann über einen *Index* oder eine numerische Darstellung ihrer Position zugegriffen werden. Listen in Python sind *nullindiziert,* was bedeutet, dass das erste Element in der Liste den Index 0 hat, das zweite Element den Index 1 und so weiter:

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

Indizes können auch negativ sein, dh ab dem Ende der Liste zählen (`-1` ist der Index des letzten Elements). Verwenden Sie also die Liste aus dem obigen Beispiel:

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

Listen sind veränderbar, sodass Sie die Werte in einer Liste ändern können:

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

Außerdem können Sie Elemente zu einer Liste hinzufügen und / oder daraus entfernen:

Objekt mit `L.append(object)` an das Ende der Liste `L.append(object)` , gibt `None` .

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

Fügen Sie ein neues Element zur Liste an einem bestimmten Index hinzu. `L.insert(index, object)`

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

Entfernen Sie das erste Vorkommen eines Werts mit `L.remove(value) None`

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

Rufen Sie den Index in der Liste des ersten Elements ab, dessen Wert x ist. Wenn kein solcher Artikel vorhanden ist, wird ein Fehler angezeigt.

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

Länge der Liste zählen

```
len(names)
6
```

Zählen Sie das Vorkommen eines Elements in der Liste

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

Kehren Sie die Liste um

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

Artikel mit Index `L.pop([index])` der letzte Artikel) mit `L.pop([index])` und den Artikel zurückgeben

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

Sie können die Listenelemente wie folgt durchlaufen:

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

Tuples

Ein `tuple` ähnelt einer Liste mit der Ausnahme, dass es eine feste Länge und unveränderlich ist. Daher können die Werte im Tupel nicht geändert werden, und die Werte können nicht zum Tupel hinzugefügt oder daraus entfernt werden. Tupel werden im Allgemeinen für kleine Sammlungen von Werten verwendet, die nicht geändert werden müssen, z. B. eine IP-Adresse und einen Port. Tupel werden mit Klammern anstelle von eckigen Klammern dargestellt:

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

Die gleichen Indexierungsregeln für Listen gelten auch für Tupel. Tupel können auch verschachtelt sein und die Werte können alle gültigen gültigen Python-Werte sein.

Ein Tupel mit nur einem Member muss folgendermaßen definiert werden (beachten Sie das Komma):

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

oder

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

oder einfach mit der `tuple`

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

Wörterbücher

Ein `dictionary` in Python ist eine Sammlung von Schlüssel-Wert-Paaren. Das Wörterbuch ist von geschweiften Klammern umgeben. Jedes Paar wird durch ein Komma getrennt, und der Schlüssel und der Wert werden durch einen Doppelpunkt getrennt. Hier ist ein Beispiel:

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

Um einen Wert zu erhalten, beziehen Sie sich auf seinen Schlüssel:

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

Sie können auch alle Schlüssel in einem Wörterbuch abrufen und diese dann durchlaufen:

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

Wörterbücher ähneln stark der JSON-Syntax. Das native `json` Modul in der Python-Standardbibliothek kann zum Konvertieren zwischen JSON und Wörterbüchern verwendet werden.

einstellen

Ein `set` ist eine Sammlung von Elementen ohne Wiederholungen und ohne Einfügereihenfolge, aber sortierte Reihenfolge. Sie werden in Situationen verwendet, in denen es nur wichtig ist, dass einige Dinge zusammengruppiert werden und nicht in welcher Reihenfolge. Bei großen Datengruppen ist es viel schneller zu prüfen, ob ein Element in einer `set` ist oder nicht `set` als dasselbe für eine `list` .

Das Definieren eines `set` ist dem Definieren eines `dictionary` sehr ähnlich:

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

Oder Sie können einen `set` mit einer vorhandenen `list` erstellen:

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

Überprüfen Sie die Mitgliedschaft in der `set` mit `in` :

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

Sie können eine `set` genau wie eine Liste durchlaufen, aber denken Sie daran: Die Werte werden in einer beliebigen, von der Implementierung definierten Reihenfolge angezeigt.

defaultdict

Ein `defaultdict` ist ein Wörterbuch mit einem Standardwert für Schlüssel, sodass auf Schlüssel, für die kein expliziter Wert definiert wurde, fehlerfrei zugegriffen werden kann. `defaultdict` ist besonders nützlich, wenn es sich bei den Werten im Wörterbuch um Sammlungen (Listen, Diagramme usw.) handelt, die nicht jedes Mal initialisiert werden müssen, wenn ein neuer Schlüssel verwendet wird.

Ein `defaultdict` niemals einen KeyError aus. Für einen nicht vorhandenen Schlüssel wird der Standardwert zurückgegeben.

Betrachten Sie beispielsweise das folgende Wörterbuch

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

Wenn wir versuchen, auf einen nicht vorhandenen Schlüssel zuzugreifen, gibt Python einen Fehler wie folgt zurück

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

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

KeyError: 'Alabama'
```

Versuchen wir es mit einem `defaultdict` . Es befindet sich im Kollektionsmodul.

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

Wir haben hier einen Standardwert (**Boston**) festgelegt, falls der Schlüssel nicht vorhanden ist. Füllen Sie nun das Diktat wie zuvor:

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

Wenn wir versuchen, mit einem nicht vorhandenen Schlüssel auf das Diktat zuzugreifen, gibt Python den Standardwert zurück, z. B. Boston

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

und gibt die erstellten Werte für einen vorhandenen Schlüssel wie ein normales `dictionary`

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

Hilfsprogramm

Python hat mehrere Funktionen, die in den Interpreter integriert sind. Wenn Sie Informationen zu Schlüsselwörtern, integrierten Funktionen, Modulen oder Themen erhalten möchten, öffnen Sie eine Python-Konsole und geben Sie Folgendes ein:

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

Sie erhalten Informationen, indem Sie die Schlüsselwörter direkt eingeben:

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

oder innerhalb des Dienstprogramms:

```
help> help
```

die eine Erklärung zeigen wird:

```
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)
```

Sie können auch Unterklassen von Modulen anfordern:

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

Sie können die Hilfe verwenden, um auf die Dokumentfolgen der verschiedenen importierten Module zuzugreifen. Versuchen Sie beispielsweise Folgendes:

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

und Sie erhalten einen Fehler

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

Nun erhalten Sie eine Liste der verfügbaren Methoden im Modul, jedoch erst, nachdem Sie es importiert haben.

Schließen Sie den Helfer mit `quit`

Ein Modul erstellen

Ein Modul ist eine importierbare Datei, die Definitionen und Anweisungen enthält.

Ein Modul kann durch Erstellen einer `.py` Datei erstellt werden.

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

Funktionen in einem Modul können durch Importieren des Moduls verwendet werden.

Für von Ihnen erstellte Module müssen sich diese im selben Verzeichnis befinden wie die Datei, in die Sie sie importieren. (Sie können sie jedoch auch mit den im Lieferumfang enthaltenen Modulen im Python-Verzeichnis lib ablegen, sollten aber möglichst vermieden werden.)

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

Module können von anderen Modulen importiert werden.

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

Bestimmte Funktionen eines Moduls können importiert werden.

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

Module können Aliasing sein.

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

Ein Modul kann ein eigenständiges ausführbares Skript sein.

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

Starte es!

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

Wenn sich das Modul in einem Verzeichnis befindet und von Python erkannt werden muss, sollte das Verzeichnis eine Datei mit dem Namen `__init__.py` .

String-Funktion - str () und repr ()

Es gibt zwei Funktionen, mit denen eine lesbare Darstellung eines Objekts erhalten werden kann.

`repr(x)` ruft `x.__repr__()` : eine Darstellung von `x` . `eval` konvertiert das Ergebnis dieser Funktion normalerweise zurück in das ursprüngliche Objekt.

`str(x)` ruft `x.__str__()` : Eine vom Menschen lesbare Zeichenfolge, die das Objekt beschreibt. Dies kann einige technische Details auslassen.

---

repr ()

Bei vielen Typen versucht diese Funktion, eine Zeichenfolge zurückzugeben, die ein Objekt mit demselben Wert ergibt, wenn es an `eval()` . Ansonsten handelt es sich bei der Darstellung um eine in spitze Klammern eingeschlossene Zeichenfolge, die den Namen des Objekttyps sowie zusätzliche Informationen enthält. Dies beinhaltet häufig den Namen und die Adresse des Objekts.

str ()

Bei Strings wird der String selbst zurückgegeben. Der Unterschied zwischen diesem und `repr(object)` besteht darin, dass `str(object)` nicht immer versucht, einen String zurückzugeben, der für `eval()` zulässig ist. Vielmehr ist es das Ziel, eine druckbare oder "lesbare" Zeichenfolge zurückzugeben. Wenn kein Argument angegeben ist, wird der leere String `''` .

Beispiel 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
```

Beispiel 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)'
```

Beim Schreiben einer Klasse können Sie diese Methoden überschreiben, um zu tun, was Sie möchten:

```
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)
```

Mit der obigen Klasse können wir die Ergebnisse sehen:

```
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
```

Installation externer Module mit pip

`pip` ist dein Freund, wenn du ein Paket aus der Fülle von Auswahlmöglichkeiten im python package index (PyPI) installieren musst. `pip` ist bereits installiert, wenn Sie Python 2> = 2.7.9 oder Python 3> = 3.4 verwenden, das von python.org heruntergeladen wurde. Bei Computern mit Linux oder einem anderen * nix mit einem systemeigenen Paketmanager muss `pip` häufig manuell installiert werden.

Bei Instanzen, auf denen sowohl Python 2 als auch Python 3 installiert sind, bezieht sich `pip` häufig auf Python 2 und `pip3` auf Python 3. Bei der Verwendung von `pip` werden nur Pakete für Python 2 installiert, und `pip3` installiert nur Pakete für Python 3.

Paket suchen / installieren

Die Suche nach einem Paket ist so einfach wie das Tippen

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

Das Installieren eines Pakets ist so einfach wie das Eintippen *(in einer Terminal- / Eingabeaufforderung, nicht im Python-Interpreter).*

```
$ 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
```

Dabei ist `xxx` die Versionsnummer des Pakets, das Sie installieren möchten.

Wenn sich Ihr Server hinter einem Proxy befindet, können Sie das Paket mit dem folgenden Befehl installieren:

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

Installierte Pakete aktualisieren

Wenn neue Versionen installierter Pakete angezeigt werden, werden sie nicht automatisch auf Ihrem System installiert. Um einen Überblick darüber zu erhalten, welche Ihrer installierten Pakete veraltet sind, führen Sie Folgendes aus:

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

Um ein bestimmtes Paket zu aktualisieren, verwenden Sie

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

Das Aktualisieren aller veralteten Pakete ist keine Standardfunktion von `pip` .

Pip aufrüsten

Sie können Ihre vorhandene Pip-Installation mit den folgenden Befehlen aktualisieren

- Unter Linux oder Mac OS X:

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

  Möglicherweise müssen Sie `sudo` with pip auf einigen Linux-Systemen verwenden

- Unter Windows:

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

  oder

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

Weitere Informationen zu pip finden Sie hier .

Installation von Python 2.7.x und 3.x

> **Hinweis** : Die folgenden Anweisungen werden für Python 2.7 geschrieben (sofern nicht anders angegeben): Anweisungen für Python 3.x sind ähnlich.

WINDOWS

Laden Sie zunächst die neueste Version von Python 2.7 von der offiziellen Website (https://www.python.org/downloads/) herunter. Die Version wird als MSI-Paket bereitgestellt. Um es manuell zu installieren, doppelklicken Sie einfach auf die Datei.

Standardmäßig wird Python in einem Verzeichnis installiert:

```
C:\Python27\
```

Warnung: Die Installation ändert die Umgebungsvariable PATH nicht automatisch.

Angenommen, Ihre Python-Installation befindet sich in C: \ Python27, und fügen Sie dies zu Ihrem PFAD hinzu:

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

Um zu prüfen, ob die Python-Installation gültig ist, schreiben Sie in cmd:

```
python --version
```

Python 2.x und 3.x Seite an Seite

So installieren und verwenden Sie Python 2.x und 3.x nebeneinander auf einem Windows-Computer:

1. Installieren Sie Python 2.x mit dem MSI-Installationsprogramm.
    - Stellen Sie sicher, dass Python für alle Benutzer installiert ist.
    - Optional: Fügen Sie Python zu `PATH` , damit Python 2.x von der Befehlszeile aus mit `python` .
2. Installieren Sie Python 3.x mit dem entsprechenden Installationsprogramm.
    - Stellen Sie erneut sicher, dass Python für alle Benutzer installiert ist.
    - Optional: Fügen Sie Python zu `PATH` , damit Python 3.x von der Befehlszeile aus mit `python` . Dies kann die `PATH` Einstellungen von Python 2.x außer Kraft setzen. Überprüfen Sie deshalb Ihren `PATH` und stellen Sie sicher, dass er nach Ihren Präferenzen konfiguriert ist.
    - Stellen Sie sicher, dass Sie den `py launcher` für alle Benutzer installieren.

Python 3 installiert das Python-Startprogramm, mit dem Python 2.x und Python 3.x austauschbar über die Befehlszeile gestartet werden können:

```
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.
>>>
```

Um die entsprechende Version von `pip` für eine bestimmte Python-Version zu verwenden, verwenden Sie:

```
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

Die neuesten Versionen von CentOS, Fedora, Redhat Enterprise (RHEL) und Ubuntu werden mit Python 2.7 geliefert.

Um Python 2.7 manuell unter Linux zu installieren, führen Sie im Terminal Folgendes aus:

```
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
```

Fügen Sie auch den Pfad des neuen Python in der Umgebungsvariable PATH hinzu. Wenn sich neuer Python in `/root/python-2.7.X` führen Sie den `export PATH = $PATH:/root/python-2.7.X`

Um zu prüfen, ob die Python-Installation ein gültiges Schreiben im Terminal ist:

```
python --version
```

Ubuntu (aus Quelle)

Wenn Sie Python 3.6 benötigen, können Sie es wie unten gezeigt von der Quelle installieren (Ubuntu 16.10 und 17.04 haben die Version 3.6 im universellen Repository). Die folgenden Schritte müssen für Ubuntu 16.04 und niedrigere Versionen befolgt werden:

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

Mac OS

Wir sprechen zwar davon, dass macOS mit Python 2.7.10 installiert ist, aber diese Version ist

veraltet und gegenüber dem regulären Python geringfügig geändert.

> Die mit OS X gelieferte Version von Python eignet sich hervorragend zum Lernen, nicht aber für die Entwicklung. Die mit OS X gelieferte Version ist möglicherweise nicht mehr aktuell als die offizielle Python-Version, die als stabile Produktionsversion gilt. (Quelle)

Homebrew installieren:

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

Installieren Sie Python 2.7:

```
brew install python
```

Verwenden Sie für Python 3.x stattdessen den Befehl `brew install python3` .

Erste Schritte mit Python Language online lesen: https://riptutorial.com/de/python/topic/193/erste-schritte-mit-python-language

Kapitel 2: * args und ** kwargs

Bemerkungen

Es gibt ein paar Dinge zu beachten:

1. Die Namen `args` und `kwargs` werden `kwargs` verwendet, sie sind nicht Teil der Sprachspezifikation. Somit sind diese gleichwertig:

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

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

2. Sie dürfen nicht mehr als einen `args` oder mehr als einen `kwargs` Parameter haben (diese sind jedoch nicht erforderlich)

```
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. Wenn auf `*args` ein Positionsargument folgt, handelt es sich nur um Argumente, die nur nach Namen übergeben werden können. Ein einzelner Stern kann anstelle von `*args` , um Werte als Schlüsselwortargumente zu erzwingen, ohne eine variadische Parameterliste anzugeben. Nur-Parameterparameterlisten sind nur in Python 3 verfügbar.

```
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` müssen in der Parameterliste den letzten Platz `**kwargs` .

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

Examples

Verwenden von * args beim Schreiben von Funktionen

Sie können den Stern * beim Schreiben einer Funktion verwenden, um alle positionellen (dh unbenannten) Argumente in einem Tupel zu sammeln:

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

Aufrufmethode:

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

In diesem Aufruf wird farg wie immer zugewiesen, und die beiden anderen werden in der Reihenfolge, in der sie empfangen wurden, dem Args-Tupel zugeführt.

Verwenden von ** Warnungen beim Schreiben von Funktionen

Sie können eine Funktion definieren, die eine beliebige Anzahl von Schlüsselwort (benannten) Argumenten verwendet, indem Sie den doppelten Stern `**` vor einem Parameternamen verwenden:

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

Beim Aufruf der Methode erstellt Python ein Wörterbuch aller Schlüsselwortargumente und macht es im Funktionskörper verfügbar:

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

Beachten Sie, dass der ** kwargs-Parameter in der Funktionsdefinition immer der letzte

Parameter sein muss und nur mit den Argumenten übereinstimmt, die nach den vorherigen übergeben wurden.

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

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

Innerhalb des Funktionskörpers wird `kwargs` wie ein Wörterbuch manipuliert. Um auf einzelne Elemente in `kwargs` zuzugreifen, `kwargs` Sie diese wie bei einem normalen Wörterbuch:

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

`print_kwargs(a="two", b=1)` aufrufen, wird die folgende Ausgabe `print_kwargs(a="two", b=1)` :

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

Verwendung von * args beim Aufruf von Funktionen

Ein häufiger Anwendungsfall für `*args` in einer Funktionsdefinition ist das Delegieren der Verarbeitung an eine umschlossene oder geerbte Funktion. Ein typisches Beispiel ist die `__init__` Methode einer Klasse

```
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
```

Hier wird der `a` wird der Parameter von der Kind - Klasse , nachdem alle anderen Argumente verarbeitet (Lage- und Keyword) geleitet werden , auf - und verarbeitet - der Basisklasse.

Zum Beispiel:

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

Was hier passiert, ist, dass die Klasse `B` `__init__` die Argumente `1, 2, 3` sieht. Es weiß, dass es ein positionelles Argument (`a`) braucht, daher greift es das erste Argument, das in (`1`) übergeben wurde, also im Bereich der Funktion `a == 1` .

Als nächstes sieht es, dass es eine beliebige Anzahl von Positionsargumenten (`*args`) nehmen muss, sodass es den Rest der übergebenen Positionsargumente (`1, 2`) übernimmt und in `*args` stopft. Nun (im Umfang der Funktion) `args == [2, 3]` .

Dann ruft es die Funktion `__init__` Klasse `A` mit `*args` . Python sieht das `*` vor args und "entpackt" die Liste in Argumente. Wenn in diesem Beispiel die `__init__` Funktion der Klasse `B` die `__init__` Funktion der Klasse `A` `__init__` , werden die Argumente `2, 3` (dh `A(2, 3)`) übergeben.

Schließlich setzt es seine eigene `x` Eigenschaft auf das erste Positionsargument `a` , das gleich `1` .

Verwenden von ** Warnungen beim Aufruf von Funktionen

Sie können ein Wörterbuch verwenden, um den Parametern der Funktion Werte zuzuweisen. Verwenden Sie den Parameternamen als Schlüssel im Wörterbuch und den Wert dieser an jeden Schlüssel gebundenen Argumente:

```
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)
```

Verwendung von * args beim Aufruf von Funktionen

Die Verwendung des Operators `*` für ein Argument beim Aufruf einer Funktion hat den Effekt, dass die Liste oder ein Tupel-Argument entpackt wird

```
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
```

Beachten Sie, dass die Länge des mit Sternchen versehenen Arguments der Anzahl der Argumente der Funktion entsprechen muss.

Ein übliches Python-Idiom besteht darin, den Entpackungsoperator `*` mit der `zip` Funktion zu verwenden, um seine Auswirkungen umzukehren:

```
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)
```

Nur für Schlüsselwörter und für Schlüsselwörter erforderliche Argumente

In Python 3 können Sie Funktionsargumente definieren, die auch ohne Standardwerte nur per Schlüsselwort zugewiesen werden können. Dies erfolgt durch Verwendung von Stern *, um zusätzliche Positionsparameter zu verwenden, ohne die Schlüsselwortparameter festzulegen. Alle Argumente nach dem * sind Argumente, die nur aus einem Schlüsselwort bestehen. Beachten Sie, dass Argumente, die nur aus Schlüsselwörtern bestehen, keinen Standardwert haben, beim Aufruf der Funktion jedoch erforderlich sind.

```
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
```

Kwarg-Werte mit einem Wörterbuch füllen

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

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

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

** Warnungen und Standardwerte

So verwenden Sie Standardwerte mit ** kwargs

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

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

* args und ** kwargs online lesen: https://riptutorial.com/de/python/topic/2475/--args-und----kwargs

Kapitel 3: 2to3 Werkzeug

Syntax

- $ 2to3 [-options] Pfad / to / file.py

Parameter

Parameter	Beschreibung
Dateiname / Verzeichnisname	2to3 akzeptiert eine Liste von Dateien oder Verzeichnissen, die als Argument umgewandelt werden sollen. Die Verzeichnisse werden rekursiv für Python-Quellen durchlaufen.
Möglichkeit	**Option Beschreibung**
-f FIX, --fix = FIX	Geben Sie die anzuwendenden Transformationen an. Standard: Alle. Liste der verfügbaren Transformationen mit `--list-fixes`
-j PROZESSE, --processes = PROZESSE	Führen Sie 2to3 gleichzeitig aus
-x NOFIX, --nofix = NOFIX	Eine Transformation ausschließen
-l, --list-fixes	Liste der verfügbaren Transformationen
-p, --print-Funktion	Ändern Sie die Grammatik so, dass `print()` als Funktion betrachtet wird
-v, --verbose	Ausführlichere Ausgabe
--keine Unterschiede	Geben Sie keine Differenzen des Refactorings aus
-w	Schreibe modifizierte Dateien zurück
-n, --nobackups	Erstellen Sie keine Sicherungen von geänderten Dateien
-o OUTPUT_DIR, --output-dir = OUTPUT_DIR	Platzieren Sie die Ausgabedateien in diesem Verzeichnis, anstatt die Eingabedateien zu überschreiben. Erfordert das Flag `-n` , da Sicherungsdateien nicht erforderlich sind, wenn die Eingabedateien nicht geändert werden.
-W, --write-unverändert-Dateien	Ausgabedateien schreiben, selbst wenn keine Änderungen erforderlich waren. Nützlich bei `-o` damit ein vollständiger Quellbaum übersetzt und kopiert wird. Impliziert `-w` .

Parameter	Beschreibung
--add-Suffix = ADD_SUFFIX	Geben Sie eine Zeichenfolge an, die an alle Ausgabedateinamen angehängt werden soll. Benötigt `-n` wenn es nicht leer ist. Beispiel: `--add-suffix='3'` generiert `.py3` Dateien.

Bemerkungen

Das 2to3-Tool ist ein Python-Programm, mit dem der in Python 2.x geschriebene Code in Python 3.x-Code konvertiert wird. Das Tool liest den Quellcode von Python 2.x und wendet eine Reihe von Fixierern an, um diesen in gültigen Python 3.x-Code umzuwandeln.

Das 2to3-Tool ist in der Standardbibliothek als lib2to3 verfügbar. Es enthält einen umfangreichen Satz Fixierer, der nahezu den gesamten Code verarbeiten kann. Da es sich bei lib2to3 um eine generische Bibliothek handelt, können Sie Ihre eigenen Fixer für 2to3 schreiben.

Examples

Grundlegende Verwendung

Betrachten Sie den folgenden Python2.x-Code. Speichern Sie die Datei als `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)
```

In der obigen Datei befinden sich mehrere inkompatible Zeilen. Die Methode `raw_input()` wurde in Python 3.x durch `input()` und `print` ist keine Anweisung mehr, sondern eine Funktion. Dieser Code kann mit dem 2to3-Tool in Python 3.x-Code konvertiert werden.

Unix

```
$ 2to3 example.py
```

Windows

```
> path/to/2to3.py example.py
```

Wenn Sie den obigen Code ausführen, werden die Unterschiede zur ursprünglichen Quelldatei ausgegeben (siehe unten).

```
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
```

Die Änderungen können mit der Markierung -w in die Quelldatei zurückgeschrieben werden. Eine Sicherungskopie der Originaldatei namens `example.py.bak` wird erstellt, sofern das Flag -n nicht angegeben ist.

Unix

```
$ 2to3 -w example.py
```

Windows

```
> path/to/2to3.py -w example.py
```

Jetzt wurde die Datei `example.py` von Python 2.x in Python 3.x-Code konvertiert.

Nach Beendigung enthält `example.py` den folgenden gültigen Python3.x-Code:

Python 3.x 3.0

```
def greet(name):
    print("Hello, {0}!".format(name))
print("What's your name?")
name = input()
greet(name)
```

2to3 Werkzeug online lesen: https://riptutorial.com/de/python/topic/5320/2to3-werkzeug

Kapitel 4: Abstrakte Basisklassen (abc)

Examples

Einstellen der ABCMeta-Metaklasse

Abstrakte Klassen sind Klassen, die vererbt werden sollen, vermeiden jedoch die Implementierung bestimmter Methoden und hinterlassen nur Methodensignaturen, die von Unterklassen implementiert werden müssen.

Abstrakte Klassen sind nützlich, um Klassenabstraktionen auf hoher Ebene zu definieren und durchzusetzen, ähnlich dem Konzept von Schnittstellen in typisierten Sprachen, ohne dass eine Methodenimplementierung erforderlich ist.

Ein konzeptioneller Ansatz zum Definieren einer abstrakten Klasse besteht darin, die Klassenmethoden auszulagern und dann einen NotImplementedError auszulösen, wenn darauf zugegriffen wird. Dadurch wird verhindert, dass untergeordnete Klassen auf übergeordnete Methoden zugreifen, ohne sie vorher zu überschreiben. So wie:

```
class Fruit:

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


class Apple(Fruit):
    pass


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

Das Erstellen einer abstrakten Klasse auf diese Weise verhindert die unzulässige Verwendung von Methoden, die nicht überschrieben werden, und ermutigt sicherlich dazu, Methoden in untergeordneten Klassen zu definieren, erzwingen jedoch keine Definition. Mit dem Modul `abc` können wir verhindern, dass untergeordnete Klassen instanziiert werden, wenn sie die abstrakten Klassenmethoden ihrer Eltern und Vorfahren nicht überschreiben:

```
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.
```

Es ist jetzt möglich, die Unterklasse einfach zu überschreiben und zu überschreiben:

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

Warum / Wie werden ABCMeta und @abstractmethod verwendet?

Abstrakte Basisklassen (ABCs) erzwingen, welche abgeleiteten Klassen bestimmte Methoden der Basisklasse implementieren.

Um zu verstehen, wie das funktioniert und warum wir es verwenden sollten, betrachten wir ein Beispiel, das Van Rossum gefallen würde. Nehmen wir an, wir haben eine Basisklasse "MontyPython" mit zwei Methoden (joke & punchline), die von allen abgeleiteten Klassen implementiert werden müssen.

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

    def punchline(self):
        raise NotImplementedError()

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

Wenn wir ein Objekt instanziieren und zwei Methoden aufrufen, erhalten wir (wie erwartet) einen Fehler mit der `punchline()` -Methode.

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

Dies ermöglicht es uns jedoch immer noch, ein Objekt der ArgumentClinic-Klasse zu instanziieren, ohne einen Fehler zu erhalten. Tatsächlich erhalten wir keine Fehlermeldung, bis wir die Punchline () suchen.

Dies wird durch die Verwendung des ABC-Moduls (Abstract Base Class) vermieden. Mal sehen, wie das mit demselben Beispiel funktioniert:

```
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"
```

Wenn wir dieses Mal versuchen, ein Objekt aus der unvollständigen Klasse zu instanziieren, erhalten wir sofort einen TypeError!

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

In diesem Fall ist es einfach, die Klasse abzuschließen, um alle TypeErrors zu vermeiden:

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

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

Dieses Mal, wenn Sie ein Objekt instanziieren, funktioniert es!

Abstrakte Basisklassen (abc) online lesen: https://riptutorial.com/de/python/topic/5442/abstrakte-basisklassen--abc-

Kapitel 5: Abstrakter Syntaxbaum

Examples

Analysieren Sie Funktionen in einem Python-Skript

Dieser analysiert ein Python-Skript und meldet für jede definierte Funktion die Zeilennummer, an der die Funktion begann, wo die Signatur endet, wo der Docstring endet und wo die Funktionsdefinition endet.

```
#!/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)
```

Abstrakter Syntaxbaum online lesen: https://riptutorial.com/de/python/topic/5370/abstrakter-syntaxbaum

Kapitel 6: Ähnlichkeiten in der Syntax, Bedeutungsunterschiede: Python vs. JavaScript

Einführung

Es kommt manchmal vor, dass zwei Sprachen denselben oder einem ähnlichen Syntaxausdruck unterschiedliche Bedeutungen zuweisen. Wenn beide Sprachen für einen Programmierer von Interesse sind, hilft das Verdeutlichen dieser Verzweigungspunkte, die beiden Sprachen in ihren Grundlagen und Feinheiten zu verstehen.

Examples

`in` mit Listen

```
2 in [2, 3]
```

In Python wird dies als True ausgewertet, in JavaScript jedoch als False. Dies liegt daran, dass in Python in Checks geprüft wird, ob ein Wert in einer Liste enthalten ist, also 2 als erstes Element in [2, 3] steht. In JavaScript in wird mit Objekten verwendet und überprüft, ob ein Objekt die Eigenschaft mit dem durch den Wert angegebenen Namen enthält. JavaScript betrachtet also [2, 3] als Objekt oder eine Schlüsselwertzuordnung wie folgt:

```
{'0': 2, '1': 3}
```

und prüft, ob es eine Eigenschaft oder einen Schlüssel '2' enthält. Die Ganzzahl 2 wird stumm in die Zeichenfolge '2' umgewandelt.

Ähnlichkeiten in der Syntax, Bedeutungsunterschiede: Python vs. JavaScript online lesen: https://riptutorial.com/de/python/topic/10766/ahnlichkeiten-in-der-syntax--bedeutungsunterschiede--python-vs--javascript

Kapitel 7: Alternativen zum Wechseln von Anweisungen aus anderen Sprachen

Bemerkungen

Es gibt *KEINE* switch - Anweisung in Python als Sprache Design - Wahl. Es wurde ein PEP (PEP-3103) für das abgelehnte Thema erstellt.

In Python finden Sie eine Vielzahl von Rezepten, wie Sie Ihre eigenen Anweisungen zum Wechseln ausführen können, und hier versuche ich, die sinnvollsten Optionen vorzuschlagen. Hier sind ein paar Orte zu überprüfen:

- 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

Verwenden Sie das, was die Sprache bietet: das if / else-Konstrukt.

Wenn Sie ein `switch` / `case` Konstrukt verwenden möchten, ist es am einfachsten, das gute alte `if` / `else` Konstrukt zu verwenden:

```
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!")
```

Es mag überflüssig und nicht immer hübsch aussehen, aber das ist bei weitem der effizienteste Weg, und es erledigt die Aufgabe:

```
>>> 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
```

Benutze ein Diktat von Funktionen

Ein weiterer unkomplizierter Weg ist das Erstellen eines Funktionswörterbuchs:

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

dann fügen Sie eine Standardfunktion hinzu:

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

und Sie verwenden die Get-Methode des Wörterbuchs, um die Funktion mit dem Wert zu überprüfen und auszuführen. Wenn im Wörterbuch kein Wert vorhanden ist, wird `default_case` ausgeführt.

```
>>> 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
```

Sie können auch etwas syntaktischen Zucker herstellen, damit der Schalter schöner aussieht:

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

>>> run_switch(1)
one
```

Verwenden Sie die Klassenprüfung

Sie können eine Klasse verwenden, um die Switch- / Case-Struktur nachzuahmen. Im Folgenden wird die Introspektion einer Klasse verwendet (mit der Funktion `getattr()` , die eine Zeichenfolge in eine gebundene Methode einer Instanz auflöst), um den "case" `getattr()` aufzulösen.

Dann wird diese Introspecting-Methode auf die `__call__` Methode gesetzt, um den Operator `()` zu überladen.

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

```
    __call__ = switch
```

Damit es noch schöner aussieht, subclassieren wir die `SwitchBase` Klasse (dies könnte jedoch in einer Klasse geschehen), und definieren den gesamten `case` als Methoden:

```
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!')
```

so können wir es endlich benutzen:

```
>>> 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!
```

Verwenden eines Kontextmanagers

Eine andere Art, die sehr lesbar und elegant ist, aber weit weniger effizient als eine if / else-Struktur ist, besteht darin, eine Klasse wie folgt zu erstellen, die den zu vergleichenden Wert liest und speichert, um sich im Kontext als aufrufbar darzustellen gibt true zurück, wenn er mit dem gespeicherten Wert übereinstimmt:

```
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
```

dann ist das Definieren der Fälle fast eine Übereinstimmung mit dem realen `switch` / `case` Konstrukt (innerhalb einer Funktion sichtbar gemacht, um die Darstellung zu erleichtern):

```
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!')
```

Die Ausführung wäre also:

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

Nota Bene :

- Diese Lösung wird als das auf pypi verfügbare *Switch*- Modul angeboten .

Alternativen zum Wechseln von Anweisungen aus anderen Sprachen online lesen: https://riptutorial.com/de/python/topic/4268/alternativen-zum-wechseln-von-anweisungen-aus-anderen-sprachen

Kapitel 8: ArcPy

Bemerkungen

In diesem Beispiel wird ein Suchcursor aus dem Data Access (da) -Modul von ArcPy verwendet.

Verwechseln Sie die arcpy.da.SearchCursor-Syntax nicht mit dem früheren und langsameren arcpy.SearchCursor ().

Das Datenzugriffsmodul (arcpy.da) ist erst seit ArcGIS 10.1 for Desktop verfügbar.

Examples

Der Wert eines Feldes für alle Zeilen der Feature-Class in der File-Geodatabase mit dem Suchcursor wird gedruckt

So drucken Sie ein Testfeld (TestField) aus einer Test-Feature-Class (TestFC) in einer Testdatei-Geodatabase (Test.gdb) in einem temporären Ordner (C: \ Temp):

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

createDissolvedGDB zum Erstellen einer Datei gdb im Arbeitsbereich

```
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
```

ArcPy online lesen: https://riptutorial.com/de/python/topic/4693/arcpy

Kapitel 9: Arrays

Einführung

"Arrays" in Python sind nicht die Arrays in herkömmlichen Programmiersprachen wie C und Java, sondern näher an Listen. Eine Liste kann eine Sammlung von homogenen oder heterogenen Elementen sein und kann Ints, Strings oder andere Listen enthalten.

Parameter

Parameter	Einzelheiten
`b`	Stellt eine vorzeichenbehaftete Ganzzahl der Größe 1 Byte dar
`B`	Repräsentiert eine vorzeichenlose Ganzzahl der Größe 1 Byte
`c`	Stellt Zeichen der Größe 1 Byte dar
`u`	Stellt ein Unicode-Zeichen der Größe 2 Byte dar
`h`	Stellt eine vorzeichenbehaftete Ganzzahl mit einer Größe von 2 Byte dar
`H`	Stellt eine vorzeichenlose Ganzzahl mit einer Größe von 2 Byte dar
`i`	Stellt eine vorzeichenbehaftete Ganzzahl mit einer Größe von 2 Byte dar
`I`	Stellt eine vorzeichenlose Ganzzahl mit einer Größe von 2 Byte dar
`w`	Stellt ein Unicode-Zeichen der Größe 4 Byte dar
`l`	Stellt eine vorzeichenbehaftete Ganzzahl mit einer Größe von 4 Byte dar
`L`	Repräsentiert eine vorzeichenlose Ganzzahl der Größe 4 Byte
`f`	Stellt einen Gleitkommawert mit einer Größe von 4 Byte dar
`d`	Stellt einen Gleitkommawert mit einer Größe von 8 Byte dar

Examples

Grundlegende Einführung in Arrays

Ein Array ist eine Datenstruktur, in der Werte desselben Datentyps gespeichert werden. In Python ist dies der Hauptunterschied zwischen Arrays und Listen.

Während Python-Listen Werte enthalten können, die verschiedenen Datentypen entsprechen,

können Arrays in Python nur Werte enthalten, die demselben Datentyp entsprechen. In diesem Tutorial werden wir die Python-Arrays anhand einiger Beispiele verstehen.

Wenn Sie mit Python noch nicht vertraut sind, beginnen Sie mit dem Python-Einführungsartikel.

Um Arrays in der Python-Sprache verwenden zu können, müssen Sie das Standard- `array` Modul importieren. Das liegt daran, dass Array kein grundlegender Datentyp ist, wie Strings, Integer usw. So können Sie `array` Module in Python importieren:

```
from array import *
```

Nachdem Sie das `array` Modul importiert haben, können Sie ein Array deklarieren. So machen Sie es:

```
arrayIdentifierName = array(typecode, [Initializers])
```

In der obigen `arrayIdentifierName` ist `arrayIdentifierName` der Name des Arrays. Mit `typecode` Python den Typ des Arrays kennen. `Initializers` sind die Werte, mit denen das Array initialisiert wird.

Typcodes sind die Codes, mit denen der Typ der Arraywerte oder der Typ des Arrays definiert wird. Die Tabelle im Parameterabschnitt zeigt die möglichen Werte, die Sie beim Deklarieren eines Arrays und seines Typs verwenden können.

Hier ist ein reales Beispiel für die Deklaration von Python-Arrays:

```
my_array = array('i',[1,2,3,4])
```

In dem obigen Beispiel ist der verwendete Typcode `i` . Dieser Typcode repräsentiert eine vorzeichenbehaftete Ganzzahl mit einer Größe von 2 Byte.

Hier ist ein einfaches Beispiel eines Arrays mit 5 Ganzzahlen

```
from array import *
my_array = array('i', [1,2,3,4,5])
for i in my_array:
    print(i)
# 1
# 2
# 3
# 4
# 5
```

Zugriff auf einzelne Elemente über Indizes

Auf einzelne Elemente kann über Indizes zugegriffen werden. Python-Arrays sind nullindiziert. Hier ist ein Beispiel :

```
my_array = array('i', [1,2,3,4,5])
```

```
print(my_array[1])
# 2
print(my_array[2])
# 3
print(my_array[0])
# 1
```

Hängen Sie mithilfe der append () -Methode einen beliebigen Wert an das Array an

```
my_array = array('i', [1,2,3,4,5])
my_array.append(6)
# array('i', [1, 2, 3, 4, 5, 6])
```

Beachten Sie, dass der Wert `6` an die vorhandenen Array-Werte angehängt wurde.

Fügen Sie mit der Methode insert () einen Wert in ein Array ein

Wir können die Methode `insert()` verwenden, um einen Wert an einem beliebigen Index des Arrays einzufügen. Hier ist ein Beispiel :

```
my_array = array('i', [1,2,3,4,5])
my_array.insert(0,0)
#array('i', [0, 1, 2, 3, 4, 5])
```

Im obigen Beispiel wurde der Wert 0 an Index 0 eingefügt. Beachten Sie, dass das erste Argument der Index ist, während das zweite Argument der Wert ist.

Erweitern Sie das Python-Array mit der extend () -Methode

Ein Python-Array kann mit der `extend()` -Methode um mehrere Werte `extend()` werden. Hier ist ein Beispiel :

```
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])
```

Wir sehen, dass das Array my_array mit Werten von `my_extnd_array` .

Fügen Sie mithilfe der fromlist () -Methode Elemente aus der Liste in das Array ein

Hier ist ein Beispiel:

```
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])
```

Wir sehen also, dass die Werte 11,12 und 13 von Liste `c` zu `my_array` .

Entfernen Sie ein Arrayelement mit der remove () - Methode

Hier ist ein Beispiel :

```
my_array = array('i', [1,2,3,4,5])
my_array.remove(4)
# array('i', [1, 2, 3, 5])
```

Wir sehen, dass das Element 4 aus dem Array entfernt wurde.

Entferne das letzte Array-Element mit der pop () -Methode

`pop` entfernt das letzte Element aus dem Array. Hier ist ein Beispiel :

```
my_array = array('i', [1,2,3,4,5])
my_array.pop()
# array('i', [1, 2, 3, 4])
```

Wir sehen also, dass das letzte Element (`5`) aus dem Array herausgeplatzt wurde.

Rufen Sie mit Hilfe der index () -Methode jedes Element über seinen Index ab

`index()` gibt den ersten Index des übereinstimmenden Werts zurück. Denken Sie daran, dass Arrays nullindiziert sind.

```
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
```

Beachten Sie in diesem zweiten Beispiel, dass nur ein Index zurückgegeben wurde, obwohl der Wert zweimal im Array vorhanden ist

Umkehren eines Python-Arrays mithilfe der reverse () -Methode

Die `reverse()` -Methode macht das, was der Name sagt - das Array wird umgekehrt. Hier ist ein Beispiel :

```
my_array = array('i', [1,2,3,4,5])
my_array.reverse()
# array('i', [5, 4, 3, 2, 1])
```

Abrufen von Array-Pufferinformationen über die buffer_info () -Methode

Diese Methode stellt die Startadresse des Array-Puffers im Speicher und die Anzahl der Elemente

im Array bereit. Hier ist ein Beispiel:

```
my_array = array('i', [1,2,3,4,5])
my_array.buffer_info()
(33881712, 5)
```

Überprüfen Sie die Anzahl der Vorkommen eines Elements mithilfe der Methode count ()

`count()` gibt die Anzahl zurück, zu der das Element in einem Array angezeigt wird. Im folgenden Beispiel sehen wir, dass der Wert `3` zweimal vorkommt.

```
my_array = array('i', [1,2,3,3,5])
my_array.count(3)
# 2
```

Konvertieren Sie das Array mithilfe der tostring () -Methode in einen String

`tostring()` konvertiert das Array in einen String.

```
my_char_array = array('c', ['g','e','e','k'])
# array('c', 'geek')
print(my_char_array.tostring())
# geek
```

Konvertieren Sie das Array mithilfe der tolist () -Methode in eine Python-Liste mit denselben Elementen

Wenn Sie eine Python benötigen `list` Objekt, können Sie das verwenden `tolist()` Methode , um Ihr Array in eine Liste zu konvertieren.

```
my_array = array('i', [1,2,3,4,5])
c = my_array.tolist()
# [1, 2, 3, 4, 5]
```

Hängen Sie einen String mit der fromstring () -Methode an das Char-Array an

Sie können eine Zeichenfolge mit `fromstring()` an ein Zeichenarray anhängen.

```
my_char_array = array('c', ['g','e','e','k'])
my_char_array.fromstring("stuff")
print(my_char_array)
#array('c', 'geekstuff')
```

Arrays online lesen: https://riptutorial.com/de/python/topic/4866/arrays

Kapitel 10: Asyncio-Modul

Examples

Coroutine und Delegation Syntax

Vor der Veröffentlichung von Python 3.5+ verwendete das `asyncio` Modul Generatoren, um asynchrone Aufrufe nachzuahmen, und verfügte daher über eine andere Syntax als das aktuelle Python 3.5-Release.

Python 3.x 3.5

In Python 3.5 wurden die `async` und `await` Schlüsselwörter eingeführt. Beachten Sie das Fehlen von Klammern um den Aufruf von `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

Vor Python 3.5 wurde der Dekorateur `@asyncio.coroutine` verwendet, um eine Coroutine zu definieren. Die Ausbeute aus Ausdruck wurde für die Generatordelegierung verwendet. Beachten Sie die Klammern um den `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

Hier ein Beispiel, das zeigt, wie zwei Funktionen asynchron ausgeführt werden können:

```
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)
```

Asynchrone Executoren

Hinweis: Verwendet die Python 3.5+ async / await-Syntax

`asyncio` unterstützt die Verwendung von in `concurrent.futures` gefundenen `Executor` Objekten, um Aufgaben asynchron zu planen. Ereignisschleifen haben die Funktion `run_in_executor()` die ein `Executor` Objekt, ein `Callable` und die Callable-Parameter übernimmt.

Planen einer Aufgabe für einen `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))
```

Jede Ereignisschleife verfügt außerdem über einen "Standard" `Executor` Slot, der einem `Executor` zugewiesen werden kann. Um einen `Executor` zuzuweisen und Aufgaben aus der Schleife zu planen, verwenden Sie die Methode `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))
```

Es gibt zwei Haupttypen von `Executor` in `concurrent.futures` , den `ThreadPoolExecutor` und den `ProcessPoolExecutor` . Der `ThreadPoolExecutor` enthält einen Pool von Threads , die entweder manuell auf eine bestimmte Anzahl von Fäden durch den Konstruktor oder Standardwerte für die Anzahl der Kerne auf dem Maschinenzeit 5. eingestellt werden `ThreadPoolExecutor` verwendet den Pool von Threads Aufgaben es auszuführen zugeordnet ist und Im Allgemeinen besser bei CPU-gebundenen Operationen als bei E / A-gebundenen Operationen. `ProcessPoolExecutor` dies mit dem `ProcessPoolExecutor` der für jede ihm zugewiesene Aufgabe einen neuen Prozess erzeugt. Der `ProcessPoolExecutor` kann nur Aufgaben und Parameter übernehmen, die kommissionierbar sind. Die häufigsten nicht auswählbaren Aufgaben sind die Methoden von Objekten. Wenn Sie die Methode eines Objekts als Task in einem `Executor` `ThreadPoolExecutor` müssen, müssen Sie einen `ThreadPoolExecutor` .

UVLoop verwenden

`uvloop` ist eine Implementierung für `asyncio.AbstractEventLoop` basierend auf libuv (Wird von nodejs verwendet). Es ist mit 99% der `asyncio` Funktionen `asyncio` und ist viel schneller als das herkömmliche `asyncio.EventLoop` . `uvloop` ist derzeit unter Windows nicht verfügbar. Installieren Sie es mit `pip install uvloop` .

```
import asyncio
import uvloop

if __name__ == "__main__":
    asyncio.set_event_loop(uvloop.new_event_loop())
    # Do your stuff here ...
```

Sie können die Ereignisschleifen-Factory auch ändern, indem Sie die `EventLoopPolicy` auf die in `uvloop` .

```
import asyncio
import uvloop

if __name__ == "__main__":
    asyncio.set_event_loop_policy(uvloop.EventLoopPolicy())
    loop = asyncio.new_event_loop()
```

Synchronisationsprimitiv: Ereignis

Konzept

Verwenden Sie ein `Event` , **um die Planung mehrerer Coroutinen** zu **synchronisieren** .

Vereinfacht gesagt, ist ein Ereignis wie eine Waffe, die auf ein Rennen abgefeuert wird: Es lässt die Läufer die Startblöcke verlassen.

Beispiel

```
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)
```

Ausgabe:

Verbraucher B wartet
Verbraucher Ein Warten
EVENT SET
Verbraucher B ausgelöst
Consumer A ausgelöst

Ein einfacher Websocket

Hier `asyncio` wir mit `asyncio` einen einfachen Echo- `asyncio` . Wir definieren Coroutinen für die Verbindung zu einem Server und zum Senden / Empfangen von Nachrichten. Die Communcations des websocket werden in einem laufen `main` Koroutine, die durch eine Ereignisschleife ausgeführt wird. Dieses Beispiel wurde aus einem früheren Beitrag geändert.

```
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())
```

Häufiges Missverständnis über Asyncio

Das häufigste Missverständnis über `asnycio` besteht `asnycio` darin, dass Sie alle Aufgaben parallel ausführen können, indem Sie die GIL (globale Interpretersperre) umgehen und parallel dazu blockierende Jobs (in separaten Threads) ausführen. es tut **nicht**

`asyncio` (und Bibliotheken, die mit `asyncio`) bauen auf Coroutines auf: Funktionen, die (gemeinsam) den Steuerungsfluss an die aufrufende Funktion zurückgeben. Beachten Sie `asyncio.sleep` in den obigen Beispielen. Dies ist ein Beispiel für eine nicht blockierende Coroutine, die 'im Hintergrund' wartet und den Steuerungsfluss an die aufrufende Funktion zurückgibt (wenn mit `await` aufgerufen). `time.sleep` ist ein Beispiel für eine `time.sleep` . Der Ablauf des Programms wird dort einfach `time.sleep` und erst nach `time.sleep` .

Ein reales Beispiel ist die `requests` die (vorerst) nur aus Sperrfunktionen besteht. Es gibt keine Parallelität, wenn Sie eine seiner Funktionen innerhalb von `asyncio` . `aiohttp` dagegen wurde für `asyncio` . Seine Coroutinen werden gleichzeitig laufen.

- Wenn Sie langlaufende CPU-gebundene Aufgaben haben, die Sie parallel `asyncio` ist `asyncio` **nichts** für Sie. dazu benötigen Sie `threads` oder `multiprocessing` .
- wenn Sie IO-gebundene Arbeitsplätze ausgeführt wird , *können* Sie sie gleichzeitig mit

laufen `asyncio` .

Asyncio-Modul online lesen: https://riptutorial.com/de/python/topic/1319/asyncio-modul

Kapitel 11: Attribut-Zugriff

Syntax

- `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

Grundlegender Attributzugriff mit der Punktnotation

Nehmen wir eine Probestunde.

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

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

In Python können Sie das Attribut *Titel* der Klasse Zugriff auf die Punktnotation verwendet.

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

Wenn ein Attribut nicht vorhanden ist, gibt Python einen Fehler aus:

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

Setter, Getter & Eigenschaften

Für die Datenkapselung möchten Sie manchmal ein Attribut haben, dessen Wert von anderen Attributen stammt oder generell, welcher Wert momentan berechnet werden soll. Die Standardmethode, um mit dieser Situation umzugehen, ist das Erstellen einer Methode, Getter oder Setter genannt.

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

Im obigen Beispiel ist es leicht zu sehen, was passiert, wenn wir ein neues Buch erstellen, das einen Titel und einen Autor enthält. Wenn alle Bücher, die wir unserer Bibliothek hinzufügen, über

Autoren und Titel verfügen, können wir die Getter und Setter überspringen und die Punktnotation verwenden. Angenommen, wir haben einige Bücher, die keinen Autor haben, und wir möchten den Autor auf "Unbekannt" setzen. Oder wenn sie mehrere Autoren haben und wir planen, eine Liste der Autoren zurückzugeben.

In diesem Fall können wir einen Getter und einen Setter für das *Author*- Attribut erstellen.

```
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
```

Dieses Schema wird nicht empfohlen.

Ein Grund ist, dass es einen Haken gibt: Nehmen wir an, wir haben unsere Klasse mit dem Attribut public und ohne Methoden entworfen. Die Leute haben es schon oft benutzt und Code wie folgt geschrieben:

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

Jetzt haben wir ein Problem. Weil *Autor* kein Attribut ist! Python bietet eine Lösung für dieses Problem, die als Eigenschaften bezeichnet wird. Eine Methode zum Abrufen von Eigenschaften wird mit der @ -Eigenschaft vor dem Header versehen. Die Methode, die wir als Setter verwenden möchten, ist mit @ attributeName.setter davor versehen.

Deshalb haben wir jetzt unsere neue aktualisierte Klasse.

```
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
```

Beachten Sie, dass in Python normalerweise nicht mehrere Methoden mit demselben Namen und

einer unterschiedlichen Anzahl von Parametern vorhanden sind. In diesem Fall erlaubt Python dies jedoch aufgrund der verwendeten Dekorateure.

Wenn wir den Code testen:

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

Attribut-Zugriff online lesen: https://riptutorial.com/de/python/topic/4392/attribut-zugriff

Kapitel 12: Audio

Examples

Audio mit Pyglet

```
import pyglet
audio = pyglet.media.load("audio.wav")
audio.play()
```

Weitere Informationen finden Sie unter Pyglet

Mit WAV-Dateien arbeiten

gewinnt

- Windows-Umgebung

```
import winsound
winsound.PlaySound("path_to_wav_file.wav", winsound.SND_FILENAME)
```

Welle

- Unterstützt Mono / Stereo
- Unterstützt keine Komprimierung / Dekomprimierung

```
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)
```

Konvertieren Sie eine beliebige Sounddatei mit Python und 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()
```

Hinweis:

- http://superuser.com/questions/507386/why-would-i-choose-libav-over-ffmpeg-or-is-there-ever-unterschied
- Was sind die Unterschiede und Ähnlichkeiten zwischen ffmpeg, libav und avconv?

Die Windows-Pieptöne werden abgespielt

Windows bietet eine explizite Schnittstelle, über die Sie mit dem Modul `winsound` rohe Pieptöne bei einer bestimmten Frequenz und Dauer `winsound` können.

```
import winsound
freq = 2500 # Set frequency To 2500 Hertz
dur = 1000 # Set duration To 1000 ms == 1 second
winsound.Beep(freq, dur)
```

Audio online lesen: https://riptutorial.com/de/python/topic/8189/audio

Kapitel 13: Ausnahmen

Einführung

Fehler, die während der Ausführung erkannt werden, werden als Ausnahmen bezeichnet und sind nicht unbedingt tödlich. Die meisten Ausnahmen werden nicht von Programmen behandelt. Es ist möglich, Programme zu schreiben, die ausgewählte Ausnahmen behandeln. Es gibt bestimmte Funktionen in Python, um Ausnahmen und Ausnahmelogik zu behandeln. Außerdem haben Ausnahmen eine reichhaltige `BaseException` , die alle vom `BaseException` Typ erben.

Syntax

- *Ausnahme* auslösen
- # erhöhen eine Ausnahme, die bereits ausgelöst wurde
- *Ausnahme* von *Ursache auslösen* # Python 3 - Ausnahmeursache festlegen
- *Ausnahme* von None auslösen # Python 3 - unterdrückt den gesamten Ausnahmekontext
- Versuchen:
- außer *[Ausnahmetypen] [* als *Bezeichner]* :
- sonst:
- endlich:

Examples

Ausnahmen aufwerfen

Wenn Ihr Code auf eine Bedingung stößt, mit der er nicht umgehen kann, beispielsweise ein falscher Parameter, sollte die entsprechende Ausnahme ausgelöst werden.

```
def even_the_odds(odds):
    if odds % 2 != 1:
        raise ValueError("Did not get an odd number")

    return odds + 1
```

Ausnahmen fangen

Verwenden Sie `try...except:` um Ausnahmen abzufangen. Sie sollten so genau wie möglich eine Ausnahme angeben:

```
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.
```

Die angegebene Ausnahmeklasse - in diesem Fall `ZeroDivisionError` - `ZeroDivisionError` alle Ausnahmen, die dieser Klasse oder einer Unterklasse dieser Ausnahmebedingung angehören.

`ZeroDivisionError` ist beispielsweise eine Unterklasse von `ArithmeticError` :

```
>>> ZeroDivisionError.__bases__
(<class 'ArithmeticError'>,)
```

Das Folgende wird also immer noch den `ZeroDivisionError` fangen:

```
try:
    5 / 0
except ArithmeticError:
    print("Got arithmetic error")
```

Ausführen von Bereinigungscode mit

Manchmal möchten Sie möglicherweise, dass unabhängig von der aufgetretenen Ausnahme etwas geschieht, beispielsweise wenn Sie einige Ressourcen bereinigen müssen.

Der `finally` Block einer `try` Klausel wird unabhängig davon ausgeführt, ob Ausnahmen ausgelöst wurden.

```
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)
```

Dieses Muster wird mit Kontextmanagern (mit der Anweisung `with`) oft besser behandelt.

Ausnahmen erneut erhöhen

Manchmal möchten Sie eine Ausnahme erfassen, um sie zu prüfen, z. B. zu Protokollierungszwecken. Nach der Inspektion möchten Sie, dass die Ausnahme wie zuvor weitergegeben wird.

Verwenden Sie in diesem Fall einfach die `raise` Anweisung ohne Parameter.

```
try:
    5 / 0
except ZeroDivisionError:
    print("Got an error")
    raise
```

Beachten Sie jedoch, dass jemand weiter oben im Aufrufer-Stack die Ausnahme noch fangen und irgendwie damit umgehen kann. Die fertige Ausgabe kann in diesem Fall ein Ärgernis sein, da dies auf jeden Fall passieren wird (gefangen oder nicht gefangen). Daher empfiehlt es sich, eine andere Ausnahme auszulösen, die Ihren Kommentar zur Situation sowie die ursprüngliche Ausnahme enthält:

```
try:
    5 / 0
except ZeroDivisionError as e:
    raise ZeroDivisionError("Got an error", e)
```

Dies hat jedoch den Nachteil, dass die Ausnahmespur auf genau diese `raise` reduziert wird `raise` während die `raise` ohne Argument die ursprüngliche Ausnahmespur beibehält.

In Python 3 können Sie den ursprünglichen Stapel beibehalten, indem Sie die `raise` - `from` - Syntax verwenden:

```
    raise ZeroDivisionError("Got an error") from e
```

Kettenausnahmen mit Raise from

Bei der Behandlung einer Ausnahme können Sie eine weitere Ausnahme auslösen. Wenn Sie `IOError` beim Lesen aus einer Datei einen `IOError` , möchten Sie möglicherweise einen anwendungsspezifischen Fehler `IOError` , der stattdessen den Benutzern Ihrer Bibliothek angezeigt wird.

Python 3.x 3.0

Sie können Ausnahmen verketten, um zu zeigen, wie die Behandlung von Ausnahmen abläuft:

```
>>> 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
```

Ausnahmehierarchie

Die Ausnahmebehandlung erfolgt basierend auf einer Ausnahmehierarchie, die durch die Vererbungsstruktur der Ausnahmeklassen bestimmt wird.

Beispielsweise sind `IOError` und `OSError` beide Unterklassen von `EnvironmentError` . Code, der

einen `IOError` , fängt keinen `OSError` . Code, der einen `EnvironmentError` `IOError` , fängt jedoch sowohl `IOError` als auch `OSError` s ab.

Die Hierarchie der integrierten Ausnahmen:

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
```

Ausnahmen sind auch Objekte

Ausnahmen sind nur reguläre Python-Objekte, die von der integrierten `BaseException` erben. Ein Python-Skript kann die Anweisung `raise` verwenden, `raise` Ausführung zu unterbrechen, sodass Python an diesem Punkt einen Stack-Trace des Aufrufstapels und eine Darstellung der Ausnahmeinstanz ausgibt. Zum Beispiel:

```
>>> 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!
```

was besagt, dass ein `ValueError` mit der Meldung `'Example error!'` wurde durch unsere `failing_function()` , die im Interpreter ausgeführt wurde.

Anrufcode kann beliebige Ausnahmen aller Art behandeln, die ein Anruf auslösen kann:

```
>>> try:
...     failing_function()
... except ValueError:
...     print('Handled the error')
Handled the error
```

Sie können halten , die Ausnahme - Objekte erhalten , indem sie in der Zuordnung `except...` Teil des Ausnahmebehandlung Code:

```
>>> try:
...     failing_function()
... except ValueError as e:
...     print('Caught exception', repr(e))
Caught exception ValueError('Example error!',)
```

Eine vollständige Liste der integrierten Python-Ausnahmen mit ihren Beschreibungen finden Sie in der Python-Dokumentation: https://docs.python.org/3.5/library/exceptions.html . Und hier ist die vollständige Liste hierarchisch angeordnet: Ausnahmehierarchie .

Benutzerdefinierte Ausnahmetypen erstellen

Erstellen Sie eine Klasse, die von `Exception` erbt:

```
class FooException(Exception):
    pass
try:
    raise FooException("insert description here")
except FooException:
```

```
    print("A FooException was raised.")
```

oder ein anderer Ausnahmetyp:

```
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))
```

Fangen Sie nicht alles!

Während es oft versucht ist, jede `Exception` zu fangen:

```
try:
    very_difficult_function()
except Exception:
    # log / try to reconnect / exit gratiously
finally:
    print "The END"
    # it runs no matter what execute.
```

Oder sogar alles (dazu gehören `BaseException` und alle `BaseException` einschließlich `Exception`):

```
try:
    even_more_difficult_function()
except:
    pass  # do whatever needed
```

In den meisten Fällen ist es eine schlechte Praxis. Es könnte mehr als beabsichtigt sein, z. B. `SystemExit` , `KeyboardInterrupt` und `MemoryError` Jeder sollte im Allgemeinen anders behandelt werden als `MemoryError` System- oder Logikfehler. Dies bedeutet auch, dass es kein klares Verständnis dafür gibt, was der interne Code falsch machen kann und wie er sich von diesem Zustand richtig erholen kann. Wenn Sie jeden Fehler feststellen, wissen Sie nicht, welcher Fehler aufgetreten ist oder wie er zu beheben ist.

Dies wird häufiger als "Bug Masking" bezeichnet und sollte vermieden werden. Lassen Sie Ihr Programm abstürzen, anstatt im Hintergrund zu versagen oder gar zu verschlimmern, und zwar auf einer tieferen Ausführungsebene. (Stellen Sie sich vor, es ist ein Transaktionssystem)

Normalerweise werden diese Konstrukte auf der äußersten Ebene des Programms verwendet und protokollieren die Details des Fehlers, sodass der Fehler behoben werden kann oder der Fehler genauer behandelt werden kann.

Mehrere Ausnahmen abfangen

Es gibt mehrere Möglichkeiten, mehrere Ausnahmen abzufangen .

Die erste besteht darin, ein Tupel der Ausnahmetypen zu erstellen, die auf dieselbe Weise abgefangen und behandelt werden sollen. In diesem Beispiel ignoriert der Code die Ausnahmen `KeyError` und `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.")
```

Wenn Sie unterschiedliche Ausnahmen auf unterschiedliche Weise behandeln möchten, können Sie für jeden Typ einen eigenen Ausnahmeblock bereitstellen. In diesem Beispiel fangen wir immer noch `KeyError` und `AttributeError` , behandeln die Ausnahmen jedoch auf unterschiedliche Weise.

```
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)
```

Praktische Beispiele für die Behandlung von Ausnahmen

Benutzereingabe

Stellen Sie sich vor, Sie möchten, dass ein Benutzer über die Eingabe eine Nummer `input` . Sie möchten sicherstellen, dass die Eingabe eine Zahl ist. Sie können `try` / mit `except` davon verwenden:

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.')
```

Hinweis: Python 2.x verwendet stattdessen `raw_input` . die Funktion `input` existiert in Python 2.x hat aber eine andere Semantik. Im obigen Beispiel würde die `input` auch Ausdrücke wie `2 + 2` akzeptieren, die eine Zahl ergeben.

Wenn die Eingabe nicht in eine Ganzzahl konvertiert werden konnte, wird ein `ValueError` . Sie können es mit `except` fangen. Wenn keine Ausnahme ausgelöst wird `break` springt `break` aus der Schleife. Nach der Schleife enthält `nb` eine ganze Zahl.

Wörterbücher

Stellen Sie sich über eine Liste von aufeinanderfolgenden ganzen Zahlen iterieren, wie `range(n)` , und Sie haben eine Liste der Wörterbücher `d` , die Informationen über die Dinge enthält zu tun , wenn Sie einige bestimmte ganze Zahlen auftreten, sagen *die überspringen* `d[i]` *Nächsten.*

```
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
```

Ein `KeyError` wird `KeyError` , wenn Sie versuchen, einen Wert aus einem Wörterbuch für einen nicht vorhandenen Schlüssel `KeyError` .

Sonst

Code in einem else-Block wird nur ausgeführt, wenn vom Code im `try` Block keine Ausnahmen ausgelöst wurden. Dies ist nützlich, wenn Sie Code haben, den Sie nicht ausführen möchten, wenn eine Ausnahme ausgelöst wird. Sie möchten jedoch nicht, dass durch diesen Code ausgelöste Ausnahmen abgefangen werden.

Zum Beispiel:

```
try:
    data = {1: 'one', 2: 'two'}
    print(data[1])
except KeyError as e:
    print('key not found')
else:
    raise ValueError()
# Output: one
# Output: ValueError
```

Beachten Sie, dass diese Art von `else:` nicht mit einer `if` -Klausel zu einem `elif` . Wenn Sie Folgendes haben, `if` es unter dem `else:` eingerückt bleiben muss:

```
try:
    ...
except ...:
    ...
else:
    if ...:
```

```
    ...
elif ...:
    ...
else:
    ...
```

Ausnahmen online lesen: https://riptutorial.com/de/python/topic/1788/ausnahmen

Kapitel 14: Bedienmodul

Examples

Operatoren als Alternative zu einem Infix-Operator

Für jeden Infix-Operator, zB `+` gibt es eine `operator` (`operator.add` für `+`):

```
1 + 1
# Output: 2
from operator import add
add(1, 1)
# Output: 2
```

Obwohl in der Hauptdokumentation angegeben ist, dass für die arithmetischen Operatoren nur numerische Eingaben zulässig sind, *ist dies* möglich:

```
from operator import mul
mul('a', 10)
# Output: 'aaaaaaaaaa'
mul([3], 3)
# Output: [3, 3, 3]
```

Siehe auch: Mapping von Operation zu Operator-Funktion in der offiziellen Python-Dokumentation .

Methodenaufruf

Anstelle dieser `lambda` , die die Methode explizit aufruft:

```
alist = ['wolf', 'sheep', 'duck']
list(filter(lambda x: x.startswith('d'), alist))     # Keep only elements that start with 'd'
# Output: ['duck']
```

Man könnte eine Operator-Funktion verwenden, die dasselbe tut:

```
from operator import methodcaller
list(filter(methodcaller('startswith', 'd'), alist)) # Does the same but is faster.
# Output: ['duck']
```

Itemgetter

Gruppieren der Schlüssel-Wert-Paare eines Wörterbuchs nach dem Wert mit `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}}
```

was einer `lambda` Funktion wie folgt entspricht (aber schneller ist):

```
dict((i, dict(v)) for i, v in groupby(adict.items(), lambda x: x[1]))
```

Oder eine Liste von Tupeln nach dem zweiten Element sortieren, zuerst das erste Element als sekundär:

```
alist_of_tuples = [(5,2), (1,3), (2,2)]
sorted(alist_of_tuples, key=itemgetter(1,0))
# Output: [(2, 2), (5, 2), (1, 3)]
```

Bedienmodul online lesen: https://riptutorial.com/de/python/topic/257/bedienmodul

Kapitel 15: Befehlszeilenargumente analysieren

Einführung

Die meisten Befehlszeilentools sind auf Argumente angewiesen, die bei der Ausführung an das Programm übergeben werden. Anstatt zur Eingabe aufzufordern, erwarten diese Programme, dass Daten oder bestimmte Flags (die zu Booleans werden) gesetzt werden. Dies ermöglicht es dem Benutzer und anderen Programmen, die Python-Datei auszuführen, wobei die Daten beim Start übergeben werden. In diesem Abschnitt wird die Implementierung und Verwendung von Befehlszeilenargumenten in Python erläutert und veranschaulicht.

Examples

Hallo Welt in Schwierigkeiten

Das folgende Programm sagt dem Benutzer Hallo. Es erfordert ein Positionsargument, den Namen des Benutzers, und die Begrüßung kann auch mitgeteilt werden.

```
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!
```

Für weitere Details lesen Sie bitte die argparse-Dokumentation .

Grundlegendes Beispiel mit docopt

docopt konvertiert Befehlszeilenargumente auf den Kopf. Statt die Argumente zu analysieren, **schreiben** Sie einfach **die Verwendungszeichenfolge** für Ihr Programm, und docopt **analysiert die Verwendungszeichenfolge** und verwendet sie zum Extrahieren der Befehlszeilenargumente.

```
"""
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)
```

Probeläufe:

```
$ 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'}
```

Setzen Sie sich gegenseitig ausschließende Argumente mit argparse

Wenn zwei oder mehr Argumente sich gegenseitig ausschließen sollen. Sie können die Funktion `argparse.ArgumentParser.add_mutually_exclusive_group()` . Im folgenden Beispiel können entweder foo oder bar vorhanden sein, jedoch nicht beide gleichzeitig.

```
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
```

Wenn Sie versuchen, das Skript `--foo --bar` Argumente `--foo` und `--bar` , beschwert sich das Skript mit der folgenden Meldung.

```
error: argument -b/--bar: not allowed with argument -f/--foo
```

Befehlszeilenargumente mit argv verwenden

Wenn ein Python-Skript von der Befehlszeile aus aufgerufen wird, kann der Benutzer zusätzliche **Befehlszeilenargumente angeben,** die an das Skript übergeben werden. Diese Argumente werden an den Programmierer von der Systemvariablen zur Verfügung `sys.argv` („argv“ ist ein traditioneller Name in den meisten Programmiersprachen verwendet, und es bedeutet **„arg** UMENT **v** ector“).

`sys.argv` ist das erste Element in der Liste `sys.argv` der Name des Python-Skripts selbst, während die übrigen Elemente die Token sind, die der Benutzer beim Aufrufen des Skripts `sys.argv` .

```
# 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']
```

Hier ist ein weiteres Beispiel für die Verwendung von `argv` . Wir entfernen zunächst das ursprüngliche Element von sys.argv, da es den Namen des Skripts enthält. Dann kombinieren wir den Rest der Argumente in einem einzigen Satz und drucken diesen Satz vor dem Namen des aktuell angemeldeten Benutzers (damit ein Chat-Programm emuliert wird).

```
import getpass
import sys

words = sys.argv[1:]
sentence = " ".join(words)
print("[%s] %s" % (getpass.getuser(), sentence))
```

Der Algorithmus, der häufig verwendet wird, wenn eine Anzahl nicht-positioneller Argumente "manuell" analysiert wird, besteht darin, die Liste `sys.argv` . Eine Möglichkeit ist, die Liste durchzugehen und jedes Element davon zu platzieren:

```
# 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()
```

Benutzerdefinierte Parser-Fehlernachricht mit Argumenten

Sie können Parser-Fehlermeldungen entsprechend Ihren Skriptanforderungen erstellen. Dies geschieht über die Funktion `argparse.ArgumentParser.error` . Das folgende Beispiel zeigt, dass das Skript eine Verwendung und eine `--foo` an `stderr` `--foo` wenn `--foo` wird, aber nicht `--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
```

Angenommen, Ihr Skriptname lautet sample.py, und wir führen `python sample.py --foo ds_in_fridge` : `python sample.py --foo ds_in_fridge`

Das Skript wird sich mit folgendem beschweren:

```
usage: sample.py [-h] [-f FOO] [-b BAR]
sample.py: error: --foo requires --bar. You did not specify bar.
```

Konzeptionelle Gruppierung von Argumenten mit argparse.add_argument_group ()

Wenn Sie argparse ArgumentParser () erstellen und Ihr Programm mit '-h' ausführen, erhalten Sie eine automatisierte Verwendungsnachricht, in der erläutert wird, mit welchen Argumenten Sie Ihre Software ausführen können. Positionsargumente und bedingte Argumente sind standardmäßig in zwei Kategorien unterteilt. Beispiel: Ein kleines Skript (example.py) und die Ausgabe, wenn Sie `python example.py -h` ausführen.

```
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
```

Es gibt Situationen, in denen Sie Ihre Argumente in weitere konzeptionelle Abschnitte unterteilen möchten, um Ihren Benutzer zu unterstützen. Beispielsweise möchten Sie möglicherweise alle Eingabeoptionen in einer Gruppe und alle Optionen zur Ausgabeformatierung in einer anderen Gruppe haben. Das obige Beispiel kann angepasst werden, um die `--foo_*` args von den `--bar_*` args zu trennen.

```
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()
```

Was erzeugt diese Ausgabe, wenn `python example.py -h` wird:

```
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
```

Erweitertes Beispiel mit docopt und docopt_dispatch

Wie bei docopt bilden Sie mit [docopt_dispatch] Ihre `--help` in die Variable `__doc__` Ihres Einstiegspunktmoduls. Dort rufen Sie `dispatch` mit dem doc-String als Argument auf, damit der Parser darüber laufen kann.

Anstatt die Argumente manuell zu handhaben (was normalerweise in einer if / else-Struktur mit hoher Zyklizität endet), überlassen Sie es dem Versand, indem Sie nur angeben, wie Sie die Menge der Argumente behandeln wollen.

Dafür gibt es den `dispatch.on` Dekorator: Sie geben ihm das Argument oder die Folge von Argumenten, die die Funktion auslösen sollen, und diese Funktion wird mit den übereinstimmenden Werten als Parameter ausgeführt.

```
"""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__)
```

Befehlszeilenargumente analysieren online lesen:
https://riptutorial.com/de/python/topic/1382/befehlszeilenargumente-analysieren

Kapitel 16: Benutzerdefinierte Fehler / Ausnahmen auslösen

Einführung

In Python gibt es viele Ausnahmen, die das Programm dazu zwingen, einen Fehler auszugeben, wenn ein Fehler auftritt.

Manchmal müssen Sie jedoch benutzerdefinierte Ausnahmen erstellen, die Ihrem Zweck dienen.

In Python können Benutzer solche Ausnahmen definieren, indem sie eine neue Klasse erstellen. Diese Ausnahmeklasse muss direkt oder indirekt von der Ausnahmeklasse abgeleitet werden. Die meisten der integrierten Ausnahmen sind auch von dieser Klasse abgeleitet.

Examples

Benutzerdefinierte Ausnahme

Hier haben wir eine benutzerdefinierte Ausnahme namens CustomError erstellt, die von der Exception-Klasse abgeleitet wird. Diese neue Ausnahme kann wie andere Ausnahmen ausgelöst werden, wenn die Erhöhungsanweisung mit einer optionalen Fehlermeldung verwendet wird.

```
class CustomError(Exception):
       pass

x = 1

if x == 1:
    raise CustomError('This is custom error')
```

Ausgabe:

```
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
```

Fange eine benutzerdefinierte Ausnahme

Dieses Beispiel zeigt, wie benutzerdefinierte Ausnahmen abgerufen werden

```
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))
```

Ausgabe:

```
Catched CustomError :Can you catch me ?
```

Benutzerdefinierte Fehler / Ausnahmen auslösen online lesen:
https://riptutorial.com/de/python/topic/10882/benutzerdefinierte-fehler---ausnahmen-auslosen

Kapitel 17: Benutzerdefinierte Methoden

Examples

Benutzerdefinierte Methodenobjekte erstellen

Benutzerdefinierte Methodenobjekte können erstellt werden, wenn ein Attribut einer Klasse abgerufen wird (möglicherweise über eine Instanz dieser Klasse), wenn dieses Attribut ein benutzerdefiniertes Funktionsobjekt, ein ungebundenes benutzerdefiniertes Methodenobjekt oder ein Klassenmethodenobjekt ist.

```
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'>>
```

Wenn das Attribut ein benutzerdefiniertes Methodenobjekt ist, wird ein neues Methodenobjekt nur erstellt, wenn die Klasse, von der es abgerufen wird, mit der im ursprünglichen Methodenobjekt gespeicherten Klasse übereinstimmt oder davon abgeleitet ist. Andernfalls wird das ursprüngliche Methodenobjekt unverändert verwendet.

```
# 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
```

Schildkröte zum Beispiel

Im Folgenden finden Sie ein Beispiel für die Verwendung einer benutzerdefinierten Funktion, die in einem Skript einfach (∞) mal aufgerufen werden kann.

```
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
```

Benutzerdefinierte Methoden online lesen:
https://riptutorial.com/de/python/topic/3965/benutzerdefinierte-methoden

Kapitel 18: Binärdaten

Syntax

- pack (fmt, v1, v2, ...)
- entpacken (fmt, puffer)

Examples

Formatieren Sie eine Liste von Werten in ein Byteobjekt

```
from struct import pack

print(pack('I3c', 123, b'a', b'b', b'c'))  # b'{\x00\x00\x00abc'
```

Entpacken Sie ein Byte-Objekt gemäß einer Formatzeichenfolge

```
from struct import unpack

print(unpack('I3c', b'{\x00\x00\x00abc'))  # (123, b'a', b'b', b'c')
```

Struktur verpacken

Das Modul " **struct** " bietet die Möglichkeit, Python-Objekte als zusammenhängende Byte-Blöcke zu packen oder eine Menge von Byte in Python-Strukturen zu zerlegen.

Die Pack-Funktion akzeptiert eine Formatzeichenfolge und ein oder mehrere Argumente und gibt eine Binärzeichenfolge zurück. Das sieht fast so aus, als würden Sie eine Zeichenfolge formatieren, mit der Ausnahme, dass die Ausgabe keine Zeichenfolge ist, sondern ein Teil von Bytes.

```
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)
```

Ausgabe:

> Native Byte-Reihenfolge: kleiner Byte-Block: '\ x03 \ x00 \ x00 \ x00 \ x04 \ x00 \ x05'
> Byte-Block entpackt: (3, 4, 5) Byte-Block: '\ x03 \ x00 \ x00 \ x00 \ x04 \ x00 \ x05 \ x00 '

Sie können die Bytereihenfolge des Netzwerks für Daten verwenden, die vom Netzwerk empfangen werden, oder Packdaten, um sie an das Netzwerk zu senden.

```
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)
```

Ausgabe:

> Byte-Byte-Byte-Reihenfolge: '\ x03 \ x00 \ x04 \ x00 \ x05 \ x00'
>
> Reihenfolge der Bytes des Byte-Chunks: '\ x00 \ x03 \ x00 \ x04 \ x00 \ x05'

Sie können die Optimierung optimieren, indem Sie den Aufwand für die Zuweisung eines neuen Puffers vermeiden, indem Sie einen zuvor erstellten Puffer bereitstellen.

```
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)
```

Ausgabe:

> Byte-Block: '\ x03 \ x00 \ x04 \ x00 \ x05 \ x00 \ x00 \ x00'
>
> Byte-Block: '\ x00 \ x00 \ x03 \ x00 \ x04 \ x00 \ x05 \ x00'

Binärdaten online lesen: https://riptutorial.com/de/python/topic/2978/binardaten

Kapitel 19: Bitweise Operatoren

Einführung

Bitweise Operationen ändern binäre Zeichenfolgen auf Bitebene. Diese Operationen sind unglaublich einfach und werden vom Prozessor direkt unterstützt. Diese wenigen Operationen sind für die Arbeit mit Gerätetreibern, Low-Level-Grafik, Kryptografie und Netzwerkkommunikation erforderlich. Dieser Abschnitt enthält nützliche Kenntnisse und Beispiele für die bitweisen Operatoren von Python.

Syntax

- x << y # Bitweise Linksverschiebung
- x >> y # Bitweise Rechtsverschiebung
- x & y # Bitweise UND
- x | y # Bitweise ODER
- ~ x # Bitweise NICHT
- x ^ y # Bitweise XOR

Examples

Bitweises AND

Der Operator `&` führt ein binäres **AND aus** , wobei ein Bit kopiert wird, wenn es in **beiden** Operanden vorhanden ist. Das bedeutet:

```
# 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
```

Bitweises ODER

Die `|` Der Operator führt ein binäres "oder" aus, wobei ein Bit kopiert wird, wenn es in einem der

Operanden vorhanden ist. Das bedeutet:

```
# 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
```

Bitweises XOR (exklusives ODER)

Der Operator `^` führt eine binäre **XOR aus,** in der eine binäre `1` genau dann kopiert wird, wenn es sich um den Wert genau **eines** Operanden handelt. Eine andere Möglichkeit dies zu sagen ist, dass das Ergebnis nur `1` , wenn die Operanden unterschiedlich sind. Beispiele beinhalten:

```
# 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
```

Bitweise Linksverschiebung

Der Operator `<<` führt eine bitweise "Linksverschiebung" aus, bei der der Wert des linken Operanden um die vom rechten Operanden angegebene Anzahl Bits nach links verschoben wird.

```
# 2 = 0b10
2 << 2
# Out: 8
# 8 = 0b1000

bin(2 << 2)
# Out: 0b1000
```

Das Durchführen einer Verschiebung um `1` nach links entspricht der Multiplikation mit `2` :

```
7 << 1
# Out: 14
```

Das Durchführen einer Verschiebung des linken Bits um `n` entspricht der Multiplikation mit `2**n` :

```
3 << 4
# Out: 48
```

Bitweise Rechtsverschiebung

Der Operator `>>` führt eine bitweise "Rechtsverschiebung" aus, bei der der Wert des linken Operanden um die vom rechten Operanden angegebene Anzahl von Bits nach rechts verschoben wird.

```
# 8 = 0b1000
8 >> 2
# Out: 2
# 2 = 0b10

bin(8 >> 2)
# Out: 0b10
```

Das Durchführen einer Verschiebung um `1` nach rechts entspricht der Division von Ganzzahlen durch `2` :

```
36 >> 1
# Out: 18

15 >> 1
# Out: 7
```

Das Durchführen einer Verschiebung um `n` rechts nach rechts entspricht der ganzzahligen Division um `2**n` :

```
48 >> 4
# Out: 3

59 >> 3
# Out: 7
```

Bitweises NICHT

Der Operator `~` dreht alle Bits in der Zahl um. Da Computer vorzeichenbehaftete Zahlendarstellungen verwenden - vor allem die Zweierkomplementschreibweise zur Kodierung negativer Binärzahlen, wobei negative Zahlen mit einer führenden Zahl (1) anstelle einer führenden Null (0) geschrieben werden.

Das bedeutet, wenn Sie 8 Bits für die Darstellung Ihrer Zweierkomplement-Zahlen verwenden, würden Sie Muster von `0000 0000` bis `0111 1111` , dass sie Zahlen von 0 bis 127 darstellen, und `1xxx xxxx` für negative Zahlen.

Acht-Bit-Zwei-Komplement-Zahlen

Bits	Vorzeichenloser Wert	Zwei-Komplementwert
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

Im Wesentlichen bedeutet dies, dass `1010 0110` einen vorzeichenlosen Wert von 166 hat (durch Hinzufügen von `(128 * 1) + (64 * 0) + (32 * 1) + (16 * 0) + (8 * 0) + (4 * 1) + (2 * 1) + (1 * 0)`) hat es einen Zweierkomplementwert von -90 (durch Hinzufügen von `(128 * 1) - (64 * 0) - (32 * 1) - (16 * 0) - (8 * 0) - (4 * 1) - (2 * 1) - (1 * 0)` und den Wert ergänzen).

Auf diese Weise reichen negative Zahlen bis -128 (`1000 0000`). Null (0) wird als `0000 0000` und minus Eins (-1) als `1111 1111` .

Im Allgemeinen bedeutet dies jedoch `~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
```

Beachten Sie , dass die Gesamtwirkung dieses Vorgangs bei Anwendung auf positive Zahlen

zusammengefasst werden kann:

```
~n -> -|n+1|
```

Bei einer Anwendung auf negative Zahlen lautet der entsprechende Effekt:

```
~-n -> |n-1|
```

Die folgenden Beispiele veranschaulichen diese letzte Regel ...

```
# -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
```

Inplace-Operationen

Alle Bitwise-Operatoren (außer `~`) verfügen über eine eigene Version

```
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
```

Bitweise Operatoren online lesen: https://riptutorial.com/de/python/topic/730/bitweise-operatoren

Kapitel 20: Boolesche Operatoren

Examples

und

Bewertet das zweite Argument genau dann, wenn beide Argumente wahr sind. Ansonsten wird das erste Falsey-Argument ausgewertet.

```
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)
```

Die `1` ,s in dem obigen Beispiel kann an jeden truthy Wert geändert werden, und die `0` ‘ s kann auf jeden Falsey Wert geändert werden.

oder

Bewertet das erste wahrheitsgemäße Argument, wenn eines der Argumente wahr ist Wenn beide Argumente falsch sind, wird das zweite Argument bewertet.

```
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)
```

Die `1` ,s in dem obigen Beispiel kann an jeden truthy Wert geändert werden, und die `0` ‘ s kann auf jeden Falsey Wert geändert werden.

nicht

Es gibt das Gegenteil der folgenden Anweisung zurück:

```
x = True
y = not x # y = False

x = False
y = not x # y = True
```

Kurzschlussauswertung

Python wertet boolesche Ausdrücke minimal aus .

```
>>> 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
```

"und" und "oder" geben nicht garantiert einen Boolean zurück

Wenn Sie `or` , wird entweder der erste Wert im Ausdruck zurückgegeben, wenn er wahr ist, andernfalls wird der zweite Wert blind angezeigt. Dh `or` entspricht:

```
def or_(a, b):
    if a:
        return a
    else:
        return b
```

Bei `and` wird der erste Wert zurückgegeben, wenn er falsch ist, ansonsten wird der letzte Wert zurückgegeben:

```
def and_(a, b):
    if not a:
        return a
    else:
        return b
```

Ein einfaches Beispiel

In Python können Sie ein einzelnes Element mit zwei binären Operatoren vergleichen - einen auf beiden Seiten:

```
if 3.14 < x < 3.142:
    print("x is near pi")
```

In vielen (den meisten?) Programmiersprachen wird dies im Gegensatz zur regulären Mathematik bewertet: `(3.14 < x) < 3.142` , aber in Python wird es wie `3.14 < x and x < 3.142` , genau wie die meisten Nichtprogrammierer würde erwarten.

Boolesche Operatoren online lesen: https://riptutorial.com/de/python/topic/1731/boolesche-operatoren

Kapitel 21: ChemPy - Python-Paket

Einführung

ChemPy ist ein Python-Paket, das hauptsächlich zur Lösung und Behebung von Problemen in der physikalischen, analytischen und anorganischen Chemie entwickelt wurde. Es ist ein kostenloses Open-Source-Python-Toolkit für Anwendungen in den Bereichen Chemie, Chemieingenieurwesen und Materialwissenschaft.

Examples

Analysieren von Formeln

```
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
```

In der Zusammensetzung werden die Ordnungszahlen (und 0 für die Ladung) als Schlüssel verwendet, und die Anzahl jeder Art wurde zum jeweiligen Wert.

Ausgleichende Stöchiometrie einer chemischen Reaktion

```
 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%'}
```

Ausgleichende Reaktionen

```
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
```

Chemische Gleichgewichte

```
 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
```

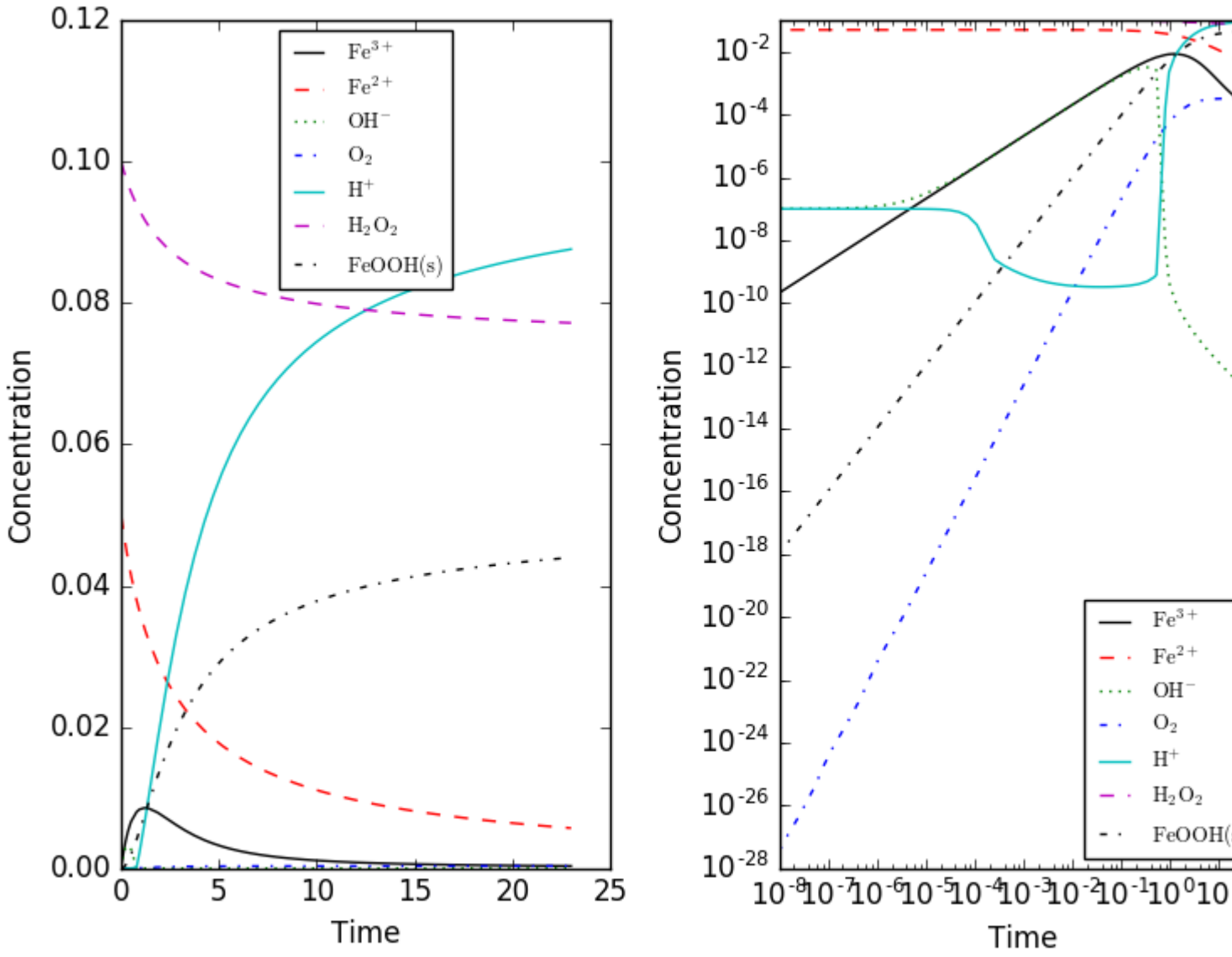
Ionenstärke

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

Chemische Kinetik (System gewöhnlicher Differentialgleichungen)

```
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()
```



ChemPy - Python-Paket online lesen: https://riptutorial.com/de/python/topic/10625/chempy---python-paket

Kapitel 22: CLI-Unterbefehle mit präziser Hilfeausgabe

Einführung

Verschiedene Möglichkeiten zum Erstellen von Unterbefehlen wie `hg` oder `svn` mit der genauen Befehlszeilenschnittstelle und Hilfeausgabe (siehe Abschnitt "Anmerkungen")

Das Analysieren von Befehlszeilenargumenten deckt ein breiteres Thema der Argumentanalyse ab.

Bemerkungen

Verschiedene Möglichkeiten zum Erstellen von Unterbefehlen wie `hg` oder `svn` mit der Befehlszeilenschnittstelle in der Hilfemeldung

```
usage: sub <command>

commands:

  status -  show status
  list   -  print list
```

Examples

Nativer Weg (keine Bibliotheken)

```
"""
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())
```

Ausgabe ohne Argumente:

```
usage: sub <command>

commands:

  status -  show status
  list   -  print list
```

Pros:

- keine deps
- Jeder sollte das lesen können
- vollständige Kontrolle über die Formatierung der Hilfe

argparse (Standard-Hilfeformatierer)

```
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())
```

Ausgabe ohne Argumente:

```
usage: sub {status,list} ...

positional arguments:
  {status,list}
    status       show status
    list         print list
```

Pros:

- kommt mit Python
- Option Parsing ist enthalten

argparse (benutzerdefinierte Formatierungshilfe)

Erweiterte Version von http://www.riptutorial.com/python/example/25282/argparse--default-help-formatter- diese festgelegte Hilfeausgabe

```
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())
```

Ausgabe ohne Argumente:

```
usage: sub <command>

commands:

  status -  show status
  list   -  print list
```

CLI-Unterbefehle mit präziser Hilfeausgabe online lesen:
https://riptutorial.com/de/python/topic/7701/cli-unterbefehle-mit-praziser-hilfeausgabe

Kapitel 23: Codeblöcke, Ausführungsrahmen und Namespaces

Einführung

Ein Codeblock ist ein Stück Python-Programmtext, der als Einheit ausgeführt werden kann, beispielsweise als Modul, Klassendefinition oder Funktionskörper. Einige Codeblöcke (wie Module) werden normalerweise nur einmal ausgeführt, andere (wie Funktionskörper) können mehrmals ausgeführt werden. Codeblöcke können textuell andere Codeblöcke enthalten. Codeblöcke können andere Codeblöcke (die möglicherweise textuell in ihnen enthalten sind) als Teil ihrer Ausführung aufrufen, z.

Examples

Codeblock-Namespaces

Codeblocktyp	Globaler Namensraum	Lokaler Namensraum
Modul	ns für das Modul	gleich wie global
Skript (Datei oder Befehl)	ns für `__main__`	gleich wie global
Interaktiver Befehl	ns für `__main__`	gleich wie global
Klassendefinition	globale ns des enthaltenden Blocks	neuer Namespace
Funktionskörper	globale ns des enthaltenden Blocks	neuer Namespace
String wurde an die `exec` Anweisung übergeben	globale ns des enthaltenden Blocks	lokaler Namespace des enthaltenden Blocks
Zeichenfolge an `eval()`	globale ns des anrufers	lokale ns des anrufers
Datei gelesen von `execfile()`	globale ns des anrufers	lokale ns des anrufers
Ausdruck gelesen von `input()`	globale ns des anrufers	lokale ns des anrufers

Codeblöcke, Ausführungsrahmen und Namespaces online lesen:
https://riptutorial.com/de/python/topic/10741/codeblocke--ausfuhrungsrahmen-und-namespaces

Kapitel 24: Commonwealth-Ausnahmen

Einführung

Hier in Stack Overflow sehen wir häufig Duplikate, die über dieselben Fehler sprechen: `"ImportError: No module named '??????'`, `SyntaxError: invalid syntax` oder `NameError: name '???' is not defined`. Dies ist ein Versuch, sie zu reduzieren und Dokumentation zu haben, auf die verlinkt werden kann.

Examples

Einrückungsfehler (oder EinzugssyntaxErrors)

In den meisten anderen Sprachen ist die Einrückung nicht zwingend, sondern in Python (und anderen Sprachen: frühe Versionen von FORTRAN, Makefiles, Whitespace (esoterische Sprache) usw.). Dies ist jedoch nicht der Fall. wenn Sie Code von einem Beispiel zu Ihrem eigenen kopieren oder einfach, wenn Sie neu sind.

IndentationError / SyntaxError: unerwarteter Einzug

Diese Ausnahme wird ausgelöst, wenn die Einrückung ohne Grund steigt.

Beispiel

Es gibt keinen Grund, das Niveau hier zu erhöhen:

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")
```

Hier gibt es zwei Fehler: den letzten und dass die Einrückung keiner Einrückungsebene entspricht. Es wird jedoch nur eine gezeigt:

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 passt zu keiner äußeren Einrückungsebene

Sieht aus, als ob Sie nicht vollständig entrückt wurden.

Beispiel

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: Ein eingerückter Block wurde erwartet

Nach einem Doppelpunkt (und dann einer neuen Zeile) muss die Einrückungsebene erhöht werden. Dieser Fehler wird ausgelöst, wenn dies nicht der Fall ist.

Beispiel

```
if ok:
doStuff()
```

Anmerkung : Verwenden Sie das Schlüsselwort `pass` (das macht absolut nichts), um einfach eine `if` , `else` , `except` `class` , `method` oder `definition` aber nicht zu sagen, was passiert, wenn der Aufruf / condition true ist (tun Sie es aber später oder im Fall von `except` : einfach nichts tun):

```
def foo():
    pass
```

IndentationError: Inkonsistente Verwendung von Tabulatoren und Leerzeichen beim Einzug

Beispiel

```
def foo():
    if ok:
      return "Two != Four != Tab"
        return "i dont care i do whatever i want"
```

So vermeiden Sie diesen Fehler

Verwenden Sie keine Tabs. Es wird von `PEP8` , dem Style Guide für Python, abgeraten.

1. **Stellen** Sie Ihren Editor so ein, dass er 4 **Leerzeichen** für die Einrückung verwendet.
2. Machen Sie eine Suche und ersetzen Sie sie, um alle Registerkarten durch 4 Leerzeichen zu ersetzen.
3. Stellen Sie sicher, dass Ihr Editor so eingestellt ist, **dass die** Registerkarten als 8 Leerzeichen **angezeigt** werden, sodass Sie den Fehler leicht erkennen und beheben können.

Sehen Sie sich diese Frage an, wenn Sie mehr erfahren möchten.

TypeErrors

Diese Ausnahmen werden verursacht, wenn der Typ eines Objekts unterschiedlich sein sollte

TypeError: [Definition / Methode] dauert? Positionsargumente aber? wurde gegeben

Eine Funktion oder Methode wurde mit mehr (oder weniger) Argumenten als denjenigen aufgerufen, die sie akzeptieren kann.

Beispiel

Wenn mehr Argumente angegeben werden:

```
def foo(a): return a
foo(a,b,c,d) #And a,b,c,d are defined
```

Wenn weniger Argumente angegeben werden:

```
def foo(a,b,c,d): return a += b + c + d
foo(a) #And a is defined
```

Hinweis : Wenn Sie eine unbekannte Anzahl von Argumenten verwenden möchten, können Sie `*args` oder `**kwargs` . Siehe * args und ** kwargs

---

TypeError: nicht unterstützte Operandentypen für [Operand]: '???' und '???'

Einige Typen können je nach Operand nicht zusammen betrieben werden.

Beispiel

Zum Beispiel: `+` wird zum Verketten und Hinzufügen verwendet, aber Sie können keine davon für beide Typen verwenden. Um zum Beispiel eine machen versuchen , `set` durch die Verkettung (`+` ing) `'set1'` und `'tuple1'` gibt den Fehler. Code:

```
set1, tuple1 = {1,2}, (3,4)
a = set1 + tuple1
```

Einige Typen (zB: `int` und `string`) verwenden beide `+` jedoch für verschiedene Dinge:

```
b = 400 + 'foo'
```

Oder sie können nicht einmal für irgendetwas verwendet werden:

```
c = ["a","b"] - [1,2]
```

Sie können aber beispielsweise einen `float` zu einem `int` hinzufügen:

```
d = 1 + 1.0
```

TypeError: '???' Objekt ist nicht iterierbar / subskribierbar:

Für ein Objekt iterable sein kann sequentielle Indizes nimmt von Null ausgehend , bis der Indizes

nicht mehr gültig ist , und ein `IndexError` angehoben wird (Technisch: es ist eine haben , hat `__iter__` Methode , die ein zurückgibt `__iterator__` , oder das eines definiert `__getitem__` Verfahren , das tut was zuvor erwähnt wurde).

Beispiel

Hier sagen wir, dass `bar` der nullte Punkt von 1. ist.

```
foo = 1
bar = foo[0]
```

Dies ist eine diskrete Version: In diesem Beispiel `for` Versuche einstellen `x amount[0]` , das erste Element in einem abzählbaren , aber es kann nicht , weil Menge ist ein int:

```
amount = 10
for x in amount: print(x)
```

TypeError: '???' Objekt ist nicht aufrufbar

Sie definieren eine Variable und rufen sie später auf (wie bei einer Funktion oder Methode).

Beispiel

```
foo = "notAFunction"
foo()
```

NameFehler: Name "???" ist nicht definiert

Wird ausgelöst, wenn Sie versucht haben, eine Variable, Methode oder Funktion zu verwenden, die nicht initialisiert wurde (zumindest nicht vorher). Mit anderen Worten, es wird ausgelöst, wenn ein angeforderter lokaler oder globaler Name nicht gefunden wird. Möglicherweise haben Sie den Namen des Objekts falsch geschrieben oder vergessen, etwas zu `import` . Vielleicht auch in einem anderen Bereich. Wir werden diese mit separaten Beispielen behandeln.

Es ist einfach nirgendwo im Code definiert

Es ist möglich, dass Sie die Initialisierung vergessen haben, insbesondere wenn es sich um eine Konstante handelt

```
foo   # This variable is not defined
bar() # This function is not defined
```

Vielleicht ist es später definiert:

```
baz()

def baz():
    pass
```

Oder es wurde nicht `import` :

```
#needs import math

def sqrt():
    x = float(input("Value: "))
    return math.sqrt(x)
```

Python-Bereiche und die LEGB-Regel:

Die sogenannte LEGB-Regel spricht über die Python-Bereiche. Der Name basiert auf den verschiedenen Bereichen, sortiert nach den entsprechenden Prioritäten:

```
Local → Enclosed → Global → Built-in.
```

- **L** ocal: Variablen, die nicht global deklariert oder in einer Funktion zugewiesen sind.
- **E** nclosing: Variablen in einer Funktion definiert , die innerhalb einer anderen Funktion gewickelt ist.
- **G** lobale: Variablen erklärt global oder auf der obersten Ebene einer Datei zugeordnet.
- **B** uilt-in: Variablen vorbelegt im eingebauten Namen Modul.

Als Beispiel:

```
for i in range(4):
    d = i * 2
print(d)
```

`d` ist zugänglich, weil die `for` Schleife keinen neuen Gültigkeitsbereich markiert, aber wenn dies der Fall wäre, hätten wir einen Fehler und sein Verhalten wäre ähnlich:

```
def noaccess():
    for i in range(4):
        d = i * 2
noaccess()
print(d)
```

Python sagt, dass `NameError: name 'd' is not defined`

Andere Fehler

AssertError

Die `assert` Anweisung existiert in fast jeder Programmiersprache. Wenn Sie das tun:

```
assert condition
```

oder:

```
assert condition, message
```

Das ist gleichbedeutend mit:

```
if __debug__:
    if not condition: raise AssertionError(message)
```

Zusicherungen können eine optionale Nachricht enthalten und Sie können sie deaktivieren, wenn Sie mit dem Debuggen fertig sind.

Hinweis : Die eingebaute Variable **debug** ist unter normalen Umständen True, False, wenn die Optimierung angefordert wird (Befehlszeilenoption -O). Zuweisungen zum **Debuggen** sind illegal. Der Wert für die integrierte Variable wird beim Start des Interpreters festgelegt.

KeyboardInterrupt

Fehler ausgelöst , wenn der Benutzer die Interrupt - Taste drückt, normalerweise `Strg` + `C` oder `del`.

ZeroDivisionError

Sie haben versucht, `1/0` zu berechnen, das nicht definiert ist. Sehen Sie sich dieses Beispiel an, um die Teiler einer Zahl zu finden:

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)
```

`ZeroDivisionError` da die `for` Schleife diesen Wert `x` zuordnet. Stattdessen sollte es sein:

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)
```

Syntaxfehler bei gutem Code

Die grobe Mehrheit der Zeit, zu der ein SyntaxError auf eine uninteressante Zeile verweist, bedeutet, dass ein Problem in der Zeile davor vorliegt (in diesem Beispiel fehlt eine Klammer):

```
def my_print():
    x = (1 + 1
    print(x)
```

Kehrt zurück

```
  File "<input>", line 3
    print(x)
        ^
SyntaxError: invalid syntax
```

Der häufigste Grund für dieses Problem sind nicht übereinstimmende Klammern / Klammern, wie das Beispiel zeigt.

In Python 3 gibt es einen wichtigen Vorbehalt für Druckanweisungen:

Python 3.x 3.0

```
 >>> print "hello world"
  File "<stdin>", line 1
    print "hello world"
                      ^
SyntaxError: invalid syntax
```

Da die Anweisung `print` durch die Funktion `print()` , möchten Sie:

```
print("hello world")  # Note this is valid for both Py2 & Py3
```

Commonwealth-Ausnahmen online lesen:
https://riptutorial.com/de/python/topic/9300/commonwealth-ausnahmen

Kapitel 25: Conditionals

Einführung

Bedingte Ausdrücke mit Schlüsselwörtern wie if, elif und anderen geben Python-Programmen die Möglichkeit, abhängig von einer booleschen Bedingung verschiedene Aktionen auszuführen: True oder False. In diesem Abschnitt wird die Verwendung von Python-Bedingungen, Booleschen Logik und ternären Anweisungen beschrieben.

Syntax

- <Ausdruck> wenn <Bedingt> sonst <Ausdruck> # Ternärer Operator

Examples

wenn, elif und sonst

In Python können Sie eine Reihe von conditionals mit definieren , `if` für die ersten, `elif` für den Rest, bis die endgültigen (optional) `else` für alles , was nicht von dem anderen conditionals gefangen.

```
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.")
```

`Number is bigger than 2` Ausgänge `Number is bigger than 2`

Die Verwendung von `else if` anstelle von `elif` löst einen Syntaxfehler aus und ist nicht zulässig.

Bedingter Ausdruck (oder "Der ternäre Operator")

Der ternäre Operator wird für inline bedingte Ausdrücke verwendet. Es wird am besten für einfache, präzise Operationen verwendet, die leicht zu lesen sind.

- Die Reihenfolge der Argumente unterscheidet sich von vielen anderen Sprachen (wie z. B. C, Ruby, Java usw.). Dies kann zu Fehlern führen, wenn Personen, die mit Pythons "überraschendem" Verhalten nicht vertraut sind, sie verwenden (sie können die Reihenfolge umkehren).
- Einige finden es "unhandlich", da es dem normalen Gedankenfluss zuwiderläuft (zuerst an den Zustand denken und dann an die Auswirkungen).

```
n = 5

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

Das Ergebnis dieses Ausdrucks wird so sein, wie er auf Englisch gelesen wird. Wenn der Bedingungsausdruck "True" ist, wird der Ausdruck auf der linken Seite ausgewertet, ansonsten auf der rechten Seite.

Tenäre Operationen können auch verschachtelt werden, wie hier:

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

Sie bieten auch eine Methode zum Einbinden von Bedingungen in Lambda-Funktionen .

Wenn Aussage

```
if condition:
    body
```

Die `if` Anweisungen prüfen die Bedingung. Wenn es als `True` ausgewertet wird, führt es den Hauptteil der `if` Anweisung aus. Wenn es als `False` ausgewertet wird, wird der Körper übersprungen.

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

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

Die Bedingung kann ein beliebiger gültiger Ausdruck sein:

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

Else Aussage

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

Die else-Anweisung führt ihren Rumpf nur dann aus, wenn alle vorausgehenden Bedingungsanweisungen zu False ausgewertet werden.

```
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!
```

Boolesche Logikausdrücke

Boolesche Logikausdrücke geben nicht nur `True` oder `False` , sondern geben den *Wert zurück* , der als `True` oder `False` interpretiert wurde. Es ist eine Pythonic-Methode, um Logik darzustellen, für die ansonsten ein If-else-Test erforderlich wäre.

Und Betreiber

Der Operator `and` wertet alle Ausdrücke aus und gibt den letzten Ausdruck zurück, wenn alle Ausdrücke den Wert `True` . Andernfalls wird der erste Wert zurückgegeben, der den Wert `False` ergibt:

```
>>> 1 and 2
2

>>> 1 and 0
0

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

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

Oder Betreiber

Der Operator `or` wertet die Ausdrücke von links nach rechts aus und gibt den ersten Wert, der den Wert `True` ergibt, oder den letzten Wert zurück (wenn keiner `True`).

```
>>> 1 or 2
1

>>> None or 1
1

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

Faule Auswertung

Wenn Sie diesen Ansatz verwenden, denken Sie daran, dass die Auswertung faul ist. Ausdrücke, die zur Ermittlung des Ergebnisses nicht ausgewertet werden müssen, werden nicht ausgewertet. Zum Beispiel:

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

Im obigen Beispiel wird `print_me` niemals ausgeführt, da Python feststellen kann, dass der gesamte Ausdruck `False` wenn er auf `0` (`False`) trifft. `print_me` Sie dies, wenn `print_me` muss, um Ihre Programmlogik zu bedienen.

Testen auf mehrere Bedingungen

Ein häufiger Fehler bei der Prüfung auf mehrere Bedingungen ist die falsche Anwendung der Logik.

In diesem Beispiel wird geprüft, ob zwei Variablen jeweils größer als 2 sind. Die Anweisung wird als - `if (a) and (b > 2)` ausgewertet. Dies führt zu einem unerwarteten Ergebnis, da `bool(a)` als `True` ausgewertet wird, wenn `a` nicht Null ist.

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

yes
```

Jede Variable muss separat verglichen werden.

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

no
```

Ein anderer, ähnlicher Fehler wird gemacht, wenn geprüft wird, ob eine Variable einen von mehreren Werten hat. Die Anweisung in diesem Beispiel wird als - `if (a == 3) or (4) or (6)` ausgewertet. Dies führt zu einem unerwarteten Ergebnis, da `bool(4)` und `bool(6)` jeweils `True`

```
>>> a = 1
```

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

yes
```

Wieder muss jeder Vergleich separat durchgeführt werden

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

no
```

Die Verwendung des in-Operators ist die kanonische Schreibweise.

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

no
```

Wahrheitswerte

Die folgenden Werte werden als Falsey betrachtet, da sie bei Anwendung auf einen booleschen Operator als `False` ausgewertet werden.

- Keiner
- Falsch
- `0` oder ein beliebiger numerischer Wert gleich Null, zum Beispiel `0L` , `0.0` , `0j`
- Leere Sequenzen: `''` , `""` , `()` , `[]`
- Leere Zuordnungen: `{}`
- Benutzerdefinierte Typen, bei denen die Methoden `__bool__` oder `__len__` 0 oder `False`

Alle anderen Werte in Python werden mit `True` ausgewertet.

Hinweis: Ein häufiger Fehler besteht darin, einfach nach der Falschheit einer Operation zu suchen, die verschiedene Falsey-Werte zurückgibt, bei denen der Unterschied von Bedeutung ist. Verwenden Sie zum Beispiel `if foo()` anstelle des expliziteren `if foo() is None`

Verwenden der Funktion cmp zum Abrufen des Vergleichsergebnisses zweier Objekte

Python 2 enthält eine `cmp` Funktion, mit der Sie feststellen können, ob ein Objekt kleiner als ein anderes Objekt ist oder diesem entspricht. Diese Funktion kann verwendet werden, um eine Auswahl aus einer Liste basierend auf einer dieser drei Optionen auszuwählen.

Angenommen, Sie müssen `'greater than'` drucken `'greater than'` wenn `x > y` , `'less than'` wenn `x < y` und `'equal'` wenn `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)` gibt die folgenden Werte zurück

Vergleich	Ergebnis
x <y	-1
x == y	0
x> y	1

Diese Funktion wird auf Python entfernt 3. Sie können die Verwendung `cmp_to_key(func)` in sich Helferfunktion `functools` in Python 3 zu alten Vergleichsfunktionen auf die wichtigsten Funktionen zu konvertieren.

Bedingte Ausdrucksauswertung mit List Comprehensions

Mit Python können Sie Listenverständnisse hacken, um bedingte Ausdrücke auszuwerten.

Zum Beispiel,

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

Beispiel:

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

Hier gibt `n<=15` `False` (was in Python gleich 0 ist). Was Python bewertet, ist also:

```
[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

Die eingebaute __cmp __`__cmp__` Methode gab 3 mögliche Werte zurück: 0, 1, -1, wobei cmp (x, y) 0: zurückgab, wenn beide Objekte gleich 1 waren: x> y -1: x <y

Dies könnte bei List Comprehensions verwendet werden, um das erste (dh Index 0), zweite (dh

Index 1) und letzte (dh Index -1) Element der Liste zurückzugeben. Geben Sie uns eine Bedingung dieses Typs:

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

Schließlich wertet Python in allen obigen Beispielen beide Zweige aus, bevor er einen auswählt. Um nur den ausgewählten Zweig auszuwerten:

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

Durch das Hinzufügen von `()` am Ende wird sichergestellt, dass die Lambda-Funktionen nur am Ende aufgerufen / ausgewertet werden. Daher bewerten wir nur den ausgewählten Zweig.

Beispiel:

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

Testen Sie, ob ein Objekt Keine ist, und weisen Sie es zu

Oft möchten Sie einem Objekt etwas zuweisen, wenn es `None` ist, um `None` , dass es noch nicht zugewiesen wurde. Wir werden ein `aDate` .

Der einfachste Weg, dies zu tun, ist die Verwendung des `is None` Tests.

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

(Beachten Sie, dass es mehr Pythonic ist, um zu sagen, `is None` statt `== None` .)

Dies kann jedoch geringfügig optimiert werden, indem die Vorstellung ausgenutzt wird, dass " `not None` in einem boolschen Ausdruck als " `True` ausgewertet wird. Der folgende Code ist gleichwertig:

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

Es gibt aber einen mehr pythonischen Weg. Der folgende Code ist ebenfalls gleichwertig:

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

Dies führt eine Kurzschlussbewertung durch . Wenn `aDate` initialisiert ist und `not None` , wird es sich ohne Nettoeffekt zugewiesen. Wenn `is None` , wird `datetime.date.today()` einem `aDate` zugewiesen.

Conditionals online lesen: https://riptutorial.com/de/python/topic/1111/conditionals

Kapitel 26: configparser

Einführung

Dieses Modul stellt die ConfigParser-Klasse bereit, die eine grundlegende Konfigurationssprache in INI-Dateien implementiert. Sie können dies verwenden, um Python-Programme zu schreiben, die vom Endbenutzer leicht angepasst werden können.

Syntax

- Jede neue Zeile enthält ein neues Schlüsselwertpaar, getrennt durch das Zeichen =
- Schlüssel können in Abschnitte unterteilt werden
- In der INI-Datei wird jeder Abschnittstitel in Klammern geschrieben: []

Bemerkungen

Alle Rückgabewerte von `ConfigParser.ConfigParser().get` sind Strings. Dank `eval` kann es in gängigere Typen umgewandelt werden

Examples

Grundlegende Verwendung

In der config.ini:

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

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

In Python:

```
from ConfigParser import ConfigParser
config = ConfigParser()

#Load configuration file
config.read("config.ini")

# Access the key "debug" in "DEFAULT" section
config.get("DEFAULT", "debug")
# Return 'True'

# Access the key "path" in "FILES" destion
config.get("FILES", "path")
# Return '/path/to/file'
```

Konfigurationsdatei programmatisch erstellen

Konfigurationsdatei enthält Abschnitte, jeder Abschnitt enthält Schlüssel und Werte. Das configparser-Modul kann zum Lesen und Schreiben von Konfigurationsdateien verwendet werden. Erstellen der Konfigurationsdatei: -

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

Die Ausgabedatei enthält folgende Struktur

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

Wenn Sie ein bestimmtes Feld ändern möchten, rufen Sie das Feld ab und weisen Sie den Wert zu

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

configparser online lesen: https://riptutorial.com/de/python/topic/9186/configparser

Kapitel 27: CSV lesen und schreiben

Examples

TSV-Datei schreiben

Python

```
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'])
```

Ausgabedatei

```
$ cat /tmp/output.tsv

name    field
Dijkstra    Computer Science
Shelah    Math
Aumann    Economic Sciences
```

Pandas benutzen

Schreiben Sie eine CSV-Datei aus einem `dict` oder einem `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")
```

Lesen Sie eine CSV-Datei als `DataFrame` und konvertieren Sie sie in ein `dict` :

```
df = pd.read_csv("data.csv")
d = df.to_dict()
```

CSV lesen und schreiben online lesen: https://riptutorial.com/de/python/topic/2116/csv-lesen-und-schreiben

Kapitel 28: ctypes

Einführung

`ctypes` ist eine in Python integrierte Bibliothek, die exportierte Funktionen aus nativen kompilierten Bibliotheken aufruft.

Hinweis: Da diese Bibliothek kompilierten Code verarbeitet, ist sie relativ vom Betriebssystem abhängig.

Examples

Grundlegende Verwendung

Nehmen wir an, wir wollen die `ntohl` Funktion von `libc ntohl` .

Zuerst müssen wir `libc.so` laden:

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

Dann erhalten wir das Funktionsobjekt:

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

Und jetzt können wir einfach die Funktion aufrufen:

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

Was genau das tut, was wir erwarten.

Häufige Fehler

Laden einer Datei fehlgeschlagen

Der erste mögliche Fehler ist das Laden der Bibliothek. In diesem Fall wird normalerweise ein OSError ausgelöst.

Dies liegt entweder daran, dass die Datei nicht existiert (oder vom Betriebssystem nicht gefunden

wird):

```
>>> 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
```

Wie Sie sehen können, ist der Fehler klar und bezeichnend.

Der zweite Grund ist, dass die Datei gefunden wurde, aber nicht das richtige Format hat.

```
>>> cdll.LoadLibrary("libc.so")
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "/usr/lib/python3.5/ctypes/__init__.py", line 425, in LoadLibrary
    return self._dlltype(name)
File "/usr/lib/python3.5/ctypes/__init__.py", line 347, in __init__
    self._handle = _dlopen(self._name, mode)
OSError: /usr/lib/i386-linux-gnu/libc.so: invalid ELF header
```

In diesem Fall ist die Datei eine Skriptdatei und keine `.so` Datei. Dies kann auch passieren, wenn Sie versuchen, eine `.dll` Datei auf einem Linux-Computer oder eine 64-Bit-Datei auf einem 32-Bit-Python-Interpreter zu öffnen. Wie Sie sehen, ist der Fehler in diesem Fall etwas ungenauer und erfordert einige Eingrabungen.

Fehler beim Zugriff auf eine Funktion

Wenn wir die `.so` Datei erfolgreich geladen `.so` , müssen wir wie im ersten Beispiel auf unsere Funktion zugreifen.

Wenn eine nicht vorhandene Funktion verwendet wird, wird ein `AttributeError` ausgelöst:

```
>>> 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
```

Basisobjekt für ctypes

Das grundlegendste Objekt ist ein int:

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

```
c_long(12)
```

Nun bezieht sich `obj` auf einen Speicherblock, der den Wert 12 enthält.

Dieser Wert kann direkt abgerufen und sogar geändert werden:

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

Da sich `obj` auf einen Speicherplatz bezieht, können wir auch die Größe und den Speicherort ermitteln:

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

ctypes Arrays

Wie jeder gute C-Programmierer weiß, wird ein einzelner Wert Sie nicht so weit bringen. Was uns wirklich in Fahrt bringen wird, sind Arrays!

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

Dies ist kein tatsächliches Array, aber es ist verdammt nah! Wir haben eine Klasse erstellt, die ein Array von 16 `int` s kennzeichnet.

Jetzt müssen wir es nur noch initialisieren:

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

Jetzt ist `arr` ein tatsächliches Array, das die Zahlen von 0 bis 15 enthält.

Sie können wie jede Liste aufgerufen werden:

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

Und wie jedes andere `ctypes` Objekt hat es auch eine Größe und einen Ort:

```
>>> sizeof(arr)
```

```
64 # sizeof(c_int) * 16
>>> hex(addressof(arr))
'0xc000l0ff'
```

Wrapping-Funktionen für ctypes

In einigen Fällen akzeptiert eine C-Funktion einen Funktionszeiger. Als avid `ctypes` Benutzer möchten wir diese Funktionen verwenden und sogar die Python-Funktion als Argumente übergeben.

Definieren wir eine Funktion:

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

Nun nimmt diese Funktion zwei Argumente an und gibt ein Ergebnis desselben Typs zurück. Nehmen wir im Beispiel an, dass type ein int ist.

Wie beim Array-Beispiel können wir ein Objekt definieren, das diesen Prototyp kennzeichnet:

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

Dieser Prototyp bezeichnet eine Funktion, die ein `c_int` (das erste Argument) `c_int` und zwei `c_int` Argumente (die anderen Argumente) akzeptiert.

Lassen Sie uns nun die Funktion umschließen:

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

Funktionsprototypen haben mehr Verwendung: Sie können `ctypes` function (wie `libc.ntohl`) `libc.ntohl` und überprüfen, ob beim Aufrufen der Funktion die richtigen Argumente verwendet werden.

```
>>> 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)
```

Komplexe Verwendung

Lassen Sie uns alle obigen Beispiele in einem komplexen Szenario kombinieren: Verwenden Sie die `lfind` Funktion von `libc` .

Weitere Informationen zu dieser Funktion finden Sie in der Manpage . Ich bitte Sie dringend, es zu lesen, bevor Sie fortfahren.

Zuerst definieren wir die richtigen Prototypen:

```
>>> 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)
```

Dann lassen Sie uns die Variablen erstellen:

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

Und jetzt definieren wir die Vergleichsfunktion:

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

Beachten Sie, dass `x` und `y` `POINTER(c_int)` sind `POINTER(c_int)` müssen wir sie dereferenzieren und ihre Werte verwenden, um den im Speicher gespeicherten Wert vergleichen zu können.

Jetzt können wir alles miteinander kombinieren:

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

`ptr` ist der zurückgegebene Leerzeiger. Wenn der `key` nicht in `arr` gefunden wurde, `arr` der Wert `None` , aber in diesem Fall haben wir einen gültigen Wert erhalten.

Jetzt können wir es konvertieren und auf den Wert zugreifen:

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

Wir können auch sehen, dass `ptr` auf den korrekten Wert innerhalb von `arr` :

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

ctypes online lesen: https://riptutorial.com/de/python/topic/9050/ctypes

Kapitel 29: Das base64-Modul

Einführung

Die Base 64-Codierung stellt ein allgemeines Schema für die Codierung von Binärzeichen in das ASCII-Zeichenfolgenformat mit Radix 64 dar. Das Base64-Modul ist Teil der Standardbibliothek, dh es wird zusammen mit Python installiert. Das Verständnis von Bytes und Strings ist für dieses Thema von entscheidender Bedeutung und kann hier überprüft werden . In diesem Thema wird erläutert, wie Sie die verschiedenen Funktionen und Nummernbasis des base64-Moduls verwenden.

Syntax

- base64.b64encode (s, altchars = None)
- base64.b64decode (s, altchars = None, validate = False)
- base64.standard_b64encode (s)
- base64.standard_b64decode (s)
- base64.urlsafe_b64encode (s)
- base64.urlsafe_b64decode (s)
- base64.b32encode (s)
- base64.b32decode (s)
- base64.b16encode (s)
- base64.b16decode (s)
- base64.a85encode (b, foldspaces = False, wrapcol = 0, pad = False, adobe = False)
- base64.a85decode (b, foldpaces = False, adobe = False, ignorechars = b \ t \ n \ r \ v
- base64.b85encode (b, pad = False)
- base64.b85decode (b)

Parameter

Parameter	Beschreibung
`base64.b64encode(s, altchars=None)`	
s	Ein Byte ähnliches Objekt
Altäre	Ein Byte ähnliches Objekt der Länge 2+ von Zeichen, das die Zeichen '+' und '=' beim Erstellen des Base64-Alphabets ersetzt. Zusätzliche Zeichen werden ignoriert.
`base64.b64decode(s, altchars=None, validate=False)`	
s	Ein Byte ähnliches Objekt

Parameter	Beschreibung
Altäre	Ein Byte ähnliches Objekt der Länge 2+ von Zeichen, das die Zeichen '+' und '=' beim Erstellen des Base64-Alphabets ersetzt. Zusätzliche Zeichen werden ignoriert.
bestätigen	Wenn valide True ist, werden die Zeichen, die nicht im normalen Base64-Alphabet oder im alternativen Alphabet sind **,** vor der Auffüllprüfung nicht gelöscht
`base64.standard_b64encode(s)`	
s	Ein Byte ähnliches Objekt
`base64.standard_b64decode(s)`	
s	Ein Byte ähnliches Objekt
`base64.urlsafe_b64encode(s)`	
s	Ein Byte ähnliches Objekt
`base64.urlsafe_b64decode(s)`	
s	Ein Byte ähnliches Objekt
`b32encode(s)`	
s	Ein Byte ähnliches Objekt
`b32decode(s)`	
s	Ein Byte ähnliches Objekt
`base64.b16encode(s)`	
s	Ein Byte ähnliches Objekt
`base64.b16decode(s)`	
s	Ein Byte ähnliches Objekt
`base64.a85encode(b, foldspaces=False, wrapcol=0, pad=False, adobe=False)`	
b	Ein Byte ähnliches Objekt
Falträume	Wenn foldspaces auf True gesetzt ist, wird das Zeichen 'y' anstelle von 4 aufeinander folgenden Leerzeichen verwendet.
Wrapcol	Die Anzahl der Zeichen vor einem Zeilenumbruch

Parameter	Beschreibung
	(0 bedeutet keine Zeilenumbrüche)
Pad	Wenn pad true ist, werden die Bytes vor der Codierung auf ein Vielfaches von 4 aufgefüllt
Adobe	Wenn adobe den Wert True hat, wird die kodierte Sequenz mit '<~' und '' ~> 'wie bei Adobe-Produkten eingerahmt
`base64.a85decode(b, foldspaces=False, adobe=False, ignorechars=b'\t\n\r\v')`	
b	Ein Byte ähnliches Objekt
Falträume	Wenn foldspaces auf True gesetzt ist, wird das Zeichen 'y' anstelle von 4 aufeinander folgenden Leerzeichen verwendet.
Adobe	Wenn adobe den Wert True hat, wird die kodierte Sequenz mit '<~' und '' ~> 'wie bei Adobe-Produkten eingerahmt
Ignorarechars	Ein bytesähnliches Objekt aus Zeichen, das bei der Kodierung ignoriert werden soll
`base64.b85encode(b, pad=False)`	
b	Ein Byte ähnliches Objekt
Pad	Wenn pad true ist, werden die Bytes vor der Codierung auf ein Vielfaches von 4 aufgefüllt
`base64.b85decode(b)`	
b	Ein Byte ähnliches Objekt

Bemerkungen

Bis zum Erscheinen von Python 3.4 funktionierten base64-Codierungs- und -Decodierungsfunktionen nur mit `bytes` oder `bytearray` Typen. Jetzt akzeptieren diese Funktionen alle bytesähnlichen Objekte .

Examples

Codierung und Decodierung von Base64

Um das base64-Modul in Ihr Skript aufzunehmen, müssen Sie es zuerst importieren:

```
import base64
```

Die base64-Codierungs- und -Decodierungsfunktionen erfordern beide ein bytesähnliches Objekt . Um unseren String in Bytes zu bekommen, müssen Sie ihn mit der integrierten Codierungsfunktion von Python codieren. Am häufigsten wird die `UTF-8` Codierung verwendet. Eine vollständige Liste dieser Standardcodierungen (einschließlich Sprachen mit unterschiedlichen Zeichen) finden Sie hier in der offiziellen Python-Dokumentation. Im Folgenden finden Sie ein Beispiel zum Codieren einer Zeichenfolge in Bytes:

```
s = "Hello World!"
b = s.encode("UTF-8")
```

Die Ausgabe der letzten Zeile wäre:

```
b'Hello World!'
```

Das Präfix `b` wird verwendet, um anzugeben, dass der Wert ein Byteobjekt ist.

Um diese Bytes zu codieren, verwenden wir die Funktion `base64.b64encode()` :

```
import base64
s = "Hello World!"
b = s.encode("UTF-8")
e = base64.b64encode(b)
print(e)
```

Dieser Code würde folgendes ausgeben:

```
b'SGVsbG8gV29ybGQh'
```

die noch im Byte-Objekt ist. Um einen String aus diesen Bytes zu erhalten, können wir die `decode()` Methode von Python mit der `UTF-8` Codierung verwenden:

```
import base64
s = "Hello World!"
b = s.encode("UTF-8")
e = base64.b64encode(b)
s1 = e.decode("UTF-8")
print(s1)
```

Die Ausgabe wäre dann:

```
SGVsbG8gV29ybGQh
```

Wenn wir den String codieren und dann decodieren `base64.b64decode()` , können wir die `base64.b64decode()` -Methode verwenden:

```
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)
```

Wie Sie vielleicht erwartet haben, wäre die Ausgabe die Originalzeichenfolge:

```
Base64 Encoded: SGVsbG8gV29ybGQh
Hello World!
```

Codierung und Decodierung von Base32

Das base64-Modul enthält auch Codierungs- und Decodierungsfunktionen für Base32. Diese Funktionen sind den Base64-Funktionen sehr ähnlich:

```
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)
```

Dies würde die folgende Ausgabe erzeugen:

```
Base32 Encoded: JBSWY3DPEBLW64TMMQQQ====
Hello World!
```

Codierung und Decodierung von Base16

Das base64-Modul enthält auch Codierungs- und Decodierungsfunktionen für Base16. Die Basis 16 wird am häufigsten als **Hexadezimal bezeichnet** . Diese Funktionen sind den Funktionen Base64 und Base32 sehr ähnlich:

```
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)
```

Dies würde die folgende Ausgabe erzeugen:

```
Base16 Encoded: 48656C6C6F20576F726C6421
Hello World!
```

Kodierung und Dekodierung von ASCII85

Adobe hat eine eigene Kodierung namens **ASCII85** erstellt, die der von **Base85** ähnelt, jedoch Unterschiede aufweist. Diese Kodierung wird häufig in Adobe PDF-Dateien verwendet. Diese Funktionen wurden in Python Version 3.4 veröffentlicht. Ansonsten `base64.a85encode()` die Funktionen `base64.a85encode()` und `base64.a85encode()` der vorherigen:

```
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)
```

Dies gibt Folgendes aus:

```
ASCII85 Encoded: 87cURD]i,"Ebo80
Hello World!
```

Codierung und Decodierung von Base85

Genau wie die Funktionen Base64, Base32 und Base16 sind die Codierungs- und Decodierungsfunktionen von `base64.b85encode()` und `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)
```

was gibt folgendes aus:

```
Base85 Encoded: NM&qnZy;B1a%^NF
Hello World!
```

Das base64-Modul online lesen: https://riptutorial.com/de/python/topic/8678/das-base64-modul

Kapitel 30: Das dis-Modul

Examples

Konstanten im dis-Modul

```
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', '...
```

Was ist Python-Bytecode?

Python ist ein Hybridinterpreter. Wenn ein Programm ausgeführt wird, fügt es es zunächst in *Bytecode zusammen,* der dann im Python-Interpreter (auch als *virtuelle Python-Maschine bezeichnet*) ausgeführt werden kann. Das Modul `dis` in der Standardbibliothek kann verwendet werden, um den Python-Bytecode durch Disassemblieren von Klassen, Methoden, Funktionen und Codeobjekten lesbar zu machen.

```
>>> 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
```

Der Python-Interpreter ist stapelbasiert und verwendet ein First-In-Last-Out-System.

Jeder Operationscode (Opcode) in der Python-Assemblersprache (der Bytecode) nimmt eine feste Anzahl von Elementen aus dem Stapel und gibt eine feste Anzahl von Elementen an den Stapel zurück. Wenn auf dem Stack nicht genügend Elemente für einen Opcode vorhanden sind, stürzt der Python-Interpreter möglicherweise ohne Fehlermeldung ab.

Module demontieren

Um ein Python-Modul zu zerlegen, muss dieses zunächst in eine `.pyc` Datei umgewandelt werden (Python kompiliert). Um dies zu tun, renne

```
python -m compileall <file>.py
```

Dann laufen Sie in einem Dolmetscher

```
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
```

Dadurch wird ein Python-Modul kompiliert und die Bytecode-Anweisungen mit `dis` ausgegeben. Das Modul wird niemals importiert, so dass die Verwendung mit nicht vertrauenswürdigem Code sicher ist.

Das dis-Modul online lesen: https://riptutorial.com/de/python/topic/1763/das-dis-modul

Kapitel 31: Das Ländereinstellungsmodul

Bemerkungen

Python 2-Dokumente: [https://docs.python.org/2/library/locale.html#locale.currency]

Examples

Währungsformatierung in US-Dollar mit dem Gebietsschema-Modul

```
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'
```

Das Ländereinstellungsmodul online lesen: https://riptutorial.com/de/python/topic/1783/das-landereinstellungsmodul

Kapitel 32: Das os-Modul

Einführung

Dieses Modul bietet eine tragbare Möglichkeit, betriebssystemabhängige Funktionen zu verwenden.

Syntax

- import os

Parameter

Parameter	Einzelheiten
Pfad	Ein Pfad zu einer Datei. Das `os.path.sep` kann mit `os.path.sep` bestimmt werden.
Modus	Die gewünschte Erlaubnis in Oktal (zB `0700`)

Examples

Erstellen Sie ein Verzeichnis

```
os.mkdir('newdir')
```

Wenn Sie die Berechtigungen angeben müssen, können Sie die optionale Verwendung `mode` Argument:

```
os.mkdir('newdir', mode=0700)
```

Aktuelles Verzeichnis abrufen

Verwenden Sie die Funktion `os.getcwd()` :

```
print(os.getcwd())
```

Bestimmen Sie den Namen des Betriebssystems

Das `os` Modul stellt eine Schnittstelle bereit, um zu bestimmen, auf welchem Betriebssystem der Code aktuell ausgeführt wird.

```
os.name
```

Dies kann in Python 3 einen der folgenden Werte zurückgeben:

- `posix`
- `nt`
- `ce`
- `java`

Detailliertere Informationen können von `sys.platform` abgerufen `sys.platform`

Ein Verzeichnis entfernen

Entfernen Sie das Verzeichnis unter `path` :

```
os.rmdir(path)
```

Sie sollten `os.remove()` nicht verwenden, um ein Verzeichnis zu entfernen. Diese Funktion ist für *Dateien* und die Verwendung in Verzeichnissen führt zu einem `OSError`

Einem Symlink folgen (POSIX)

Manchmal müssen Sie das Ziel eines Symlinks bestimmen. `os.readlink` macht das:

```
print(os.readlink(path_to_symlink))
```

Ändern Sie die Berechtigungen für eine Datei

```
os.chmod(path, mode)
```

Dabei ist `mode` die gewünschte Berechtigung in Oktal.

Makedirs - rekursive Verzeichniserstellung

Dazu ein lokales Verzeichnis mit folgendem Inhalt:

```
└── dir1
    ├── subdir1
    └── subdir2
```

Wir möchten dasselbe Unterverzeichnis1, Unterverzeichnis2 unter einem neuen Verzeichnis dir2 erstellen, das noch nicht vorhanden ist.

```
import os

os.makedirs("./dir2/subdir1")
os.makedirs("./dir2/subdir2")
```

Ausführen dieser Ergebnisse in

```
├── dir1
│   ├── subdir1
│   └── subdir2
└── dir2
    ├── subdir1
    └── subdir2
```

dir2 wird nur erstellt, wenn es zum ersten Mal für die Erstellung von Unterverzeichnis1 benötigt wird.

Wenn wir stattdessen **os.mkdir** verwendet **hätten** , hätten wir eine Ausnahme gehabt, weil dir2 noch nicht existiert hätte.

```
    os.mkdir("./dir2/subdir1")
OSError: [Errno 2] No such file or directory: './dir2/subdir1'
```

os.makedirs mag es nicht, wenn das Zielverzeichnis bereits existiert. Wenn wir es noch einmal ausführen:

```
OSError: [Errno 17] File exists: './dir2/subdir1'
```

Dies kann jedoch leicht behoben werden, indem Sie die Ausnahme abfangen und prüfen, ob das Verzeichnis erstellt wurde.

```
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
```

Das os-Modul online lesen: https://riptutorial.com/de/python/topic/4127/das-os-modul

Kapitel 33: Dateien entpacken

Einführung

Zum Extrahieren oder Dekomprimieren einer Tarball-, ZIP- oder Gzip-Datei werden die Python-Tarfile-, Zipfile- und Gzip-Module bereitgestellt. Das Tarfile-Modul von Python bietet die Funktion `TarFile.extractall(path=".", members=None)` zum Extrahieren aus einer Tarball-Datei. Das Zipfile-Modul von Python bietet die Funktion `ZipFile.extractall([path[, members[, pwd]]])` zum Extrahieren oder Entpacken von ZIP-komprimierten Dateien. Schließlich stellt das Gzip-Modul von Python die GzipFile-Klasse zum Dekomprimieren bereit.

Examples

Verwenden Sie Python ZipFile.extractall (), um eine ZIP-Datei zu dekomprimieren

```
file_unzip = 'filename.zip'
unzip = zipfile.ZipFile(file_unzip, 'r')
unzip.extractall()
unzip.close()
```

Verwenden von Python TarFile.extractall () zum Dekomprimieren eines Tarballs

```
file_untar = 'filename.tar.gz'
untar = tarfile.TarFile(file_untar)
untar.extractall()
untar.close()
```

Dateien entpacken online lesen: https://riptutorial.com/de/python/topic/9505/dateien-entpacken

Kapitel 34: Dateien und Ordner E / A

Einführung

Wenn Sie Daten speichern, lesen oder kommunizieren möchten, ist das Arbeiten mit den Dateien eines Betriebssystems mit Python sowohl notwendig als auch einfach. Im Gegensatz zu anderen Sprachen, in denen für die Dateieingabe und -ausgabe komplexes Lesen und Schreiben von Objekten erforderlich ist, vereinfacht Python den Vorgang, indem nur Befehle zum Öffnen, Lesen und Schreiben der Datei benötigt werden. In diesem Thema wird erläutert, wie Python mit Dateien auf dem Betriebssystem kommunizieren kann.

Syntax

- file_object = open (Dateiname [, Zugriffsmodus] [, Pufferung])

Parameter

Parameter	Einzelheiten
Dateiname	den Pfad zu Ihrer Datei oder, wenn sich die Datei im Arbeitsverzeichnis befindet, den Dateinamen Ihrer Datei
Zugriffsmodus	ein Zeichenfolgewert, der bestimmt, wie die Datei geöffnet wird
Pufferung	Ein ganzzahliger Wert, der für die optionale Zeilenpufferung verwendet wird

Bemerkungen

Vermeiden Sie das plattformübergreifende Encoding Hell

Bei der Verwendung von Pythons integriertem `open()` ist es am besten, das `encoding` immer zu übergeben, wenn der Code plattformübergreifend ausgeführt werden soll. Der Grund dafür ist, dass die Standardkodierung eines Systems von Plattform zu Plattform unterschiedlich ist.

Während `linux` Systeme tatsächlich `utf-8` als Standard verwenden, gilt dies ***nicht*** unbedingt für Mac und Windows.

So überprüfen Sie die Standardkodierung eines Systems:

```
import sys
sys.getdefaultencoding()
```

von jedem Python-Interpreter.

Daher ist es ratsam, immer eine Codierung zu spezifizieren, um sicherzustellen, dass die Strings, mit denen Sie arbeiten, so codiert werden, wie Sie sie für wahrscheinlich halten.

```
with open('somefile.txt', 'r', encoding='UTF-8') as f:
    for line in f:
        print(line)
```

Examples

Dateimodi

Es gibt verschiedene Modi, mit denen Sie eine Datei öffnen können, die durch den `mode` angegeben wird. Diese schließen ein:

- `'r'` - Lesemodus. Der Standard. Sie können die Datei nur lesen, nicht ändern. In diesem Modus muss die Datei vorhanden sein.
- `'w'` - Schreibmodus. Falls nicht vorhanden, wird eine neue Datei erstellt. Andernfalls wird die Datei gelöscht und Sie können schreiben.
- `'a'` - Anfügemodus. Es werden Daten an das Ende der Datei geschrieben. Die Datei wird nicht gelöscht, und die Datei muss für diesen Modus vorhanden sein.
- `'rb'` - Lesemodus im Binärmodus. Dies ist ähnlich wie bei `r` nur dass das Lesen im binären Modus erzwungen wird. Dies ist auch eine Standardeinstellung.
- `'r+'` - Lesemodus und gleichzeitig Schreibmodus. Auf diese Weise können Sie Dateien gleichzeitig lesen und beschreiben, ohne `r` und `w` .
- `'rb+'` - Lese- und Schreibmodus im Binärmodus. Dasselbe wie `r+` außer dass die Daten binär sind
- `'wb'` - Schreibmodus in binär. Dasselbe wie `w` außer dass die Daten binär sind.
- `'w+'` - Schreib- und Lesemodus. Genau wie `r+` aber wenn die Datei nicht existiert, wird eine neue erstellt. Andernfalls wird die Datei überschrieben.
- `'wb+'` - Schreib- und Lesemodus im Binärmodus. Dasselbe wie `w+` aber die Daten sind binär.
- `'ab'` - im binären Modus anhängen. Ähnlich wie `a` dass die Daten binär sind.
- `'a+'` - Anhänge- und Lesemodus. Ähnlich wie `w+` da eine neue Datei erstellt wird, wenn die Datei nicht vorhanden ist. Andernfalls befindet sich der Dateizeiger am Ende der Datei, sofern vorhanden.
- `'ab+'` - Anfügen und Lesen im Binärmodus. Dasselbe wie `a+` außer dass die Daten binär sind.

```
with open(filename, 'r') as f:
    f.read()
with open(filename, 'w') as f:
    f.write(filedata)
with open(filename, 'a') as f:
    f.write('\n' + newdata)
```

	r	r +	w	w +	ein	a +
Lesen	✔	✔	✘	✔	✘	✔
Schreiben	✘	✔	✔	✔	✔	✔
Erzeugt eine Datei	✘	✘	✔	✔	✔	✔
Löscht die Datei	✘	✘	✔	✔	✘	✘
Ausgangsposition	Start	Start	Start	Start	Ende	Ende

Python 3 fügte einen neuen Modus für die `exclusive creation` damit Sie nicht versehentlich vorhandene Dateien abschneiden oder überschreiben.

- `'x'` - offen für exklusive Erstellung, wird `FileExistsError` wenn die Datei bereits vorhanden ist
- `'xb'` - offen für den Schreibmodus für exklusive Schreibvorgänge in binär. Dasselbe wie `x` außer dass die Daten binär sind.
- `'x+'` - Lese- und Schreibmodus. Ähnlich wie `w+` da eine neue Datei erstellt wird, wenn die Datei nicht vorhanden ist. Andernfalls wird `FileExistsError` .
- `'xb+'` - Schreib- und Lesemodus. Genau wie `x+` aber die Daten sind binär

	x	x +
Lesen	✘	✔
Schreiben	✔	✔
Erzeugt eine Datei	✔	✔
Löscht die Datei	✘	✘
Ausgangsposition	Start	Start

Erlauben Sie einem, Ihren offenen Code für Ihre Datei pythonischer zu schreiben:

Python 3.x 3.3

```
try:
    with open("fname", "r") as fout:
        # Work with your open file
except FileExistsError:
```

```
    # Your error handling goes here
```

In Python 2 hätten Sie so etwas gemacht

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
```

Zeile für Zeile eine Datei lesen

Die einfachste Möglichkeit, eine Datei zeilenweise zu durchlaufen:

```
with open('myfile.txt', 'r') as fp:
    for line in fp:
        print(line)
```

`readline()` ermöglicht eine detailliertere Kontrolle der zeilenweisen Iteration. Das Beispiel unten entspricht dem obigen Beispiel:

```
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)
```

Die gemeinsame Verwendung von for-Loop-Iterator und readline () gilt als schlechte Praxis.

Normalerweise wird die `readlines()` -Methode zum Speichern einer iterierbaren Auflistung der Dateizeilen verwendet:

```
with open("myfile.txt", "r") as fp:
    lines = fp.readlines()
for i in range(len(lines)):
    print("Line " + str(i) + ": " + line)
```

Dies würde Folgendes drucken:

Zeile 0: Hallo

Zeile 1: Welt

Den vollständigen Inhalt einer Datei abrufen

Die bevorzugte Methode für Datei-E / A ist die Verwendung des Schlüsselworts `with` . Dadurch wird sichergestellt, dass das Dateihandle geschlossen wird, sobald das Lesen oder Schreiben abgeschlossen ist.

```
with open('myfile.txt') as in_file:
    content = in_file.read()

print(content)
```

oder zu handhaben, die Datei manuell zu schließen, können Sie verzichten `with` und rufen Sie einfach `close` Sie sich:

```
in_file = open('myfile.txt', 'r')
content = in_file.read()
print(content)
in_file.close()
```

Denken Sie daran, dass Sie ohne die Verwendung einer `with` Anweisung die Datei aus Versehen offen halten können, falls eine unerwartete Ausnahme auftritt:

```
in_file = open('myfile.txt', 'r')
raise Exception("oops")
in_file.close()  # This will never be called
```

In eine Datei schreiben

```
with open('myfile.txt', 'w') as f:
    f.write("Line 1")
    f.write("Line 2")
    f.write("Line 3")
    f.write("Line 4")
```

Wenn Sie `myfile.txt` öffnen, werden Sie `myfile.txt` sehen:

> Zeile 1Line 2Line 3Line 4

Python fügt nicht automatisch Zeilenumbrüche ein. Sie müssen dies manuell tun:

```
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")
```

> Linie 1
> Zeile 2
> Zeile 3
> Zeile 4

Verwenden Sie `os.linesep` als `os.linesep` , wenn Sie im Textmodus geöffnete Dateien schreiben (Standardeinstellung). Verwenden Sie stattdessen `\n` .

Wenn Sie eine Codierung angeben möchten, fügen Sie einfach die `encoding` Parameter an die `open` Funktion:

```
with open('my_file.txt', 'w', encoding='utf-8') as f:
    f.write('utf-8 text')
```

Es ist auch möglich, die print-Anweisung zum Schreiben in eine Datei zu verwenden. Die Mechanismen unterscheiden sich in Python 2 von Python 3, das Konzept ist jedoch das gleiche, dass Sie die Ausgabe, die auf den Bildschirm gegangen wäre, nehmen und sie stattdessen in eine Datei senden kann.

Python 3.x 3.0

```
with open('fred.txt', 'w') as outfile:
    s = "I'm Not Dead Yet!"
    print(s) # writes to stdout
    print(s, file = outfile) # writes to outfile

    #Note: it is possible to specify the file parameter AND write to the screen
    #by making sure file ends up with a None value either directly or via a variable
    myfile = None
    print(s, file = myfile) # writes to stdout
    print(s, file = None)   # writes to stdout
```

In Python 2 hätten Sie so etwas gemacht

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
```

Im Gegensatz zur Verwendung der Schreibfunktion fügt die Druckfunktion automatisch Zeilenumbrüche ein.

Inhalte einer Datei in eine andere Datei kopieren

```
with open(input_file, 'r') as in_file, open(output_file, 'w') as out_file:
    for line in in_file:
        out_file.write(line)
```

- Verwenden des `shutil` Moduls:

```
import shutil
shutil.copyfile(src, dst)
```

Prüfen Sie, ob eine Datei oder ein Pfad vorhanden ist

Verwenden Sie den EAFP- Codierstil und `try` zu öffnen.

```
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
```

Dadurch werden auch Race-Bedingungen vermieden, wenn ein anderer Prozess die Datei zwischen der Prüfung und der Verwendung gelöscht hat. Diese Wettlaufsituation kann in folgenden Fällen auftreten:

- Verwenden des `os` Moduls:

  ```
  import os
  os.path.isfile('/path/to/some/file.txt')
  ```

Python 3.x 3.4

- `pathlib` :

  ```
  import pathlib
  path = pathlib.Path('/path/to/some/file.txt')
  if path.is_file():
      ...
  ```

Um zu überprüfen, ob ein bestimmter Pfad existiert oder nicht, können Sie das oben beschriebene EAFP-Verfahren befolgen oder den Pfad explizit überprüfen:

```
import os
path = "/home/myFiles/directory1"

if os.path.exists(path):
    ## Do stuff
```

Kopieren Sie eine Verzeichnisstruktur

```
import shutil
source='//192.168.1.2/Daily Reports'
destination='D:\\Reports\\Today'
shutil.copytree(source, destination)
```

Das Zielverzeichnis **darf noch nicht existieren** .

Dateien iterieren (rekursiv)

Um alle Dateien, auch in Unterverzeichnissen, zu iterieren, verwenden Sie os.walk:

```
import os
for root, folders, files in os.walk(root_dir):
    for filename in files:
        print root, filename
```

root_dir kann "." um vom aktuellen Verzeichnis oder einem anderen Pfad aus zu starten.

Python 3.x 3.5

Wenn Sie auch Informationen über die Datei erhalten möchten, können Sie die effizientere Methode os.scandir wie folgt verwenden :

```
for entry in os.scandir(path):
   if not entry.name.startswith('.') and entry.is_file():
       print(entry.name)
```

Lesen Sie eine Datei zwischen einem Zeilenbereich

Nehmen wir an, Sie möchten nur zwischen bestimmten Zeilen einer Datei iterieren

`itertools` können Sie `itertools`

```
import itertools

with open('myfile.txt', 'r') as f:
    for line in itertools.islice(f, 12, 30):
        # do something here
```

Dies wird durch die Zeilen 13 bis 20 gelesen, da bei der Python-Indexierung von 0 aus begonnen wird. Daher wird Zeile 1 als 0 indiziert

Da können auch einige zusätzliche Zeilen gelesen werden, indem hier das `next()` Schlüsselwort verwendet wird.

Wenn Sie das Dateiobjekt als iterierbares Objekt verwenden, verwenden Sie die `readline()` Anweisung hier nicht, da die beiden Verfahren zum Durchlaufen einer Datei nicht miteinander gemischt werden sollen

Zufälliger Dateizugriff mit mmap

Durch Verwendung des `mmap` Moduls kann der Benutzer willkürlich auf Positionen in einer Datei zugreifen, indem er die Datei in den Arbeitsspeicher einfügt. Dies ist eine Alternative zur Verwendung normaler Dateivorgänge.

```
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()
```

Ersetzen von Text in einer Datei

```
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='')
```

Prüfen, ob eine Datei leer ist

```
>>> import os
>>> os.stat(path_to_file).st_size == 0
```

oder

```
>>> import os
>>> os.path.getsize(path_to_file) > 0
```

Beide geben jedoch eine Ausnahme aus, wenn die Datei nicht vorhanden ist. Um zu vermeiden, einen solchen Fehler abfangen zu müssen, führen Sie folgende Schritte aus:

```
import os
def is_empty_file(fpath):
    return os.path.isfile(fpath) and os.path.getsize(fpath) > 0
```

was einen `bool` Wert `bool` .

Dateien und Ordner E / A online lesen: https://riptutorial.com/de/python/topic/267/dateien-und-ordner-e---a

Kapitel 35: Daten kopieren

Examples

Durchführen einer flachen Kopie

Eine flache Kopie ist eine Kopie einer Sammlung, ohne eine Kopie ihrer Elemente auszuführen.

```
>>> import copy
>>> c = [[1,2]]
>>> d = copy.copy(c)
>>> c is d
False
>>> c[0] is d[0]
True
```

Durchführen einer tiefen Kopie

Wenn Sie verschachtelte Listen haben, ist es wünschenswert, auch die verschachtelten Listen zu klonen. Diese Aktion wird als Tiefenkopie bezeichnet.

```
>>> import copy
>>> c = [[1,2]]
>>> d = copy.deepcopy(c)
>>> c is d
False
>>> c[0] is d[0]
False
```

Durchführen einer flachen Kopie einer Liste

Sie können mit Slices flache Kopien von Listen erstellen.

```
>>> l1 = [1,2,3]
>>> l2 = l1[:]     # Perform the shallow copy.
>>> l2
[1,2,3]
>>> l1 is l2
False
```

Wörterbuch kopieren

Ein Dictionary - Objekt hat die Methode `copy` . Es führt eine flache Kopie des Wörterbuchs aus.

```
>>> d1 = {1:[]}
>>> d2 = d1.copy()
>>> d1 is d2
False
>>> d1[1] is d2[1]
```

```
True
```

Ein Set kopieren

Sets haben auch eine `copy` . Mit dieser Methode können Sie eine flache Kopie erstellen.

```
>>> s1 = {()}
>>> s2 = s1.copy()
>>> s1 is s2
False
>>> s2.add(3)
>>> s1
{[]}
>>> s2
{3,[]}
```

Daten kopieren online lesen: https://riptutorial.com/de/python/topic/920/daten-kopieren

Kapitel 36: Datenbankzugriff

Bemerkungen

Python kann viele verschiedene Arten von Datenbanken verarbeiten. Für jeden dieser Typen gibt es eine andere API. Um die Ähnlichkeit zwischen diesen verschiedenen APIs zu fördern, wurde PEP 249 eingeführt.

> Diese API wurde definiert, um die Ähnlichkeit zwischen den Python-Modulen zu fördern, die für den Zugriff auf Datenbanken verwendet werden. Auf diese Weise hoffen wir, eine Konsistenz zu erreichen, die zu leicht verständlicheren Modulen, zu generell mehr portablem Code für Datenbanken und zu einer größeren Reichweite der Datenbankkonnektivität von Python führt. PEP-249

Examples

Zugriff auf MySQL-Datenbank mit MySQLdb

Als Erstes müssen Sie mit der connect-Methode eine Verbindung zur Datenbank herstellen. Danach benötigen Sie einen Cursor, der mit dieser Verbindung arbeitet.

Verwenden Sie die Ausführungsmethode des Cursors, um mit der Datenbank zu interagieren, und machen Sie die Änderungen gelegentlich mit der Festschreibungsmethode des Verbindungsobjekts fest.

Vergessen Sie nicht, den Cursor und die Verbindung zu schließen.

Hier ist eine Dbconnect-Klasse mit allem, was Sie brauchen.

```
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()
```

Die Interaktion mit der Datenbank ist einfach. Nachdem Sie das Objekt erstellt haben, verwenden

Sie einfach die Ausführungsmethode.

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

Wenn Sie eine gespeicherte Prozedur aufrufen möchten, verwenden Sie die folgende Syntax. Beachten Sie, dass die Parameterliste optional ist.

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

Nachdem die Abfrage abgeschlossen ist, können Sie auf mehrere Arten auf die Ergebnisse zugreifen. Das Cursorobjekt ist ein Generator, der alle Ergebnisse abrufen oder als Schleife ausführen kann.

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

Wenn Sie eine Schleife verwenden möchten, die direkt den Generator verwendet:

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

Wenn Sie Änderungen an der Datenbank festschreiben möchten:

```
db.commit_db()
```

Wenn Sie den Cursor und die Verbindung schließen wollen:

```
db.close_db()
```

SQLite

SQLite ist eine leichtgewichtige, festplattenbasierte Datenbank. Da kein separater Datenbankserver erforderlich ist, wird er häufig zum Prototyping oder für kleine Anwendungen verwendet, die häufig von einem einzelnen Benutzer oder von einem Benutzer zu einem bestimmten Zeitpunkt verwendet werden.

```
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()
```

Der obige Code stellt eine Verbindung zu der Datenbank her, die in der Datei mit dem Namen `users.db` gespeichert ist. Die Datei wird zuerst erstellt, wenn sie noch nicht vorhanden ist. Sie können über SQL-Anweisungen mit der Datenbank interagieren.

Das Ergebnis dieses Beispiels sollte sein:

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

Die SQLite-Syntax: Eine eingehende Analyse

Fertig machen

1. Importieren Sie das sqlite-Modul mit

   ```
   >>> import sqlite3
   ```

2. Um das Modul verwenden zu können, müssen Sie zuerst ein Connection-Objekt erstellen, das die Datenbank darstellt. Hier werden die Daten in der Datei example.db gespeichert:

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

   Alternativ können Sie auch den speziellen Namen `:memory:` erstellen Sie eine temporäre Datenbank im RAM:

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

3. Sobald Sie eine `Connection` , können Sie ein `Cursor` Objekt erstellen und seine Methode `execute()` aufrufen, um SQL-Befehle auszuführen:

   ```
   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()
```

Wichtige Eigenschaften und Funktionen der `Connection`

1. `isolation_level`

   Mit diesem Attribut wird die aktuelle Isolationsstufe abgerufen oder festgelegt. Keine für den Autocommit-Modus oder eine der `DEFERRED` , `IMMEDIATE` oder `EXCLUSIVE` .

2. `cursor`

   Das Cursorobjekt wird verwendet, um SQL-Befehle und Abfragen auszuführen.

3. `commit()`

   Übernimmt die aktuelle Transaktion.

4. `rollback()`

   Macht alle Änderungen rückgängig, die seit dem letzten Aufruf von `commit()`

5. `close()`

   Schließt die Datenbankverbindung. `commit()` automatisch aufgerufen. Wenn `close()` aufgerufen wird, ohne vorher `commit()` aufzurufen (vorausgesetzt, Sie befinden sich nicht im Autocommit-Modus), gehen alle vorgenommenen Änderungen verloren.

6. `total_changes`

   Ein Attribut, das die Gesamtzahl der Zeilen protokolliert, die seit dem Öffnen der Datenbank geändert, gelöscht oder eingefügt wurden.

7. `execute` , `executemany` und `executescript`

   Diese Funktionen funktionieren genauso wie die des Cursorobjekts. Dies ist eine Abkürzung, da der Aufruf dieser Funktionen über das Verbindungsobjekt zur Erstellung eines Cursor-Zwischenobjekts und zum Aufrufen der entsprechenden Methode des Cursorobjekts führt

8. `row_factory`

   Sie können dieses Attribut in eine aufrufbare Eigenschaft ändern, die den Cursor und die ursprüngliche Zeile als Tupel akzeptiert und die tatsächliche Ergebniszeile zurückgibt.

```
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
```

Wichtige Funktionen des `Cursor`

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

   Führt eine *einzelne* SQL-Anweisung aus. Die SQL-Anweisung kann parametrisiert werden (dh Platzhalter statt SQL-Literale). Das Modul sqlite3 unterstützt zwei Arten von Platzhaltern: Fragezeichen `?` ("Qmark style") und benannte Platzhalter `: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())
   ```

   Achtung: Verwenden Sie `%s` zum Einfügen von Zeichenfolgen in SQL-Befehle, da Ihr Programm dadurch anfällig für einen SQL-Injection-Angriff werden kann (siehe SQL Injection).

2. `executemany(sql, seq_of_parameters)`

   Führt einen SQL-Befehl für alle in der Sequenz sql gefundenen Parametersequenzen oder -zuordnungen aus. Das Modul sqlite3 ermöglicht auch die Verwendung eines Iterators, der anstelle einer Sequenz Parameter liefert.

   ```
   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)
   ```

   Sie können auch Iterator-Objekte als Parameter an das ausführende Unternehmen übergeben, und die Funktion durchläuft jedes Tupel von Werten, das der Iterator zurückgibt. Der Iterator muss ein Tupel von Werten zurückgeben.

   ```
   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)`

   Dies ist eine nicht dem Standard entsprechende Methode zum gleichzeitigen Ausführen mehrerer SQL-Anweisungen. Es gibt zuerst eine `COMMIT` Anweisung aus und führt dann das SQL-Skript aus, das es als Parameter erhält.

   `sql_script` kann eine Instanz von `str` oder `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
     );
     """)
```

   Die nächsten Funktionen werden in Verbindung mit `SELECT` Anweisungen in SQL verwendet. Um Daten nach der Ausführung einer `SELECT` Anweisung abzurufen, können Sie den Cursor entweder als Iterator behandeln, die Methode `fetchone()` des Cursors `fetchone()` , um eine

einzelne übereinstimmende Zeile `fetchall()` , oder `fetchall()` aufrufen, um eine Liste der übereinstimmenden Zeilen `fetchall()` .

Beispiel für das Iterator-Formular:

```
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()`

   Ruft die nächste Zeile eines Abfrageergebnissatzes ab und gibt eine einzelne Sequenz zurück oder Keine, wenn keine weiteren Daten verfügbar sind.

```
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)`

   Ruft die nächsten Zeilen eines Abfrageergebnisses ab (durch Größe angegeben) und gibt eine Liste zurück. Wenn size nicht angegeben wird, gibt fetchmany eine einzelne Zeile zurück. Eine leere Liste wird zurückgegeben, wenn keine weiteren Zeilen verfügbar sind.

```
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()`

Ruft alle (verbleibenden) Zeilen eines Abfrageergebnisses ab und gibt eine Liste zurück.

```
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)]
```

SQLite- und Python-Datentypen

SQLite unterstützt nativ die folgenden Typen: NULL, INTEGER, REAL, TEXT, BLOB.

Auf diese Weise werden die Datentypen konvertiert, wenn von SQL zu Python oder umgekehrt gewechselt wird.

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

PostgreSQL-Datenbankzugriff mit psycopg2

psycopg2 ist der beliebteste PostgreSQL-Datenbankadapter, der leicht und effizient ist. Es ist die aktuelle Implementierung des PostgreSQL-Adapters.

> Seine Hauptmerkmale sind die vollständige Implementierung der Python DB API 2.0-Spezifikation und die Threadsicherheit (mehrere Threads können dieselbe Verbindung gemeinsam nutzen).

Herstellen einer Verbindung zur Datenbank und Erstellen einer Tabelle

```
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()
```

Daten in die Tabelle einfügen:

```
# 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)""")
```

Tabellendaten abrufen:

```
# 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]))
```

Die Ausgabe des obigen wäre:

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

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

Und jetzt wissen Sie die Hälfte von allem, was Sie über ***psycopg2*** wissen ***müssen*** ! :)

Oracle-Datenbank

Voraussetzungen:

- cx_Oracle-Paket - Hier finden Sie alle Versionen
- Oracle Instant Client - Für Windows x64 , Linux x64

Konfiguration:

- Installieren Sie das cx_Oracle-Paket wie folgt:

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

- Extrahieren Sie den Oracle Instant Client und legen Sie die Umgebungsvariablen wie folgt fest:

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

Verbindung herstellen:

```
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()
```

Datenbankversion abrufen:

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

Sample-Ausgabe: ['12 ',' 1 ',' 0 ',' 2 ',' 0 ']

Abfrage ausführen: SELECT

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

Die Ausgabe erfolgt in Python-Tupeln:

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

(23, 'HR ASSOCIATE', 'MENSCHLICHE RESSOURCEN', 6)

Abfrage ausführen: INSERT

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

Wenn Sie Einfüge- / Aktualisierungs- / Löschvorgänge in einer Oracle-Datenbank durchführen,

sind die Änderungen nur in Ihrer Sitzung verfügbar, bis Sie ein `commit` ausführen. Wenn die aktualisierten Daten für die Datenbank festgeschrieben sind, stehen sie anderen Benutzern und Sitzungen zur Verfügung.

Abfrage ausführen: INSERT mit Bind-Variablen

Referenz

Bindungsvariablen ermöglichen es Ihnen, Anweisungen mit neuen Werten erneut auszuführen, ohne dass die Anweisung erneut analysiert werden muss. Bindungsvariablen verbessern die Wiederverwendbarkeit von Code und können das Risiko von SQL Injection-Angriffen verringern.

```
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()
```

Verbindung schließen:

```
_db_connection.close()
```

Die close () -Methode schließt die Verbindung. Alle Verbindungen, die nicht explizit geschlossen wurden, werden automatisch freigegeben, wenn das Skript endet.

Verbindung

Verbindung herstellen

Gemäß PEP 249 sollte die Verbindung zu einer Datenbank mithilfe eines `connect()` Konstruktors hergestellt werden, der ein `Connection` Objekt zurückgibt. Die Argumente für diesen Konstruktor sind datenbankabhängig. In den datenbankspezifischen Themen finden Sie die relevanten Argumente.

```
import MyDBAPI

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

Dieses Verbindungsobjekt verfügt über vier Methoden:

1: schließen

```
con.close()
```

Schließt die Verbindung sofort. Beachten Sie, dass die Verbindung automatisch geschlossen wird, wenn die `Connection.__del__` Methode aufgerufen wird. Alle ausstehenden Transaktionen werden implizit zurückgesetzt.

2: begehen

```
con.commit()
```

Überträgt jede ausstehende Transaktion in die Datenbank.

3: Rollback

```
con.rollback()
```

Rollt zum Beginn einer ausstehenden Transaktion zurück. Mit anderen Worten: Dies bricht jede nicht festgeschriebene Transaktion in der Datenbank ab.

4: Cursor

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

Gibt ein `Cursor` Objekt zurück. Damit werden Transaktionen in der Datenbank ausgeführt.

Sqlalchemy verwenden

So verwenden Sie sqlalchemy für die Datenbank:

```
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
```

Jetzt kann diese Engine verwendet werden: zB mit Pandas, um Dataframes direkt von MySQL abzurufen

```
import pandas as pd

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

Datenbankzugriff online lesen: https://riptutorial.com/de/python/topic/4240/datenbankzugriff

Kapitel 37: Datenserialisierung

Syntax

- unpickled_string = pickle.loads (string)
- unpickled_string = pickle.load (file_object)
- pickled_string = pickle.dumps ([('', 'cmplx'), {('object',): Keine}], pickle.HIGHEST_PROTOCOL)
- pickle.dump (('', 'cmplx'), {('object',): Keine}], file_object, pickle.HIGHEST_PROTOCOL)
- unjsoned_string = json.loads (string)
- 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)

Parameter

Parameter	Einzelheiten
`protocol`	Verwendung von `pickle` oder `cPickle` , ist es die Methode , die Objekte werden Serialized / deserialisiert. `pickle.HIGHEST_PROTOCOL` möchten `pickle.HIGHEST_PROTOCOL` hier `pickle.HIGHEST_PROTOCOL` , was die neueste Methode bedeutet.

Bemerkungen

Warum JSON verwenden?

- Sprachübergreifende Unterstützung
- Für Menschen lesbar
- Anders als bei Pickle besteht keine Gefahr, beliebigen Code auszuführen

Warum nicht JSON verwenden?

- Unterstützt keine Pythonic-Datentypen
- Schlüssel in Wörterbüchern dürfen nur Datentypen für Zeichenfolgen sein.

Warum Pickle?

- Gute Möglichkeit, Pythonic zu serialisieren (Tupel, Funktionen, Klassen)
- Schlüssel in Wörterbüchern können jeden Datentyp haben.

Warum nicht Pickle?

- Sprachübergreifende Unterstützung fehlt
- Es ist nicht sicher, beliebige Daten zu laden

Examples

Serialisierung mit JSON

JSON ist eine vielsprachige Methode zur Serialisierung von Daten

Unterstützte Datentypen: *int* , *float* , *boolean* , *string* , *list* und *dict* . Weitere Informationen finden Sie unter -> JSON-Wiki

Hier ist ein Beispiel, das die **grundlegende** Verwendung von **JSON veranschaulicht** : -

```
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)
```

Ausführliche Informationen zu JSON finden Sie im JSON-Modul .

Serialisierung mit Pickle

Hier ist ein Beispiel, das die **grundlegende** Verwendung von **Pickle** demonstriert: -

```
# 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)
```

Siehe Pickle für detaillierte Informationen über Pickle.

WARNUNG : Die offizielle Dokumentation zu Pickle macht deutlich, dass es keine Sicherheitsgarantien gibt. Laden Sie keine Daten, deren Herkunft Sie nicht kennen.

Datenserialisierung online lesen: https://riptutorial.com/de/python/topic/3347/datenserialisierung

Kapitel 38: Datenserialisierung von Pickles

Syntax

- pickle.dump (Objekt, Datei, Protokoll) #So serialisiert ein Objekt
- pickle.load (file) #Umserialisieren eines Objekts
- pickle.dumps (object, protocol) # Um ein Objekt in Bytes zu serialisieren
- pickle.loads (buffer) # Um ein Objekt von Bytes zu trennen

Parameter

Parameter	Einzelheiten
Objekt	Das Objekt, das gespeichert werden soll
Datei	Die geöffnete Datei, die das Objekt enthalten soll
Protokoll	Das zum Beizen des Objekts verwendete Protokoll (optionaler Parameter)
Puffer	Ein Byte-Objekt, das ein serialisiertes Objekt enthält

Bemerkungen

Pickleable Typen

Die folgenden Objekte sind picklierbar.

- `None` , `True` und `False`
- Zahlen (aller Art)
- Saiten (aller Art)
- `tuple` s, `list` s, `set` s und `dict` s , die nur picklable Objekte
- Funktionen, die auf der obersten Ebene eines Moduls definiert sind
- eingebaute Funktionen
- Klassen, die auf der obersten Ebene eines Moduls definiert sind
    - Instanzen solcher Klassen, deren `__dict__` oder das Ergebnis des Aufrufs von `__getstate__()` picklierbar ist (Details finden Sie in den offiziellen Dokumenten).

Basierend auf der offiziellen Python-Dokumentation .

`pickle` und Sicherheit

Das Pickle-Modul ist **nicht sicher** . Es sollte nicht verwendet werden, wenn die serialisierten Daten von einer nicht vertrauenswürdigen Partei, z. B. über das Internet, empfangen werden.

Examples

Verwenden von Pickle zum Serialisieren und Deserialisieren eines Objekts

Das `pickle` Modul implementiert einen Algorithmus zum Umwandeln eines beliebigen Python-Objekts in eine Reihe von Bytes. Dieser Vorgang wird auch als **Serialisierung** des Objekts bezeichnet. Der das Objekt repräsentierende Byte-Datenstrom kann dann übertragen oder gespeichert und später rekonstruiert werden, um ein neues Objekt mit denselben Eigenschaften zu erstellen.

Für den einfachsten Code verwenden wir die Funktionen `dump()` und `load()` .

Das Objekt wird serialisiert

```
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)
```

Das Objekt deserialisieren

```
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)
```

Pickle- und Byte-Objekte verwenden

Es ist auch möglich , in serialisiert und deserialisiert aus Byte Objekten, die unter Verwendung von `dumps` und `loads` - Funktion, die äquivalent ist zu `dump` und `load` .

```
serialized_data = pickle.dumps(data, pickle.HIGHEST_PROTOCOL)
# type(serialized_data) is bytes

deserialized_data = pickle.loads(serialized_data)
# deserialized_data == data
```

Angepasste Daten anpassen

Einige Daten können nicht gebeizt werden. Andere Daten sollten aus anderen Gründen nicht gepickt werden.

Was `__getstate__` wird, kann in der `__getstate__` Methode definiert werden. Diese Methode muss etwas Pickelbares zurückgeben.

Auf der gegenüberliegenden Seite ist `__setstate__` : Er erhält das, was `__getstate__` erstellt hat, und muss das Objekt initialisieren.

```
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
```

Jetzt kann das gemacht werden:

```
>>> 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
```

Die Implementierung hier enthält eine Liste mit einem Wert: `[self.important_data]` . Das war nur ein Beispiel, `__getstate__` hätte alles zurückgeben können, was picklierbar ist, solange `__setstate__` weiß, wie die Gegenseite zu tun ist. Eine gute Alternative ist ein Wörterbuch aller

Werte: `{'important_data': self.important_data}`.

Konstruktor wird nicht aufgerufen! Beachten Sie, dass im vorherigen Beispiel `a2` in `pickle.loads` ohne dass `A.__init__` `A.__setstate__` musste `A.__setstate__` alles initialisieren, das `__init__` initialisiert hätte, wenn es aufgerufen würde.

Datenserialisierung von Pickles online lesen:
https://riptutorial.com/de/python/topic/2606/datenserialisierung-von-pickles

Kapitel 39: Datenvisualisierung mit Python

Examples

Matplotlib

Matplotlib ist eine mathematische Plotterbibliothek für Python, die eine Vielzahl verschiedener Plotfunktionen bietet.

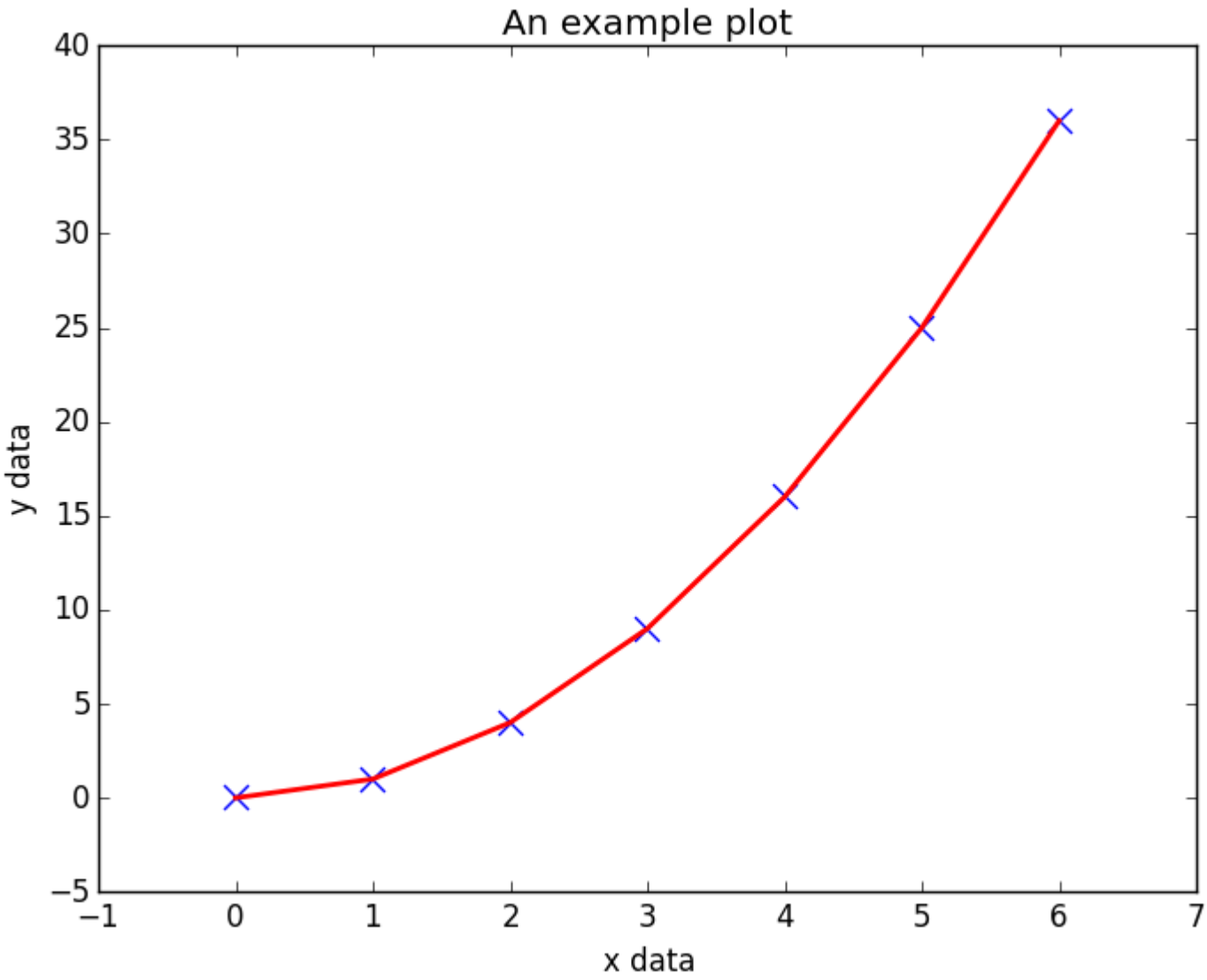
Die Matplotlib-Dokumentation finden Sie hier , die SO-Dokumente stehen hier zur Verfügung.

Matplotlib bietet zwei verschiedene Methoden zum Plotten, die jedoch größtenteils austauschbar sind:

- Erstens bietet matplotlib die `pyplot` Schnittstelle, eine direkte und einfach zu verwendende Schnittstelle, die das `pyplot` komplexer Diagramme in einem MATLAB-ähnlichen Stil ermöglicht.
- Zweitens ermöglicht matplotlib dem Benutzer, die verschiedenen Aspekte (Achsen, Linien, Hilfsstriche usw.) direkt über ein objektbasiertes System zu steuern. Dies ist schwieriger, ermöglicht jedoch die vollständige Kontrolle über die gesamte Grafik.

Im Folgenden sehen Sie ein Beispiel für die Verwendung der `pyplot` Schnittstelle zum `pyplot` einiger generierter Daten:

```
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()
```



Beachten Sie, dass `plt.show()` bekanntermaßen in einigen Umgebungen problematisch ist, weil `matplotlib.pyplot` im interaktiven Modus ausgeführt wird. Wenn dies der Fall ist, kann das Blockierungsverhalten explizit überschrieben werden, indem das optionale Argument `plt.show(block=True)` , um das Problem zu mildern.

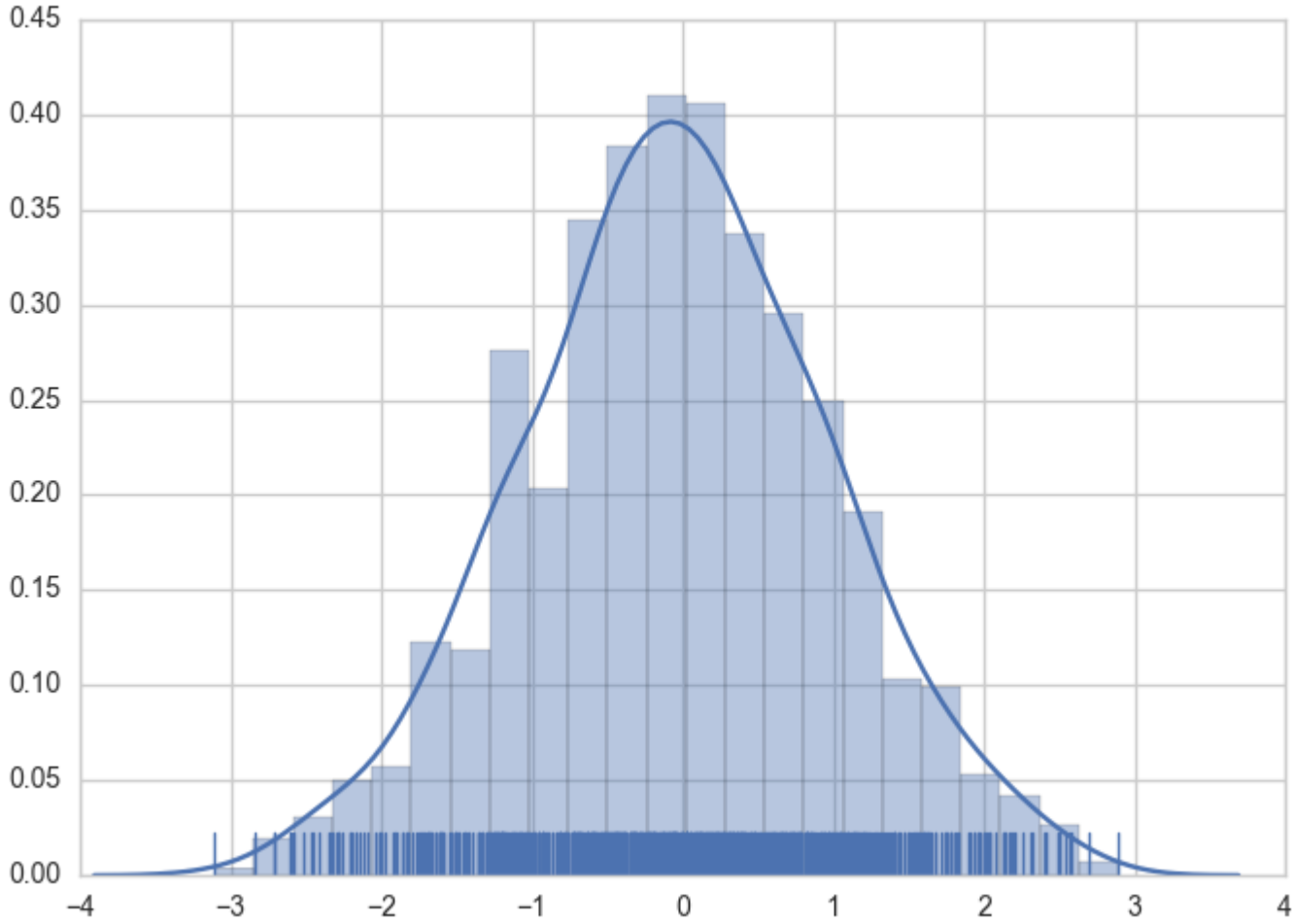
Seaborn

Seaborn ist eine Hülle um Matplotlib, die das Erstellen gemeinsamer statistischer Diagramme vereinfacht. Die Liste der unterstützten Diagramme umfasst univariate und bivariate Verteilungsdiagramme, Regressionsdiagramme und eine Reihe von Methoden zum Zeichnen kategorialer Variablen. Die vollständige Liste der Grundstücke, die Seaborn bietet, finden Sie in der API-Referenz .

Das Erstellen von Diagrammen in Seaborn ist so einfach wie das Aufrufen der entsprechenden Grafikfunktion. Hier ein Beispiel zum Erstellen eines Histogramms, einer Schätzung der Kerndichte und eines Rug-Plots für zufällig generierte Daten.

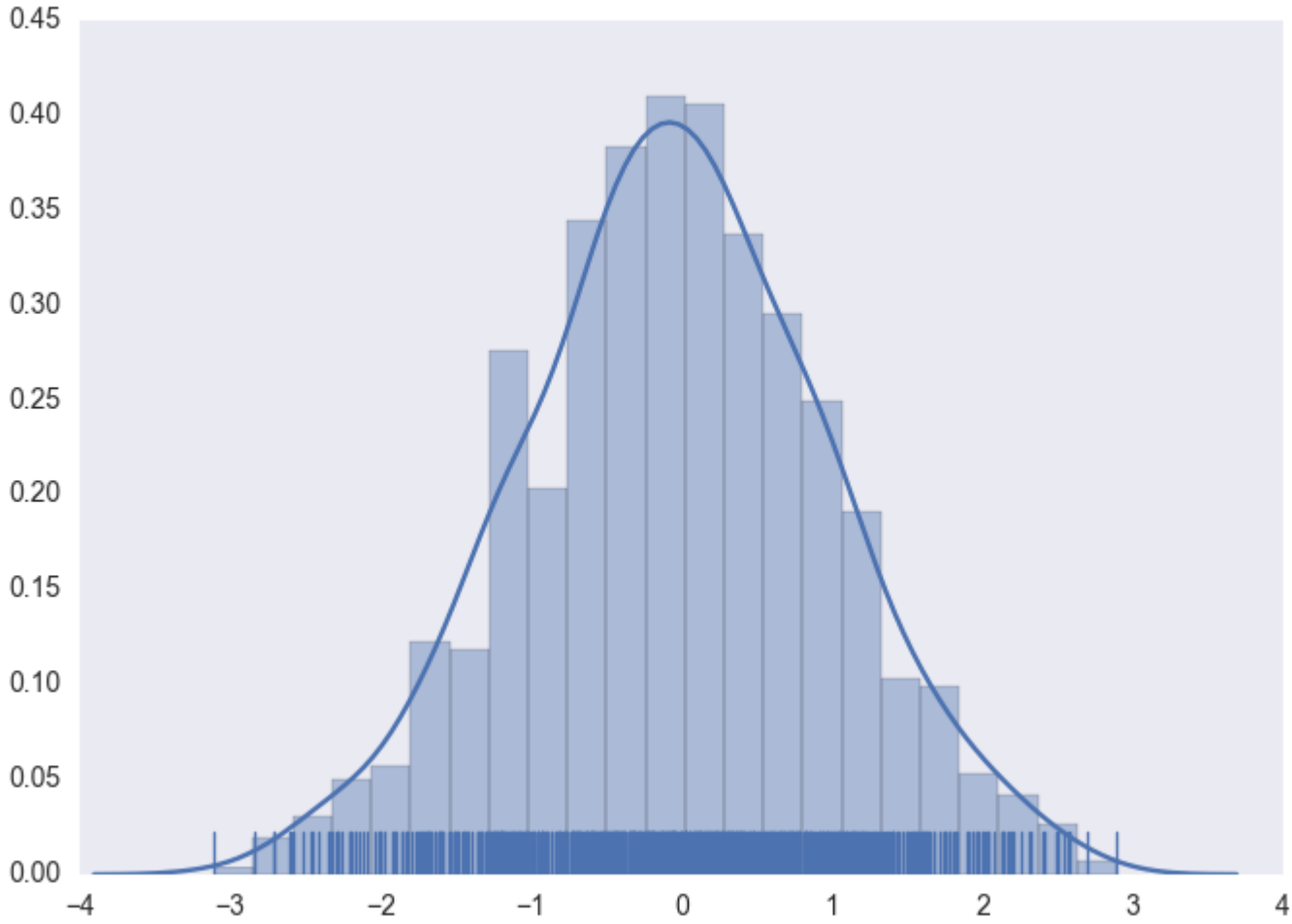
```
import numpy as np  # numpy used to create data from plotting
```

```
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)
```



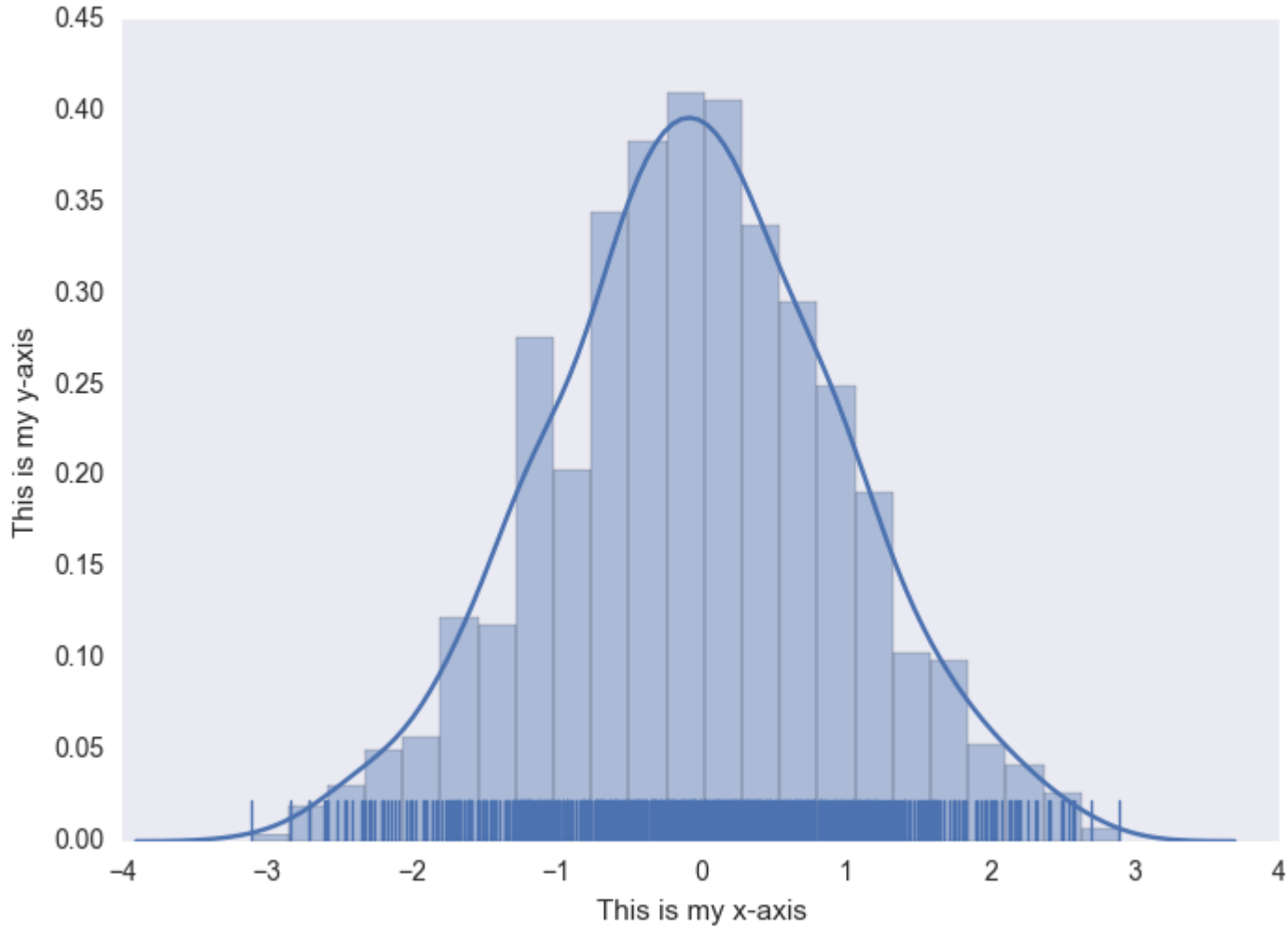
Der Stil des Diagramms kann auch mit einer deklarativen Syntax gesteuert werden.

```
# 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)
```



Als zusätzlicher Bonus können noch normale Matplotlib-Befehle auf Seaborn-Plots angewendet werden. Hier ein Beispiel zum Hinzufügen von Achsentiteln zu unserem zuvor erstellten Histogramm.

```
# 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 ist ein 3D-Visualisierungswerkzeug für wissenschaftliche Daten. Es verwendet das Visualization Tool Kit oder VTK unter der Haube. Mit der Kraft von VTK kann **MayaVI** eine Vielzahl von dreidimensionalen Darstellungen und Abbildungen **erstellen** . Es ist als separate Softwareanwendung und auch als Bibliothek verfügbar. Ähnlich wie Matplotlib bietet diese Bibliothek eine objektorientierte Programmiersprachenschnittstelle, um Plots zu erstellen, ohne **VTK** zu kennen.

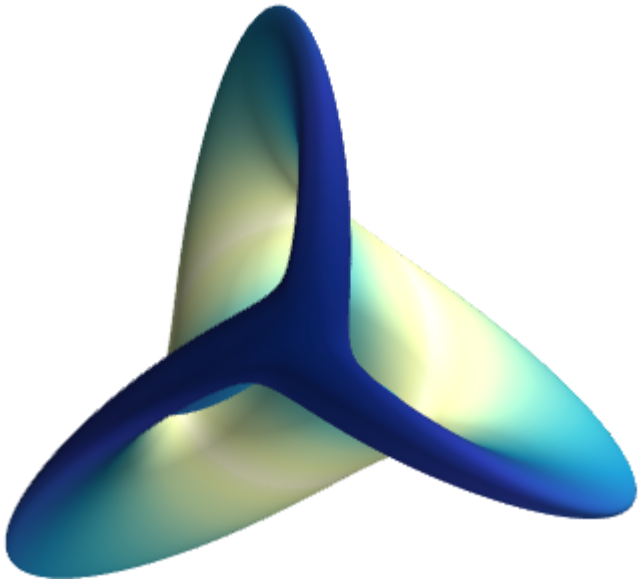
MayaVI ist nur in der Python 2.7x-Serie verfügbar! Es wird gehofft, bald in Python 3-x-Serie verfügbar zu sein! (Obwohl einige Erfolge bei der Verwendung der Abhängigkeiten in Python 3 zu verzeichnen sind)

Dokumentation finden Sie hier . Einige Galeriebeispiele finden Sie hier

Hier ist ein Beispieldiagramm, das mit **MayaVI** aus der Dokumentation erstellt wurde.

```
# 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 ist eine moderne Plattform zum Plotten und zur Datenvisualisierung. **Plotly** ist eine Bibliothek für **Python** , **R** , **JavaScript** , **Julia** und **MATLAB** . Es kann auch als Webanwendung mit diesen Sprachen verwendet werden.

Benutzer können die plotly-Bibliothek installieren und nach der Benutzerauthentifizierung offline verwenden. Die Installation dieser Bibliothek und Offline - Authentifizierung ist gegeben hier . Die Diagramme können auch in **Jupyter Notebooks erstellt** werden.

Die Verwendung dieser Bibliothek erfordert ein Konto mit Benutzername und Kennwort. Dadurch wird der Arbeitsbereich zum Speichern von Plots und Daten in der Cloud bereitgestellt.

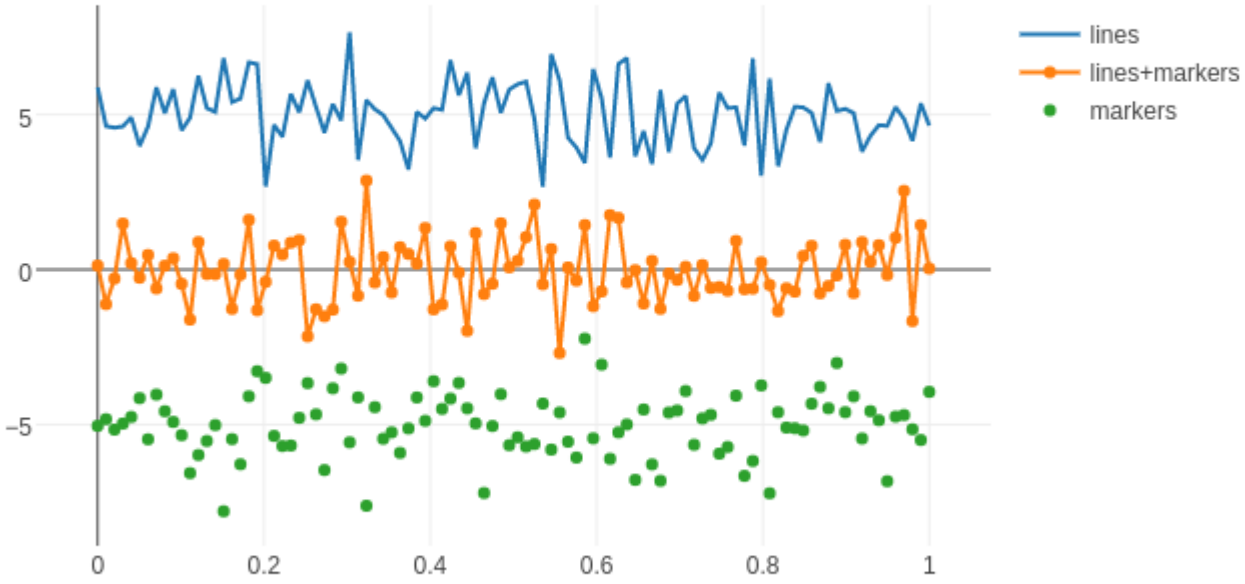
Die kostenlose Version der Bibliothek hat einige geringfügig eingeschränkte Funktionen und ist für die Erstellung von 250 Plots pro Tag konzipiert. Die kostenpflichtige Version bietet alle

Funktionen, unbegrenzte Plot-Downloads und mehr privaten Datenspeicher. Für weitere Details kann man die Hauptseite hier besuchen.

Dokumentation und Beispiele finden Sie hier

Ein Beispieldiagramm aus den Dokumentationsbeispielen:

```
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')
```



Datenvisualisierung mit Python online lesen:
https://riptutorial.com/de/python/topic/2388/datenvisualisierung-mit-python

Kapitel 40: Datum und Uhrzeit

Bemerkungen

Python bietet sowohl integrierte Methoden als auch externe Bibliotheken zum Erstellen, Ändern, Analysieren und Bearbeiten von Datum und Uhrzeit .

Examples

Parsen einer Zeichenfolge in ein Zeitzonenobjekt mit Zeitzone

Python 3.2+ unterstützt das `%z` Format, wenn eine Zeichenfolge in ein `datetime` Objekt `datetime` wird.

> UTC-Offset in der Form `+HHMM` oder `-HHMM` (leere Zeichenfolge, wenn das Objekt nicht aktiv ist).

Python 3.x 3.2

```
import datetime
dt = datetime.datetime.strptime("2016-04-15T08:27:18-0500", "%Y-%m-%dT%H:%M:%S%z")
```

Für andere Python-Versionen können Sie eine externe Bibliothek wie `dateutil` , wodurch das Analysieren einer Zeichenfolge mit Zeitzone in ein `datetime` Objekt schnell ist.

```
import dateutil.parser
dt = dateutil.parser.parse("2016-04-15T08:27:18-0500")
```

Die Variable `dt` ist jetzt ein `datetime` Objekt mit dem folgenden Wert:

```
datetime.datetime(2016, 4, 15, 8, 27, 18, tzinfo=tzoffset(None, -18000))
```

Einfache Datumsberechnung

Termine existieren nicht isoliert. Es ist üblich, dass Sie die Zeit zwischen den Datumsangaben oder das Datum für morgen bestimmen müssen. Dies kann mit `timedelta` Objekten erreicht werden

```
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)
```

Dies führt zu ähnlichen Ergebnissen:

```
Today: 2016-04-15
Yesterday: 2016-04-14
Tomorrow: 2016-04-16
Difference between tomorrow and yesterday: 2 days, 0:00:00
```

Grundlegende Verwendung von datetime-Objekten

Das datetime-Modul enthält drei Haupttypen von Objekten - date, time und datetime.

```
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)
```

Arithmetische Operationen für diese Objekte werden nur innerhalb desselben Datentyps unterstützt. Wenn Sie eine einfache Arithmetik mit Instanzen verschiedener Typen ausführen, wird ein TypeError-Wert erzeugt.

```
# 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)
```

Iteriere über Termine

Manchmal möchten Sie einen Datumsbereich von einem Startdatum bis zu einem Enddatum durchlaufen. Sie können dies mit einer `datetime` Bibliothek und einem `timedelta` tun:

```
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)
```

Was produziert:

```
2016-07-21
2016-07-22
2016-07-23
2016-07-24
2016-07-25
2016-07-26
2016-07-27
```

Analysieren einer Zeichenfolge mit einem kurzen Zeitzonennamen in ein Zeitzonenobjekt mit Datum und Uhrzeit

Wenn Sie die `dateutil` Bibliothek wie im vorherigen Beispiel zum Analysieren von Zeitzonen mit Zeitzonen verwenden , ist es auch möglich, Zeitmarken mit einem angegebenen "kurzen" Zeitzonennamen zu parsen.

Für Datumsangaben, die mit kurzen Zeitzonennamen oder Abkürzungen formatiert sind, die im Allgemeinen mehrdeutig sind (z. B. CST, z. B. Central Standard Time, China Standard Time, Cuba Standard Time usw.) - weitere Informationen finden Sie hier , ist es erforderlich, eine Zuordnung zwischen der Abkürzung für die Zeitzone und dem `tzinfo` Objekt `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)
```

Nach dem Ausführen dieses:

```
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'))
```

Es ist erwähnenswert, dass bei Verwendung einer `pytz` Zeitzone mit dieser Methode diese *nicht* richtig lokalisiert wird:

```
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})
```

Dadurch wird einfach die `pytz` Zeitzone an die datetime angehängt:

```
dt.tzinfo # Will be in Local Mean Time!
# <DstTzInfo 'America/New_York' LMT-1 day, 19:04:00 STD>
```

Wenn Sie diese Methode verwenden, sollten Sie nach dem Parsen wahrscheinlich den naiven Teil der datetime neu `localize` :

```
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>)
```

Zeitzonenbasierte Datenzeiten erstellen

Standardmäßig sind alle `datetime` Objekte nicht aktiv. Um sie `tzinfo` zu machen, müssen Sie ein `tzinfo` Objekt anhängen, das den UTC-Offset und die Zeitzonenabkürzung als Funktion von Datum und Uhrzeit bereitstellt.

Versetzte Zeitzonen korrigiert

Für Zeitzonen, die einen festen Versatz von UTC haben, stellt das Modul `datetime` in Python 3.2 und `datetime` die `timezone` bereit, eine konkrete Implementierung von `tzinfo` , die ein Zeit- `timedelta` und einen (optionalen) Namensparameter benötigt:

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'
```

Für Python-Versionen vor 3.2 ist es erforderlich, eine Drittanbieter-Bibliothek wie `dateutil` .

`dateutil` stellt eine äquivalente Klasse, `tzoffset` , zur `tzoffset` , die (ab Version 2.5.3) Argumente der Form `dateutil.tz.tzoffset(tzname, offset)` , wobei der `offset` in Sekunden angegeben wird:

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'
```

Zonen mit Sommerzeit

Für Zonen mit Sommerzeit stellen Python-Standardbibliotheken keine Standardklasse bereit. Daher ist die Verwendung einer Drittanbieter-Bibliothek erforderlich. `pytz` und `dateutil` sind beliebte Bibliotheken, die Zeitzonenklassen anbieten.

Neben statischen Zeitzonen bietet `dateutil` Zeitzonen-Klassen, die Sommerzeit verwenden (siehe Dokumentation zum `tz` Modul). Mit der Methode `tz.gettz()` Sie ein Zeitzonenobjekt `datetime` , das direkt an den `datetime` Konstruktor übergeben werden kann:

```
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
```

VORSICHT : Seit Version 2.5.3 verarbeitet `dateutil` mehrdeutige Datumsangaben nicht korrekt und verwendet immer das *spätere* Datum. Es ist nicht möglich, ein Objekt mit einer `dateutil` Zeitzone zu `dateutil` , die beispielsweise `2015-11-01 1:30 EDT-4` , da dies *während* einer Sommerzeitumstellung erfolgt.

Bei Verwendung von `pytz` alle Kantenfälle ordnungsgemäß `pytz` . `pytz` Zeitzonen sollten *jedoch* nicht direkt durch den Konstruktor an Zeitzonen angehängt werden. Stattdessen sollte eine `pytz` Zeitzone mit der `localize` Methode der Zeitzone angefügt werden:

```
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
```

Beachten Sie, dass Sie, wenn Sie eine Datums-Zeit-Arithmetik in einer `pytz` aware-Zeitzone durchführen, entweder die Berechnungen in UTC durchführen müssen (wenn Sie absolut abgelaufene Zeit wünschen) oder das Ergebnis mit `normalize()` aufrufen:

```
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
```

Fuzzy-Datetime-Analyse (Extrahieren von Datetime aus einem Text)

Es ist möglich, ein Datum aus einem Text mit dem `dateutil` Parser in einem "Fuzzy" -Modus zu extrahieren, wobei Komponenten der Zeichenfolge, die nicht als Teil eines Datums erkannt werden, ignoriert werden.

```
from dateutil.parser import parse

dt = parse("Today is January 1, 2047 at 8:21:00AM", fuzzy=True)
print(dt)
```

`dt` ist jetzt ein *`datetime` Objekt,* und `datetime.datetime(2047, 1, 1, 8, 21)` gedruckt.

Zwischen Zeitzonen wechseln

Um zwischen Zeitzonen zu wechseln, benötigen Sie datetime-Objekte, die Zeitzonen berücksichtigen.

```
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.
```

Parsen eines beliebigen ISO 8601-Zeitstempels mit minimalen Bibliotheken

Python unterstützt die Analyse von Zeitstempeln nach ISO 8601 nur eingeschränkt. Für `strptime` Sie genau wissen, in welchem Format es sich befindet. Als Komplikation ist die Stringifizierung einer `datetime` ein Zeitstempel nach ISO 8601 mit Leerzeichen als Trennzeichen und 6-stelligem Bruch:

```
str(datetime.datetime(2016, 7, 22, 9, 25, 59, 555555))
# '2016-07-22 09:25:59.555555'
```

Ist der Bruch 0, wird kein Bruchteil ausgegeben

```
str(datetime.datetime(2016, 7, 22, 9, 25, 59, 0))
# '2016-07-22 09:25:59'
```

Diese beiden Formulare benötigen jedoch ein *anderes* Format für die `strptime` . Darüber hinaus unterstützt `strptime' does not support at all parsing minute timezones that have a` Folgendes enthalten: `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` kann nicht.

Es gibt eine Einzeldateibibliothek namens `iso8601` die nur Zeitstempel von ISO 8601 und nur sie richtig analysiert.

Es unterstützt Brüche und Zeitzonen und das `T` Trennzeichen mit einer einzigen Funktion:

```
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' ...>)
```

Wenn keine Zeitzone eingestellt ist, wird der `iso8601.parse_date` auf UTC gesetzt. Die Standardzone kann mit `default_zone` Schlüsselwortargument `default_zone` geändert werden. Bemerkenswert ist , wenn dies `None` anstelle des Standard, dann werden diese Zeitstempel , die stattdessen als naive Datetimes keine explizite Zeitzone haben werden zurückgegeben:

```
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>)
```

Zeitstempel in Datumszeit konvertieren

Das `datetime` Modul kann einen POSIX- `timestamp` in ein ITC- `datetime` Objekt `datetime` .

Die Epoche ist am 1. Januar 1970 um Mitternacht.

```
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)
```

Monate genau von einem Datum abziehen

Verwenden des `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)
```

Verwenden des `dateutils` Moduls

```
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)
```

Zeitunterschiede berechnen

Das `timedelta` Modul ist nützlich, um Unterschiede zwischen den Zeiten zu berechnen:

```
from datetime import datetime, timedelta
now = datetime.now()
then = datetime(2016, 5, 23)     # datetime.datetime(2016, 05, 23, 0, 0, 0)
```

Die Angabe der Uhrzeit ist optional, wenn ein neues `datetime` Objekt erstellt wird

```
delta = now-then
```

`delta` ist vom Typ `timedelta`

```
print(delta.days)
# 60
print(delta.seconds)
# 40826
```

Um einen Tag nach dem Tag und einen Tag vor dem Datum zu erhalten, könnten wir Folgendes verwenden:

n Tag nach dem Datum:

```
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 Tage vor dem Datum:

```
 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)
```

Erhalten Sie einen Zeitstempel nach ISO 8601

Ohne Zeitzone mit Mikrosekunden

```
from datetime import datetime

datetime.now().isoformat()
# Out: '2016-07-31T23:08:20.886783'
```

Mit Zeitzone, mit Mikrosekunden

```
from datetime import datetime
from dateutil.tz import tzlocal

datetime.now(tzlocal()).isoformat()
# Out: '2016-07-31T23:09:43.535074-07:00'
```

Mit Zeitzone, ohne Mikrosekunden

```
from datetime import datetime
from dateutil.tz import tzlocal

datetime.now(tzlocal()).replace(microsecond=0).isoformat()
# Out: '2016-07-31T23:10:30-07:00'
```

Siehe ISO 8601 für weitere Informationen über die ISO 8601 - Format.

Datum und Uhrzeit online lesen: https://riptutorial.com/de/python/topic/484/datum-und-uhrzeit

Kapitel 41: Datumsformatierung

Examples

Zeit zwischen zwei Datumszeiten

```
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
```

Zeichenfolge in ein datetime-Objekt analysieren

Verwendet C - Standard - Format - Codes .

```
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)
```

Ausgeben des datetime-Objekts in eine Zeichenfolge

Verwendet C - Standard - Format - Codes .

```
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
```

Datumsformatierung online lesen: https://riptutorial.com/de/python/topic/7284/datumsformatierung

Kapitel 42: Debuggen

Examples

Der Python-Debugger: Schrittweises Debuggen mit _pdb_

Die Python Standard Library enthält eine interaktive Debugging-Bibliothek namens *pdb* . *pdb* verfügt über umfangreiche Funktionen, wobei die am häufigsten verwendete Möglichkeit ist, ein Programm schrittweise *durchzugehen* .

Um sofort mit dem schrittweisen Debugging zu beginnen, verwenden Sie Folgendes:

```
python -m pdb <my_file.py>
```

Dadurch wird der Debugger in der ersten Zeile des Programms gestartet.

Normalerweise möchten Sie einen bestimmten Abschnitt des Codes für das Debugging verwenden. Dazu importieren wir die pdb-Bibliothek und *unterbrechen* mit *set_trace ()* den Fluss dieses *problematischen* Beispielcodes.

```
import pdb

def divide(a, b):
    pdb.set_trace()
    return a/b
    # What's wrong with this? Hint: 2 != 3

print divide(1, 2)
```

Wenn Sie dieses Programm ausführen, wird der interaktive Debugger gestartet.

```
python foo.py
> ~/scratch/foo.py(5)divide()
-> return a/b
(Pdb)
```

Häufig wird dieser Befehl in einer Zeile verwendet, sodass er mit einem einzelnen # -Zeichen auskommentiert werden kann

```
 import pdf; pdb.set_trace()
```

Bei der Eingabeaufforderung *(Pdb)* können Befehle eingegeben werden. Diese Befehle können Debugger-Befehle oder Python sein. Um Variablen zu drucken, können wir *p* aus dem Debugger oder den Python- *Druck verwenden* .

```
(Pdb) p a
1
(Pdb) print a
```

```
1
```

Um eine Liste aller lokalen Variablen anzuzeigen, verwenden Sie

```
locals
```

eingebaute Funktion

Dies sind gute Debugger-Befehle, die Sie kennen sollten:

```
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)
```

Der Debugger kann Python auch interaktiv auswerten:

```
-> 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]
```

Hinweis:

Wenn einer Ihrer Variablennamen mit den Debugger-Befehlen übereinstimmt, verwenden Sie ein Ausrufezeichen ' **!** 'vor dem var, um explizit auf die Variable und nicht auf den Debuggerbefehl zu verweisen. Beispielsweise kann es oft vorkommen, dass Sie den Variablennamen ' **c** ' für einen Zähler verwenden und Sie ihn möglicherweise im Debugger drucken möchten. Ein einfacher ' **c'** -Befehl würde die Ausführung bis zum nächsten Haltepunkt fortsetzen. Verwenden Sie stattdessen ' **! C** ', um den Wert der Variablen wie folgt zu drucken:

```
(Pdb) !c
```

```
4
```

Über IPython und ipdb

Wenn IPython (oder Jupyter) installiert ist, kann der Debugger folgendermaßen aufgerufen werden:

```
import ipdb
ipdb.set_trace()
```

Bei Erreichen wird der Code beendet und gedruckt:

```
 /home/usr/ook.py(3)<module>()
      1 import ipdb
      2 ipdb.set_trace()
----> 3 print("Hello world!")

ipdb>
```

Das bedeutet natürlich, dass man den Code bearbeiten muss. Es gibt einen einfacheren Weg:

```
from IPython.core import ultratb
sys.excepthook = ultratb.FormattedTB(mode='Verbose',
                                     color_scheme='Linux',
                                     call_pdb=1)
```

Dies bewirkt, dass der Debugger aufgerufen wird, wenn eine nicht erfasste Ausnahme ausgelöst wird.

Remote-Debugger

`rpdb` müssen Sie Python-Code debuggen, der von einem anderen Prozess ausgeführt wird. In diesem Fall ist `rpdb` .

> rpdb ist ein Wrapper um pdb, der stdin und stdout zu einem Socket-Handler umleitet. Standardmäßig wird der Debugger an Port 4444 geöffnet

Verwendungszweck:

```
# In the Python file you want to debug.
import rpdb
rpdb.set_trace()
```

Und dann müssen Sie dies im Terminal ausführen, um eine Verbindung zu diesem Prozess herzustellen.

```
# Call in a terminal to see the output
$ nc 127.0.0.1 4444
```

Und du wirst pdb Promt bekommen

```
> /home/usr/ook.py(3)<module>()
-> print("Hello world!")
(Pdb)
```

Debuggen online lesen: https://riptutorial.com/de/python/topic/2077/debuggen

Kapitel 43: Dekorateure

Einführung

Decorator-Funktionen sind Software-Designmuster. Sie ändern die Funktionalität einer Funktion, Methode oder Klasse dynamisch, ohne dass Sie Unterklassen direkt verwenden oder den Quellcode der dekorierten Funktion ändern müssen. Bei richtiger Anwendung können Dekorateure zu mächtigen Werkzeugen im Entwicklungsprozess werden. Dieses Thema behandelt die Implementierung und Anwendung von Decorator-Funktionen in Python.

Syntax

- def decorator_function (f): pass # definiert einen Dekorateur mit dem Namen decorator_function
- @decorator_function
  def decor_function (): pass # Die Funktion ist jetzt mit decorator_function umschlossen (dekoriert mit)
- `@decorator_function` = decorator_function (`@decorator_function`) # Dies entspricht der Verwendung der syntaktischen Zucker `@decorator_function`

Parameter

Parameter	Einzelheiten
f	Die zu dekorierende Funktion (verpackt)

Examples

Dekorateurfunktion

Dekorateure verbessern das Verhalten anderer Funktionen oder Methoden. Jede Funktion, die eine Funktion als Parameter übernimmt und eine erweiterte Funktion zurückgibt, kann als **Dekorator verwendet werden** .

```
# 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.")
```

Die `@` -Notation ist ein syntaktischer Zucker, der dem folgenden entspricht:

```
my_function = super_secret_function(my_function)
```

Es ist wichtig, dies zu beachten, um zu verstehen, wie die Dekorateure arbeiten. Diese "ungezugte" Syntax macht deutlich, warum die Decorator-Funktion eine Funktion als Argument verwendet und warum sie eine andere Funktion zurückgeben soll. Es zeigt auch, was passiert, wenn Sie *keine* Funktion zurückgeben:

```
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
```

Daher definieren wir normalerweise eine *neue Funktion* im Dekorateur und geben sie zurück. Diese neue Funktion würde zuerst etwas tun, was sie tun muss, dann die ursprüngliche Funktion aufrufen und schließlich den Rückgabewert verarbeiten. Betrachten Sie diese einfache Dekorationsfunktion, die die Argumente druckt, die die ursprüngliche Funktion empfängt, und sie dann aufruft.

```
#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.
```

Dekorateur Klasse

Wie in der Einleitung erwähnt, ist ein Dekorateur eine Funktion, die auf eine andere Funktion angewendet werden kann, um ihr Verhalten zu verbessern. Der syntaktische Zucker entspricht dem Folgenden: `my_func = decorator(my_func)` . Aber was wäre, wenn der `decorator` stattdessen eine Klasse wäre? Die Syntax würde weiterhin funktionieren, mit der Ausnahme, dass `my_func` jetzt

durch eine Instanz der `decorator` Klasse ersetzt wird. Wenn diese Klasse die magische Methode `__call__()` implementiert, ist es weiterhin möglich, `my_func` als Funktion zu verwenden:

```
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.
```

Beachten Sie, dass eine mit einem Klassendekorateur dekorierte Funktion aus Sicht der Typenprüfung nicht mehr als "Funktion" betrachtet wird:

```
import types
isinstance(testfunc, types.FunctionType)
# False
type(testfunc)
# <class '__main__.Decorator'>
```

Dekorationsmethoden

Für Dekorationsmethoden müssen Sie eine zusätzliche `__get__` Methode definieren:

```
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()
```

Im Dekorateur.

Warnung!

Klassendekorateure erzeugen nur eine Instanz für eine bestimmte Funktion, so dass das Dekorieren einer Methode mit einem Klassendekorateur denselben Dekorator für alle Instanzen dieser Klasse verwendet:

```
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
```

Einen Dekorateur wie eine dekorierte Funktion aussehen lassen

Dekorateure entfernen normalerweise Funktionsmetadaten, da sie nicht gleich sind. Dies kann zu Problemen führen, wenn Meta-Programme für den dynamischen Zugriff auf Funktionsmetadaten verwendet werden. Metadaten enthalten auch die Dokumentzeichenfolgen und den Namen der Funktion. `functools.wraps` lässt die dekorierte Funktion wie die ursprüngliche Funktion aussehen, indem sie mehrere Attribute in die Wrapper-Funktion kopiert.

```
from functools import wraps
```

Die beiden Methoden zum Umwickeln eines Dekorateurs erreichen beim Verbergen der ursprünglichen Funktion dasselbe. Es gibt keinen Grund, die Funktionsversion der Klassenversion

vorzuziehen, es sei denn, Sie verwenden bereits eine Version über der anderen.

Als eine Funktion

```
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__
```

'Prüfung'

Als eine Klasse

```
class Decorator(object):
    def __init__(self, func):
        # Copies name, module, annotations and docstring to the instance.
        self._wrapped = wraps(func)(self)

    def __call__(self, *args, **kwargs):
        return self._wrapped(*args, **kwargs)

@Decorator
def test():
    """Docstring of test."""
    pass

test.__doc__
```

"Docstring des Tests."

Dekorateur mit Argumenten (Dekorateurfabrik)

Ein Dekorateur hat nur ein Argument: die zu dekorierende Funktion. Es gibt keine Möglichkeit, andere Argumente zu übergeben.

Zusätzliche Argumente sind jedoch oft erwünscht. Der Trick besteht darin, eine Funktion zu erstellen, die willkürliche Argumente verwendet und einen Dekorator zurückgibt.

Dekorateurfunktionen

```
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()
```

Der Dekorateur möchte Ihnen sagen: Hallo Welt

Wichtige Notiz:

Bei solchen Dekorateurfabriken **müssen** Sie den Dekorateur mit einem Paar Klammern anrufen:

```
@decoratorfactory # Without parentheses
def test():
    pass

test()
```

TypeError: decorator () fehlt 1 erforderliches Positionsargument: 'func'

Dekorateur Klassen

```
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()
```

Im Dekorateur mit Argumenten (10,)

Erstellen Sie eine Einzelklasse mit einem Dekorateur

Ein Singleton ist ein Muster, das die Instantiierung einer Klasse auf eine Instanz / ein Objekt beschränkt. Mit einem Dekorator können wir eine Klasse als Singleton definieren, indem die Klasse gezwungen wird, entweder eine vorhandene Instanz der Klasse zurückzugeben oder eine neue Instanz zu erstellen (falls diese nicht existiert).

```
def singleton(cls):
    instance = [None]
    def wrapper(*args, **kwargs):
        if instance[0] is None:
            instance[0] = cls(*args, **kwargs)
        return instance[0]

    return wrapper
```

Dieser Dekorator kann zu jeder Klassendeklaration hinzugefügt werden und stellt sicher, dass höchstens eine Instanz der Klasse erstellt wird. Alle nachfolgenden Aufrufe geben die bereits vorhandene Klasseninstanz zurück.

```
@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
```

Es ist also egal, ob Sie über Ihre lokale Variable auf die Klasseninstanz verweisen oder eine andere "Instanz" erstellen, Sie erhalten immer dasselbe Objekt.

Verwenden eines Dekorators, um eine Funktion festzulegen

```
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()
```

Dekorateure online lesen: https://riptutorial.com/de/python/topic/229/dekorateure

Kapitel 44: Deque-Modul

Syntax

- dq = deque () # Erzeugt ein leeres Deque
- dq = deque (iterable) # Erzeugt einen Deque mit einigen Elementen
- dq.append (object) # Fügt ein Objekt rechts von der Deque hinzu
- dq.appendleft (object) # Fügt ein Objekt links vom Deque hinzu
- dq.pop () -> object # Entfernt das Objekt ganz rechts und gibt es zurück
- dq.popleft () -> object # Entfernt das linke Objekt und gibt es zurück
- dq.extend (iterable) # Fügt einige Elemente rechts vom Deque hinzu
- dq.extendleft (iterable) # Fügt einige Elemente links vom Deque hinzu

Parameter

Parameter	Einzelheiten
`iterable`	Erzeugt den Deque mit ursprünglichen Elementen, die von einem anderen iterierbaren Element kopiert wurden.
`maxlen`	Begrenzt, wie groß der Deque sein kann, wobei alte Elemente als neue ausgegeben werden.

Bemerkungen

Diese Klasse ist nützlich, wenn Sie ein Objekt benötigen, das einer Liste ähnelt, die schnelle Anhänge- und Popoperationen von beiden Seiten zulässt (der Name `deque` steht für " *double-ended queue* ").

Die zur Verfügung gestellten Methoden sind in der Tat sehr ähnlich, mit der Ausnahme, dass einige wie `pop` , `append` oder `extend` mit `left` . Die `deque` Datenstruktur sollte einer Liste vorgezogen werden, wenn Elemente an beiden Enden häufig eingefügt und gelöscht werden müssen, da dies zu einer konstanten Zeit O (1) möglich ist.

Examples

Basic Deque mit

Die wichtigsten Methoden, die für diese Klasse nützlich sind, sind `popleft` und `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])
```

Begrenzung der Deque-Größe

Verwenden Sie den Parameter `maxlen` , während Sie eine Deque erstellen, um die Größe der Deque zu begrenzen:

```
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)
```

Verfügbare Methoden in deque

Leeren deque erstellen:

```
dl = deque()  # deque([]) creating empty deque
```

Erstellen von Deque mit einigen Elementen:

```
dl = deque([1, 2, 3, 4])  # deque([1, 2, 3, 4])
```

Element zu deque hinzufügen:

```
dl.append(5)  # deque([1, 2, 3, 4, 5])
```

Element links von deque hinzufügen:

```
dl.appendleft(0)  # deque([0, 1, 2, 3, 4, 5])
```

Liste der Elemente zu deque hinzufügen:

```
dl.extend([6, 7])  # deque([0, 1, 2, 3, 4, 5, 6, 7])
```

Hinzufügen einer Liste von Elementen von links:

```
dl.extendleft([-2, -1])  # deque([-1, -2, 0, 1, 2, 3, 4, 5, 6, 7])
```

Mit dem Element `.pop()` wird ein Element auf der rechten Seite entfernt:

```
dl.pop()  # 7 => deque([-1, -2, 0, 1, 2, 3, 4, 5, 6])
```

Verwenden `.popleft()` Elements `.popleft()` zum Entfernen eines Elements von der linken Seite:

```
dl.popleft()  # -1 deque([-2, 0, 1, 2, 3, 4, 5, 6])
```

Element nach seinem Wert entfernen:

```
dl.remove(1)  # deque([-2, 0, 2, 3, 4, 5, 6])
```

Kehren Sie die Reihenfolge der Elemente in deque um:

```
dl.reverse()  # deque([6, 5, 4, 3, 2, 0, -2])
```

Breite erste Suche

Der Deque ist die einzige Python-Datenstruktur mit schnellen Warteschlangenoperationen . (Hinweis `queue.Queue` ist normalerweise nicht geeignet, da sie für die Kommunikation zwischen Threads `queue.Queue` ist.) Ein grundlegender Anwendungsfall einer Queue ist die erste Suche in der Breite .

```
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
```

Angenommen, wir haben eine einfache gerichtete Grafik:

```
graph = {1:[2,3], 2:[4], 3:[4,5], 4:[3,5], 5:[]}
```

Wir können nun die Entfernungen von einer Startposition aus finden:

```
>>> bfs(graph, 1)
{1: 0, 2: 1, 3: 1, 4: 2, 5: 2}

>>> bfs(graph, 3)
{3: 0, 4: 1, 5: 1}
```

Deque-Modul online lesen: https://riptutorial.com/de/python/topic/1976/deque-modul

Kapitel 45: Der Dolmetscher (Befehlszeilenkonsole)

Examples

Allgemeine Hilfe anfordern

Wenn die `help` in der Konsole ohne Argumente aufgerufen wird, bietet Python eine interaktive Hilfskonsole, in der Sie Informationen zu Python-Modulen, Symbolen, Schlüsselwörtern usw. erhalten.

```
>>> 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".
```

Bezugnehmend auf den letzten Ausdruck

Verwenden Sie einen Unterstrich _ um den Wert des letzten Ergebnisses aus Ihrem letzten Ausdruck in der Konsole zu erhalten.

```
>>> 2 + 2
4
>>> _
4
>>> _ + 6
10
```

Dieser magische Unterstrichwert wird nur aktualisiert, wenn ein Python-Ausdruck verwendet wird, der einen Wert ergibt. Das Definieren von Funktionen oder für Schleifen ändert den Wert nicht. Wenn der Ausdruck eine Ausnahme auslöst, werden keine Änderungen an _ .

```
>>> "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'
```

Denken Sie daran, dass diese magische Variable nur im interaktiven Python-Interpreter verfügbar ist. Das Ausführen von Skripts führt dies nicht aus.

Öffnen der Python-Konsole

Die Konsole für die Hauptversion von Python kann normalerweise geöffnet werden, indem Sie in Ihre Windows-Konsole `py` oder auf anderen Plattformen `python` eingeben.

```
$ 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.
>>>
```

Wenn Sie über mehrere Versionen verfügen, werden ihre ausführbaren Dateien standardmäßig `python2` bzw. `python3` .

Dies hängt natürlich von den ausführbaren Python-Dateien in PATH ab.

Die PYTHONSTARTUP-Variable

Sie können eine Umgebungsvariable namens PYTHONSTARTUP für die Python-Konsole festlegen. Wenn Sie die Python-Konsole aufrufen, wird diese Datei ausgeführt, sodass Sie der Konsole zusätzliche Funktionen hinzufügen können, z.

Wenn die PYTHONSTARTUP-Variable auf den Speicherort einer Datei gesetzt wurde, die Folgendes enthält:

```
print("Welcome!")
```

Das Öffnen der Python-Konsole würde dann zu dieser zusätzlichen Ausgabe führen:

```
$ 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!
>>>
```

Kommandozeilenargumente

Python verfügt über verschiedene Befehlszeilenschalter, die an `py` . Diese können durch Ausführen von `py --help` , die diese Ausgabe auf Python 3.4 liefert:

```
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.
```

Hilfe zu einem Objekt erhalten

Die Python-Konsole fügt eine neue Funktion hinzu, `help` , mit der Informationen zu einer Funktion oder einem Objekt abgerufen werden können.

Bei einer Funktion druckt `help` seine Signatur (Argumente) und seinen Docstring, falls die Funktion eine hat.

```
>>> 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.
```

Für ein Objekt listet `help` die Dokumentzeichenfolge des Objekts und die verschiedenen Elementfunktionen auf, die das Objekt hat.

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

Der Dolmetscher (Befehlszeilenkonsole) online lesen:
https://riptutorial.com/de/python/topic/2473/der-dolmetscher--befehlszeilenkonsole-

Kapitel 46: Designmuster

Einführung

Ein Entwurfsmuster ist eine allgemeine Lösung für ein häufig auftretendes Problem in der Softwareentwicklung. In diesem Dokumentationsthema werden Beispiele für gängige Entwurfsmuster in Python beschrieben.

Examples

Strategiemuster

Dieses Entwurfsmuster wird Strategiemuster genannt. Es wird verwendet, um eine Familie von Algorithmen zu definieren, jeden zu kapseln und austauschbar zu machen. Das Strategieentwurfsmuster lässt einen Algorithmus unabhängig von den Kunden variieren, die ihn verwenden.

Zum Beispiel können Tiere auf viele verschiedene Arten "laufen". Gehen kann als eine Strategie betrachtet werden, die von verschiedenen Tierarten umgesetzt wird:

```
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))
```

Das Ausführen dieses Beispiels würde die folgende Ausgabe erzeugen:

```
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
```

Beachten Sie, dass in Sprachen wie C ++ oder Java dieses Muster mithilfe einer abstrakten Klasse oder einer Schnittstelle implementiert wird, um eine Strategie zu definieren. In Python ist es sinnvoller, nur einige Funktionen extern zu definieren, die mithilfe von `types.MethodType` dynamisch zu einer Klasse `types.MethodType` .

Einführung in Design Patterns und Singleton Pattern

Design Patterns bieten Lösungen für `commonly occurring problems` beim Softwaredesign. Die Entwurfsmuster wurden zuerst von `GoF(Gang of Four)` in dem sie die häufig vorkommenden Muster als Probleme beschrieben, die immer wieder auftreten, und Lösungen für diese Probleme.

Designmuster haben vier wesentliche Elemente:

1. `The pattern name` ist ein Handle, mit dem wir ein Designproblem, seine Lösungen und Konsequenzen in ein oder zwei Worten beschreiben können.
2. `The problem` beschreibt, wann das Muster angewendet wird.
3. `The solution` beschreibt die Elemente, aus denen das Design besteht, sowie deren Beziehungen, Verantwortlichkeiten und Kollaborationen.
4. `The consequences` sind die Ergebnisse und Kompromisse bei der Anwendung des Musters.

Vorteile von Design Patterns:

1. Sie können in mehreren Projekten wiederverwendet werden.
2. Die architektonische Ebene der Probleme kann gelöst werden
3. Sie sind langjährig erprobt und bewährt. Dies ist die Erfahrung von Entwicklern und

Architekten
4. Sie haben Zuverlässigkeit und Abhängigkeit

Entwurfsmuster können in drei Kategorien unterteilt werden:

1. Kreationelles Muster
2. Strukturelles Muster
3. Verhaltensmuster

`Creational Pattern` - Sie befassen sich mit der Erstellung des Objekts und isolieren die Details der Objekterstellung.

`Structural Pattern` - Sie gestalten die Struktur von Klassen und Objekten so, dass sie komponieren können, um größere Ergebnisse zu erzielen.

`Behavioral Pattern` - Sie befassen sich mit der Interaktion zwischen Objekten und der Verantwortung von Objekten.

Singleton-Muster :

Es handelt sich um eine Art `creational pattern` , das einen Mechanismus bereitstellt, um nur ein und ein Objekt eines bestimmten Typs zu haben und einen globalen Zugangspunkt bereitzustellen.

Zum Beispiel kann Singleton in Datenbankoperationen verwendet werden, bei denen Datenbankobjekte die Datenkonsistenz beibehalten sollen.

Implementierung

Wir können Singleton Pattern in Python implementieren, indem Sie nur eine Instanz der Singleton-Klasse erstellen und dasselbe Objekt erneut bedienen.

```
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)
```

Ausgabe:

```
('Object created', <__main__.Singleton object at 0x10a7cc310>)
('Object2 created', <__main__.Singleton object at 0x10a7cc310>)
```

Beachten Sie, dass dieses Muster in Sprachen wie C ++ oder Java implementiert wird, indem Sie den Konstruktor privat machen und eine statische Methode erstellen, die die Objektinitialisierung

durchführt. Auf diese Weise wird beim ersten Aufruf ein Objekt erstellt, und die Klasse gibt danach dasselbe Objekt zurück. In Python haben wir jedoch keine Möglichkeit, private Konstruktoren zu erstellen.

Fabrikmuster

Das Fabrikmuster ist auch ein `Creational pattern` . Der Begriff `factory` bedeutet, dass eine Klasse für das Erstellen von Objekten anderer Typen verantwortlich ist. Es gibt eine Klasse, die als Factory fungiert und mit Objekten und Methoden verknüpft ist. Der Client erstellt ein Objekt durch Aufrufen der Methoden mit bestimmten Parametern. Factory erstellt das Objekt des gewünschten Typs und gibt es an den Client zurück.

```
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)
```

Ausgabe:

```
Which music you want to play Mp3 or Ogg"Ogg"
Playing .ogg music!
```

`MusicFactory` ist hier die Factory-Klasse, die je nach Wahl des Benutzers entweder ein Objekt vom Typ `Mp3` oder `Ogg` .

Proxy

Proxy-Objekte werden häufig verwendet, um den geschützten Zugriff auf ein anderes Objekt sicherzustellen. Diese interne Geschäftslogik möchten wir nicht mit Sicherheitsanforderungen belasten.

Angenommen, wir möchten garantieren, dass nur Benutzer bestimmter Berechtigungen auf die Ressource zugreifen können.

Proxy-Definition: (Es wird sichergestellt, dass nur Benutzer, die tatsächlich Reservierungen sehen können, denesen reservation_service verwenden können.)

```
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)
```

LEISTUNGEN

- **wir vermeiden Änderungen in `ReservationService` wenn Zugriffsbeschränkungen geändert werden**
- **Wir mischen keine geschäftsbezogenen Daten (`date_from` , `date_to` , `reservations_count`) mit `date_to` Konzepten (Benutzerberechtigungen) im Service.**
- **Consumer (`StatsService`) ist ebenfalls frei von Logik für Berechtigungen**

CAVEATS

- Die Proxy-Schnittstelle entspricht immer genau dem Objekt, das sie verbirgt, so dass der Benutzer, der den von Proxy umschlossenen Dienst in Anspruch nimmt, nicht einmal die Anwesenheit eines Proxy erkannt hat.

Designmuster online lesen: https://riptutorial.com/de/python/topic/8056/designmuster

Kapitel 47: Deskriptor

Examples

Einfacher Deskriptor

Es gibt zwei verschiedene Arten von Deskriptoren. `__get__()` werden als Objekte definiert, die sowohl eine `__get__()` als auch eine `__set__()` -Methode definieren, wohingegen Nicht-Deskriptoren nur eine `__get__()` -Methode definieren. Diese Unterscheidung ist wichtig, wenn Sie Überschreibungen und den Namensraum des Wörterbuchs einer Instanz in Betracht ziehen. Wenn ein Datendeskriptor und ein Eintrag im Wörterbuch einer Instanz denselben Namen haben, hat der Datendeskriptor Vorrang. Wenn jedoch ein Nicht-Daten-Deskriptor und ein Eintrag im Wörterbuch einer Instanz denselben Namen haben, hat der Eintrag des Instanzwörterbuchs Vorrang.

Um einen schreibgeschützten Datendeskriptor zu erstellen, definieren Sie sowohl **get** () als auch **set** (), wobei **set** () beim Aufruf einen AttributeError auslöst. Das Definieren der **set** () - Methode mit einem Platzhalter für die Ausnahmebereitstellung reicht aus, um sie zu einem Datendeskriptor zu machen.

```
descr.__get__(self, obj, type=None) --> value
descr.__set__(self, obj, value) --> None
descr.__delete__(self, obj) --> None
```

Ein umgesetztes Beispiel:

```
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
```

Zwei-Wege-Konvertierungen

Beschreibungsobjekte können dazu führen, dass verwandte Objektattribute automatisch auf Änderungen reagieren.

Angenommen, wir möchten einen Oszillator mit einer bestimmten Frequenz (in Hertz) und einer Periode (in Sekunden) modellieren. Wenn wir die Frequenz aktualisieren, möchten wir, dass die Periode aktualisiert wird, und wenn wir die Periode aktualisieren, möchten wir, dass die Frequenz aktualisiert wird:

```
 >>> 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
```

Wir wählen einen der Werte (Frequenz in Hertz) als "Anker", dh den Wert, der ohne Konvertierung eingestellt werden kann, und schreiben eine Deskriptorklasse dafür:

```
class Hertz(object):
    def __get__(self, instance, owner):
        return self.value

    def __set__(self, instance, value):
        self.value = float(value)
```

Der "andere" Wert (Zeitraum in Sekunden) wird in Bezug auf den Anker definiert. Wir schreiben eine Deskriptorklasse, die unsere Konvertierungen durchführt:

```
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)
```

Jetzt können wir die Oszillator-Klasse schreiben:

```
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
```

Deskriptor online lesen: https://riptutorial.com/de/python/topic/3405/deskriptor

Kapitel 48: Die Druckfunktion

Examples

Grundlagen zum Drucken

In Python 3 und höher ist `print` eher eine Funktion als ein Schlüsselwort.

```
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
```

Sie können auch eine Reihe von Parametern zum `print` :

```
print(foo, bar, baz)
# out: 1 bar 3.14
```

Eine andere Möglichkeit zum `print` mehrerer Parameter besteht in der Verwendung von `+`

```
print(str(foo) + " " + bar + " " + str(baz))
# out: 1 bar 3.14
```

Bei der Verwendung von `+` zum Drucken mehrerer Parameter sollten Sie jedoch aufpassen, dass der Typ der Parameter identisch ist. Wenn Sie versuchen, das obige Beispiel ohne die Umwandlung in eine `string` zuerst zu drucken, würde dies zu einem Fehler führen, da versucht würde, der Zeichenfolge `"bar"` die Zahl `1` und die Zahl `3.14` hinzuzufügen.

```
# Wrong:
# type:int  str  float
print(foo + bar + baz)
# will result in an error
```

Dies liegt daran, dass der Inhalt des `print` zuerst ausgewertet wird:

```
print(4 + 5)
# out: 9
print("4" + "5")
# out: 45
print([4] + [5])
# out: [4, 5]
```

Andernfalls kann die Verwendung von `+` für einen Benutzer sehr hilfreich sein, um die Ausgabe von Variablen zu lesen. Im folgenden Beispiel ist die Ausgabe sehr einfach zu lesen!

Das folgende Skript veranschaulicht dies

```
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))
```

Sie können verhindern, dass die `print` automatisch einen Zeilenvorschub druckt, indem Sie den Parameter `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?
```

Wenn Sie in eine Datei schreiben möchten, können Sie es als Parameter übergeben `file` :

```
with open('my_file.txt', 'w+') as my_file:
    print("this goes to the file!", file=my_file)
```

das geht in die Datei!

Parameter drucken

Sie können mehr als nur Text drucken. `print` hat auch einige Parameter, die Ihnen helfen.

Argument `sep` : Platziert eine Zeichenfolge zwischen den Argumenten.

Müssen Sie eine Liste von Wörtern drucken, die durch ein Komma oder eine andere Zeichenfolge getrennt sind?

```
>>> print('apples','bannas', 'cherries', sep=', ')
apple, bannas, cherries
>>> print('apple','banna', 'cherries', sep=', ')
apple, banna, cherries
>>>
```

Argument `end` : Verwenden Sie am Ende etwas anderes als eine neue Zeile

Ohne das `end` Argument schreiben alle `print()` Funktionen eine Zeile und gehen dann zum Anfang der nächsten Zeile. Sie können es ändern, um nichts zu tun (verwenden Sie eine leere Zeichenfolge von '') oder doppelten Abstand zwischen Absätzen, indem Sie zwei Zeilenumbrüche verwenden.

```
>>> print("<a", end=''); print(" class='jidn'" if 1 else "", end=''); print("/>")
<a class='jidn'/>
>>> print("paragraph1", end="\n\n"); print("paragraph2")
paragraph1
```

```
paragraph2
>>>
```

Argument `file` : Ausgabe irgendwo anders als sys.stdout senden.

Jetzt können Sie Ihren Text entweder an stdout, an eine Datei oder an StringIO senden, ohne sich darum zu kümmern, was Ihnen gegeben wird. Wenn es wie eine Datei quittiert, funktioniert es wie eine Datei.

```
>>> 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

>>>
```

Es gibt einen vierten Parameter `flush` der den Stream zwangsweise löscht.

Die Druckfunktion online lesen: https://riptutorial.com/de/python/topic/1360/die-druckfunktion

Kapitel 49: Die Pass-Anweisung

Syntax

- bestehen

Bemerkungen

Warum sollten Sie dem Dolmetscher sagen, dass er nichts ausdrücklich tun soll? Python hat die syntaktische Anforderung, dass Codeblöcke (nachdem `if` , `except` `def` , `class` usw.) nicht leer sein dürfen.

Aber manchmal ist ein leerer Codeblock an sich nützlich. Ein leerer `class` kann eine neue, andere Klasse definieren, z. B. eine Ausnahme, die abgefangen werden kann. Ein leerer `except` kann der einfachste Weg sein, um "Bitte um Verzeihung später" auszudrücken, wenn es nichts gibt, um das man um Vergebung bitten kann. Wenn ein Iterator alle schweren Bewegungen ausführt, kann eine leere `for` Schleife zum Ausführen des Iterators hilfreich sein.

Wenn also in einem Codeblock nichts passieren soll, ist ein `pass` erforderlich, damit ein solcher Block keinen `IndentationError` . Alternativ kann jede Aussage (einschließlich nur eines zu bewertenden Begriffs, wie das `Ellipsis` Literal `...` oder eine Zeichenfolge, meistens ein Docstring) verwendet werden, aber der `pass` macht deutlich, dass tatsächlich nichts passieren soll und nicht benötigt wird tatsächlich ausgewertet und (zumindest temporär) im Speicher abgelegt werden. Hier ist eine kleine kommentierte Sammlung der häufigsten Verwendungsmöglichkeiten von `pass` , die mir begegnet sind - zusammen mit einigen Kommentaren zur guten und schlechten Praxis.

- Ignorieren (alle oder) eine bestimmte Art von `Exception` (Beispiel aus `xml`):

  ```
  try:
      self.version = "Expat %d.%d.%d" % expat.version_info
  except AttributeError:
      pass # unknown
  ```

  Hinweis: Ignorieren aller Arten von Erhöhungen, wie im folgenden Beispiel von `pandas` , ist in der Regel schlechte Praxis betrachtet, weil es auch Ausnahmen abfängt , die wahrscheinlich auf den Aufrufer übergeben werden soll, zB `KeyboardInterrupt` oder `SystemExit` (oder sogar `HardwareIsOnFireError` - Wie wissen Sie , Sie sind nicht auf einer benutzerdefinierten Box mit spezifischen Fehlern ausgeführt, über die einige aufrufende Anwendungen informiert werden möchten.

  ```
  try:
      os.unlink(filename_larry)
  except:
      pass
  ```

  Verwenden Sie stattdessen mindestens `except Error:` oder in diesem Fall vorzugsweise

`except OSError:` ist eine weitaus bessere Praxis. Eine schnelle Analyse aller von mir installierten Python-Module ergab, dass mehr als 10% aller `except ...: pass` Anweisungen alle Ausnahmen abfangen, sodass es bei der Python-Programmierung immer noch ein häufiges Muster ist.

- Ableiten einer Ausnahmeklasse, die kein neues Verhalten hinzufügt (z. B. in `scipy`):

```
class CompileError(Exception):
    pass
```

  In ähnlicher Weise haben Klassen, die als abstrakte Basisklasse gedacht sind, häufig ein explizit leeres `__init__` oder andere Methoden, die von Unterklassen abgeleitet werden sollen. (zB `pebl`)

```
class _BaseSubmittingController(_BaseController):
    def submit(self, tasks): pass
    def retrieve(self, deferred_results): pass
```

- Das Testen dieses Codes wird für einige Testwerte ordnungsgemäß ausgeführt, ohne sich um die Ergebnisse zu kümmern (von `mpmath`):

```
for x, error in MDNewton(mp, f, (1,-2), verbose=0,
                         norm=lambda x: norm(x, inf)):
    pass
```

- In Klassen- oder Funktionsdefinitionen ist häufig bereits ein Docstring als *obligatorische Anweisung vorhanden* , die als einzige Funktion im Block ausgeführt werden muss. In solchen Fällen kann der Block *zusätzlich* zum Docstring `pass` enthalten `pass` um zu sagen: "Dies soll tatsächlich nichts `pebl` ". Zum Beispiel in `pebl` :

```
class ParsingError(Exception):
    """Error encountered while parsing an ill-formed datafile."""
    pass
```

- In einigen Fällen wird " `pass` als Platzhalter verwendet, um zu sagen "Diese Methode / Klasse / Wenn-Block / ... wurde noch nicht implementiert, aber dies ist der `Ellipsis` Ort dafür", obwohl ich persönlich das `Ellipsis` Literal vorziehen `Ellipsis` `...` (HINWEIS: nur Python-3), um genau zwischen diesem und dem absichtlichen "no-op" im vorherigen Beispiel zu unterscheiden. Wenn ich zum Beispiel ein Modell mit großen Strichen schreibe, schreibe ich vielleicht

```
def update_agent(agent):
    ...
```

  wo andere haben könnten

```
def update_agent(agent):
    pass
```

Vor

```
def time_step(agents):
    for agent in agents:
        update_agent(agent)
```

Als Erinnerung daran, die Funktion `update_agent` zu einem späteren Zeitpunkt auszufüllen, führen Sie jedoch bereits einige Tests durch, um `update_agent` , ob sich der Rest des Codes wie beabsichtigt verhält. (Eine dritte Option für diesen Fall ist die `raise NotImplementedError` . Dies ist insbesondere in zwei Fällen hilfreich: Entweder *„Diese abstrakte Methode sollte von jeder Unterklasse implementiert werden, es gibt keine generische Möglichkeit, sie in dieser Basisklasse zu definieren“*) oder *„Diese Funktion , mit diesem Namen ist in dieser Version noch nicht implementiert, aber so wird die Signatur aussehen "*)

Examples

Ausnahme ignorieren

```
try:
    metadata = metadata['properties']
except KeyError:
    pass
```

Erstellen Sie eine neue Ausnahme, die abgefangen werden kann

```
class CompileError(Exception):
    pass
```

Die Pass-Anweisung online lesen: https://riptutorial.com/de/python/topic/6891/die-pass-anweisung

Kapitel 50: Die spezielle Variable __name__

Einführung

Die spezielle Variable `__name__` wird verwendet, um zu prüfen, ob eine Datei als Modul importiert wurde oder nicht, und um eine Funktion, eine Klasse oder ein `__name__` anhand ihres Attributs `__name__` zu identifizieren.

Bemerkungen

Die Python-Sondervariable `__name__` wird auf den Namen des `__name__` Moduls gesetzt. Auf der obersten Ebene (z. B. im interaktiven Interpreter oder in der Hauptdatei) wird `'__main__'` . Dies kann verwendet werden, um einen Anweisungsblock auszuführen, wenn ein Modul direkt ausgeführt und nicht importiert wird.

Das zugehörige spezielle Attribut `obj.__name__` wird in Klassen, importierten Modulen und Funktionen *(einschließlich Methoden) gefunden* und gibt bei der Definition den Namen des Objekts an.

Examples

__name__ == '__main__'

Die spezielle Variable `__name__` wird vom Benutzer nicht festgelegt. Es wird meistens verwendet, um zu prüfen, ob das Modul von alleine ausgeführt wird oder nicht, weil ein `import` durchgeführt wurde. Um zu vermeiden, dass Ihr Modul beim Importieren bestimmte Teile des Codes `if __name__ == '__main__'` , prüfen Sie, `if __name__ == '__main__'` .

Lassen Sie **module_1.py** nur eine Zeile lang sein:

```
import module2.py
```

Und mal sehen was passiert, abhängig von **module2.py**

Situation 1

module2.py

```
print('hello')
```

Wenn Sie module1.py ausführen, wird `hello` gedruckt
Wird module2.py ausgeführt, wird `hello` gedruckt

Situation 2

module2.py

```
if __name__ == '__main__':
    print('hello')
```

Wenn Sie module1.py ausführen, wird nichts gedruckt
Wird module2.py ausgeführt, wird `hello` gedruckt

Funktionsklasse_oder_Modul .__ Name__

Das spezielle Attribut `__name__` einer Funktion, Klasse oder eines Moduls ist eine Zeichenfolge, die ihren Namen enthält.

```
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
```

Das `__name__` Attribut ist jedoch nicht der Name der Variablen, die auf die Klasse, Methode oder Funktion verweist, sondern der Name, den sie bei der Definition erhalten hat.

```
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
```

Dies kann unter anderem zum Debuggen verwendet werden:

```
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
```

Zur Protokollierung verwenden

Beim Konfigurieren der integrierten `logging` wird häufig ein Protokollierer mit dem Namen `__name__` des aktuellen Moduls erstellt:

```
logger = logging.getLogger(__name__)
```

Dies bedeutet, dass der vollständig qualifizierte Name des Moduls in den Protokollen angezeigt wird, sodass Sie leichter erkennen können, woher die Nachrichten stammen.

Die spezielle Variable __name__ online lesen: https://riptutorial.com/de/python/topic/1223/die-spezielle-variable---name--

Kapitel 51: Django

Einführung

Django ist ein Python-Web-Framework auf hohem Niveau, das eine schnelle Entwicklung und ein sauberes, pragmatisches Design fördert. Entwickelt von erfahrenen Entwicklern, wird der Aufwand für die Webentwicklung groß geschrieben, sodass Sie sich auf das Schreiben Ihrer App konzentrieren können, ohne das Rad neu zu erfinden. Es ist kostenlos und Open Source.

Examples

Hallo Welt mit Django

Erstellen Sie ein einfaches `Hello World` Beispiel mit Ihrem Django.

Stellen Sie sicher, dass Sie zuerst Django auf Ihrem PC installiert haben.

Öffne ein Terminal und tippe: python -c "import django"
-> Wenn kein Fehler auftritt, ist Django bereits installiert.

Jetzt können wir ein Projekt in Django erstellen. Dafür schreiben Sie unten auf das Terminal:
django-admin startprojekt halloWelt

Der obige Befehl erstellt ein Verzeichnis mit dem Namen HelloWorld.
Verzeichnisstruktur wird wie folgt aussehen:
Hallo Welt
| --helloworld
| **init** .py
| | --settings.py
| | --urls.py
| | --wsgi.py
| --manage.py

Ansichten schreiben (Referenz aus der Django-Dokumentation)

Eine Ansichtsfunktion oder kurz Ansicht ist einfach eine Python-Funktion, die eine Webanforderung annimmt und eine Webantwort zurückgibt. Diese Antwort kann der HTML-Inhalt einer Webseite sein oder irgendetwas. Die Dokumentation besagt, dass wir Ansichten schreiben können, wo auch immer, aber besser in views.py in unserem Projektverzeichnis geschrieben wird.

Hier ist eine Ansicht, die eine Hallo-Weltnachricht zurückgibt. (Views.py)

```
from django.http import HttpResponse

define helloWorld(request):
    return HttpResponse("Hello World!! Django Welcomes You.")
```

lass uns den Code Schritt für Schritt verstehen.

- Zuerst importieren wir die Klasse HttpResponse aus dem Modul django.http.
- Als Nächstes definieren wir eine Funktion namens helloWorld. Dies ist die Ansichtsfunktion. Jede Ansichtsfunktion nimmt ein HttpRequest-Objekt als ersten Parameter an, der normalerweise als request bezeichnet wird.

  Beachten Sie, dass der Name der Ansichtsfunktion keine Rolle spielt. Es muss nicht auf eine bestimmte Weise benannt werden, damit Django es erkennen kann. Wir haben es halloWorld hier genannt, so dass klar sein wird, was es tut.
- Die Sicht gibt ein HttpResponse-Objekt zurück, das die generierte Antwort enthält. Jede Ansichtsfunktion ist dafür verantwortlich, ein HttpResponse-Objekt zurückzugeben.

Für weitere Informationen zu Django-Ansichten klicken Sie hier

Zuordnung von URLs zu Ansichten
Um diese Ansicht unter einer bestimmten URL anzuzeigen, müssen Sie eine URLconf erstellen.

Vorher sollten wir verstehen, wie Django Anfragen bearbeitet.

- Django bestimmt das zu verwendende Wurzel-URLconf-Modul.
- Django lädt dieses Python-Modul und sucht nach den variablen URL-Mustern. Dies sollte eine Python-Liste von django.conf.urls.url () -Instanzen sein.
- Django durchläuft die einzelnen URL-Muster der Reihe nach und stoppt beim ersten, das der angeforderten URL entspricht.
- Sobald eine der Regexen übereinstimmt, importiert Django die angegebene Ansicht, eine einfache Python-Funktion.

So sehen unsere URLconf gleich aus:

```
from django.conf.urls import url
from . import views #import the views.py from current directory

urlpatterns = [
   url(r'^helloworld/$', views.helloWorld),
]
```

Für weitere Informationen zu Django Urls klicken Sie hier

Wechseln Sie nun in das Verzeichnis HelloWorld und schreiben Sie unten auf den Befehl Terminal.
python manage.py runserver

Standardmäßig wird der Server bei 127.0.0.1:8000 ausgeführt

Öffnen Sie Ihren Browser und geben Sie 127.0.0.1:8000/helloworld/ ein. Die Seite zeigt Ihnen "Hallo Welt !! Django heißt Sie willkommen."

Django online lesen: https://riptutorial.com/de/python/topic/8994/django

Kapitel 52: Dynamische Code-Ausführung mit "exec" und "eval"

Syntax

- eval (Ausdruck [, Globals = Keine [, Einheimische = Keine]]
- exec (Objekt)
- exec (Objekt, Globals)
- exec (Objekt, Globals, Einheimische)

Parameter

Streit	Einzelheiten
`expression`	Der Ausdruck Code als String oder ein `code` - Objekt
`object`	Die Anweisung Code als String oder ein `code` - Objekt
`globals`	Das für globale Variablen zu verwendende Wörterbuch. Wenn für Einheimische keine Angabe gemacht wird, wird dies auch für Einheimische verwendet. Wenn nicht angegeben, werden die `globals()` des aufrufenden Bereichs verwendet.
`locals`	Ein *Mapping*- Objekt, das für lokale Variablen verwendet wird. Wenn nicht angegeben, wird stattdessen die für `globals` verwendet. Wenn beide weggelassen werden, werden die `globals()` und `locals()` des aufrufenden Bereichs für `globals` bzw. `locals` verwendet.

Bemerkungen

In `exec` , wenn `globals` ist `locals` (dh sie auf das gleiche Objekt verweisen), wird der Code ausgeführt , als ob es auf der Modulebene ist. Wenn `globals` und `locals` unterschiedliche Objekte sind, wird der Code so ausgeführt, als wäre er in einem *Klassenkörper* .

Wenn das `globals` Objekt übergeben wird, der `globals` `__builtins__` Schlüssel jedoch nicht angegeben ist, werden die `__builtins__` Funktionen und Namen von Python automatisch zum globalen Bereich hinzugefügt. Um die Verfügbarkeit von Funktionen wie `print` oder `isinstance` im ausgeführten Bereich zu unterdrücken, lassen Sie `globals` den Schlüssel `__builtins__` auf den Wert `None` . Dies ist jedoch keine Sicherheitsfunktion.

Die Python 2-spezifische Syntax sollte nicht verwendet werden. Die Python 3-Syntax funktioniert in Python 2. Daher sind die folgenden Formulare veraltet: <s>

- `exec object`
- `exec object in globals`

- `exec object in globals, locals`

Examples

Aussagen mit exec auswerten

```
>>> code = """for i in range(5):\n    print('Hello world!')"""
>>> exec(code)
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

Auswerten eines Ausdrucks mit eval

```
>>> expression = '5 + 3 * a'
>>> a = 5
>>> result = eval(expression)
>>> result
20
```

Einen Ausdruck vorkompilieren, um ihn mehrmals auszuwerten

`compile` eingebaute Funktion `compile` kann verwendet werden, um einen Ausdruck in ein Codeobjekt vorzukompilieren. Dieses Codeobjekt kann dann an eval übergeben werden. Dies beschleunigt die wiederholte Ausführung des ausgewerteten Codes. Der dritte zu `compile` Parameter muss die Zeichenfolge `'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
```

Auswerten eines Ausdrucks mit eval mit benutzerdefinierten Globals

```
>>> variables = {'a': 6, 'b': 7}
>>> eval('a * b', globals=variables)
42
```

Als Plus kann sich der Code hier nicht versehentlich auf die außerhalb definierten Namen beziehen:

```
>>> 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
```

Bei Verwendung von `defaultdict` können zum Beispiel undefinierte Variablen auf Null gesetzt werden:

```
>>> from collections import defaultdict
>>> variables = defaultdict(int, {'a': 42})
>>> eval('a * c', globals=variables)  # note that 'c' is not explicitly defined
0
```

Auswertung eines Strings, der ein Python-Literal enthält, mit ast.literal_eval

Wenn Sie über eine Zeichenfolge verfügen, die Python-Literale enthält, z. B. Zeichenfolgen, Floats usw., können Sie `ast.literal_eval` , um den Wert anstelle von `eval` auszuwerten. Dies hat die zusätzliche Eigenschaft, nur bestimmte Syntax zuzulassen.

```
>>> import ast
>>> code = """(1, 2, {'foo': 'bar'})"""
>>> object = ast.literal_eval(code)
>>> object
(1, 2, {'foo': 'bar'})
>>> type(object)
<class 'tuple'>
```

Dies ist jedoch nicht sicher für die Ausführung von Code, der von nicht vertrauenswürdigen Benutzern bereitgestellt wird, und es ist trivial, einen Interpreter mit sorgfältig ausgearbeiteten Eingaben zum Absturz zu bringen

```
>>> import ast
>>> ast.literal_eval('()' * 1000000)
[5]    21358 segmentation fault (core dumped)  python3
```

Hier ist die Eingabe eine Zeichenfolge von `()` eine Million Mal wiederholt wird, was zu einem Absturz des CPython-Parsers führt. CPython-Entwickler betrachten Fehler im Parser nicht als Sicherheitsprobleme.

Ausführen von Code, der von nicht vertrauenswürdigem Benutzer mit exec, eval oder ast.literal_eval bereitgestellt wird

Es ist nicht möglich, `eval` oder `exec` zu verwenden, um Code von nicht vertrauenswürdigen Benutzern sicher auszuführen. Selbst `ast.literal_eval` neigt im Parser zum Absturz. Es kann manchmal vor böswilliger Code-Ausführung geschützt werden, schließt aber nicht aus, dass der Parser oder der Tokenizer vollständig abstürzt.

Um Code von einem nicht vertrauenswürdigen Benutzer auszuwerten, müssen Sie sich an ein Drittanbieter-Modul wenden oder vielleicht Ihren eigenen Parser und Ihre eigene virtuelle Maschine in Python schreiben.

Dynamische Code-Ausführung mit "exec" und "eval" online lesen:
https://riptutorial.com/de/python/topic/2251/dynamische-code-ausfuhrung-mit--exec--und--eval-

Kapitel 53: Eigenschaftsobjekte

Bemerkungen

Hinweis : Stellen **Sie** in Python 2 sicher, dass Ihre Klasse von einem Objekt erbt (wodurch es zu einer Klasse mit neuem Stil wird), damit alle Features der Eigenschaften verfügbar sind.

Examples

Verwenden des @property-Dekorators

Mit dem `@property` Dekorator können Methoden in einer Klasse definiert werden, die als Attribute fungieren. Ein Beispiel, in dem dies nützlich sein kann, ist das Freilegen von Informationen, die eine anfängliche (teure) Suche und ein einfaches Abrufen danach erfordern.

In Anbetracht einiger Module `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
```

Dann

```
>>> 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
```

Verwenden des @property-Dekorators für Lese- und Schreibeigenschaften

Wenn Sie `@property` verwenden `@property` , um ein benutzerdefiniertes Verhalten zum Festlegen und `@property` zu implementieren, verwenden Sie dieses Muster:

```
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:])
```

Um dies zu benutzen:

```
>>> 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
```

Nur einen Getter, Setter oder Deleter eines Eigenschaftsobjekts überschreiben

Wenn Sie von einer Klasse mit einer Eigenschaft erben, können Sie eine neue Implementierung für eine oder mehrere der Eigenschaften- `getter` , `setter` oder `deleter` Funktionen bereitstellen, indem Sie auf das Eigenschaftsobjekt *auf die übergeordnete Klasse* verweisen:

```
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)
```

Sie können auch einen Setter oder Deleter hinzufügen, wenn zuvor noch keiner in der Basisklasse vorhanden war.

Eigenschaften ohne Dekorateure verwenden

Die Verwendung der Decorator-Syntax (mit dem @) ist zwar bequem, verdeckt aber auch etwas. Sie können Eigenschaften ohne Dekorateure direkt verwenden. Das folgende Python 3.x-Beispiel zeigt dies:

```
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)
```

Eigenschaftsobjekte online lesen: https://riptutorial.com/de/python/topic/2050/eigenschaftsobjekte

Kapitel 54: Einfache mathematische Operatoren

Einführung

Python führt selbstständig mathematische Operatoren aus, einschließlich Ganzzahl- und Gleitkommadivision, Multiplikation, Exponentiation, Addition und Subtraktion. Das math-Modul (in allen Standard-Python-Versionen enthalten) bietet erweiterte Funktionen wie trigonometrische Funktionen, Root-Operationen, Logarithmen und viele mehr.

Bemerkungen

Numerische Typen und ihre Metaklassen

Die `numbers` Modul enthält die abstrakten Metaklassen für die numerischen Typen:

Unterklassen	Zahlen.Zahl	Zahlen.Integral	Zahlen.Rational	Zahlen.Real	Zahlen.Komp
bool	✓	✓	✓	✓	✓
int	✓	✓	✓	✓	✓
Fraktionen.Fraktion	✓	-	✓	✓	✓
schweben	✓	-	-	✓	✓
Komplex	✓	-	-	-	✓
decimal.Decimal	✓	-	-	-	-

Examples

Zusatz

```
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
```

Mögliche Kombinationen (eingebaute Typen):

- `int` und `int` (gibt ein `int`)
- `int` und `float` (gibt einen `float`)
- `int` und `complex` (ergibt einen `complex`)
- `float` und `float` (gibt einen `float`)
- `float` und `complex` (ergibt einen `complex`)
- `complex` und `complex` (ergibt einen `complex`)

Hinweis: Der Operator `+` wird auch zum Verketten von Zeichenfolgen, Listen und Tupeln verwendet:

```
"first string " + "second string"    # = 'first string second string'

[1, 2, 3] + [4, 5, 6]                # = [1, 2, 3, 4, 5, 6]
```

Subtraktion

```
a, b = 1, 2

# Using the "-" operator:
b - a                  # = 1


import operator        # contains 2 argument arithmetic functions
operator.sub(b, a)     # = 1
```

Mögliche Kombinationen (eingebaute Typen):

- `int` und `int` (gibt ein `int`)
- `int` und `float` (gibt einen `float`)
- `int` und `complex` (ergibt einen `complex`)
- `float` und `float` (gibt einen `float`)
- `float` und `complex` (ergibt einen `complex`)
- `complex` und `complex` (ergibt einen `complex`)

Multiplikation

```
a, b = 2, 3

a * b                  # = 6

import operator
```

```
operator.mul(a, b)      # = 6
```

Mögliche Kombinationen (eingebaute Typen):

- `int` und `int` (gibt ein `int`)
- `int` und `float` (gibt einen `float`)
- `int` und `complex` (ergibt einen `complex`)
- `float` und `float` (gibt einen `float`)
- `float` und `complex` (ergibt einen `complex`)
- `complex` und `complex` (ergibt einen `complex`)

Hinweis: Der Operator `*` wird auch für die wiederholte Verkettung von Zeichenfolgen, Listen und Tupeln verwendet:

```
3 * 'ab'  # = 'abababab'
3 * ('a', 'b')  # = ('a', 'b', 'a', 'b', 'a', 'b')
```

Einteilung

Python führt eine Ganzzahldivision aus, wenn beide Operanden Ganzzahlen sind. Das Verhalten der Divisionsoperatoren von Python hat sich gegenüber Python 2.x und 3.x geändert (siehe auch Integer Division).

```
a, b, c, d, e = 3, 2, 2.0, -3, 10
```

Python 2.x 2.7

In Python 2 hängt das Ergebnis des Operators '/' vom Typ des Zählers und des Nenners ab.

```
a / b                   # = 1

a / c                   # = 1.5

d / b                   # = -2

b / a                   # = 0

d / e                   # = -1
```

Beachten Sie, dass, da sowohl `a` als auch `b` `int` , das Ergebnis ein `int` .

Das Ergebnis wird immer abgerundet (Floored).

Da `c` ein Float ist, ist das Ergebnis von `a / c` ein `float` .

Sie können auch das Operator-Modul verwenden:

```
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

Was ist, wenn Sie eine Float-Division wünschen:

Empfohlen:

```
from __future__ import division # applies Python 3 style division to the entire module
a / b                   # = 1.5
a // b                  # = 1
```

Okay (wenn Sie sich nicht für das gesamte Modul bewerben möchten):

```
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
```

Nicht empfohlen (kann TypeError auslösen, z. B. wenn das Argument komplex ist):

```
float(a) / b            # = 1.5
a / float(b)            # = 1.5
```

Python 2.x 2.2

Der '//' - Operator in Python 2 erzwingt unabhängig vom Typ die Unterteilung.

```
a // b                # = 1
a // c                # = 1.0
```

Python 3.x 3.0

In Python 3 führt der Operator `/` unabhängig vom Typ eine 'echte' Division aus. Der Operator `//` führt die Bodenteilung durch und verwaltet den Typ.

```
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
```

Mögliche Kombinationen (eingebaute Typen):

- `int` und `int` (ergibt ein `int` in Python 2 und ein `float` in Python 3)
- `int` und `float` (gibt einen `float`)

- `int` und `complex` (ergibt einen `complex`)
- `float` und `float` (gibt einen `float`)
- `float` und `complex` (ergibt einen `complex`)
- `complex` und `complex` (ergibt einen `complex`)

Weitere Informationen finden Sie in PEP 238 .

Exponentierung

```
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
```

Ein weiterer Unterschied zwischen dem eingebauten `pow` und `math.pow` besteht darin, dass der eingebaute `pow` drei Argumente akzeptieren kann:

```
a, b, c = 2, 3, 2

pow(2, 3, 2)            # 0, calculates (2 ** 3) % 2, but as per Python docs,
                        #    does so more efficiently
```

Spezialfunktionen

Die Funktion `math.sqrt(x)` berechnet die Quadratwurzel von `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)
```

Um andere Wurzeln zu berechnen, z. B. eine Würfelwurzel, erhöhen Sie die Anzahl auf den Kehrwert des Wurzelgrades. Dies kann mit jeder der Exponentialfunktionen oder dem Operator erfolgen.

```
 import math
 x = 8
 math.pow(x, 1/3) # evaluates to 2.0
 x**(1/3) # evaluates to 2.0
```

Die Funktion `math.exp(x)` berechnet `e ** x` .

```
math.exp(0)  # 1.0
math.exp(1)  # 2.718281828459045 (e)
```

Die Funktion `math.expm1(x)` berechnet `e ** x - 1` . Wenn `x` klein ist, ergibt sich eine deutlich bessere Genauigkeit als `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      # 1.0000005000001666667083333341666...
```

Logarithmen

Standardmäßig berechnet die Funktion `math.log` den Logarithmus einer Zahl, Basis e. Sie können optional eine Basis als zweites Argument angeben.

```
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)
```

Es gibt spezielle Varianten der `math.log` Funktion für verschiedene Basen.

```
# 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)
```

Inplace-Operationen

In Anwendungen ist es üblich, Code wie folgt zu haben:

```
a = a + 1
```

oder

```
a = a * 2
```

Es gibt eine effektive Verknüpfung für diese Vor-Ort-Operationen:

```
a += 1
# and
a *= 2
```

Jeder mathematische Operator kann vor dem Zeichen '=' verwendet werden, um eine Inplace-Operation durchzuführen:

- `-=` dekrementiere die Variable an Ort und Stelle
- `+=` Inkrementieren Sie die Variable an Ort und Stelle
- `*=` Multipliziere die Variable an Ort und Stelle
- `/=` teilen Sie die Variable an Ort und Stelle
- `//=` Boden teilen die Variable in Platz # Python 3
- `%=` Gibt den Modul der Variablen an Ort und Stelle zurück
- `**=` zu einer Kraft vor Ort erhöhen

Es gibt andere Operatoren für die bitweisen Operatoren (`^` , `|` etc).

Trigonometrische Funktionen

```
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
```

> Beachten Sie, dass `math.hypot(x, y)` auch die Länge des Vektors (oder die euklidische Entfernung) vom Ursprung `(0, 0)` bis zum Punkt `(x, y)` .
>
> Um den euklidischen Abstand zwischen zwei Punkten `(x1, y1)` & `(x2, y2)` zu berechnen `(x2, y2)` Sie `math.hypot` wie folgt verwenden
>
> ```
> math.hypot(x2-x1, y2-y1)
> ```

Um aus Radiant -> Grad und Grad -> Radiant zu konvertieren, verwenden Sie `math.degrees` und `math.radians`

```
math.degrees(a)
# Out: 57.29577951308232

math.radians(57.29577951308232)
# Out: 1.0
```

Modul

Wie in vielen anderen Sprachen verwendet Python den Operator `%` zur Berechnung des Moduls.

```
3 % 4      # 3
10 % 2     # 0
6 % 4      # 2
```

Oder mit dem `operator` Modul:

```
import operator

operator.mod(3 , 4)      # 3
operator.mod(10 , 2)     # 0
operator.mod(6 , 4)      # 2
```

Sie können auch negative Zahlen verwenden.

```
-9 % 7      # 5
9 % -7      # -5
-9 % -7     # -2
```

Wenn Sie das Ergebnis der Ganzzahldivision und des Moduls ermitteln müssen, können Sie die `divmod` Funktion als Abkürzung verwenden:

```
quotient, remainder = divmod(9, 4)
# quotient = 2, remainder = 1 as 4 * 2 + 1 == 9
```

Einfache mathematische Operatoren online lesen:
https://riptutorial.com/de/python/topic/298/einfache-mathematische-operatoren

Kapitel 55: Einführung in RabbitMQ mit AMQPStorm

Bemerkungen

Die neueste Version von AMQPStorm ist verfügbar unter pypi oder Sie können es installieren mit pip

```
pip install amqpstorm
```

Examples

So verwenden Sie Nachrichten von RabbitMQ

Beginnen Sie mit dem Importieren der Bibliothek.

```
from amqpstorm import Connection
```

Wenn Sie Nachrichten verwenden, müssen Sie zunächst eine Funktion definieren, um die eingehenden Nachrichten zu verarbeiten. Dies kann eine beliebige aufrufbare Funktion sein und muss ein Nachrichtenobjekt oder ein Nachrichtentupel (abhängig vom in `to_tuple` definierten `start_consuming`) `start_consuming` .

Neben der Verarbeitung der Daten aus der eingehenden Nachricht müssen wir die Nachricht auch bestätigen oder ablehnen. Dies ist wichtig, da wir RabbitMQ mitteilen müssen, dass wir die Nachricht ordnungsgemäß empfangen und verarbeitet haben.

```
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)
```

Als Nächstes müssen wir die Verbindung zum RabbitMQ-Server einrichten.

```
connection = Connection('127.0.0.1', 'guest', 'guest')
```

Danach müssen wir einen Kanal einrichten. Jede Verbindung kann mehrere Kanäle haben. Im Allgemeinen wird bei Multithread-Tasks empfohlen, einen pro Thread zu haben (dies ist jedoch nicht erforderlich).

```
channel = connection.channel()
```

Sobald wir unseren Kanal eingerichtet haben, müssen wir RabbitMQ mitteilen, dass wir damit beginnen möchten, Nachrichten zu verbrauchen. In diesem Fall verwenden wir unsere zuvor definierte Funktion `on_message` , um alle unsere verbrauchten Nachrichten zu verarbeiten.

Die Warteschlange, die wir auf dem RabbitMQ-Server `simple_queue` , wird `simple_queue` , und wir teilen RabbitMQ mit, dass wir alle eingehenden Nachrichten bestätigen werden, sobald wir damit fertig sind.

```
channel.basic.consume(callback=on_message, queue='simple_queue', no_ack=False)
```

Schließlich müssen wir die E / A-Schleife starten, um die Verarbeitung der vom RabbitMQ-Server gelieferten Nachrichten zu starten.

```
channel.start_consuming(to_tuple=False)
```

So veröffentlichen Sie Nachrichten an RabbitMQ

Beginnen Sie mit dem Importieren der Bibliothek.

```
from amqpstorm import Connection
from amqpstorm import Message
```

Als nächstes müssen wir eine Verbindung zum RabbitMQ-Server herstellen.

```
connection = Connection('127.0.0.1', 'guest', 'guest')
```

Danach müssen wir einen Kanal einrichten. Jede Verbindung kann mehrere Kanäle haben. Im Allgemeinen wird bei Multithread-Tasks empfohlen, einen pro Thread zu haben (dies ist jedoch nicht erforderlich).

```
channel = connection.channel()
```

Sobald wir unseren Kanal eingerichtet haben, können wir mit der Vorbereitung unserer Nachricht beginnen.

```
# Message Properties.
properties = {
    'content_type': 'text/plain',
    'headers': {'key': 'value'}
}

# Create the message.
```

```
message = Message.create(channel=channel, body='Hello World!', properties=properties)
```

Jetzt können wir die Nachricht `publish` indem wir einfach `publish` aufrufen und einen `routing_key` . In diesem Fall senden wir die Nachricht an eine Warteschlange mit dem Namen `simple_queue` .

```
message.publish(routing_key='simple_queue')
```

So erstellen Sie eine verzögerte Warteschlange in RabbitMQ

Zuerst müssen wir zwei Basiskanäle einrichten, einen für die Hauptwarteschlange und einen für die Verzögerungswarteschlange. In meinem Beispiel am Ende füge ich ein paar zusätzliche Flags hinzu, die nicht erforderlich sind, aber den Code zuverlässiger machen. wie `confirm delivery` , `delivery_mode` und `durable` . Weitere Informationen hierzu finden Sie im RabbitMQ- Handbuch .

Nachdem wir die Kanäle eingerichtet haben, fügen wir dem Hauptkanal eine Bindung hinzu, mit der wir Nachrichten vom Verzögerungskanal an unsere Hauptwarteschlange senden können.

```
channel.queue.bind(exchange='amq.direct', routing_key='hello', queue='hello')
```

Als Nächstes müssen wir unseren Verzögerungskanal so konfigurieren, dass Nachrichten nach Ablauf an die Hauptwarteschlange weitergeleitet werden.

```
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 *(Nachricht - Zeit zum Leben)*

  Normalerweise wird dies zum automatischen Entfernen alter Nachrichten in der Warteschlange nach einer bestimmten Dauer verwendet. Durch Hinzufügen zweier optionaler Argumente können Sie dieses Verhalten ändern. Stattdessen lässt sich dieser Parameter in Millisekunden festlegen, wie lange Nachrichten in der Verzögerungswarteschlange verbleiben.

- x-dead-letter-routing-key

  Diese Variable ermöglicht es uns, die Nachricht nach ihrem Ablauf in eine andere Warteschlange zu übertragen, anstatt sie standardmäßig zu entfernen.

- x-dead-letter-exchange

  Diese Variable bestimmt, mit welcher Exchange die Nachricht von hello_delay in die Hello-Warteschlange übertragen wurde.

Veröffentlichung in der Verzögerungswarteschlange

Wenn Sie alle grundlegenden Pika-Parameter eingerichtet haben, senden Sie einfach eine

Nachricht in die Verzögerungswarteschlange.

```
delay_channel.basic.publish(exchange='',
                            routing_key='hello_delay',
                            body='test',
                            properties={'delivery_mod': 2})
```

Nachdem Sie das Skript ausgeführt haben, sollten die folgenden Warteschlangen in Ihrem RabbitMQ-Verwaltungsmodul erstellt werden.

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	

Beispiel.

```
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")
```

Einführung in RabbitMQ mit AMQPStorm online lesen:
https://riptutorial.com/de/python/topic/3373/einfuhrung-in-rabbitmq-mit-amqpstorm

Kapitel 56: Einsatz

Examples

Conda-Paket hochladen

Bevor Sie beginnen, müssen Sie Folgendes haben:

Anaconda auf Ihrem System installiert Konto bei Binstar Wenn Sie Anaconda 1.6+ nicht verwenden, installieren Sie den binstar- Befehlszeilenclient:

```
$ conda install binstar
$ conda update binstar
```

Wenn Sie Anaconda nicht verwenden, ist der Binstar auch auf pypi verfügbar:

```
$ pip install binstar
```

Jetzt können wir uns einloggen:

```
$ binstar login
```

Testen Sie Ihr Login mit dem whoami-Befehl:

```
$ binstar whoami
```

Wir werden ein Paket mit einer einfachen "Hallo Welt" -Funktion hochladen. Beginnen Sie, indem Sie zunächst mein Demo-Paket-Repo von Github beziehen:

```
$ git clone https://github.com/<NAME>/<Package>
```

Dies ist ein kleines Verzeichnis, das so aussieht:

```
package/
        setup.py
        test_package/
           __init__.py
           hello.py
           bld.bat
           build.sh
           meta.yaml
```

`Setup.py` ist die Standard-Python-Build-Datei, und `hello.py` hat unsere einzige Funktion hello_world ().

Die `bld.bat` , `build.sh` und `meta.yaml` sind Skripte und Metadaten für das `Conda` Paket. Auf der Conda-Build- Seite finden Sie weitere Informationen zu diesen drei Dateien und ihrem Zweck.

Jetzt erstellen wir das Paket, indem wir Folgendes ausführen:

```
$ conda build test_package/
```

Das ist alles, um ein Conda-Paket zu erstellen.

Der letzte Schritt ist das Hochladen auf binstar, indem die letzte Zeile des Ausdrucks kopiert und eingefügt wird, nachdem der Befehl conda build test_package / ausgeführt wurde. Auf meinem System lautet der Befehl:

```
$ binstar upload /home/xavier/anaconda/conda-bld/linux-64/test_package-0.1.0-py27_0.tar.bz2
```

Da Sie zum ersten Mal ein Paket und ein Release erstellen, werden Sie aufgefordert, einige Textfelder auszufüllen, die alternativ über die Web-App erfolgen könnten.

Sie werden ein absolutes `done` ausgedruckt , um zu bestätigen Sie erfolgreich Ihre Conda Paket Binstar hochgeladen haben.

Einsatz online lesen: https://riptutorial.com/de/python/topic/4064/einsatz

Kapitel 57: einstellen

Syntax

- empty_set = set () # initialisiert eine leere Menge
- literal_set = {'foo', 'bar', 'baz'} # Konstruiere einen Satz mit 3 Strings darin
- set_from_list = set (['foo', 'bar', 'baz']) # ruft die Set-Funktion für ein neues Set auf
- set_from_iter = set (x für x im Bereich (30)) # verwendet beliebige iterierbare Elemente, um eine Menge zu erstellen
- set_from_iter = {x für x in [random.randint (0,10) für i im Bereich (10)]} # alternative Schreibweise

Bemerkungen

Sets sind *ungeordnet* und haben eine *sehr schnelle Suchzeit* (Amortized O (1), wenn Sie technische Informationen erhalten möchten). Es ist großartig zu verwenden, wenn Sie eine Sammlung von Dingen haben, die Reihenfolge spielt keine Rolle und Sie werden häufig nach Namen suchen. Wenn es sinnvoller ist, Elemente nach einer Indexnummer zu suchen, sollten Sie stattdessen eine Liste verwenden. Wenn die Reihenfolge wichtig ist, ziehen Sie auch eine Liste in Betracht.

Sets sind *veränderbar* und können daher nicht gehasht werden. Sie können sie also nicht als Wörterbuchschlüssel verwenden oder in andere Sets einfügen oder an anderen Stellen, an denen Hash-Typen erforderlich sind. In solchen Fällen können Sie ein unveränderliches `frozenset` .

Die Elemente eines Sets müssen *hashbar sein* . Dies bedeutet, dass sie eine korrekte `__hash__` Methode haben, die mit `__eq__` . Im Allgemeinen sind veränderliche Typen wie `list` oder `set` nicht hashierbar und können nicht in einen Satz eingefügt werden. Wenn dieses Problem `dict` , sollten Sie `dict` und unveränderliche Schlüssel in Betracht ziehen.

Examples

Holen Sie sich die einzigartigen Elemente einer Liste

Nehmen wir an, Sie haben eine Liste von Restaurants - vielleicht lesen Sie sie aus einer Datei. Sie interessieren sich für die *einzigartigen* Restaurants in der Liste. Der beste Weg, die eindeutigen Elemente aus einer Liste zu erhalten, besteht darin, sie in einen Satz umzuwandeln:

```
restaurants = ["McDonald's", "Burger King", "McDonald's", "Chicken Chicken"]
unique_restaurants = set(restaurants)
print(unique_restaurants)
# prints {'Chicken Chicken', "McDonald's", 'Burger King'}
```

Beachten Sie, dass sich das Set nicht in der gleichen Reihenfolge wie die ursprüngliche Liste befindet. Das ist, weil Mengen *ungeordnet sind* , genau wie `dict` .

Dies kann leicht wieder in eine umgewandelt werden `List` mit Pythons eingebaute `list` Funktion, eine weitere Liste geben , die die gleiche Liste wie das Original , aber ohne Duplikate:

```
list(unique_restaurants)
# ['Chicken Chicken', "McDonald's", 'Burger King']
```

Es ist auch üblich, dies als eine Zeile zu sehen:

```
# Removes all duplicates and returns another list
list(set(restaurants))
```

Jetzt können alle Operationen, die in der ursprünglichen Liste ausgeführt werden könnten, erneut ausgeführt werden.

Operationen an Sets

mit anderen Sets

```
# 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
```

mit einzelnen Elementen

```
# 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!
```

Setoperationen geben neue Sets zurück, verfügen jedoch über die entsprechenden In-Place-Versionen:

Methode	In-Place-Betrieb	In-Place-Methode
Union	s | = t	aktualisieren
Überschneidung	s & = t	intersection_update
Unterschied	s - = t	difference_update
symmetrischer_unterschied	s ^ = t	symmetric_difference_update

Zum Beispiel:

```
s = {1, 2}
s.update({3, 4})   # s == {1, 2, 3, 4}
```

Sets im Vergleich zu Multisets

Sets sind ungeordnete Sammlungen verschiedener Elemente. Manchmal möchten wir jedoch mit ungeordneten Sammlungen von Elementen arbeiten, die nicht notwendigerweise verschieden sind, und die Multiplizitäten der Elemente verfolgen.

Betrachten Sie dieses Beispiel:

```
>>> setA = {'a','b','b','c'}
>>> setA
set(['a', 'c', 'b'])
```

Durch das Speichern der Zeichenfolgen `'a'` , `'b'` , `'b'` , `'c'` in einer festgelegten Datenstruktur haben wir die Information verloren, dass `'b'` zweimal vorkommt. Wenn Sie die Elemente in einer Liste speichern, bleiben diese Informationen natürlich erhalten

```
>>> listA = ['a','b','b','c']
>>> listA
['a', 'b', 'b', 'c']
```

Eine Listendatenstruktur führt jedoch eine zusätzliche nicht benötigte Reihenfolge ein, die unsere Berechnungen verlangsamt.

Zur Implementierung von Multisets stellt Python die `Counter` Klasse aus dem `collections` Modul (ab Version 2.7) bereit:

Python 2.x 2.7

```
>>> from collections import Counter
>>> counterA = Counter(['a','b','b','c'])
>>> counterA
Counter({'b': 2, 'a': 1, 'c': 1})
```

`Counter` ist ein Wörterbuch, in dem Elemente als Wörterbuchschlüssel und ihre Zählwerte als Wörterbuchwerte gespeichert werden. Und wie alle Wörterbücher ist es eine ungeordnete Sammlung.

Festlegen von Operationen mit Methoden und eingebauten Elementen

Wir definieren zwei Mengen `a` und `b`

```
>>> a = {1, 2, 2, 3, 4}
>>> b = {3, 3, 4, 4, 5}
```

> HINWEIS: `{1}` erstellt einen Satz eines Elements, `{}` erstellt jedoch ein leeres `dict` . Die korrekte Methode zum Erstellen eines leeren Sets ist `set()` .

Überschneidung

`a.intersection(b)` gibt eine neue Menge mit Elementen zurück, die in `a` und `b`

```
>>> a.intersection(b)
{3, 4}
```

Union

`a.union(b)` gibt eine neue Menge mit Elementen zurück, die in `a` und `b`

```
>>> a.union(b)
{1, 2, 3, 4, 5}
```

Unterschied

`a.difference(b)` gibt eine neue Menge mit Elementen zurück, die in `a` aber nicht in `b`

```
>>> a.difference(b)
{1, 2}
>>> b.difference(a)
```

```
{5}
```

Symmetrischer Unterschied

`a.symmetric_difference(b)` gibt eine neue Menge mit Elementen zurück, die entweder in `a` oder `b` jedoch nicht in beiden

```
>>> a.symmetric_difference(b)
{1, 2, 5}
>>> b.symmetric_difference(a)
{1, 2, 5}
```

HINWEIS: `a.symmetric_difference(b) == b.symmetric_difference(a)`

Teilmenge und Obermenge

`c.issubset(a)` prüft, ob sich jedes Element von `c` in `a` .

`a.issuperset(c)` prüft, ob sich jedes Element von `c` in `a` .

```
>>> c = {1, 2}
>>> c.issubset(a)
True
>>> a.issuperset(c)
True
```

Die letzteren Operationen haben gleichwertige Operatoren, wie unten gezeigt:

Methode	Operator
`a.intersection(b)`	`a & b`
`a.union(b)`	`a | b`
`a.difference(b)`	`a - b`
`a.symmetric_difference(b)`	`a ^ b`
`a.issubset(b)`	`a <= b`
`a.issuperset(b)`	`a >= b`

Disjunkte Sätze

Die Sätze `a` und `d` sind nicht zusammenhängend, wenn kein Element in `a` auch in `d` und umgekehrt.

```
>>> 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
```

Mitgliedschaft testen

Die eingebaute `in` Keyword sucht nach Vorkommen

```
>>> 1 in a
True
>>> 6 in a
False
```

Länge

Die eingebaute `len()` Funktion gibt die Anzahl der Elemente im Satz zurück

```
>>> len(a)
4
>>> len(b)
3
```

Satz von Sets

```
{{1,2}, {3,4}}
```

führt zu:

```
TypeError: unhashable type: 'set'
```

Verwenden `frozenset` stattdessen `frozenset` :

```
{frozenset({1, 2}), frozenset({3, 4})}
```

einstellen online lesen: https://riptutorial.com/de/python/topic/497/einstellen

Kapitel 58: Enum

Bemerkungen

Aufzählungen wurden in Version 3.4 von Python hinzugefügt PEP 435 .

Examples

Erstellen einer Enumeration (Python 2.4 bis 3.3)

Enumerationen wurden von Python 3.4 bis Python 2.4 über Python 3.3 zurückportiert. Den enum34- Backport erhalten Sie von PyPI.

```
pip install enum34
```

Die Erstellung eines Enums ist identisch mit der Funktionsweise von 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
```

Iteration

Enumerationen sind wiederholbar:

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

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

Enum online lesen: https://riptutorial.com/de/python/topic/947/enum

Kapitel 59: Erste Schritte mit GZip

Einführung

Dieses Modul bietet eine einfache Schnittstelle zum Komprimieren und Dekomprimieren von Dateien wie die GNU-Programme gzip und gunzip.

Die Datenkomprimierung wird vom zlib-Modul bereitgestellt.

Das gzip-Modul stellt die GzipFile-Klasse bereit, die nach Pythons File-Objekt modelliert ist. Die GzipFile-Klasse liest und schreibt Dateien im GZIP-Format, wobei die Daten automatisch komprimiert oder dekomprimiert werden, sodass sie wie ein gewöhnliches Dateiobjekt aussehen.

Examples

Lesen und Schreiben von GNU-ZIP-Dateien

```
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)
```

Speichern Sie es als 1gzip_write.py1.Führen Sie es über das Terminal aus.

```
$ python gzip_write.py

application/x-gzip; charset=binary
example.txt.gz contains 68 bytes of compressed data
```

Erste Schritte mit GZip online lesen: https://riptutorial.com/de/python/topic/8993/erste-schritte-mit-gzip

Kapitel 60: Erstellen eines Windows-Dienstes mit Python

Einführung

Headless-Prozesse (ohne Benutzeroberfläche) in Windows werden als Services bezeichnet. Sie können mit Standard-Windows-Steuerelementen wie der Befehlskonsole, Powershell oder der Registerkarte Dienste im Task-Manager gesteuert (gestartet, gestoppt usw.) werden. Ein gutes Beispiel ist eine Anwendung, die Netzwerkdienste bereitstellt, beispielsweise eine Webanwendung, oder eine Sicherungsanwendung, die verschiedene Hintergrundarchivierungsaufgaben ausführt. Es gibt mehrere Möglichkeiten, eine Python-Anwendung als Dienst in Windows zu erstellen und zu installieren.

Examples

Ein Python-Skript, das als Dienst ausgeführt werden kann

Die in diesem Beispiel verwendeten Module sind Teil von pywin32 (Python für Windows-Erweiterungen). Je nachdem, wie Sie Python installiert haben, müssen Sie dies möglicherweise separat installieren.

```
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)
```

Dies ist nur eine Platte. Ihr Anwendungscode, der wahrscheinlich ein separates Skript aufruft, würde in die Funktion main () gehen.

Sie müssen dies auch als Dienst installieren. Die beste Lösung hierfür scheint momentan der Einsatz von Non-sucking Service Manager zu sein . Dadurch können Sie einen Dienst installieren und stellen eine GUI zum Konfigurieren der Befehlszeile bereit, die der Dienst ausführt. Für Python können Sie dies tun, wodurch der Dienst auf einmal erstellt wird:

```
nssm install MyServiceName c:\python27\python.exe c:\temp\myscript.py
```

Wo my_script.py das obige Boilerplate-Skript ist, wurde es so geändert, dass es das Anwendungsskript oder den Code in der Funktion main () aufruft. Beachten Sie, dass der Dienst das Python-Skript nicht direkt ausführt, sondern den Python-Interpreter und das Hauptskript an die Befehlszeile weitergibt.

Alternativ können Sie Tools verwenden, die im Windows Server Resource Kit für Ihre Betriebssystemversion bereitgestellt werden, um den Dienst zu erstellen.

Ausführen einer Flask-Webanwendung als Dienst

Dies ist eine Variation des generischen Beispiels. Sie müssen nur Ihr App-Skript importieren und die Methode `run()` in der Funktion `main()` des Dienstes aufrufen. In diesem Fall verwenden wir auch das Multiprocessing-Modul aufgrund eines Problems, das auf `WSGIRequestHandler` zugreift.

```
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)
```

Angepasst von http://stackoverflow.com/a/25130524/318488

Erstellen eines Windows-Dienstes mit Python online lesen:
https://riptutorial.com/de/python/topic/9065/erstellen-eines-windows-dienstes-mit-python

Kapitel 61: Erstellen Sie eine virtuelle Umgebung mit Virtualenvwrapper in Windows

Examples

Virtuelle Umgebung mit Virtualenvwrapper für Windows

Angenommen, Sie müssen an drei verschiedenen Projekten arbeiten. Projekt A, Projekt B und Projekt C. Projekt A und Projekt B benötigen Python 3 und einige erforderliche Bibliotheken. Für Projekt C benötigen Sie jedoch Python 2.7 und abhängige Bibliotheken.

Daher empfiehlt es sich, diese Projektumgebungen voneinander zu trennen. Zum Erstellen einer separaten virtuellen Python-Umgebung müssen die folgenden Schritte ausgeführt werden:

Schritt 1: Installieren Sie pip mit diesem Befehl: `python -m pip install -U pip`

Schritt 2: Dann installieren Sie das Paket "virtualenvwrapper-win" mit dem Befehl (Befehl kann von Windows Power Shell ausgeführt werden)

```
pip install virtualenvwrapper-win
```

Schritt 3: Erstellen Sie mit dem Befehl `mkvirtualenv python_3.5` eine neue virtualenv-Umgebung

Schritt 4: Aktivieren Sie die Umgebung mit dem Befehl:

```
workon < environment name>
```

Hauptbefehle für 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.
```

Erstellen Sie eine virtuelle Umgebung mit Virtualenvwrapper in Windows online lesen: https://riptutorial.com/de/python/topic/9984/erstellen-sie-eine-virtuelle-umgebung-mit-virtualenvwrapper-in-windows

Kapitel 62: Erweiterungen schreiben

Examples

Hallo Welt mit C-Erweiterung

Die folgende C-Quelldatei (die wir zu Demonstrationszwecken `hello.c` nennen) erzeugt ein Erweiterungsmodul namens `hello` , das eine einzige Funktion `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
```

Führen Sie den folgenden Befehl in Ihrem bevorzugten Terminal aus, um die Datei mit dem `gcc` Compiler zu kompilieren:

```
gcc /path/to/your/file/hello.c -o /path/to/your/file/hello
```

Um die zuvor beschriebene Funktion `greet()` auszuführen, erstellen Sie eine Datei im selben Verzeichnis und nennen Sie sie `hello.py`

```
import hello          # imports the compiled library
hello.greet("Hello!") # runs the greet() function with "Hello!" as an argument
```

Eine offene Datei an C-Erweiterungen übergeben

Übergeben Sie ein geöffnetes Dateiobjekt von Python an den C-Erweiterungscode.

Sie können die Datei mit der `PyObject_AsFileDescriptor` Funktion in einen Integer-Dateideskriptor `PyObject_AsFileDescriptor` :

```
PyObject *fobj;
int fd = PyObject_AsFileDescriptor(fobj);
if (fd < 0){
    return NULL;
}
```

Verwenden Sie `PyFile_FromFd` um einen Integer-Dateideskriptor zurück in ein Python-Objekt zu `PyFile_FromFd` .

```
int fd; /* Existing file descriptor */
PyObject *fobj = PyFile_FromFd(fd, "filename","r",-1,NULL,NULL,NULL,1);
```

C-Erweiterung mit c ++ und Boost

Dies ist ein grundlegendes Beispiel für eine *C-Erweiterung* , die C ++ und Boost verwendet .

C ++ - Code

C ++ - Code in hello.cpp einfügen:

```
#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)
      ;
}
```

Um dies in ein Python-Modul zu kompilieren, benötigen Sie die Python-Header und die Boost-Bibliotheken. Dieses Beispiel wurde auf Ubuntu 12.04 mit Python 3.4 und Gcc erstellt. Boost wird auf vielen Plattformen unterstützt. Bei Ubuntu wurden die benötigten Pakete installiert mit:

```
sudo apt-get install gcc libboost-dev libpython3.4-dev
```

Kompilieren der Quelldatei in eine .so-Datei, die später als Modul importiert werden kann, sofern sie sich im Python-Pfad befindet:

```
gcc -shared -o hello.so -fPIC -I/usr/include/python3.4 hello.cpp -lboost_python-py34 -
lboost_system -l:libpython3.4m.so
```

Der Python-Code in der Datei example.py:

```
import hello

print(hello.get_hello())

h = hello.Hello("World hello!")
print(h.as_list(3))
```

Dann `python3 example.py` die folgende Ausgabe:

```
Hello world!
['World hello!', 'World hello!', 'World hello!']
```

Erweiterungen schreiben online lesen: https://riptutorial.com/de/python/topic/557/erweiterungen-schreiben

Kapitel 63: Externe Datendateien mit Pandas eingeben, unterteilen und ausgeben

Einführung

In diesem Abschnitt wird der grundlegende Code zum Lesen, Untersetzen und Schreiben externer Dateien mit Pandas beschrieben.

Examples

Grundlegender Code zum Importieren, Subset und Schreiben von externen Datendateien mit 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)
```

Externe Datendateien mit Pandas eingeben, unterteilen und ausgeben online lesen: https://riptutorial.com/de/python/topic/8854/externe-datendateien-mit-pandas-eingeben--unterteilen-und-ausgeben

Kapitel 64: Filter

Syntax

- Filter (Funktion, iterierbar)
- itertools.ifilter (Funktion, iterierbar)
- future_builtins.filter (Funktion, iterierbar)
- itertools.ifilterfalse (Funktion, iterierbar)
- itertools.filterfalse (Funktion, iterierbar)

Parameter

Parameter	Einzelheiten
Funktion	*aufrufbare,* die den Zustand oder bestimmt `None` zum Filtern *(Positions-only)* , dann verwenden , um die Identitätsfunktion
iterable	iterierbar, dass gefiltert wird (nur *positionell*)

Bemerkungen

In den meisten Fällen ist ein Verständnis- oder Generatorausdruck lesbarer, leistungsfähiger und effizienter als `filter()` oder `ifilter()` .

Examples

Grundlegende Verwendung des Filters

Verworfene Elemente einer Sequenz anhand einiger Kriterien `filter` :

```
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>
```

Filter ohne Funktion

Wenn der Funktionsparameter `None` , wird die Identitätsfunktion verwendet:

```
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
```

Als Kurzschlussprüfung filtern

`filter` (Python 3.x) und `ifilter` (Python 2.x) zurückkehren , einen Generator , so dass sie sehr praktisch sein kann , wenn ein Kurzschlusstest wie das Erstellen `or` oder `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
```

So finden Sie das erste Element, das kleiner als 100 ist:

```
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)
```

Die `next` gibt das nächste (in diesem Fall das erste) Element an und ist daher der Grund, warum es einen Kurzschluss gibt.

Komplementärfunktion: filterfalse, ifilterfalse

Es gibt eine ergänzende Funktion zum `filter` im `itertools` Modul:

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
```

die funktionieren genau wie die *Generator* - `filter` , sondern halten nur die Elemente , die sind `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)
```

Filter online lesen: https://riptutorial.com/de/python/topic/201/filter

Kapitel 65: Flasche

Einführung

Flask ist ein Python-Mikro-Web-Framework, das für die Ausführung großer Websites wie Pintrest, Twilio und Linkedin verwendet wird. In diesem Thema werden die vielfältigen Funktionen von Flask für die Entwicklung des Front- und Back-End-Webs erläutert und veranschaulicht.

Syntax

- @ app.route ("/ urlpath", method = ["GET", "POST", "DELETE", "PUTS", "HEAD", "OPTIONS"])
- @ app.route ("/ urlpath / <param>", method = ["GET", "POST", "DELETE", "PUTS", "HEAD", "OPTIONS"])

Examples

Die Grundlagen

Das folgende Beispiel ist ein Beispiel für einen Basis-Server:

```
# 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()
```

Beim Ausführen dieses Skripts (mit allen richtigen Abhängigkeiten installiert) sollte ein lokaler Server gestartet werden. Der Host ist `127.0.0.1` allgemein als **localhost bezeichnet** . Dieser Server wird standardmäßig auf Port **5000 ausgeführt** . Um auf Ihren Webserver zuzugreifen, öffnen Sie einen Webbrowser und geben Sie die URL `localhost:5000` oder `127.0.0.1:5000` (kein Unterschied). Derzeit kann nur Ihr Computer auf den Webserver zugreifen.

`app.run()` verfügt über die drei Parameter **host** , **port** und **debug** . Der Host ist standardmäßig `127.0.0.1` `0.0.0.0` Sie diesen `0.0.0.0` auf `0.0.0.0` wird Ihr Webserver von jedem Gerät in Ihrem Netzwerk aus über Ihre private IP-Adresse in der URL erreichbar. Der Port ist standardmäßig 5000, aber wenn der Parameter auf Port `80` , müssen Benutzer keine Portnummer angeben, da

Browser standardmäßig Port 80 verwenden. Bei der Debug-Option ist es hilfreich, diesen Parameter während des Entwicklungsprozesses (niemals in Produktion) auf "True" zu setzen, da Ihr Server neu startet, wenn Änderungen an Ihrem Flask-Projekt vorgenommen werden.

```
if __name__ == "__main__":
    app.run(host="0.0.0.0", port=80, debug=True)
```

Routing-URLs

Mit Flask wird das URL-Routing traditionell mit Dekoratoren durchgeführt. Diese Dekoratoren können für statisches Routing sowie für Routing-URLs mit Parametern verwendet werden. Stellen Sie sich für das folgende Beispiel vor, dass dieses Flask-Skript die Website `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"
```

Mit dieser letzten Route können Sie feststellen, dass bei Angabe einer URL mit / users / und dem Profilnamen ein Profil zurückgegeben werden kann. Da es `@app.route()` ineffizient und chaotisch wäre, für jeden Benutzer ein `@app.route()` , bietet Flask an, Parameter aus der URL zu übernehmen:

```
@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
```

HTTP-Methoden

Die zwei häufigsten HTTP-Methoden sind **GET** und **POST** . Flask kann abhängig von der verwendeten HTTP-Methode unterschiedlichen Code von derselben URL ausführen. In einem Webservice mit Konten ist es zum Beispiel am bequemsten, die Anmeldeseite und den Anmeldevorgang über dieselbe URL weiterzuleiten. Eine GET-Anforderung, die auch beim Öffnen einer URL in Ihrem Browser erfolgt, sollte das Anmeldeformular anzeigen, während eine POST-Anforderung (mit Anmeldedaten) separat verarbeitet werden soll. Eine Route wird auch erstellt, um die HTTP-Methode DELETE und PUT zu verarbeiten.

```
@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"
```

Um den Code etwas zu vereinfachen, können wir das `request` aus der Flasche importieren.

```
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"
```

Um Daten aus der POST-Anforderung abzurufen, müssen wir das `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"]
```

Dateien und Vorlagen

Anstatt unser HTML-Markup in die return-Anweisungen `render_template()` , können Sie die Funktion `render_template()` verwenden:

```
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)
```

Dazu wird unsere Vorlagendatei `about-us.html` . Um sicherzustellen, dass unsere Anwendung diese Datei finden kann, müssen wir unser Verzeichnis in folgendem Format organisieren:

```
- application.py
/templates
    - about-us.html
    - login-form.html
/static
    /styles
        - about-style.css
        - login-style.css
    /scripts
        - about-script.js
        - login-script.js
```

Am wichtigsten ist, dass Verweise auf diese Dateien im HTML-Code folgendermaßen aussehen:

```
<link rel="stylesheet" type="text/css", href="{{url_for('static', filename='styles/about-
style.css')}}">
```

`about-style.css` wird die Anwendung `about-style.css` im `about-style.css` unter dem statischen Ordner nach " `about-style.css` zu suchen. Das gleiche Pfadformat gilt für alle Verweise auf Bilder, Stile, Skripts oder Dateien.

Jinja Templating

Ähnlich wie Meteor.js lässt sich Flask gut in Front-End-Templating-Services integrieren. Flask verwendet standardmäßig Jinja Templating. Mithilfe von Vorlagen können kleine Codefragmente in der HTML-Datei verwendet werden, z. B. Bedingungen oder Schleifen.

Wenn wir eine Vorlage rendern, werden alle Parameter außerhalb des Vorlagendateinamens an den HTML-Vorlagendienst übergeben. Die folgende Route wird den Benutzernamen und das Verbindungsdatum (von einer anderen Stelle) an den HTML-Code übergeben.

```
@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)
```

Wenn diese Vorlage gerendert wird, kann sie die von der Funktion `render_template()` an sie übergebenen Variablen verwenden. Hier sind die Inhalte von `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>
```

Die folgenden Trennzeichen werden für verschiedene Interpretationen verwendet:

- `{% ... %}` bezeichnet eine Aussage
- `{{ ... }}` bezeichnet einen Ausdruck, in dem eine Vorlage ausgegeben wird
- `{# ... #}` kennzeichnet einen Kommentar (nicht in der Vorlagenausgabe enthalten)
- `{# ... ##` impliziert, dass der Rest der Zeile als Anweisung interpretiert werden soll

Das Anforderungsobjekt

Das `request` liefert Informationen zu der Anforderung, die an die Route gestellt wurde. Um dieses Objekt verwenden zu können, muss es aus dem Flaschenmodul importiert werden:

```
from flask import request
```

URL-Parameter

In vorherigen Beispielen wurden `request.method` und `request.form` verwendet. Wir können jedoch auch die Eigenschaft `request.args` , um ein Wörterbuch der Schlüssel / Werte in den URL-Parametern abzurufen.

```
@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"
```

Um sich in diesem Zusammenhang korrekt zu authentifizieren, ist die folgende URL erforderlich (Ersetzung des Benutzernamens durch einen beliebigen Benutzernamen):

`www.example.com/api/users/guido-van-rossum?key=pa55w0Rd`

Datei-Uploads

Wenn ein Dateiupload Teil des übermittelten Formulars in einer POST-Anforderung war, können die Dateien mit dem `request` werden:

```
@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
```

Kekse

Die Anfrage kann auch Cookies in einem Wörterbuch enthalten, die den URL-Parametern ähneln.

```
@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")
```

Flasche online lesen: https://riptutorial.com/de/python/topic/8682/flasche

Kapitel 66: Functools-Modul

Examples

teilweise

Die `partial` erstellt eine Teilfunktionsanwendung aus einer anderen Funktion. Es wird verwendet, *um* Werte an einige Argumente der Funktion (oder Schlüsselwortargumente) zu *binden* und eine *Aufruffunktion* ohne die bereits definierten Argumente zu erzeugen.

```
>>> from functools import partial
>>> unhex = partial(int, base=16)
>>> unhex.__doc__ = 'Convert base16 string to int'
>>> unhex('ca11ab1e')
3390155550
```

`partial()` erlaubt, wie der Name schon sagt, eine partielle Bewertung einer Funktion. Schauen wir uns folgendes Beispiel an:

```
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
```

Wenn `g` erstellt wird, wird `f` mit vier Argumenten (`a, b, c, x`) auch für die ersten drei Argumente `a, b, c,` Die Bewertung von `f` ist abgeschlossen, wenn `g` aufgerufen wird, `g(2)` , wodurch das vierte Argument an `f` .

Eine Möglichkeit, an `partial` zu denken, ist ein Schieberegister; ein Argument zu einer Zeit in eine Funktion einschieben. `partial` praktisch für Fälle, in denen Daten als Stream eingehen und wir nur ein Argument übergeben können.

total_ordering

Wenn Sie eine bestellbare Klasse erstellen möchten, müssen Sie normalerweise die Methoden `__eq()__` , `__lt__()` , `__le__()` , `__gt__()` und `__ge__()` .

Der `total_ordering` Dekorator, der auf eine Klasse angewendet wird, erlaubt die Definition von `__eq__()` und nur eines zwischen `__lt__()` , `__le__()` , `__gt__()` und `__ge__()` , und erlaubt dennoch alle `__ge__()` für die Klasse.

```
@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))
```

Der Dekorateur verwendet eine Zusammensetzung der bereitgestellten Methoden und algebraischen Operationen, um die anderen Vergleichsmethoden abzuleiten. Wenn wir beispielsweise `__lt__()` und `__eq()__` definiert haben und `__lt__()` ableiten `__gt__()` , können wir einfach `not __lt__() and not __eq()__` prüfen.

Hinweis : Die Funktion `total_ordering` ist erst seit Python 2.7 verfügbar.

reduzieren

In Python 3.x, die `reduce` bereits erläuterte Funktion hier wurde von den Einbauten entfernt und muss nun von importiert werden `functools` .

```
from functools import reduce
def factorial(n):
    return reduce(lambda a, b: (a*b), range(1, n+1))
```

lru_cache

Der `@lru_cache` Dekorateur kann mit einer Verpackung , eine teure, rechenintensive Funktion verwendet werden Least Recently Used - Cache. Auf diese Weise können Funktionsaufrufe gespeichert werden, sodass zukünftige Aufrufe mit denselben Parametern sofort zurückgegeben werden können, anstatt neu berechnet zu werden.

```
@lru_cache(maxsize=None)  # Boundless cache
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

>>> fibonacci(15)
```

In dem obigen Beispiel der Wert der `fibonacci(3)` wird nur einmal berechnet, während , wenn `fibonacci` keine LRU - Cache hat, `fibonacci(3)` würde nach oben von 230 mal berechnet wurde. Daher ist `@lru_cache` besonders für rekursive Funktionen oder für die dynamische Programmierung `@lru_cache` , bei denen eine teure Funktion mit denselben exakten Parametern mehrmals aufgerufen werden könnte.

`@lru_cache` hat zwei Argumente

- `maxsize` : Anzahl der zu speichernden Anrufe. Wenn die Anzahl der eindeutigen Aufrufe `maxsize` überschreitet, werden die zuletzt verwendeten Aufrufe durch den LRU-Cache

entfernt.

- `typed` (hinzugefügt in 3.3): Flag zum Bestimmen, ob äquivalente Argumente verschiedener Typen zu unterschiedlichen Cache-Datensätzen gehören (dh, wenn `3.0` und `3` als unterschiedliche Argumente zählen)

Wir können auch Cache-Statistiken sehen:

```
>>> fib.cache_info()
CacheInfo(hits=13, misses=16, maxsize=None, currsize=16)
```

HINWEIS : Da `@lru_cache` Wörterbücher zum Zwischenspeichern von Ergebnissen verwendet, müssen alle Parameter für die Funktion hashierbar sein, damit der Cache funktioniert.

Offizielle Python-Dokumente für `@lru_cache` . `@lru_cache` wurde in 3.2 hinzugefügt.

cmp_to_key

Python hat seine Sortiermethoden geändert, um eine Schlüsselfunktion zu akzeptieren. Diese Funktionen nehmen einen Wert an und geben einen Schlüssel zurück, mit dem die Arrays sortiert werden.

Alte Vergleichsfunktionen verwendeten zwei Werte und geben -1, 0 oder +1 zurück, wenn das erste Argument klein, gleich oder größer als das zweite Argument ist. Dies ist nicht kompatibel mit der neuen Tastenfunktion.

`functools.cmp_to_key` kommt `functools.cmp_to_key` ins `functools.cmp_to_key` :

```
>>> import functools
>>> import locale
>>> sorted(["A", "S", "F", "D"], key=functools.cmp_to_key(locale.strcoll))
['A', 'D', 'F', 'S']
```

Beispiel genommen und angepasst aus der Python Standard Library-Dokumentation .

Functools-Modul online lesen: https://riptutorial.com/de/python/topic/2492/functools-modul

Kapitel 67: Funktionale Programmierung in Python

Einführung

Die Funktionsprogrammierung zerlegt ein Problem in eine Reihe von Funktionen. Im Idealfall nehmen Funktionen nur Eingaben auf und erzeugen Ausgaben, und sie haben keinen internen Status, der die Ausgabe für eine bestimmte Eingabe beeinflusst.

Examples

Lambda-Funktion

Eine anonyme Inline-Funktion, die mit Lambda definiert ist. Die Parameter des Lambda sind links vom Doppelpunkt definiert. Der Funktionskörper wird rechts vom Doppelpunkt definiert. Das Ergebnis der Ausführung des Funktionskörpers wird (implizit) zurückgegeben.

```
s=lambda x:x*x
s(2)    =>4
```

Kartenfunktion

Map übernimmt eine Funktion und eine Sammlung von Elementen. Es erstellt eine neue leere Sammlung, führt die Funktion für jedes Element in der ursprünglichen Sammlung aus und fügt jeden Rückgabewert in die neue Sammlung ein. Es gibt die neue Sammlung zurück.

Dies ist eine einfache Map, die eine Namensliste verwendet und eine Liste der Längen dieser Namen zurückgibt:

```
name_lengths = map(len, ["Mary", "Isla", "Sam"])
print(name_lengths)    =>[4, 4, 3]
```

Funktion reduzieren

Reduzieren nimmt eine Funktion und eine Sammlung von Elementen. Es gibt einen Wert zurück, der durch Kombinieren der Elemente erstellt wird.

Dies ist eine einfache Reduzierung. Sie gibt die Summe aller Elemente in der Sammlung zurück.

```
total = reduce(lambda a, x: a + x, [0, 1, 2, 3, 4])
print(total)    =>10
```

Filterfunktion

Der Filter übernimmt eine Funktion und eine Sammlung. Es gibt eine Auflistung aller Elemente zurück, für die die Funktion True zurückgegeben hat.

```
arr=[1,2,3,4,5,6]
[i for i in filter(lambda x:x>4,arr)]    # outputs[5,6]
```

Funktionale Programmierung in Python online lesen:
https://riptutorial.com/de/python/topic/9552/funktionale-programmierung-in-python

Kapitel 68: Funktionen

Einführung

Funktionen in Python bieten organisierten, wiederverwendbaren und modularen Code zum Ausführen bestimmter Aktionen. Funktionen vereinfachen den Codierungsprozess, verhindern redundante Logik und erleichtern die Verfolgung des Codes. In diesem Thema wird die Deklaration und Verwendung von Funktionen in Python beschrieben.

Python hat viele *integrierte Funktionen* wie `print()` , `input()` und `len()` . Neben den integrierten Funktionen können Sie auch eigene Funktionen erstellen, um spezifischere Aufgaben auszuführen. Diese Funktionen werden als *benutzerdefinierte Funktionen bezeichnet* .

Syntax

- def *funktionsname* (*arg1, ... argN, * args, kw1, kw2 = default, ..., ** kwargs*): *Anweisungen*
- Lambda *arg1, ... argN, * args, kw1, kw2 = default, ..., ** kwargs* : *Ausdruck*

Parameter

Parameter	Einzelheiten
arg1 , ..., *argN*	Regelmäßige Argumente
* *args*	Unbenannte Positionsargumente
kw1 , ..., *kwN*	Nur ausschließliche Argumente
** *Kwargs*	Der Rest der Schlüsselwortargumente

Bemerkungen

5 grundlegende Dinge, die Sie mit Funktionen machen können:

- Weisen Sie den Variablen Funktionen zu

```
def f():
  print(20)
y = f
y()
# Output: 20
```

- Funktionen innerhalb anderer Funktionen definieren (verschachtelte Funktionen)

```
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
```

- Funktionen können andere Funktionen zurückgeben

```
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
```

- Funktionen können als Parameter an andere Funktionen übergeben werden

```
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
```

- Innere Funktionen haben Zugriff auf den umschließenden Bereich (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!
```

Zusätzliche Ressourcen

- Mehr über Funktionen und Dekorateure: https://www.thecodeship.com/patterns/guide-to-python-function-decorators/

Examples

Einfache Funktionen definieren und aufrufen

Die `def` Anweisung ist die gebräuchlichste Methode zum Definieren einer Funktion in Python. Diese Anweisung ist eine sogenannte *Einzelklausel-Verbundanweisung* mit der folgenden Syntax:

```
def function_name(parameters):
    statement(s)
```

`function_name` ist als *Bezeichner* der Funktion bekannt. Da eine Funktionsdefinition eine ausführbare Anweisung ist, *bindet* ihre Ausführung den Funktionsnamen an das Funktionsobjekt,

das später mit dem Bezeichner aufgerufen werden kann.

`parameters` ist eine optionale Liste von Bezeichnern, die beim Aufruf der Funktion an die als Argumente angegebenen Werte gebunden werden. Eine Funktion kann eine beliebige Anzahl von Argumenten haben, die durch Kommas getrennt sind.

`statement(s)` - auch *Funktionskörper genannt* - sind eine nicht leere Folge von Anweisungen, die bei jedem Aufruf der Funktion ausgeführt werden. Dies bedeutet, dass ein Funktionskörper nicht wie ein *eingerückter Block* leer sein darf.

Hier ist ein Beispiel für eine einfache Funktionsdefinition, die dazu dient, `Hello` jedem Aufruf zu drucken:

```
def greet():
    print("Hello")
```

Rufen wir nun die definierte Funktion `greet()` :

```
greet()
# Out: Hello
```

Dies ist ein weiteres Beispiel für eine Funktionsdefinition, die ein einzelnes Argument verwendet und bei jedem Aufruf der Funktion den übergebenen Wert anzeigt:

```
def greet_two(greeting):
    print(greeting)
```

Danach muss die Funktion `greet_two()` mit einem Argument aufgerufen werden:

```
greet_two("Howdy")
# Out: Howdy
```

Sie können diesem Funktionsargument auch einen Standardwert zuweisen:

```
def greet_two(greeting="Howdy"):
    print(greeting)
```

Jetzt können Sie die Funktion aufrufen, ohne einen Wert anzugeben:

```
greet_two()
# Out: Howdy
```

Sie werden feststellen, dass Sie im Gegensatz zu vielen anderen Sprachen keinen Rückgabetyp der Funktion explizit angeben müssen. Python-Funktionen können über das Schlüsselwort return Werte eines beliebigen Typs `return` . Eine Funktion kann eine beliebige Anzahl verschiedener Typen zurückgeben!

```
def many_types(x):
    if x < 0:
```

```
        return "Hello!"
    else:
        return 0

print(many_types(1))
print(many_types(-1))

# Output:
0
Hello!
```

Solange dies vom Aufrufer korrekt gehandhabt wird, ist dies ein absolut gültiger Python-Code.

Eine Funktion, die das Ende der Ausführung ohne return-Anweisung erreicht, gibt immer `None` .

```
def do_nothing():
    pass

print(do_nothing())
# Out: None
```

Wie bereits erwähnt, muss eine Funktionsdefinition einen Funktionskörper haben, eine nicht leere Anweisungsfolge. Daher wird die `pass` Anweisung als Funktionshauptteil verwendet. Dies ist eine Nulloperation - wenn sie ausgeführt wird, passiert nichts. Es macht was es bedeutet, es überspringt. Es ist nützlich als Platzhalter, wenn eine Anweisung syntaktisch erforderlich ist, jedoch kein Code ausgeführt werden muss.

Werte von Funktionen zurückgeben

Funktionen können `return` einen Wert, den Sie direkt verwenden können:

```
def give_me_five():
    return 5

print(give_me_five())  # Print the returned value
# Out: 5
```

oder speichern Sie den Wert zur späteren Verwendung:

```
num = give_me_five()
print(num)             # Print the saved returned value
# Out: 5
```

oder verwenden Sie den Wert für alle Operationen:

```
print(give_me_five() + 10)
# Out: 15
```

Wenn in der Funktion " `return` auftritt, wird die Funktion sofort verlassen und nachfolgende Operationen werden nicht ausgewertet:

```
def give_me_another_five():
    return 5
    print('This statement will not be printed. Ever.')

print(give_me_another_five())
# Out: 5
```

Sie können auch `return` mehrere Werte (in Form eines Tupels):

```
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
```

Eine Funktion *ohne* `return` Anweisung gibt implizit `None` . Ebenso wird eine Funktion mit einer `return` - Anweisung, aber kein Rückgabewert oder Variable kehrt `None` .

Funktion mit Argumenten definieren

Argumente werden in Klammern hinter dem Funktionsnamen definiert:

```
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)
```

Der Funktionsname und seine Liste der Argumente werden als *Signatur* der Funktion bezeichnet. Jedes benannte Argument ist effektiv eine lokale Variable der Funktion.

Geben Sie beim Aufrufen der Funktion Werte für die Argumente an, indem Sie sie nacheinander auflisten

```
divide(10, 2)
# output: 5
```

oder spezifizieren Sie sie in beliebiger Reihenfolge anhand der Namen aus der Funktionsdefinition:

```
divide(divisor=2, dividend=10)
# output: 5
```

Definieren einer Funktion mit optionalen Argumenten

Optionale Argumente können definiert werden, indem dem Argumentnamen (mit `=`) ein Standardwert zugewiesen wird:

```
def make(action='nothing'):
    return action
```

Der Aufruf dieser Funktion ist auf drei verschiedene Arten möglich:

```
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
```

Warnung

Veränderbare Typen (`list` , `dict` , `set` usw.) sollten sorgfältig behandelt werden, wenn sie als **Standardattribut angegeben** werden. Jede Änderung des Standardarguments ändert es dauerhaft. Siehe Definieren einer Funktion mit optionalen veränderbaren Argumenten .

Funktion mit mehreren Argumenten definieren

Man kann einer Funktion so viele Argumente geben, wie man möchte, die einzigen festen Regeln sind, dass jeder Argumentname eindeutig sein muss und dass optionale Argumente hinter den nicht-optionalen Argumenten stehen müssen:

```
def func(value1, value2, optionalvalue=10):
    return '{0} {1} {2}'.format(value1, value2, optionalvalue1)
```

Beim Aufruf der Funktion können Sie entweder jedes Schlüsselwort ohne Namen angeben, aber dann ist die Reihenfolge von Bedeutung:

```
print(func(1, 'a', 100))
# Out: 1 a 100

print(func('abc', 14))
# abc 14 10
```

Oder kombinieren Sie die Argumente mit Namen und außerhalb. Dann müssen diejenigen mit Namen denen ohne Namen folgen, aber die Reihenfolge der Namen mit Namen spielt keine Rolle:

```
print(func('This', optionalvalue='StackOverflow Documentation', value2='is'))
# Out: This is StackOverflow Documentation
```

Definieren einer Funktion mit einer beliebigen Anzahl von Argumenten

Beliebige Anzahl von Positionsargumenten:

Das Definieren einer Funktion, die eine beliebige Anzahl von Argumenten annehmen kann, kann durch Anfügen eines der Argumente mit einem `*`

```
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
```

Sie **können keine** Standardwerte für `args` , z. B. führt `func(*args=[1, 2, 3])` zu einem Syntaxfehler (wird nicht kompiliert).

Sie **können** diese nicht beim Namen `TypeError` , `TypeError` `func(*args=[1, 2, 3])` einen `TypeError` .

Aber wenn Sie bereits Ihre Argumente in einem Array (oder jede andere `Iterable` :), **können** Sie Ihre Funktion wie folgt aufrufen `func(*my_stuff)` .

Auf diese Argumente (`*args`) kann über einen Index zugegriffen werden. Beispielsweise gibt `args[0]` das erste Argument zurück

Beliebige Anzahl von Schlüsselwortargumenten

Sie können eine beliebige Anzahl von Argumenten mit einem Namen verwenden, indem Sie ein Argument in der Definition mit **zwei** `*` :

```
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
```

Sie **können** diese Namen **nicht ohne** Namen `TypeError` , z. B. wird durch `func(1, 2, 3)` ein `TypeError` .

`kwargs` ist ein einfaches Python-Wörterbuch. Beispielsweise gibt `args['value1']` den Wert für das Argument `value1` . Vergewissern Sie sich vorher, dass ein solches Argument `KeyError` oder ein `KeyError` wird `KeyError` .

Warnung

Sie können diese mit anderen optionalen und erforderlichen Argumenten mischen, aber die Reihenfolge innerhalb der Definition ist wichtig.

Die **positionellen / Keyword-** Argumente stehen an erster Stelle. (Erforderliche Argumente).
Dann kommt die **willkürlichen** `*arg` Argumente. (Wahlweise).
Als Nächstes kommen **nur ausschließliche Keyword-** Argumente. (Erforderlich).
Schließlich kommt das **beliebige Stichwort** `**kwargs` . (Wahlweise).

```
#       |-positional-|-optional-|---keyword-only--|-optional-|
def func(arg1, arg2=10 , *args, kwarg1, kwarg2=2, **kwargs):
     pass
```

- `arg1` muss angegeben werden, andernfalls wird ein `TypeError` . Es kann als positionelles (`func(10)`) oder Schlüsselwortargument `func(arg1=10)`) angegeben werden.
- `kwarg1` muss ebenfalls angegeben werden, es kann jedoch nur als Schlüsselwortargument angegeben werden: `func(kwarg1=10)` .
- `arg2` und `kwarg2` sind optional. Wenn der Wert geändert werden soll, gelten die gleichen Regeln wie für `arg1` (entweder positional oder Schlüsselwort) und `kwarg1` (nur Schlüsselwort).
- `*args` fängt zusätzliche Positionsparameter ab. Beachten Sie jedoch, dass `arg1` und `arg2` als Positionsargumente `arg2` müssen, um Argumente an `*args` : `func(1, 1, 1, 1)` .
- `**kwargs` alle zusätzlichen Schlüsselwortparameter. In diesem Fall ein Parameter, der nicht `arg1` , `arg2` , `kwarg1` oder `kwarg2` . Zum Beispiel: `func(kwarg3=10)` .
- In Python 3 können Sie mit `*` allein angeben, dass alle nachfolgenden Argumente als Schlüsselwörter angegeben werden müssen. Zum Beispiel wird die `math.isclose` Funktion in Python 3.5 und höher mit `def math.isclose (a, b, *, rel_tol=1e-09, abs_tol=0.0)` definiert `def math.isclose (a, b, *, rel_tol=1e-09, abs_tol=0.0)` , was bedeutet, dass die ersten beiden Argumente positionell angegeben werden können, die optionalen jedoch Der dritte und der vierte Parameter können nur als Schlüsselwortargumente angegeben werden.

Python 2.x unterstützt keine reinen Keyword-Parameter. Dieses Verhalten kann mit `kwargs` emuliert `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 ...
```

Hinweis zum Benennen

Die Konvention der optionalen Positionsargumente Benennung `args` und optionale Schlüsselwort Argumente `kwargs` ist nur eine Konvention Sie keine Namen verwenden , **die** Sie mögen , **aber** es ist nützlich , um die Konvention zu folgen , damit andere wissen , was Sie tun, *oder sich auch später* so bitte tun.

Hinweis zur Einzigartigkeit

Jede Funktion kann mit **keiner oder einer** `*args` und **keiner oder einer** `**kwargs` jedoch nicht mit mehr als einer davon. `*args` **muss** auch das letzte Positionsargument sein und `**kwargs` muss der letzte Parameter sein. Wenn Sie versuchen, mehr als eine von beiden zu verwenden, **führt** dies zu einer Syntaxfehler-Ausnahme.

Hinweis zu Verschachtelungsfunktionen mit optionalen Argumenten

Es ist möglich, Nest solche Funktionen und die übliche Konvention ist es, die Elemente zu entfernen , dass der Code bereits behandelt hat , **aber** wenn Sie die Parameter durch die Über werden müssen Sie optionale Positions args mit einem passieren `*` Präfix und optionale Schlüsselwort args mit einem `**` Präfix , andernfalls werden args with als Liste oder Tupel und kwargs als einzelnes Wörterbuch übergeben. z.B:

```
def fn(**kwargs):
    print(kwargs)
    f1(**kwargs)

def f1(**kwargs):
    print(len(kwargs))

fn(a=1, b=2)
# Out:
# {'a': 1, 'b': 2}
# 2
```

Definieren einer Funktion mit optionalen veränderbaren Argumenten

Es gibt ein Problem bei der Verwendung **optionaler Argumente** mit einem **veränderbaren Standardtyp** (beschrieben in Funktion mit optionalen Argumenten definieren), die möglicherweise zu unerwartetem Verhalten führen.

Erläuterung

Dieses Problem tritt auf, weil die Standardargumente einer Funktion **einmalig** an dem Punkt initialisiert **werden** , an dem die Funktion *definiert ist* , und **nicht** (wie viele andere Sprachen), wenn die Funktion *aufgerufen wird* . Die Standardwerte werden in der `__defaults__` des Funktionsobjekts gespeichert.

```
def f(a, b=42, c=[]):
    pass

print(f.__defaults__)
# Out: (42, [])
```

Für **unveränderliche** Typen (siehe Argumentübergabe und Mutabilität) ist dies kein Problem, da die Variable nicht mutiert werden kann. es kann immer nur neu zugewiesen werden, wobei der ursprüngliche Wert unverändert bleibt. Daher ist garantiert, dass Folger den gleichen Standardwert hat. Bei einem **veränderbaren** Typ kann der ursprüngliche Wert jedoch geändert werden, indem die verschiedenen Memberfunktionen aufgerufen werden. Bei aufeinanderfolgenden Aufrufen der Funktion wird daher nicht garantiert, dass sie den ursprünglichen Standardwert haben.

```
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]
```

> **Hinweis:** Einige IDEs wie PyCharm geben eine Warnung aus, wenn ein veränderlicher Typ als Standardattribut angegeben wird.

Lösung

Wenn Sie sicherstellen möchten, dass das Standardargument immer das von Ihnen in der Funktionsdefinition angegebene ist, verwenden Sie als Standardargument **immer** einen unveränderlichen Typ.

Um dies zu erreichen, wenn ein veränderlicher Typ als Standard benötigt wird, verwenden Sie `None` (unveränderlich) als Standardargument und weisen dann der Argumentvariablen den tatsächlichen Standardwert zu, wenn dieser mit `None` übereinstimmt.

```
def append(elem, to=None):
    if to is None:
        to = []

    to.append(elem)
    return to
```

Lambda (Inline / Anonym) Funktionen

Das Schlüsselwort `lambda` erstellt eine Inline-Funktion, die einen einzelnen Ausdruck enthält. Der Wert dieses Ausdrucks gibt die Funktion zurück, wenn sie aufgerufen wird.

Betrachten Sie die Funktion:

```
def greeting():
    return "Hello"
```

welche, wenn aufgerufen als:

```
print(greeting())
```

druckt:

```
Hello
```

Dies kann wie folgt als Lambda-Funktion geschrieben werden:

```
greet_me = lambda: "Hello"
```

> Beachten Sie den Hinweis am Ende dieses Abschnitts bezüglich der Zuordnung von Lambdas zu Variablen. Tun Sie es im Allgemeinen nicht.

Dadurch wird eine Inline-Funktion mit dem Namen `greet_me` , die `Hello` zurückgibt. Beachten Sie, dass Sie beim Erstellen einer Funktion mit Lambda kein `return` schreiben. Der Wert nach `:` wird automatisch zurückgegeben.

Sobald eine Variable zugewiesen wurde, kann sie wie eine reguläre Funktion verwendet werden:

```
print(greet_me())
```

druckt:

```
Hello
```

`lambda` s kann auch Argumente annehmen:

```
strip_and_upper_case = lambda s: s.strip().upper()

strip_and_upper_case("  Hello   ")
```

gibt den String zurück:

```
HELLO
```

Sie können ebenso wie normale Funktionen eine beliebige Anzahl von Argumenten / Schlüsselwortargumenten annehmen.

```
greeting = lambda x, *args, **kwargs: print(x, args, kwargs)
greeting('hello', 'world', world='world')
```

druckt:

```
hello ('world',) {'world': 'world'}
```

`lambda` Werte werden üblicherweise für kurze Funktionen verwendet, die sich an dem Punkt definieren lassen, an dem sie aufgerufen werden (normalerweise mit `sorted` , `filter` und `map`).

In dieser Zeile wird beispielsweise eine Liste von Zeichenfolgen sortiert, wobei der Groß- und Kleinschreibung am Anfang und am Ende ignoriert wird:

```
sorted( [" foo ", "    bAR", "BaZ    "], key=lambda s: s.strip().upper())
# Out:
# ['    bAR', 'BaZ    ', ' foo ']
```

Sortierliste, wobei Whitespaces ignoriert werden:

```
sorted( [" foo ", "    bAR", "BaZ    "], key=lambda s: s.strip())
# Out:
# ['BaZ    ', '    bAR', ' foo ']
```

Beispiele mit `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']
```

Beispiele mit numerischen Listen:

```
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]
```

Andere Funktionen (mit / ohne Argumente) können aus einer Lambda-Funktion heraus aufgerufen werden.

```
def foo(msg):
    print(msg)

greet = lambda x = "hello world": foo(x)
greet()
```

druckt:

```
hello world
```

Dies ist nützlich, da `lambda` möglicherweise nur einen Ausdruck enthält und durch die Verwendung einer untergeordneten Funktion mehrere Anweisungen ausgeführt werden können.

HINWEIS

Beachten Sie, dass PEP-8 (der offizielle Python-Styleguide) nicht empfiehlt, Variablen Lambdas zuzuweisen (wie in den ersten beiden Beispielen):

> Verwenden Sie immer eine def-Anweisung anstelle einer Zuweisungsanweisung, die einen Lambda-Ausdruck direkt an einen Bezeichner bindet.
>
> Ja:
>
> ```
> def f(x): return 2*x
> ```
>
> Nein:
>
> ```
> f = lambda x: 2*x
> ```
>
> Die erste Form bedeutet, dass der Name des resultierenden Funktionsobjekts spezifisch `f` anstelle des generischen `<lambda>` . Dies ist im Allgemeinen für Tracebacks und Stringdarstellungen sinnvoller. Die Verwendung der Zuweisungsanweisung eliminiert den einzigen Vorteil, den ein Lambda-Ausdruck gegenüber einer expliziten `def` Anweisung bieten kann (dh, er kann in einen größeren Ausdruck eingebettet werden).

Argumentübergabe und Veränderlichkeit

Zunächst einige Begriffe:

- **Argument (*Aktualparameter*):** Die tatsächliche Variable, die an eine Funktion übergeben wird.
- **Parameter *(formal* parameter):** der empfangende Variable, die in einer Funktion verwendet wird.

In Python werden Argumente per *Zuweisung übergeben* (im Gegensatz zu anderen Sprachen, in denen Argumente per Wert / Referenz / Zeiger übergeben werden können).

- Durch die Mutation eines Parameters wird das Argument mutiert (wenn der Typ des Arguments variabel ist).

```
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
```

- Durch die Neuzuweisung des Parameters wird das Argument nicht erneut zugewiesen.

```
def foo(x):         # here x is the parameter, when we call foo(y) we assign y to x
    x[0] = 9        # This mutates the list labelled by both x and y
    x = [1, 2, 3]  # x is now labeling a different list (y is unaffected)
    x[2] = 8        # This mutates x's list, not y's list

y = [4, 5, 6]       # y is the argument, x is the parameter
foo(y)              # Pretend that we wrote "x = y", then go to line 1
y
# Out: [9, 5, 6]
```

In Python weisen wir Variablen keine Werte zu, stattdessen *binden* (dh Namen zuweisen) Variablen (als *Namen betrachtet*) an Objekte.

- **Unveränderlich:** Ganzzahlen, Strings, Tupel usw. Alle Operationen machen Kopien.
- **Veränderbar:** Listen, Wörterbücher, Sets usw. Operationen können mutieren oder nicht.

```
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]
```

Schließung

Schließungen in Python werden durch Funktionsaufrufe erstellt. Der Aufruf von `makeInc` erstellt hier eine Bindung für `x` , auf die in der Funktion `inc` verwiesen wird. Jeder Aufruf von `makeInc` erstellt eine neue Instanz dieser Funktion, aber jede Instanz hat einen Link zu einer anderen Bindung von `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
```

Beachten Sie, dass während eines regulären Abschlusses die eingeschlossene Funktion alle Variablen vollständig von ihrer umgebenden Umgebung erbt. In diesem Konstrukt hat die eingeschlossene Funktion nur Lesezugriff auf die geerbten Variablen, kann ihnen jedoch keine Zuweisungen vornehmen

```
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 bietet die `nonlocal` Anweisung (Nonlocal Variables) zum Realisieren eines vollständigen Schließens mit verschachtelten Funktionen.

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
```

Rekursive Funktionen

Eine rekursive Funktion ist eine Funktion, die sich in ihrer Definition aufruft. Zum Beispiel die mathematische Funktion Faktorial, definiert durch `factorial(n) = n*(n-1)*(n-2)*...*3*2*1` . kann als programmiert werden

```
def factorial(n):
    #n here should be an integer
    if n == 0:
        return 1
    else:
        return n*factorial(n-1)
```

Die Ausgänge hier sind:

```
factorial(0)
#out 1
factorial(1)
#out 1
factorial(2)
#out 2
factorial(3)
#out 6
```

wie erwartet. Beachten Sie, dass diese Funktion rekursiv ist, da die zweite `return factorial(n-1)` die Funktion in ihrer Definition aufruft.

Einige rekursive Funktionen können mit Lambda implementiert werden, die faktorielle Funktion mit Lambda wäre etwa so:

```
factorial = lambda n: 1 if n == 0 else n*factorial(n-1)
```

Die Funktion wird wie oben ausgegeben.

Rekursionslimit

Die Tiefe der möglichen Rekursion ist begrenzt, abhängig von der Python-Implementierung. Wenn der Grenzwert erreicht ist, wird eine RuntimeError-Ausnahme ausgelöst:

```
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!
```

Sie können den Grenzwert für die Rekursionstiefe mithilfe von `sys.setrecursionlimit(limit)` ändern und diesen Grenzwert mit `sys.getrecursionlimit()` überprüfen.

```
sys.setrecursionlimit(2000)
cursing(0)
# Out: I recursed 1997 times!
```

Bei Python 3.5 ist die Ausnahme ein `RecursionError` , der von `RuntimeError` abgeleitet `RuntimeError` .

Verschachtelte Funktionen

Funktionen in Python sind erstklassige Objekte. Sie können in beliebigem Umfang definiert werden

```
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
```

Funktionen, die ihren einschließenden Bereich erfassen, können wie andere Objekte weitergegeben werden

```
def make_adder(n):
    def adder(x):
        return n + x
    return adder
add5 = make_adder(5)
add6 = make_adder(6)
add5(10)
#Out: 15
add6(10)
#Out: 16

def repeatedly_apply(func, n, x):
    for i in range(n):
        x = func(x)
    return x

repeatedly_apply(add5, 5, 1)
#Out: 26
```

Iterable und Wörterbuch auspacken

Mit Funktionen können Sie folgende Arten von Parametern angeben: positional, named, variable positional, Keyword args (kwargs). Hier ist eine klare und prägnante Verwendung jedes Typs.

```
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'
```

Erzwingen die Verwendung benannter Parameter

Alle Parameter, die nach dem ersten Stern in der Funktionssignatur angegeben werden, sind nur ein Schlüsselwort.

```
def f(*a, b):
    pass

f(1, 2, 3)
# TypeError: f() missing 1 required keyword-only argument: 'b'
```

In Python 3 ist es möglich, ein einzelnes Sternchen in die Funktionssignatur einzufügen, um sicherzustellen, dass die verbleibenden Argumente nur mit Schlüsselwortargumenten übergeben werden.

```
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
```

Rekursives Lambda mit zugewiesener Variable

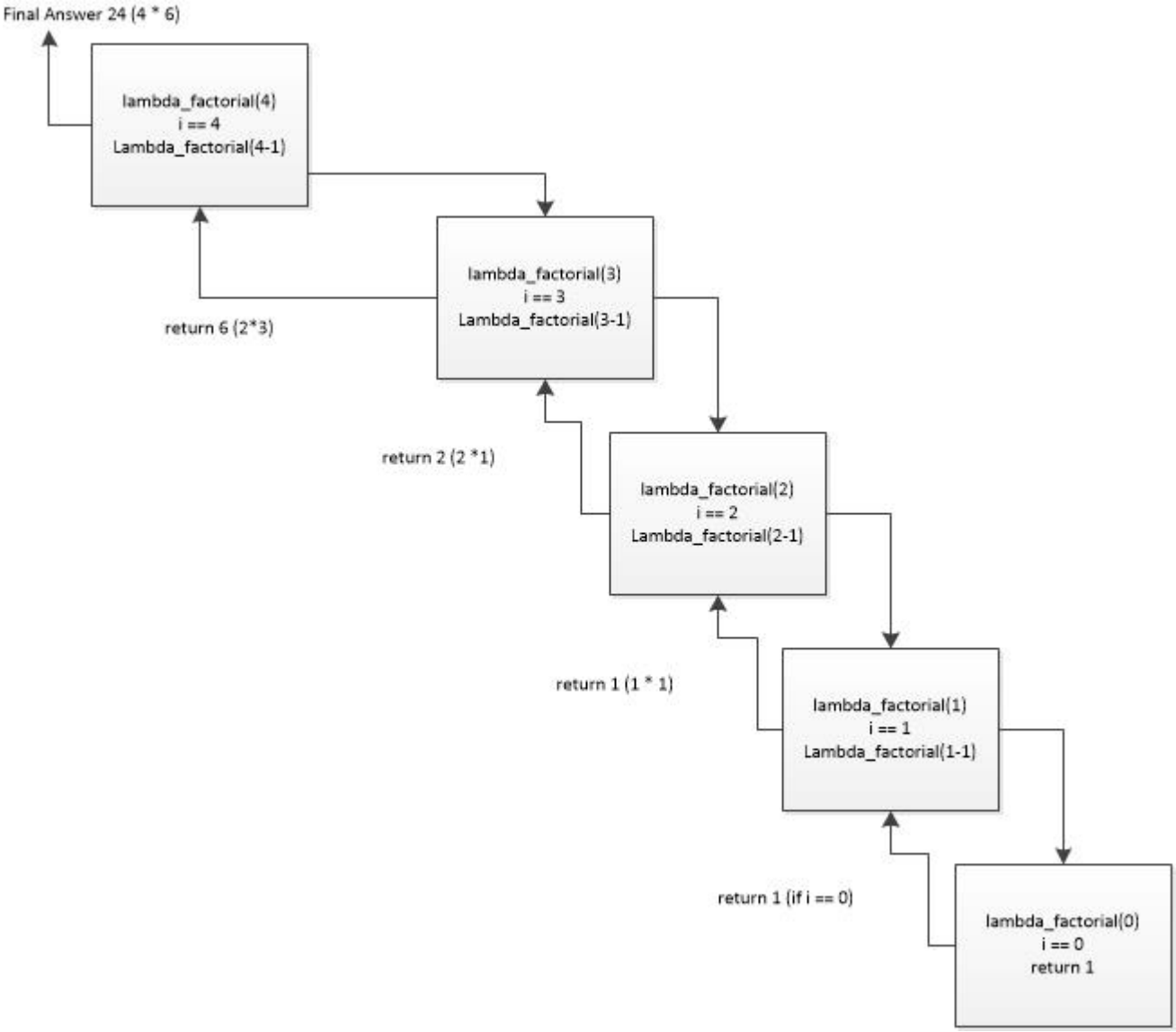
Eine Methode zum Erstellen rekursiver Lambda-Funktionen umfasst das Zuweisen der Funktion zu einer Variablen und das anschließende Referenzieren dieser Variablen innerhalb der Funktion selbst. Ein häufiges Beispiel hierfür ist die rekursive Berechnung der Fakultät einer Zahl - wie im

folgenden Code dargestellt:

```
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
```

Beschreibung des Codes

Die Lambda - Funktion, durch seine variable Zuordnung wird ein Wert (4) übergeben , die sie auswertet und gibt 1 zurück , wenn es 0 ist , sonst gibt er den aktuellen Wert (`i`) * eine andere Berechnung durch die Lambda - Funktion des Wertes - 1 (`i-1`). Dies wird fortgesetzt, bis der übergebene Wert auf 0 dekrementiert ist (`return 1`). Ein Prozess, der visualisiert werden kann als:



Funktionen online lesen: https://riptutorial.com/de/python/topic/228/funktionen

Kapitel 69: Funktionen mit Listenargumenten definieren

Examples

Funktion und Anruf

Listen als Argumente sind nur eine weitere Variable:

```
def func(myList):
    for item in myList:
        print(item)
```

und kann im Funktionsaufruf selbst übergeben werden:

```
func([1,2,3,5,7])

1
2
3
5
7
```

Oder als Variable:

```
aList = ['a','b','c','d']
func(aList)

a
b
c
d
```

Funktionen mit Listenargumenten definieren online lesen:
https://riptutorial.com/de/python/topic/7744/funktionen-mit-listenargumenten-definieren

Kapitel 70: Generatoren

Einführung

Generatoren sind faule Iteratoren, die von Generatorfunktionen (mit `yield`) oder Generatorausdrücken (mit `(an_expression for x in an_iterator)`) erstellt werden.

Syntax

- Ausbeute `<expr>`
- Ertrag von `<expr>`
- `<var>` = Ertrag `<expr>`
- next (`<iter>`)

Examples

Iteration

Ein Generatorobjekt unterstützt das *Iteratorprotokoll* . Das heißt, es stellt eine `next()` Methode (`__next__()` in Python 3.x) `__next__()` , die zum schrittweisen Durchlaufen der Ausführung verwendet wird, und ihre `__iter__` Methode gibt sich selbst zurück. Dies bedeutet, dass ein Generator in einem beliebigen Sprachkonstrukt verwendet werden kann, das generische iterierbare Objekte unterstützt.

```
# 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]
```

Die nächste () Funktion

Das `next()` eingebaute ist ein praktischer Wrapper, der verwendet werden kann, um einen Wert von einem beliebigen Iterator (einschließlich eines Generator-Iterators) zu empfangen und einen Standardwert anzugeben, falls der Iterator erschöpft ist.

```
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
# ...
```

Die Syntax ist `next(iterator[, default])` . Wenn der Iterator endet und ein Standardwert übergeben wurde, wird er zurückgegeben. Wenn kein Standardwert angegeben wurde, wird `StopIteration` .

Objekte an einen Generator senden

Zusätzlich zum Empfangen von Werten von einem Generator ist es möglich , ein Objekt mit der `send()` -Methode an einen Generator zu `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
```

Was hier passiert, ist folgendes:

- Wenn Sie zum ersten Mal `next(generator)` aufrufen, `next(generator)` das Programm zur ersten `yield` Anweisung und gibt den Wert von `total` an diesem Punkt zurück, dh 0. Die Ausführung des Generators wird an diesem Punkt ausgesetzt.
- Wenn Sie dann `generator.send(x)` aufrufen, übernimmt der Interpreter das Argument `x` und macht es zum Rückgabewert der letzten `yield` Anweisung, die dem `value` zugewiesen `value` . Der Generator fährt dann wie gewohnt fort, bis er den nächsten Wert ergibt.
- Wenn Sie schließlich `next(generator)` aufrufen, behandelt das Programm dies so, als würden

Sie `None` an den Generator senden. `None` ist nichts Besonderes. In diesem Beispiel wird jedoch `None` als spezieller Wert verwendet, um den Generator zum Anhalten aufzufordern.

Generatorausdrücke

Es ist möglich, Generator-Iteratoren mit einer verständnisartigen Syntax zu erstellen.

```
generator = (i * 2 for i in range(3))

next(generator)  # 0
next(generator)  # 2
next(generator)  # 4
next(generator)  # raises StopIteration
```

Wenn einer Funktion nicht unbedingt eine Liste übergeben werden muss, können Sie Zeichen speichern (und die Lesbarkeit verbessern), indem Sie einen Generatorausdruck in einen Funktionsaufruf einfügen. Die Klammer aus dem Funktionsaufruf macht Ihren Ausdruck implizit zu einem Generatorausdruck.

```
sum(i ** 2 for i in range(4))  # 0^2 + 1^2 + 2^2 + 3^2 = 0 + 1 + 4 + 9 = 14
```

Darüber hinaus sparen Sie Speicherplatz, da Sie nicht die gesamte Liste laden, über die Sie iterieren (`[0, 1, 2, 3]` im obigen Beispiel), der Generator die Verwendung von Werten durch Python zulässt.

Einführung

Generatorausdrücke ähneln Listen-, Wörterbuch- und Mengenverstehen, sind jedoch in Klammern eingeschlossen. Die Klammern müssen nicht vorhanden sein, wenn sie als einziges Argument für einen Funktionsaufruf verwendet werden.

```
expression = (x**2 for x in range(10))
```

In diesem Beispiel werden die 10 ersten perfekten Quadrate generiert, einschließlich 0 (wobei x = 0 ist).

Generatorfunktionen sind regulären Funktionen ähnlich, außer dass sie eine oder mehrere `yield` in ihrem Körper haben. Solche Funktionen können nicht `return` alle Werte (jedoch leer `return` s erlaubt sind , wenn Sie den Generator früh stoppen wollen).

```
def function():
    for x in range(10):
        yield x**2
```

Diese Generatorfunktion entspricht dem vorherigen Generatorausdruck und gibt denselben aus.

Hinweis : Alle Generatorausdrücke haben ihre eigenen *äquivalenten* Funktionen, nicht jedoch umgekehrt.

Ein Generatorausdruck kann ohne Klammern verwendet werden, wenn sich sonst beide Klammern wiederholen würden:

```
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'>
```

Anstatt:

```
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))
```

Aber nicht:

```
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)
```

Beim Aufruf einer Generatorfunktion wird ein **Generatorobjekt** erzeugt, das später wiederholt werden kann. Im Gegensatz zu anderen Iteratortypen können Generatorobjekte nur einmal durchlaufen werden.

```
g1 = function()
print(g1)  # Out: <generator object function at 0x1012e1888>
```

Beachten Sie, dass der Rumpf eines Generators **nicht** sofort ausgeführt wird: Wenn Sie `function()` im obigen Beispiel aufrufen, gibt er sofort ein Generatorobjekt zurück, ohne die erste Druckanweisung auszuführen. Dadurch können Generatoren weniger Speicher verbrauchen als Funktionen, die eine Liste zurückgeben. Außerdem können Generatoren erstellt werden, die unendlich lange Sequenzen erzeugen.

Aus diesem Grund werden Generatoren häufig in der Datenwissenschaft und in anderen Kontexten mit großen Datenmengen verwendet. Ein weiterer Vorteil ist, dass anderer Code die von einem Generator ausgegebenen Werte sofort verwenden kann, ohne auf die Erstellung der vollständigen Sequenz zu warten.

Wenn Sie jedoch die von einem Generator erzeugten Werte mehr als einmal verwenden müssen und die Erzeugung dieser Werte mehr kostet als das Speichern, ist es möglicherweise besser, die ermittelten Werte als `list` zu speichern, als die Sequenz neu zu generieren. Weitere Informationen finden Sie unten unter 'Generator zurücksetzen'.

In der Regel wird ein Generatorobjekt in einer Schleife oder in einer Funktion verwendet, die eine Iteration erfordert:

```
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]
```

Da Generatorobjekte Iteratoren sind, kann man sie mit der `next()` Funktion manuell durchlaufen. Dadurch werden die ermittelten Werte bei jedem nachfolgenden Aufruf einzeln zurückgegeben.

Unter der Haube führt Python bei jedem Aufruf von `next()` in einem Generator Anweisungen im Rumpf der Generatorfunktion aus, bis die nächste `yield` . An diesem Punkt wird das Argument des `yield` und der Punkt, an dem dies geschehen ist, gespeichert. Wenn Sie `next()` erneut aufrufen, wird die Ausführung ab diesem Punkt fortgesetzt und bis zur nächsten `yield` Anweisung fortgesetzt.

Wenn Python das Ende der Generatorfunktion erreicht, ohne weitere `yield` , wird eine `StopIteration` Ausnahme `StopIteration` (dies ist normal, alle Iteratoren verhalten sich auf dieselbe Weise).

```
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
```

Beachten Sie, dass Generatorobjekte in Python 2 über `.next()` -Methoden verfügten, mit denen die ermittelten Werte manuell durchlaufen werden konnten. In Python 3 wurde diese Methode für alle Iteratoren durch den Standard `.__next__()` .

Generator zurücksetzen

Denken Sie daran , dass Sie nur durch die Objekte , die von einem Generator *einmal* laufen können. Wenn Sie die Objekte in einem Skript bereits durchlaufen haben, führt jeder weitere Versuch zu `None` .

Wenn Sie die von einem Generator generierten Objekte mehr als einmal verwenden müssen, können Sie entweder die Generatorfunktion erneut definieren und ein zweites Mal verwenden, oder Sie können die Ausgabe der Generatorfunktion bei der ersten Verwendung in einer Liste speichern. Eine erneute Definition der Generatorfunktion ist eine gute Option, wenn Sie mit großen Datenmengen arbeiten und das Speichern einer Liste aller Datenelemente viel Speicherplatz beanspruchen würde. Umgekehrt, wenn die anfängliche Generierung der Elemente

teuer ist, können Sie es vorziehen, die generierten Elemente in einer Liste zu speichern, damit Sie sie wiederverwenden können.

Verwenden eines Generators, um Fibonacci-Nummern zu finden

Ein praktischer Anwendungsfall eines Generators besteht darin, Werte einer unendlichen Reihe zu durchlaufen. Hier ein Beispiel für das Finden der ersten zehn Terme der Fibonacci-Sequenz .

```
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

Unendliche Sequenzen

Generatoren können unendliche Sequenzen darstellen:

```
def integers_starting_from(n):
    while True:
        yield n
        n += 1

natural_numbers = integers_starting_from(1)
```

Eine unendliche Zahlenfolge wie oben kann auch mit Hilfe von `itertools.count` . Der obige Code könnte wie folgt geschrieben werden

```
natural_numbers = itertools.count(1)
```

Sie können Generatoren-Verständnis für unendliche Generatoren verwenden, um neue Generatoren herzustellen:

```
multiples_of_two = (x * 2 for x in natural_numbers)
multiples_of_three = (x for x in natural_numbers if x % 3 == 0)
```

Beachten Sie, dass ein unendlicher Generator kein Ende hat. Wenn Sie ihn also an eine Funktion übergeben, die versucht, den Generator vollständig zu verbrauchen, hat dies **schwerwiegende Folgen** :

```
list(multiples_of_two)  # will never terminate, or raise an OS-specific error
```

Verwenden Sie stattdessen List / Set- `xrange` mit `range` (oder `xrange` für Python <3.0):

```
first_five_multiples_of_three = [next(multiples_of_three) for _ in range(5)]
```

```
# [3, 6, 9, 12, 15]
```

oder verwenden Sie `itertools.islice()` , um den Iterator in eine Teilmenge zu schneiden:

```
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]
```

Beachten Sie, dass der Originalgenerator ebenso wie alle anderen Generatoren, die von demselben "Root" stammen, aktualisiert wird:

```
next(natural_numbers)    # yields 16
next(multiples_of_two)   # yields 34
next(multiples_of_four)  # yields 24
```

Eine unendliche Sequenz kann auch mit einer `for` Schleife durchlaufen werden . Stellen Sie sicher, dass Sie eine bedingte `break` Anweisung einfügen, damit die Schleife eventuell beendet wird:

```
for idx, number in enumerate(multiplies_of_two):
    print(number)
    if idx == 9:
        break  # stop after taking the first 10 multiplies of two
```

Klassisches Beispiel - Fibonacci-Zahlen

```
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
```

Alle Werte aus einem anderen iterierbar

Python 3.x 3.3

Verwenden Sie die `yield from` wenn Sie alle Werte aus einer anderen Iteration ermitteln wollen:

```
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]
```

Dies funktioniert auch mit Generatoren.

```
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]
```

Coroutinen

Generatoren können verwendet werden, um Coroutinen zu implementieren:

```
# 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 werden häufig zum Implementieren von Zustandsmaschinen verwendet, da sie in erster Linie nützlich sind, um Prozeduren mit einer Methode zu erstellen, für die ein Zustand erforderlich ist, um ordnungsgemäß zu funktionieren. Sie bearbeiten einen vorhandenen Zustand und geben den nach Abschluss des Vorgangs erhaltenen Wert zurück.

Rendite mit Rekursion: Alle Dateien in einem Verzeichnis werden rekursiv aufgelistet

Importieren Sie zunächst die Bibliotheken, die mit Dateien arbeiten:

```
from os import listdir
from os.path import isfile, join, exists
```

Eine Hilfsfunktion, um nur Dateien aus einem Verzeichnis zu lesen:

```
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
```

Eine weitere Hilfsfunktion, um nur die Unterverzeichnisse zu erhalten:

```
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
```

Verwenden Sie nun diese Funktionen, um alle Dateien innerhalb eines Verzeichnisses und aller seiner Unterverzeichnisse (mit Generatoren) rekursiv abzurufen:

```
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
```

Diese Funktion kann durch den `yield from` vereinfacht werden:

```
def get_files_recursive(directory):
    yield from get_files(directory)
    for subdirectory in get_directories(directory):
        yield from get_files_recursive(subdirectory)
```

Parallele Iteration über Generatoren

Um mehrere Generatoren parallel zu durchlaufen, verwenden Sie den `zip` eingebauten Code:

```
for x, y in zip(a,b):
    print(x,y)
```

Ergebnisse in:

```
1 x
2 y
3 z
```

In Python 2 sollten `itertools.izip` stattdessen `itertools.izip` verwenden. Hier können wir auch sehen, dass alle `zip` Funktionen Tupel ergeben.

Beachten Sie, dass zip die Iteration stoppt, sobald einer der iterierbaren Elemente keine Elemente mehr enthält. Wenn Sie so lange iterieren `itertools.zip_longest()` wie es am längsten durchlaufen wird, verwenden Sie `itertools.zip_longest()` .

Code zum Erstellen von Listen umgestalten

Angenommen, Sie verfügen über komplexen Code, der eine Liste erstellt und zurückgibt, indem Sie mit einer leeren Liste beginnen und wiederholt an sie anhängen:

```
def create():
    result = []
    # logic here...
    result.append(value) # possibly in several places
    # more logic...
    return result # possibly in several places

values = create()
```

Wenn es nicht praktikabel ist, die innere Logik durch ein Listenverständnis zu ersetzen, können Sie die gesamte Funktion in einen Generator vor Ort umwandeln und dann die Ergebnisse sammeln:

```
def create_gen():
    # logic...
    yield value
    # more logic
    return # not needed if at the end of the function, of course

values = list(create_gen())
```

Wenn die Logik rekursiv ist, verwenden Sie `yield from` , um alle Werte des rekursiven Aufrufs in ein "abgeflachtes" Ergebnis aufzunehmen:

```
def preorder_traversal(node):
    yield node.value
    for child in node.children:
        yield from preorder_traversal(child)
```

Suchen

Die `next` Funktion ist auch ohne Iteration nützlich. Durch Übergeben eines Generatorausdrucks an `next` schnell nach dem ersten Vorkommen eines Elements suchen, das mit einem Prädikat übereinstimmt. Verfahrenscode wie

```
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)
```

kann ersetzt werden durch:

```
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.
```

Zu diesem Zweck kann es wünschenswert sein, einen Alias wie `first = next` oder eine Wrapper-Funktion zu erstellen, um die Ausnahme zu konvertieren:

```
def first(generator):
    try:
        return next(generator)
    except StopIteration:
        raise ValueError
```

Generatoren online lesen: https://riptutorial.com/de/python/topic/292/generatoren

Kapitel 71: Graph-Werkzeug

Einführung

Mit den Python-Werkzeugen können Sie ein Diagramm erstellen

Examples

PyDotPlus

PyDotPlus ist eine verbesserte Version des alten pydot-Projekts, das eine Python-Schnittstelle für die Punktsprache von Graphviz bietet.

Installation

Für die neueste stabile Version:

```
pip install pydotplus
```

Für die Entwicklungsversion:

```
pip install https://github.com/carlos-jenkins/pydotplus/archive/master.zip
```

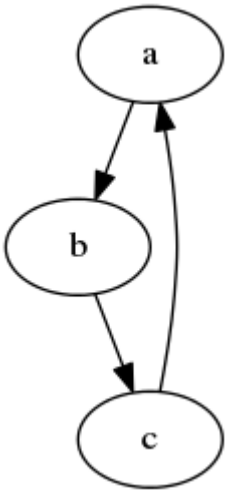
Laden Sie das Diagramm wie in einer DOT-Datei definiert

- Es wird angenommen, dass die Datei im DOT-Format vorliegt. Es wird geladen, analysiert und eine Punktklasse zurückgegeben, die den Graphen darstellt. Zum Beispiel eine einfache 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.
```

Sie erhalten eine SVG (Scalable Vector Graphics) wie folgt:



PyGraphviz

Holen Sie sich PyGraphviz aus dem Python Package Index unter
http://pypi.python.org/pypi/pygraphviz

oder installiere es mit:

```
pip install pygraphviz
```

Es wird versucht, eine geeignete Version zu finden und zu installieren, die Ihrem Betriebssystem und der Python-Version entspricht.

Sie können die Entwicklungsversion (auf github.com) installieren mit:

```
pip install git://github.com/pygraphviz/pygraphviz.git#egg=pygraphviz
```

Holen Sie sich PyGraphviz aus dem Python Package Index unter
http://pypi.python.org/pypi/pygraphviz

oder installiere es mit:

```
easy_install pygraphviz
```

Es wird versucht, eine geeignete Version zu finden und zu installieren, die Ihrem Betriebssystem und der Python-Version entspricht.

Laden Sie das Diagramm wie in einer DOT-Datei definiert

- Es wird angenommen, dass die Datei im DOT-Format vorliegt. Es wird geladen, analysiert und eine Punktklasse zurückgegeben, die den Graphen darstellt. Zum Beispiel eine einfache demo.dot:

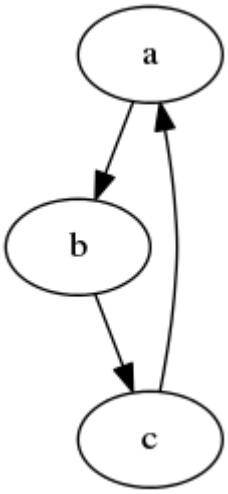
  digraph demo1 {a -> b -> c; c -> a; }

- Laden Sie es und zeichnen Sie es.

```
import pygraphviz as pgv
G = pgv.AGraph("demo.dot")
```

```
G.draw('test', format='svg', prog='dot')
```

Sie erhalten eine SVG (Scalable Vector Graphics) wie folgt:



Graph-Werkzeug online lesen: https://riptutorial.com/de/python/topic/9483/graph-werkzeug

Kapitel 72: Grundflüche mit Python

Bemerkungen

Curses ist ein grundlegendes Modul zur Terminal- (oder Zeichenanzeige-) Verarbeitung von Python. Dies kann zum Erstellen von Terminal-basierten Benutzeroberflächen oder TUIs verwendet werden.

Dies ist ein Python-Port einer populäreren C-Bibliothek 'ncurses'

Examples

Beispiel für ein einfaches Aufruf

```
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()
```

Die helper-Funktion wrapper ().

Während der grundlegende Aufruf oben einfach ist, bietet das Paket curses die `wrapper(func, ...)` Funktion `wrapper(func, ...)` . Das folgende Beispiel enthält das Äquivalent von oben:

```
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)
```

In diesem `stdscr` initialisiert der Wrapper Flüche, erstellt `stdscr` , ein WindowObject und übergibt sowohl stdscr als auch weitere Argumente an `func` . Wenn `func` zurückkehrt, stellt der `wrapper` das Terminal wieder her, bevor das Programm beendet wird.

Grundflüche mit Python online lesen: https://riptutorial.com/de/python/topic/5851/grundfluche-mit-python

Kapitel 73: Grundlegende Eingabe und Ausgabe

Examples

Input () und raw_input () verwenden

Python 2.x 2.3

`raw_input` wartet darauf, dass der Benutzer Text `raw_input` und das Ergebnis als String `raw_input` .

```
foo = raw_input("Put a message here that asks the user for input")
```

Im obigen Beispiel speichert `foo` die Eingaben des Benutzers.

Python 3.x 3.0

`input` wartet, bis der Benutzer Text eingibt und das Ergebnis als String zurückgibt.

```
foo = input("Put a message here that asks the user for input")
```

Im obigen Beispiel speichert `foo` die Eingaben des Benutzers.

Verwendung der Druckfunktion

Python 3.x 3.0

In Python 3 hat die Druckfunktion die Form einer Funktion:

```
print("This string will be displayed in the output")
# This string will be displayed in the output

print("You can print \n escape characters too.")
# You can print escape characters too.
```

Python 2.x 2.3

In Python 2 war print ursprünglich eine Anweisung (siehe unten).

```
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.
```

Hinweis: Bei Verwendung `from __future__ import print_function` in Python 2 können Benutzer die Funktion `print()` wie Python 3-Code verwenden. Dies ist nur in Python 2.6 und höher verfügbar.

Funktion, um den Benutzer zur Eingabe einer Nummer aufzufordern

```
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)
```

Und um es zu benutzen:

```
user_number = input_number("input a number: ", "that's not a number!")
```

Oder wenn Sie keine "Fehlermeldung" wünschen:

```
user_number = input_number("input a number: ")
```

Einen String ohne Zeilenumbruch am Ende drucken

Python 2.x 2.3

Um eine Zeile mit `print` fortzusetzen, beenden Sie die `print` Anweisung in Python 2.x mit einem Komma. Es wird automatisch ein Leerzeichen hinzugefügt.

```
print "Hello,",
print "World!"
# Hello, World!
```

Python 3.x 3.0

In Python 3.x, die `print` hat die Funktion eines optionalen `end` Parameter das ist , was es am Ende der gegebenen Zeichenfolge gedruckt wird . Standardmäßig handelt es sich um ein Zeilenvorschubzeichen, das dem folgenden entspricht:

```
print("Hello, ", end="\n")
print("World!")
# Hello,
# World!
```

Sie könnten aber auch andere Zeichenketten übergeben

```
print("Hello, ", end="")
print("World!")
# Hello, World!

print("Hello, ", end="<br>")
print("World!")
# Hello, <br>World!

print("Hello, ", end="BREAK")
print("World!")
# Hello, BREAKWorld!
```

Wenn Sie mehr Kontrolle über die Ausgabe wünschen, können Sie `sys.stdout.write` :

```
import sys

sys.stdout.write("Hello, ")
sys.stdout.write("World!")
# Hello, World!
```

Aus stdin lesen

Python-Programme können von Unix-Pipelines lesen. Hier ist ein einfaches Beispiel, wie man aus `stdin` liest:

```
import sys

for line in sys.stdin:
    print(line)
```

`sys.stdin` dass `sys.stdin` ein Stream ist. Das bedeutet, dass die for-Schleife erst beendet wird, wenn der Stream beendet ist.

Sie können nun die Ausgabe eines anderen Programms wie folgt in Ihr Python-Programm einleiten:

```
$ cat myfile | python myprogram.py
```

In diesem Beispiel kann `cat myfile` ein beliebiger Unix-Befehl sein, der an `stdout cat myfile` .

Alternativ kann das Fileinput-Modul nützlich sein:

```
import fileinput
for line in fileinput.input():
    process(line)
```

Eingabe aus einer Datei

Eingaben können auch aus Dateien gelesen werden. Dateien können mit der integrierten Funktion `open` . Die Verwendung einer `with <command> as <name>` -Syntax (als 'Context Manager' bezeichnet) macht es sehr einfach, `open` und ein Handle für die Datei zu erhalten:

```
with open('somefile.txt', 'r') as fileobj:
    # write code here using fileobj
```

Dadurch wird sichergestellt, dass die Datei automatisch geschlossen wird, wenn die Codeausführung den Block verlässt.

Dateien können in verschiedenen Modi geöffnet werden. Im obigen Beispiel wird die Datei schreibgeschützt geöffnet. Um eine vorhandene Datei nur zum Lesen zu öffnen, verwenden Sie `r` . Wenn Sie diese Datei als Bytes lesen möchten, verwenden Sie `rb` . Um Daten an eine vorhandene Datei anzuhängen, verwenden Sie `a` . Verwenden Sie `w` , um eine Datei zu erstellen oder vorhandene Dateien mit demselben Namen zu überschreiben. Mit `r+` können Sie eine Datei zum Lesen und Schreiben öffnen. Das erste Argument von `open()` ist der Dateiname, das zweite ist der Modus. Wenn der Modus leer bleibt, wird standardmäßig `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']
```

Wenn die Größe der Datei sehr klein ist, können Sie den gesamten Inhalt der Datei in den Speicher lesen. Wenn die Datei sehr groß ist, ist es oft besser, Zeile für Zeile oder nach Abschnitten zu lesen und die Eingabe in derselben Schleife zu verarbeiten. Das zu tun:

```
with open('shoppinglist.txt', 'r') as fileobj:
    # this method reads line by line:
    lines = []
    for line in fileobj:
        lines.append(line.strip())
```

Beachten Sie beim Lesen von Dateien die betriebssystemspezifischen Zeilenumbruchszeichen. Obwohl `for line in fileobj` automatisch entfernt werden, ist es immer sicher, `strip()` in den gelesenen Zeilen aufzurufen, wie oben gezeigt.

Geöffnete Dateien (`fileobj` in den obigen Beispielen) zeigen immer auf einen bestimmten Ort in der Datei. Wenn sie zum ersten Mal geöffnet werden, zeigt das Datei-Handle auf den Anfang der Datei, die Position `0` . Der Datei-Handle kann seine aktuelle Position mit `tell` anzeigen:

```
fileobj = open('shoppinglist.txt', 'r')
pos = fileobj.tell()
print('We are at %u.' % pos) # We are at 0.
```

Beim Lesen des gesamten Inhalts wird die Position des Dateihandlers am Ende der Datei angezeigt:

```
content = fileobj.read()
end = fileobj.tell()
print('This file was %u characters long.' % end)
# This file was 22 characters long.
fileobj.close()
```

Die Dateihandlerposition kann auf das gesetzt werden, was benötigt wird:

```
fileobj = open('shoppinglist.txt', 'r')
fileobj.seek(7)
pos = fileobj.tell()
print('We are at character #%u.' % pos)
```

Sie können während eines Anrufs auch eine beliebige Länge aus dem Dateiinhalt lesen. Dazu übergeben Sie ein Argument für `read()` . Wenn `read()` ohne Argument aufgerufen wird, wird es bis zum Ende der Datei gelesen. Wenn Sie ein Argument übergeben, liest es je nach Modus (`rb` und `r`) diese Anzahl an Bytes oder Zeichen:

```
# 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()
```

Um den Unterschied zwischen Zeichen und Bytes zu demonstrieren:

```
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'>
```

Grundlegende Eingabe und Ausgabe online lesen:
https://riptutorial.com/de/python/topic/266/grundlegende-eingabe-und-ausgabe

Kapitel 74: gruppiere nach()

Einführung

Mit der Methode `itertools.groupby()` können Entwickler in Python Werte einer iterierbaren Klasse basierend auf einer angegebenen Eigenschaft in eine andere iterierbare Menge von Werten gruppieren.

Syntax

- itertools.groupby (iterable, key = None oder eine Funktion)

Parameter

Parameter	Einzelheiten
iterable	Jeder Python iterierbar
Schlüssel	Funktion (Kriterien), nach der das Iterierbare gruppiert werden soll

Bemerkungen

groupby () ist knifflig, aber eine allgemeine Regel, die bei der Verwendung zu beachten ist:

Sortieren Sie die Elemente, die Sie gruppieren möchten, immer mit demselben Schlüssel, den Sie für die Gruppierung verwenden möchten

Es wird empfohlen, dass sich der Leser die Dokumentation hier ansieht und anhand einer Klassendefinition erläutert, wie diese erklärt wird.

Examples

Beispiel 1

Sagen Sie, Sie haben die Schnur

```
s = 'AAAABBBCCDAABBB'
```

und Sie möchten es teilen, so dass alle 'A' in einer Liste stehen und so mit 'B' und 'C' usw. Sie könnten so etwas tun

```
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
```

Ergebnisse in

```
{'A': ['A', 'A', 'A', 'A', 'A', 'A'],
 'B': ['B', 'B', 'B', 'B', 'B', 'B'],
 'C': ['C', 'C'],
 'D': ['D']}
```

Bei großen Datenmengen würden Sie diese Elemente jedoch im Speicher aufbauen. Hier kommt groupby () ins Spiel

Wir könnten das gleiche Ergebnis auf eine effizientere Weise erzielen, indem Sie Folgendes tun

```
# 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
```

Ergebnisse in

```
{'A': ['A', 'A'], 'B': ['B', 'B', 'B'], 'C': ['C', 'C'], 'D': ['D']}
```

Beachten Sie, dass die Anzahl von 'A' im Ergebnis, wenn Sie group by verwendet haben, geringer ist als die tatsächliche Anzahl von 'A' in der ursprünglichen Zeichenfolge. Wir können diesen Informationsverlust vermeiden, indem wir die Elemente in s sortieren, bevor Sie sie an c übergeben (siehe unten)

```
c = groupby(sorted(s))

dic = {}
for k, v in c:
    dic[k] = list(v)
dic
```

Ergebnisse in

```
{'A': ['A', 'A', 'A', 'A', 'A', 'A'], 'B': ['B', 'B', 'B', 'B', 'B', 'B'], 'C': ['C', 'C'],
'D': ['D']}
```

Jetzt haben wir alle unsere 'A'.

Beispiel 2

Dieses Beispiel zeigt, wie der Standardschlüssel ausgewählt wird, wenn wir keinen angeben

```
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
```

Ergebnisse in

```
{1: [1, 1],
 2: [2],
 3: [3],
 ('persons', 'man', 'woman'): [('persons', 'man', 'woman')],
 'cow': ['cow'],
 'dog': ['dog'],
 10: [10],
 11: [11],
 'goat': ['goat']}
```

Beachten Sie hierbei, dass das Tupel als Ganzes in dieser Liste als eine Taste zählt

Beispiel 3

Beachten Sie in diesem Beispiel, dass Mulato und Kamel in unserem Ergebnis nicht angezeigt werden. Nur das letzte Element mit dem angegebenen Schlüssel wird angezeigt. Das letzte Ergebnis für c löscht tatsächlich zwei vorherige Ergebnisse. Aber schau dir die neue Version an, bei der ich die Daten zuerst auf demselben Schlüssel sortiert habe.

```
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
```

Ergebnisse in

```
{'c': ['camel'],
 'd': ['dog', 'donkey'],
 'g': ['goat'],
 'm': ['mongoose', 'malloo'],
 'persons': [('persons', 'man', 'woman')],
 'w': ['wombat']}
```

Sortierte Version

```
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
```

Ergebnisse in

```
['cow', 'cat', 'camel', 'dog', 'donkey', 'goat', 'mulato', 'mongoose', 'malloo', ('persons',
'man', 'woman'), 'wombat']

{'c': ['cow', 'cat', 'camel'],
 'd': ['dog', 'donkey'],
 'g': ['goat'],
 'm': ['mulato', 'mongoose', 'malloo'],
 'persons': [('persons', 'man', 'woman')],
 'w': ['wombat']}
```

Beispiel 4

In diesem Beispiel sehen wir, was passiert, wenn wir verschiedene Arten von Iterationen verwenden.

```
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
```

Ergebnisse in

```
{'animal': [('animal', 'bear'), ('animal', 'duck')],
 'plant': [('plant', 'cactus')],
 'vehicle': [('vehicle', 'harley'),
  ('vehicle', 'speed boat'),
  ('vehicle', 'school bus')]}
```

Dieses Beispiel ist im Wesentlichen dasselbe wie das darüber. Der einzige Unterschied ist, dass ich alle Tupel in Listen geändert habe.

```
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
```

Ergebnisse

```
{'animal': [['animal', 'bear'], ['animal', 'duck']],
 'plant': [['plant', 'cactus']],
 'vehicle': [['vehicle', 'harley'],
  ['vehicle', 'speed boat'],
  ['vehicle', 'school bus']]}
```

gruppiere nach() online lesen: https://riptutorial.com/de/python/topic/8690/gruppiere-nach--

Kapitel 75: Hashlib

Einführung

hashlib implementiert eine gemeinsame Schnittstelle für viele verschiedene sichere Hash- und Message-Digest-Algorithmen. Enthalten sind die sicheren FIPS-Hash-Algorithmen SHA1, SHA224, SHA256, SHA384 und SHA512.

Examples

MD5-Hash einer Zeichenfolge

Dieses Modul implementiert eine gemeinsame Schnittstelle zu vielen verschiedenen sicheren Hash- und Message-Digest-Algorithmen. Enthalten sind die sicheren FIPS-Hash-Algorithmen SHA1, SHA224, SHA256, SHA384 und SHA512 (definiert in FIPS 180-2) sowie der RSA-MD5-Algorithmus (definiert in Internet RFC 1321).

Für jeden Hashtyp gibt es eine Konstruktormethode. Alle geben ein Hash-Objekt mit derselben einfachen Schnittstelle zurück. Beispiel: Verwenden Sie `sha1()` , um ein SHA1- `sha1()` zu erstellen.

```
hash.sha1()
```

Konstruktoren für Hash-Algorithmen, die in diesem Modul immer vorhanden sind, sind `md5()` , `sha1()` , `sha224()` , `sha256()` , `sha384()` und `sha512()` .

Sie können dieses Objekt nun mit der `update()` -Methode mit beliebigen Zeichenfolgen füttern. Sie können es jederzeit nach dem Digest der Verkettung der bisher eingespeisten Zeichenfolgen mit den Methoden `digest()` oder `hexdigest()` .

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

> Aktualisieren Sie das Hash-Objekt mit dem String arg. Wiederholte Aufrufe entsprechen einem einzigen Aufruf mit der Verkettung aller Argumente: m.update (a); m.update (b) ist äquivalent zu m.update (a + b).

```
hash.digest()
```

> Gibt den Auszug der Strings zurück, die bisher an die update () -Methode übergeben wurden. Dies ist eine Zeichenfolge aus digest_size-Bytes, die Nicht-ASCII-Zeichen enthalten kann, einschließlich null Bytes.

```
hash.hexdigest()
```

> Wie digest (), es sei denn, der Digest wird als String mit doppelter Länge zurückgegeben und enthält nur hexadezimale Ziffern. Dies kann verwendet werden,

um den Wert sicher in E-Mails oder anderen nicht-binären Umgebungen auszutauschen.

Hier ist ein Beispiel:

```
>>> 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
```

oder:

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

Algorithmus von OpenSSL

Es gibt auch einen generischen `new()` Konstruktor, der den Stringnamen des gewünschten Algorithmus als ersten Parameter verwendet, um den Zugriff auf die oben aufgeführten Hashes sowie auf andere Algorithmen zu ermöglichen, die Ihre OpenSSL-Bibliothek möglicherweise bietet. Die genannten Konstruktoren sind viel schneller als `new()` und sollten bevorzugt werden.

Verwenden von `new()` mit einem von OpenSSL bereitgestellten Algorithmus:

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

Hashlib online lesen: https://riptutorial.com/de/python/topic/8980/hashlib

Kapitel 76: Häufige Fehler

Einführung

Python ist eine Sprache, die klar und lesbar sein soll, ohne Unklarheiten und unerwartete Verhaltensweisen. Leider sind diese Ziele nicht in allen Fällen erreichbar. Aus diesem Grund gibt es in Python einige Eckpunkte, in denen es möglicherweise etwas anderes tut als erwartet.

In diesem Abschnitt werden einige Probleme beschrieben, die beim Schreiben von Python-Code auftreten können.

Examples

Ändern Sie die Reihenfolge, die Sie durchlaufen

Eine `for` Schleife durchläuft eine Sequenz, daher kann das **Ändern dieser Sequenz innerhalb der Schleife zu unerwarteten Ergebnissen führen** (insbesondere beim Hinzufügen oder Entfernen von Elementen):

```
alist = [0, 1, 2]
for index, value in enumerate(alist):
    alist.pop(index)
print(alist)
# Out: [1]
```

Hinweis: Mit `list.pop()` werden Elemente aus der Liste entfernt.

Das zweite Element wurde nicht gelöscht, da die Iteration der Reihe nach die Indizes durchläuft. Die obige Schleife wiederholt sich zweimal mit folgenden Ergebnissen:

```
# 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]
```

Dieses Problem tritt auf, weil sich die Indizes ändern, während sie sich in Richtung des Indexanstiegs ändern. Um dieses Problem zu vermeiden, können Sie **die Schleife rückwärts durchlaufen** :

```
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]
```

Wenn Sie die Schleife am Ende durchlaufen und Elemente entfernen (oder hinzufügen), hat dies keinen Einfluss auf die Indexe der Elemente, die sich zuvor in der Liste befanden. In diesem Beispiel werden also alle Elemente, die auch von `alist` stammen, `alist` .

Ein ähnliches Problem entsteht beim **Einfügen oder Anhängen von Elementen an eine Liste, die Sie durchlaufen** , was zu einer Endlosschleife führen kann:

```
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]
```

Ohne die `break` fügt die Schleife `'a'` solange dem Computer nicht genügend Speicher zur Verfügung steht und das Programm weiterlaufen darf. In einer Situation wie dieser wird normalerweise bevorzugt, eine neue Liste zu erstellen und der neuen Liste Elemente hinzuzufügen, während Sie die ursprüngliche Liste durchlaufen.

Bei Verwendung einer `for` Schleife können **Sie die Listenelemente nicht mit der Platzhaltervariablen ändern** :

```
alist = [1,2,3,4]
for item in alist:
    if item % 2 == 0:
        item = 'even'
print(alist)
# Out: [1,2,3,4]
```

Im obigen Beispiel **ändert das Ändern eines `item` nichts an der ursprünglichen Liste** . Sie müssen den `alist[2]` (`alist[2]`) verwenden, und `enumerate()` funktioniert dafür gut:

```
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']
```

Eine **`while` Schleife ist** in manchen Fällen die bessere Wahl:

Wenn Sie **alle Elemente** in der Liste **löschen möchten** :

```
zlist = [0, 1, 2]
while zlist:
    print(zlist[0])
    zlist.pop(0)
print('After: zlist =', zlist)

# Out: 0
#      1
#      2
# After: zlist = []
```

Wenn Sie `zlist` einfach zurücksetzen, `zlist` dasselbe Ergebnis.

```
zlist = []
```

Das obige Beispiel kann auch mit `len()` kombiniert werden, um nach einem bestimmten Punkt zu stoppen oder um alle Elemente außer `x` zu löschen:

```
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]
```

Oder zum **Durchlaufen einer Liste, während Elemente gelöscht werden, die eine bestimmte Bedingung erfüllen** (in diesem Fall werden alle geraden Elemente gelöscht):

```
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]
```

Beachten Sie, dass Sie `i` nicht erhöhen, nachdem Sie ein Element gelöscht haben. Durch das Löschen des Elements in `zlist[i]` hat sich der Index des nächsten Elements um eins verringert. `zlist[i]` mit demselben Wert für `i` bei der nächsten Iteration überprüfen, werden Sie das nächste Element in der Liste korrekt überprüfen .

Ein anderer Weg, über das Entfernen unerwünschter Elemente aus einer Liste nachzudenken, besteht darin **, gewünschte Elemente zu einer neuen Liste hinzuzufügen** . Das folgende Beispiel ist eine Alternative zu diesem Beispiel `while` Schleife:

```
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]
```

Hier führen wir die gewünschten Ergebnisse in eine neue Liste ein. Die temporäre Liste kann dann optional der ursprünglichen Variablen zugewiesen werden.

Mit diesem Gedankengang können Sie eine der elegantesten und leistungsstärksten Funktionen von Python aufrufen, **Listenverständnisse** , die temporäre Listen eliminieren und von der zuvor diskutierten In-Place-Liste / Index-Mutationsideologie abweichen.

```
zlist = [1,2,3,4,5]
[item for item in zlist if item % 2 != 0]
# Out: [1, 3, 5]
```

Veränderliches Standardargument

```
def foo(li=[]):
    li.append(1)
    print(li)

foo([2])
# Out: [2, 1]
foo([3])
# Out: [3, 1]
```

Dieser Code verhält sich wie erwartet, aber was ist, wenn wir kein Argument übergeben?

```
foo()
# Out: [1] As expected...

foo()
# Out: [1, 1]  Not as expected...
```

Dies liegt daran, dass Standardargumente von Funktionen und Methoden nicht zur Laufzeit, sondern zur **Definitionszeit** ausgewertet werden. Also nur wir jemals eine einzige Instanz der haben `li` Liste.

Um dies zu umgehen, verwenden Sie nur unveränderliche Typen für Standardargumente:

```
def foo(li=None):
    if not li:
        li = []
    li.append(1)
    print(li)

foo()
```

```
# Out: [1]

foo()
# Out: [1]
```

Eine Verbesserung und `if not li` korrekt als `False` ausgewertet, gilt dies für viele andere Objekte, z. Die folgenden Beispielargumente können zu unbeabsichtigten Ergebnissen führen:

```
x = []
foo(li=x)
# Out: [1]

foo(li="")
# Out: [1]

foo(li=0)
# Out: [1]
```

Der idiomatische Ansatz besteht darin, das Argument direkt gegen das `None` Objekt zu prüfen:

```
def foo(li=None):
    if li is None:
        li = []
    li.append(1)
    print(li)

foo()
# Out: [1]
```

Listenmultiplikation und gemeinsame Referenzen

Betrachten Sie den Fall der Erstellung einer verschachtelten Listenstruktur durch Multiplikation:

```
li = [[]] * 3
print(li)
# Out: [[], [], []]
```

Auf den ersten Blick denken wir, wir hätten eine Liste mit 3 verschachtelten Listen. Versuchen wir, `1` an die erste anzuhängen:

```
li[0].append(1)
print(li)
# Out: [[1], [1], [1]]
```

`1` wurde auf alle Listen in den beigefügten `li` .

Der Grund ist, dass `[[]] * 3` keine `list` mit 3 verschiedenen `list` . Vielmehr erstellt er eine `list` mit mindestens 3 Verweise auf die gleiche `list` Objekt. Wenn wir also an `li[0]` anhängen, ist die Änderung in allen Unterelementen von `li` sichtbar. Dies ist gleichbedeutend mit:

```
li = []
element = [[]]
```

```
li = element + element + element
print(li)
# Out: [[], [], []]
element.append(1)
print(li)
# Out: [[1], [1], [1]]
```

Dies kann weiter bestätigt werden, wenn wir die Speicheradressen der enthaltenen `list` mit `id` ausdrucken:

```
li = [[]] * 3
print([id(inner_list) for inner_list in li])
# Out: [6830760, 6830760, 6830760]
```

Die Lösung besteht darin, die inneren Listen mit einer Schleife zu erstellen:

```
li = [[] for _ in range(3)]
```

Anstatt eine einzige `list` erstellen und anschließend drei Verweise darauf zu erstellen, erstellen wir nun drei unterschiedliche Listen. Dies kann wiederum mit der `id` Funktion überprüft werden:

```
print([id(inner_list) for inner_list in li])
# Out: [6331048, 6331528, 6331488]
```

Sie können das auch tun. Es bewirkt, dass bei jedem `append` Aufruf eine neue leere Liste erstellt wird.

```
>>> li = []
>>> li.append([])
>>> li.append([])
>>> li.append([])
>>> for k in li: print(id(k))
...
4315469256
4315564552
4315564808
```

Verwenden Sie den Index nicht, um eine Sequenz zu durchlaufen.

Nicht:

```
for i in range(len(tab)):
    print(tab[i])
```

Tun :

```
for elem in tab:
    print(elem)
```

`for` wird die meisten Iterationsvorgänge für Sie automatisieren.

Verwenden Sie Aufzählung, wenn Sie wirklich sowohl den Index als auch das Element benötigen .

```
for i, elem in enumerate(tab):
     print((i, elem))
```

Seien Sie vorsichtig, wenn Sie "==" verwenden, um nach "Wahr" oder "Falsch" zu suchen

```
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
```

Überprüfen Sie nicht, ob Sie dies tun können, und führen Sie den Fehler aus

Pythonistas sagen normalerweise "Es ist einfacher, um Vergebung zu bitten als um Erlaubnis".

Nicht:

```
if os.path.isfile(file_path):
    file = open(file_path)
else:
    # do something
```

Tun:

```
try:
    file = open(file_path)
except OSError as e:
    # do something
```

Oder noch besser mit `Python 2.6+` :

```
with open(file_path) as file:
```

Es ist viel besser, weil es viel generischer ist. Sie können `try/except` auf fast alles anwenden. Sie müssen sich nicht darum kümmern, was zu tun ist, um dies zu verhindern, sondern nur den Fehler, den Sie riskieren.

Nicht gegen Typ prüfen

Python ist dynamisch typisiert. Wenn Sie also nach Typ suchen, verlieren Sie die Flexibilität. Verwenden Sie stattdessen das Eingeben von Enten, indem Sie das Verhalten überprüfen. Wenn Sie eine Zeichenfolge in einer Funktion erwarten, konvertieren Sie ein Objekt mit `str()` in eine Zeichenfolge. Wenn Sie mit einer Liste rechnen, verwenden Sie `list()` , um alle Iterierbaren in eine Liste zu konvertieren.

Nicht:

```
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)
```

Tun:

```
def foo(name) :
    print(str(name).lower())

def bar(listing) :
    l = list(listing)
    l.extend((1, 2, 3))
    return ", ".join(l)
```

Auf die letzte Weise akzeptiert `foo` jedes Objekt. `bar` akzeptiert Strings, Tupel, Sets, Listen und vieles mehr. Billig TROCKEN.

Mischen Sie keine Leerzeichen und Tabulatoren

Objekt* als erstes übergeordnetes *Objekt verwenden

Das ist knifflig, aber es wird Sie beißen, wenn Ihr Programm wächst. In `Python 2.x` gibt es alte und neue Klassen. Die Alten sind gut, alt. Sie verfügen nicht über einige Funktionen und können bei der Vererbung unangenehm sein. Um nutzbar zu sein, muss jede Ihrer Klassen den "neuen Stil" haben. Dazu machen Sie es vom `object` erben.

Nicht:

```
class Father:
    pass
```

```
class Child(Father):
    pass
```

Tun:

```
class Father(object):
    pass


class Child(Father):
    pass
```

In `Python 3.x` alle Klassen einen neuen Stil, so dass Sie dies nicht tun müssen.

Initialisieren Sie Klassenattribute nicht außerhalb der init- Methode

Menschen, die aus anderen Sprachen kommen, finden es verlockend, weil Sie dies in Java oder PHP tun. Sie schreiben den Klassennamen, listen Ihre Attribute auf und geben ihnen einen Standardwert. Es scheint in Python zu funktionieren, aber das funktioniert nicht so, wie Sie denken. Dadurch werden Klassenattribute (statische Attribute) eingerichtet. Wenn Sie versuchen, das Objektattribut abzurufen, erhalten Sie dessen Wert, wenn es nicht leer ist. In diesem Fall werden die Klassenattribute zurückgegeben. Daraus ergeben sich zwei große Gefahren:

- Wenn das Klassenattribut geändert wird, wird der Anfangswert geändert.
- Wenn Sie ein veränderbares Objekt als Standardwert festlegen, wird dasselbe Objekt für alle Instanzen gemeinsam verwendet.

Nicht (es sei denn, Sie möchten statisch):

```
class Car(object):
    color = "red"
    wheels = [Wheel(), Wheel(), Wheel(), Wheel()]
```

Tun :

```
class Car(object):
    def __init__(self):
        self.color = "red"
        self.wheels = [Wheel(), Wheel(), Wheel(), Wheel()]
```

Integer- und String-Identität

Python verwendet internes Caching für eine Reihe von ganzen Zahlen, um den unnötigen Aufwand durch wiederholtes Erstellen zu reduzieren.

Dies kann zu einem verwirrenden Verhalten beim Vergleich von Integer-Identitäten führen:

```
>>> -8 is (-7 - 1)
False
>>> -3 is (-2 - 1)
```

```
True
```

und mit einem anderen Beispiel:

```
>>> (255 + 1) is (255 + 1)
True
>>> (256 + 1) is (256 + 1)
False
```

Warte was?

Wir können sehen , dass die Identität Betrieb `is` Ausbeuten `True` für einige ganze Zahlen (`-3` , `256`) , aber nicht für andere (`-8` , `257`).

Genauer gesagt, Ganzzahlen im Bereich `[-5, 256]` werden beim Start des Interpreters intern zwischengespeichert und nur einmal erstellt. Als solche sind sie **identisch** und mit ihrer Identität zu vergleichen `is` ergibt `True` ; Ganzzahlen außerhalb dieses Bereichs werden (normalerweise) on-the-fly erstellt und ihre Identitäten werden mit `False` verglichen.

Dies ist eine häufige Fallstricke, da dies ein üblicher Bereich für Tests ist, aber oft genug versagt der Code beim späteren Bereitstellungsprozess (oder schlechterer Produktion), ohne dass ein offensichtlicher Grund vorliegt, nachdem er in der Entwicklung perfekt gearbeitet hat.

Die Lösung besteht darin **, Werte immer mit dem Gleichheitsoperator** (`==`) und **nicht mit** dem Identitätsoperator (`is`) zu vergleichen.

Python hält auch Strings häufig verwendete Verweise auf und in ähnlicher Weise verwirrende Verhalten führen können , wenn Identitäten zu vergleichen (dh unter Verwendung `is`) von Strings.

```
>>> 'python' is 'py' + 'thon'
True
```

Die Zeichenfolge `'python'` wird häufig verwendet, daher hat Python ein Objekt, das alle Verweise auf die Zeichenfolge `'python'` verwenden.

Bei ungewöhnlichen Zeichenfolgen schlägt der Vergleich der Identität fehl, selbst wenn die Zeichenfolgen gleich sind.

```
>>> '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
```

Vergleichen Sie die Zeichenfolgenwerte wie bei der Ganzzahl **immer mit dem Gleichheitsoperator** (`==`) und **nicht mit** dem Identitätsoperator (`is`).

Zugriff auf Attribute von Int-Literalen

Sie haben vielleicht gehört, dass alles in Python ein Objekt ist, sogar Literale. Dies bedeutet

beispielsweise, dass `7` ein Objekt ist, das heißt, es hat Attribute. Eines dieser Attribute ist beispielsweise die `bit_length` . Es gibt die Menge an Bits zurück, die zur Darstellung des Werts benötigt wird, für den er aufgerufen wird.

```
x = 7
x.bit_length()
# Out: 3
```

Wenn Sie sehen, dass der obige Code funktioniert, könnten Sie intuitiv denken, dass `7.bit_length()` funktionieren würde, nur um herauszufinden, dass dies einen `SyntaxError` . Warum? weil der Interpreter zwischen einem Attributzugriff und einer Floating-Nummer unterscheiden muss (z. B. `7.2` oder `7.bit_length()`). Das kann nicht, und deshalb wird eine Ausnahme ausgelöst.

Es gibt verschiedene Möglichkeiten, auf die Attribute eines `int` Literalen zuzugreifen:

```
# parenthesis
(7).bit_length()
# a space
7 .bit_length()
```

Die Verwendung von zwei Punkten (wie `7..bit_length()`) funktioniert in diesem Fall nicht, da dadurch ein `float` Literal erstellt wird und Floats nicht über die `bit_length()` -Methode verfügen.

Dieses Problem besteht nicht beim Zugriff auf `float` Attribute der `float` Literale, da der Interperter "intelligent" genug ist, um zu wissen, dass ein `float` Literal nicht zwei enthalten kann `.` , zum Beispiel:

```
7.2.as_integer_ratio()
# Out: (8106479329266893, 1125899906842624)
```

Verkettung oder Operator

Beim Testen auf einen von mehreren Gleichheitsvergleichen:

```
if a == 3 or b == 3 or c == 3:
```

es ist verlockend, dies zu verkürzen

```
if a or b or c == 3: # Wrong
```

Das ist falsch; Der Operator `or` hat eine niedrigere Priorität als `==` . Der Ausdruck wird also wie `if (a) or (b) or (c == 3):` Der richtige Weg ist die explizite Überprüfung aller Bedingungen:

```
if a == 3 or b == 3 or c == 3:  # Right Way
```

Alternativ kann die eingebaute `any()` Funktion anstelle von verketteten `or` Operatoren verwendet werden:

```
if any([a == 3, b == 3, c == 3]): # Right
```

Oder, um es effizienter zu machen:

```
if any(x == 3 for x in (a, b, c)): # Right
```

Oder um es kürzer zu machen:

```
if 3 in (a, b, c): # Right
```

Hier verwenden wir die `in` Operator zu testen , ob der Wert in einem Tupel vorhanden ist , die Werte enthalten , die wir gegen vergleichen wollen.

Ebenso ist es falsch zu schreiben

```
if a == 1 or 2 or 3:
```

was sollte als geschrieben werden

```
if a in (1, 2, 3):
```

sys.argv [0] ist der Name der ausgeführten Datei

Das erste Element von `sys.argv[0]` ist der Name der gerade ausgeführten Python-Datei. Die restlichen Elemente sind die Skriptargumente.

```
# 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']
```

Wörterbücher sind nicht geordnet

Sie können erwarten, dass ein Python-Wörterbuch nach Schlüsseln sortiert wird, z. B. einer C ++ `std::map` . Dies ist jedoch nicht der Fall:

```
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 hat keine eingebaute Klasse, die ihre Elemente automatisch nach Schlüssel sortiert.

Wenn das Sortieren jedoch kein Muss ist und Sie möchten, dass sich das Wörterbuch nur an die Reihenfolge der Einfügung der Schlüssel / Wert-Paare erinnert, können Sie `collections.OrderedDict` verwenden. `collections.OrderedDict` :

```
from collections import OrderedDict

oDict = OrderedDict([('first', 1), ('second', 2), ('third', 3)])

print([k for k in oDict])
# Out: ['first', 'second', 'third']
```

`OrderedDict` Sie, dass das Initialisieren eines `OrderedDict` mit einem Standardwörterbuch das Wörterbuch in keiner Weise für Sie sortiert. Alles , was diese Struktur macht , ist die Reihenfolge der Schlüsseleinführungs zu *bewahren.*

Die Implementierung von Wörterbüchern wurde in Python 3.6 geändert , um den Speicherverbrauch zu verbessern. Ein Nebeneffekt dieser neuen Implementierung ist, dass auch die Reihenfolge der an eine Funktion übergebenen Schlüsselwortargumente beibehalten wird:

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
```

> **Caveat:** Vorsicht , dass *„die Reihenfolge erhaltender Aspekt dieser neuen Implementierung wird eine Implementierung Detail betrachtet und soll nicht als verlässlich angesehen werden“* , wie es in der Zukunft ändern kann.

Globale Interpreter-Sperre (GIL) und blockierende Threads

Über Pythons GIL wurde viel geschrieben . Dies kann manchmal zu Verwirrung führen, wenn Sie mit Multi-Threaded-Anwendungen (nicht zu verwechseln mit Multiprozess-Anwendungen) umgehen.

Hier ist ein Beispiel:

```
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")
```

Sie würden erwarten, dass `Calculating...` direkt nach dem Start des Threads ausgedruckt wird. Wir wollten, dass die Berechnung in einem neuen Thread durchgeführt wird! Tatsächlich sehen Sie, dass es gedruckt wird, nachdem die Berechnung abgeschlossen ist. `math.factorial` liegt daran, dass der neue Thread auf einer C-Funktion (`math.factorial`) `math.factorial` die die GIL während der `math.factorial` sperrt.

Es gibt ein paar Möglichkeiten, dies zu umgehen. Die erste besteht darin, Ihre faktorielle Funktion in nativem Python zu implementieren. Dadurch kann der Haupt-Thread die Kontrolle übernehmen, während Sie sich in Ihrer Schleife befinden. Der Nachteil ist, dass diese Lösung **viel** langsamer ist, da wir die C-Funktion nicht mehr verwenden.

```
def calc_fact(num):
    """ A slow version of factorial in native Python """
    res = 1
    while num >= 1:
        res = res * num
        num -= 1
    return res
```

Sie können auch eine gewisse Zeit lang `sleep` bevor Sie mit der Ausführung beginnen. Hinweis: Dies erlaubt Ihrem Programm nicht, die Berechnung innerhalb der C-Funktion zu unterbrechen, aber Ihr Hauptthread kann nach dem Spawn fortfahren, was Sie vielleicht erwarten.

```
def calc_fact(num):
    sleep(0.001)
    math.factorial(num)
```

Variables Durchsickern in Listenverständnissen und für Schleifen

Betrachten Sie das folgende Listenverständnis

Python 2.x 2.7

```
i = 0
a = [i for i in range(3)]
print(i) # Outputs 2
```

Dies tritt nur in Python 2 auf, da das Listenverständnis die Schleifensteuerungsvariable in den umgebenden Geltungsbereich (Quelle) "leckt". Dieses Verhalten kann zu schwer zu findenden Fehlern führen und ***wurde in Python 3 behoben*** .

Python 3.x 3.0

```
i = 0
a = [i for i in range(3)]
print(i) # Outputs 0
```

Ebenso haben for-Schleifen keinen privaten Gültigkeitsbereich für ihre Iterationsvariable

```
i = 0
for i in range(3):
    pass
print(i) # Outputs 2
```

Diese Art von Verhalten tritt sowohl in Python 2 als auch in Python 3 auf.

Um Probleme mit undichten Variablen zu vermeiden, verwenden Sie neue Variablen in Listenverständnissen und für Schleifen.

Mehrfachrückgabe

Die Funktion xyz liefert zwei Werte a und b:

```
def xyz():
  return a, b
```

Code, der xyz aufruft, speichert das Ergebnis in einer Variablen, vorausgesetzt, xyz gibt nur einen Wert zurück:

```
t = xyz()
```

Der Wert von `t` ist eigentlich ein Tupel (a, b), sodass jede Aktion, die sich auf `t` bezieht, vorausgesetzt, dass es sich nicht um ein Tupel handelt, **tief** im Code mit einem unerwarteten **Fehler in** Bezug auf Tupel fehlschlagen kann.

> TypeError: Typ Tupel definiert keine ... Methode

Die Lösung wäre zu tun:

```
a, b = xyz()
```

Anfänger werden Schwierigkeiten haben, den Grund für diese Nachricht zu finden, indem sie nur die Tupel-Fehlermeldung lesen!

Pythonic JSON-Schlüssel

```
my_var = 'bla';
api_key = 'key';
...lots of code here...
params = {"language": "en", my_var: api_key}
```

Wenn Sie mit JavaScript vertraut sind, ist die Variablenbewertung in Python-Wörterbüchern nicht

das, was Sie erwarten. Diese Anweisung in JavaScript würde das `params` Objekt wie folgt ergeben:

```
{
    "language": "en",
    "my_var": "key"
}
```

In Python würde sich jedoch folgendes Wörterbuch ergeben:

```
{
    "language": "en",
    "bla": "key"
}
```

`my_var` wird ausgewertet und sein Wert wird als Schlüssel verwendet.

Häufige Fehler online lesen: https://riptutorial.com/de/python/topic/3553/haufige-fehler

Kapitel 77: Heapq

Examples

Größte und kleinste Objekte einer Kollektion

Um die größten Elemente in einer Auflistung zu finden, verfügt das `heapq` Modul über eine Funktion, die als `nlargest` wird. Wir übergeben ihm zwei Argumente. Das erste ist die Anzahl der Elemente, die wir abrufen möchten, das zweite ist der Name der Sammlung:

```
import heapq


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

In ähnlicher Weise verwenden wir die `nsmallest` Funktion, um die kleinsten Objekte in einer Sammlung zu finden:

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

`nsmallest` Funktionen `nlargest` und `nsmallest` ein optionales Argument (Schlüsselparameter) für komplizierte Datenstrukturen. Das folgende Beispiel zeigt die Verwendung der Eigenschaft `age` , um die ältesten und jüngsten Personen aus dem `people` Dictionary abzurufen:

```
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'}]
```

Kleinster Artikel in einer Kollektion

Die interessanteste Eigenschaft eines `heap` ist, dass das kleinste Element immer das erste Element ist: `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]
```

Heapq online lesen: https://riptutorial.com/de/python/topic/7489/heapq

Kapitel 78: HTML-Analyse

Examples

Suchen Sie nach einem Element in BeautifulSoup einen Text

Stellen Sie sich vor, Sie haben folgendes HTML:

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

Und Sie müssen den Text "John Smith" nach dem `label` Element finden.

In diesem Fall können Sie das `label` Element nach Text `.next_sibling` und dann die `.next_sibling` Eigenschaft verwenden :

```
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())
```

Druckt `John Smith` .

CSS-Selektoren in BeautifulSoup verwenden

BeautifulSoup unterstützt CSS-Selektoren nur begrenzt , deckt jedoch die am häufigsten verwendeten ab. Verwenden Sie die `select()` -Methode, um mehrere Elemente zu finden, und `select_one()` , um ein einzelnes Element zu finden.

Grundbeispiel:

```
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())
```

Drucke:

```
item1
item2
item3
```

PyQuery

pyquery ist eine jquery-ähnliche Bibliothek für Python. Es hat sehr gut Unterstützung für CSS-Selektoren.

```
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)
```

HTML-Analyse online lesen: https://riptutorial.com/de/python/topic/1384/html-analyse

Kapitel 79: Ijson

Einführung

ijson ist eine großartige Bibliothek für die Arbeit mit JSON-Dateien in Python. Leider wird standardmäßig ein reiner Python-JSON-Parser als Backend verwendet. Eine viel höhere Leistung kann mit einem C-Backend erreicht werden.

Examples

Einfaches Beispiel

Beispielbeispiel Aus einem Benchmarking

```
import ijson

def load_json(filename):
    with open(filename, 'r') as fd:
        parser = ijson.parse(fd)
        ret = {'builders': {}}
        for prefix, event, value in parser:
            if (prefix, event) == ('builders', 'map_key'):
                buildername = value
                ret['builders'][buildername] = {}
            elif prefix.endswith('.shortname'):
                ret['builders'][buildername]['shortname'] = value

        return ret

if __name__ == "__main__":
    load_json('allthethings.json')
```

JSON FILE LINK

Ijson online lesen: https://riptutorial.com/de/python/topic/8342/ijson

Kapitel 80: In CSV von String oder List schreiben

Einführung

Das Schreiben in eine CSV-Datei ist in den meisten Punkten dem Schreiben in eine reguläre Datei nicht unähnlich und ziemlich unkompliziert. Ich werde nach besten Kräften die einfachste und effizienteste Herangehensweise an das Problem abdecken.

Parameter

Parameter	Einzelheiten
offen (**"/ Pfad /"** , "Modus")	Geben Sie den Pfad zu Ihrer CSV-Datei an
offen (Pfad, **"Modus"**)	Festlegen des Modus zum Öffnen der Datei (Lesen, Schreiben usw.)
csv.writer (**Datei** , Trennzeichen)	Geöffnete CSV-Datei hier übergeben
csv.writer (Datei, **Trennzeichen = "**)	Trennzeichen oder Muster angeben

Bemerkungen

```
open( path, "wb")
```

`"wb"` - Schreibmodus.

Der `b` Parameter in `"wb"` wir verwendet haben, ist nur erforderlich, wenn Sie ihn im Binärmodus öffnen möchten. `"wb"` ist nur in einigen Betriebssystemen wie Windows erforderlich.

```
csv.writer ( csv_file, delimiter=',' )
```

Das hier verwendete Trennzeichen ist `,` weil jede Datenzelle in einer Zeile den Vornamen, den Nachnamen und das Alter enthalten soll. Da unsere Liste entlang der aufgespalten wird `,` auch erweist es sich sehr bequem für uns.

Examples

Grundlegendes Schreibbeispiel

```
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)
```

Anhängen eines Strings als Newline in einer CSV-Datei

```
def append_to_csv(input_string):
    with open("fileName.csv", "a") as csv_file:
        csv_file.write(input_row + "\n")
```

In CSV von String oder List schreiben online lesen:
https://riptutorial.com/de/python/topic/10862/in-csv-von-string-oder-list-schreiben

Kapitel 81: Indizieren und Schneiden

Syntax

- obj [start: stop: step]
- Slice (Stop)
- Slice (Start, Stop [, Schritt])

Parameter

Paramer	Beschreibung
`obj`	Das Objekt, aus dem Sie ein "Unterobjekt" extrahieren möchten
`start`	Der Index von `obj` , von dem aus das `obj` gestartet werden soll (denken Sie daran, dass Python `obj` ist, dh das erste Element von `obj` hat den Index `0`). Wenn nicht angegeben, ist der Standardwert `0` .
`stop`	Der (nicht inklusive) Index von `obj` , an dem das `obj` enden soll. Wenn nicht angegeben, wird standardmäßig `len(obj)` .
`step`	Ermöglicht die Auswahl nur jedes `step` . Wenn nicht angegeben, ist der Standardwert `1` .

Bemerkungen

Sie können das Konzept der Aufteilung von Zeichenfolgen mit dem der Aufteilung anderer Sequenzen vereinheitlichen, indem Sie Zeichenfolgen als unveränderliche Sammlung von Zeichen anzeigen. Dabei wird davon ausgegangen, dass ein Unicode-Zeichen durch eine Zeichenfolge der Länge 1 dargestellt wird.

In der mathematischen Notation können Sie das Slicing in Betracht ziehen, um ein halboffenes Intervall von `[start, end)` , das heißt, dass der Start enthalten ist, das Ende jedoch nicht. Die Halboffenheit des Intervalls hat den Vorteil, dass `len(x[:n]) = n` wobei `len(x) >= n` , während das zu Beginn geschlossene Intervall den Vorteil hat, dass `x[n:n+1] = [x[n]]` wobei `x` eine Liste mit `len(x) >= n` , wodurch die Konsistenz zwischen Indizierung und Slicing-Notation erhalten bleibt.

Examples

Grundschneiden

Bei jedem iterierbaren Element (z. B. einer Zeichenfolge, Liste usw.) können Sie in Python einen Teilstring oder eine Unterliste seiner Daten abschneiden und zurückgeben.

Format für das Schneiden:

```
iterable_name[start:stop:step]
```

woher,

- `start` ist der erste Index des Slice. Standardeinstellung ist 0 (der Index des ersten Elements)
- `stop` einen Punkt nach dem letzten Index des Slice. Standardeinstellung auf len (iterable)
- `step` ist die Schrittgröße (besser erklärt durch die Beispiele unten)

Beispiele:

```
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))
```

Darüber hinaus kann eine der oben genannten Funktionen mit der definierten Schrittgröße verwendet werden:

```
a[::2]        # "ace" (every 2nd element)
a[1:4:2]      # "bd" (from index 1, to index 4 (excluded), every 2nd element)
```

Indizes können negativ sein. In diesem Fall werden sie vom Ende der Sequenz aus berechnet

```
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())
```

Schrittgrößen können auch negativ sein. In diesem Fall durchläuft Slice die Liste in umgekehrter Reihenfolge:

```
a[3:1:-1]   # "dc" (from index 2 to None (default), in reverse order)
```

Dieses Konstrukt ist nützlich, um eine Wiederholung rückgängig zu machen

```
a[::-1]     # "fedcba" (from last element (default len()-1), to first, in reverse order(-1))
```

Beachten Sie, dass für negative Schritte der Standardwert `end_index None` (siehe 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)
```

Erstellen einer flachen Kopie eines Arrays

Eine schnelle Methode zum Erstellen einer Kopie eines Arrays (im Gegensatz zum Zuweisen einer Variablen mit einem anderen Verweis auf das ursprüngliche Array) ist:

```
arr[:]
```

Untersuchen wir die Syntax. `[:]` bedeutet, dass `start` , `end` und `slice` alle weggelassen werden. Der Standardwert ist `0` , `len(arr)` und `1` , was bedeutet, dass das von uns angeforderte Unterfeld alle Elemente von `arr` vom Anfang bis zum Ende enthält.

In der Praxis sieht das so aus:

```
arr = ['a', 'b', 'c']
copy = arr[:]
arr.append('d')
print(arr)    # ['a', 'b', 'c', 'd']
print(copy)   # ['a', 'b', 'c']
```

Wie Sie sehen, hat `arr.append('d') d` zu `arr` hinzugefügt, aber die `copy` blieb unverändert!

Beachten Sie, dass dies eine *flache* Kopie ist und mit `arr.copy()` identisch ist.

Objekt umkehren

Sie können Slices verwenden, um einen `str` , eine `list` oder ein `tuple` (oder im Grunde jedes Sammlungsobjekt, das das Slicing mit dem Parameter step implementiert) sehr einfach umzukehren. Hier ein Beispiel zum Umkehren eines Strings, obwohl dies auch für die anderen oben aufgeführten Typen gilt:

```
s = 'reverse me!'
s[::-1]    # '!em esrever'
```

Sehen wir uns kurz die Syntax an. `[::-1]` bedeutet, dass das Slice vom Anfang bis zum Ende der Zeichenfolge sein sollte (da `start` und `end` sind) und ein Schritt von `-1` bedeutet, dass es sich umgekehrt durch die Zeichenfolge bewegen soll.

Indizierung benutzerdefinierter Klassen: __getitem__, __setitem__ und __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]
```

Dies ermöglicht das Aufteilen und Indizieren für den Elementzugriff:

```
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
```

Während das Setzen und Löschen von Elementen nur die durch *Kommas getrennte* ganzzahlige Indizierung zulässt (keine Teilung):

```
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]
```

Slice-Zuordnung

Ein weiteres nettes Feature, das Slices verwendet, ist die Slice-Zuweisung. Mit Python können Sie neue Slices zuweisen, um alte Slices einer Liste in einem einzigen Vorgang zu ersetzen.

Das heißt, wenn Sie eine Liste haben, können Sie mehrere Mitglieder in einer einzigen Zuweisung ersetzen:

```
lst = [1, 2, 3]
lst[1:3] = [4, 5]
print(lst) # Out: [1, 4, 5]
```

Die Zuweisung sollte auch in der Größe nicht übereinstimmen. Wenn Sie also ein altes Slice durch ein neues Slice ersetzen möchten, das sich in der Größe unterscheidet, können Sie Folgendes tun:

```
lst = [1, 2, 3, 4, 5]
lst[1:4] = [6]
print(lst) # Out: [1, 6, 5]
```

Es ist auch möglich, die bekannte Slicing-Syntax zu verwenden, um beispielsweise die gesamte Liste zu ersetzen:

```
lst = [1, 2, 3]
lst[:] = [4, 5, 6]
print(lst) # Out: [4, 5, 6]
```

Oder nur die letzten beiden Mitglieder:

```
lst = [1, 2, 3]
lst[-2:] = [4, 5, 6]
print(lst) # Out: [1, 4, 5, 6]
```

Schneiden Sie Objekte

Slices sind Objekte in sich und können mit der eingebauten Funktion `slice()` in Variablen gespeichert werden. Slice-Variablen können verwendet werden, um den Code lesbarer zu machen und die Wiederverwendung zu fördern.

```
>>> 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']
```

Grundlegende Indizierung

Python-Listen sind 0-basiert, *dh* das erste Element in der Liste kann über den Index `0` aufgerufen werden

```
arr = ['a', 'b', 'c', 'd']
print(arr[0])
>> 'a'
```

Auf das zweite Element in der Liste können Sie über Index `1` zugreifen, das dritte Element auf Index `2` usw.

```
print(arr[1])
>> 'b'
print(arr[2])
>> 'c'
```

Sie können negative Indizes auch verwenden, um auf Elemente vom Ende der Liste aus zuzugreifen. z.B. Index `-1` gibt Ihnen das letzte Element der Liste und Index `-2` gibt Ihnen das vorletzte Element der Liste:

```
print(arr[-1])
>> 'd'
print(arr[-2])
>> 'c'
```

Wenn Sie versuchen, auf einen Index zuzugreifen, der nicht in der Liste enthalten ist, wird ein `IndexError` :

```
print arr[6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Indizieren und Schneiden online lesen: https://riptutorial.com/de/python/topic/289/indizieren-und-schneiden

Kapitel 82: Inkompatibilitäten von Python 2 zu Python 3

Einführung

Im Gegensatz zu den meisten Sprachen unterstützt Python zwei Hauptversionen. Seit 2008, als Python 3 veröffentlicht wurde, haben viele den Übergang vollzogen, viele jedoch nicht. In beiden Abschnitten werden die wichtigsten Unterschiede zwischen Python 2 und Python 3 erläutert.

Bemerkungen

Momentan gibt es zwei unterstützte Versionen von Python: 2.7 (Python 2) und 3.6 (Python 3). Zusätzlich erhalten die Versionen 3.3 und 3.4 Sicherheitsupdates im Quellformat.

Python 2.7 ist abwärtskompatibel mit den meisten früheren Python-Versionen und kann Python-Code in den meisten 1.x- und 2.x-Versionen von Python unverändert ausführen. Es ist breit verfügbar, mit einer umfangreichen Sammlung von Paketen. Es wird auch von den CPython-Entwicklern als veraltet eingestuft und erhält nur Sicherheits- und Bugfix-Entwicklungen. Die CPython-Entwickler beabsichtigen, diese Version der Sprache im Jahr 2020 aufzugeben.

Laut Python Enhancement Proposal 373 sind nach dem 25. Juni 2016 keine zukünftigen zukünftigen Versionen von Python 2 geplant, aber Bug-Fixes und Sicherheitsupdates werden bis 2020 unterstützt. (Es wird nicht angegeben, welches Datum in 2020 das Verfallsdatum von Python sein wird 2.)

Python 3 brach absichtlich die Rückwärtskompatibilität auf, um die Bedenken der Sprachentwickler mit dem Kern der Sprache zu berücksichtigen. Python 3 erhält neue Entwicklungen und neue Funktionen. Es ist die Version der Sprache, mit der die Sprachentwickler fortfahren möchten.

In der Zeit zwischen der ersten Version von Python 3.0 und der aktuellen Version wurden einige Funktionen von Python 3 in Python 2.6 zurückportiert, und andere Teile von Python 3 wurden erweitert, um eine mit Python 2 kompatible Syntax zu erhalten. Daher ist das Schreiben möglich Python, das sowohl für Python 2 als auch für Python 3 verwendet werden kann, indem zukünftige Importe und spezielle Module (wie **sechs**) verwendet werden.

Zukünftige Importe müssen am Anfang Ihres Moduls stehen:

```
from __future__ import print_function
# other imports and instructions go after __future__
print('Hello world')
```

Weitere Informationen zum `__future__` Modul finden Sie auf der entsprechenden Seite in der Python-Dokumentation .

Das 2to3-Tool ist ein Python-Programm, das Python 2.x-Code in Python 3.x-Code konvertiert, siehe auch die Python-Dokumentation .

Das Paket 6 enthält Dienstprogramme für die Python 2/3-Kompatibilität:

- einheitlicher Zugriff auf umbenannte Bibliotheken
- Variablen für String- / Unicode-Typen
- Funktionen für Methoden, die entfernt oder umbenannt wurden

Eine Referenz für Unterschiede zwischen Python 2 und Python 3 finden Sie hier .

Examples

Anweisung drucken vs. Druckfunktion

In Python 2 ist `print` eine Aussage:

Python 2.x 2.7

```
print "Hello World"
print                           # print a newline
print "No newline",             # add trailing comma to remove newline
print >>sys.stderr, "Error"     # print to stderr
print("hello")                  # print "hello", since ("hello") == "hello"
print()                         # print an empty tuple "()"
print 1, 2, 3                   # print space-separated arguments: "1 2 3"
print(1, 2, 3)                  # print tuple "(1, 2, 3)"
```

In Python 3 ist `print()` eine Funktion mit Schlüsselwortargumenten für allgemeine Zwecke:

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)"
```

Die Druckfunktion hat folgende Parameter:

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

`sep` trennt die Objekte, die Sie zum Drucken übergeben. Zum Beispiel:

```
print('foo', 'bar', sep='~') # out: foo~bar
print('foo', 'bar', sep='.') # out: foo.bar
```

`end` ist das Ende der Druckanweisung, gefolgt von. Zum Beispiel:

```
print('foo', 'bar', end='!') # out: foo bar!
```

Ein erneuter Druckvorgang nach einer Anweisung, die nicht am Zeilenende endet, *wird* in derselben Zeile gedruckt:

```
print('foo', end='~')
print('bar')
# out: foo~bar
```

Hinweis: Für zukünftige Kompatibilität, *print* auch in Python 2.6 ab zur Verfügung steht; Sie kann jedoch nur verwendet werden, wenn das Analysieren der *Anweisung* `print` mit deaktiviert ist

```
from __future__ import print_function
```

Diese Funktion hat genau dasselbe Format wie Python 3, jedoch fehlt der `flush` Parameter.

Weitere Informationen finden Sie in PEP 3105 .

Zeichenfolgen: Bytes im Vergleich zu Unicode

Python 2.x 2.7

In Python 2 gibt es zwei Varianten von string: Bytes mit Typ (`str`) und solche mit Text (Type) (`unicode`).

In Python 2 ist ein Objekt vom Typ `str` immer eine Bytefolge, wird jedoch häufig für Text- und Binärdaten verwendet.

Ein String-Literal wird als Byte-String interpretiert.

```
s = 'Cafe'    # type(s) == str
```

Es gibt zwei Ausnahmen: Sie können ein *Unicode- (Text-) Literal* explizit definieren, indem Sie dem Literal das Präfix `u` voranstellen:

```
s = u'Café'   # type(s) == unicode
b = 'Lorem ipsum'  # type(b) == str
```

Alternativ können Sie angeben, dass die String-Literale eines ganzen Moduls Unicode-Textliterale erstellen sollen:

```
from __future__ import unicode_literals

s = 'Café'   # type(s) == unicode
b = 'Lorem ipsum'  # type(b) == unicode
```

Um zu prüfen, ob Ihre Variable eine Zeichenfolge ist (entweder Unicode oder eine Byte-

Zeichenfolge), können Sie Folgendes verwenden:

```
isinstance(s, basestring)
```

Python 3.x 3.0

In Python 3 ist der `str` Typ ein Unicode-Texttyp.

```
s = 'Cafe'           # type(s) == str
s = 'Café'           # type(s) == str (note the accented trailing e)
```

Darüber hinaus fügte Python 3 ein `bytes` Objekt hinzu , das für binäre "Blobs" oder zum Schreiben in codierungsunabhängige Dateien geeignet ist. Um ein Byte-Objekt zu erstellen, können Sie ein String-Literal mit einem Präfix `b` oder die `encode` Methode des Strings aufrufen:

```
# Or, if you really need a byte string:
s = b'Cafe'          # type(s) == bytes
s = 'Café'.encode()  # type(s) == bytes
```

Um zu testen, ob ein Wert eine Zeichenfolge ist, verwenden Sie:

```
isinstance(s, str)
```

Python 3.x 3.3

Es ist auch möglich, String-Literalen ein Präfix " `u` voranzustellen, um die Kompatibilität zwischen Python 2- und Python 3-Codebasen zu erleichtern. Da in Python 3 standardmäßig alle Zeichenfolgen Unicode sind, hat das Voranstellen eines Zeichenfolgenlitals mit `u` keine Auswirkungen:

```
u'Cafe' == 'Cafe'
```

Das unformatierte Unicode-String-Präfix `ur` Python 2 wird jedoch nicht unterstützt:

```
>>> ur'Café'
  File "<stdin>", line 1
    ur'Café'
            ^
SyntaxError: invalid syntax
```

Beachten Sie, dass Sie ein Python 3-Textobjekt (`str`) `encode` müssen, um es in eine `bytes` Darstellung dieses Texts zu konvertieren. Die Standardkodierung dieser Methode ist UTF-8 .

Sie können `decode` , um ein `bytes` nach dem Unicode-Text zu fragen:

```
>>> b.decode()
'Café'
```

Python 2.x 2.6

Während der `bytes` Typ sowohl in Python 2 als auch in 3 vorhanden ist, ist der `unicode` Typ nur in Python 2 vorhanden. Um implizite Unicode-Zeichenfolgen von Python 3 in Python 2 zu verwenden, fügen Sie Folgendes in Ihre Codedatei ein:

```
from __future__ import unicode_literals
print(repr("hi"))
# u'hi'
```

Python 3.x 3.0

Ein weiterer wichtiger Unterschied ist, dass die Indizierung von Bytes in Python 3 zu einer `int` Ausgabe führt:

```
b"abc"[0] == 97
```

Wenn Sie in einer Größe von 1 schneiden, erhalten Sie ein Objekt mit der Länge 1 Byte:

```
b"abc"[0:1] == b"a"
```

Darüber hinaus behebt Python 3 einige ungewöhnliche Verhaltensweisen beim Unicode, dh das Umkehren von Bytezeichenfolgen in Python 2. Zum Beispiel wurde das folgende Problem behoben:

```
# -*- 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
```

Integer Division

Das Standard- **Divisionssymbol** (`/`) funktioniert in Python 3 und Python 2 unterschiedlich, wenn es auf Ganzzahlen angewendet wird.

Beim Dividieren einer Ganzzahl durch eine andere Ganzzahl in Python 3 stellt die Divisionsoperation `x / y` eine **echte Division dar** (verwendet `__truediv__` Methode `__truediv__`) und erzeugt ein Fließkomma-Ergebnis. Die gleiche Operation in Python 2 stellt eine **klassische Division dar** , die das Ergebnis in Richtung negative Unendlichkeit abrundet (auch bekannt als das *Wort ergreifen*).

Zum Beispiel:

Code	Python 2-Ausgabe	Python 3-Ausgabe
`3 / 2`	1	1,5
`2 / 3`	0	0,6666666666666666
`-3 / 2`	-2	-1,5

Das Rundungsverhalten gegen Null wurde in Python 2.2 nicht mehr unterstützt , bleibt aber aus Gründen der Abwärtskompatibilität in Python 2.7 und wurde in Python 3 entfernt.

Hinweis: Um ein *Fließergebnis* in Python 2 (ohne Rundung) zu erhalten, können wir einen der Operanden mit dem Dezimalpunkt angeben. Das obige Beispiel von `2/3` das in Python 2 `0` ergibt, wird als `2 / 3.0` oder `2.0 / 3` oder `2.0/3.0` , um `0.6666666666666666` zu erhalten

Code	Python 2-Ausgabe	Python 3-Ausgabe
`3.0 / 2.0`	1,5	1,5
`2 / 3.0`	0,6666666666666666	0,6666666666666666
`-3.0 / 2`	-1,5	-1,5

Es gibt auch den **Floor-Division-Operator** (`//`), der in beiden Versionen gleich funktioniert: Er wird auf die nächste Ganzzahl abgerundet. (obwohl bei Verwendung von Floats ein Float zurückgegeben wird) In beiden Versionen ist der `//` Operator `__floordiv__` .

Code	Python 2-Ausgabe	Python 3-Ausgabe
`3 // 2`	1	1
`2 // 3`	0	0
`-3 // 2`	-2	-2
`3.0 // 2.0`	1,0	1,0
`2.0 // 3`	0,0	0,0
`-3 // 2.0`	-2,0	-2,0

Man kann eine echte Division oder eine Bodendivision explizit durch native Funktionen im `operator` erzwingen:

```
from operator import truediv, floordiv
assert truediv(10, 8) == 1.25           # equivalent to `/` in Python 3
assert floordiv(10, 8) == 1             # equivalent to `//`
```

Die Verwendung von Bedienfunktionen für jeden Bereich kann zwar klar und explizit sein, jedoch langwierig sein. Das Verhalten des Operators `/` ändern wird häufig bevorzugt. Es ist üblich, das typische Abteilungsverhalten zu eliminieren, indem `from __future__ import division` die erste Anweisung in jedem Modul hinzugefügt wird:

```
# needs to be the first statement in a module
from __future__ import division
```

Code	Python 2-Ausgabe	Python 3-Ausgabe
`3 / 2`	1,5	1,5
`2 / 3`	0,6666666666666666	0,6666666666666666
`-3 / 2`	-1,5	-1,5

`from __future__ import division` garantiert, dass der Operator `/` eine echte Division darstellt, und zwar nur innerhalb der Module, die den `__future__` Import enthalten. `__future__` gibt es keine zwingenden Gründe, die nicht in allen neuen Modulen aktiviert werden.

Hinweis : Einige andere Programmiersprachen verwenden eine *Rundung gegen Null* (Verkürzung) und nicht wie Python eine *negative Unendlichkeit* (dh in diesen Sprachen `-3 / 2 == -1`). Dieses Verhalten kann zu Verwirrung beim Portieren oder Vergleichen von Code führen.

Hinweis zu Float-Operanden : Als Alternative zur `from __future__ import division` kann man das übliche Divisionssymbol `/` und sicherstellen, dass mindestens einer der Operanden ein Float ist: `3 / 2.0 == 1.5` . Dies kann jedoch als schlechte Praxis betrachtet werden. Es ist einfach zu einfach, `average = sum(items) / len(items)` zu schreiben und vergessen, eines der Argumente in Float zu setzen. Darüber hinaus kann es vorkommen, dass solche Fälle während des Testens häufig nicht beachtet werden, z. B. wenn Sie ein Array mit `float` Werten testen, aber ein Array mit `int` Werten in der Produktion erhalten. Wenn derselbe Code in Python 3 verwendet wird, funktionieren Programme, die erwarten, dass 3/2 `3 / 2 == 1` True ist, nicht ordnungsgemäß.

In PEP 238 finden Sie detailliertere Gründe, warum der Abteilungsoperator in Python 3 geändert wurde und warum eine Aufteilung nach alter Art vermieden werden sollte.

Weitere *Informationen* zur Unterteilung finden Sie im Thema *Simple Math* .

Reduzieren ist kein integrierter Bestandteil mehr

In Python 2, `reduce` ist entweder als eine integrierte Funktion oder von dem `functools` - Paket (Version 2.6 ab), während in Python 3 `reduce` nur verfügbar ist `functools` . Die Syntax für das `reduce` in Python2 und Python3 ist jedoch dieselbe und wird `reduce(function_to_reduce, list_to_reduce)` .

Nehmen wir als Beispiel an, eine Liste auf einen einzelnen Wert zu reduzieren, indem Sie jede der benachbarten Zahlen teilen. Hier verwenden wir die `truediv` Funktion aus der `operator` Bibliothek.

In Python 2.x ist es so einfach wie:

Python 2.x 2.3

```
>>> my_list = [1, 2, 3, 4, 5]
>>> import operator
>>> reduce(operator.truediv, my_list)
0.008333333333333333
```

In Python 3.x wird das Beispiel etwas komplizierter:

Python 3.x 3.0

```
>>> my_list = [1, 2, 3, 4, 5]
>>> import operator, functools
>>> functools.reduce(operator.truediv, my_list)
0.008333333333333333
```

Wir können auch `from functools import reduce` , um das Aufrufen von `reduce` mit dem Namensraumnamen zu vermeiden.

Unterschiede zwischen Range- und Xrange-Funktionen

In Python 2 gibt die `range` Funktion eine Liste zurück, während `xrange` ein spezielles `xrange` Objekt erstellt. `xrange` handelt es sich um eine unveränderliche Sequenz, die im Gegensatz zu anderen integrierten Sequenztypen keine Slicing-Funktion unterstützt und keine `index` oder `count` Methoden hat:

Python 2.x 2.3

```
print(range(1, 10))
# Out: [1, 2, 3, 4, 5, 6, 7, 8, 9]

print(isinstance(range(1, 10), list))
# Out: True

print(xrange(1, 10))
# Out: xrange(1, 10)

print(isinstance(xrange(1, 10), xrange))
# Out: True
```

In Python 3 wurde `xrange` um die `range` erweitert, wodurch nun ein `range` wird. Es gibt keinen `xrange` Typ:

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
```

Darüber hinaus unterstützt `range` seit Python 3.2 auch das Schneiden, `index` und `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
```

Die Verwendung eines speziellen Sequenztyps anstelle einer Liste hat den Vorteil, dass der Interpreter keinen Speicher für eine Liste reservieren und diese füllen muss:

Python 2.x 2.3

```
# range(10000000000000000)
# The output would be:
# Traceback (most recent call last):
#  File "<stdin>", line 1, in <module>
# MemoryError

print(xrange(10000000000000000))
# Out: xrange(10000000000000000)
```

Da das letztere Verhalten im Allgemeinen erwünscht ist, wurde das erstere in Python 3 entfernt. Wenn Sie noch eine Liste in Python 3 haben möchten, können Sie den Konstruktor `list()` einfach für ein `range` :

Python 3.x 3.0

```
print(list(range(1, 10)))
# Out: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Kompatibilität

Um die Kompatibilität zwischen beiden Versionen von Python 2.x und Python 3.x aufrechtzuerhalten, können Sie das `builtins` Modul aus dem externen Paket `future` , um sowohl *Vorwärtskompatibilität* als auch *Rückwärtskompatibilität* zu erreichen:

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
```

Der `range` in der `future` Bibliothek unterstützt das Aufteilen, `index` und `count` in allen Python-Versionen, genau wie die in Python 3.2+ integrierte Methode.

Iterables auspacken

Python 3.x 3.0

In Python 3 können Sie ein iterierbares Paket auspacken, ohne die genaue Anzahl der darin enthaltenen Elemente zu kennen, und sogar eine Variable muss das Ende des iterierbaren Elements enthalten. Dafür geben Sie eine Variable an, die eine Liste von Werten erfassen kann. Dazu wird vor dem Namen ein Sternchen eingefügt. Zum Beispiel das Auspacken einer `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
```

Hinweis : Wenn Sie die `*variable` , ist die `variable` immer eine Liste, auch wenn der Originaltyp keine Liste war. Abhängig von der Anzahl der Elemente in der ursprünglichen Liste kann es null oder mehr Elemente enthalten.

```
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
```

Entpacken eines `str` :

```
begin, *tail = "Hello"
print(begin)
# Out: 'H'
print(tail)
# Out: ['e', 'l', 'l', 'o']
```

Beispiel für das Auspacken eines `date` ; _ wird in diesem Beispiel als Wegwerfvariable verwendet

(wir interessieren uns nur für den `year`):

```
person = ('John', 'Doe', (10, 16, 2016))
*_, (*_, year_of_birth) = person
print(year_of_birth)
# Out: 2016
```

Es ist erwähnenswert, dass Sie, da `*` eine variable Anzahl von Elementen auffrisst, nicht zwei `*` s *für dieselbe Iteration* in einer Zuweisung haben können :

```
*head, *tail = [1, 2]
# Out: SyntaxError: two starred expressions in assignment
```

Python 3.x 3.5

Bisher haben wir das Auspacken in Aufgaben besprochen. `*` und `**` wurden in Python 3.5 erweitert . Es ist jetzt möglich, mehrere Auspackvorgänge in einem Ausdruck auszuführen:

```
{*range(4), 4, *(5, 6, 7)}
# Out: {0, 1, 2, 3, 4, 5, 6, 7}
```

Python 2.x 2.0

Es ist auch möglich, ein iterierbares in Funktionsargumente zu entpacken:

```
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

Beim Auspacken eines Wörterbuchs werden zwei benachbarte Sterne `**` (PEP 448) verwendet:

```
tail = {'y': 2, 'z': 3}
{'x': 1, **tail}
 # Out: {'x': 1, 'y': 2, 'z': 3}
```

Dies ermöglicht das Überschreiben alter Werte und das Zusammenführen von Wörterbüchern.

```
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 entfernte das Tupel beim Auspacken von Funktionen. Daher funktioniert das Folgende in Python 3 nicht

```
# 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
```

Ausführliche Informationen finden Sie in PEP 3113 .

Ausnahmen erhöhen und behandeln

Dies ist die Python 2 Syntax, die Kommas beachten `,` auf die `raise` und `except` Linien:

Python 2.x 2.3

```
try:
    raise IOError, "input/output error"
except IOError, exc:
    print exc
```

In Python 3, die `,` wird Syntax fallen gelassen und ersetzt durch die Klammer und die `as` Stichwort:

```
try:
    raise IOError("input/output error")
except IOError as exc:
    print(exc)
```

Aus Gründen der Rückwärtskompatibilität ist die Python 3-Syntax auch in Python 2.6 verfügbar. Sie sollte daher für alle neuen Codes verwendet werden, die nicht mit früheren Versionen kompatibel sind.

Python 3.x 3.0

Python 3 fügt außerdem eine Ausnahme-Verkettung hinzu , wobei Sie signalisieren können, dass eine andere Ausnahme die *Ursache* für diese Ausnahme war. Zum Beispiel

```
try:
    file = open('database.db')
except FileNotFoundError as e:
    raise DatabaseError('Cannot open {}') from e
```

Die Ausnahme, die in der `except` Anweisung `__cause__` wird, ist vom Typ `DatabaseError` , aber die ursprüngliche Ausnahme wird als das Attribut `__cause__` dieser Ausnahme markiert. Wenn das Traceback angezeigt wird, wird die ursprüngliche Ausnahme auch im Traceback angezeigt:

```
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')
```

Wenn Sie in einem Wurf `except` Block *ohne* explizite Verkettungs:

```
try:
    file = open('database.db')
except FileNotFoundError as e:
    raise DatabaseError('Cannot open {}')
```

Das Traceback ist

```
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

Keiner wird in Python 2.x unterstützt. Die ursprüngliche Ausnahme und ihr Traceback gehen verloren, wenn im Ausnahmeblock eine andere Ausnahme ausgelöst wird. Der folgende Code kann zur Kompatibilität verwendet werden:

```
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

Um die zuvor geworfene Ausnahme zu "vergessen", verwenden Sie die `raise from None`

```
try:
    file = open('database.db')
except FileNotFoundError as e:
    raise DatabaseError('Cannot open {}') from None
```

Nun wäre das Traceback einfach

```
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
DatabaseError('Cannot open database.db')
```

Oder um es mit Python 2 und 3 kompatibel zu machen, können Sie das Paket sechs wie folgt verwenden:

```
import six
try:
    file = open('database.db')
except FileNotFoundError as e:
    six.raise_from(DatabaseError('Cannot open {}'), None)
```

.next () -Methode für Iteratoren umbenannt

In Python 2 kann ein Iterator durchlaufen werden, indem eine Methode aufgerufen wird, die als `next` auf dem Iterator selbst aufgerufen wird:

Python 2.x 2.3

```
g = (i for i in range(0, 3))
g.next()  # Yields 0
g.next()  # Yields 1
g.next()  # Yields 2
```

In Python 3 wurde die `.next` Methode in `.__next__` umbenannt, `.__next__` ihre "magische" Rolle bestätigt, sodass der Aufruf von `.next` einen `AttributeError` `.next` . Der korrekte Weg, um auf diese Funktionalität in Python 2 und Python 3 zuzugreifen, besteht darin, die `next` *Funktion* mit dem Iterator als Argument aufzurufen.

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
```

Dieser Code ist von Version 2.6 bis zu aktuellen Versionen portierbar.

Vergleich verschiedener Typen

Python 2.x 2.3

Objekte verschiedener Typen können verglichen werden. Die Ergebnisse sind willkürlich, aber konsistent. Sie sind so angeordnet, dass `None` weniger als alles andere ist, numerische Typen sind kleiner als nicht numerische Typen und alles andere ist lexikographisch nach Typ geordnet. Ein `int` ist also kleiner als ein `str` und ein `tuple` ist größer als eine `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
```

Dies wurde ursprünglich durchgeführt, damit eine Liste gemischter Typen sortiert werden konnte und Objekte nach Typ gruppiert wurden:

```
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

Beim Vergleich verschiedener (nicht numerischer) Typen wird eine Ausnahme ausgelöst:

```
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()
```

Um gemischte Listen in Python 3 nach Typen zu sortieren und Kompatibilität zwischen Versionen zu erreichen, müssen Sie einen Schlüssel für die sortierte Funktion angeben:

```
>>> 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}]
```

Wenn Sie `str` als `key` verwenden, wird jedes Element nur zu Vergleichszwecken temporär in eine Zeichenfolge umgewandelt. Anschließend wird die Zeichenfolgendarstellung mit `[` , `'` , `{` oder `0-9` beginnend `0-9` Sie kann diese (und alle folgenden Zeichen) sortieren.

Benutzereingabe

In Python 2 werden Benutzereingaben mit der Funktion `raw_input` akzeptiert.

Python 2.x 2.3

```
user_input = raw_input()
```

Während in Python 3 eine Benutzereingabe unter Verwendung der akzeptierten `input`

Python 3.x 3.0

```
user_input = input()
```

In Python 2 ist der `input` wird Funktionseingabe akzeptieren und *interpretieren.* Dies ist zwar nützlich, hat jedoch mehrere Sicherheitsaspekte und wurde in Python 3 entfernt. Für den Zugriff auf dieselbe Funktionalität kann `eval(input())` verwendet werden.

Um ein Skript für die beiden Versionen portabel zu halten, können Sie den folgenden Code oben in Ihr Python-Skript einfügen:

```
try:
    input = raw_input
except NameError:
    pass
```

Wörterbuchmethode ändert sich

In Python 3 unterscheiden sich viele der Wörterbuchmethoden ziemlich von Python 2, und viele wurden ebenfalls entfernt: `has_key` , `iter*` und `view*` sind weg. Anstelle von `d.has_key(key)` , der lange veraltet war, muss man jetzt `key in d` .

In Python 2 geben die `keys` , `values` und `items` Wörterbuchmethoden Listen zurück. In Python 3 geben sie stattdessen *Ansichtsobjekte zurück* . Die Ansichtsobjekte sind keine Iteratoren und unterscheiden sich in zweierlei Hinsicht von ihnen:

- Sie haben Größe (man kann die `len` Funktion verwenden)
- Sie können viele Male wiederholt werden

Wie bei Iteratoren spiegeln sich auch die Änderungen im Wörterbuch in den Ansichtsobjekten wider.

Python 2.7 hat diese Methoden aus Python 3 zurückportiert; Sie sind als `viewkeys` , `viewvalues` und `viewitems` . Um Python 2-Code in Python 3-Code umzuwandeln, lauten die entsprechenden Formulare:

- `d.keys()` , `d.values()` und `d.items()` von Python 2 sollten in `list(d.keys())` , `list(d.values())` und `list(d.items())`
- `d.iterkeys()` , `d.itervalues()` und `d.iteritems()` sollten in `iter(d.keys())` oder noch besser `iter(d)` geändert werden. `iter(d.values())` bzw. `iter(d.items())`
- und schließlich kann die Methode Python 2.7 `d.viewkeys()` , `d.viewvalues()` und `d.viewitems()` durch `d.keys()` , `d.values()` und `d.items()` .

Das Portieren von Python 2-Code, der während der Mutation über Wörterbuchschlüssel, -werte oder -elemente *iteriert* , ist manchmal schwierig. Erwägen:

```
d = {'a': 0, 'b': 1, 'c': 2, '!': 3}
for key in d.keys():
    if key.isalpha():
        del d[key]
```

Der Code sieht so aus, als würde er in Python 3 ähnlich funktionieren, aber die `keys` Methode gibt

ein Ansichtsobjekt zurück, keine Liste. Wenn das Wörterbuch die Größe ändert, während es iteriert wird, stürzt der Python 3-Code mit der `RuntimeError: dictionary changed size during iteration` . Die Lösung ist natürlich, `for key in list(d)` richtig zu schreiben.

In ähnlicher Weise verhalten sich Ansichtsobjekte anders als Iteratoren: Sie können `next()` für sie verwenden, und Sie können *die* Iteration nicht *fortsetzen* . es würde stattdessen neu starten; wenn Python 2 - Code den Rückgabewert gibt `d.iterkeys()` , `d.itervalues()` oder `d.iteritems()` ein Verfahren , das ein Iterator anstelle eines *iterable* erwartet, dass dann sollte `iter(d)` , `iter(d.values())` oder `iter(d.items())` in Python 3.

exec-Anweisung ist eine Funktion in Python 3

In Python 2 ist `exec` eine Anweisung mit einer speziellen Syntax: `exec code [in globals[, locals]]`. In Python 3 ist `exec` nun eine Funktion: `exec(code, [, globals[, locals]])` und die Python 2-Syntax `SyntaxError` einen `SyntaxError` .

Da `print` von der Anweisung in eine Funktion `__future__` wurde, wurde auch ein `__future__` Import hinzugefügt. Es gibt jedoch keine `from __future__ import exec_function` , da dies nicht erforderlich ist: Die exec-Anweisung in Python 2 kann auch mit einer Syntax verwendet werden, die genau dem Aufruf der `exec` Funktion in Python 3 entspricht. Sie können also die Anweisungen ändern

Python 2.x 2.3

```
exec 'code'
exec 'code' in global_vars
exec 'code' in global_vars, local_vars
```

zu formen

Python 3.x 3.0

```
exec('code')
exec('code', global_vars)
exec('code', global_vars, local_vars)
```

und die letzteren Formen funktionieren garantiert in Python 2 und Python 3 identisch.

hasattr-Funktionsfehler in Python 2

Wenn in Python 2 eine Eigenschaft einen Fehler `hasattr` , ignoriert `hasattr` diese Eigenschaft und gibt `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
```

Dieser Fehler wurde in Python3 behoben. Wenn Sie also Python 2 verwenden, verwenden Sie

```
try:
    a.get
except AttributeError:
    print("no get property!")
```

oder `getattr` stattdessen `getattr`

```
p = getattr(a, "get", None)
if p is not None:
    print(p)
else:
    print("no get property!")
```

Umbenannte Module

Einige Module der Standardbibliothek wurden umbenannt:

Alte Bezeichnung	Neuer Name
_winreg	Winreg
ConfigParser	configparser
copy_reg	copyreg
Warteschlange	Warteschlange
SocketServer	Socketserver
_markupbase	Markupbase
Repr	verdanken
test.test_support	test.support
Tkinter	tkinter
tkFileDialog	tkinter.filedialog

Alte Bezeichnung	Neuer Name
urllib / urllib2	urllib, urllib.parse, urllib.error, urllib.response, urllib.request, urllib.robotparser

Einige Module wurden sogar von Dateien in Bibliotheken konvertiert. Nehmen Sie als Beispiel tkinter und urllib von oben.

Kompatibilität

Wenn Sie die Kompatibilität zwischen den Versionen Python 2.x und 3.x beibehalten, können Sie das `future` externe Paket verwenden , um das Importieren von Standard-Bibliothekspaketen der obersten Ebene mit Python 3.x-Namen in Python 2.x-Versionen zu ermöglichen.

Oktalkonstanten

In Python 2 kann ein Oktal-Literal als definiert werden

```
>>> 0755  # only Python 2
```

Um Querkompatibilität zu gewährleisten, verwenden Sie

```
0o755  # both Python 2 and Python 3
```

Alle Klassen sind in Python 3 neue Klassen.

In Python `3.x` alle Klassen Klassen im *neuen Stil* . Wenn Sie eine neue Klasse definieren, erbt Python implizit von `object` . Die Angabe eines `object` in einer `class` ist daher vollständig optional:

Python 3.x 3.0

```
class X: pass
class Y(object): pass
```

Beide Klassen enthalten jetzt `object` in ihrer `mro` (Methodenauflösungsreihenfolge):

Python 3.x 3.0

```
>>> X.__mro__
(__main__.X, object)

>>> Y.__mro__
(__main__.Y, object)
```

In Python `2.x` Klassen standardmäßig Klassen im alten Stil. Sie erben nicht implizit vom `object` . Dies bewirkt , dass die Semantik von Klassen je nach unterscheiden , wenn wir ausdrücklich hinzufügen `object` als `class` :

Python 2.x 2.3

```
class X: pass
class Y(object): pass
```

Wenn wir versuchen, `__mro__` von `Y` zu drucken, wird in diesem Fall eine ähnliche Ausgabe wie in der Python `3.x` Ausgabe `__mro__` :

Python 2.x 2.3

```
>>> Y.__mro__
(<class '__main__.Y'>, <type 'object'>)
```

Dies geschieht, weil wir bei der Definition von `Y` explizit von Objekt geerbt haben: `class Y(object): pass` . Für die Klasse `X` die *nicht* vom Objekt erbt, ist das `__mro__` Attribut nicht vorhanden. Der Zugriff auf dieses Objekt führt zu einem `AttributeError` .

Um **die Kompatibilität** zwischen beiden Python-Versionen **sicherzustellen** , können Klassen mit `object` als Basisklasse definiert werden:

```
class mycls(object):
    """I am fully compatible with Python 2/3"""
```

Wenn `__metaclass__` Variable `__metaclass__` im globalen Gültigkeitsbereich auf `type` gesetzt ist, sind alle nachfolgend definierten Klassen in einem bestimmten Modul implizit im neuen Stil, ohne explizit von einem `object` erben zu müssen:

```
__metaclass__ = type

class mycls:
    """I am also fully compatible with Python 2/3"""
```

Entfernte Operatoren <> und ``, auch mit! = Und repr ()

In Python 2 ist `<>` ein Synonym für `!=` ; Ebenso ist `` `foo` `` ein Synonym für `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
```

encode / decode to hex nicht mehr verfügbar

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'
```

Wie in der Fehlermeldung vorgeschlagen, können Sie jedoch das `codecs` Modul verwenden, um dasselbe Ergebnis zu erzielen:

```
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'
```

Beachten Sie, dass `codecs.encode` ein `bytes` Objekt zurückgibt. Um ein `str` Objekt zu erhalten, `decode` einfach in ASCII:

```
codecs.encode(b'\x1d\xea\xdb\xee\xff', 'hex').decode('ascii')
# Out: '1deadbeeff'
```

CMP-Funktion in Python 3 entfernt

In Python 3 wurde die integrierte `cmp` -Funktion zusammen mit der speziellen Methode `__cmp__` .

Aus der Dokumentation:

> Die Funktion `cmp()` sollte als verschwunden betrachtet werden, und die spezielle Methode `__cmp__()` wird nicht mehr unterstützt. Verwenden Sie `__lt__()` zum Sortieren, `__eq__()` mit `__hash__()` und andere umfassende Vergleiche nach Bedarf. (Wenn Sie wirklich die `cmp()` Funktionalität benötigen, können Sie den Ausdruck `(a > b) - (a < b)` als Entsprechung für `cmp(a, b)` .)

Darüber hinaus akzeptieren alle integrierten Funktionen, die den Parameter `cmp` akzeptiert haben, nur noch den Parameter für das `key` .

Im `functools` Modul gibt es auch die nützliche Funktion `cmp_to_key(func)` , mit der Sie eine `cmp` Funktion in eine `key cmp` Funktion konvertieren können:

> Wandeln Sie eine Vergleichsfunktion im alten Stil in eine Schlüsselfunktion um. Wird mit Werkzeugen verwendet, die Schlüsselfunktionen (z. B. `sorted()` , `min()` , `max()` , `heapq.nlargest()` , `heapq.nsmallest()` , `itertools.groupby()`) `itertools.groupby()` . Diese Funktion wird hauptsächlich als Übergangswerkzeug für Programme verwendet, die von Python 2 konvertiert werden und die Verwendung von Vergleichsfunktionen unterstützen.

Durchgesickerte Variablen im Listenverständnis

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!'
```

Wie aus dem Beispiel zu ersehen ist, wurde in Python 2 der Wert von `x` durchgesickert: Es maskierte `hello world!` und `U` ausgedruckt, da dies der letzte Wert von `x` als die Schleife endete.

In Python 3 `x` die ursprünglich definierte `hello world!` gedruckt `hello world!` , da die lokale Variable

aus dem Listenverständnis keine Variablen aus dem umgebenden Bereich maskiert.

Außerdem haben weder Generatorausdrücke (in Python seit 2.5 verfügbar) noch Dictionary oder Set Comprehensions (die von Python 3 auf Python 2.7 zurückgespielt wurden) Variablen in Python 2.

Beachten Sie, dass Variablen sowohl in Python 2 als auch in Python 3 bei Verwendung einer for-Schleife in den umgebenden Bereich eindringen:

```
x = 'hello world!'
vowels = []
for x in 'AEIOU':
    vowels.append(x)
print(x)
# Out: 'U'
```

Karte()

`map()` ist ein integriertes Element, das zum Anwenden einer Funktion auf Elemente einer Iteration nützlich ist. In Python 2 gibt `map` eine Liste zurück. In Python 3 gibt `map` ein *Kartenobjekt zurück* , bei dem es sich um einen Generator handelt.

```
# Python 2.X
>>> map(str, [1, 2, 3, 4, 5])
['1', '2', '3', '4', '5']
>>> type(_)
>>> <class 'list'>

# Python 3.X
>>> map(str, [1, 2, 3, 4, 5])
<map object at 0x*>
>>> type(_)
<class 'map'>

# We need to apply map again because we "consumed" the previous map....
>>> map(str, [1, 2, 3, 4, 5])
>>> list(_)
['1', '2', '3', '4', '5']
```

In Python 2 können Sie `None` als Identitätsfunktion übergeben. Dies funktioniert in Python 3 nicht mehr.

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
```

Wenn Sie in Python 2 mehr als ein iterierbares Argument übergeben, `itertools.izip_longest` `map` die kürzeren iterierbaren `itertools.izip_longest` mit `None` (ähnlich wie `itertools.izip_longest`). In Python 3 stoppt die Iteration nach der kürzesten Iteration.

In Python 2:

Python 2.x 2.3

```
>>> map(None, [1, 2, 3], [1, 2], [1, 2, 3, 4, 5])
[(1, 1, 1), (2, 2, 2), (3, None, 3), (None, None, 4), (None, None, 5)]
```

In Python 3:

Python 3.x 3.0

```
>>> list(map(lambda x, y, z: (x, y, z), [1, 2, 3], [1, 2], [1, 2, 3, 4, 5]))
[(1, 1, 1), (2, 2, 2)]

# to obtain the same padding as in Python 2 use zip_longest from itertools
>>> import itertools
>>> list(itertools.zip_longest([1, 2, 3], [1, 2], [1, 2, 3, 4, 5]))
[(1, 1, 1), (2, 2, 2), (3, None, 3), (None, None, 4), (None, None, 5)]
```

Hinweis : Verwenden **Sie** anstelle von `map` List Comprehensions, die mit Python 2/3 kompatibel sind. Ersetzen der `map(str, [1, 2, 3, 4, 5])` :

```
>>> [str(i) for i in [1, 2, 3, 4, 5]]
['1', '2', '3', '4', '5']
```

filter (), map () und zip () geben Iteratoren statt Sequenzen zurück

Python 2.x 2.7

In Python 2- `filter` eingebaute Funktionen für `map` und `zip` eine Sequenz zurück. `map` und `zip` immer eine Liste zurück, während bei `filter` der Rückgabetyp vom Typ des angegebenen Parameters abhängt:

```
>>> 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

In Python 3 `filter` stattdessen `map` und `zip` Iterator:

```
>>> it = filter(lambda x: x.isalpha(), 'a1b2c3')
```

```
>>> it
<filter object at 0x00000098A55C2518>
>>> ''.join(it)
'abc'
>>> it = map(lambda x: x * x, [0, 1, 2])
>>> it
<map object at 0x000000E0763C2D30>
>>> list(it)
[0, 1, 4]
>>> it = zip([0, 1, 2], [3, 4, 5])
>>> it
<zip object at 0x000000E0763C52C8>
>>> list(it)
[(0, 3), (1, 4), (2, 5)]
```

Da Python 2 `itertools.izip` gleichwertig zu Python 3 ist, wurde `zip` `izip` auf Python 3 entfernt.

Absolute / Relative Importe

In Python 3 ändert PEP 404 die Funktionsweise von Importen von Python 2. *Implizite relative* Importe sind in Paketen nicht mehr zulässig, und Importe `from ... import *` sind nur im Code auf Modulebene zulässig.

So erzielen Sie Python 3-Verhalten in Python 2:

- Die absolute `from __future__ import absolute_import` kann mit `from __future__ import absolute_import`
- *explizite relative* Importe werden anstelle *impliziter relativer* Importe gefördert

---

Zur Verdeutlichung kann ein Modul in Python 2 den Inhalt eines anderen Moduls, das sich in demselben Verzeichnis befindet, wie folgt importieren:

```
import foo
```

Beachten Sie, dass der Speicherort von `foo` allein durch die Importanweisung mehrdeutig ist. Von dieser Art des impliziten relativen Imports wird daher abgeraten, explizite relative Importe zu bevorzugen, die wie folgt aussehen:

```
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
```

Der Punkt `.` erlaubt eine explizite Deklaration der Modulposition innerhalb des Verzeichnisbaums.

Mehr zu den relativen Importen

Betrachten Sie ein benutzerdefiniertes Paket namens `shapes` . Die Verzeichnisstruktur sieht wie folgt aus:

```
shapes
├── __init__.py
|
├── circle.py
|
├── square.py
|
└── triangle.py
```

`circle.py` , `square.py` und `triangle.py` importieren alle `util.py` als Modul. Wie beziehen sie sich auf ein Modul auf derselben Ebene?

```
 from . import util # use util.PI, util.sq(x), etc
```

ODER

```
 from .util import * #use PI, sq(x), etc to call functions
```

Die . wird für relative Importe auf gleicher Ebene verwendet.

Betrachten Sie nun ein alternatives Layout des `shapes` Moduls:

```
shapes
├── __init__.py
|
├── circle
|   ├── __init__.py
|   └── circle.py
|
├── square
|   ├── __init__.py
|   └── square.py
|
├── triangle
|   ├── __init__.py
|   ├── triangle.py
|
└── util.py
```

Wie beziehen sich diese 3 Klassen nun auf util.py?

```
 from .. import util # use util.PI, util.sq(x), etc
```

ODER

```
 from ..util import * # use PI, sq(x), etc to call functions
```

Das . . wird für relative Importe auf übergeordneter Ebene verwendet. Mehr hinzufügen . s mit der Anzahl der Ebenen zwischen dem Elternteil und dem Kind.

Datei I / O

`file` ist in 3.x kein eingebauter Name mehr (`open` funktioniert noch).

Interne Details der Datei-E / A wurden in das Standardbibliothek- `io` Modul verschoben, das auch die neue Heimat von `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'
```

Der Dateimodus (Text vs. Binär) bestimmt nun den Datentyp, der durch Lesen einer Datei (und des zum Schreiben erforderlichen Typs) erzeugt wird:

```
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
```

Die Kodierung für Textdateien wird standardmäßig auf das gesetzt, was von `locale.getpreferredencoding(False)` . Um eine Kodierung explizit anzugeben, verwenden Sie den Parameter `encoding` das `encoding` :

```
with open('old_japanese_poetry.txt', 'shift_jis') as text:
    haiku = text.read()
```

Die Funktion round () ist die Funktion "break-break" und "return"

runde () krawatte brechen

In Python 2 wird `round()` für eine Zahl verwendet, die nahe an zwei ganzen Zahlen liegt, diejenige, die am weitesten von 0 entfernt ist. Zum Beispiel:

Python 2.x 2.7

```
round(1.5)  # Out: 2.0
round(0.5)  # Out: 1.0
round(-0.5)  # Out: -1.0
round(-1.5)  # Out: -2.0
```

In Python 3 gibt `round()` jedoch die gerade ganze Zahl (auch als *Rundung der Bankiers bezeichnet*) zurück. Zum Beispiel:

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
```

Die round () - Funktion folgt der *halben bis geraden Rundungsstrategie* , bei der halbe Zahlen auf die nächste gerade ganze Zahl `round(2.5)` zum Beispiel gibt `round(2.5)` jetzt 2 statt 3.0 zurück).

Wie in Wikipedia angegeben , wird dies auch als *unverzerrte Rundung* , *konvergente Rundung* , *Statistiker-Rundung* , *niederländische Rundung* , *Gauß-Rundung* oder *ungerade Rundung bezeichnet* .

Eine halbe bis gleichmäßige Rundung ist Teil des IEEE 754- Standards und auch der Standardrundungsmodus in Microsofts .NET.

Diese Rundungsstrategie verringert tendenziell den gesamten Rundungsfehler. Da im Durchschnitt die Anzahl der gerundeten Zahlen gleich der Anzahl der abgerundeten Zahlen ist, werden Rundungsfehler aufgehoben. Andere Rundungsmethoden tendieren eher dazu, den Durchschnittsfehler nach oben oder nach unten zu neigen.

---

round () Rückgabetyp

Die `round()` Funktion gibt einen `float` Typ in Python 2.7 zurück

Python 2.x 2.7

```
round(4.8)
# 5.0
```

Wenn in Python 3.0 das zweite Argument (Anzahl der Ziffern) weggelassen wird, wird ein `int` .

Python 3.x 3.0

```
round(4.8)
# 5
```

Richtig, Falsch und Keiner

In Python 2 sind `True` , `False` und `None` integrierte Konstanten. Dies bedeutet, dass es möglich ist, sie neu zuzuweisen.

Python 2.x 2.0

```
True, False = False, True
True   # False
False  # True
```

Mit Python 2.4 ist dies mit `None` nicht möglich.

Python 2.x 2.4

```
None = None  # SyntaxError: cannot assign to None
```

In Python 3 sind `True` , `False` und `None` jetzt Schlüsselwörter.

Python 3.x 3.0

```
True, False = False, True  # SyntaxError: can't assign to keyword

None = None  # SyntaxError: can't assign to keyword
```

Rückgabewert beim Schreiben in ein Dateiobjekt

Wenn Sie direkt in ein Dateihandle schreiben, wird in Python 2 `None` :

Python 2.x 2.3

```
hi = sys.stdout.write('hello world\n')
# Out: hello world
type(hi)
# Out: <type 'NoneType'>
```

In Python 3 wird beim Schreiben in einen Handle die Anzahl der beim Schreiben von Text geschriebenen Zeichen und die Anzahl der beim Schreiben von Bytes geschriebenen Bytes zurückgegeben:

Python 3.x 3.0

```
import sys

char_count = sys.stdout.write('hello world \n')
# Out: hello world 
char_count
# Out: 14

byte_count = sys.stdout.buffer.write(b'hello world \xf0\x9f\x90\x8d\n')
# Out: hello world 
byte_count
# Out: 17
```

long vs. int

In Python 2 wird jede ganze Zahl, die größer als C `ssize_t` ist, in den `long` Datentyp konvertiert, der durch ein `L` Suffix im Literal angegeben wird. Zum Beispiel bei einem 32-Bit-Build von 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
```

In Python 3 wurde jedoch der `long` Datentyp entfernt. egal wie groß die ganze Zahl ist, es wird eine `int` .

Python 3.x 3.0

```
2**1024
# Output:
17976931348623159077293051907890247336179769789423065727343008115773267580550096313270847732240753602112

print(-(2**1024))
# Output: -
17976931348623159077293051907890247336179769789423065727343008115773267580550096313270847732240753602112

type(2**1024)
# Output: <class 'int'>
```

Klasse Boolescher Wert

Python 2.x 2.7

Wenn Sie in Python 2 einen booleschen Klassenwert selbst definieren möchten, müssen Sie die `__nonzero__` Methode in Ihrer Klasse implementieren. Der Wert ist standardmäßig 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

In Python 3 wird `__bool__` anstelle von `__nonzero__`

```
class MyClass:
    def __bool__(self):
        return False

my_instance = MyClass()
print(bool(MyClass))       # True
print(bool(my_instance))   # False
```

Inkompatibilitäten von Python 2 zu Python 3 online lesen:
https://riptutorial.com/de/python/topic/809/inkompatibilitaten-von-python-2-zu-python-3

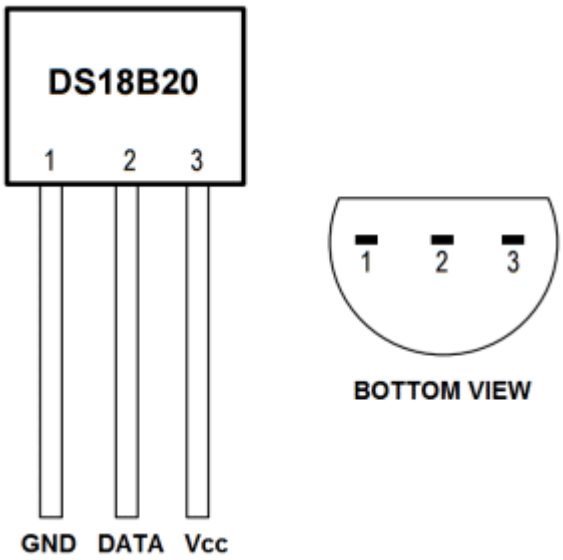
Kapitel 83: IoT-Programmierung mit Python und Himbeer-PI

Examples

Beispiel - Temperatursensor

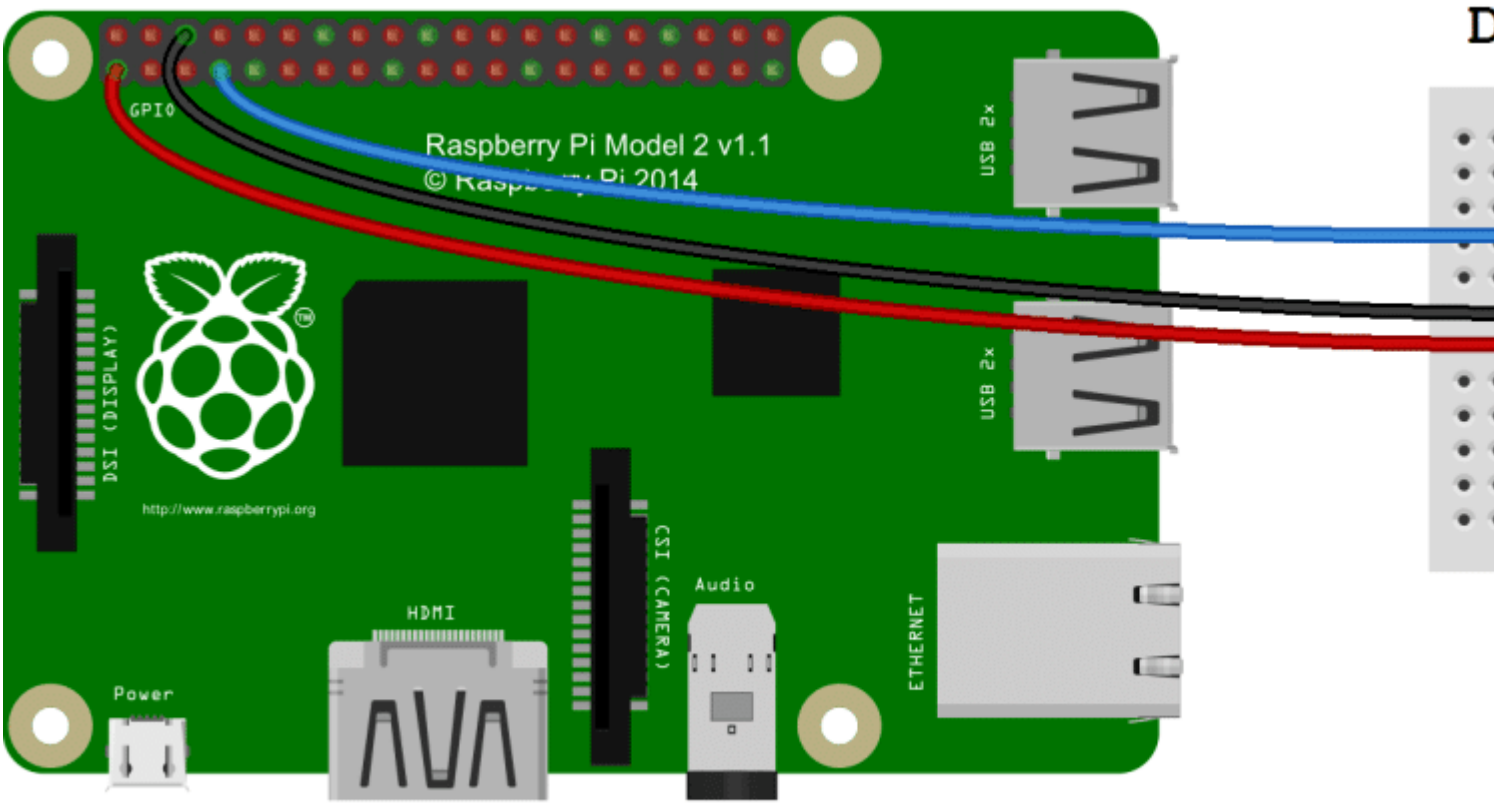
Schnittstelle des DS18B20 mit Himbeer-Pi

Anschluss des DS18B20 mit Himbeer-Pi



Sie können sehen, dass es drei Terminals gibt

1. Vcc
2. Gnd
3. Daten (Ein-Draht-Protokoll)



R1 ist ein Widerstand von 4,7 kΩ, um den Spannungspegel zu erhöhen

1. **Vcc** sollte an einen der 5-V- oder 3,3-V-Pins von Raspberry Pi (PIN: 01, 02, 04, 17) angeschlossen werden.
2. **Gnd** sollte mit einem der Gnd-Pins von Raspberry Pi verbunden werden (PIN: 06, 09, 14, 20, 25).
3. **DATA** sollte verbunden sein (PIN: 07)

Aktivieren der Ein-Draht-Schnittstelle von der RPi-Seite

4. Melden Sie sich mit Putty oder einem anderen Linux / Unix-Terminal bei Raspberry Pi an.
5. Öffnen Sie nach dem Login die Datei /boot/config.txt in Ihrem bevorzugten Browser.

   nano /boot/config.txt

6. `dtoverlay=w1-gpio` nun diese Zeile `dtoverlay=w1-gpio` am Ende der Datei hinzu.
7. Starten Sie nun den Raspberry Pi `sudo reboot` .
8. `sudo modprobe g1-gpio` bei Raspberry Pi an und führen Sie `sudo modprobe g1-gpio`
9. Führen Sie dann `sudo modprobe w1-therm`
10. Gehen Sie nun in das Verzeichnis / sys / bus / w1 / devices `cd /sys/bus/w1/devices`
11. Nun werden Sie ein virtuelles Verzeichnis Ihres Temperatursensors ausgehend von 28 -

******** gefunden.

12. Gehe in dieses Verzeichnis `cd 28-********`

13. Nun gibt es einen Dateinamen **w1-Slave** . Diese Datei enthält die Temperatur und andere Informationen wie CRC. `cat w1-slave` .

Schreiben Sie nun ein Modul in Python, um die Temperatur abzulesen

```
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)
```

Über dem Python-Modul wird die Temperatur gegen Adresse für unendliche Zeit gedruckt. Der Parameter RATE ist definiert, um die Frequenz der Temperaturabfrage vom Sensor zu ändern oder anzupassen.

GPIO-Pin-Diagramm

1. [https://www.element14.com/community/servlet/JiveServlet/previewBody/73950-102-11-339300/pi3_gpio.png?

IoT-Programmierung mit Python und Himbeer-PI online lesen:
https://riptutorial.com/de/python/topic/10735/iot-programmierung-mit-python-und-himbeer-pi

Kapitel 84: Iterables und Iteratoren

Examples

Iterator vs Iterable vs Generator

Ein **Iterable** ist ein Objekt, das einen **Iterator zurückgeben kann** . Jedes Objekt mit einem Status, das eine `__iter__` Methode hat und einen Iterator zurückgibt, ist ein `__iter__` . Es kann sich auch um ein Objekt *ohne* Status handeln, das eine `__getitem__` Methode implementiert. - Die Methode kann Indizes annehmen (beginnend bei Null) und einen `IndexError` wenn die Indizes nicht mehr gültig sind.

Die `str` Klasse von Python ist ein Beispiel für eine `__getitem__` .

Ein **Iterator** ist ein Objekt, das den nächsten Wert in einer Sequenz erzeugt, wenn Sie `next(*object*)` für ein Objekt aufrufen. Darüber hinaus ist jedes Objekt mit einer `__next__` Methode ein Iterator. Ein Iterator erhöht die `StopIteration` nachdem der Iterator erschöpft ist, und *kann* an dieser Stelle *nicht* erneut verwendet werden.

Iterable Klassen:

Iterable-Klassen definieren eine `__iter__` und eine `__next__` Methode. Beispiel für eine iterierbare Klasse:

```
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())
```

Der Versuch, die abstrakte Klasse aus dem `collections` Modul zu instanziieren, um dies besser zu verstehen.

Beispiel:

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__
```

Behandeln Sie die Python 3-Kompatibilität für iterierbare Klassen in Python 2, indem Sie folgende Schritte ausführen:

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
```

Beide sind jetzt Iteratoren und können durchgeschleift werden:

```
ex1 = MyIterableClass()
ex2 = MySequence()

for (item) in (ex1): #code
for (item) in (ex2): #code
```

Generatoren sind einfache Möglichkeiten, Iteratoren zu erstellen. Ein Generator *ist* ein Iterator und ein Iterator ist ein Iterator.

Was kann iterierbar sein

Iterable kann alles sein, für das Elemente **einzeln** empfangen *werden, nur weiterleiten* . Eingebaute Python-Sammlungen sind wiederholbar:

```
[1, 2, 3]     # list, iterate over items
(1, 2, 3)     # tuple
{1, 2, 3}     # set
{1: 2, 3: 4}  # dict, iterate over keys
```

Generatoren geben iterables zurück:

```
def foo():  # foo isn't iterable yet...
    yield 1

res = foo()  # ...but res already is
```

Iteration über ganze iterable

```
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]
```

Überprüfen Sie nur ein Element in iterable

Verwenden Sie das Auspacken, um das erste Element zu extrahieren und sicherzustellen, dass es das einzige ist:

```
a, = iterable

def foo():
    yield 1

a, = foo()  # a = 1

nums = [1, 2, 3]
a, = nums  # ValueError: too many values to unpack
```

Extrahieren Sie die Werte einzeln

Beginnen Sie mit `iter()` , um den **Iterator** für iterable zu verwenden, und verwenden Sie `next()` , um Elemente nacheinander `StopIteration` bis `StopIteration` wird, um das Ende `StopIteration` :

```
s = {1, 2}   # or list or generator or even iterator
i = iter(s)  # get iterator
a = next(i)  # a = 1
b = next(i)  # b = 2
c = next(i)  # raises StopIteration
```

Iterator ist nicht wiedereintrittsfähig!

```
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()
```

Iterables und Iteratoren online lesen: https://riptutorial.com/de/python/topic/2343/iterables-und-iteratoren

Kapitel 85: Itertools-Modul

Syntax

- `import itertools`

Examples

Elemente aus einem iterierbaren Objekt mithilfe einer Funktion gruppieren

Beginnen Sie mit einer Iteration, die gruppiert werden muss

```
lst = [("a", 5, 6), ("b", 2, 4), ("a", 2, 5), ("c", 2, 6)]
```

Generieren Sie den gruppierten Generator nach dem zweiten Element in jedem Tupel:

```
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)]
```

Es werden nur Gruppen aufeinanderfolgender Elemente gruppiert. Möglicherweise müssen Sie nach demselben Schlüssel sortieren, bevor Sie groupby aufrufen. Beispiel: (Letztes Element wird geändert)

```
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)]
```

Die von groupby zurückgegebene Gruppe ist ein Iterator, der vor der nächsten Iteration ungültig ist. Das Folgende funktioniert beispielsweise nicht, wenn Sie die Gruppen nach Schlüssel sortieren möchten. Gruppe 5 ist unten leer. Wenn Gruppe 2 abgerufen wird, wird sie ungültig 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 []
```

Um die Sortierung korrekt durchzuführen, erstellen Sie vor dem Sortieren eine Liste im Iterator

```
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)]
```

Nehmen Sie ein Stück eines Generators

Mit Itertools "islice" können Sie einen Generator schneiden:

```
results = fetch_paged_results()  # returns a generator
limit = 20  # Only want the first 20 results
for data in itertools.islice(results, limit):
    print(data)
```

Normalerweise können Sie einen Generator nicht schneiden:

```
def gen():
    n = 0
    while n < 20:
        n += 1
        yield n

for part in gen()[:3]:
    print(part)
```

Werde geben

```
Traceback (most recent call last):
  File "gen.py", line 6, in <module>
    for part in gen()[:3]:
TypeError: 'generator' object is not subscriptable
```

Dies funktioniert jedoch:

```
import itertools

def gen():
    n = 0
    while n < 20:
        n += 1
        yield n

for part in itertools.islice(gen(), 3):
    print(part)
```

Beachten Sie, dass Sie wie ein normales Slice auch Argumente für `start` , `stop` und `step` :

```
itertools.islice(iterable, 1, 30, 3)
```

itertools.product

Mit dieser Funktion können Sie das kartesische Produkt einer Liste von Iterablen durchlaufen.

Zum Beispiel,

```
for x, y in itertools.product(xrange(10), xrange(10)):
    print x, y
```

ist äquivalent zu

```
for x in xrange(10):
    for y in xrange(10):
        print x, y
```

Wie alle Python-Funktionen, die eine variable Anzahl von Argumenten akzeptieren, können wir mit dem Operator * eine Liste an itertools.product zum Entpacken übergeben.

Somit,

```
its = [xrange(10)] * 2
for x,y in itertools.product(*its):
    print x, y
```

erzeugt die gleichen Ergebnisse wie in den beiden vorherigen Beispielen.

```
>>> 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

Einführung:

Diese einfache Funktion erzeugt eine unendliche Anzahl von Zahlen. Zum Beispiel...

```
for number in itertools.count():
    if number > 20:
        break
    print(number)
```

Beachten Sie, dass wir brechen müssen oder es für immer druckt!

Ausgabe:

```
0
1
2
3
4
5
6
7
8
9
10
```

Argumente:

`count()` benötigt zwei Argumente, `start` und `step` :

```
for number in itertools.count(start=10, step=4):
    print(number)
    if number > 20:
        break
```

Ausgabe:

```
10
14
18
22
```

itertools.takewhile

Mit itertools.takewhile können Sie Elemente aus einer Sequenz übernehmen, bis eine Bedingung zuerst `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)
```

Dies gibt `[0, 2, 4, 12, 18]` .

Beachten Sie, dass die erste Zahl, die das Prädikat verletzt (dh die Funktion, die einen booleschen Wert `is_even` , `13` . Sobald sich `takewhile` trifft, `takewhile` ein Wert auf einen `False` Wert für das angegebene Prädikat trifft, bricht es aus.

Der **Ausgang** von **erzeugten** `takewhile` ist ähnlich das Ausgangssignal von dem unten Code erzeugt.

```
def takewhile(predicate, iterable):
    for x in iterable:
        if predicate(x):
            yield x
        else:
            break
```

Hinweis: Die Verkettung der Ergebnisse, die durch das `takewhile` von `takewhile` und `dropwhile` des `dropwhile` erzeugt werden, erzeugt das ursprüngliche iterierbare Ergebnis.

```
result = list(itertools.takewhile(is_even, lst)) + list(itertools.dropwhile(is_even, lst))
```

itertools

Mit itertools.drop können Sie Elemente aus einer Sequenz übernehmen, nachdem eine Bedingung zuerst `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)
```

Dies gibt `[13, 14, 22, 23, 44]` .

(*Dieses Beispiel ist das gleiche wie das Beispiel für* `takewhile` *jedoch mit* `dropwhile` *.*)

Beachten Sie, dass die erste Zahl, die das Prädikat verletzt (dh die Funktion, die einen booleschen Wert `is_even` , `13` . Alle Elemente davor werden verworfen.

Der **Ausgang** von **erzeugten** `dropwhile` ist ähnlich das Ausgangssignal von dem unten Code erzeugt.

```
def dropwhile(predicate, iterable):
    iterable = iter(iterable)
    for x in iterable:
        if not predicate(x):
            yield x
            break
    for x in iterable:
        yield x
```

Die Verkettung der Ergebnisse, die durch `takewhile` und `dropwhile` erzeugt werden, erzeugt das ursprüngliche iterable.

```
result = list(itertools.takewhile(is_even, lst)) + list(itertools.dropwhile(is_even, lst))
```

Zippen Sie zwei Iteratoren, bis beide erschöpft sind

Ähnlich wie bei der eingebauten Funktion `zip()` wird `itertools.zip_longest` auch nach dem Ende der kürzeren von zwei Iterationen `itertools.zip_longest` .

```
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)
```

Ein optionales `fillvalue` Argument kann wie `fillvalue` übergeben werden (`fillvalue ''`):

```
for i in zip_longest(a, b, fillvalue='Hogwash!'):
    x, y = i  # Note that zip longest returns the values as a tuple
    print(x, y)
```

In Python 2.6 und 2.7 heißt diese Funktion `itertools.izip_longest` .

Kombinationsmethode im Itertools-Modul

`itertools.combinations` einen Generator der *k*- Kombinationsfolge einer Liste.

Mit anderen Worten: Es wird ein Generator von Tupeln aller möglichen k-weisen Kombinationen der Eingabeliste zurückgegeben.

Zum Beispiel:

Wenn Sie eine Liste haben:

```
a = [1,2,3,4,5]
b = list(itertools.combinations(a, 2))
print b
```

Ausgabe:

```
[(1, 2), (1, 3), (1, 4), (1, 5), (2, 3), (2, 4), (2, 5), (3, 4), (3, 5), (4, 5)]
```

Die obige Ausgabe ist ein Generator, der in eine Liste von Tupeln aller möglichen *paarweisen* Kombinationen der Eingabeliste `a`

Sie können auch alle 3 Kombinationen finden:

```
a = [1,2,3,4,5]
b = list(itertools.combinations(a, 3))
```

```
print b
```

Ausgabe:

```
[(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)]
```

Mehrere Iteratoren miteinander verketten

Verwenden Sie `itertools.chain` , um einen einzelnen Generator zu erstellen, der die Werte von mehreren Generatoren nacheinander liefert.

```
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))
```

Ergebnisse in:

```
'1 2 3 4 x y z'
```

Als alternativen Konstruktor können Sie die Klassenmethode `chain.from_iterable` die als einzigen Parameter eine iterierbare Option `chain.from_iterable` iterierbare Werte verwendet. Um das gleiche Ergebnis wie oben zu erhalten:

```
' '.join(chain.from_iterable([a,b])
```

Während `chain` eine beliebige Anzahl von Argumenten `chain.from_iterable` ist `chain.from_iterable` die einzige Möglichkeit, eine *unendliche* Anzahl von `chain.from_iterable` .

itertools.repeat

N mal wiederholen:

```
>>> import itertools
>>> for i in itertools.repeat('over-and-over', 3):
...     print(i)
over-and-over
over-and-over
over-and-over
```

Erhalten Sie eine kumulierte Summe von Zahlen in einem iterierbaren Element

Python 3.x 3.2

`accumulate` ergibt eine kumulative Summe (oder Produkt) von Zahlen

```
>>> 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]
```

Durchlaufen Sie Elemente in einem Iterator

`cycle` ist ein unendlicher Iterator.

```
>>> import itertools as it
>>> it.cycle('ABCD')
A B C D A B C D A B C D ...
```

Achten Sie daher darauf, Grenzen zu verwenden, um eine Endlosschleife zu vermeiden. Beispiel:

```
>>> # 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` gibt einen Generator mit aufeinanderfolgenden Permutationen der Elemente in der iterierbaren Länge zurück.

```
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)]
```

Wenn die Liste `a` doppelte Elemente enthält, haben die resultierenden Permutationen doppelte Elemente. Sie können `set` , um eindeutige Permutationen zu erhalten:

```
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)}
```

Itertools-Modul online lesen: https://riptutorial.com/de/python/topic/1564/itertools-modul

Kapitel 86: JSON-Modul

Bemerkungen

Die vollständige Dokumentation einschließlich versionsspezifischer Funktionen finden Sie in der offiziellen Dokumentation .

Typen

Standardwerte

Das `json` Modul übernimmt standardmäßig das Kodieren und Dekodieren der folgenden Typen:

Deserialisierungsarten:

JSON	Python
Objekt	diktieren
Array	Liste
Schnur	str
Nummer (int)	int
Nummer (real)	schweben
wahr falsch	Wahr falsch
Null	Keiner

Das `json` Modul versteht auch `NaN` , `Infinity` und `-Infinity` als ihre entsprechenden Float-Werte, was außerhalb der JSON-Spezifikation liegt.

Serialisierungsarten:

Python	JSON
diktieren	Objekt
Liste, Tupel	Array
str	Schnur
int, float, (int / float) abgeleitete Enums	Nummer

Python	JSON
Wahr	wahr
Falsch	falsch
Keiner	Null

Um die Kodierung von `NaN` , `Infinity` und `-Infinity` , müssen Sie mit `allow_nan=False` kodieren. Dadurch wird ein `ValueError` wenn Sie versuchen, diese Werte zu kodieren.

Kundenspezifische (De-) Serialisierung

Es gibt verschiedene Hooks, mit denen Sie mit Daten umgehen können, die unterschiedlich dargestellt werden müssen. Die Verwendung von `functools.partial` ermöglicht es Ihnen, die relevanten Parameter aus praktischen `functools.partial` teilweise auf diese Funktionen anzuwenden.

Serialisierung:

Sie können eine Funktion bereitstellen, die Objekte bearbeitet, bevor sie wie folgt serialisiert werden:

```
# my_json module

import json
from functools import partial

def serialise_object(obj):
    # Do something to produce json-serialisable data
    return dict_obj

dump = partial(json.dump, default=serialise_object)
dumps = partial(json.dumps, default=serialise_object)
```

De-Serialisierung:

Es gibt verschiedene Hooks, die von den Json-Funktionen behandelt werden, z. B. object_hook und parse_float. Eine vollständige Liste Ihrer Python-Version finden Sie hier .

```
# 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)
```

Weitere benutzerdefinierte (De) Serialisierung:

Das `json` Modul ermöglicht auch die Erweiterung / Ersetzung des `json.JSONEncoder` und des `json.JSONDecoder` , um verschiedene Typen zu behandeln. Die oben dokumentierten Hooks können als Standardwerte hinzugefügt werden, indem eine gleichnamige Methode erstellt wird. Um diese zu verwenden, übergeben Sie die Klasse einfach als `cls` Parameter an die entsprechende Funktion. Die Verwendung von `functools.partial` ermöglicht es Ihnen, den cls-Parameter zur Vereinfachung auf diese Funktionen anzuwenden, z

```
# 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

JSON aus Python-Diktieren erstellen

```
import json
d = {
    'foo': 'bar',
    'alice': 1,
    'wonderland': [1, 2, 3]
}
json.dumps(d)
```

Das obige Snippet gibt Folgendes zurück:

```
'{"wonderland": [1, 2, 3], "foo": "bar", "alice": 1}'
```

Erstellen von Python-Diktaten aus JSON

```
import json
s = '{"wonderland": [1, 2, 3], "foo": "bar", "alice": 1}'
json.loads(s)
```

Das obige Snippet gibt Folgendes zurück:

```
{u'alice': 1, u'foo': u'bar', u'wonderland': [1, 2, 3]}
```

Daten in einer Datei speichern

Das folgende Snippet codiert die in `d` gespeicherten Daten in JSON und speichert sie in einer Datei (ersetzen Sie `filename` durch den tatsächlichen Dateinamen).

```
import json

d = {
    'foo': 'bar',
    'alice': 1,
    'wonderland': [1, 2, 3]
}

with open(filename, 'w') as f:
    json.dump(d, f)
```

Daten aus einer Datei abrufen

Das folgende Snippet öffnet eine JSON-codierte Datei (Ersetzen Sie `filename` durch den tatsächlichen Dateinamen) und gibt das in der Datei gespeicherte Objekt zurück.

```
import json

with open(filename, 'r') as f:
    d = json.load(f)
```

`load` vs.` load`, `dump` vs.` dumps`

Das `json` Modul enthält Funktionen zum Lesen und Schreiben in und aus Unicode-Zeichenfolgen sowie zum Lesen und Schreiben in und aus Dateien. Diese werden durch nachgestellte `s` im Funktionsnamen unterschieden. In diesen Beispielen verwenden wir ein StringIO-Objekt, aber für jedes dateiähnliche Objekt gelten dieselben Funktionen.

Hier verwenden wir die String-basierten Funktionen:

```
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": []}
```

Und hier verwenden wir die dateibasierten Funktionen:

```
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": []}
```

Wie Sie sehen, besteht der Hauptunterschied darin, dass Sie beim Ablegen von Json-Daten das Dateihandle an die Funktion übergeben müssen, anstatt den Rückgabewert zu erfassen. Erwähnenswert ist auch, dass Sie vor dem Lesen oder Schreiben den Anfang der Datei suchen müssen, um eine Beschädigung der Daten zu vermeiden. Wenn Sie eine Datei öffnen, wird der Cursor auf Position `0`

```
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": []}
```

Wenn Sie beide Arten des Umgangs mit json-Daten haben, können Sie idiomatisch und effizient mit auf json `pyspark` Formaten arbeiten, wie z.

```
# 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')
```

Aufruf von "json.tool" von der Befehlszeile aus, um die JSON-Ausgabe zu drucken

Einige JSON-Datei "foo.json" wie:

```
{"foo": {"bar": {"baz": 1}}}
```

Wir können das Modul direkt von der Befehlszeile aus aufrufen (den Dateinamen als Argument

übergeben), um es hübsch auszudrucken:

```
$ python -m json.tool foo.json
{
    "foo": {
        "bar": {
            "baz": 1
        }
    }
}
```

Das Modul wird auch Eingaben von STDOUT entgegennehmen, also könnten wir (in Bash) auch Folgendes tun:

```
$ cat foo.json | python -m json.tool
```

JSON-Ausgabe formatieren

Nehmen wir an, wir haben folgende Daten:

```
>>> data = {"cats": [{"name": "Tubbs", "color": "white"}, {"name": "Pepper", "color":
"black"}]}
```

Wenn Sie dies einfach als JSON-Dumping ausführen, macht das hier nichts Besonderes:

```
>>> print(json.dumps(data))
{"cats": [{"name": "Tubbs", "color": "white"}, {"name": "Pepper", "color": "black"}]}
```

Einzug festlegen, um eine schönere Ausgabe zu erhalten

Wenn wir hübsch drucken möchten, können wir eine `indent` festlegen:

```
>>> print(json.dumps(data, indent=2))
{
  "cats": [
    {
      "name": "Tubbs",
      "color": "white"
    },
    {
      "name": "Pepper",
      "color": "black"
    }
  ]
}
```

Schlüssel alphabetisch sortieren, um eine konsistente Ausgabe zu erhalten

Standardmäßig ist die Reihenfolge der Schlüssel in der Ausgabe nicht definiert. Wir können sie in alphabetischer Reihenfolge erhalten, um sicherzustellen, dass wir immer die gleiche Ausgabe erhalten:

```
>>> print(json.dumps(data, sort_keys=True))
{"cats": [{"color": "white", "name": "Tubbs"}, {"color": "black", "name": "Pepper"}]}
```

Whitespace entfernen, um kompakte Ausgabe zu erhalten

Wir möchten vielleicht die unnötigen Leerzeichen entfernen, was durch das Setzen von Trennzeichen-Strings von den Standardwerten `', '` und `': '`

```
>>>print(json.dumps(data, separators=(',', ':')))
{"cats":[{"name":"Tubbs","color":"white"},{"name":"Pepper","color":"black"}]}
```

JSON-Codierung von benutzerdefinierten Objekten

Wenn wir nur folgendes versuchen:

```
import json
from datetime import datetime
data = {'datetime': datetime(2016, 9, 26, 4, 44, 0)}
print(json.dumps(data))
```

Wir erhalten eine Fehlermeldung, dass `TypeError: datetime.datetime(2016, 9, 26, 4, 44) is not JSON serializable`.

Um das datetime-Objekt ordnungsgemäß serialisieren zu können, müssen Sie benutzerdefinierten Code für die Konvertierung schreiben:

```
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)
```

und dann diese Encoder-Klasse anstelle von `json.dumps`:

```
encoder = DatetimeJSONEncoder()
print(encoder.encode(data))
# prints {"datetime": "2016-09-26T04:44:00"}
```

JSON-Modul online lesen: https://riptutorial.com/de/python/topic/272/json-modul

Kapitel 87: Kartenfunktion

Syntax

- map (Funktion, iterable [, * additional_iterables])
- future_builtins.map (Funktion, iterable [, * additional_iterables])
- itertools.imap (Funktion, iterable [, * additional_iterables])

Parameter

Parameter	Einzelheiten
Funktion	Funktion für das Mapping (muss so viele Parameter wie iterierbare Werte annehmen) (nur *Position*)
iterable	Die Funktion wird auf jedes Element des Iterierbaren angewendet (nur für *Position*).
* additional_iterables	sehen Sie iterable, aber beliebig viele (*optional* , nur *positionell*)

Bemerkungen

Alles, was mit `map` kann, kann auch mit `comprehensions` :

```
list(map(abs, [-1,-2,-3]))     # [1, 2, 3]
[abs(i) for i in [-1,-2,-3]]   # [1, 2, 3]
```

Sie benötigen allerdings `zip` wenn Sie mehrere iterable haben:

```
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]
```

Listenverständnisse sind effizient und können in vielen Fällen schneller als die `map` Testen Sie daher die Zeiten beider Ansätze, wenn Geschwindigkeit für Sie wichtig ist.

Examples

Grundlegende Verwendung von map, itertools.imap und future_builtins.map

Die Kartenfunktion ist die einfachste unter den für die Funktionsprogrammierung verwendeten Python-Einbauten. `map()` wendet eine bestimmte Funktion auf jedes Element in einer iterierbaren

Funktion an:

```
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>
```

Eine Python 3-kompatible `map` ist im Modul `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>
```

Alternativ kann man in Python 2 `imap` von `itertools` , um einen Generator zu erhalten

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>
```

Das Ergebnis kann explizit in eine `list` , um die Unterschiede zwischen Python 2 und 3 zu beseitigen:

```
list(map(len, names))
# Out: [4, 5, 6]
```

`map()` kann durch ein äquivalentes *Listenverständnis* oder *Generatorausdruck ersetzt werden* :

```
[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>
```

Zuordnen jedes Werts in einem iterierbaren Element

Sie können zum Beispiel den absoluten Wert jedes Elements annehmen:

```
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]
```

Anonyme Funktion unterstützt auch das Mappen einer Liste:

```
map(lambda x:x*2, [1, 2, 3, 4, 5])
# Out: [2, 4, 6, 8, 10]
```

oder Dezimalwerte in Prozent umrechnen:

```
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]
```

oder Dollar in Euro umrechnen (Wechselkurs):

```
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` ist eine bequeme Methode, um Parameter von Funktionen so `functools.partial` , dass sie mit `map` anstatt `lambda` oder benutzerdefinierte Funktionen zu erstellen.

Zuordnungswerte verschiedener iterables

Berechnen Sie zum Beispiel den Durchschnitt jedes `i` ten Elements mehrerer iterierbarer Elemente:

```
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]
```

Es gibt unterschiedliche Anforderungen, wenn je nach Version von Python mehr als ein iterierbares Element an die `map` :

- Die Funktion muss so viele Parameter enthalten, wie iterierbare Werte vorhanden sind:

  ```
  def median_of_three(a, b, c):
      return sorted((a, b, c))[1]

  list(map(median_of_three, measurement1, measurement2))
  ```

  TypeError: median_of_three () fehlt 1 erforderliches Positionsargument: 'c'

```
list(map(median_of_three, measurement1, measurement2, measurement3, measurement3))
```

TypeError: median_of_three () benötigt 3 Positionsargumente, aber 4 wurden angegeben

Python 2.x 2.0.1

- `map` : Die Zuordnung Iterierten solange man iterable noch nicht vollständig verbraucht ist , sondern übernimmt `None` von dem vollständig verbraucht Iterables:

```
import operator

measurement1 = [100, 111, 99, 97]
measurement2 = [102, 117]

# Calculate difference between elements
list(map(operator.sub, measurement1, measurement2))
```

TypeError: nicht unterstützte Operandentypen für -: 'int' und 'NoneType'

- `itertools.imap` und `future_builtins.map` : Das Mapping stoppt, sobald eine iterable stoppt:

```
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

- Das Mapping stoppt, sobald eine Iteration stoppt:

```
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]
```

Transponieren mit Map: Verwenden von "None" als Funktionsargument (nur 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: 'NoneType'-Objekt ist nicht aufrufbar

Es gibt jedoch eine Problemumgehung, um ähnliche Ergebnisse zu erzielen:

```
def conv_to_list(*args):
    return list(args)

list(map(conv_to_list, *image))
# Out: [[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

Serien- und Parallelmapping

map () ist eine integrierte Funktion, dh sie ist überall verfügbar, ohne dass eine 'import'-Anweisung verwendet werden muss. Es ist überall wie print () verfügbar. Wenn Sie sich Beispiel 5 ansehen, werden Sie feststellen, dass ich eine Importanweisung verwenden musste, bevor ich Pretty Print (import pprint) verwenden konnte. Daher ist pprint keine eingebaute Funktion

Serienmapping

In diesem Fall wird jedes Argument des Iterers in aufsteigender Reihenfolge als Argument an die Abbildungsfunktion übergeben. Dies tritt auf, wenn wir nur eine einzige Abbildung haben, für die eine Abbildung möglich ist, und die Mapping-Funktion ein einziges Argument erfordert.

Beispiel 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
```

führt in

```
['fly is an insect', 'ant is an insect', 'beetle is an insect', 'cankerworm is an insect']
```

Beispiel 2

```
print(list(map(len, insects))) # the len function is executed each item in the insect list
```

führt in

```
[3, 3, 6, 10]
```

Parallele Abbildung

In diesem Fall wird jedes Argument der Mapping-Funktion parallel von allen iterierbaren Elementen (eines von jedem iterierbaren Element) abgerufen. Die Anzahl der gelieferten Iterables muss daher mit der Anzahl der von der Funktion benötigten Argumente übereinstimmen.

```
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)
```

Beispiel 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)))
```

führt in

```
TypeError: len() takes exactly one argument (4 given)
```

Beispiel 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)))
```

führt in

```
TypeError: animals() missing 3 required positional arguments: 'x', 'y', and 'z'
```

Beispiel 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)))
```

führt in

```
 ['Fly, lion, african buffalo, and chicken ARE ALL ANIMALS',
 'Ant, tiger, moose, and dove ARE ALL ANIMALS',
 'Beetle, leopard, okapi, and mouse ARE ALL ANIMALS',
 'Cankerworm, arctic fox, parakeet, and pig ARE ALL ANIMALS']
```

Kartenfunktion online lesen: https://riptutorial.com/de/python/topic/333/kartenfunktion

Kapitel 88: Kissen

Examples

Bilddatei lesen

```
from PIL import Image

im = Image.open("Image.bmp")
```

Konvertieren Sie Dateien in JPEG

```
from __future__ import print_function
import os, sys
from PIL import Image

for infile in sys.argv[1:]:
    f, e = os.path.splitext(infile)
    outfile = f + ".jpg"
    if infile != outfile:
        try:
            Image.open(infile).save(outfile)
        except IOError:
            print("cannot convert", infile)
```

Kissen online lesen: https://riptutorial.com/de/python/topic/6841/kissen

Kapitel 89: kivy - Plattformübergreifendes Python-Framework für die NUI-Entwicklung

Einführung

NUI: Eine natürliche Benutzeroberfläche (NUI) ist ein System für die Mensch-Computer-Interaktion, das der Benutzer durch intuitive Aktionen im Zusammenhang mit dem natürlichen, alltäglichen menschlichen Verhalten ausführt.

Kivy ist eine Python-Bibliothek für die Entwicklung von Multi-Touch-fähigen Medienanwendungen, die auf verschiedenen Geräten installiert werden können. Multi-Touch bezieht sich auf die Fähigkeit einer berührungsempfindlichen Oberfläche (normalerweise ein Touchscreen oder ein Trackpad), um Eingaben von zwei oder mehr Kontaktpunkten gleichzeitig zu erfassen oder zu erfassen.

Examples

Erste App

So erstellen Sie eine Kivy-Anwendung

1. Unterklasse die **App-** Klasse
2. Implementieren Sie die **Build-** Methode, die das Widget zurückgibt.
3. Instanziieren Sie die Klasse und rufen Sie den **Lauf auf** .

```
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()
```

Erläuterung

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

Die obige Anweisung importiert die übergeordnete Klassen- **App** . Dies wird in Ihrem Installationsverzeichnis vorhanden sein: Installationsverzeichnis / kivy / app.py

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

Die obige Anweisung importiert das ux-Element **Label** . Alle ux-Elemente befinden sich in Ihrem Installationsverzeichnis Ihr_Installationsverzeichnis / kivy / uix /.

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

Die obige Anweisung dient zum Erstellen Ihrer App und der Klassenname ist Ihr App-Name. Diese Klasse wird von der übergeordneten App-Klasse übernommen.

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

Die obige Anweisung überschreibt die Build-Methode der App-Klasse. Das gibt das Widget zurück, das angezeigt werden muss, wenn Sie die App starten.

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

Die obige Anweisung ist der Hauptteil der Build-Methode. Es gibt das Label mit seinem Text **Hallo Welt zurück** .

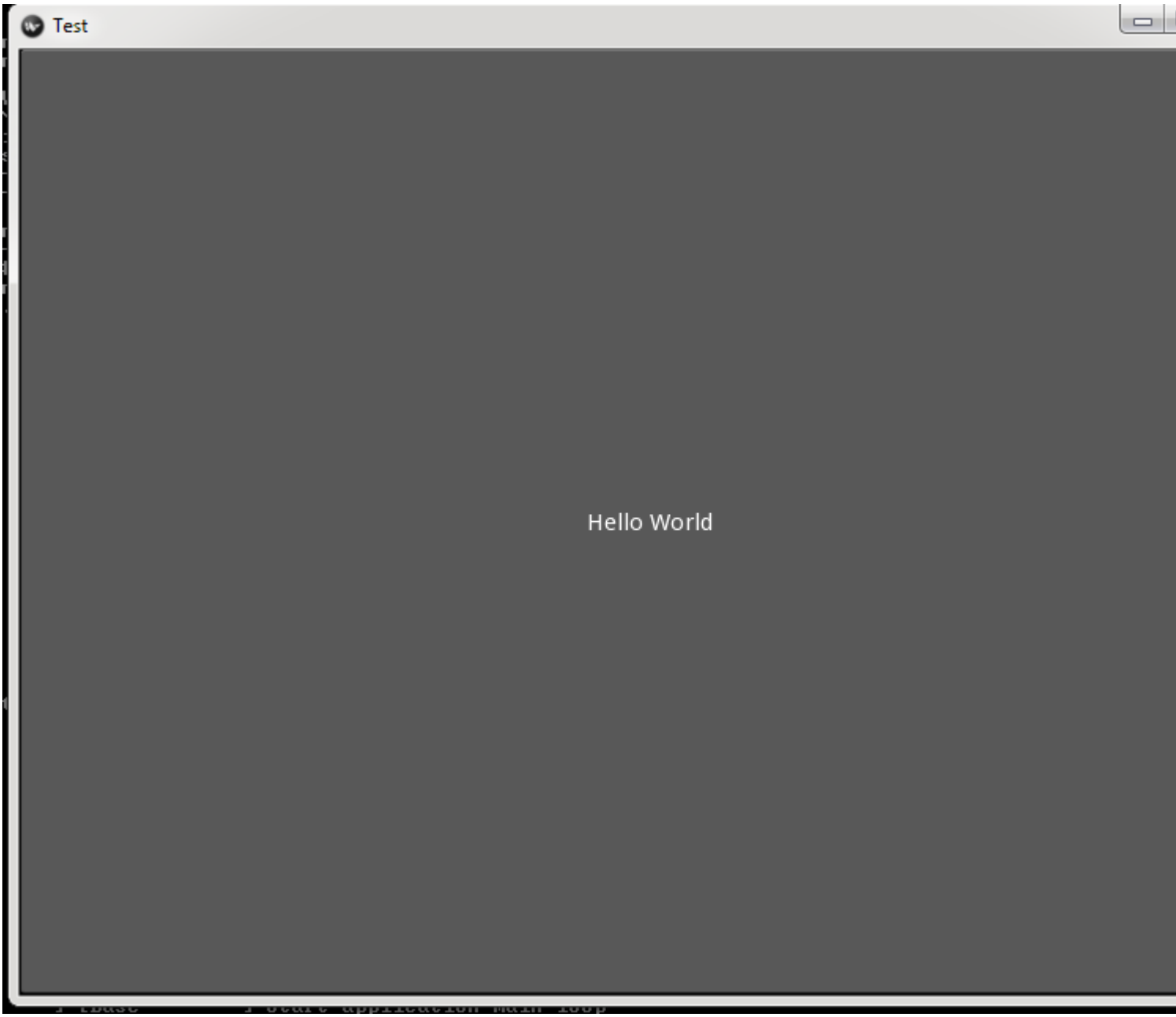
```
if __name__ == '__main__':
```

Die obige Anweisung ist der Einstiegspunkt, von dem aus der Python-Interpreter mit der Ausführung Ihrer App beginnt.

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

Die obige Anweisung Initialisiert Ihre Test-Klasse durch Erstellen ihrer Instanz. Rufen Sie die App-Klassenfunktion run () auf.

Ihre App sieht wie in der Abbildung unten aus.



kivy - Plattformübergreifendes Python-Framework für die NUI-Entwicklung online lesen: https://riptutorial.com/de/python/topic/10743/kivy---plattformubergreifendes-python-framework-fur-die-nui-entwicklung

Kapitel 90: Klassen

Einführung

Python bietet sich nicht nur als beliebte Skriptsprache an, sondern unterstützt auch das objektorientierte Programmierparadigma. Klassen beschreiben Daten und stellen Methoden zum Manipulieren dieser Daten bereit, die alle in einem einzigen Objekt enthalten sind. Darüber hinaus ermöglichen Klassen eine Abstraktion, indem konkrete Implementierungsdetails von abstrakten Repräsentationen von Daten getrennt werden.

Code, der Klassen verwendet, ist im Allgemeinen einfacher zu lesen, zu verstehen und zu verwalten.

Examples

Grundvererbung

Die Vererbung in Python basiert auf ähnlichen Ideen, die in anderen objektorientierten Sprachen wie Java, C ++ usw. verwendet werden. Eine neue Klasse kann wie folgt von einer vorhandenen Klasse abgeleitet werden.

```
class BaseClass(object):
    pass

class DerivedClass(BaseClass):
    pass
```

Die `BaseClass` ist die bereits bestehende *(Eltern)* Klasse und die `DerivedClass` ist die neue *(Kind)* Klasse , die (oder *Unterklassen)* Attribute erbt `BaseClass` . **Hinweis** : Ab Python 2.2 erben alle Klassen implizit von der `object` , der Basisklasse für alle integrierten Typen.

Im folgenden Beispiel definieren wir eine übergeordnete `Rectangle` , die implizit vom `object` erbt:

```
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)
```

Die `Rectangle` kann als Basisklasse zum Definieren einer `Square` , da ein Quadrat ein Sonderfall eines Rechtecks ist.

```
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
```

Die `Square` Klasse erbt automatisch alle Attribute der `Rectangle` Klasse sowie der Objektklasse. `super()` wird verwendet, um die `__init__()` Methode der `Rectangle` Klasse aufzurufen, wobei im Wesentlichen alle überschriebenen Methoden der Basisklasse `__init__()` . **Hinweis** : Für `super()` in Python 3 keine Argumente erforderlich.

Abgeleitete Klassenobjekte können auf die Attribute ihrer Basisklassen zugreifen und diese ändern:

```
r.area()
# Output: 12
r.perimeter()
# Output: 14

s.area()
# Output: 4
s.perimeter()
# Output: 8
```

Integrierte Funktionen, die mit Vererbung arbeiten

`issubclass(DerivedClass, BaseClass)` : `issubclass(DerivedClass, BaseClass) True` wenn `DerivedClass` eine Unterklasse der `BaseClass`

`isinstance(s, Class)` : `isinstance(s, Class) True` wenn s eine Instanz von `Class` oder eine der abgeleiteten Klassen von `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
```

Klassen- und Instanzvariablen

Instanzvariablen sind für jede Instanz eindeutig, während Klassenvariablen von allen Instanzen gemeinsam genutzt werden.

```
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
```

Auf Klassenvariablen kann auf Instanzen dieser Klasse zugegriffen werden. Wenn Sie jedoch das Klassenattribut zuweisen, wird eine Instanzvariable erstellt, die die Klassenvariable überschattet

```
c2.x = 4
c2.x
# 4
C.x
# 2
```

Beachten Sie, dass das *Ändern von* Klassenvariablen aus Instanzen zu unerwarteten Konsequenzen führen kann.

```
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]
```

Gebundene, ungebundene und statische Methoden

Die Idee von gebundenen und ungebundenen Methoden wurde in Python 3 entfernt . Wenn Sie in Python 3 eine Methode innerhalb einer Klasse deklarieren, verwenden Sie ein `def` Schlüsselwort und erstellen so ein Funktionsobjekt. Dies ist eine reguläre Funktion, und die umgebende Klasse fungiert als Namespace. Im folgenden Beispiel deklarieren wir die Methode `f` innerhalb der Klasse `A` und werden zu einer Funktion `Af` :

Python 3.x 3.0

```
class A(object):
    def f(self, x):
        return 2 * x
A.f
# <function A.f at ...>  (in Python 3.x)
```

In Python 2 war das Verhalten anders: Funktionsobjekte innerhalb der Klasse wurden implizit durch Objekte des Typs `instancemethod` , die als *ungebundene Methoden bezeichnet wurden,* da sie nicht an eine bestimmte Klasseninstanz gebunden waren. Es war möglich, auf die zugrunde liegende Funktion mit der `.__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 ...>
```

Letztere Verhaltensweisen werden durch Inspektion bestätigt - Methoden werden in Python 3 als Funktionen erkannt, während die Unterscheidung in Python 2 beibehalten wird.

Python 3.x 3.0

```
import inspect

inspect.isfunction(A.f)
# True
inspect.ismethod(A.f)
# False
```

Python 2.x 2.3

```
import inspect

inspect.isfunction(A.f)
# False
inspect.ismethod(A.f)
# True
```

In beiden Versionen von Python kann die Funktion / Methode `Af` direkt aufgerufen werden,

vorausgesetzt, Sie übergeben eine Instanz der Klasse `A` als erstes Argument.

```
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
```

Angenommen, `a` ist eine Instanz der Klasse `A` , was ist dann `af` ? Nun, intuitiv sollte dies die gleiche Methode `f` der Klasse `A` , es sollte nur irgendwie "wissen", dass sie auf das Objekt `a` angewendet wurde - in Python wird diese Methode als *an* `a` *gebunden* .

Die wichtigsten Details sind folgende: Das Schreiben von `af` ruft die magische `__getattribute__` Methode von `a` , die zuerst prüft, ob `a` ein Attribut mit dem Namen `f` (nicht), und dann die Klasse `A` überprüft, ob sie eine Methode mit einem solchen Namen enthält (tut dies) und erstellt ein neues Objekt `m` vom Typ `method` das den Verweis auf das ursprüngliche `Af` in `m.__func__` und einen Verweis auf das Objekt `a` in `m.__self__` . Wenn dieses Objekt als Funktion aufgerufen wird, führt es einfach Folgendes aus: `m(...) => m.__func__(m.__self__, ...)` . Daher wird dieses Objekt als **gebundene Methode bezeichnet,** da es beim Aufruf weiß, das Objekt anzugeben, an das es gebunden war, als erstes Argument. (Diese Dinge funktionieren in Python 2 und 3 auf dieselbe Weise).

```
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
```

Schließlich verfügt Python über **Klassenmethoden** und **statische Methoden** - spezielle Arten von Methoden. Klassenmethoden funktionieren auf die gleiche Weise wie reguläre Methoden, außer dass sie beim Aufruf an ein Objekt an die *Klasse* des Objekts und nicht an das Objekt gebunden werden. Also ist `m.__self__ = type(a)` . Wenn Sie eine solche gebundene Methode aufrufen, übergibt sie die Klasse `a` als erstes Argument. Statische Methoden sind noch einfacher: Sie binden gar nichts und geben die zugrunde liegende Funktion einfach ohne Transformationen zurück.

```
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
```

Beachten Sie, dass Klassenmethoden auch dann an die Klasse gebunden sind, wenn auf die Instanz zugegriffen wird:

```
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
```

Es ist erwähnenswert, dass auf der untersten Ebene Funktionen, Methoden, statische Methoden usw. eigentlich Deskriptoren sind , die die speziellen Methoden `__get__` , `__set__` und optional `__del__` . Weitere Informationen zu Klassenmethoden und statischen Methoden finden Sie unter:

- Was ist der Unterschied zwischen @staticmethod und @classmethod in Python?
- Bedeutung von @classmethod und @staticmethod für Anfänger?

Klassen im neuen Stil und im alten Stil

Python 2.x 2.2.0

In Python 2.2 wurden *Klassen* mit neuem *Stil* eingeführt, um *Klassen* und *Typen* zu vereinheitlichen. Sie erben vom `object` der obersten Ebene. *Eine Klasse im neuen Stil ist ein benutzerdefinierter Typ* und ist den integrierten Typen sehr ähnlich.

```
# 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
```

Klassen im *alten Stil* erben **nicht** vom `object` . Im alten Stil Instanzen sind immer mit einem eingebauten in implementierten `instance` Art.

```
# old-style class
class Old:
    pass

# old-style instance
old = Old()

old.__class__
# <class __main__.Old at ...>
type(old)
# <type 'instance'>
issubclass(Old, object)
# False
```

Python 3.x 3.0.0

In Python 3 wurden Klassen alten Stils entfernt.

Klassen im neuen Stil in Python 3 erben implizit vom `object` , so dass keine Angabe von `MyClass(object)` mehr erforderlich ist.

```
class MyClass:
    pass

my_inst = MyClass()

type(my_inst)
# <class '__main__.MyClass'>
my_inst.__class__
# <class '__main__.MyClass'>
issubclass(MyClass, object)
# True
```

Standardwerte für Instanzvariablen

Wenn die Variable einen Wert eines unveränderlichen Typs enthält (z. B. eine Zeichenfolge), können Sie einen solchen Standardwert zuweisen

```
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
```

Beim Initialisieren von veränderlichen Objekten wie Listen im Konstruktor muss vorsichtig vorgegangen werden. Betrachten Sie das folgende Beispiel:

```
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
```

Dieses Verhalten wird durch die Tatsache verursacht, dass in Python Standardparameter bei der Funktionsausführung und nicht bei der Funktionsdeklaration gebunden sind. Um eine Standardinstanzvariable zu erhalten, die nicht von Instanzen gemeinsam genutzt wird, sollte ein Konstrukt wie das folgende verwendet werden:

```
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
```

Siehe auch Mutable Default Arguments und "Least Astonishment" und das Mutable Default Argument .

Mehrfachvererbung

Python verwendet den C3-Linearisierungsalgorithmus , um die Reihenfolge zu bestimmen, in der Klassenattribute aufgelöst werden, einschließlich Methoden. Dies wird als Method Resolution Order (MRO) bezeichnet.

Hier ist ein einfaches Beispiel:

```
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'
```

Wenn wir nun FooBar instanziieren, wenn wir das foo-Attribut nachschlagen, sehen wir, dass das Attribut von Foo zuerst gefunden wird

```
fb = FooBar()
```

und

```
>>> fb.foo
'attr foo of Foo'
```

Hier ist der MRO von FooBar:

```
>>> FooBar.mro()
[<class '__main__.FooBar'>, <class '__main__.Foo'>, <class '__main__.Bar'>, <type 'object'>]
```

Es kann einfach gesagt werden, dass der Python-MRO-Algorithmus ist

1. Tiefe zuerst (zB `FooBar` dann `Foo`), sofern nicht
2. Ein gemeinsam genutztes übergeordnetes `object` (`object`) wird von einem untergeordneten `Bar` (`Bar`) und blockiert
3. Keine zirkulären Beziehungen erlaubt.

Das heißt, Bar kann nicht von FooBar erben, während FooBar von Bar erbt.

Ein umfassendes Beispiel in Python finden Sie im Wikipedia-Eintrag .

Ein weiteres mächtiges Feature in der Vererbung ist `super` . super kann übergeordnete Klassenfunktionen abrufen.

```
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()
```

Mehrfachvererbung mit der init-Methode der Klasse. Wenn jede Klasse eine eigene init-Methode hat, versuchen wir die Mehrfachvererbung. Dann wird nur die init-Methode der Klasse aufgerufen, die zuerst erbt.

für folgende Beispiel nur **init** - Methode Foo Klasse immer genannt **Bar** Klasse init nicht immer genannt

```
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()
```

Ausgabe:

```
foobar init
foo init
```

Das bedeutet jedoch nicht, dass die **Bar-** Klasse nicht erbt. Die Instanz der finalen **FooBar-** Klasse ist auch eine Instanz der **Bar-** Klasse und der **Foo-** Klasse.

```
print isinstance(a,FooBar)
print isinstance(a,Foo)
print isinstance(a,Bar)
```

Ausgabe:

```
True
True
True
```

Deskriptoren und gepunktete Lookups

Deskriptoren sind Objekte, die (in der Regel) Attribute von Klassen sind und über besondere Methoden von `__get__` , `__set__` oder `__delete__` .

Datenbezeichner jeder haben `__set__` oder `__delete__`

Diese können die gepunktete Suche in einer Instanz steuern und werden zum Implementieren von Funktionen, `staticmethod` , `classmethod` und `property` . Eine gepunktete Lookup (zB Instanz `foo` der Klasse `Foo` aufzublicken Attribut `bar` - dh `foo.bar`) verwendet den folgenden Algorithmus:

1. `bar` wird in der Klasse `Foo` , `Foo` . Wenn es dort ist und ein **Datendeskriptor ist** , wird der Datendeskriptor verwendet. Auf diese Weise kann eine `property` Zugriff auf Daten in einer Instanz steuern, und Instanzen können dies nicht überschreiben. Wenn kein **Data Descriptor vorhanden** ist, dann

2. `bar` wird in der Instanz `__dict__` . Deshalb können wir Methoden, die von einer Instanz aufgerufen werden, mit einer gepunkteten Suche überschreiben oder blockieren. Wenn in

der Instanz ein `bar` vorhanden ist, wird dieser verwendet. Wenn nicht, dann wir

3. In der Klasse `Foo` nach `bar` suchen. Wenn es sich um einen **Deskriptor handelt** , wird das Deskriptorprotokoll verwendet. So werden Funktionen (in diesem Zusammenhang ungebundene Methoden), `classmethod` und `staticmethod` implementiert. Sonst gibt es einfach das Objekt dort zurück oder es gibt einen `AttributeError`

Klassenmethoden: alternative Initialisierer

Klassenmethoden bieten alternative Möglichkeiten zum Erstellen von Klasseninstanzen. Schauen wir uns ein Beispiel an.

Nehmen wir an, wir haben eine relativ einfache `Person` Klasse:

```
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 + ".")
```

Es kann praktisch sein, Instanzen dieser Klasse zu erstellen, die einen vollständigen Namen anstelle des Vor- und Nachnamens angeben. Eine Möglichkeit, dies zu tun, besteht darin, `last_name` ein optionaler Parameter zu sein. `last_name` dieser Parameter nicht angegeben wird, haben wir den vollständigen Namen in folgender `last_name` :

```
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 + ".")
```

Es gibt jedoch zwei Hauptprobleme mit diesem Codebit:

1. Die Parameter `first_name` und `last_name` sind jetzt irreführend, da Sie einen vollständigen Namen für `first_name` . Wenn es mehr Fälle und / oder mehr Parameter gibt, die diese Art von Flexibilität aufweisen, kann die Verzweigung von if / elif / else schnell ärgerlich werden.

2. Nicht ganz so wichtig, aber dennoch hervorzuheben: Was ist, wenn `last_name None` , aber `first_name` nicht durch Leerzeichen in zwei oder mehr Dinge aufgeteilt wird? Wir haben noch

eine weitere Ebene der Eingabevalidierung und / oder Ausnahmebehandlung ...

Klassenmethoden eingeben `from_full_name` eines einzelnen Initialisierers erstellen wir einen separaten Initialisierer mit dem Namen `from_full_name` und dekorieren ihn mit dem (integrierten) `classmethod` Dekorator.

```
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 + ".")
```

Beachten Sie `cls` statt `self` als erstes Argument für `from_full_name` . Klassenmethoden sind auf die Gesamtklasse angewendet wird , *nicht* eine Instanz einer bestimmten Klasse (was `self` bezeichnet in der Regel). Also, wenn `cls` unsere ist `Person` - Klasse, dann ist der zurückgegebene Wert von der `from_full_name` Klassenmethode ist `Person(first_name, last_name, age)` , die verwendete `Person` ,s `__init__` eine Instanz der erstellen `Person` Klasse. Wenn wir insbesondere eine Unterklasse `Employee` of `Person` `from_full_name` , funktioniert `from_full_name` auch in der `Employee` Klasse.

Um zu zeigen, dass dies wie erwartet funktioniert, erstellen wir Instanzen von `Person` auf mehrere `__init__` , ohne die Verzweigung in `__init__` :

```
In [2]: bob = Person("Bob", "Bobberson", 42)

In [3]: alice = Person.from_full_name("Alice Henderson", 31)

In [4]: bob.greet()
Hello, my name is Bob Bobberson.

In [5]: alice.greet()
Hello, my name is Alice Henderson.
```

Weitere Referenzen:

- Python @classmethod und @staticmethod für Anfänger?
- https://docs.python.org/2/library/functions.html#classmethod
- https://docs.python.org/3.5/library/functions.html#classmethod

Klassenaufbau

Die Klassenzusammensetzung erlaubt explizite Beziehungen zwischen Objekten. In diesem Beispiel leben Menschen in Städten, die zu Ländern gehören. Die Zusammensetzung ermöglicht es den Menschen, auf die Anzahl aller in ihrem Land lebenden Menschen zuzugreifen:

```
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
```

Affen-Patching

In diesem Fall bedeutet "Affen-Patching", dass einer Klasse eine neue Variable oder Methode hinzugefügt wird, nachdem diese definiert wurde. Angenommen, wir haben Klasse `A` als definiert

```
class A(object):
```

```
    def __init__(self, num):
        self.num = num

    def __add__(self, other):
        return A(self.num + other.num)
```

Aber jetzt möchten wir später im Code eine weitere Funktion hinzufügen. Angenommen, diese Funktion ist wie folgt.

```
def get_num(self):
    return self.num
```

Aber wie fügen wir das als Methode in `A` ? Das ist einfach, wir legen diese Funktion im Wesentlichen in `A` mit einer Zuweisungsanweisung ab.

```
A.get_num = get_num
```

Warum funktioniert das? Denn Funktionen sind Objekte wie jedes andere Objekt und Methoden sind Funktionen, die zur Klasse gehören.

Die Funktion `get_num` soll für alle vorhandenen (bereits erstellten) sowie für die neuen Instanzen von `A` verfügbar sein

Diese Zusätze sind automatisch für alle Instanzen dieser Klasse (oder ihrer Unterklassen) verfügbar. Zum Beispiel:

```
foo = A(42)

A.get_num = get_num

bar = A(6);

foo.get_num() # 42

bar.get_num() # 6
```

Beachten Sie, dass diese Technik im Gegensatz zu einigen anderen Sprachen nicht für bestimmte integrierte Typen geeignet ist und nicht als guter Stil betrachtet wird.

Alle Klassenmitglieder auflisten

Die Funktion `dir()` kann verwendet werden, um eine Liste der Mitglieder einer Klasse abzurufen:

```
dir(Class)
```

Zum Beispiel:

```
>>> 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']
```

Es ist üblich, nur nach "nicht-magischen" Mitgliedern zu suchen. Dies kann mit einem einfachen Verständnis durchgeführt werden, in dem Mitglieder aufgelistet werden, deren Namen nicht mit __ :

```
>>> [m for m in dir(list) if not m.startswith('__')]
['append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse',
'sort']
```

Vorsichtsmaßnahmen:

Klassen können eine `__dir__()` -Methode definieren. Wenn diese Methode vorhanden ist, wird beim Aufruf von `dir()` `__dir__()` aufgerufen. `__dir__()` versucht Python, eine Liste der Mitglieder der Klasse zu erstellen. Dies bedeutet, dass die dir-Funktion unerwartete Ergebnisse haben kann. Zwei wichtige Zitate aus der offiziellen Python-Dokumentation :

> Wenn das Objekt **dir** () nicht bereitstellt, versucht die Funktion, Informationen aus dem **Dict-** Attribut des Objekts (falls definiert) und aus seinem **Typobjekt** zu sammeln. Die Ergebnisliste ist nicht notwendigerweise vollständig und kann ungenau sein, wenn das Objekt über eine benutzerdefinierte **getattr** () verfügt.
>
> **Hinweis:** Da dir () hauptsächlich zur Vereinfachung der Verwendung an einer interaktiven Eingabeaufforderung bereitgestellt wird, versucht es, einen interessanten Satz von Namen anzugeben, anstatt einen rigoros oder konsistent definierten Satz von Namen anzugeben, und sein detailliertes Verhalten kann sich ändern Veröffentlichungen. Beispielsweise sind Metaklassenattribute nicht in der Ergebnisliste enthalten, wenn das Argument eine Klasse ist.

Einführung in den Unterricht

Eine Klasse fungiert als Vorlage, die die grundlegenden Merkmale eines bestimmten Objekts definiert. Hier ist ein Beispiel:

```
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))
```

Es gibt ein paar Dinge zu beachten, wenn Sie das obige Beispiel betrachten.

1. Die Klasse besteht aus *Attributen* (Daten) und *Methoden* (Funktionen).
2. Attribute und Methoden werden einfach als normale Variablen und Funktionen definiert.
3. Wie in dem entsprechenden Dokumentstring angegeben, wird die Methode `__init__()` als *Initialisierer bezeichnet* . Sie entspricht dem Konstruktor in anderen objektorientierten Sprachen und ist die Methode, die zuerst ausgeführt wird, wenn Sie ein neues Objekt oder eine neue Instanz der Klasse erstellen.
4. Attribute, die für die gesamte Klasse gelten, werden zuerst definiert und als *Klassenattribute bezeichnet* .
5. Attribute, die für eine bestimmte Instanz einer Klasse (ein Objekt) gelten, werden als *Instanzattribute bezeichnet* . Sie sind im Allgemeinen in `__init__()` . Dies ist nicht erforderlich, wird jedoch empfohlen (da außerhalb von `__init__()` definierte Attribute das Risiko `__init__()` dem Definieren darauf zugegriffen zu werden).
6. Jede in der Klassendefinition enthaltene Methode übergibt das betreffende Objekt als ersten Parameter. Das Wort `self` wird für diesen Parameter verwendet (die Verwendung von `self` ist eigentlich eine Konvention, da das Wort `self` in Python keine inhärente Bedeutung hat, aber dies ist eine der am meisten respektierten Konventionen von Python, und Sie sollten es immer beachten).
7. Diejenigen, die zur objektorientierten Programmierung in anderen Sprachen verwendet werden, können durch einige Dinge überrascht sein. Eines ist, dass Python kein reales Konzept `private` Elemente hat. Daher ahmt alles standardmäßig das Verhalten des `public` C ++ / Java-Schlüsselworts nach. Weitere Informationen finden Sie im Beispiel "Private Class Members" auf dieser Seite.
8. Einige Methoden der Klasse haben folgende Form: `__functionname__(self, other_stuff)` . Alle diese Methoden werden "magische Methoden" genannt und sind ein wichtiger Bestandteil von Klassen in Python. Beispielsweise wird die Überladung von Operatoren in Python mit magischen Methoden implementiert. Weitere Informationen finden Sie in der entsprechenden Dokumentation .

Lassen Sie uns nun einige Beispiele unserer `Person` Klasse machen!

```
>>> # Instances
>>> kelly = Person("Kelly")
>>> joseph = Person("Joseph")
>>> john_doe = Person("John Doe")
```

Wir haben derzeit drei `Person` , `kelly` , `joseph` und `john_doe` .

Mit dem Punktoperator können wir von jeder Instanz aus auf die Attribute der Klasse zugreifen . Beachten Sie noch einmal den Unterschied zwischen Klassen- und Instanzattributen:

```
>>> # Attributes
```

```
>>> kelly.species
'Homo Sapiens'
>>> john_doe.species
'Homo Sapiens'
>>> joseph.species
'Homo Sapiens'
>>> kelly.name
'Kelly'
>>> joseph.name
'Joseph'
```

Wir können die Methoden der Klasse mit demselben Punktoperator ausführen `.`:

```
>>> # Methods
>>> john_doe.__str__()
'John Doe'
>>>  print(john_doe)
'John Doe'
>>>  john_doe.rename("John")
'Now my name is John'
```

Eigenschaften

Python-Klassen unterstützen **Eigenschaften** , die wie reguläre Objektvariablen aussehen, jedoch die Möglichkeit haben, benutzerdefiniertes Verhalten und Dokumentation anzufügen.

```
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
```

Die Objekte der Klasse `MyClass` haben *anscheinend* eine Eigenschaft `.string` . Das Verhalten wird jedoch jetzt streng kontrolliert:

```
mc = MyClass()
mc.string = "String!"
print(mc.string)
del mc.string
```

Neben der nützlichen Syntax wie oben ermöglicht die Eigenschaftssyntax die Validierung oder das

Hinzufügen anderer Erweiterungen zu diesen Attributen. Dies kann besonders bei öffentlichen APIs nützlich sein, bei denen dem Benutzer eine Hilfestellung gegeben werden sollte.

Eine weitere häufige Verwendung von Eigenschaften besteht darin, dass die Klasse "virtuelle Attribute" darstellen kann. Attribute, die nicht gespeichert werden, sondern nur auf Anforderung berechnet werden.

```
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
```

Singleton-Klasse

Ein Singleton ist ein Muster, das die Instantiierung einer Klasse auf eine Instanz / ein Objekt beschränkt. Weitere Informationen zu Python-Singleton-Entwurfsmustern finden Sie hier .

```
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
```

Eine andere Methode ist, Ihre Klasse zu schmücken. Anhand des Beispiels aus dieser Antwort erstellen Sie eine Singleton-Klasse:

```
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)
```

Zur Verwendung können Sie die `Instance` verwenden

```
@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
```

Klassen online lesen: https://riptutorial.com/de/python/topic/419/klassen

Kapitel 91: Kommentare und Dokumentation

Syntax

- # Dies ist ein einzeiliger Kommentar
- print ("") # Dies ist ein Inline-Kommentar
- "" "
  Das ist
  ein mehrzeiliger Kommentar
  "" "

Bemerkungen

Entwickler sollten die PEP257 - Docstring Conventions- Richtlinien befolgen . In einigen Fällen enthalten Style-Guides (z. B. Google Style Guide) oder Dokumentation, die Drittanbieter (z. B. Sphinx) darstellen, zusätzliche Konventionen für Dokumentstrings.

Examples

Einzeilige, inline und mehrzeilige Kommentare

Kommentare werden zum Erklären von Code verwendet, wenn der Basiscode selbst nicht klar ist.

Python ignoriert Kommentare und führt daher keinen Code aus oder führt zu Syntaxfehlern für einfache englische Sätze.

Einzeilige Kommentare beginnen mit dem Hash-Zeichen (`#`) und enden am Zeilenende.

- Einzeiliger Kommentar:

```
# This is a single line comment in Python
```

- Inline-Kommentar:

```
print("Hello World")  # This line prints "Hello World"
```

- Kommentare, die sich über mehrere Zeilen erstrecken, haben an beiden Enden `"""` oder `'''` . Dies ist mit einer mehrzeiligen Zeichenfolge identisch, sie können jedoch als Kommentar verwendet werden:

```
"""
This type of comment spans multiple lines.
These are mostly used for documentation of functions, classes and modules.
"""
```

Programmgesteuerter Zugriff auf Docstrings

Docstrings werden - im Gegensatz zu regulären Kommentaren - als Attribut der von ihnen dokumentierten Funktion gespeichert, sodass Sie programmgesteuert darauf zugreifen können.

Eine Beispielfunktion

```
def func():
    """This is a function that does nothing at all"""
    return
```

Die docstring kann mit dem zugegriffen werden `__doc__` Attribut:

```
print(func.__doc__)
```

> Dies ist eine Funktion, die überhaupt nichts bewirkt

```
help(func)
```

> Hilfe zur Funktion `func` im Modul `__main__` :
>
> ```
> func()
> ```
>
> Dies ist eine Funktion, die überhaupt nichts bewirkt

Eine weitere Beispielfunktion

`function.__doc__` ist nur die eigentliche Dokumentzeichenfolge als Zeichenfolge, während die `help` allgemeine Informationen zu einer Funktion einschließlich der Dokumentzeichenfolge enthält. Hier ist ein hilfreicheres Beispiel:

```
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)
```

> Hilfe zur Funktion `greet` in Modul `__main__` :
>
> ```
> greet(name, greeting='Hello')
> ```
>
> Drucken einen Gruß an den Benutzer `name`
> Optionale Parameterbegrüßung kann die `greeting` ändern.

Vorteile von Docstrings gegenüber regulären Kommentaren

Wenn Sie in einer Funktion keinen Dokumentstring oder einen regulären Kommentar eingeben, ist dies weniger hilfreich.

```
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__)
```

Keiner

```
help(greet)
```

Hilfe zur Funktionsbegrüßung im Modul **main** :

```
greet(name, greeting='Hello')
```

Schreiben Sie Dokumentation mit docstrings

Ein Dokumentstring ist ein mehrzeiliger Kommentar, der zum Dokumentieren von Modulen, Klassen, Funktionen und Methoden verwendet wird. Es muss die erste Aussage der Komponente sein, die es beschreibt.

```
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.
    """
```

Auf den Wert des Docstrings kann innerhalb des Programms zugegriffen werden und er wird beispielsweise vom Befehl `help` .

Syntaxkonventionen

PEP 257

PEP 257 definiert einen Syntaxstandard für Docstring-Kommentare. Es erlaubt grundsätzlich zwei Arten:

- Einzeilige Docstrings:

Gemäß PEP 257 sollten sie mit kurzen und einfachen Funktionen verwendet werden. Alles ist in einer Zeile angeordnet, zB:

```
def hello():
    """Say hello to your friends."""
    print("Hello my friends!")
```

Der docstring endet mit einem Punkt, das Verb sollte in der Imperativform sein.

- Mehrzeilige Docstrings:

Mehrzeilige Docstrings sollten für längere, komplexere Funktionen, Module oder Klassen verwendet werden.

```
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)
```

Sie beginnen mit einer kurzen Zusammenfassung (entspricht dem Inhalt eines einzeiligen Docstrings), die sich in derselben Zeile wie die Anführungszeichen oder in der nächsten Zeile befinden kann, und enthält zusätzliche Detail- und Listenparameter sowie Rückgabewerte.

Hinweis PEP 257 definiert, welche Informationen in einem Docstring angegeben werden sollen. Es definiert nicht, in welchem Format es angegeben werden soll. Dies war der Grund dafür, dass andere Parteien und Dokumentations-Analysewerkzeuge ihre eigenen Standards für die Dokumentation festlegen, von denen einige unten und in dieser Frage aufgeführt sind .

Sphinx

Sphinx ist ein Tool zum Generieren von HTML-basierter Dokumentation für Python-Projekte auf der Grundlage von Dokumentstrings. Die verwendete Auszeichnungssprache ist reStructuredText . Sie definieren ihre eigenen Standards für die Dokumentation, pythonhosted.org bietet eine sehr gute Beschreibung dieser Standards. Das Sphinx-Format wird beispielsweise von der pyCharm-IDE verwendet .

Eine Funktion würde mit dem Format Sphinx / reStructuredText folgendermaßen dokumentiert:

```
def hello(name, language="en"):
    """Say hello to a person.

    :param name: the name of the person
    :type name: str
    :param language: the language in which the person should be greeted
```

```
    :type language: str
    :return: a number
    :rtype: int
    """

    print(greeting[language]+" "+name)
    return 4
```

Google Python Style Guide

Google hat den Google Python Style Guide veröffentlicht, in dem Kodierungskonventionen für Python definiert sind, einschließlich Dokumentationskommentare. Im Vergleich zur Sphinx / reST sagen viele Leute, dass die Dokumentation gemäß den Richtlinien von Google für Menschen besser lesbar ist.

Die oben erwähnte Seite pythonhosted.org enthält auch einige Beispiele für eine gute Dokumentation gemäß dem Google Style Guide.

Mit dem Napoleon- Plugin kann Sphinx auch Dokumentation im Google Style Guide-kompatiblen Format analysieren.

Eine Funktion würde im Google Style Guide-Format folgendermaßen dokumentiert:

```
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
```

Kommentare und Dokumentation online lesen:
https://riptutorial.com/de/python/topic/4144/kommentare-und-dokumentation

Kapitel 92: Komplexe Mathematik

Syntax

- cmath.rect (Absolutwert, Phase)

Examples

Fortgeschrittene komplexe Arithmetik

Das Modul `cmath` enthält zusätzliche Funktionen zur Verwendung komplexer Zahlen.

```
import cmath
```

Dieses Modul kann die Phase einer komplexen Zahl im Bogenmaß berechnen:

```
z = 2+3j # A complex number
cmath.phase(z) # 0.982793723247329
```

Es erlaubt die Konvertierung zwischen der kartesischen (rechteckigen) und polaren Darstellung komplexer Zahlen:

```
cmath.polar(z) # (3.605551275463989, 0.982793723247329)
cmath.rect(2, cmath.pi/2) # (0+2j)
```

Das Modul enthält die komplexe Version von

- Exponentielle und logarithmische Funktionen (`log` ist `log10` der natürliche Logarithmus und `log10` der dezimale Logarithmus):

  ```
  cmath.exp(z) # (-7.315110094901103+1.0427436562359045j)
  cmath.log(z) # (1.2824746787307684+0.982793723247329j)
  cmath.log10(-100) # (2+1.3643763538418412j)
  ```

- Quadratwurzeln:

  ```
  cmath.sqrt(z) # (1.6741492280355401+0.8959774761298381j)
  ```

- Trigonometrische Funktionen und ihre Umkehrung:

  ```
  cmath.sin(z)  # (9.15449914691143-4.168906959966565j)
  cmath.cos(z)  # (-4.189625690968807-9.109227893755337j)
  cmath.tan(z)  # (-0.003764025641504249+1.003238627353 61j)
  cmath.asin(z) # (0.5706527843210994+1.9833870299165355j)
  cmath.acos(z) # (1.0001435424737972-1.9833870299165355j)
  cmath.atan(z) # (1.4099210495965755+0.229072682968538 78j)
  cmath.sin(z)**2 + cmath.cos(z)**2 # (1+0j)
  ```

- Hyperbolische Funktionen und ihre Umkehrung:

```
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
```

Grundlegende komplexe Arithmetik

Python bietet integrierte Unterstützung für komplexe Arithmetik. Die imaginäre Einheit wird mit `j` :

```
z = 2+3j # A complex number
w = 1-7j # Another complex number
```

Komplexe Zahlen können summiert, subtrahiert, multipliziert, dividiert und potenziert werden:

```
z + w # (3-4j)
z - w # (1+10j)
z * w # (23-11j)
z / w # (-0.38+0.34j)
z**3  # (-46+9j)
```

Python kann auch die Real- und Imaginärteile komplexer Zahlen extrahieren und deren absoluten Wert und Konjugat berechnen:

```
z.real # 2.0
z.imag # 3.0
abs(z) # 3.605551275463989
z.conjugate() # (2-3j)
```

Komplexe Mathematik online lesen: https://riptutorial.com/de/python/topic/1142/komplexe-mathematik

Kapitel 93: Kontextmanager ("mit" - Anweisung)

Einführung

Während die Kontextmanager von Python weit verbreitet sind, wissen nur wenige den Zweck ihrer Verwendung. Diese Anweisungen, die häufig beim Lesen und Schreiben von Dateien verwendet werden, unterstützen die Anwendung bei der Einsparung von Systemspeicher und verbessern die Ressourcenverwaltung, indem sie sicherstellen, dass bestimmte Ressourcen nur für bestimmte Prozesse verwendet werden. In diesem Thema wird die Verwendung der Kontextmanager von Python erläutert und veranschaulicht.

Syntax

- mit "context_manager" (als "Alias") (, "Context_manager" (als "Alias")?) *:

Bemerkungen

Kontextmanager sind in PEP 343 definiert. Sie sind als prägnanter Mechanismus für das Ressourcenmanagement gedacht, als `try ... finally` Konstrukte. Die formale Definition lautet wie folgt.

> In diesem PEP stellen `__enter__()` Methoden `__enter__()` und `__exit__()` , die beim Eintritt in den Rumpf der with-Anweisung aufgerufen werden.

Anschließend wird die `with` Anweisung wie folgt definiert.

```
with EXPR as VAR:
    BLOCK
```

Die Übersetzung der obigen Aussage lautet:

```
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

Einführung in Kontextmanager und die with-Anweisung

Ein Kontextmanager ist ein Objekt, das benachrichtigt wird, wenn ein Kontext (ein Codeblock) *beginnt* und *endet* . Sie verwenden häufig eine mit der `with` Anweisung. Es kümmert sich um die Benachrichtigung.

Beispielsweise sind Dateiobjekte Kontextmanager. Wenn ein Kontext endet, wird das Dateiobjekt automatisch geschlossen:

```
open_file = open(filename)
with open_file:
    file_contents = open_file.read()

# the open_file object has automatically been closed.
```

Das obige Beispiel wird normalerweise mit dem Schlüsselwort `as` vereinfacht:

```
with open(filename) as open_file:
    file_contents = open_file.read()

# the open_file object has automatically been closed.
```

Alles, was die Ausführung des Blocks beendet, führt zum Aufruf der Exit-Methode des Kontextmanagers. Dies schließt Ausnahmen ein und kann nützlich sein, wenn ein Fehler dazu führt, dass Sie eine geöffnete Datei oder Verbindung vorzeitig beenden. Das Beenden eines Skripts ohne ordnungsgemäßes Schließen von Dateien / Verbindungen ist eine schlechte Idee, die zu Datenverlust oder anderen Problemen führen kann. Durch die Verwendung eines Kontextmanagers können Sie sicherstellen, dass immer Vorkehrungen getroffen werden, um auf diese Weise Schäden oder Verluste zu vermeiden. Diese Funktion wurde in Python 2.5 hinzugefügt.

Einem Ziel zuweisen

Viele Kontextmanager geben bei der Eingabe ein Objekt zurück. Sie können dieses Objekt in der `with` Anweisung einem neuen Namen zuweisen.

Wenn Sie beispielsweise eine Datenbankverbindung in einer `with` Anweisung verwenden, erhalten Sie ein Cursorobjekt:

```
with database_connection as cursor:
    cursor.execute(sql_query)
```

Dateiobjekte geben sich selbst zurück, wodurch das Dateiobjekt geöffnet und als Kontextmanager in einem Ausdruck verwendet werden kann:

```
with open(filename) as open_file:
    file_contents = open_file.read()
```

Schreiben Sie Ihren eigenen Kontextmanager

Ein Kontextmanager ist ein Objekt, das zwei magische Methoden `__enter__()` und `__exit__()` implementiert (obwohl es auch andere Methoden implementieren kann):

```
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
```

Wenn der Kontext mit einer Ausnahme beendet wird , wird die Information über diese Ausnahme als Triple geführt werden `exc_type` , `exc_value` , `traceback` (das sind die gleichen Variablen wie die zurück `sys.exc_info()` Funktion). Wenn der Kontext normal beendet wird, sind alle drei Argumente `None` .

Wenn eine Ausnahme auftritt und an die `__exit__` Methode übergeben wird, kann die Methode `True` , um die Ausnahme zu unterdrücken, oder die Ausnahme wird am Ende der `__exit__` Funktion erneut `__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
```

Beachten Sie, dass im zweiten Beispiel der `__exit__` Handler weiterhin ausgeführt wird, obwohl eine Ausnahme in der Mitte des `__exit__` der mit-Anweisung `__exit__` , bevor die Ausnahme an den äußeren Bereich weitergegeben wird.

Wenn Sie nur eine `__exit__` Methode benötigen, können Sie die Instanz des Kontextmanagers zurückgeben:

```
class MyContextManager:
    def __enter__(self):
        return self

    def __exit__(self):
        print('something')
```

Schreiben Sie Ihren eigenen Kontextmanager mit der Generatorsyntax

Dank des Decorators `contextlib.contextmanager` ist es auch möglich, einen Kontextmanager unter Verwendung der Generatorsyntax zu schreiben:

```
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))
```

produziert:

```
Enter
Right in the middle with cm = 3
Exit
```

Der Dekorator vereinfacht das Schreiben eines Kontextmanagers durch Konvertieren eines Generators in einen. Alles vor dem Ertragsausdruck wird zur `__enter__` Methode, der `__enter__` wird zum vom Generator zurückgegebenen Wert (der an eine Variable in der with-Anweisung gebunden werden kann), und alles, was nach dem Ertragsausdruck geht, wird zur `__exit__` Methode.

Wenn eine Ausnahmebedingung vom Kontextmanager behandelt werden muss, kann ein `try..except..finally` Block in den Generator geschrieben werden, und jede im `with` `try..except..finally` Ausnahme wird von diesem Ausnahmeblock behandelt.

```
@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)
```

Dies erzeugt:

```
Enter
Dividing by cm = 0
Caught error
Cleaning up
Exit
```

Mehrere Kontextmanager

Sie können mehrere Content Manager gleichzeitig öffnen:

```
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')
```

Dies hat den gleichen Effekt wie das Schachteln von Kontextmanagern:

```
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')
```

Ressourcen verwalten

```
class File():
    def __init__(self, filename, mode):
        self.filename = filename
        self.mode = mode

    def __enter__(self):
        self.open_file = open(self.filename, self.mode)
        return self.open_file

    def __exit__(self, *args):
        self.open_file.close()
```

`__init__()` Methode `__init__()` richtet das Objekt ein. In diesem Fall `__init__()` den Dateinamen und den Modus für das Öffnen der Datei ein. `__enter__()` öffnet die Datei und `__exit__()` sie zurück. `__exit__()` schließt sie einfach.

Mit diesen magischen Methoden (`__enter__` , `__exit__`) können Sie Objekte implementieren, die problemlos `with` der with-Anweisung verwendet werden können.

Dateiklasse verwenden:

```
for _ in range(10000):
    with File('foo.txt', 'w') as f:
        f.write('foo')
```

Kontextmanager ("mit" -Anweisung) online lesen:
https://riptutorial.com/de/python/topic/928/kontextmanager---mit---anweisung-

Kapitel 94: Leistungsoptimierung

Bemerkungen

Wenn Sie versuchen, die Leistung eines Python-Skripts zu verbessern, sollten Sie in erster Linie den Engpass Ihres Skripts ermitteln und beachten, dass keine Optimierung eine schlechte Wahl der Datenstrukturen oder einen Fehler in Ihrem Algorithmusdesign ausgleichen kann. Sie können Leistungsengpässe identifizieren, indem Sie Ihr Skript profilieren . Zweitens sollten Sie nicht zu früh im Codierprozess auf Kosten der Lesbarkeit / des Designs / der Qualität optimieren. Donald Knuth äußerte sich zur Optimierung wie folgt:

> „Wir sollten kleine Wirkungsgrade vergessen, etwa 97% der Zeit: vorzeitige Optimierung ist die Wurzel allen Übels. Dennoch sollten wir unsere Chancen in den kritischen 3% nicht aufgeben. “

Examples

Code-Profiling

In erster Linie sollten Sie in der Lage sein, den Engpass Ihres Skripts zu finden und zu beachten, dass keine Optimierung eine schlechte Wahl der Datenstruktur oder einen Fehler in Ihrem Algorithmusdesign ausgleichen kann. Zweitens sollten Sie nicht zu früh im Codierprozess auf Kosten der Lesbarkeit / des Designs / der Qualität optimieren. Donald Knuth äußerte sich zur Optimierung wie folgt:

> "Wir sollten kleine Wirkungsgrade vergessen, etwa 97% der Zeit: vorzeitige Optimierung ist die Wurzel allen Übels. Dennoch sollten wir unsere Chancen in diesen kritischen 3% nicht aufgeben."

Um Ihren Code zu profilieren, stehen Ihnen mehrere Werkzeuge zur Verfügung: `cProfile` (oder das langsamere `profile`) aus der Standardbibliothek, `line_profiler` und `timeit` . Jeder von ihnen dient einem anderen Zweck.

`cProfile` ist ein deterministischer Profiler: Funktionsaufruf, Funktionsrückkehr und Ausnahmeereignisse werden überwacht, und für die Intervalle zwischen diesen Ereignissen (bis zu 0,001 Sekunden) werden genaue Zeitpunkte festgelegt. Die Bibliotheksdokumentation ([https://docs.python.org/2/library/profile.html __1]) liefert einen einfachen Anwendungsfall

```
import cProfile
def f(x):
    return "42!"
cProfile.run('f(12)')
```

Oder wenn Sie es vorziehen, Teile Ihres vorhandenen Codes einzuwickeln:

```
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()
```

Dadurch werden Ausgaben wie in der folgenden Tabelle erstellt, in denen Sie schnell sehen können, wo Ihr Programm die meiste Zeit verbringt, und die zu optimierenden Funktionen identifizieren.

```
         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}
```

Das Modul `line_profiler` ([https://github.com/rkern/line_profiler([1]) ist hilfreich, um eine zeilenweise Analyse des Codes zu erhalten. Dies ist offensichtlich nicht für lange Skripte zu beherrschen, sondern richtet sich an Snippets. Weitere Informationen finden Sie in der Dokumentation. Der einfachste Weg, um loszulegen, ist die Verwendung des Kernprof-Skripts, wie auf der Paketseite erläutert. Beachten Sie, dass Sie die Funktion (en) für das Profil manuell festlegen müssen.

```
$ kernprof -l script_to_profile.py
```

kernprof erstellt eine Instanz von LineProfiler und fügt sie mit dem `__builtins__` in den `__builtins__` Namespace ein. Es wurde geschrieben, um als Dekorateur verwendet zu werden, so dass Sie in Ihrem Skript die Funktionen dekorieren, die Sie mit `@profile` profilieren `@profile` .

```
@profile
def slow_function(a, b, c):
    ...
```

Das Standardverhalten von Kernprof besteht darin, die Ergebnisse in eine Binärdatei `script_to_profile.py.lprof` . Sie können kernprof anweisen, die formatierten Ergebnisse sofort mit der Option [-v / - view] am Terminal anzuzeigen. Ansonsten können Sie die Ergebnisse später wie folgt anzeigen:

```
$ python -m line_profiler script_to_profile.py.lprof
```

Schließlich bietet `timeit` eine einfache Möglichkeit zum Testen eines `timeit` oder eines kleinen Ausdrucks sowohl über die Befehlszeile als auch über die Python-Shell. Dieses Modul beantwortet Fragen, z. B. ist es schneller, ein Listenverständnis auszuführen oder die integrierte `list()` wenn Sie einen Satz in eine Liste umwandeln. Suchen Sie nach dem `setup` Schlüsselwort oder der

Option `-s` , um den Setup-Code hinzuzufügen.

```
>>> import timeit
>>> timeit.timeit('"-".join(str(n) for n in range(100))', number=10000)
0.8187260627746582
```

von einem Terminal

```
$ python -m timeit '"-".join(str(n) for n in range(100))'
10000 loops, best of 3: 40.3 usec per loop
```

Leistungsoptimierung online lesen:
https://riptutorial.com/de/python/topic/5889/leistungsoptimierung

Kapitel 95: List Destructuring (auch bekannt als Ein- und Auspacken)

Examples

Zerstörungsauftrag

In Zuweisungen können Sie eine Iterable mit der Syntax "Unpacking" in Werte aufteilen:

Zerstörung als Werte

```
a, b = (1, 2)
print(a)
# Prints: 1
print(b)
# Prints: 2
```

Wenn Sie versuchen, mehr als die Länge des Iterationsprogramms zu entpacken, wird eine Fehlermeldung angezeigt:

```
a, b, c = [1]
# Raises: ValueError: not enough values to unpack (expected 3, got 1)
```

Python 3.x 3.0

Zerstörung als Liste

Sie können eine Liste unbekannter Länge mit der folgenden Syntax entpacken:

```
head, *tail = [1, 2, 3, 4, 5]
```

Hier extrahieren wir den ersten Wert als Skalar und die anderen Werte als Liste:

```
print(head)
# Prints: 1
print(tail)
# Prints: [2, 3, 4, 5]
```

Welches ist äquivalent zu:

```
l = [1, 2, 3, 4, 5]
head = l[0]
tail = l[1:]
```

Es funktioniert auch mit mehreren Elementen oder Elementen am Ende der Liste:

```
a, b, *other, z = [1, 2, 3, 4, 5]
print(a, b, z, other)
# Prints: 1 2 5 [3, 4]
```

Werte in Destructuring-Zuweisungen ignorieren

Wenn Sie nur an einem bestimmten Wert interessiert sind, können Sie mit _ angeben, dass Sie nicht interessiert sind. Hinweis: Dies wird immer noch _ , nur verwenden die meisten Leute es nicht als Variable.

```
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

Ignorieren von Listen in Zerstörungszuweisungen

Schließlich können Sie viele Werte mit der `*_` Syntax in der Zuweisung ignorieren:

```
a, *_ = [1, 2, 3, 4, 5]
print(a)
# Prints: 1
```

Das ist nicht wirklich interessant, da Sie stattdessen die Indexierung auf der Liste verwenden könnten. Wo es schön wird, ist, die ersten und letzten Werte in einer Aufgabe zu behalten:

```
 a, *_, b = [1, 2, 3, 4, 5]
 print(a, b)
 # Prints: 1 5
```

oder mehrere Werte auf einmal extrahieren:

```
 a, _, b, _, c, *_ = [1, 2, 3, 4, 5, 6]
 print(a, b, c)
 # Prints: 1 3 5
```

Argumente für Verpackungsfunktionen

In Funktionen können Sie eine Reihe von obligatorischen Argumenten definieren:

```
def fun1(arg1, arg2, arg3):
    return (arg1,arg2,arg3)
```

Dadurch wird die Funktion nur dann aufrufbar, wenn die drei Argumente angegeben werden:

```
fun1(1, 2, 3)
```

Sie können die Argumente als optional definieren, indem Sie Standardwerte verwenden:

```
def fun2(arg1='a', arg2='b', arg3='c'):
    return (arg1,arg2,arg3)
```

So können Sie die Funktion auf viele verschiedene Arten aufrufen:

```
fun2(1)              → (1,b,c)
fun2(1, 2)           → (1,2,c)
fun2(arg2=2, arg3=3) → (a,2,3)
...
```

Sie können die Destruktursyntax jedoch auch verwenden, um Argumente zu *packen* , sodass Sie Variablen mithilfe einer `list` oder eines `dict` zuweisen können.

Eine Liste mit Argumenten packen

Stellen Sie sich vor, Sie haben eine Liste von Werten

```
l = [1,2,3]
```

Sie können die Funktion mit der Liste der Werte als Argument mit der `*` -Syntax aufrufen:

```
fun1(*l)
# Returns: (1,2,3)
fun1(*['w', 't', 'f'])
# Returns: ('w','t','f')
```

Wenn Sie jedoch keine Liste bereitstellen, deren Länge der Anzahl der Argumente entspricht:

```
fun1(*['oops'])
# Raises: TypeError: fun1() missing 2 required positional arguments: 'arg2' and 'arg3'
```

Schlüsselwortargumente packen

Jetzt können Sie Argumente auch mithilfe eines Wörterbuchs packen. Mit dem Operator `**` können Sie Python anweisen, das `dict` als Parameterwerte zu entpacken:

```
d = {
  'arg1': 1,
  'arg2': 2,
  'arg3': 3
}
fun1(**d)
# Returns: (1, 2, 3)
```

Wenn die Funktion nur Positionsargumente hat (diejenigen ohne Standardwerte), müssen Sie das Wörterbuch mit allen erwarteten Parametern enthalten und keine zusätzlichen Parameter haben.

```
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'
```

Für Funktionen mit optionalen Argumenten können Sie die Argumente auf dieselbe Weise als Wörterbuch packen:

```
fun2(**d)
# Returns: (1, 2, 3)
```

Dort können Sie jedoch Werte weglassen, da diese durch die Standardwerte ersetzt werden:

```
fun2(**{'arg2': 2})
# Returns: ('a', 2, 'c')
```

Und wie zuvor können Sie keine zusätzlichen Werte angeben, die keine vorhandenen Parameter sind:

```
fun2(**{'arg1':1, 'arg2':2, 'arg3':3, 'arg4':4})
# Raises: TypeError: fun2() got an unexpected keyword argument 'arg4'
```

In der Praxis können Funktionen sowohl positionelle als auch optionale Argumente haben, und es funktioniert gleich:

```
def fun3(arg1, arg2='b', arg3='c')
    return (arg1, arg2, arg3)
```

Sie können die Funktion nur mit einem iterierbaren Aufruf aufrufen:

```
fun3(*[1])
# Returns: (1, 'b', 'c')
fun3(*[1,2,3])
# Returns: (1, 2, 3)
```

oder nur mit einem Wörterbuch:

```
fun3(**{'arg1':1})
# Returns: (1, 'b', 'c')
fun3(**{'arg1':1, 'arg2':2, 'arg3':3})
# Returns: (1, 2, 3)
```

oder Sie können beide im selben Anruf verwenden:

```
fun3(*[1,2], **{'arg3':3})
# Returns: (1,2,3)
```

Beachten Sie jedoch, dass Sie nicht mehrere Werte für dasselbe Argument angeben können:

```
fun3(*[1,2], **{'arg2':42, 'arg3':3})
# Raises: TypeError: fun3() got multiple values for argument 'arg2'
```

Funktionsargumente auspacken

Wenn Sie eine Funktion erstellen möchten, die eine beliebige Anzahl von Argumenten akzeptieren kann und die Position oder den Namen des Arguments nicht zum Zeitpunkt des "Kompilierens" erzwingen soll, ist dies möglich.

```
def fun1(*args, **kwargs):
    print(args, kwargs)
```

Die Parameter `*args` und `**kwargs` sind spezielle Parameter, die auf ein `tuple` bzw. ein `dict` sind:

```
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}
```

Wenn Sie sich genügend Python-Code anschauen, werden Sie schnell feststellen, dass er häufig verwendet wird, wenn Sie Argumente an eine andere Funktion übergeben. Zum Beispiel, wenn Sie die String-Klasse erweitern möchten:

```
class MyString(str):
    def __init__(self, *args, **kwarg):
        print('Constructing MyString')
        super(MyString, self).__init__(*args, **kwarg)
```

List Destructuring (auch bekannt als Ein- und Auspacken) online lesen: https://riptutorial.com/de/python/topic/4282/list-destructuring--auch-bekannt-als-ein--und-auspacken-

Kapitel 96: Liste

Einführung

Die Python- **Liste** ist eine allgemeine Datenstruktur, die in Python-Programmen häufig verwendet wird. Sie werden in anderen Sprachen gefunden, oft als *dynamische Arrays bezeichnet* . Sie sind sowohl *veränderlich* als auch ein *Sequenzdatentyp* , der es ihnen ermöglicht, zu *indizieren* und in *Scheiben* zu *schneiden* . Die Liste kann verschiedene Arten von Objekten enthalten, einschließlich anderer Listenobjekte.

Syntax

- [Wert, Wert, ...]
- Liste ([iterable])

Bemerkungen

`list` ist ein bestimmter Typ von iterable, aber es ist nicht der einzige, der in Python existiert. Manchmal ist es besser, `set` , `tuple` oder `dictionary`

`list` ist der in Python angegebene Name für dynamische Arrays (ähnlich dem `vector<void*>` aus C ++ oder Javas `ArrayList<Object>`). Es ist keine verknüpfte Liste.

Der Zugriff auf Elemente erfolgt in konstanter Zeit und ist sehr schnell. Das Anhängen von Elementen am Ende der Liste wird mit konstanter Zeit abgeschrieben, es kann jedoch vorkommen, dass die gesamte `list` und kopiert wird.

Listenverständnisse beziehen sich auf Listen.

Examples

Auf Listenwerte zugreifen

Python-Listen sind nullindiziert und verhalten sich wie Arrays in anderen Sprachen.

```
lst = [1, 2, 3, 4]
lst[0]  # 1
lst[1]  # 2
```

`IndexError` versuchen, auf einen Index außerhalb der Grenzen der Liste `IndexError` wird ein `IndexError` .

```
lst[4]  # IndexError: list index out of range
```

Negative Indizes werden vom *Ende* der Liste aus gezählt.

```
lst[-1]  # 4
lst[-2]  # 3
lst[-5]  # IndexError: list index out of range
```

Dies ist funktional äquivalent zu

```
lst[len(lst)-1]  # 4
```

Listen erlauben die Verwendung der *Slice-Notation* als `lst[start:end:step]` . Der Ausgang der slice Notation ist ein neues Listenelement aus dem Index enthält `start` bis `end-1` . Wenn Optionen weggelassen werden, beginnt der `start` standardmäßig mit dem Anfang der Liste, `end` bis Ende der Liste und `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
```

Aus diesem Grund können Sie eine umgekehrte Version der Liste durch Aufrufen ausdrucken

```
lst[::-1]    # [4, 3, 2, 1]
```

Wenn Sie Schrittlängen mit negativen Beträgen verwenden, muss der Startindex größer als der Endindex sein. Andernfalls wird eine leere Liste angezeigt.

```
lst[3:1:-1] # [4, 3]
```

Die Verwendung von negativen Schrittindizes entspricht dem folgenden Code:

```
reversed(lst)[0:2] # 0 = 1 -1
                   # 2 = 3 -1
```

Die verwendeten Indizes sind um 1 niedriger als diejenigen, die bei der negativen Indexierung verwendet werden, und sind umgekehrt.

Erweitertes Schneiden

Wenn Listen in Scheiben geschnitten werden, wird die `__getitem__()` Methode des `__getitem__()` mit einem `slice` Objekt aufgerufen. Python verfügt über eine integrierte Slice-Methode zum Generieren von Slice-Objekten. Wir können dies verwenden, *um* ein Slice zu *speichern* und es später wiederzuverwenden.

```
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'
```

Dies kann von großem Nutzen sein, indem Sie unseren Objekten Slicing-Funktionen bereitstellen, indem Sie `__getitem__` in unserer Klasse überschreiben.

Listen Sie Methoden und unterstützte Operatoren auf

Beginnend mit einer gegebenen Liste `a` :

```
a = [1, 2, 3, 4, 5]
```

1. `append(value)` - fügt ein neues Element an das Ende der Liste an.

   ```
   # 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"]
   ```

   Beachten Sie, dass die `append()` -Methode nur ein neues Element an das Ende der Liste anfügt. Wenn Sie eine Liste an eine andere Liste anhängen, wird die Liste, die Sie anfügen, am Ende der ersten Liste zu einem einzelnen Element.

   ```
   # 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)` - erweitert die Liste, indem Elemente aus einem anderen Enumerable angehängt werden.

   ```
   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]
```

Listen können auch mit dem Operator `+` verkettet werden. Beachten Sie, dass dadurch keine der ursprünglichen Listen geändert wird:

```
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])` - `index(value, [startIndex])` den Index des ersten Vorkommens des Eingabewerts. Wenn der Eingabewert nicht in der Liste enthalten ist, wird eine `ValueError` Ausnahme `ValueError` . Wenn ein zweites Argument angegeben wird, wird die Suche an dem angegebenen Index gestartet.

```
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)` - fügt den `value` unmittelbar vor dem angegebenen `index` . Somit nach dem Einsetzen das neue Element einnimmt `index` .

```
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])` - Entfernt das Element am `index` und gibt es zurück. Ohne Argument wird das letzte Element der Liste entfernt und zurückgegeben.

```
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)` - Entfernt das erste Vorkommen des angegebenen Werts. Wenn der angegebene Wert nicht gefunden werden kann, wird ein `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()` - kehrt die Liste um und gibt `None` .

```
a.reverse()
# a: [8, 7, 6, 5, 4, 3, 2, 1]
```

Es gibt auch andere Möglichkeiten, eine Liste umzukehren .

8. `count(value)` - zählt die Anzahl der Vorkommen eines Wertes in der Liste.

```
a.count(7)
# Returns: 2
```

9. `sort()` - Sortiert die Liste in numerischer und lexikografischer Reihenfolge und gibt `None` .

```
a.sort()
# a = [1, 2, 3, 4, 5, 6, 7, 8]
# Sorts the list in numerical order
```

Listen können auch umgekehrt werden, wenn sie mit dem `reverse=True` Flag in der `sort()` -Methode sortiert werden.

```
a.sort(reverse=True)
# a = [8, 7, 6, 5, 4, 3, 2, 1]
```

Wenn Sie nach Attributen von Elementen sortieren möchten, können Sie das `key` :

```
import datetime

class Person(object):
    def __init__(self, name, birthday, height):
        self.name = name
        self.birthday = birthday
        self.height = height

    def __repr__(self):
        return self.name

l = [Person("John Cena", datetime.date(1992, 9, 12), 175),
     Person("Chuck Norris", datetime.date(1990, 8, 28), 180),
     Person("Jon Skeet", datetime.date(1991, 7, 6), 185)]

l.sort(key=lambda item: item.name)
# l: [Chuck Norris, John Cena, Jon Skeet]

l.sort(key=lambda item: item.birthday)
# l: [Chuck Norris, Jon Skeet, John Cena]
```

```
l.sort(key=lambda item: item.height)
# l: [John Cena, Chuck Norris, Jon Skeet]
```

Im Falle einer Liste von Diktaten ist das Konzept dasselbe:

```
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]
```

Sortieren nach Subdikt:

```
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]
```

Besser kann man mit `attrgetter` und `itemgetter`

Listen können auch mit `attrgetter` und `itemgetter` Funktionen des Bedienermoduls sortiert werden. Diese können zur Verbesserung der Lesbarkeit und Wiederverwendbarkeit beitragen. Hier sind einige Beispiele,

```
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` kann auch einen Index erhalten. Dies ist hilfreich, wenn Sie nach Indizes eines Tupels sortieren möchten.

```
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)]
```

Verwenden Sie den `attrgetter` wenn Sie nach Attributen eines Objekts sortieren möchten.

```
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()` - entfernt alle Elemente aus der Liste

    ```
    a.clear()
    # a = []
    ```

11. **Replikation** - Durch Multiplizieren einer vorhandenen Liste mit einer ganzen Zahl wird eine größere Liste erstellt, die aus diesen vielen Kopien des Originals besteht. Dies kann zum Beispiel für die Listeninitialisierung nützlich sein:

    ```
    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]
    ```

    Seien Sie vorsichtig, wenn Ihre Liste Verweise auf Objekte enthält (z. B. eine Liste von Listen). Weitere Informationen finden Sie unter Häufige Fallstricke - Listenmultiplikation und allgemeine Verweise .

12. **Element** löschen - Es ist möglich, mehrere Elemente in der Liste mit dem Schlüsselwort `del` und der Slice-Notation zu löschen:

    ```
    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. **Kopieren**

    Die Standardzuordnung "=" weist dem neuen Namen eine Referenz der ursprünglichen Liste zu. Das heißt, sowohl der ursprüngliche Name als auch der neue Name zeigen auf dasselbe Listenobjekt. Änderungen, die über eines von ihnen vorgenommen werden, werden in einem anderen angezeigt. Dies ist oft nicht das, was Sie beabsichtigten.

```
b = a
a.append(6)
# b: [1, 2, 3, 4, 5, 6]
```

Wenn Sie eine Kopie der Liste erstellen möchten, haben Sie die folgenden Optionen.

Sie können es schneiden:

```
new_list = old_list[:]
```

Sie können die eingebaute list () - Funktion verwenden:

```
new_list = list(old_list)
```

Sie können generic copy.copy () verwenden:

```
import copy
new_list = copy.copy(old_list) #inserts references to the objects found in the original.
```

Dies ist etwas langsamer als list (), da zuerst der Datentyp von old_list ermittelt werden muss.

Wenn die Liste Objekte enthält und Sie diese auch kopieren möchten, verwenden Sie generic copy.deepcopy ():

```
import copy
new_list = copy.deepcopy(old_list) #inserts copies of the objects found in the original.
```

Offensichtlich die langsamste und am meisten Speicher benötigende Methode, aber manchmal unvermeidlich.

Python 3.x 3.0

`copy()` - Gibt eine flache Kopie der Liste zurück

```
    aa = a.copy()
    # aa = [1, 2, 3, 4, 5]
```

Länge einer Liste

Verwenden Sie `len()` , um die eindimensionale Länge einer Liste abzurufen.

```
len(['one', 'two'])  # returns 2

len(['one', [2, 3], 'four'])  # returns 3, not 4
```

`len()` funktioniert auch mit Strings, Wörterbüchern und anderen Datenstrukturen, die Listen ähneln.

Beachten Sie, dass `len()` eine integrierte Funktion ist, keine Methode eines Listenobjekts.

Beachten Sie auch, dass die Kosten für `len()` `O(1)` sind. Dies bedeutet, dass die Länge einer Liste unabhängig von ihrer Länge gleich lang dauert.

Iteration über eine Liste

Python unterstützt die Verwendung einer `for` Schleife direkt in einer Liste:

```
my_list = ['foo', 'bar', 'baz']
for item in my_list:
    print(item)

# Output: foo
# Output: bar
# Output: baz
```

Sie können auch die Position jedes Artikels gleichzeitig ermitteln:

```
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
```

Die andere Möglichkeit, eine Liste basierend auf dem Indexwert zu durchlaufen:

```
for i in range(0,len(my_list)):
    print(my_list[i])
#output:
>>>
foo
bar
baz
```

Beachten Sie, dass das Ändern von Elementen in einer Liste während des Iterierens zu unerwarteten Ergebnissen führen kann:

```
for item in my_list:
    if item == 'foo':
        del my_list[0]
    print(item)

# Output: foo
# Output: baz
```

In diesem letzten Beispiel haben wir das erste Element bei der ersten Iteration gelöscht. Dies führte jedoch dazu, dass die `bar` übersprungen wurde.

Prüfen, ob ein Element in einer Liste enthalten ist

Python macht es sehr einfach zu prüfen, ob ein Element in einer Liste enthalten ist. Verwenden Sie einfach den `in` Operator.

```
lst = ['test', 'twest', 'tweast', 'treast']

'test' in lst
# Out: True

'toast' in lst
# Out: False
```

> Hinweis: Der `in` Operator on Sets ist asymptotisch schneller als auf Listen. Wenn Sie die `list` mehrmals für potenziell große Listen verwenden müssen, können Sie Ihre `list` in einen `set` konvertieren und das Vorhandensein von Elementen im `set` testen.

```
slst = set(lst)
'test' in slst
# Out: True
```

Listenelemente umkehren

Sie können die `reversed` Funktion verwenden, die einen Iterator an die stornierte Liste zurückgibt:

```
In [3]: rev = reversed(numbers)

In [4]: rev
Out[4]: [9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Beachten Sie, dass die Liste "Zahlen" durch diese Operation unverändert bleibt und in derselben Reihenfolge bleibt, in der sie ursprünglich war.

Zum Umkehren können Sie auch die `reverse` Methode verwenden .

Sie können eine Liste auch umkehren (tatsächlich erhalten Sie eine Kopie, die ursprüngliche Liste ist davon nicht betroffen), indem Sie die Slicing-Syntax verwenden und das dritte Argument (den Schritt) auf -1 setzen:

```
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]
```

Überprüfen, ob die Liste leer ist

Die Leere einer Liste ist mit dem booleschen `False` verknüpft. Sie müssen also nicht `len(lst) == 0` überprüfen, sondern nur `lst` oder `not lst`

```
lst = []
if not lst:
    print("list is empty")
```

```
# Output: list is empty
```

Listen verketten und zusammenführen

1. **Die einfachste Möglichkeit,** `list1` und `list2` **zu verketten** :

   ```
   merged = list1 + list2
   ```

2. **`zip` gibt eine Liste von Tupeln zurück** , wobei das i-te Tupel das i-te Element aus jeder der Argumentsequenzen oder iterierbaren Elemente enthält:

   ```
   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
   ```

   Wenn die Listen unterschiedliche Längen haben, enthält das Ergebnis nur so viele Elemente wie das kürzeste:

   ```
   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
   ```

   Für `itertools.zip_longest` mit ungleicher Länge bis zur längsten mit `None` s verwenden Sie `itertools.zip_longest` (`itertools.izip_longest` in Python 2).

   ```
   alist = ['a1', 'a2', 'a3']
   blist = ['b1']
   clist = ['c1', 'c2', 'c3', 'c4']

   for a,b,c in itertools.zip_longest(alist, blist, clist):
       print(a, b, c)

   # Output:
   # a1 b1 c1
   # a2 None c2
   ```

```
# a3 None c3
# None None c4
```

3. **Einfügen in einen bestimmten Indexwert:**

```
alist = [123, 'xyz', 'zara', 'abc']
alist.insert(3, [2009])
print("Final List :", alist)
```

Ausgabe:

```
Final List : [123, 'xyz', 'zara', 2009, 'abc']
```

Alle und alle

Sie können `all()` , um zu bestimmen, ob alle Werte in einer iterierbaren Komponente als "True" ausgewertet werden

```
nums = [1, 1, 0, 1]
all(nums)
# False
chars = ['a', 'b', 'c', 'd']
all(chars)
# True
```

In ähnlicher Weise bestimmt `any()` , ob ein oder mehrere Werte in einem iterierbaren Wert True ergeben

```
nums = [1, 1, 0, 1]
any(nums)
# True
vals = [None, None, None, False]
any(vals)
# False
```

Während dieses Beispiel eine Liste verwendet, ist es wichtig zu beachten, dass diese eingebauten Ins mit beliebigen Iteratoren, einschließlich Generatoren, funktionieren.

```
vals = [1, 2, 3, 4]
any(val > 12 for val in vals)
# False
any((val * 2) > 6 for val in vals)
# True
```

Entfernen Sie doppelte Werte in der Liste

Das Entfernen doppelter Werte in einer Liste kann durch Konvertieren der Liste in eine `set` (dh eine ungeordnete Sammlung verschiedener Objekte) erfolgen. Wenn eine `list` benötigt wird, kann der Satz mithilfe der Funktionsliste `list()` wieder in eine Liste konvertiert werden:

```
names = ["aixk", "duke", "edik", "tofp", "duke"]
list(set(names))
# Out: ['duke', 'tofp', 'aixk', 'edik']
```

Beachten Sie, dass durch das Konvertieren einer Liste in einen Satz die ursprüngliche Reihenfolge verloren geht.

Um die Reihenfolge der Liste zu erhalten, kann ein `OrderedDict`

```
import collections
>>> collections.OrderedDict.fromkeys(names).keys()
# Out: ['aixk', 'duke', 'edik', 'tofp']
```

Zugriff auf Werte in der verschachtelten Liste

Beginnend mit einer dreidimensionalen Liste:

```
alist = [[[1,2],[3,4]], [[5,6,7],[8,9,10], [12, 13, 14]]]
```

Zugriff auf Elemente in der Liste:

```
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
```

Unterstützungsvorgänge durchführen:

```
alist[0][0].append(11)
print(alist[0][0][2])
#11
#Appends 11 to the end of the first list in the first list
```

Verschachtelte for-Schleifen verwenden, um die Liste zu drucken:

```
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]
```

Beachten Sie, dass diese Operation in einem Listenverständnis oder sogar als Generator zur Erzeugung von Wirkungsgraden verwendet werden kann, z.

```
[col for row in alist for col in row]
```

```
#[[1, 2, 11], [3, 4], [5, 6, 7], [8, 9, 10], [12, 13, 14]]
```

Nicht alle Elemente in den äußeren Listen müssen selbst Listen sein:

```
alist[1].insert(2, 15)
#Inserts 15 into the third position in the second list
```

Eine andere Möglichkeit, verschachtelte for-Schleifen zu verwenden. Der andere Weg ist besser, aber ich musste dies gelegentlich verwenden:

```
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]
```

Verwenden von Slices in einer verschachtelten Liste:

```
print(alist[1][1:])
#[[8, 9, 10], 15, [12, 13, 14]]
#Slices still work
```

Die endgültige Liste:

```
print(alist)
#[[[1, 2, 11], [3, 4]], [[5, 6, 7], [8, 9, 10], 15, [12, 13, 14]]]
```

Listenvergleich

Es ist möglich, Listen und andere Sequenzen mit Vergleichsoperatoren lexikographisch zu vergleichen. Beide Operanden müssen vom gleichen Typ sein.

```
[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
```

Wenn eine der Listen am Anfang der anderen enthalten ist, gewinnt die kürzeste Liste.

```
[1, 10] < [1, 10, 100]
# True
```

Initialisieren einer Liste mit einer festen Anzahl von Elementen

Für **unveränderliche** Elemente (z. B. `None` , String-Literale usw.):

```
my_list = [None] * 10
my_list = ['test'] * 10
```

Bei **veränderlichen** Elementen führt das gleiche Konstrukt dazu, dass sich alle Elemente der Liste auf dasselbe Objekt beziehen, beispielsweise für eine Menge:

```
>>> 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}]
```

Um die Liste mit einer festen Anzahl **verschiedener veränderlicher** Objekte zu initialisieren, verwenden Sie **Folgendes** :

```
my_list=[{1} for _ in range(10)]
```

Liste online lesen: https://riptutorial.com/de/python/topic/209/liste

Kapitel 97: Listen Sie Verständnis auf

Einführung

Listenverständnisse in Python sind prägnante, syntaktische Konstrukte. Sie können verwendet werden, um Listen aus anderen Listen zu generieren, indem auf jedes Element in der Liste Funktionen angewendet werden. Im folgenden Abschnitt wird die Verwendung dieser Ausdrücke erläutert und veranschaulicht.

Syntax

- [x + 1 für x in (1, 2, 3)] # Listenverständnis, ergibt [2, 3, 4]
- (x + 1 für x in (1, 2, 3)) # Generatorausdruck ergibt 2, dann 3, dann 4
- [x für x in (1, 2, 3), wenn x% 2 == 0] # Listenverständnis mit Filter, ergibt [2]
- [x + 1 wenn x% 2 == 0, sonst x für x in (1, 2, 3)] # Listenverständnis mit ternär
- [x + 1, wenn x% 2 == 0, sonst x für x im Bereich (-3,4), wenn x> 0] # Listenverständnis mit ternärer und Filterung
- {x für x in (1, 2, 2, 3)} # Setverstehen, ergibt {1, 2, 3}
- {k: v für k, v in [('a', 1), ('b', 2)]} # dict Verständnis, ergibt {'a': 1, 'b': 2} (Python 2.7+ und 3.0+ nur)
- [x + y für x in [1, 2] für y in [10, 20]] # Verschachtelte Schleifen, ergibt [11, 21, 12, 22]
- [x + y für x in [1, 2, 3], wenn x> 2 für y in [3, 4, 5]] # Bedingung am 1. für Schleife geprüft
- [x + y für x in [1, 2, 3] für y in [3, 4, 5], wenn x> 2] # Bedingung am 2. für Schleife geprüft
- [x für x in xrange (10), wenn x% 2 == 0] # Bedingung geprüft, wenn geschleifte Zahlen ungerade Zahlen sind

Bemerkungen

Verständnis sind syntaktische Konstrukte, die Datenstrukturen oder Ausdrücke definieren, die für eine bestimmte Sprache eindeutig sind. Durch die korrekte Verwendung von Verständnis werden diese in leicht verständliche Ausdrücke neu interpretiert. Als Ausdrücke können sie verwendet werden:

- auf der rechten Seite der Aufgaben
- als Argumente für Funktionsaufrufe
- im Körper einer Lambda-Funktion
- als eigenständige Aussagen. (Zum Beispiel: `[print(x) for x in range(10)]`)

Examples

Listenverständnisse

Ein Listenverständnis erstellt eine neue `list` indem auf jedes Element eines iterierbaren Elements ein Ausdruck angewendet wird . Die grundlegendste Form ist:

```
[ <expression> for <element> in <iterable> ]
```

Es gibt auch eine optionale 'if'-Bedingung:

```
[ <expression> for <element> in <iterable> if <condition> ]
```

Jedes `<element>` in `<iterable>` wird an `<expression>` wenn die (optionale) `<condition>` als wahr ausgewertet wird . Alle Ergebnisse werden sofort in der neuen Liste zurückgegeben. Generatorausdrücke werden träge ausgewertet, Listenauffassungen bewerten jedoch den gesamten Iterator sofort, wobei der Speicher proportional zur Länge des Iterators ist.

So erstellen Sie eine `list` von quadrierten Ganzzahlen:

```
squares = [x * x for x in (1, 2, 3, 4)]
# squares: [1, 4, 9, 16]
```

Der `for` Ausdruck setzt `x` für jeden Wert aus `(1, 2, 3, 4)` . Das Ergebnis des Ausdrucks `x * x` wird an eine interne `list` angehängt. Die interne `list` wird den variablen `squares` zugewiesen, wenn sie abgeschlossen sind.

Abgesehen von einer Geschwindigkeitssteigerung (wie hier erklärt) entspricht ein Listenverständnis ungefähr der folgenden for-Schleife:

```
squares = []
for x in (1, 2, 3, 4):
    squares.append(x * x)
# squares: [1, 4, 9, 16]
```

Der auf jedes Element angewendete Ausdruck kann so komplex sein wie nötig:

```
# 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']
```

---

sonst

`else` kann in List-Verständniskonstrukten verwendet werden. Seien Sie jedoch vorsichtig bei der Syntax. Die if / else-Klauseln sollten vor der `for` Schleife verwendet werden, nicht nach:

```
# 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']
```

Beachten Sie, dass dies ein anderes Sprachkonstrukt verwendet, einen bedingten Ausdruck , der selbst nicht zur Verständnissyntax gehört . Das `if` after- `for…in` *ist* ein Teil von List-Verständnis und dient zum *Filtern von* Elementen aus der Quelle.

Doppelte Iteration

Die Reihenfolge der doppelten Iteration `[... for x in ... for y in ...]` ist entweder natürlich oder kontraintuitiv. Als Faustregel gilt ein Äquivalent `for` Schleife:

```
def foo(i):
    return i, i + 0.5

for i in range(3):
    for x in foo(i):
        yield str(x)
```

Dies wird zu:

```
[str(x)
    for i in range(3)
        for x in foo(i)
]
```

Dies kann als `[str(x) for i in range(3) for x in foo(i)]` in eine Zeile komprimiert werden.

In-Place-Mutation und andere Nebenwirkungen

Verstehen Sie vor dem Verwenden des Listenverständnisses den Unterschied zwischen Funktionen, die für ihre Nebeneffekte (*Mutations*- oder *In-Place-* Funktionen) aufgerufen werden, die normalerweise `None` , und Funktionen, die einen interessanten Wert zurückgeben.

Viele Funktionen (vor allem *reine* Funktionen) nehmen einfach ein Objekt und geben ein Objekt zurück. Eine *In-Place*- Funktion ändert das vorhandene Objekt, was als *Nebeneffekt bezeichnet wird* . Andere Beispiele umfassen Eingabe- und Ausgabeoperationen wie Drucken.

`list.sort()` sortiert eine Liste *an Ort und Stelle* (bedeutet, dass die ursprüngliche Liste `list.sort()`) und gibt den Wert `None` . Daher funktioniert es nicht wie erwartet in einem Listenverständnis:

```
[x.sort() for x in [[2, 1], [4, 3], [0, 1]]]
# [None, None, None]
```

Stattdessen `sorted()` eine sortierte `list` anstatt direkt zu sortieren:

```
[sorted(x) for x in [[2, 1], [4, 3], [0, 1]]]
# [[1, 2], [3, 4], [0, 1]]
```

Die Verwendung von Nachteilen für Nebenwirkungen ist möglich, z. B. E / A- oder In-Place-Funktionen. Eine for-Schleife ist jedoch in der Regel besser lesbar. Während dies in Python 3 funktioniert:

```
[print(x) for x in (1, 2, 3)]
```

Verwenden Sie stattdessen:

```
for x in (1, 2, 3):
    print(x)
```

In einigen Situationen *sind* Nebeneffekt Funktionen geeignet für Liste Verständnis. `random.randrange()` hat den Nebeneffekt, den Zustand des Zufallszahlengenerators zu ändern, gibt aber auch einen interessanten Wert zurück. `next()` kann außerdem auf einem Iterator aufgerufen werden.

Der folgende Zufallswertgenerator ist nicht rein, macht jedoch Sinn, da der Zufallsgenerator bei jeder Auswertung des Ausdrucks zurückgesetzt wird:

```
from random import randrange
[randrange(1, 7) for _ in range(10)]
# [2, 3, 2, 1, 1, 5, 2, 4, 3, 5]
```

Whitespace in Listenverständnissen

Kompliziertere Listenverständnisse können eine unerwünschte Länge erreichen oder weniger lesbar werden. Obwohl es in Beispielen weniger üblich ist, ist es möglich, ein Listenverständnis wie folgt in mehrere Zeilen aufzuteilen:

```
[
    x for x
    in 'foo'
    if x not in 'bar'
]
```

Wörterbuch Verständnis

Ein Wörterbuchverständnis ähnelt einem Listenverständnis, außer dass es ein Wörterbuchobjekt anstelle einer Liste erzeugt.

Ein grundlegendes Beispiel:

Python 2.x 2.7

```
{x: x * x for x in (1, 2, 3, 4)}
# Out: {1: 1, 2: 4, 3: 9, 4: 16}
```

Das ist nur eine andere Schreibweise:

```
dict((x, x * x) for x in (1, 2, 3, 4))
# Out: {1: 1, 2: 4, 3: 9, 4: 16}
```

Wie bei einem Listenverständnis können wir eine bedingte Anweisung innerhalb des Diktierverstehens verwenden, um nur die Diktierelemente zu erzeugen, die ein bestimmtes Kriterium erfüllen.

Python 2.x 2.7

```
{name: len(name) for name in ('Stack', 'Overflow', 'Exchange') if len(name) > 6}
# Out: {'Exchange': 8, 'Overflow': 8}
```

Oder mit einem Generatorausdruck umgeschrieben.

```
dict((name, len(name)) for name in ('Stack', 'Overflow', 'Exchange') if len(name) > 6)
# Out: {'Exchange': 8, 'Overflow': 8}
```

Beginnend mit einem Wörterbuch und Verwenden des Wörterbuchverständnisses als Schlüsselwertpaarfilter

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}
```

Schlüssel und Wert des Wörterbuchs umschalten (Wörterbuch umkehren)

Wenn Sie über ein *Diktier* verfügen, das einfache *Hashwerte enthält* (doppelte Werte können unerwartete Ergebnisse haben):

```
my_dict = {1: 'a', 2: 'b', 3: 'c'}
```

und Sie wollten die Schlüssel und Werte austauschen, Sie können je nach Codierstil verschiedene Methoden verwenden:

- `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

Wenn Ihr Wörterbuch groß ist , *sollten Sie itertools importieren* und `izip` oder `imap` .

Wörterbücher zusammenführen

Kombinieren Sie Wörterbücher und überschreiben Sie gegebenenfalls alte Werte mit einem verschachtelten Wörterbuchverständnis.

```
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}
```

Das Auspacken von Wörterbüchern (PEP 448) kann jedoch bevorzugt sein.

Python 3.x 3.5

```
{**dict1, **dict2}
# Out: {'w': 1, 'x': 2, 'y': 2, 'z': 2}
```

Hinweis : Die Wörterbücher wurden in Python 3.0 hinzugefügt und im Gegensatz zu den Listenfunktionen, die in 2.0 hinzugefügt wurden, auf 2.7+ zurückportiert. Versionen <2.7 können Generatorausdrücke und das `dict()` verwenden, um das Verhalten von Wörterbuchverstehen zu simulieren.

Generator-Ausdrücke

Generatorausdrücke sind Listenverständnissen sehr ähnlich. Der Hauptunterschied besteht darin, dass nicht alle Ergebnisse auf einmal erstellt werden. Es erstellt ein Generatorobjekt, das dann wiederholt werden kann.

Sehen Sie sich zum Beispiel den Unterschied im folgenden Code an:

```
# 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>
```

Dies sind zwei sehr unterschiedliche Objekte:

- Die folgende Liste gibt ein Verständnis `list` Objekt während der Generator ein Verständnis kehrt `generator` .
- `generator` können nicht indiziert werden und verwenden die `next` Funktion, um die Elemente in der richtigen Reihenfolge zu erhalten.

Hinweis : Wir verwenden `xrange` da auch hier ein Generatorobjekt erstellt wird. Wenn wir den Bereich verwenden würden, würde eine Liste erstellt. `xrange` ist auch nur in einer späteren Version von Python 2 vorhanden. In Python 3 gibt `range` lediglich einen Generator zurück. Weitere Informationen finden Sie im Beispiel *Unterschiede zwischen Range- und Xrange-Funktionen* .

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

> HINWEIS: Die Funktion `g.next()` sollte durch `next(g)` und `xrange` mit `range` da `Iterator.next()` und `xrange()` in Python 3 nicht vorhanden sind.

Obwohl beide auf ähnliche Weise durchlaufen werden können:

```
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
"""
```

Anwendungsfälle

Generatorausdrücke werden träge ausgewertet, was bedeutet, dass sie jeden Wert nur dann erzeugen und zurückgeben, wenn der Generator wiederholt wird. Dies ist häufig nützlich, wenn Sie große Datensätze durchlaufen, um zu vermeiden, dass ein Duplikat des Datensatzes im Arbeitsspeicher erstellt werden muss:

```
for square in (x**2 for x in range(1000000)):
    #do something
```

Ein weiterer häufiger Anwendungsfall ist das Vermeiden der Iteration über eine gesamte Iterierbarkeit, wenn dies nicht erforderlich ist. In diesem Beispiel wird bei jeder `get_objects()` von `get_objects()` ein Element von einer Remote-API abgerufen. Es können Tausende von Objekten vorhanden sein, die einzeln abgerufen werden müssen, und wir müssen nur wissen, ob ein Objekt existiert, das mit einem Muster übereinstimmt. Durch Verwendung eines Generatorausdrucks, wenn ein Objekt auf das Muster passt.

```
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
```

Verstehen festlegen

Das Satzverständnis ähnelt dem Listen- und Wörterbuchverständnis , erzeugt jedoch einen Satz , bei dem es sich um eine ungeordnete Sammlung eindeutiger Elemente handelt.

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'])
```

Live Demo

Denken Sie daran, dass Sets ungeordnet sind. Dies bedeutet, dass die Reihenfolge der Ergebnisse in der Gruppe von der in den obigen Beispielen dargestellten abweichen kann.

Hinweis : Das Satzverständnis ist ab Python 2.7+ verfügbar, im Gegensatz zu Listenverständnissen, die in 2.0 hinzugefügt wurden. In Python 2.2 bis Python 2.6 kann die Funktion `set()` mit einem Generatorausdruck verwendet werden, um dasselbe Ergebnis zu erzeugen:

Python 2.x 2.2

```
set(x for x in range(5))
# Out: {0, 1, 2, 3, 4}
```

Vermeiden Sie sich wiederholende und teure Operationen mit Bedingungsklausel

Betrachten Sie das folgende Listenverständnis:

```
>>> 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, ...]
```

Dies führt zu zwei Aufrufen von `f(x)` für 1.000 Werte von `x` : einen Aufruf zum Erzeugen des Werts und den anderen zum Überprüfen der `if` Bedingung. Wenn `f(x)` eine besonders teure Operation ist, kann dies erhebliche Auswirkungen auf die Leistung haben. Schlimmer noch, wenn der Aufruf von `f()` Nebenwirkungen hat, kann dies zu überraschenden Ergebnissen führen.

Stattdessen sollten Sie die teure Operation nur einmal für jeden Wert von `x` auswerten, indem Sie eine iterierbare Zwischenstufe (Generatorausdruck) wie folgt erstellen:

```
>>> [v for v in (f(x) for x in range(1000)) if v > 10]
[16, 25, 36, ...]
```

Oder mit der eingebauten Karte äquivalent:

```
>>> [v for v in map(f, range(1000)) if v > 10]
[16, 25, 36, ...]
```

Eine andere Möglichkeit, die zu einem besser lesbaren Code führen könnte, besteht darin, das partielle Ergebnis (`v` im vorherigen Beispiel) in eine Iteration (z. B. eine Liste oder ein Tupel) zu setzen und anschließend zu iterieren. Da `v` das einzige Element in der Iteration sein wird, haben wir jetzt einen Bezug auf die Ausgabe unserer langsamen Funktion, die nur einmal berechnet wird:

```
>>> [v for x in range(1000) for v in [f(x)] if v > 10]
[16, 25, 36, ...]
```

In der Praxis kann die Logik des Codes jedoch komplizierter sein, und es ist wichtig, dass er lesbar bleibt. Im Allgemeinen wird eine separate Generatorfunktion gegenüber einem komplexen Einzeiler empfohlen:

```
>>> 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, ...]
```

Eine weitere Möglichkeit , die Berechnung zu verhindern , `f(x)` mehrfach ist es, die verwenden `@functools.lru_cache()` (Python 3.2 +) Dekorateur auf `f(x)` . Auf diese Weise wird, da die Ausgabe von `f` für die Eingabe `x` bereits einmal berechnet wurde, der zweite Funktionsaufruf des ursprünglichen Listenverständnisses so schnell wie eine Wörterbuchsuche. Bei diesem Ansatz wird zur Verbesserung der Effizienz Memoization verwendet, die mit der Verwendung von Generatorausdrücken vergleichbar ist.

Angenommen, Sie müssen eine Liste abflachen

```
l = [[1, 2, 3], [4, 5, 6], [7], [8, 9]]
```

Einige der Methoden könnten sein:

```
reduce(lambda x, y: x+y, l)

sum(l, [])

list(itertools.chain(*l))
```

Listenverständnis würde jedoch die beste zeitliche Komplexität bieten.

```
[item for sublist in l for item in sublist]
```

Die auf + basierenden Verknüpfungen (einschließlich der implizierten Verwendung in summe) sind notwendigerweise O (L ^ 2), wenn L-Unterlisten vorhanden sind. Da die Zwischenergebnisliste immer länger wird, wird bei jedem Schritt ein neues Zwischenergebnislistenobjekt angezeigt zugewiesen, und alle Elemente des vorherigen Zwischenergebnisses müssen kopiert werden (sowie einige neue am Ende hinzugefügt). Also (zur Vereinfachung und ohne tatsächlichen Verlust der Allgemeinheit) sagen Sie, dass Sie L Unterlisten von jeweils I-Elementen haben: Die ersten I-Elemente werden L-1-mal hin und her kopiert, die zweiten I-Elemente L-2-mal usw. Gesamtzahl der Kopien ist I mal die Summe von x für x von 1 bis L ausgeschlossen, dh I * (L ** 2) / 2.

Das Listenverständnis generiert nur einmal eine Liste und kopiert jedes Element (vom ursprünglichen Wohnort bis zur Ergebnisliste) ebenfalls genau einmal.

Verständnis für Tupel

Die `for` Klausel eines Listenverständnisses kann mehrere Variablen angeben:

```
[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]
```

Das ist wie bei regulären `for` Schleifen:

```
for x, y in [(1,2), (3,4), (5,6)]:
    print(x+y)
# 3
# 7
# 11
```

Wenn der Ausdruck, mit dem das Verständnis beginnt, ein Tupel ist, muss er jedoch in Klammern gesetzt werden:

```
[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)]
```

Zählen von Vorkommnissen anhand des Verständnisses

Wenn wir die Anzahl der Elemente in einem iterierbaren Element zählen möchten, die bestimmte Bedingungen erfüllen, können wir das Verständnis nutzen, um eine idiomatische Syntax zu erzeugen:

```
# 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
```

Das Grundkonzept kann wie folgt zusammengefasst werden:

1. Iteriere über die Elemente im `range(1000)` .
2. Verketten Sie alle erforderlichen `if` Bedingungen.
3. Verwenden Sie 1 als *Ausdruck* , um eine 1 für jeden Artikel zurückzugeben, der die Bedingungen erfüllt.
4. Fassen Sie alle `1` s zusammen, um die Anzahl der Elemente zu bestimmen, die die Bedingungen erfüllen.

Hinweis : Hier sammeln wir nicht die `1` in einer Liste (beachten Sie das Fehlen eckiger Klammern), sondern übergeben die 1 direkt an die `sum` , die sie aufsummiert. Dies wird als *Generatorausdruck bezeichnet* , der einem Verständnis ähnlich ist.

Typen in einer Liste ändern

Quantitative Daten werden häufig als Strings eingelesen, die vor der Verarbeitung in numerische Typen umgewandelt werden müssen. Die Typen aller Listenelemente können entweder mit List Comprehension oder der Funktion `map()` konvertiert werden .

```
# 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]
```

Listen Sie Verständnis auf online lesen: https://riptutorial.com/de/python/topic/196/listen-sie-verstandnis-auf

Kapitel 98: Listenaufteilung (Auswählen von Listenteilen)

Syntax

- a [start: end] # Elemente beginnen bis Ende-1
- a [start:] # Elemente beginnen durch den Rest des Arrays
- a [: end] # Elemente vom Anfang bis Ende-1
- a [start: end: step] # Schritt für Schritt nicht vorbei
- a [:] # eine Kopie des gesamten Arrays
- Quelle

Bemerkungen

- `lst[::-1]` Sie eine umgekehrte Kopie der Liste
- `start` oder `end` kann eine negative Zahl sein, dh sie zählt vom Ende des Arrays statt vom Anfang. So:

```
a[-1]    # last item in the array
a[-2:]   # last two items in the array
a[:-2]   # everything except the last two items
```

(Quelle)

Examples

Verwenden Sie das dritte Argument "step"

```
lst = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']

lst[::2]
# Output: ['a', 'c', 'e', 'g']

lst[::3]
# Output: ['a', 'd', 'g']
```

Auswahl einer Unterliste aus einer Liste

```
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']
```

Umkehren einer Liste mit dem Schneiden

```
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]
```

Verschieben einer Liste mit dem Schneiden

```
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]
```

Listenaufteilung (Auswählen von Listenteilen) online lesen:
https://riptutorial.com/de/python/topic/1494/listenaufteilung--auswahlen-von-listenteilen-

Kapitel 99: Listenverständnisse

Einführung

Ein Listenverständnis ist ein syntaktisches Werkzeug zum Erstellen von Listen auf natürliche und prägnante Weise, wie im folgenden Code veranschaulicht, um eine Liste mit Quadraten der Zahlen 1 bis 10 zu erstellen: `[i ** 2 for i in range(1,11)]` Die Dummy - `i` aus einer bestehenden Liste `range` ist ein neues Element Muster machen verwendet. Es wird verwendet, wenn eine for-Schleife in weniger ausdrucksstarken Sprachen erforderlich ist.

Syntax

- [i für i in range (10)] # grundlegendes Listenverständnis
- [i for i in xrange (10)] # grundlegendes Listenverständnis mit Generatorobjekt in Python 2.x
- [i für i im Bereich (20), wenn i% 2 == 0] # mit Filter
- [x + y für x in [1, 2, 3] für y in [3, 4, 5]] # verschachtelten Schleifen
- [i wenn i> 6, sonst 0 für i im Bereich (10)] # ternärer Ausdruck
- [i wenn i> 4 sonst 0 für i im Bereich (20), wenn i% 2 == 0] # mit Filter und ternärem Ausdruck
- [[x + y für x in [1, 2, 3]] für y in [3, 4, 5]] # geschachteltes Listenverständnis

Bemerkungen

Listenverständnisse wurden in PEP 202 beschrieben und in Python 2.0 eingeführt.

Examples

Bedingte Listenverständnisse

Bei einem Listenverständnis können Sie eine oder mehrere `if` Bedingungen anhängen, um Werte zu filtern.

```
[<expression> for <element> in <iterable> if <condition>]
```

Für jedes `<element>` in `<iterable>` ; Wenn `<condition>` zu `True` ausgewertet wird, fügen Sie der zurückgegebenen Liste `<expression>` (normalerweise eine Funktion von `<element>`) hinzu.

Dies kann zum Beispiel verwendet werden, um nur gerade Zahlen aus einer Folge von ganzen Zahlen zu extrahieren:

```
[x for x in range(10) if x % 2 == 0]
# Out: [0, 2, 4, 6, 8]
```

Live-Demo

Der obige Code entspricht:

```
even_numbers = []
for x in range(10):
    if x % 2 == 0:
        even_numbers.append(x)

print(even_numbers)
# Out: [0, 2, 4, 6, 8]
```

Auch ein bedingtes Listenverständnis der Form `[e for x in y if c]` (wobei `e` und `c` Ausdrücke in Form von `x`) ist gleichbedeutend mit `list(filter(lambda x: c, map(lambda x: e, y)))` .

Beachten Sie trotz des gleichen Ergebnisses, dass das erste Beispiel fast doppelt so schnell ist wie das zweite. Für Neugierige ist dies eine nette Erklärung für den Grund.

Beachten Sie, dass dies ganz anders ist als der Bedingungsausdruck `... if ... else ...` (manchmal auch als ternärer Ausdruck bezeichnet), den Sie für den Teil `<expression>` des Listenverständnisses verwenden können. Betrachten Sie das folgende Beispiel:

```
[x if x % 2 == 0 else None for x in range(10)]
# Out: [0, None, 2, None, 4, None, 6, None, 8, None]
```

Live-Demo

Hier ist der Bedingungsausdruck kein Filter, sondern ein Operator, der den für die Listenelemente zu verwendenden Wert bestimmt:

```
<value-if-condition-is-true> if <condition> else <value-if-condition-is-false>
```

Dies wird deutlicher, wenn Sie es mit anderen Operatoren kombinieren:

```
[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]
```

Live-Demo

Wenn Sie Python 2.7 verwenden, ist `xrange` aus verschiedenen Gründen möglicherweise besser als der `range` wie in der `xrange` Dokumentation beschrieben .

```
[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]
```

Der obige Code entspricht:

```
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]
```

Man kann ternäre Ausdrücke und `if` Bedingungen kombinieren. Der ternäre Operator arbeitet mit dem gefilterten Ergebnis:

```
[x if x > 2 else '*' for x in range(10) if x % 2 == 0]
# Out: ['*', '*', 4, 6, 8]
```

Dasselbe konnte nicht nur von einem ternären Operator erreicht werden:

```
[x if (x > 2 and x % 2 == 0) else '*' for x in range(10)]
# Out:['*', '*', '*', '*', 4, '*', 6, '*', 8, '*']
```

Siehe auch: Filter , die häufig eine ausreichende Alternative zu bedingten Listenverständnissen darstellen.

Listenverständnisse mit verschachtelten Schleifen

Listenverständnisse können verschachtelt `for` Schleifen verwendet werden. Sie können eine beliebige Anzahl von verschachtelten for-Schleifen innerhalb eines Listenverständnisses codieren. Jeder `for` Schleife kann ein optionaler `if` Test zugeordnet werden. Dabei ist die Reihenfolge der `for` ist Konstrukte die gleiche Reihenfolge wie bei einer Reihe von verschachtelten Schreiben `for` Anweisungen. Die allgemeine Struktur von Listenverstehen sieht folgendermaßen aus:

```
[ expression for target1 in iterable1 [if condition1]
             for target2 in iterable2 [if condition2]...
             for targetN in iterableN [if conditionN] ]
```

Der folgende Code reduziert beispielsweise eine Liste mit mehreren `for` Anweisungen:

```
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]
```

kann als Listenverständnis mit mehreren `for` Konstrukte äquivalent geschrieben werden:

```
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]
```

Live Demo

Sowohl in der erweiterten Form als auch im Listenverständnis steht die äußere Schleife (erste Anweisung) an erster Stelle.

Das verschachtelte Verständnis ist nicht nur kompakter, sondern auch wesentlich schneller.

```
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
```

Der Aufwand für den Funktionsaufruf oben beträgt ungefähr *140ns* .

Inline, `if` s ähnlich verschachtelt sind und an einer beliebigen Stelle nach der ersten `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]
```

Live Demo

Der besseren Lesbarkeit halber sollten Sie jedoch traditionelle *For-Loops verwenden* . Dies gilt insbesondere, wenn das Schachteln mehr als zwei Ebenen tief ist und / oder die Logik des Verständnisses zu komplex ist. Das Verständnis für mehrere verschachtelte Schleifenlisten kann fehleranfällig sein oder unerwartete Ergebnisse liefern.

Refactoring-Filter und Karte zum Auflisten von Verständnis

Die `filter` oder `map` sollten häufig durch Listenverständnisse ersetzt werden. Guido Van Rossum beschreibt dies gut in einem offenen Brief aus dem Jahr 2005 :

> `filter(P, S)` wird fast immer klarer geschrieben als `[x for x in S if P(x)]` , und dies hat den großen Vorteil, dass die häufigsten Verwendungen Vergleichselemente beinhalten, z. B. `x==42` und definierend Ein Lambda dafür erfordert nur viel mehr Aufwand für den Leser (plus das Lambda ist langsamer als das Listenverständnis). Dies gilt umso mehr für die `map(F, S)` die zu `[F(x) for x in S]` . In vielen Fällen können Sie jedoch stattdessen Generatorausdrücke verwenden.

Die folgenden Codezeilen gelten als " *nicht pythonisch* " und verursachen Fehler in vielen Python-Linters.

```
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
```

Nach dem, was wir aus dem vorherigen Zitat gelernt haben, können wir diese `filter` und `map` in ihre entsprechenden *Listenverständnisse unterteilen* . Entfernen Sie auch die *Lambda*- Funktionen von jedem - dadurch wird der Code lesbarer.

```
# 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)]
```

Die Lesbarkeit wird noch deutlicher bei Verkettungsfunktionen. Wo aus Gründen der Lesbarkeit die Ergebnisse einer Map- oder Filterfunktion an die nächste weitergegeben werden sollten; In einfachen Fällen können diese durch ein einzelnes Listenverständnis ersetzt werden. Darüber hinaus können wir anhand des Listenverständnisses leicht erkennen, was das Ergebnis unseres Prozesses ist, wo die kognitive Belastung höher ist, wenn über den verketteten Map & Filter-Prozess vorgegangen wird.

```
# Map & Filter
filtered = filter(lambda x: x % 2 == 0, range(10))
results = map(lambda x: 2*x, filtered)

# List comprehension
results = [2*x for x in range(10) if x % 2 == 0]
```

Refactoring - Kurzanleitung

- **Karte**

  ```
  map(F, S) == [F(x) for x in S]
  ```

- **Filter**

  ```
  filter(P, S) == [x for x in S if P(x)]
  ```

Dabei sind `F` und `P` Funktionen, die jeweils Eingangswerte transformieren und einen `bool`

Verschachtelte Listenverständnisse

Geschachtelte Listenverständnisse sind im Gegensatz zu Listenverständnissen mit verschachtelten Schleifen Listenverständnisse im Listenverständnis. Der Anfangsausdruck kann

ein beliebiger Ausdruck sein, einschließlich eines anderen Listenverständnisses.

```
#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]]
```

Das verschachtelte Beispiel ist äquivalent zu

```
l = []
for y in [3, 4, 5]:
    temp = []
    for x in [1, 2, 3]:
        temp.append(x + y)
    l.append(temp)
```

Ein Beispiel, bei dem ein verschachteltes Verständnis verwendet werden kann, um eine Matrix zu transponieren.

```
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]]
```

Wie bei verschachtelten `for` Schleifen gibt es keine Grenzen dafür, wie tiefes Verständnis verschachtelt werden kann.

```
[[[i + j + k for k in 'cd'] for j in 'ab'] for i in '12']
# Out: [[['1ac', '1ad'], ['1bc', '1bd']], [['2ac', '2ad'], ['2bc', '2bd']]]
```

Iteriere zwei oder mehr Listen gleichzeitig innerhalb des Listenverständnisses

Um innerhalb des *Listenverständnisses* mehr als zwei Listen gleichzeitig zu *durchlaufen* , kann `zip()` werden:

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

Listenverständnisse online lesen: https://riptutorial.com/de/python/topic/5265/listenverstandnisse

Kapitel 100: Mathematik-Modul

Examples

Rundung: rund, Boden, Decke, Rumpf

Zusätzlich zu den integrierten in `round` Funktion, die `math` - Modul bietet den `floor` , `ceil` und `trunc` Funktionen.

```
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` und `round` immer einen `float` .

```
round(1.3)  # 1.0
```

`round` bricht immer Krawatten von Null.

```
round(0.5)  # 1.0
round(1.5)  # 2.0
```

Python 3.x 3.0

`floor` , `ceil` und `trunc` immer einen `Integral` , während `round` einen `Integral` zurückgibt, wenn sie mit einem Argument aufgerufen werden.

```
round(1.3)      # 1
round(1.33, 1)  # 1.3
```

`round` bricht Gleichungen zur nächsten geraden Zahl. Dies korrigiert die Tendenz zu größeren Zahlen, wenn eine große Anzahl von Berechnungen ausgeführt wird.

```
round(0.5)  # 0
round(1.5)  # 2
```

Warnung!

Wie bei jeder Gleitkommadarstellung können einige Brüche *nicht genau dargestellt werden* . Dies kann zu unerwartetem Rundungsverhalten führen.

```
round(2.675, 2)  # 2.67, not 2.68!
```

Warnung vor der Boden-, Abbruch- und Ganzzahlteilung negativer Zahlen

Python (und C ++ und Java) runden bei negativen Zahlen von Null ab. Erwägen:

```
>>> math.floor(-1.7)
-2.0
>>> -5 // 2
-3
```

Logarithmen

`math.log(x)` gibt den natürlichen Logarithmus (Basis `e`) von `x` .

```
math.log(math.e)  # 1.0
math.log(1)       # 0.0
math.log(100)     # 4.605170185988092
```

`math.log` kann aufgrund von Beschränkungen der Fließkommazahlen mit Zahlen nahe 1 Genauigkeit verlieren. Um Logs nahe 1 zu berechnen, verwenden Sie `math.log1p` , die den natürlichen Logarithmus von 1 plus das Argument auswertet:

```
math.log(1 + 1e-20)  # 0.0
math.log1p(1e-20)    # 1e-20
```

`math.log10` kann für Logs Base 10 verwendet werden:

```
math.log10(10)  # 1.0
```

Python 2.x 2.3.0

Bei Verwendung mit zwei Argumenten gibt `math.log(x, base)` den Logarithmus von `x` in der angegebenen `base` (dh `log(x) / log(base)` .

```
math.log(100, 10) # 2.0
math.log(27, 3)   # 3.0
math.log(1, 10)   # 0.0
```

Zeichen kopieren

In Python 2.6 und höher gibt `math.copysign(x, y) x` mit dem Zeichen von `y` . Der zurückgegebene Wert ist immer ein `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
```

Trigonometrie

Länge der Hypotenuse berechnen

```
math.hypot(2, 4) # Just a shorthand for SquareRoot(2**2 + 4**2)
# Out: 4.47213595499958
```

Grad in Radiant umrechnen

Alle `math` Funktionen erwarten **Radiant,** daher müssen Sie Grad in Radiant konvertieren:

```
math.radians(45)              # Convert 45 degrees to radians
# Out: 0.7853981633974483
```

Alle Ergebnisse der inversen trigonometischen Funktionen geben das Ergebnis im Bogenmaß zurück, daher müssen Sie es möglicherweise in Grad umrechnen.

```
math.degrees(math.asin(1))    # Convert the result of asin to degrees
# Out: 90.0
```

Sinus-, Cosinus-, Tangenten- und Umkehrfunktionen

```
# 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"
```

Neben dem `math.atan` gibt es auch eine `math.atan2` Funktion mit zwei Argumenten, die den korrekten Quadranten berechnet und Fallstricke der Division durch Null vermeidet:

```
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"
```

Hyperbolischer Sinus, Cosinus und Tangens

```
# 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
```

Konstanten

`math` enthalten zwei häufig verwendete mathematische Konstanten.

- `math.pi` - Die mathematische Konstante pi
- `math.e` - Die mathematische Konstante *e* (Basis des natürlichen Logarithmus)

```
>>> from math import pi, e
>>> pi
3.141592653589793
>>> e
2.718281828459045
>>>
```

Python 3.5 und höher haben Konstanten für unendlich und NaN ("keine Zahl"). Die ältere Syntax der Übergabe eines Strings an `float()` funktioniert immer noch.

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
```

Imaginäre Zahlen

Imaginäre Zahlen in Python werden durch ein "j" oder "J" hinter der Zielnummer dargestellt.

```
1j         # Equivalent to the square root of -1.
1j * 1j    # = (-1+0j)
```

Infinity und NaN ("keine Zahl")

In allen Versionen von Python können wir unendlich und NaN ("keine Zahl") wie folgt darstellen:

```
pos_inf = float('inf')     # positive infinity
neg_inf = float('-inf')    # negative infinity
not_a_num = float('nan')   # NaN ("not a number")
```

In Python 3.5 und höher können wir auch die definierten Konstanten `math.inf` und `math.nan` :

Python 3.x 3.5

```
pos_inf = math.inf
neg_inf = -math.inf
not_a_num = math.nan
```

Die String-Darstellungen werden als `inf` und `-inf` und `nan` angezeigt:

```
pos_inf, neg_inf, not_a_num
# Out: (inf, -inf, nan)
```

Wir können mit der `isinf` Methode auf positive oder negative Unendlichkeit `isinf` :

```
math.isinf(pos_inf)
# Out: True

math.isinf(neg_inf)
# Out: True
```

Wir können durch direkten Vergleich gezielt auf positive Unendlichkeit oder auf negative Unendlichkeit testen:

```
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 und höher ermöglicht auch die Überprüfung der Endlichkeit:

Python 3.x 3.2

```
math.isfinite(pos_inf)
# Out: False

math.isfinite(0.0)
# Out: True
```

Vergleichsoperatoren arbeiten erwartungsgemäß für positive und negative Unendlichkeit:

```
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
```

Wenn jedoch ein arithmetischer Ausdruck einen Wert erzeugt, der größer als das Maximum ist, das als `float` , wird er unendlich.

```
pos_inf == sys.float_info.max * 1.0000001
# Out: True

neg_inf == -sys.float_info.max * 1.0000001
# Out: True
```

Die Division durch Null führt jedoch nicht zu einem Ergebnis von unendlich (oder, wenn angemessen, mit negativem unendlich), sondern löst eine `ZeroDivisionError` Ausnahme aus.

```
try:
```

```
    x = 1.0 / 0.0
    print(x)
except ZeroDivisionError:
    print("Division by zero")

# Out: Division by zero
```

Rechenoperationen im Unendlichen geben nur unendliche Ergebnisse oder manchmal 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 ist niemals irgendetwas gleich, nicht einmal sich selbst. Wir können es mit der `isnan` Methode `isnan` :

```
not_a_num == not_a_num
# Out: False

math.isnan(not_a_num)
Out: True
```

NaN wird immer als "nicht gleich" verglichen, aber niemals kleiner als oder größer als:

```
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
```

Rechenoperationen an NaN geben immer NaN. Dies beinhaltet die Multiplikation mit -1: Es gibt kein "negatives NaN".

```
5.0 * not_a_num
# Out: nan

float('-nan')
# Out: nan
```

Python 3.x 3.5

```
-math.nan
# Out: nan
```

Es gibt einen geringfügigen Unterschied zwischen den alten `float` Versionen von NaN und Infinity und den `math` Bibliothekskonstanten 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 für schnellere Potenzierung

Verwenden des Timeit-Moduls über die Befehlszeile:

```
> 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
```

Der eingebaute `**` -Operator ist oft praktisch, wenn Leistung jedoch von entscheidender Bedeutung ist, verwenden Sie math.pow. Beachten Sie jedoch, dass pow Float zurückgibt, auch wenn die Argumente Ganzzahlen sind:

```
> from math import pow
> pow(5,5)
3125.0
```

Komplexe Zahlen und das cmath-Modul

Das `cmath` Modul ähnelt dem `math` Modul, definiert jedoch Funktionen, die für die komplexe Ebene geeignet sind.

Komplexe Zahlen sind vor allem ein numerischer Typ, der Teil der Python-Sprache selbst ist und nicht von einer Bibliotheksklasse bereitgestellt wird. Daher müssen wir `import cmath` für gewöhnliche arithmetische Ausdrücke nicht `import cmath` .

Beachten Sie, dass wir `j` (oder `J`) und nicht `i` .

```
z = 1 + 3j
```

Wir müssen `1j` da `j` der Name einer Variablen und nicht ein numerisches Literal ist.

```
1j * 1j
Out: (-1+0j)

1j ** 1j
```

```
# Out: (0.20787957635076193+0j)     # "i to the i"  ==  math.e ** -(math.pi/2)
```

Wir haben den `real` und den `imag` (Imaginärteil) sowie das komplexe `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
```

Die integrierten Funktionen `abs` und `complex` sind ebenfalls Teil der Sprache selbst und erfordern keinen Import:

```
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)
```

Die `complex` Funktion kann eine Zeichenfolge enthalten, darf jedoch keine Leerzeichen enthalten:

```
complex('1+1j')
# Out: (1+1j)

complex('1 + 1j')
# Exception: ValueError: complex() arg is a malformed string
```

Für die meisten Funktionen benötigen wir jedoch das Modul, zum Beispiel `sqrt` :

```
import cmath

cmath.sqrt(-1)
# Out: 1j
```

Natürlich ist das Verhalten von `sqrt` bei komplexen Zahlen und reellen Zahlen unterschiedlich. In der nicht komplexen `math` löst die Quadratwurzel einer negativen Zahl eine Ausnahme aus:

```
import math

math.sqrt(-1)
# Exception: ValueError: math domain error
```

Es werden Funktionen zum Konvertieren in und von Polarkoordinaten bereitgestellt:

```
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)
```

Das mathematische Feld der komplexen Analyse liegt außerhalb des Rahmens dieses Beispiels, aber viele Funktionen in der komplexen Ebene haben einen "Astschnitt", üblicherweise entlang der realen Achse oder der imaginären Achse. Die meisten modernen Plattformen unterstützen die "Vorzeichenlose Null", wie in IEEE 754 angegeben, wodurch die Funktionen dieser Funktionen auf beiden Seiten des Astschnittes erhalten bleiben. Das folgende Beispiel stammt aus der Python-Dokumentation:

```
cmath.phase(complex(-1.0, 0.0))
# Out: 3.141592653589793

cmath.phase(complex(-1.0, -0.0))
# Out: -3.141592653589793
```

Das `cmath` Modul bietet auch viele Funktionen mit direkten Gegenstücken aus dem `math` .

Neben `sqrt` gibt es komplexe Versionen von `exp` , `log` , `log10` , den trigonometrischen Funktionen und ihren Inversen (`sin` , `cos` , `tan` , `asin` , `acos` , `atan`) und den hyperbolischen Funktionen und ihren Inversen (`sinh` , `cosh` , `tanh` , `asinh` , `acosh` , `atanh`). Es gibt jedoch kein komplexes Gegenstück zu `math.atan2` , der Zwei-Argument-Form von arctangent.

```
cmath.log(1+1j)
# Out: (0.3465735902799726+0.7853981633974483j)

cmath.exp(1j * cmath.pi)
# Out: (-1+1.2246467991473532e-16j)   # e to the i pi == -1, within rounding error
```

Die Konstanten `pi` und `e` sind angegeben. Beachten Sie, dass diese `float` und nicht `complex` .

```
type(cmath.pi)
# Out: <class 'float'>
```

Das `cmath` Modul bietet auch komplexe Versionen von `isinf` und (für Python 3.2+) ist `isfinite` . Siehe " Infinity und NaN ". Eine komplexe Zahl wird als unendlich betrachtet, wenn entweder der Realteil oder der Imaginärteil unendlich ist.

```
cmath.isinf(complex(float('inf'), 0.0))
# Out: True
```

Ebenso bietet das `cmath` Modul eine komplexe Version von `isnan` . Siehe " Infinity und NaN ". Eine komplexe Zahl wird als "keine Zahl" betrachtet, wenn entweder ihr Realteil oder ihr Imaginärteil "keine Zahl" ist.

```
cmath.isnan(0.0, float('nan'))
```

```
# Out: True
```

Beachten Sie, dass es kein `cmath` Gegenstück zu den Konstanten `math.inf` und `math.nan` (ab Python 3.5 und höher).

Python 3.x 3.5

```
cmath.isinf(complex(0.0, math.inf))
# Out: True

cmath.isnan(complex(math.nan, 0.0))
# Out: True

cmath.inf
# Exception: AttributeError: module 'cmath' has no attribute 'inf'
```

In Python 3.5 und höher gibt es in den Modulen `cmath` und `math` eine `isclose` Methode.

Python 3.x 3.5

```
z = cmath.rect(*cmath.polar(1+1j))

z
# Out: (1.0000000000000002+1.0000000000000002j)

cmath.isclose(z, 1+1j)
# True
```

Mathematik-Modul online lesen: https://riptutorial.com/de/python/topic/230/mathematik-modul

Kapitel 101: Mehrdimensionale Arrays

Examples

Listen in Listen

Eine gute Möglichkeit, ein 2D-Array zu visualisieren, ist eine Liste mit Listen. Etwas wie das:

```
lst=[[1,2,3],[4,5,6],[7,8,9]]
```

hier die äußere Liste `lst` hat drei Dinge drin. jedes dieser Dinge ist eine andere Liste: Die erste ist: `[1,2,3]` , die zweite ist: `[4,5,6]` und die dritte ist: `[7,8,9]` . Sie können auf diese Listen auf dieselbe Weise zugreifen, wie Sie auch auf ein anderes Element einer Liste zugreifen würden:

```
print (lst[0])
#output: [1, 2, 3]

print (lst[1])
#output: [4, 5, 6]

print (lst[2])
#output: [7, 8, 9]
```

Sie können dann auf die verschiedenen Elemente in jeder dieser Listen auf dieselbe Weise zugreifen:

```
print (lst[0][0])
#output: 1

print (lst[0][1])
#output: 2
```

Hier bedeutet die erste Zahl in den `[]` -Klammern, dass die Liste an dieser Position angezeigt wird. Im obigen Beispiel haben wir die Zahl `0` , um die Liste an der 0-ten Position zu erhalten, dh `[1,2,3]` . Der zweite Satz von `[]` -Klammern bedeutet, dass der Artikel an dieser Position von der inneren Liste abgerufen wird. In diesem Fall haben wir sowohl `0` als auch `1` verwendet. Die `0` te Position in der Liste, die wir erhalten haben, ist die Nummer `1` und in der ersten Position ist es `2`

Sie können Werte auch in diesen Listen auf dieselbe Weise festlegen:

```
lst[0]=[10,11,12]
```

Nun ist die Liste `[[10,11,12],[4,5,6],[7,8,9]]` . In diesem Beispiel haben wir die gesamte erste Liste in eine komplett neue Liste geändert.

```
lst[1][2]=15
```

Nun ist die Liste `[[10,11,12],[4,5,15],[7,8,9]]` . In diesem Beispiel haben wir ein einzelnes Element in einer der inneren Listen geändert. Zuerst gingen wir in die Liste auf Position 1 und änderten das Element in Position 2 (6), jetzt sind es 15.

Listen in Listen in Listen in ...

Dieses Verhalten kann erweitert werden. Hier ist ein 3-dimensionales Array:

```
[[[111,112,113],[121,122,123],[131,132,133]],[[211,212,213],[221,222,223],[231,232,233]],[[311,312,313
```

Wie es offensichtlich ist, wird das etwas schwer lesbar. Verwenden Sie umgekehrte Schrägstriche, um die verschiedenen Dimensionen aufzuteilen:

```
[[[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]]]
```

Durch das Verschachteln der Listen auf diese Weise können Sie beliebig hohe Dimensionen erweitern.

Der Zugriff ist ähnlich wie bei 2D-Arrays:

```
print(myarray)
print(myarray[1])
print(myarray[2][1])
print(myarray[1][0][2])
etc.
```

Und das Editieren ist auch ähnlich:

```
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.
```

Mehrdimensionale Arrays online lesen:
https://riptutorial.com/de/python/topic/8186/mehrdimensionale-arrays

Kapitel 102: Metaklassen

Einführung

Metaclasses können Sie das Verhalten von Python - Klassen zu tief ändern (in Bezug darauf, wie sie definiert sind, instanziiert, zugegriffen wird , und mehr) von dem Ersatz - `type` Metaklasse , die neuen Klassen standardmäßig verwenden.

Bemerkungen

Berücksichtigen Sie beim Entwerfen Ihrer Architektur, dass viele Dinge, die mit Metaklassen erreicht werden können, auch mit einfacheren Semantiken erreicht werden können:

- Traditionelles Erbe ist oft mehr als genug.
- Klassendekorateure können Funktionalität auf Ad-hoc-Weise in Klassen einmischen.
- Python 3.6 führt `__init_subclass__()` , mit der eine Klasse an der Erstellung ihrer Unterklasse teilnehmen kann.

Examples

Grundlegende Metaklassen

Wenn `type` mit drei Argumenten aufgerufen wird, verhält es sich wie die (Meta-) Klasse und erstellt eine neue Instanz, d. H. es erzeugt eine neue Klasse / einen neuen Typ.

```
Dummy = type('OtherDummy', (), dict(x=1))
Dummy.__class__               # <type 'type'>
Dummy().__class__.__class__   # <type 'type'>
```

Es ist möglich, eine Unterklasse `type` eine benutzerdefinierte metaclass zu erstellen.

```
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
```

Jetzt haben wir eine neue benutzerdefinierte `mytype` Metaklasse, mit der Klassen auf dieselbe Weise wie `type` .

```
MyDummy = mytype('MyDummy', (), dict(x=2))
MyDummy.__class__               # <class '__main__.mytype'>
MyDummy().__class__.__class__   # <class '__main__.mytype'>
MyDummy.__custom_attribute__    # 2
```

Wenn wir mit dem Schlüsselwort `class` eine neue Klasse erstellen, wird die Metaklasse standardmäßig auf Basis der Basisklassen ausgewählt.

```
>>> class Foo(object):
...     pass

>>> type(Foo)
type
```

Im obigen Beispiel ist die einzige Basenklasse `object` also ist unsere Metaklasse der `object` , also `type` . Es ist möglich, den Standard zu überschreiben, es hängt jedoch davon ab, ob wir Python 2 oder Python 3 verwenden:

Python 2.x 2.7

Ein spezielles Klassenebenenattribut `__metaclass__` kann verwendet werden, um die Metaklasse anzugeben.

```
class MyDummy(object):
    __metaclass__ = mytype
type(MyDummy)  # <class '__main__.mytype'>
```

Python 3.x 3.0

Ein spezielles Schlüsselwort für eine `metaclass` die Metaklasse an.

```
class MyDummy(metaclass=mytype):
    pass
type(MyDummy)  # <class '__main__.mytype'>
```

Alle Schlüsselwortargumente (außer `metaclass`) in der Klassendeklaration werden an die Metaklasse übergeben. Daher gibt die `class MyDummy(metaclass=mytype, x=2) x=2` als Schlüsselwortargument an den `mytype` Konstruktor weiter.

Lesen Sie diese ausführliche Beschreibung der Python-Metaklassen für weitere Details.

Singletons mit Metaklassen

Ein Singleton ist ein Muster, das die Instantiierung einer Klasse auf eine Instanz / ein Objekt beschränkt. Weitere Informationen zu Python-Singleton-Entwurfsmustern finden Sie hier .

```
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
```

Verwenden einer Metaklasse

Metaklassensyntax

Python 2.x 2.7

```
class MyClass(object):
    __metaclass__ = SomeMetaclass
```

Python 3.x 3.0

```
class MyClass(metaclass=SomeMetaclass):
    pass
```

Python 2 und 3 Kompatibilität mit six

```
import six

class MyClass(six.with_metaclass(SomeMetaclass)):
    pass
```

Benutzerdefinierte Funktionalität mit Metaklassen

Die Funktionalität in Metaklassen kann so geändert werden, dass beim Erstellen einer Klasse eine Zeichenfolge in die Standardausgabe gedruckt wird oder eine Ausnahme ausgelöst wird. Diese Metaklasse gibt den Namen der zu erstellenden Klasse aus.

```
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
```

Sie können die Metaklasse wie folgt verwenden:

```
class Spam(metaclass=VerboseMetaclass):
```

```
    def eggs(self):
        print("[insert example string here]")
s = Spam()
s.eggs()
```

Die Standardausgabe wird sein:

```
Creating class Spam
[insert example string here]
```

Einführung in Metaklassen

Was ist eine Metaklasse?

In Python ist alles ein Objekt: Ganzzahlen, Strings, Listen, sogar Funktionen und Klassen selbst sind Objekte. Und jedes Objekt ist eine Instanz einer Klasse.

Um die Klasse eines Objekts x zu überprüfen, kann `type(x)` aufgerufen werden.

```
>>> type(5)
<type 'int'>
>>> type(str)
<type 'type'>
>>> type([1, 2, 3])
<type 'list'>

>>> class C(object):
...     pass
...
>>> type(C)
<type 'type'>
```

Die meisten Klassen in Python sind Instanzen des `type` . `type` selbst ist auch eine Klasse. Solche Klassen, deren Instanzen auch Klassen sind, werden Metaklassen genannt.

Die einfachste Metaclass

OK, also gibt es in Python bereits eine Metaklasse: `type` . Können wir noch einen schaffen?

```
class SimplestMetaclass(type):
    pass

class MyClass(object):
    __metaclass__ = SimplestMetaclass
```

Das fügt keine Funktionalität hinzu, aber es ist eine neue Metaklasse. Sehen Sie, dass MyClass jetzt eine Instanz von SimplestMetaclass ist:

```
>>> type(MyClass)
<class '__main__.SimplestMetaclass'>
```

Eine Metaklasse, die etwas tut

Eine Metaklasse, die etwas tut, überschreibt normalerweise `__new__` `type` , um einige Eigenschaften der zu erstellenden Klasse zu ändern, bevor das ursprüngliche `__new__` , das die Klasse erstellt:

```
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)
```

Die Standard-Metaklasse

Sie haben vielleicht gehört, dass alles in Python ein Objekt ist. Es stimmt, und alle Objekte haben eine Klasse:

```
>>> type(1)
int
```

Das Literal 1 ist eine Instanz von `int` . Lass uns eine Klasse deklarieren:

```
>>> class Foo(object):
...    pass
...
```

Jetzt können wir es instanziieren:

```
>>> bar = Foo()
```

Was ist die Klasse der `bar` ?

```
>>> type(bar)
Foo
```

Schön, `bar` ist ein Beispiel für `Foo` . Aber was ist die Klasse von `Foo` selbst?

```
>>> type(Foo)
type
```

Ok, `Foo` selbst ist eine Instanz des `type` . Wie wäre es mit dem `type` selbst?

```
>>> type(type)
type
```

Was ist also eine Metaklasse? Lassen Sie uns jetzt so tun, als wäre es nur ein schicker Name für die Klasse einer Klasse. Zum Mitnehmen:

- Alles ist ein Objekt in Python, also hat alles eine Klasse
- Die Klasse einer Klasse wird als Metaklasse bezeichnet
- Die Standard-Metaklasse ist `type` und bei weitem die häufigste Metaklasse

Aber warum sollten Sie sich mit Metaklassen auskennen? Nun, Python selbst ist ziemlich "hackbar", und das Konzept der Metaklasse ist wichtig, wenn Sie fortgeschrittene Dinge wie Meta-Programmierung machen oder die Initialisierung Ihrer Klassen kontrollieren möchten.

Metaklassen online lesen: https://riptutorial.com/de/python/topic/286/metaklassen

Kapitel 103: Mit ZIP-Archiven arbeiten

Syntax

- zip-Datei importieren
- *Klasse* Zip-Datei. **ZipFile** (*Datei, Modus = 'r', Komprimierung = ZIP_STORED, allowZip64 = True*)

Bemerkungen

Wenn Sie versuchen, eine Datei zu öffnen, die keine ZIP-Datei ist, wird die Ausnahme `zipfile.BadZipFile` .

In Python 2.7 wurde dies als `zipfile.BadZipfile` , und dieser alte Name wird neben dem neuen Namen in Python 3.2+ beibehalten

Examples

Zip-Dateien öffnen

Importieren Sie zunächst das `zipfile` Modul und legen Sie den Dateinamen fest.

```
import zipfile
filename = 'zipfile.zip'
```

Das Arbeiten mit ZIP-Archiven ist dem Arbeiten mit Dateien sehr ähnlich. Sie erstellen das Objekt, indem Sie die ZIP-Datei öffnen. So können Sie daran arbeiten, bevor Sie die Datei wieder schließen.

```
zip = zipfile.ZipFile(filename)
print(zip)
# <zipfile.ZipFile object at 0x0000000002E51A90>
zip.close()
```

In Python 2.7 und Python in 3 Versionen höher als 3.2 können wir die Verwendung `with` Kontext - Manager. Wir öffnen die Datei im "read" -Modus und drucken dann eine Liste mit Dateinamen:

```
with zipfile.ZipFile(filename, 'r') as z:
    print(zip)
    # <zipfile.ZipFile object at 0x0000000002E51A90>
```

Inhalt der Zipdatei überprüfen

Es gibt mehrere Möglichkeiten, den Inhalt einer ZIP-Datei zu überprüfen. Sie können das `printdir` , um eine Vielzahl von Informationen an `stdout` senden

```
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.
```

Wir können auch eine Liste von Dateinamen mit der `namelist` Methode erhalten. Hier drucken wir einfach die Liste aus:

```
with zipfile.ZipFile(filename) as zip:
    print(zip.namelist())

# Out: ['pyexpat.pyd', 'python.exe', 'python3.dll', 'python35.dll', ... etc. ...]
```

Anstelle der `namelist` können wir die `infolist` Methode `infolist` , die eine Liste von `ZipInfo` Objekten `ZipInfo` , die zusätzliche Informationen zu jeder Datei enthalten, z. B. Zeitstempel und Dateigröße:

```
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
```

Extrahieren der ZIP-Datei in ein Verzeichnis

Extrahieren Sie den gesamten Dateiinhalt einer ZIP-Datei

```
import zipfile
with zipfile.ZipFile('zipfile.zip','r') as zfile:
    zfile.extractall('path')
```

Wenn Sie zum Extrahieren einzelner Dateien die Extraktionsmethode verwenden möchten, werden Namen und Pfad als Eingabeparameter verwendet

```
import zipfile
f=open('zipfile.zip','rb')
zfile=zipfile.ZipFile(f)
for cont in zfile.namelist():
    zfile.extract(cont,path)
```

Neue Archive erstellen

Um ein neues Archiv zu erstellen, öffnen Sie die ZIP-Datei mit dem Schreibmodus.

```
import zipfile
new_arch=zipfile.ZipFile("filename.zip",mode="w")
```

Um diesem Archiv Dateien hinzuzufügen, verwenden Sie die write () - Methode.

```
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()
```

Wenn Sie eine Bytefolge in das Archiv schreiben möchten, können Sie die Methode writestr () verwenden.

```
str_bytes="string buffer"
new_arch.writestr('filename_string_in_archive.txt',str_bytes)
new_arch.close()
```

Mit ZIP-Archiven arbeiten online lesen: https://riptutorial.com/de/python/topic/3728/mit-zip-archiven-arbeiten

Kapitel 104: Mixins

Syntax

- class *ClassName* (*MainClass* , *Mixin1* , *Mixin2* , ...): # Wird verwendet, um eine Klasse mit dem Namen *ClassName* , der ersten (ersten) Klasse *MainClass* und *Mixins1* , *Mixin2* usw. zu *deklarieren* .
- class *ClassName* (*Mixin1* , *MainClass* , *Mixin2* , ...): # Die ' *main'* -Klasse muss nicht die erste Klasse sein. Es gibt wirklich keinen Unterschied zwischen ihm und dem Mixin

Bemerkungen

Das Hinzufügen eines Mixins zu einer Klasse sieht sehr nach dem Hinzufügen einer Superklasse aus, weil es so ziemlich nur das ist. Ein Objekt einer Klasse mit dem Mixin- *Foo* wird auch eine Instanz von *Foo sein* und `isinstance(instance, Foo)` gibt true zurück

Examples

Mixin

Ein **Mixin** ist eine Reihe von Eigenschaften und Methoden, die in verschiedenen Klassen verwendet werden können, die *nicht* aus einer Basisklasse stammen. In objektorientierten Programmiersprachen verwenden Sie normalerweise *Vererbung* , um Objekten unterschiedlicher Klassen die gleiche Funktionalität zu geben. Wenn eine Gruppe von Objekten über eine bestimmte Fähigkeit verfügt, ordnen Sie diese Fähigkeit einer Basisklasse zu, von der beide Objekte *erben* .

> Angenommen, Sie haben die Klassen `Car` , `Boat` und `Plane` . Objekte aus allen diesen Klassen verfügen über die Fähigkeit zu reisen, sodass sie die Funktion " `travel` . In diesem Szenario reisen sie auch alle auf dieselbe Weise. indem Sie sich eine Route besorgen und sich darauf bewegen. Um diese Funktion zu implementieren, können Sie alle Klassen von `Vehicle` ableiten und die Funktion in diese gemeinsam genutzte Klasse einfügen:

```
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):
   ...
```

Mit diesem Code können Sie die `travel` mit einem Auto (`car.travel("Montana")`), einem Boot (`boat.travel("Hawaii")`) und einem Flugzeug (`plane.travel("France")`) `plane.travel("France")`

Was ist jedoch, wenn Sie über eine Funktionalität verfügen, die für eine Basisklasse nicht verfügbar ist? Angenommen, Sie möchten beispielsweise `Car` ein Radio und die Möglichkeit geben, mit `play_song_on_station` ein Lied auf einem Radiosender `play_song_on_station` , aber Sie haben auch eine `Clock` , die auch ein Radio verwenden kann. `Car` und `Clock` können eine Basisklasse (`Machine`) gemeinsam nutzen. Es können jedoch nicht alle Maschinen Songs abspielen. `Boat` und `Plane` können nicht (zumindest in diesem Beispiel). Wie schaffen Sie es also, ohne Code zu duplizieren? Sie können ein Mixin verwenden. In Python ist es so einfach, einer Klasse ein Mixin zuzuweisen, indem Sie es der Liste der Unterklassen wie dieser hinzufügen

```
class Foo(main_super, mixin): ...
```

`Foo` werden alle Eigenschaften und Methoden der erben `main_super` , sondern auch die von `mixin` auch.

Um den Klassen `Car` und Clock die Möglichkeit zu geben, ein Radio zu verwenden, können Sie `Car` aus dem letzten Beispiel überschreiben und Folgendes schreiben:

```
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):
   ...
```

Jetzt können Sie `car.play_song_on_station(98.7)` und `clock.play_song_on_station(101.3)` , aber nicht so etwas wie `boat.play_song_on_station(100.5)`

Das Wichtigste bei Mixins ist, dass Sie damit Funktionen für viele verschiedene Objekte hinzufügen können, die keine "Haupt" -Unterklasse mit dieser Funktionalität teilen, aber trotzdem den Code für sie verwenden. Ohne Mixins wäre das Ausführen des oben genannten Beispiels viel schwieriger und / oder könnte eine gewisse Wiederholung erfordern.

Methoden in Mixins überschreiben

Mixins sind eine Art Klasse, mit der zusätzliche Eigenschaften und Methoden in eine Klasse "eingemischt" werden. Dies ist in der Regel in Ordnung, da die Mixin-Klassen die Methoden des anderen oder der Basisklasse häufig nicht überschreiben. Wenn Sie jedoch Methoden oder Eigenschaften in Ihren Mixins überschreiben, kann dies zu unerwarteten Ergebnissen führen, da in Python die Klassenhierarchie von rechts nach links definiert ist.

Nehmen Sie zum Beispiel die folgenden Klassen

```
class Mixin1(object):
    def test(self):
        print "Mixin1"

class Mixin2(object):
    def test(self):
        print "Mixin2"

class BaseClass(object):
    def test(self):
        print "Base"

class MyClass(BaseClass, Mixin1, Mixin2):
    pass
```

In diesem Fall ist die Mixin2-Klasse die Basisklasse, erweitert um Mixin1 und schließlich um BaseClass. Also, wenn wir das folgende Code-Snippet ausführen:

```
>>> x = MyClass()
>>> x.test()
Base
```

Wir sehen, dass das zurückgegebene Ergebnis von der Basisklasse stammt. Dies kann zu unerwarteten Fehlern in der Logik Ihres Codes führen und muss berücksichtigt und berücksichtigt werden

Mixins online lesen: https://riptutorial.com/de/python/topic/4359/mixins

Kapitel 105: Module importieren

Syntax

- import *modulname*
- import *modulname.submodulname*
- aus *Modulname* import *
- von *module_name* Import *submodule_name* [, *class_name, function_name,* ... etc]
- *Importiere* aus *modulname einige_namen* als *neue_namen*
- aus *modulname.submodulname* import *klassenname* [, *funktionsname* , ... etc]

Bemerkungen

Beim Importieren eines Moduls wertet Python den gesamten Code der obersten Ebene in diesem Modul aus, sodass alle Funktionen, Klassen und Variablen, die das Modul enthält, *gelernt werden* . Wenn Sie möchten, dass ein Modul an einem anderen Ort importiert wird, achten Sie auf den Code der obersten Ebene und kapseln Sie es in `if __name__ == '__main__':` wenn Sie nicht möchten, dass das Modul beim Importieren des Moduls ausgeführt wird.

Examples

Modul importieren

Verwenden Sie die `import` Anweisung:

```
>>> import random
>>> print(random.randint(1, 10))
4
```

`import module` importiert ein Modul und ermöglicht Ihnen dann, mit der Syntax `module.name` auf seine Objekte - beispielsweise Werte, Funktionen und Klassen - zu `module.name` . Im obigen Beispiel wird das `random` importiert, das die `randint` Funktion enthält. `randint` Sie `random` importieren, können Sie `randint` mit `random.randint` .

Sie können ein Modul importieren und einem anderen Namen zuweisen:

```
>>> import random as rn
>>> print(rn.randint(1, 10))
4
```

Wenn sich Ihre Python-Datei `main.py` im selben Ordner befindet wie `custom.py` . Sie können es so importieren:

```
import custom
```

Es ist auch möglich, eine Funktion aus einem Modul zu importieren:

```
>>> from math import sin
>>> sin(1)
0.8414709848078965
```

Um bestimmte Funktionen tiefer in ein Modul zu importieren, darf der Punktoperator **nur** auf der linken Seite des Schlüsselworts `import` :

```
from urllib.request import urlopen
```

In Python haben wir zwei Möglichkeiten, die Funktion von der obersten Ebene aufzurufen. Einer ist `import` und ein anderer kommt `from` . Wir sollten `import` wenn eine Namenskollision möglich ist. Nehmen wir an, wir haben `hello.py` Dateien und `world.py` Dateien mit derselben Funktion namens `function` . Dann funktioniert die `import` Anweisung gut.

```
from hello import function
from world import function

function() #world's function will be invoked. Not hello's
```

Im Allgemeinen stellt der `import` einen Namespace bereit.

```
import hello
import world

hello.function() # exclusively hello's function will be invoked
world.function() # exclusively world's function will be invoked
```

Wenn Sie sich jedoch sicher sind, gibt es in Ihrem gesamten Projekt keine Möglichkeit, denselben Funktionsnamen zu verwenden, den Sie `from` Anweisung verwenden sollten

Mehrere Importe können in derselben Zeile durchgeführt werden:

```
>>> # 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
```

Die oben gezeigten Schlüsselwörter und Syntax können auch in Kombinationen verwendet werden:

```
>>> 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
```

Bestimmte Namen aus einem Modul importieren

Anstatt das komplette Modul zu importieren, können Sie nur die angegebenen Namen importieren:

```
from random import randint # Syntax "from MODULENAME import NAME1[, NAME2[, ...]]"
print(randint(1, 10))      # Out: 5
```

`from random` ist erforderlich, da der Python-Interpreter wissen muss, von welcher Ressource er eine Funktion oder Klasse `import randint` und `import randint` gibt die Funktion oder Klasse selbst an.

Weiteres Beispiel (ähnlich wie oben):

```
from math import pi
print(pi)                  # Out: 3.14159265359
```

Das folgende Beispiel führt zu einem Fehler, da wir kein Modul importiert haben:

```
random.randrange(1, 10)    # works only if "import random" has been run before
```

Ausgänge:

```
NameError: name 'random' is not defined
```

Der Python-Interpreter versteht nicht, was Sie mit `random` meinen. Es muss deklariert werden, indem `import random` zum Beispiel hinzugefügt wird:

```
import random
random.randrange(1, 10)
```

Importieren aller Namen aus einem Modul

```
from module_name import *
```

zum Beispiel:

```
from math import *
sqrt(2)    # instead of math.sqrt(2)
ceil(2.7)  # instead of math.ceil(2.7)
```

Dadurch werden alle im `math` Modul definierten Namen in den globalen Namespace importiert, mit Ausnahme von Namen, die mit einem Unterstrich beginnen (was darauf hinweist, dass der Verfasser der Meinung ist, dass er nur zur internen Verwendung bestimmt ist).

Warnung : Wenn eine Funktion mit demselben Namen bereits definiert oder importiert wurde, wird sie **überschrieben** . Beim Importieren nur bestimmter Namen `from math import sqrt, ceil` fast immer **empfohlen** :

```
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
```

Markierte Importe sind nur auf Modulebene erlaubt. Versuche, sie in Klassen- oder Funktionsdefinitionen auszuführen, führen zu einem `SyntaxError` .

```
def f():
    from math import *
```

und

```
class A:
    from math import *
```

beide scheitern mit:

```
SyntaxError: import * only allowed at module level
```

Die spezielle Variable __all__

Module können über eine spezielle Variable mit dem Namen `__all__` zu beschränken, welche Variablen bei Verwendung `from mymodule import *` .

Gegeben das folgende Modul:

```
# mymodule.py

__all__ = ['imported_by_star']

imported_by_star = 42
not_imported_by_star = 21
```

Nur `imported_by_star` importiert wird bei der Verwendung `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
```

`not_imported_by_star` kann jedoch explizit importiert werden:

```
>>> from mymodule import not_imported_by_star
>>> not_imported_by_star
21
```

Programmatisches Importieren

Python 2.x 2.7

Verwenden `importlib` zum Importieren eines Moduls über einen Funktionsaufruf das Modul `importlib` (in Python ab Version 2.7 enthalten):

```
import importlib
random = importlib.import_module("random")
```

Die Funktion `importlib.import_module()` importiert auch das Submodul eines Pakets direkt:

```
collections_abc = importlib.import_module("collections.abc")
```

Verwenden Sie für ältere Python-Versionen das `imp` Modul.

Python 2.x 2.7

Verwenden Sie die Funktionen `imp.find_module` und `imp.load_module` , um einen programmgesteuerten Import durchzuführen.

Aus der Standard-Bibliotheksdokumentation entnommen

```
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()
```

Verwenden Sie **NICHT** `__import__()` , um Module programmgesteuert zu importieren! Es gibt subtile Details wie `sys.modules` , das `fromlist` Argument usw., die leicht zu übersehen sind, welche `importlib.import_module()` für Sie behandelt.

Importieren Sie Module von einem beliebigen Dateisystemort aus

Wenn Sie ein Modul importieren möchten, das noch nicht als integriertes Modul in der Python Standard Library oder als Side-Package vorhanden ist, können Sie dies tun, indem Sie den Pfad zu dem Verzeichnis hinzufügen, in dem sich Ihr Modul befindet `sys.path` . Dies kann nützlich sein, wenn auf einem Host mehrere Python-Umgebungen vorhanden sind.

```
import sys
sys.path.append("/path/to/directory/containing/your/module")
import mymodule
```

Es ist wichtig, dass Sie den Pfad an das *Verzeichnis* anhängen, in dem sich `mymodule` befindet, nicht den Pfad zum Modul selbst.

PEP8-Regeln für Importe

Einige empfohlene PEP8- Richtlinien für den Import:

1. Importe sollten in separaten Zeilen erfolgen:

   ```
   from math import sqrt, ceil      # Not recommended
   from math import sqrt            # Recommended
   from math import ceil
   ```

2. Bestellen Sie Importe wie folgt oben im Modul:

   - Standardbibliothekimporte
   - Verbundene Importe Dritter
   - Lokale anwendungs- / bibliotheksspezifische Importe

3. Platzhalterimporte sollten vermieden werden, da dies zu Verwirrungen bei Namen im aktuellen Namespace führt. Wenn Sie das `from module import *` , kann es unklar sein, ob ein bestimmter Name in Ihrem Code vom `module` stammt oder nicht. Dies trifft doppelt zu, wenn Sie mehrere `from module import *` Anweisungen `from module import *` Typ `from module import *` .

4. Verwenden Sie keine relativen Importe. Verwenden Sie stattdessen explizite Importe.

Submodule importieren

```
from module.submodule import function
```

Diese `function` importiert die `function` von `module.submodule` .

__import __ () - Funktion

Mit der Funktion `__import__()` können Module importiert werden, bei denen der Name nur zur Laufzeit bekannt ist

```
if user_input == "os":
```

```
    os = __import__("os")

# equivalent to import os
```

Diese Funktion kann auch verwendet werden, um den Dateipfad zu einem Modul anzugeben

```
mod = __import__(r"C:/path/to/file/anywhere/on/computer/module.py")
```

Ein Modul erneut importieren

Wenn Sie den interaktiven Interpreter verwenden, möchten Sie möglicherweise ein Modul erneut laden. Dies kann nützlich sein, wenn Sie ein Modul bearbeiten und die neueste Version importieren möchten oder wenn Sie ein Element eines vorhandenen Moduls mit einem Affen-Patch versehen haben und Ihre Änderungen rückgängig machen möchten.

Beachten Sie, dass Sie das Modul **nicht** einfach erneut `import` können, um es zurückzusetzen:

```
import math
math.pi = 3
print(math.pi)    # 3
import math
print(math.pi)    # 3
```

Dies liegt daran, dass der Interpreter jedes importierte Modul registriert. Wenn Sie versuchen, ein Modul erneut zu importieren, sieht der Interpreter dies im Register und tut nichts. Der schwerste Weg für den Reimport ist also der `import` nach dem Entfernen des entsprechenden Elements aus dem Register:

```
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
```

Aber es gibt mehr einen einfachen und einfachen Weg.

Python 2

Verwenden Sie die `reload` Funktion:

Python 2.x 2.3

```
import math
math.pi = 3
print(math.pi)    # 3
reload(math)
print(math.pi)    # 3.141592653589793
```

Python 3

Die `reload` Funktion wurde nach `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
```

Module importieren online lesen: https://riptutorial.com/de/python/topic/249/module-importieren

Kapitel 106: Müllsammlung

Bemerkungen

Im Kern ist Pythons Garbage-Collector (ab 3.5) eine einfache Implementierung der Referenzzählung. Bei jeder Referenz auf ein Objekt (z. B. `a = myobject`) wird der Referenzzähler für dieses Objekt (myobject) inkrementiert. Jedes Mal, wenn eine Referenz entfernt wird, wird der Referenzzähler dekrementiert, und sobald der Referenzzähler `0` erreicht, wissen wir, dass nichts einen Verweis auf dieses Objekt enthält, und wir können es aufheben.

Ein häufiges Missverständnis über die Funktionsweise der Python-Speicherverwaltung besteht darin, dass das Schlüsselwort `del` Objektspeicher freigibt. Das ist nicht wahr. Was tatsächlich passiert, ist, dass das Schlüsselwort `del` lediglich die refcount der Objekte dekrementiert, was bedeutet, dass, wenn Sie es so oft aufrufen, dass die refcount-Zahl null ist, das Objekt möglicherweise als Garbage-Collection erfasst wird (selbst wenn tatsächlich noch Verweise auf das an anderer Stelle im Code verfügbare Objekt vorhanden sind).

Python erstellt oder bereinigt Objekte aggressiv, wenn sie zum ersten Mal benötigt werden. Wenn ich die Zuweisung a = object () durchführe, wird der Speicher für das Objekt zu diesem Zeitpunkt zugewiesen (cpython verwendet manchmal bestimmte Objekttypen, z. B. Listen unter der Haube). Meistens ist jedoch kein freier Objektpool vorhanden, und die Zuordnung wird ausgeführt, wenn Sie sie benötigen. Sobald der refcount auf 0 dekrementiert wird, bereinigt der GC ihn.

Generationsmüllsammlung

In den 1960er Jahren entdeckte John McCarthy einen fatalen Fehler bei der erneuten Zählung der Müllsammlung, als er den von Lisp verwendeten Refcounting-Algorithmus implementierte: Was passiert, wenn sich zwei Objekte in einer zyklischen Referenz auf einander beziehen? Wie können Sie diese beiden Objekte müll sammeln, auch wenn es keine externen Verweise darauf gibt, wenn sie sich immer auf einander beziehen? Dieses Problem erstreckt sich auch auf jede zyklische Datenstruktur, beispielsweise auf Ringpuffer oder zwei aufeinanderfolgende Einträge in einer doppelt verknüpften Liste. Python versucht, dieses Problem zu beheben, indem er einen etwas anderen Garbage Collection-Algorithmus namens **Generational Garbage Collection verwendet** .

Wenn Sie also ein Objekt in Python erstellen, wird es am Ende einer doppelt verknüpften Liste hinzugefügt. Gelegentlich durchläuft Python diese Liste, prüft, auf welche Objekte sich auch die Objekte in der Liste beziehen, und wenn sie sich auch in der Liste befinden (wir werden sehen, warum sie sich nicht in einem Moment befinden), verringern sie ihre Refcounts weiter. An diesem Punkt (tatsächlich gibt es einige Heuristiken, die bestimmen, wann Dinge verschoben werden, aber nehmen wir an, dass es nach einer einzelnen Sammlung geht, um die Dinge einfach zu halten), wenn immer noch ein refcount von mehr als 0 vorhanden ist, wird in eine andere verknüpfte Liste mit dem Namen "Generation 1" befördert. (Deshalb sind nicht immer alle Objekte in der Generierungsliste 0), auf die diese Schleife seltener angewendet wird. Hier kommt die Generationsspeicherbereinigung ins Spiel. Standardmäßig gibt es in Python 3 Generationen (drei verknüpfte Objektlisten): Die erste Liste (Generation 0) enthält alle neuen Objekte. Wenn ein GC-

Zyklus auftritt und die Objekte nicht gesammelt werden, werden sie in die zweite Liste (Generation 1) verschoben. Wenn ein GC-Zyklus in der zweiten Liste ausgeführt wird und sie immer noch nicht gesammelt werden, werden sie in die dritte Liste (Generation 2) verschoben). Die dritte Generation-Liste ("Generation 2" genannt, da wir keine Indexierung durchführen) besteht aus weniger gesammelten Abfällen als den ersten beiden. Die Idee ist, dass, wenn Ihr Objekt langlebig ist, es nicht so wahrscheinlich ist, dass es gaschiert wird und möglicherweise niemals Sie sollten während der gesamten Lebensdauer Ihrer Anwendung eine GC-Analyse durchführen, sodass es nicht unnötig ist, Zeit für die Überprüfung der einzelnen GC-Laufzeiten zu verschwenden. Außerdem wird beobachtet, dass die meisten Objekte relativ schnell Müll gesammelt werden. Von nun an nennen wir diese "guten Objekte", seit sie jung sterben. Dies wird als "schwache Generationshypothese" bezeichnet und wurde erstmals in den 60er Jahren beobachtet.

Ein kurzer Hinweis: Im Gegensatz zu den ersten beiden Generationen besteht die Liste der langlebigen dritten Generation nicht aus regelmäßigem Müll. Es wird geprüft, wenn das Verhältnis von langlebigen anstehenden Objekten (diejenigen, die sich in der dritten Generierungsliste befinden, aber noch keinen GC-Zyklus hatten) zu den insgesamt in der Liste befindlichen langlebigen Objekten mehr als 25% beträgt. Dies liegt daran, dass die dritte Liste unbegrenzt ist (Dinge werden nie von einer Liste auf eine andere Liste verschoben, sodass sie nur dann verschwinden, wenn tatsächlich Müll gesammelt wird). Dies bedeutet, dass für Anwendungen, bei denen Sie viele langlebige Objekte erstellen, GC-Zyklen auf der dritten liste kann es recht lang werden. Durch die Verwendung eines Verhältnisses erzielen wir "amortisierte lineare Leistung in der Gesamtzahl der Objekte"; aka, je länger die Liste ist, desto länger dauert GC, aber desto seltener führen wir GC aus (hier ist der ursprüngliche Vorschlag von 2008 für diese Heuristik von Martin von Löwis für weiteres Lesen). Das Ausführen einer Speicherbereinigung für die dritte Generation oder "ausgereifte" Liste wird "vollständige Speicherbereinigung" genannt.

Die generationenbasierte Speicherbereinigung beschleunigt die Abläufe ungemein, da wir nicht nach Objekten suchen müssen, die wahrscheinlich nicht immer GC benötigen, aber wie hilft es uns, zyklische Referenzen zu brechen? Wahrscheinlich nicht sehr gut, stellt sich heraus. Die Funktion für die tatsächlich diese Referenzzyklen zu brechen beginnt wie folgt :

```
/* 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)
```

Der Grund, warum die Garbage Collection für Generationen dabei hilft, ist, dass wir die Länge der Liste als separate Zählung beibehalten können. Jedes Mal, wenn wir der Generation ein neues Objekt hinzufügen, erhöhen wir diese Zählung, und jedes Mal, wenn wir ein Objekt in eine andere Generation verschieben oder es freigeben, dekrementieren wir die Zählung. Theoretisch sollte diese Zählung am Ende eines GC-Zyklus (für die ersten beiden Generationen sowieso immer) immer 0 sein. Andernfalls ist alles, was in der Liste übrig ist, eine Art Zirkelreferenz, und wir können sie löschen. Es gibt jedoch noch ein weiteres Problem: Was ist, wenn die übrig gebliebenen Objekte Pythons magische Methode `__del__` ? `__del__` wird jedes Mal aufgerufen, wenn ein Python-Objekt zerstört wird. Wenn jedoch zwei Objekte in einem `__del__` über `__del__` Methoden verfügen, können wir nicht sicher sein, dass die Zerstörung eines `__del__` die anderen

__del__ __del__ Methoden nicht zerstört. Stellen Sie sich vor, wir hätten folgendes geschrieben:

```
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)
```

und wir setzen eine Instanz von A und eine Instanz von B so, dass sie aufeinander zeigen, und dann landen sie im selben Speicherbereinigungszyklus? Nehmen wir an, wir wählen uns zufällig aus und legen zuerst unsere Instanz von A ab. Die __del__ Methode von A wird aufgerufen, es wird gedruckt, dann wird A freigegeben. Als nächstes kommen wir zu B, wir nennen seine __del__ Methode und oops! Segfault! A existiert nicht mehr. Wir könnten dies beheben, indem wir zunächst alle __del__ Methoden aufrufen, die dann übrig __del__ , und dann einen weiteren Durchlauf durchführen, um wirklich alles zu löschen. Dies führt jedoch zu einem anderen Problem: Was __del__ wenn eine __del__ Methode eines Objekts einen Verweis auf das andere Objekt speichert, das gerade gecedet wird Hat uns irgendwo ein Hinweis auf uns? Wir haben immer noch einen Referenzzyklus, aber jetzt ist es nicht möglich, eines der beiden Objekte tatsächlich zu GC, auch wenn sie nicht mehr verwendet werden. Beachten Sie, dass ein Objekt selbst dann, wenn es nicht Teil einer kreisförmigen Datenstruktur ist, in seiner eigenen __del__ Methode __del__ werden kann. Python prüft dies und beendet die GC-Analyse, wenn die Anzahl der Objekte nach dem __del__ der __del__ Methode __del__ ist.

CPython beschäftigt sich damit, diese un-GC-fähigen Objekte (alles, was irgendeine Art von __del__ und eine __del__ Methode hat) auf eine globale Liste von uneinholbarem Müll zu __del__ und sie dann für alle Ewigkeit dort zu lassen:

```
/* list of uncollectable objects */
static PyObject *garbage = NULL;
```

Examples

Referenzzählung

Die überwiegende Mehrheit der Python-Speicherverwaltung wird mit Referenzzählung ausgeführt.

Jedes Mal, wenn ein Objekt referenziert wird (z. B. einer Variablen zugewiesen) wird der Referenzzähler automatisch erhöht. Wenn dereferenziert wird (z. B. die Variable verlässt den Gültigkeitsbereich), wird die Referenzanzahl automatisch verringert.

Wenn der Referenzzähler Null erreicht, wird das Objekt **sofort zerstört** und der Speicher wird sofort freigegeben. In der Mehrzahl der Fälle wird der Müllsammler also nicht benötigt.

```
>>> 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
```

Um das Konzept der Referenzen weiter zu demonstrieren:

```
>>> 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
```

Speicherbereiniger für Referenzzyklen

Der Garbage Collector wird nur benötigt, wenn Sie einen *Referenzzyklus haben* . Das einfache Beispiel eines Referenzzyklus ist einer, in dem A sich auf B bezieht und B sich auf A bezieht, während sich nichts anderes auf A oder B bezieht. Ihre Referenzzählung ist jedoch 1 und kann daher nicht allein durch den Referenzzählalgorithmus freigegeben werden.

```
>>> 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
```

Ein Referenzzyklus kann beliebig lang sein. Wenn A auf B zeigt, zeigt B auf C, zeigt auf ..., zeigt auf Z, was auf A zeigt, dann werden bis zur Speicherbereinigungsphase weder A bis Z gesammelt.

```
>>> 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
```

Auswirkungen des Befehls del

Entfernen eines Variablennamens aus dem Gültigkeitsbereich mit `del v` oder Entfernen eines Objekts aus einer Auflistung mit `del v[item]` oder `del[i:j]` oder Entfernen eines Attributs mit `del v.name` oder einer anderen Möglichkeit zum Entfernen von Verweisen auf ein Objekt, löst *keine* destructor Anrufe oder jeder Speicher in sich selbst befreit. Objekte werden nur zerstört, wenn ihre Referenzzählung null erreicht.

```
>>> 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
```

Wiederverwendung von primitiven Objekten

Eine interessante Sache, die zur Optimierung Ihrer Anwendungen beitragen kann, ist, dass Primitive auch unter der Haube neu gezählt werden. Schauen wir uns die Zahlen an. Für alle Ganzzahlen zwischen -5 und 256 verwendet Python immer dasselbe Objekt:

```
>>> import sys
>>> sys.getrefcount(1)
797
>>> a = 1
>>> b = 1
>>> sys.getrefcount(1)
799
```

Beachten Sie, dass der Refcount zunimmt, was bedeutet, dass `a` und `b` auf dasselbe zugrunde liegende Objekt verweisen, wenn sie sich auf das Primitiv `1` beziehen. Bei größeren Zahlen verwendet Python das zugrunde liegende Objekt jedoch nicht wieder:

```
>>> a = 999999999
>>> sys.getrefcount(999999999)
3
>>> b = 999999999
>>> sys.getrefcount(999999999)
3
```

Da sich der refcount für `999999999` nicht ändert, wenn er `a` und `b` zugewiesen wird, können wir daraus schließen, dass sie sich auf zwei verschiedene zugrunde liegende Objekte beziehen, obwohl ihnen beide das gleiche `999999999` zugewiesen ist.

Anzeige der Refcount eines Objekts

```
>>> import sys
>>> a = object()
>>> sys.getrefcount(a)
2
>>> b = a
>>> sys.getrefcount(a)
3
>>> del b
>>> sys.getrefcount(a)
2
```

Objekte zwangsweise freigeben

Sie können die Freigabe von Objekten erzwingen, auch wenn ihr Refcount in Python 2 und 3 nicht 0 ist.

Beide Versionen verwenden dazu das Modul `ctypes` .

WARNUNG: Wenn Sie dies tun, *wird* Ihre Python-Umgebung instabil und zum Absturz neigen, ohne dass ein Traceback erforderlich ist! Die Verwendung dieser Methode kann auch zu Sicherheitsproblemen führen (ziemlich unwahrscheinlich). Heben Sie nur Objekte auf, von denen Sie sicher sind, dass Sie sie nie wieder referenzieren. Je.

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
```

Nach dem Ausführen führt jeder Verweis auf das jetzt freigegebene Objekt dazu, dass Python entweder undefiniertes Verhalten erzeugt oder abstürzt - ohne Traceback. Es gab wahrscheinlich einen Grund, warum der Müllsammler das Objekt nicht entfernt hat ...

Wenn Sie `None` Zuordnung `None` , wird eine spezielle Meldung `Fatal Python error: deallocating None` - `Fatal Python error: deallocating None` vor dem Absturz freigeben.

Verwalten der Speicherbereinigung

Es gibt zwei Ansätze, um zu beeinflussen, wann eine Speicherbereinigung durchgeführt wird. Sie beeinflussen, wie oft der automatische Prozess durchgeführt wird und der andere Prozess manuell eine Bereinigung auslöst.

Der Speicherbereiniger kann manipuliert werden, indem die Sammelschwellen eingestellt werden, die die Frequenz beeinflussen, mit der der Kollektor abläuft. Python verwendet ein generationsbasiertes Speicherverwaltungssystem. Neue Objekte werden in der neuesten Generation - **Generation** 0 gespeichert, und mit jeder erhaltenen Sammlung werden Objekte zu älteren Generationen befördert. Nach Erreichen der letzten Generation - **Generation2** werden sie nicht mehr befördert.

Die Schwellenwerte können mit dem folgenden Snippet geändert werden:

```
import gc
gc.set_threshold(1000, 100, 10) # Values are just for demonstration purpose
```

Das erste Argument repräsentiert den Schwellenwert für das Sammeln der **Generierung0** . Jedes Mal, wenn die Anzahl der **Zuweisungen** die Anzahl der **Aufhebungen** um 1000 überschreitet, wird der Garbage Collection aufgerufen.

Die älteren Generationen werden nicht bei jedem Durchlauf bereinigt, um den Prozess zu optimieren. Das zweite und das dritte Argument sind **optional** und steuern, wie oft ältere

Generationen gereinigt werden. Wenn die **Generierung** 0 100 Mal ohne Bereinigung der **Generierung1 verarbeitet wurde** , wird die **Generierung1** verarbeitet. In ähnlicher Weise werden die Objekte in **generation2** nur verarbeitet werden , wenn die , die in **generation1** 10 mal gereinigt wurden ohne **generation2** zu berühren.

Eine manuelle Einstellung der Schwellenwerte ist von Vorteil, wenn das Programm viele kleine Objekte zuweist, ohne sie aufzuheben, was dazu führt, dass der Garbage Collector zu oft ausgeführt wird (jede Objektzuordnung der **Generation** 0_Threshold). Obwohl der Collector ziemlich schnell ist, stellt er bei einer großen Anzahl von Objekten ein Leistungsproblem dar. Wie auch immer, es gibt keine einheitliche Strategie für die Auswahl der Schwellenwerte und die Zuverlässigkeit des Anwendungsfalls.

Das manuelle Auslösen einer Sammlung kann wie im folgenden Snippet durchgeführt werden:

```
import gc
gc.collect()
```

Die Speicherbereinigung wird automatisch basierend auf der Anzahl der Zuweisungen und Aufhebungen ausgelöst, nicht auf dem verbrauchten oder verfügbaren Speicher. Wenn Sie mit großen Objekten arbeiten, kann der Speicher erschöpft sein, bevor die automatische Bereinigung ausgelöst wird. Dies ist ein guter Anwendungsfall für das manuelle Aufrufen des Speicherbereinigers.

Obwohl es möglich ist, ist es keine ermutigte Praxis. Die Vermeidung von Speicherverlusten ist die beste Option. In großen Projekten kann das Erkennen des Speicherverlusts jedoch eine schwierige Aufgabe sein, und das manuelle Auslösen einer Speicherbereinigung kann als schnelle Lösung bis zum weiteren Debugging verwendet werden.

Bei lang laufenden Programmen kann die Speicherbereinigung nach Zeit oder Ereignis ausgelöst werden. Ein Beispiel für den ersten ist ein Webserver, der nach einer festgelegten Anzahl von Anforderungen eine Sammlung auslöst. Für letzteren einen Webserver, der eine Garbage Collection auslöst, wenn ein bestimmter Anforderungstyp empfangen wird.

Warten Sie nicht, bis die Müllsammlung aufgeräumt ist

Die Tatsache, dass die Speicherbereinigung aufgeräumt wird, bedeutet nicht, dass Sie warten müssen, bis der Speicherbereinigungszyklus aufgeräumt ist.

Insbesondere sollten Sie nicht warten, bis die Garbage Collection Dateihandles, Datenbankverbindungen und Netzwerkverbindungen geschlossen hat.

zum Beispiel:

Im folgenden Code wird davon ausgegangen, dass die Datei beim nächsten Garbage Collection-Zyklus geschlossen wird, wenn f der letzte Verweis auf die Datei war.

```
>>> f = open("test.txt")
>>> del f
```

Eine explizitere Methode zum Bereinigen ist das Aufrufen von `f.close()` . Sie können es noch eleganter machen, indem Sie die `with` Anweisung verwenden, die auch als Kontextmanager bezeichnet wird :

```
>>> with open("test.txt") as f:
...     pass
...     # do something with f
>>> #now the f object still exists, but it is closed
```

Mit der `with` Anweisung können Sie Ihren Code unter der geöffneten Datei einrücken. Dies macht es explizit und einfacher zu sehen, wie lange eine Datei geöffnet bleibt. Es schließt auch immer eine Datei, auch wenn im `while` Block eine Ausnahme ausgelöst wird.

Müllsammlung online lesen: https://riptutorial.com/de/python/topic/2532/mullsammlung

Kapitel 107: Multiprocessing

Examples

Zwei einfache Prozesse ausführen

Ein einfaches Beispiel für die Verwendung mehrerer Prozesse sind zwei Prozesse (Worker), die separat ausgeführt werden. Im folgenden Beispiel werden zwei Prozesse gestartet:

- `countUp()` zählt jede Sekunde 1 aufwärts.
- `countDown()` zählt jede Sekunde 1 `countDown()` .

```
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()
```

Die Ausgabe lautet wie folgt:

```
Up:    0
Down:    3
Up:    1
Up:    2
Down:    2
Up:    3
Down:    1
Down:    0
```

Pool und Karte verwenden

```
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` ist eine Klasse, die mehrere `Workers` (Prozesse) hinter den Kulissen verwaltet und Sie als Programmierer verwenden kann.

`Pool(5)` erstellt einen neuen Pool mit 5 Prozessen, und `pool.map` funktioniert wie Map , verwendet jedoch mehrere Prozesse (die beim Erstellen des Pools definierte Menge).

Ähnliche Ergebnisse können mit `map_async` , `apply` und `apply_async` erzielt werden, `apply` in der Dokumentation zu finden sind .

Multiprocessing online lesen: https://riptutorial.com/de/python/topic/3601/multiprocessing

Kapitel 108: Multithreading

Einführung

Mit Threads können Python-Programme mehrere Funktionen gleichzeitig ausführen, anstatt eine Befehlsfolge einzeln auszuführen. In diesem Thema werden die Prinzipien für das Threading erläutert und seine Verwendung veranschaulicht.

Examples

Grundlagen des Multithreading

Mit dem `threading` Modul kann ein neuer Ausführungsthread gestartet werden, indem ein neues `threading.Thread` und ihm eine Funktion zur Ausführung zugewiesen wird:

```
import threading

def foo():
  print "Hello threading!"

my_thread = threading.Thread(target=foo)
```

Der `target` verweist auf die Funktion (oder aufrufbare Objekt) ausgeführt werden. Der Thread beginnt erst mit der Ausführung, wenn `start` für das `Thread` Objekt aufgerufen wird.

Einen Thread starten

```
my_thread.start() # prints 'Hello threading!'
```

`my_thread` nun ausgeführt und beendet wurde, wird beim `RuntimeError` Aufruf von `start` ein `RuntimeError` . Wenn Sie Ihren Thread als Daemon ausführen möchten, den `daemon=True` kwarg übergeben oder `my_thread.daemon` vor dem Aufruf von `start()` auf `True` , wird Ihr `Thread` als Daemon im Hintergrund ausgeführt.

Einen Thread verbinden

Wenn Sie einen großen Job in mehrere kleine aufteilen und gleichzeitig ausführen möchten, aber warten müssen, bis alle abgeschlossen sind, bevor Sie fortfahren, ist `Thread.join()` die Methode, nach der Sie suchen.

Angenommen, Sie möchten mehrere Seiten einer Website herunterladen und diese auf einer einzigen Seite zusammenstellen. Sie würden das tun:

```
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)
```

Einen genaueren Einblick in die Funktionsweise von `join()` finden Sie hier .

Erstellen Sie eine benutzerdefinierte Thread-Klasse

Mit der `threading.Thread` Klasse können wir eine neue benutzerdefinierte Thread-Klasse unterteilen. Wir müssen die `run` Methode in einer Unterklasse überschreiben.

```
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.")
```

Kommunikation zwischen Threads

Ihr Code enthält mehrere Threads, und Sie müssen sicher zwischen ihnen kommunizieren.

Sie können eine `Queue` aus der `queue` verwenden.

```
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()
```

Producer- und Consumer-Threads mit einer gemeinsam genutzten Warteschlange erstellen

```
q = Queue()
t1 = Thread(target=consumer, args=(q,))
t2 = Thread(target=producer, args=(q,))
t1.start()
t2.start()
```

Anlegen eines Worker-Pools

`threading` und `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)
```

`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. Auflage, von David Beazley und Brian K. Jones (O'Reilly). Copyright 2013 David Beazley und Brian Jones, 978-1-449-34037-7.

Erweiterte Verwendung von Multithreads

Dieser Abschnitt enthält einige der fortschrittlichsten Beispiele, die mit Multithreading realisiert wurden.

Erweiterter Drucker (Logger)

Ein Thread, der alles druckt, wird empfangen und ändert die Ausgabe entsprechend der Klemmenbreite. Das Schöne daran ist, dass auch die "bereits geschriebene" Ausgabe geändert wird, wenn sich die Breite des Terminals ändert.

```
#!/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)
```

Stoppbarer Thread mit einer while-Schleife

```
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!"
```

Basierend auf dieser Frage .

Multithreading online lesen: https://riptutorial.com/de/python/topic/544/multithreading

Kapitel 109: Mutable vs. Immutable (und Hashable) in Python

Examples

Veränderlich vs. unveränderlich

Es gibt zwei Arten von Typen in Python. Unveränderliche Typen und veränderliche Typen.

Unveränderliche

Ein Objekt eines unveränderlichen Typs kann nicht geändert werden. Bei jedem Versuch, das Objekt zu ändern, wird eine Kopie erstellt.

Diese Kategorie umfasst: Ganzzahlen, Floats, Komplexe, Zeichenfolgen, Bytes, Tupel, Bereiche und Frozensets.

Um diese Eigenschaft hervorzuheben, spielen wir mit der eingebauten `id` . Diese Funktion gibt die eindeutige Kennung des als Parameter übergebenen Objekts zurück. Wenn die ID gleich ist, ist dies dasselbe Objekt. Wenn es sich ändert, ist dies ein anderes Objekt. *(Einige sagen, dass dies tatsächlich die Speicheradresse des Objekts ist, aber hüten Sie sich vor ihnen, sie stammen von der dunklen Seite der Kraft ...)*

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

Okay, 1 ist nicht 3 ... Aktuelle Nachrichten ... Vielleicht nicht. Dieses Verhalten wird jedoch häufig vergessen, wenn es sich um komplexere Typen handelt, insbesondere um Strings.

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

Aha! Sehen? Wir können es ändern!

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

Nein . Auch wenn es scheint , können wir die Zeichenfolge durch die Variable mit dem Namen ändern `stack` , was wir tatsächlich tun, ist die Schaffung eines neuen Objekts das Ergebnis der Verkettung enthalten. Wir werden getäuscht, weil das alte Objekt dabei nirgendwohin geht und somit zerstört wird. In einer anderen Situation wäre das offensichtlicher gewesen:

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

In diesem Fall ist es klar, dass wir eine Kopie benötigen, wenn wir die erste Zeichenfolge beibehalten möchten. Aber ist das für andere Typen so offensichtlich?

Übung

Nun, wenn Sie wissen, wie unveränderliche Typen funktionieren, was würden Sie mit dem folgenden Code sagen? Ist es weise

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

Mutables

Ein Objekt eines wandelbaren Typ kann geändert werden, und es wird *in situ* verändert. Es werden keine impliziten Kopien erstellt.

Diese Kategorie umfasst: Listen, Wörterbücher, Bytearrays und Sets.

Lass uns weiter mit unserer kleinen `id` Funktion spielen.

```
>>> 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
```

(Als Randbemerkung verwende ich Bytes, die ASCII-Daten enthalten, um meinen Standpunkt

klarer zu machen, bedenke jedoch, dass Bytes nicht für Textdaten ausgelegt sind.

Was haben wir? Wir erstellen ein Bytearray, ändern es und verwenden die `id` . Wir können sicherstellen, dass es sich um dasselbe Objekt handelt, das geändert wurde. Keine Kopie davon.

Wenn ein Objekt häufig geändert wird, ist ein veränderlicher Typ natürlich viel besser als ein unveränderlicher Typ. Leider wird die Realität dieser Immobilie oft vergessen, wenn sie am meisten schmerzt.

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

Okay...

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

Waiiit eine Sekunde ...

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

Tatsächlich. `c` ist keine Kopie von `b` . `c` ist `b` .

Übung

Nun verstehen Sie besser, welche Nebenwirkung ein veränderlicher Typ impliziert. Können Sie erklären, was in diesem Beispiel schief läuft?

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

Veränderlich und unveränderlich als Argumente

Ein Hauptanwendungsfall, in dem ein Entwickler die Veränderlichkeit berücksichtigen muss, ist das Übergeben von Argumenten an eine Funktion. Dies ist sehr wichtig, da dies die Fähigkeit der Funktion zum Ändern von Objekten bestimmt, die nicht zu ihrem Gültigkeitsbereich gehören, oder mit anderen Worten, wenn die Funktion Nebeneffekte hat. Dies ist auch wichtig, um zu verstehen, wo das Ergebnis einer Funktion verfügbar gemacht werden muss.

```
>>> 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]
```

Hier besteht der Fehler darin zu glauben, dass `lin` als Parameter der Funktion lokal geändert werden kann. Stattdessen referenzieren `lin` und `a` und dasselbe Objekt. Da dieses Objekt veränderbar ist, wird die Änderung an Ort und Stelle vorgenommen, was bedeutet, dass das von `lin` und `a` referenzierte Objekt geändert wird. `lin` braucht nicht wirklich zurück, weil wir in Form eines Verweises auf dieses Objekt haben bereits `a` . `a` und `b` enden auf dasselbe Objekt.

Dies gilt nicht für Tupel.

```
>>> 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)
```

Zu Beginn der Funktion können `tin` und `a` Objekt auf dasselbe Objekt verweisen. Dies ist jedoch ein unveränderliches Objekt. Also , wenn die Funktion , es zu ändern versucht, `tin` ein neues Objekt mit der Modifikation erhalten, während `a` einen Verweis auf das ursprüngliche Objekt hält. In diesem Fall ist die Rücksendung von `tin` obligatorisch, oder das neue Objekt würde verloren gehen.

Übung

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

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

Hinweis: Der `reverse` arbeitet direkt.

Was denkst du über diese Funktion? Hat es Nebenwirkungen? Ist die Rückgabe notwendig? Was ist es wert, nach dem Anruf zu `saying` ? Von `focused` ? Was passiert, wenn die Funktion mit den gleichen Parametern erneut aufgerufen wird?

Mutable vs. Immutable (und Hashable) in Python online lesen:

https://riptutorial.com/de/python/topic/9182/mutable-vs--immutable--und-hashable--in-python

Kapitel 110: Neo4j und Cypher mit Py2Neo

Examples

Importieren und Authentifizieren

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

Sie müssen sicherstellen, dass Ihre Neo4j-Datenbank unter localhost: 7474 mit den entsprechenden Anmeldeinformationen vorhanden ist.

Das `graph` ist Ihre Schnittstelle zur neo4j-Instanz im Rest Ihres Python-Codes. Danken Sie es nicht, diese Variable zu einer globalen Variablen zu `__init__` .

Knoten zum Neo4j-Diagramm hinzufügen

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

Das Hinzufügen von Knoten zum Diagramm ist ziemlich einfach. `graph.merge_one` ist wichtig, da doppelte Elemente `graph.merge_one` werden. (Wenn Sie das Skript zweimal ausführen, wird der Titel beim zweiten Mal aktualisiert, und es werden keine neuen Knoten für dieselben Artikel erstellt.)

`timestamp` sollte eine Ganzzahl und keine Datumszeichenfolge sein, da neo4j nicht wirklich einen Datumsdatentyp hat. Dies führt zu Sortierproblemen, wenn Sie das Datum als '05 -06-1989 'speichern.

`article.push()` ist der Aufruf, der die Operation tatsächlich in neo4j festschreibt. Vergiss diesen Schritt nicht.

Hinzufügen von Beziehungen zum Neo4j-Diagramm

```
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` ist wichtig, um Duplikate zu vermeiden. Aber sonst ist es eine ziemlich unkomplizierte Operation. Der Beziehungsname ist auch wichtig, da Sie ihn in fortgeschrittenen Fällen verwenden würden.

Abfrage 1: Autovervollständigung bei Nachrichtentiteln

```
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
```

Dies ist ein Beispiel Chiffre Abfrage alle Knoten mit der Eigenschaft zu erhalten `name` , die mit dem Argument beginnt `text` .

Abfrage 2: Abrufen von Nachrichtenartikeln an einem bestimmten Datum nach Standort

```
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
```

Sie können diese Abfrage verwenden, um alle Nachrichtenartikel `(n)` zu finden, die durch eine Beziehung mit einem Ort `(l)` .

Cypher Query Samples

Zählen Sie Artikel, die im Laufe der Zeit mit einer bestimmten Person verbunden sind

```
MATCH (n)-[]->(l)
where l.name='Donald Trump'
RETURN n.date,count(*) order by n.date
```

Suchen Sie nach anderen Personen / Orten, die mit den gleichen Nachrichtenartikeln wie Trump

verbunden sind, mit insgesamt mindestens fünf Beziehungsknoten.

```
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
```

Neo4j und Cypher mit Py2Neo online lesen: https://riptutorial.com/de/python/topic/5841/neo4j-und-cypher-mit-py2neo

Kapitel 111: Nicht offizielle Python-Implementierungen

Examples

IronPython

Open Source-Implementierung für .NET und Mono, geschrieben in C #, lizenziert unter Apache License 2.0. Es basiert auf DLR (Dynamic Language Runtime). Es unterstützt nur Version 2.7, Version 3 wird gerade entwickelt.

Unterschiede zu CPython:

- Enge Integration mit .NET Framework.
- Zeichenfolgen sind standardmäßig Unicode.
- Unterstützt keine Erweiterungen für CPython in C.
- Leidet nicht unter Global Interpreter Lock.
- Die Leistung ist normalerweise geringer, hängt jedoch von Tests ab.

Hallo Welt

```
print "Hello World!"
```

Sie können auch .NET-Funktionen verwenden:

```
import clr
from System import Console
Console.WriteLine("Hello World!")
```

Externe Links

- Offizielle Website
- GitHub-Repository

Jython

Open Source-Implementierung für JVM, geschrieben in Java, lizenziert unter Python Software Foundation License. Es unterstützt nur Version 2.7, Version 3 wird gerade entwickelt.

Unterschiede zu CPython:

- Enge Integration mit JVM.

- Zeichenfolgen sind Unicode.
- Unterstützt keine Erweiterungen für CPython in C.
- Leidet nicht unter Global Interpreter Lock.
- Die Leistung ist normalerweise geringer, hängt jedoch von Tests ab.

Hallo Welt

```
print "Hello World!"
```

Sie können auch Java-Funktionen verwenden:

```
from java.lang import System
System.out.println("Hello World!")
```

Externe Links

- Offizielle Website
- Mercurial Repository

Verschlüsseln

Transcrypt ist ein Werkzeug, um eine relativ umfangreiche Python-Teilmenge in kompaktes, lesbares Javascript vorzukompilieren. Es hat die folgenden Eigenschaften:

- Ermöglicht die klassische OO-Programmierung mit mehrfacher Vererbung unter Verwendung der reinen Python-Syntax, die von CPythons nativem Parser analysiert wird
- Nahtlose Integration mit dem Universum von qualitativ hochwertigen, weborientierten JavaScript-Bibliotheken anstelle der desktoporientierten Python-Bibliotheken
- Hierarchisches URL-basiertes Modulsystem, das die Modulverteilung über PyPi ermöglicht
- Einfache Beziehung zwischen Python-Quellcode und generiertem JavaScript-Code zum einfachen Debuggen
- Mehrstufige Quellcaps und optionale Annotation des Zielcodes mit Quellreferenzen
- Kompakte Downloads, kBs statt MBs
- Optimierter JavaScript-Code mit Memoization (Call-Caching), um optional die Prototyp-Lookup-Kette zu umgehen
- Die Überladung des Bedieners kann lokal ein- und ausgeschaltet werden, um die lesbare numerische Mathematik zu erleichtern

Codegröße und Geschwindigkeit

Die Erfahrung hat gezeigt, dass 650 kB Python-Quellcode in etwa die gleiche Menge an JavaScript-Quellcode übersetzt. Die Geschwindigkeit entspricht der Geschwindigkeit von handgeschriebenem JavaScript und kann diese Geschwindigkeit übertreffen, wenn die

Anrufaufzeichnung aktiviert ist.

Integration mit 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>
```

Integration mit JavaScript und 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 ()
```

Integration mit anderen JavaScript-Bibliotheken

Transcrypt kann in Kombination mit jeder JavaScript-Bibliothek ohne besondere Maßnahmen oder Syntax verwendet werden. In der Dokumentation finden Sie Beispiele für ao react.js, riot.js, fabric.js und node.js.

Beziehung zwischen Python- und JavaScript-Code

Python

```
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');
```

Externe Links

- Offizielle Website: http://www.transcrypt.org/
- Repository: https://github.com/JdeH/Transcrypt

Nicht offizielle Python-Implementierungen online lesen:
https://riptutorial.com/de/python/topic/5225/nicht-offizielle-python-implementierungen

Kapitel 112: Optische Zeichenerkennung

Einführung

Bei der optischen Zeichenerkennung werden Textbilder in tatsächlichen Text umgewandelt. In diesen Beispielen finden Sie Möglichkeiten zur Verwendung von OCR in Python.

Examples

PyTesseract

PyTesseract ist ein in der Entwicklung befindliches Python-Paket für OCR.

PyTesseract zu benutzen ist ziemlich einfach:

```
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 ist Open Source und kann hier gefunden werden .

PyOCR

Ein anderes Modul ist `PyOCR` , dessen Quellcode hier ist .

Auch einfach zu bedienen und bietet mehr Funktionen als `PyTesseract` .

Zu initialisieren:

```
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.
```

Und einige Anwendungsbeispiele:

```
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
```

Optische Zeichenerkennung online lesen: https://riptutorial.com/de/python/topic/9302/optische-zeichenerkennung

Kapitel 113: os.path

Einführung

Dieses Modul implementiert einige nützliche Funktionen für Pfadnamen. Die Pfadparameter können als Zeichenfolgen oder Bytes übergeben werden. Anwendungen sollten Dateinamen als (Unicode) Zeichenketten darstellen.

Syntax

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

Examples

Pfade beitreten

Um zwei oder mehr Pfadkomponenten miteinander zu verbinden, importieren Sie zuerst das os-Modul von python und verwenden Sie dann Folgendes:

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

Der Vorteil der Verwendung von os.path ist, dass der Code über alle Betriebssysteme hinweg kompatibel bleibt, da hierbei das für die Plattform geeignete Trennzeichen verwendet wird.

Das Ergebnis dieses Befehls unter Windows lautet beispielsweise:

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

In einem Unix-Betriebssystem:

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

Absoluter Pfad vom relativen Pfad

Verwenden Sie `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'
```

Pfadkomponenten-Manipulation

So trennen Sie eine Komponente vom Pfad ab:

```
>>> 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')
```

Holen Sie sich das übergeordnete Verzeichnis

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

Wenn der angegebene Pfad existiert

um zu überprüfen, ob der angegebene Pfad existiert

```
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
```

Prüfen Sie, ob der angegebene Pfad ein Verzeichnis, eine Datei, ein symbolischer Link, ein Einhängepunkt usw. ist.

um zu überprüfen, ob der angegebene Pfad ein Verzeichnis ist

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

um zu überprüfen, ob der angegebene Pfad eine Datei ist

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

um zu überprüfen, ob der angegebene Pfad ein symbolischer Link ist

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

um zu überprüfen, ob der angegebene Pfad ein Einhängepunkt ist

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

os.path online lesen: https://riptutorial.com/de/python/topic/1380/os-path

Kapitel 114: Pandas-Transformation: Vorformung von Operationen in Gruppen und Verketten der Ergebnisse

Examples

Einfache umwandlung

Zuerst lassen wir einen Dummy-Datenrahmen erstellen

Wir gehen davon aus, dass ein Kunde n Bestellungen haben kann, eine Bestellung m Artikel haben kann und Artikel mehrmals bestellt werden können

```
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
```

.
.

Jetzt werden wir die Pandas- `transform` verwenden, um die Anzahl der Bestellungen pro Kunde zu zählen

```
# 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
```

Mehrere Ergebnisse pro Gruppe

Verwenden von `transform` , die Teilberechnungen pro Gruppe zurückgeben

Im vorherigen Beispiel hatten wir ein Ergebnis pro Client. Es können jedoch auch Funktionen angewendet werden, die unterschiedliche Werte für die Gruppe zurückgeben.

```
# 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
```

Pandas-Transformation: Vorformung von Operationen in Gruppen und Verketten der Ergebnisse online lesen: https://riptutorial.com/de/python/topic/10947/pandas-transformation--vorformung-von-operationen-in-gruppen-und-verketten-der-ergebnisse

Kapitel 115: Parallele Berechnung

Bemerkungen

Aufgrund der GIL (Global Interpretersperre) wird nur eine Instanz des Python-Interpreters in einem einzigen Prozess ausgeführt. Im Allgemeinen verbessert die Verwendung von Multithreading nur IO-gebundene Berechnungen, nicht CPU-gebundene. Das `multiprocessing` Modul wird empfohlen, wenn Sie CPU-gebundene Aufgaben parallelisieren möchten.

GIL gilt für CPython, die populärste Implementierung von Python, sowie für PyPy. Andere Implementierungen wie Jython und IronPython haben keine GIL .

Examples

Verwenden des Multiprocessing-Moduls zum Parallelisieren von Aufgaben

```
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]
```

Da die Ausführung jedes Aufrufs von `fib` parallel ausgeführt wird, ist die Ausführungszeit des vollständigen Beispiels um das **1,8-** fache **schneller** als bei sequentieller Ausführung auf einem Dualprozessor.

Python 2.2+

Übergeordnete und untergeordnete Skripts verwenden, um Code parallel auszuführen

Kind.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()
```

Dies ist nützlich für parallele, unabhängige HTTP-Anforderungs- / Antworttasks oder Datenbankauswahl / -inserts. Befehlszeilenargumente können auch an das **child.py-** Skript übergeben werden. Die Synchronisation zwischen den Skripts kann erreicht werden, indem alle Skripts regelmäßig einen separaten Server (wie eine Redis-Instanz) überprüfen.

Verwendung einer C-Erweiterung zum Parallelisieren von Aufgaben

Die Idee dabei ist, die rechenintensiven Jobs (mit speziellen Makros) unabhängig von Python nach C zu verschieben und den GIL während der Arbeit mit dem C-Code freizugeben.

```
#include "Python.h"
...
PyObject *pyfunc(PyObject *self, PyObject *args) {
    ...
    Py_BEGIN_ALLOW_THREADS
    // Threaded C code
    ...
    Py_END_ALLOW_THREADS
    ...
}
```

Verwenden des PyPar-Moduls zum Parallelisieren

PyPar ist eine Bibliothek, die die Message Passing-Schnittstelle (MPI) verwendet, um Parallelität in Python bereitzustellen. Ein einfaches Beispiel in PyPar (wie auf https://github.com/daleroberts/pypar zu sehen) sieht folgendermaßen aus:

```
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()
```

Parallele Berechnung online lesen: https://riptutorial.com/de/python/topic/542/parallele-berechnung

Kapitel 116: pip: PyPI-Paketmanager

Einführung

pip ist der am häufigsten verwendete Paketmanager für den Python Package Index und wird standardmäßig mit den neuesten Versionen von Python installiert.

Syntax

- pip <Befehl> [Optionen] wobei <Befehl> einer der folgenden Werte ist:
  - Installieren
    - Pakete installieren
  - deinstallieren
    - Pakete deinstallieren
  - einfrieren
    - Installierte Pakete im Anforderungsformat ausgeben
  - Liste
    - Installierte Pakete auflisten
  - Show
    - Informationen zu installierten Paketen anzeigen
  - Suche
    - Suchen Sie bei PyPI nach Paketen
  - Rad
    - Bauen Sie Räder nach Ihren Anforderungen
  - Postleitzahl
    - Einzelne Pakete komprimieren (veraltet)
  - entpacken
    - Einzelne Pakete entpacken (veraltet)
  - bündeln
    - Pybundles erstellen (veraltet)
  - Hilfe
    - Hilfe für Befehle anzeigen

Bemerkungen

Manchmal führt `pip` eine manuelle Erstellung von nativem Code durch. Unter Linux wählt python automatisch einen verfügbaren C-Compiler auf Ihrem System. In der folgenden Tabelle finden Sie die erforderliche Visual Studio / Visual C ++ - Version unter Windows (neuere Versionen funktionieren nicht.)

Python-Version	Visual Studio-Version	Visual C ++ - Version
2,6 - 3,2	Visual Studio 2008	Visual C ++ 9.0

Python-Version	Visual Studio-Version	Visual C ++ - Version
3.3 - 3.4	Visual Studio 2010	Visual C ++ 10.0
3,5	Visual Studio 2015	Visual C ++ 14.0

Quelle: wiki.python.org

Examples

Pakete installieren

So installieren Sie die neueste Version eines Pakets mit dem Namen `SomePackage` :

```
$ pip install SomePackage
```

So installieren Sie eine bestimmte Version eines Pakets:

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

So legen Sie eine Mindestversion für die Installation eines Pakets fest:

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

Wenn in Befehlen unter Linux / Unix die Berechtigung verweigert wird, verwenden Sie `sudo` mit den Befehlen

---

Installieren Sie aus Anforderungsdateien

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

Jede Zeile der Anforderungsdatei gibt an, dass etwas installiert werden muss, und wie bei Argumenten für die Pip-Installation. Details zum Format der Dateien finden Sie hier: Anforderungsdateiformat .

Nach der Installation des Pakets können Sie es mit dem `freeze` Befehl überprüfen:

```
$ pip freeze
```

Pakete deinstallieren

So deinstallieren Sie ein Paket:

```
$ pip uninstall SomePackage
```

Um alle Pakete aufzulisten, die mit `pip` installiert wurden

So listen Sie installierte Pakete auf:

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

Um veraltete Pakete aufzulisten und die neueste verfügbare Version anzuzeigen:

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

Aktualisieren Sie Pakete

Laufen

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

aktualisiert das Paket `SomePackage` und alle seine Abhängigkeiten. Außerdem entfernt pip vor dem Upgrade automatisch eine ältere Version des Pakets.

Um ein Upgrade von pip selbst durchzuführen, tun Sie dies

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

auf Unix oder

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

auf Windows-Maschinen.

Aktualisierung aller veralteten Pakete unter Linux

`pip` enthält nicht aktuell ein Flag, mit dem ein Benutzer alle veralteten Pakete auf einmal aktualisieren kann. Dies kann jedoch erreicht werden, indem Befehle in einer Linux-Umgebung zusammengeführt werden:

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

Dieser Befehl nimmt alle Pakete in der lokalen Virtualenv und prüft, ob sie veraltet sind. Aus dieser Liste erhält er den Paketnamen und leitet diesen an einen Befehl `pip install -U` . Am Ende dieses Prozesses sollten alle lokalen Pakete aktualisiert werden.

Aktualisierung aller veralteten Pakete unter Windows

`pip` enthält nicht aktuell ein Flag, mit dem ein Benutzer alle veralteten Pakete auf einmal aktualisieren kann. Dies kann jedoch erreicht werden, indem Befehle in einer Windows-Umgebung zusammengeführt werden:

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

Dieser Befehl nimmt alle Pakete in der lokalen Virtualenv und prüft, ob sie veraltet sind. Aus dieser Liste erhält er den Paketnamen und leitet diesen an einen Befehl `pip install -U` . Am Ende dieses Prozesses sollten alle lokalen Pakete aktualisiert werden.

Erstellen Sie eine Requirements.txt-Datei aller Pakete im System

`pip` unterstützt Sie bei der Erstellung von `requirements.txt` Dateien, indem Sie die Option `freeze` angeben.

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

Dadurch wird eine Liste aller auf dem System installierten Pakete und ihrer Version in einer Datei mit dem Namen `requirements.txt` im aktuellen Ordner gespeichert.

Erstellen Sie eine Requirements.txt-Datei mit Paketen nur in der aktuellen virtualenv

`pip` unterstützt Sie bei der Erstellung von `requirements.txt` Dateien, indem Sie die Option `freeze` angeben.

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

Der Parameter `--local` gibt nur eine Liste von Paketen und Versionen aus, die lokal in einer virtuellen Umgebung installiert sind. Globale Pakete werden nicht aufgelistet.

Verwenden einer bestimmten Python-Version mit Pip

Wenn Sie sowohl Python 3 als auch Python 2 installiert haben, können Sie angeben, welche Python-Version Sie pip verwenden möchten. Dies ist nützlich, wenn Pakete nur Python 2 oder 3 unterstützen oder wenn Sie mit beiden testen möchten.

Wenn Sie Pakete für Python 2 installieren möchten, führen Sie Folgendes aus:

```
pip install [package]
```

oder:

```
pip2 install [package]
```

Wenn Sie Pakete für Python 3 installieren möchten, gehen Sie wie folgt vor:

```
pip3 install [package]
```

Sie können auch die Installation eines Pakets für eine bestimmte Python-Installation aufrufen mit:

```
\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
```

Auf OS-X / Linux / Unix-Plattformen ist es wichtig, sich über die Unterscheidung zwischen der Systemversion von Python (die Aktualisierung macht Ihr System funktionsunfähig) und der Benutzerversion (en) von Python zu unterscheiden. Sie **können,** *je nachdem* , *welche Sie aktualisieren wollen,* müssen Sie diese Befehle mit Präfix `sudo` und ein Kennwort.

Ebenso können unter Windows einige Python-Installationen, insbesondere solche, die Teil eines anderen Pakets sind, in Systemverzeichnissen installiert werden - diese müssen Sie von einem Befehlsfenster aus im Admin-Modus aus aktualisieren - falls Sie das Gefühl haben, dass dies erforderlich ist Dies ist eine **sehr** gute Idee, um zu überprüfen, welche Python-Installation Sie mit einem Befehl wie `python -c"import sys;print(sys.path);"` oder `py -3.5 -c"import sys;print(sys.path);"` Sie können auch überprüfen, welchen Pip Sie mit `pip --version` ausführen `pip --version`

Wenn Sie unter Windows sowohl Python 2 als auch Python 3 installiert haben und Ihr Pfad und Ihr Python 3 größer als 3,4 sind, haben Sie wahrscheinlich auch den Python Launcher `py` auf Ihrem Systempfad. Sie können dann Tricks machen wie:

```
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
```

Wenn Sie mehrere Versionen von Python `virtualenv` und `venv` empfiehlt es sich dringend, sich über die virtuellen Umgebungen von Python `virtualenv` oder `venv` denen Sie sowohl die Version von Python als auch die vorhandenen Pakete isolieren können.

Pakete installieren, die noch nicht auf dem Rohr als Räder montiert sind

Viele reine Python-Pakete sind im Python-Paketindex noch nicht als Laufräder verfügbar, können aber dennoch problemlos installiert werden. Einige Pakete unter Windows geben jedoch den Fehler "dreaded vcvarsall.bat nicht gefunden" aus.

Das Problem ist, dass das Paket, das Sie installieren *möchten* , eine C- oder C ++ - Erweiterung enthält und *derzeit* nicht als vordefiniertes Rad aus dem Python- *Paketindex pypi* und unter Windows *zur* Verfügung *steht* , über die Sie nicht die zum Erstellen erforderliche Werkzeugkette haben solche Artikel.

Die einfachste Antwort lautet, auf die exzellente Site von Christoph Gohlke zu gehen und die

entsprechende Version der benötigten Bibliotheken zu finden. Durch die **Angabe** des Paketnamens muss ein **-cp *NN* -** mit Ihrer Python-Version übereinstimmen, dh, wenn Sie Windows-32-Bit-Python verwenden, *auch unter win64,* muss der Name **-win32- enthalten,** und wenn Sie den 64-Bit-Python verwenden, muss er **-win_amd64 enthalten -** und dann muss die Python-Version übereinstimmen, dh für Python 34 **muss** der Dateiname **-cp *34-*** usw. enthalten. Dies ist im Grunde die Magie, die pip auf der pypi-Site für Sie tut.

Alternativ benötigen Sie das entsprechende Windows-Entwicklungskit für die verwendete Python-Version, die Header aller Bibliotheken, zu denen das Paket, mit dem Sie versuchen, Schnittstellen zu erstellen, möglicherweise die Python-Header für die Python-Version usw. erstellt werden.

Python 2.7 verwendete Visual Studio 2008, Python 3.3 und 3.4 verwendeten Visual Studio 2010 und Python 3.5+ verwendete Visual Studio 2015.

- Installieren Sie das " Visual C ++ Compiler Package for Python 2.7 ", das auf der Website von Microsoft **oder unter** http://www.support.microsoft.com/ zur Verfügung steht
- Installieren Sie „ Windows SDK für Windows 7 und .NET Framework 4 “ (v7.1), das auf der Website von Microsoft **oder** verfügbar ist
- Installieren Sie die Visual Studio 2015 Community Edition *(oder eine spätere Version, wenn diese veröffentlicht wird)* und stellen **Sie sicher** , **dass Sie die Optionen für die Installation der C & C ++ - Unterstützung** *nicht mehr als Standard festlegen. Es wird gesagt, dass das* Herunterladen und Installieren *bis zu **8 Stunden** dauern kann* Stellen Sie daher **sicher,** dass diese Optionen beim ersten Versuch eingestellt sind.

Dann müssen Sie *möglicherweise* die Header-Dateien *in der entsprechenden Revision* für alle Bibliotheken suchen, *auf die* Ihr gewünschtes Paket verweist, und diese an geeignete Speicherorte herunterladen.

Endlich können Sie pip erstellen lassen - natürlich, wenn das Paket Abhängigkeiten hat, die Sie noch nicht haben, müssen Sie möglicherweise auch die Header-Dateien für sie finden.

Alternativen: Es ist auch *empfehlenswert* , *sowohl auf der Website von pypi als auch auf Christops Seite* nach einer etwas früheren Version des Pakets zu suchen, nach der Sie suchen. Es handelt sich dabei entweder um reinen Python oder für Ihre Plattform und Python-Version vorgefertigte Versionen gefunden, bis Ihr Paket verfügbar ist. Wenn Sie die neueste Version von Python verwenden, werden Sie möglicherweise feststellen, dass die Paketbetreuer etwas Zeit **benötigen** , um auf den neuesten Stand zu kommen. Für Projekte, die wirklich ein bestimmtes Paket **benötigen** , **müssen** Sie im Moment einen etwas älteren Python verwenden. Sie können auch auf der Quellwebsite des Pakets nachsehen, ob eine gegabelte Version vorgefertigt oder als reiner Python verfügbar ist und nach alternativen Paketen sucht, die die erforderliche Funktionalität bereitstellen, aber verfügbar sind - ein Beispiel, das Ihnen in den Sinn kommt Kissen , *aktiv gewartet* , fallen als Ersatz für PIL *derzeit nicht in 6 Jahren aktualisiert und sind nicht für Python 3 verfügbar* .

Danach würde ich jeden, der dieses Problem hat, dazu ermutigen, zum Bug-Tracker für das Paket zu gehen und ein Ticket hinzuzufügen oder, falls noch nicht vorhanden, **höflich** aufzufordern, dass die Paketbetreuer ein Rad auf pypi für Ihr spezielles Fahrzeug zur Verfügung stellen Kombination von Plattform und Python. Wenn dies geschehen ist, werden die Dinge

normalerweise mit der Zeit besser. Einige Paketbetreuer wissen nicht, dass sie eine bestimmte Kombination verpasst haben, die die Leute verwenden.

Hinweis zum Installieren von Pre-Releases

Pip folgt den Regeln von Semantic Versioning und bevorzugt freigegebene Pakete den Vorabversionen. Wenn ein bestimmtes Paket als `V0.98` und es auch einen Veröffentlichungskandidaten `V1.0-rc1` das Standardverhalten von `pip install` die Installation von `V0.98` Wenn Sie den Veröffentlichungskandidaten installieren möchten, werden *Sie empfohlen* Um *zuerst in einer virtuellen Umgebung testen zu* können, können Sie dies mit `--pip install --pre` *Paketname* oder `--pip install --pre --upgrade` *Paketname* `--pip install --pre --upgrade` . In vielen Fällen verfügen Pre-Releases oder Release-Kandidaten möglicherweise nicht über Räder, die für alle Plattform- und Versionskombinationen geeignet sind, sodass die oben genannten Probleme wahrscheinlicher sind.

Hinweis zum Installieren von Entwicklungsversionen

Sie können pip auch verwenden, um Entwicklungsversionen von Paketen von Github und anderen Standorten zu installieren. Da ein solcher Code im Fluss ist, ist es sehr unwahrscheinlich, dass Räder für ihn erstellt werden. Für unreine Pakete sind daher die Build-Tools erforderlich jederzeit gebrochen sein, so dass der Benutzer **dringend** aufgefordert wird, solche Pakete nur in einer virtuellen Umgebung zu installieren.

Für solche Installationen gibt es drei Optionen:

1. Laden Sie die komprimierte Momentaufnahme herunter, die meisten Online-Versionskontrollsysteme haben die Möglichkeit, eine komprimierte Momentaufnahme des Codes herunterzuladen. Dieser kann manuell heruntergeladen und dann mit dem `pip install` */ to / downloads / file* `pip install` *werden.* Beachten Sie, dass pip für die meisten Komprimierungsformate das Entpacken in einen Cache-Bereich usw. übernimmt.
2. Überlassen Sie pip das Herunterladen und Installieren für Sie mit: `pip install` *URL / von / package / repository* - möglicherweise müssen Sie auch die `--trusted-host` , `--client-cert` und / oder `--proxy` richtig, vor allem in einer Unternehmensumgebung. z.B:

```
    > 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
```

Beachten Sie das `git+` -Präfix der URL.

3. Klonen Sie das Repository mit `git` , `mercurial` oder anderen akzeptablen Werkzeug, *vorzugsweise ein DVCS - Tool,* und verwenden Sie `pip install` *path / to / geklonten / repo* - Dies wird **sowohl** Prozess jede requires.text Datei und führen Sie die Build- und Konfigurationsschritte *können Sie manuell ändern Verzeichnis zu Ihrem geklonten Repository und führen Sie* `pip install -r requires.txt` *und dann* `python setup.py install` , *um denselben Effekt zu erzielen.* Der große Vorteil dieses Ansatzes besteht darin, dass der anfängliche Klonvorgang möglicherweise länger dauert als der Download des Snapshots. Sie können jedoch mit git: `git pull origin master` auf den neuesten Stand bringen, und wenn die aktuelle Version Fehler enthält, können Sie die `pip uninstall` *Paketname* Verwenden Sie

dann `git checkout` Befehle, um durch den Repository-Verlauf zu früheren Versionen zu gelangen und es erneut zu versuchen.

pip: PyPI-Paketmanager online lesen: https://riptutorial.com/de/python/topic/1781/pip--pypi-paketmanager

Kapitel 117: Plotten mit Matplotlib

Einführung

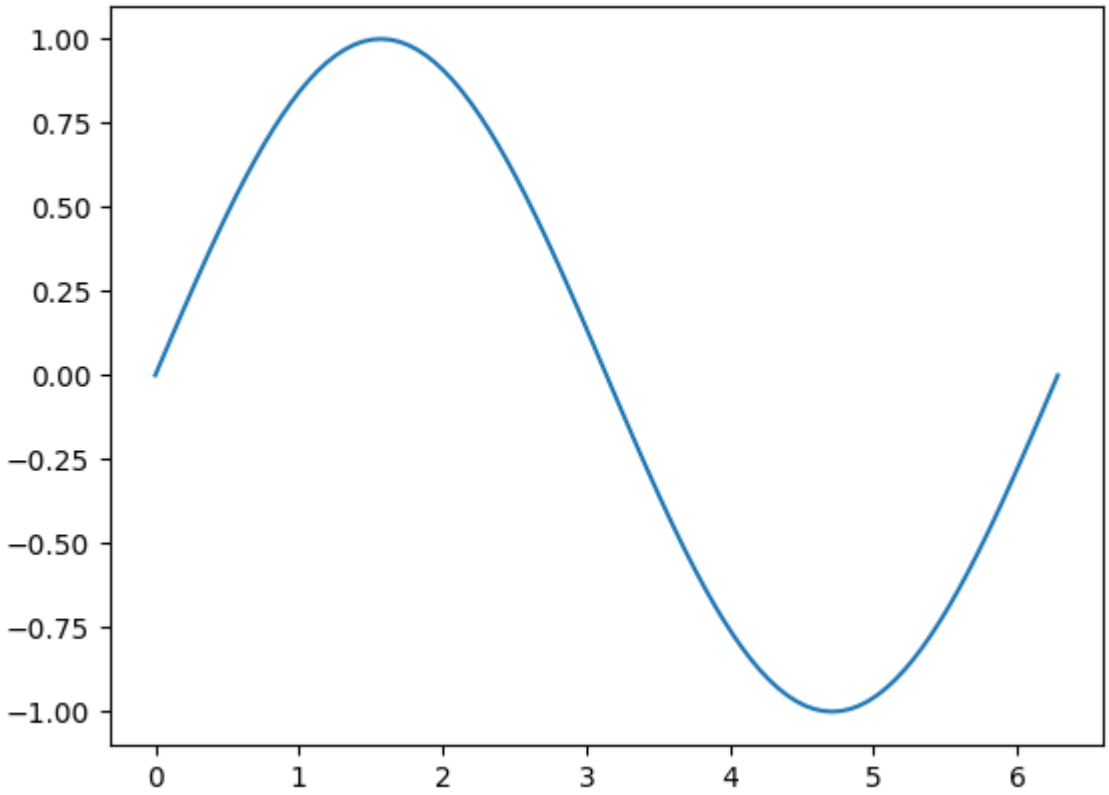
Matplotlib (https://matplotlib.org/) ist eine Bibliothek für das 2D-Zeichnen basierend auf NumPy. Hier einige grundlegende Beispiele. Weitere Beispiele finden Sie in der offiziellen Dokumentation (https://matplotlib.org/2.0.2/gallery.html und https://matplotlib.org/2.0.2/examples/index.html) sowie in http: //www.riptutorial.com/topic/881

Examples

Ein einfaches Diagramm in Matplotlib

Dieses Beispiel zeigt, wie Sie mit **Matplotlib** eine einfache Sinuskurve erstellen

```
# 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()
```

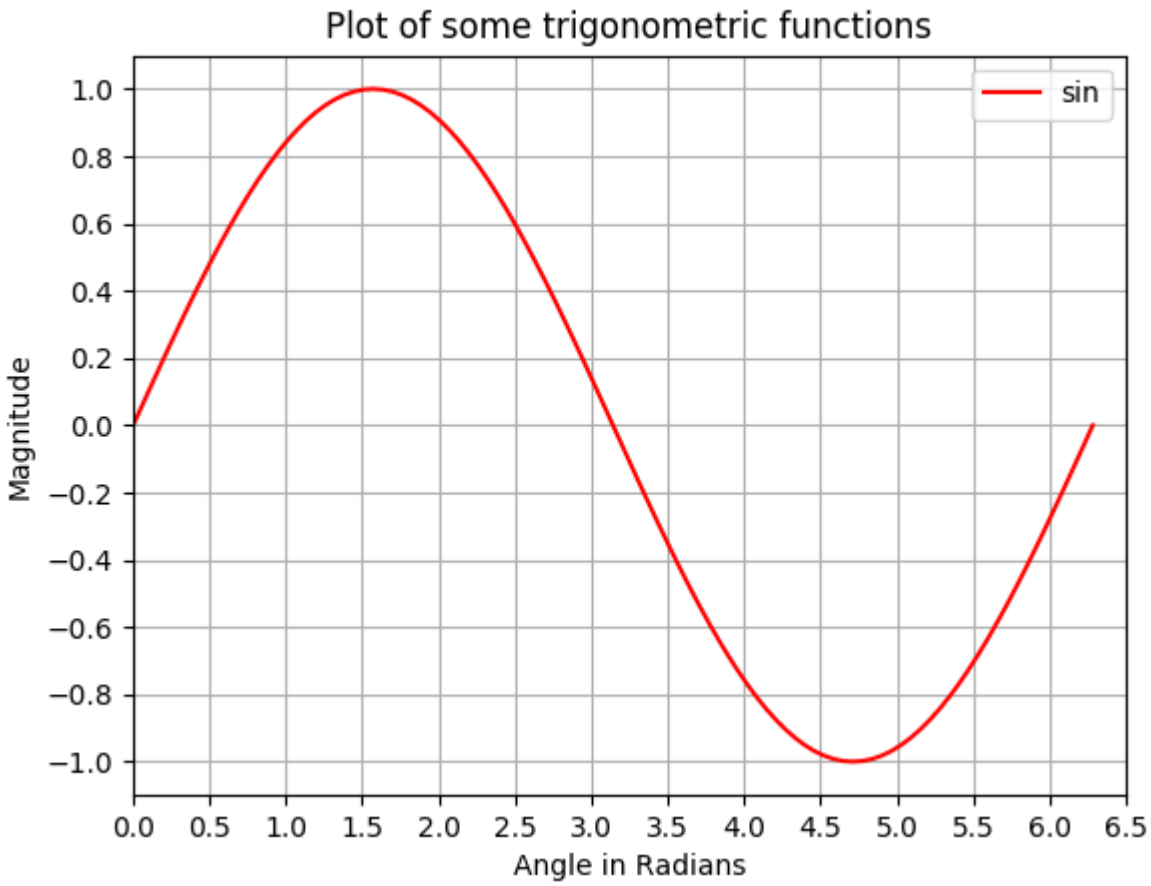


Hinzufügen weiterer Funktionen zu einer einfachen Zeichnung: Achsenbeschriftungen, Titel, Achsentexte, Raster und Legende

In diesem Beispiel nehmen wir ein Sinuskurvendiagramm und fügen weitere Features hinzu. nämlich Titel, Achsenbeschriftungen, Titel, Achsentexte, Raster und Legende.

```
# 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()
```

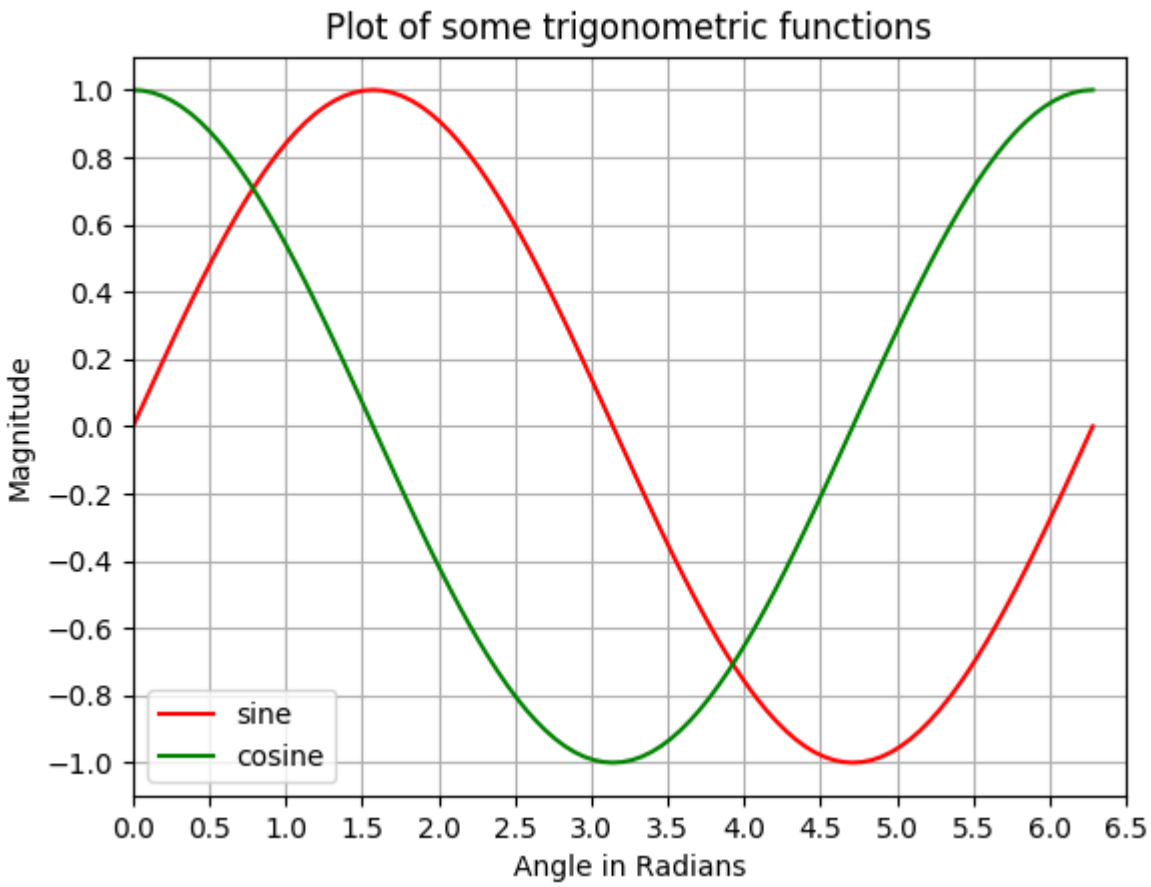


Erstellen mehrerer Plots in derselben Figur durch Überlagerung ähnlich wie bei MATLAB

In diesem Beispiel werden eine Sinuskurve und eine Cosinuskurve in derselben Figur durch Überlagern der Plots dargestellt.

```
# 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()
```



Erstellen mehrerer Plots in derselben Figur mithilfe der Plotüberlagerung mit separaten Plotbefehlen

Ähnlich wie im vorherigen Beispiel werden hier eine Sinus- und eine Cosinus-Kurve in derselben Figur mit separaten Plotbefehlen dargestellt. Dies ist mehr Pythonic und kann verwendet werden, um separate Griffe für jede Zeichnung zu erhalten.

```
# 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()
```

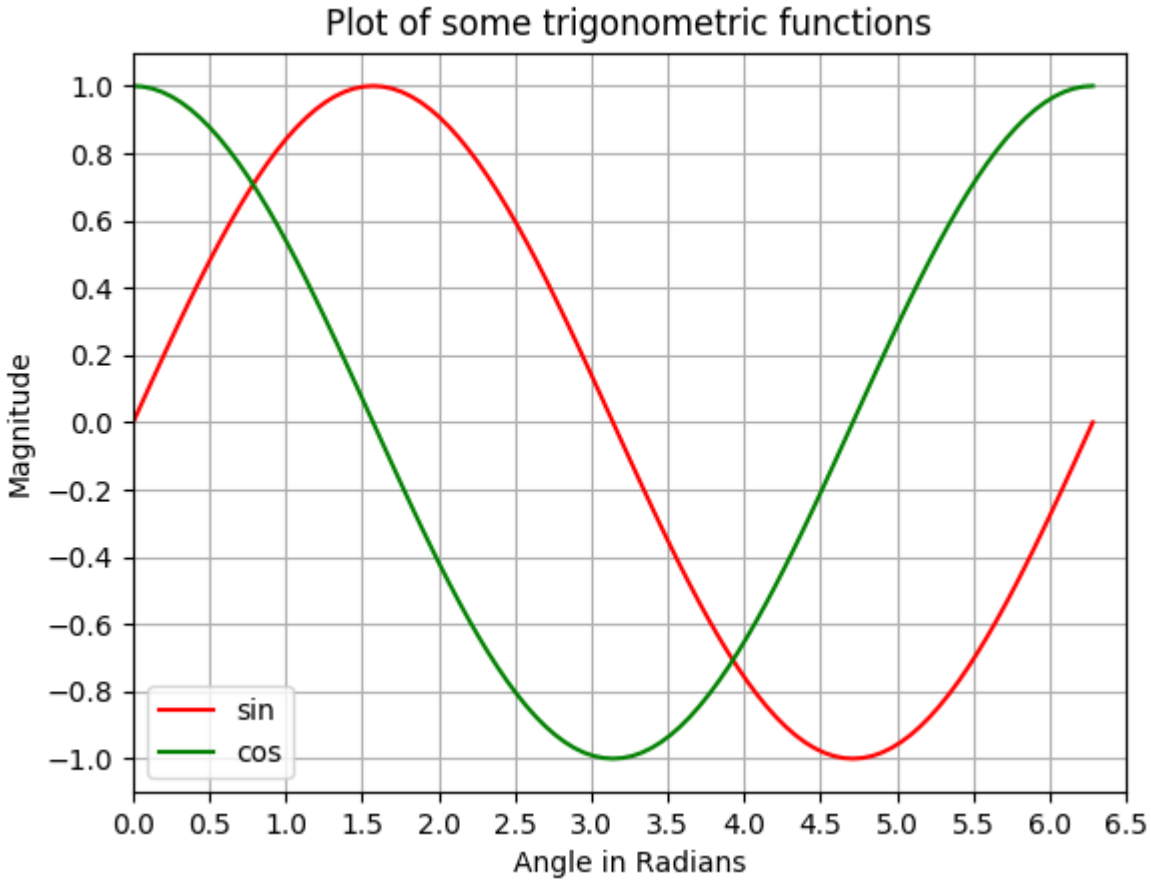


Diagramme mit gemeinsamer X-Achse, aber unterschiedlicher Y-Achse: Verwendung von twinx ()

In diesem Beispiel werden eine Sinuskurve und eine hyperbolische Sinuskurve in derselben Grafik mit einer gemeinsamen X-Achse mit verschiedenen Y-Achsen dargestellt. Dies wird durch Verwendung des **Befehls twinx () erreicht** .

```
# 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()
```

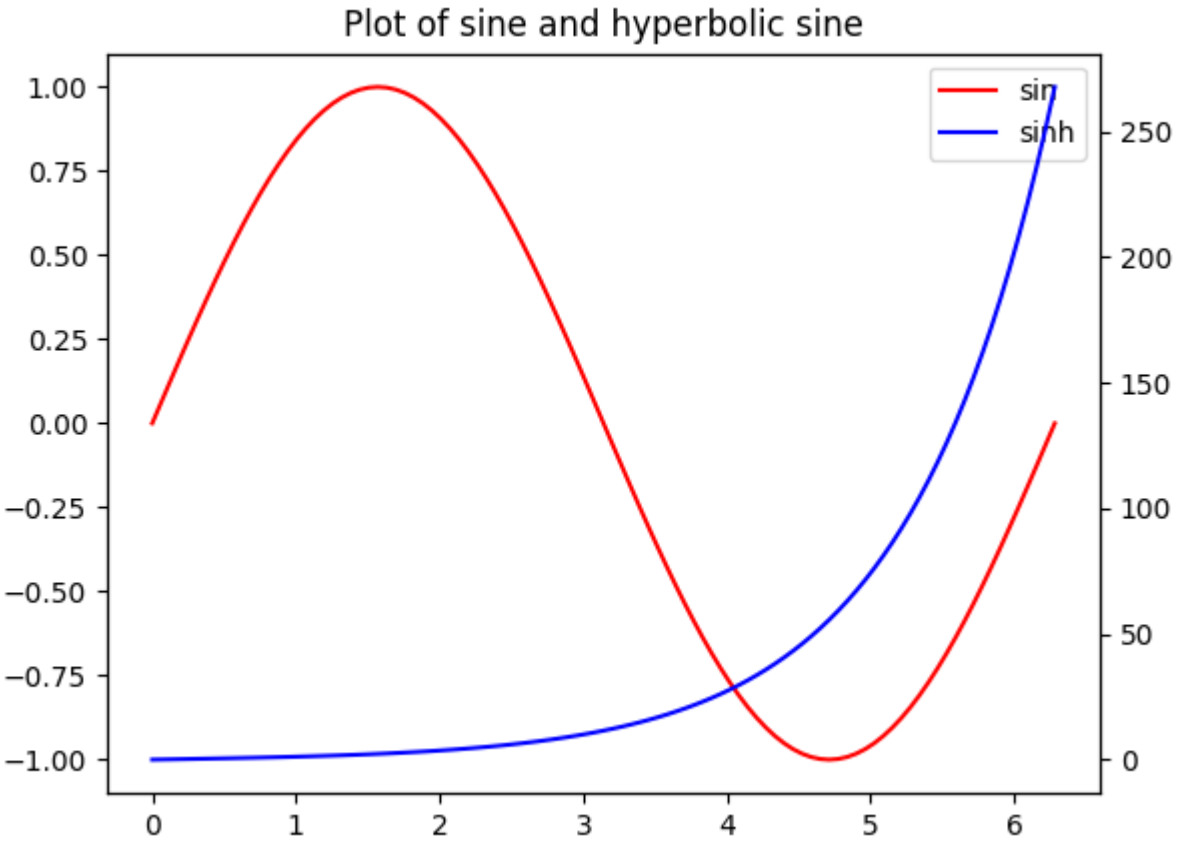
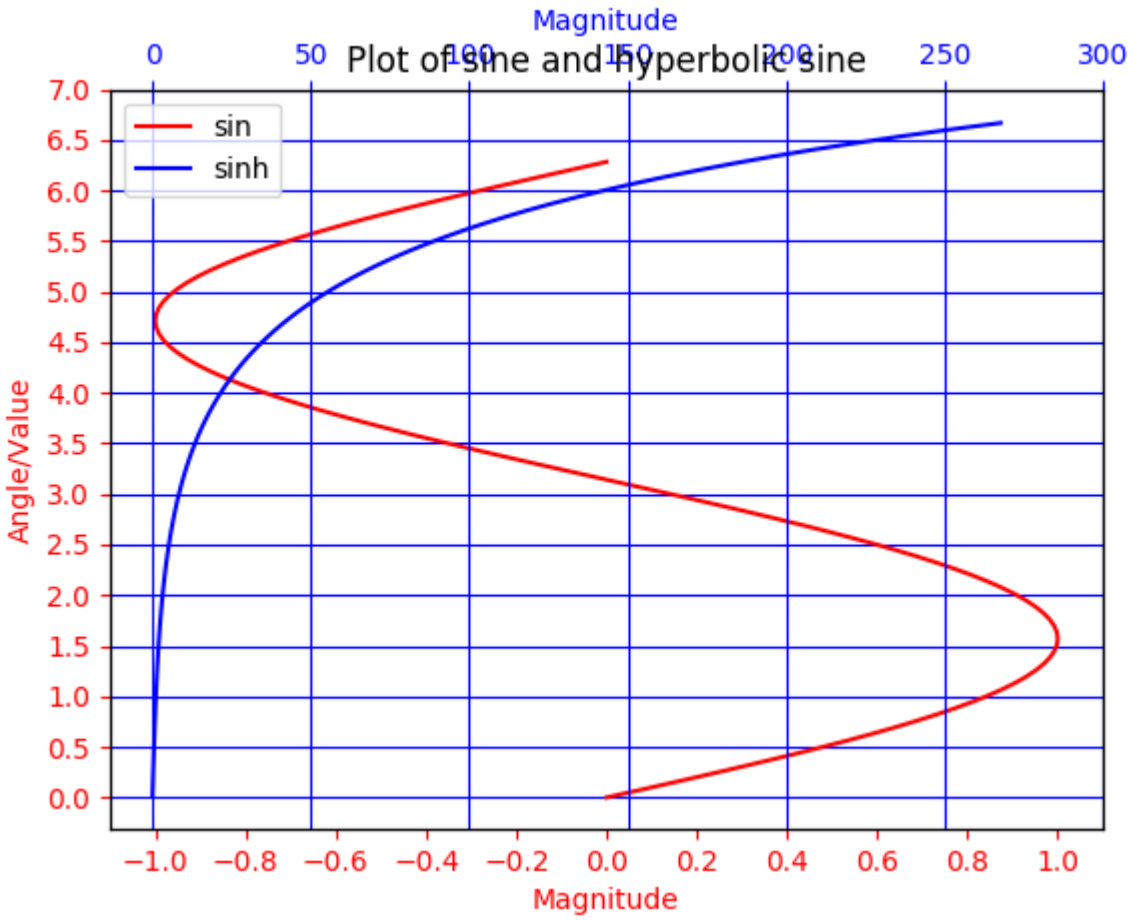


Diagramme mit gemeinsamer Y-Achse und unterschiedlicher X-Achse mit twiny ()

In diesem Beispiel wird ein Diagramm mit Kurven mit gemeinsamer y-Achse, aber unterschiedlicher x-Achse mit der **twiny () -** Methode **veranschaulicht** . Außerdem werden der Grafik einige zusätzliche Funktionen wie Titel, Legende, Beschriftungen, Raster, Achsentexte und Farben hinzugefügt.

```
# 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()
```



Plotten mit Matplotlib online lesen: https://riptutorial.com/de/python/topic/10264/plotten-mit-matplotlib

Kapitel 118: Plugin- und Erweiterungsklassen

Examples

Mixins

In der objektorientierten Programmiersprache ist ein Mixin eine Klasse, die Methoden enthält, die von anderen Klassen verwendet werden können, ohne die Elternklasse dieser anderen Klassen zu sein. Wie diese anderen Klassen Zugang zu den Methoden des Mixins erhalten, hängt von der Sprache ab.

Es bietet einen Mechanismus für die Mehrfachvererbung, indem mehrere Klassen die gemeinsame Funktionalität verwenden können, jedoch ohne die komplexe Semantik der Mehrfachvererbung. Mixins sind nützlich, wenn ein Programmierer Funktionalität zwischen verschiedenen Klassen teilen möchte. Anstatt den gleichen Code immer und immer wieder zu wiederholen, kann die allgemeine Funktionalität einfach in einem Mixin zusammengefasst und dann in jede Klasse vererbt werden, die dies erfordert.

Wenn wir mehr als ein Mixin verwenden, ist die Reihenfolge der Mixins wichtig. Hier ist ein einfaches Beispiel:

```
class Mixin1(object):
    def test(self):
        print "Mixin1"

class Mixin2(object):
    def test(self):
        print "Mixin2"

class MyClass(Mixin1, Mixin2):
    pass
```

In diesem Beispiel rufen wir `MyClass` und `test` .

```
>>> obj = MyClass()
>>> obj.test()
Mixin1
```

Ergebnis muss Mixin1 sein, da Order von links nach rechts ist. Dies kann zu unerwarteten Ergebnissen führen, wenn Superklassen hinzugefügt werden. Die umgekehrte Reihenfolge ist also besser so:

```
class MyClass(Mixin2, Mixin1):
    pass
```

Ergebnis wird sein:

```
>>> obj = MyClass()
```

```
>>> obj.test()
Mixin2
```

Mixins können verwendet werden, um benutzerdefinierte Plugins zu definieren.

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.
```

Plugins mit benutzerdefinierten Klassen

In Python 3.6 fügte PEP 487 die spezielle Methode `__init_subclass__` hinzu, mit der die Klassenanpassung ohne Verwendung von Metaklassen vereinfacht und erweitert werden kann. Daher können mit dieser Funktion einfache Plugins erstellt werden . Hier demonstrieren wir diese Funktion durch Modifizieren eines früheren Beispiels :

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.")
```

Ergebnisse:

```
PluginA().test()
# Base.
# Plugin A.

PluginB().test()
# Base.
# Plugin B.

Base.plugins
# [__main__.PluginA, __main__.PluginB]
```

Plugin- und Erweiterungsklassen online lesen: https://riptutorial.com/de/python/topic/4724/plugin--und-erweiterungsklassen

Kapitel 119: Polymorphismus

Examples

Grundpolymorphismus

Polymorphismus ist die Fähigkeit, eine Aktion an einem Objekt unabhängig von seinem Typ auszuführen. Dies wird im Allgemeinen durch Erstellen einer Basisklasse und durch zwei oder mehr Unterklassen implementiert, die alle Methoden mit derselben Signatur implementieren. Jede andere Funktion oder Methode, die diese Objekte bearbeitet, kann dieselben Methoden aufrufen, unabhängig davon, auf welchen Objekttyp sie angewendet wird, ohne dass zuvor eine Typprüfung durchgeführt werden muss. Wenn in der objektorientierten Terminologie die Klasse X die Klasse Y erweitert, wird Y als Superklasse oder Basisklasse und X als Unterklasse oder abgeleitete Klasse bezeichnet.

```
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)
```

Wir sollten diese Ausgabe sehen:

Keiner
4
6,0

Was passiert ohne Polymorphismus?
Ohne Polymorphismus kann eine Typüberprüfung erforderlich sein, bevor eine Aktion für ein Objekt ausgeführt wird, um die richtige aufzurufende Methode zu bestimmen. Das folgende **Counter-Beispiel** führt dieselbe Aufgabe wie der vorherige Code aus, aber ohne Verwendung von Polymorphismus muss die Funktion `get_area()` mehr Arbeit verrichten.

```
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)
```

Wir sollten diese Ausgabe sehen:

4
6,0

Wichtige Notiz
Beachten Sie, dass die im Counter-Beispiel verwendeten Klassen "new style" -Klassen sind und von der Objektklasse implizit erben, wenn Python 3 verwendet wird. Der Polymorphismus funktioniert sowohl in Python 2.x als auch in 3.x, aber der Code zum Beispiel Polymorphismus-Countdown löst eine Ausnahme aus, wenn er in einem Python 2.x-Interpreter ausgeführt wird, weil type (input_obj) **name** gibt statt des Klassennamens "Instanz" zurück, wenn sie nicht explizit von einem Objekt erben, wodurch der Bereich niemals zugewiesen wird.

Ente tippen

Polymorphismus ohne Vererbung in Form von Enten-Typisierung, wie er in Python aufgrund seines dynamischen Typisierungssystems verfügbar ist. Das heißt, solange die Klassen dieselben Methoden enthalten, unterscheidet der Python-Interpreter nicht zwischen ihnen, da die einzige Prüfung der Aufrufe zur Laufzeit erfolgt.

```
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)
```

Die Ausgabe ist:

> Quaaaaaack!
> Die Ente hat weiße und graue Federn.
> Die Person ahmt eine Ente nach.
> Die Person nimmt eine Feder vom Boden und zeigt sie.

Polymorphismus online lesen: https://riptutorial.com/de/python/topic/5100/polymorphismus

Kapitel 120: PostgreSQL

Examples

Fertig machen

PostgreSQL ist eine aktiv entwickelte und ausgereifte Open-Source-Datenbank. Mit dem Modul `psycopg2` können wir Abfragen in der Datenbank ausführen.

Installation mit Pip

```
pip install psycopg2
```

Grundlegende Verwendung

`my_table` wir an, wir haben eine Tabelle `my_table` in der Datenbank `my_database` die wie folgt definiert ist.

Ich würde	Vorname	Nachname
1	John	Damhirschkuh

Wir können das Modul `psycopg2` , um Abfragen in der Datenbank auf folgende Weise auszuführen.

```
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')]
```

PostgreSQL online lesen: https://riptutorial.com/de/python/topic/3374/postgresql

Kapitel 121: Potenzierung

Syntax

- value1 ** value2
- pow (wert1, wert2 [, wert3])
- value1 .__ pow __ (value2 [, value3])
- value2 .__ rpow__ (value1)
- operator.pow (Wert1, Wert2)
- Operator .__ Pow __ (Wert1, Wert2)
- math.pow (Wert1, Wert2)
- math.sqrt (value1)
- math.exp (Wert1)
- cmath.exp (Wert1)
- math.expm1 (Wert1)

Examples

Quadratwurzel: math.sqrt () und cmath.sqrt

Das `math` Modul enthält die `math.sqrt()` -Funktion, die die Quadratwurzel einer beliebigen Zahl berechnen kann (die in einen `float`). Das Ergebnis ist immer ein `float` :

```
import math

math.sqrt(9)                 # 3.0
math.sqrt(11.11)             # 3.3331666624997918
math.sqrt(Decimal('6.25'))   # 2.5
```

Die `math.sqrt()` Funktion löst einen `ValueError` wenn das Ergebnis `complex` :

```
math.sqrt(-10)
```

ValueError: mathematischer Domänenfehler

`math.sqrt(x)` ist *schneller* als `math.pow(x, 0.5)` oder `x ** 0.5` aber die Genauigkeit der Ergebnisse ist gleich. Das `cmath` Modul ist dem `math` Modul sehr ähnlich, abgesehen davon, dass es komplexe Zahlen berechnen kann und alle Ergebnisse in Form von + bi vorliegen. Es kann auch `.sqrt()` :

```
import cmath

cmath.sqrt(4)  # 2+0j
cmath.sqrt(-4) # 2j
```

Was ist mit dem `j` ? `j` ist das Äquivalent zur Quadratwurzel von -1. Alle Zahlen können in der Form a + bi oder in diesem Fall a + bj eingegeben werden. `a` ist der Realteil der Zahl wie 2 in `2+0j` . Da

es keinen Imaginärteil hat, `b` 0 ist `b` Teil des Imaginärteils der Zahl wie die in 2 repräsentiert `2j` . Da es hier keinen Realteil gibt, kann `2j` auch als `0 + 2j` .

Potenzierung mit Builtins: ** und pow ()

Die Potenzierung kann mit der eingebauten `pow` oder dem Operator `**` werden:

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

Für die meisten (alle in Python 2.x) arithmetischen Operationen ist der Ergebnistyp der Typ des breiteren Operanden. Dies gilt nicht für `**` ; Die folgenden Fälle sind Ausnahmen von dieser Regel:

- Basis: `int` , Exponent: `int < 0` :

  ```
  2 ** -3
  # Out: 0.125 (result is a float)
  ```

- Dies gilt auch für Python 3.x.
- Vor Python 2.2.0 wurde ein `ValueError` .
- Basis: `int < 0` oder `float < 0` , Exponent: `float != int`

  ```
  (-2) ** (0.5)  # also (-2.) ** (0.5)
  # Out: (8.659560562354934e-17+1.4142135623730951j) (result is complex)
  ```

- Vor Python 3.0.0 wurde ein `ValueError` .

Der `operator` Modul enthält zwei Funktionen, die den äquivalent `**` -Operator:

```
import operator
operator.pow(4, 2)       # 16
operator.__pow__(4, 3)   # 64
```

oder man könnte die `__pow__` Methode direkt aufrufen:

```
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)
```

Potenzierung mit dem math Modul: math.pow ()

Das `math` -Modul enthält eine weitere `math.pow()` Funktion. Der Unterschied zur eingebauten `pow()` -Funktion oder dem `**` Operator besteht darin, dass das Ergebnis immer ein `float`

```
import math
math.pow(2, 2)    # 4.0
math.pow(-2., 2)  # 4.0
```

Was schließt Berechnungen mit komplexen Eingaben aus:

```
math.pow(2, 2+0j)
```

TypeError: Komplexe können nicht in Float konvertiert werden

und Berechnungen, die zu komplexen Ergebnissen führen würden:

```
math.pow(-2, 0.5)
```

ValueError: mathematischer Domänenfehler

Exponentialfunktion: math.exp () und cmath.exp ()

Sowohl das `math` als auch das `cmath` Modul enthalten die Euler-Nummer: e und die Verwendung der eingebauten `pow()` oder des `**` -Operators funktioniert meistens wie `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)
```

Das Ergebnis ist jedoch anders und die direkte Verwendung der Exponentialfunktion ist zuverlässiger als die eingebaute Exponentiation mit der Basis `math.e` :

```
print(math.e ** 10)       # 22026.465794806703
print(math.exp(10))       # 22026.465794806718
print(cmath.exp(10).real) # 22026.465794806718
#     difference starts here ---------------^
```

Exponentialfunktion minus 1: math.expm1 ()

Das `math` Modul enthält die `expm1()` -Funktion, die den Ausdruck `math.e ** x - 1` für sehr kleines `x` mit einer höheren Genauigkeit `math.exp(x)` als `math.exp(x)` oder `cmath.exp(x)` :

```
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
#                            ------------------^
```

Bei sehr kleinen x wird der Unterschied größer:

```
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
#                            ^-------------------
```

Die Verbesserung ist im wissenschaftlichen Computing signifikant. Zum Beispiel enthält das Plancksche Gesetz eine Exponentialfunktion minus 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
#                                    ^------------
```

Magische Methoden und Exponentiation: Builtin, Mathe und Cmath

Angenommen, Sie haben eine Klasse, die rein ganzzahlige Werte speichert:

```
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)
```

Mit der eingebauten `pow` Funktion oder dem Operator `**` wird immer `__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__
```

Das zweite Argument der `__pow__()` -Methode kann nur über das BuiltinPow `pow()` oder durch direktes Aufrufen der Methode angegeben werden:

```
pow(Integer(2), 3, 4)           # Integer(0)
# Prints: Using __pow__ with modulo
Integer(2).__pow__(3, 4)        # Integer(0)
# Prints: Using __pow__ with modulo
```

Während die `math` immer in ein `float` konvertieren und die Float-Berechnung verwenden:

```
import math

math.pow(Integer(2), 0.5) # 1.4142135623730951
# Prints: Using __float__
```

`cmath` -Funktionen versuchen , es zu konvertieren `complex` , kann aber auch zu Rückfall `float` , wenn es keine explizite Konvertierung ist `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__
```

Weder `math` noch `cmath` funktionieren, wenn auch die `__float__()` -Methode fehlt:

```
del Integer.__float__  # Deleting __complex__ method

math.sqrt(Integer(2))  # also cmath.exp(Integer(2))
```

TypeError: ein Float ist erforderlich

Modulare Exponentiation: pow () mit 3 Argumenten

Wenn Sie `pow()` mit 3 Argumenten `pow(a, b, c)` bewertet `pow(a, b, c)` die modulare Exponentiation $a^b \bmod c$:

```
pow(3, 4, 17)   # 13
```

```
# equivalent unoptimized expression:
3 ** 4 % 17     # 13

# steps:
3 ** 4          # 81
81 % 17         # 13
```

Bei eingebauten Typen ist die modulare Potenzierung nur möglich, wenn:

- Das erste Argument ist ein `int`
- Zweites Argument ist ein `int >= 0`
- Das dritte Argument ist ein `int != 0`

Diese Einschränkungen sind auch in Python 3.x vorhanden

Zum Beispiel kann man mit der 3-Argument-Form von `pow` eine modulare inverse Funktion definieren:

```
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]
```

Wurzeln: n-te Wurzel mit gebrochenen Exponenten

Während die `math.sqrt` Funktion für den speziellen Fall der Quadratwurzeln vorgesehen ist, ist es oft zweckmäßig, den Exponentiation-Operator (`**`) mit fraktionellen Exponenten zu verwenden, um n-te `math.sqrt` auszuführen, wie z.

Die Umkehrung einer Exponentiation ist die Exponentiation durch den Kehrwert des Exponenten. Wenn Sie also eine Zahl würfeln können, indem Sie sie auf den Exponenten von 3 setzen, können Sie die Kubikwurzel einer Zahl finden, indem Sie sie auf den Exponenten von 1/3 setzen.

```
>>> x = 3
>>> y = x ** 3
>>> y
27
>>> z = y ** (1.0 / 3)
>>> z
3.0
>>> z == x
True
```

Berechnen großer ganzzahliger Wurzeln

Auch wenn Python nativ große Ganzzahlen unterstützt, kann die Verwendung der n-ten Wurzel sehr großer Zahlen in Python fehlschlagen.

```
x = 2 ** 100
cube = x ** 3
```

```
root = cube ** (1.0 / 3)
```

OverflowError: long int zu groß, um in Float konvertiert zu werden

Wenn Sie mit solchen großen Ganzzahlen arbeiten, müssen Sie eine benutzerdefinierte Funktion verwenden, um die n-te Wurzel einer Zahl zu berechnen.

```
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
```

Potenzierung online lesen: https://riptutorial.com/de/python/topic/347/potenzierung

Kapitel 122: Profilierung

Examples

%% Zeit und% Zeit in IPython

Profilierungs-Stringkonfiguration:

```
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
```

Profilierung von Schleifen über Iterables und Listen:

```
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
```

Funktion timeit ()

Profilwiederholung von Elementen in einem Array

```
>>> import timeit
>>> timeit.timeit('list(itertools.repeat("a", 100))', 'import itertools', number = 10000000)
10.997665435877963
>>> timeit.timeit('["a"]*100', number = 10000000)
7.118789926862576
```

timeit Befehlszeile

Profilierung von Zahlen

```
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 in der Befehlszeile

Der Quellcode mit der Direktive @profile vor der Funktion, für die Sie ein Profil erstellen möchten:

```
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()
```

Verwenden Sie den Befehl kernprof, um die Profilierung Zeile für Zeile zu berechnen

```
$ 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)])
```

Die Seitenanforderung ist fast immer langsamer als jede Berechnung, die auf den Informationen auf der Seite basiert.

CProfile (Preferred Profiler) verwenden

Python enthält einen Profiler namens cProfile. Dies wird im Allgemeinen der Verwendung der Zeit bevorzugt.

Es zerlegt Ihr gesamtes Skript und für jede Methode in Ihrem Skript sagt es Ihnen:

- `ncalls` : `ncalls` an, wie oft eine Methode aufgerufen wurde
- `tottime` : Gesamtzeit in der angegebenen Funktion (ohne Zeit in Aufrufen von Unterfunktionen)
- `percall` : Zeit pro Anruf. Oder der Quotient von tottime geteilt durch ncalls
- `cumtime` : Die kumulative Zeit in dieser und allen Unterfunktionen (vom Aufruf bis zum Beenden). Diese Zahl ist auch für rekursive Funktionen genau.
- `percall` : ist der Quotient aus cumtime geteilt durch primitive Aufrufe

- `filename:lineno(function)` die jeweiligen Daten jeder Funktion bereit

Der cProfiler kann einfach über die Befehlszeile aufgerufen werden:

```
$ python -m cProfile main.py
```

So sortieren Sie die zurückgegebene Liste der Profilmethoden nach der in der Methode benötigten Zeit:

```
$ python -m cProfile -s time main.py
```

Profilierung online lesen: https://riptutorial.com/de/python/topic/3818/profilierung

Kapitel 123: Protokollierung

Examples

Einführung in die Python-Protokollierung

Dieses Modul definiert Funktionen und Klassen, die ein flexibles Ereignisprotokollierungssystem für Anwendungen und Bibliotheken implementieren.

Der Hauptvorteil der Protokollierungs-API, die von einem Standardbibliotheksmodul bereitgestellt wird, besteht darin, dass alle Python-Module an der Protokollierung teilnehmen können, sodass Ihr Anwendungsprotokoll Ihre eigenen Nachrichten enthalten kann, die in Nachrichten von Drittanbieter-Modulen integriert sind.

So lass uns anfangen:

Beispielkonfiguration direkt im Code

```
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')
```

Ausgabebeispiel:

```
2016-07-26 18:53:55,332 root         DEBUG    this is a debug test
```

Beispielkonfiguration über eine INI-Datei

Angenommen, die Datei heißt logging_config.ini. Weitere Informationen zum Dateiformat finden Sie im Logging-Konfigurationsabschnitt des Logging-Lernprogramms .

```
[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
```

Verwenden `logging.config.fileConfig()` dann `logging.config.fileConfig()` im Code:

```
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')
```

Beispielkonfiguration über ein Wörterbuch

Ab Python 2.7 können Sie ein Wörterbuch mit Konfigurationsdetails verwenden. PEP 391 enthält eine Liste der obligatorischen und optionalen Elemente im Konfigurationswörterbuch.

```
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')
```

Ausnahmen für die Protokollierung

Wenn Sie Ausnahmen protokollieren möchten, können und sollten Sie die Methode `logging.exception(msg)` verwenden:

```
>>> 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
```

Übergeben Sie die Ausnahme nicht als Argument:

Da `logging.exception(msg)` ein `msg` `logging.exception(msg)` erwartet, ist es eine häufige Gefahr, die Ausnahme wie folgt an den Protokollierungsaufruf zu übergeben:

```
>>> 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
```

Es sieht zwar so aus, als wäre dies anfangs das Richtige, aber es ist problematisch, weil Ausnahmen und verschiedene Codierungen im Protokollierungsmodul zusammenarbeiten:

```
>>> 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
```

Beim Versuch, eine Ausnahme zu protokollieren, die Unicode-Zeichen enthält, schlägt dieser Weg fehl . `logging.exception(e)` wird der Stacktrace der ursprünglichen Ausnahme `logging.exception(e)` indem sie durch eine neue überschrieben wird, die während der Formatierung des `logging.exception(e)` von `logging.exception(e)` .

Natürlich kennen Sie in Ihrem eigenen Code die Kodierung in Ausnahmen. Drittanbieter-Bibliotheken können dies jedoch auf andere Weise behandeln.

Richtige Benutzung:

Wenn Sie statt der Ausnahme einfach eine Nachricht übergeben und Python seine Magie anwenden lassen, funktioniert es:

```
>>> 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
```

Wie Sie sehen, verwenden wir in diesem Fall nicht `e` , und der Aufruf von `logging.exception(...)` formatiert die jüngste Ausnahme magisch.

Protokollierung von Ausnahmen mit Nicht-FEHLER-Protokollierungsstufen

Wenn Sie eine Ausnahme mit einer anderen Protokollebene als ERROR protokollieren möchten, können Sie das Argument `exc_info` der Standardprotokollierer verwenden:

```
logging.debug('exception occurred', exc_info=1)
logging.info('exception occurred', exc_info=1)
logging.warning('exception occurred', exc_info=1)
```

Zugriff auf die Nachricht der Ausnahme

Beachten Sie, dass Bibliotheken da draußen Ausnahmen mit Meldungen als Unicode oder (utf-8, wenn Sie Glück haben) Byte-Zeichenfolgen auslösen können. Wenn Sie wirklich auf den Text einer Ausnahme zugreifen müssen, ist die einzige zuverlässige Methode, die immer funktioniert, die Verwendung von `repr(e)` oder der Zeichenfolge `%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
```

Protokollierung online lesen: https://riptutorial.com/de/python/topic/4081/protokollierung

Kapitel 124: Prozesse und Threads

Einführung

Die meisten Programme werden Zeile für Zeile ausgeführt, wobei jeweils nur ein Prozess ausgeführt wird. Durch Threads können mehrere Prozesse unabhängig voneinander fließen. Durch das Threading mit mehreren Prozessoren können Programme mehrere Prozesse gleichzeitig ausführen. In diesem Thema wird die Implementierung und Verwendung von Threads in Python beschrieben.

Examples

Globale Interpreter-Sperre

Die Leistung von Python-Multithreading kann häufig durch die **Global Interpreter Lock beeinträchtigt werden** . Kurz gesagt, obwohl in einem Python-Programm mehrere Threads vorhanden sein können, kann nur eine Bytecode-Anweisung unabhängig von der Anzahl der CPUs gleichzeitig ausgeführt werden.

Daher kann Multithreading in Fällen, in denen Vorgänge durch externe Ereignisse blockiert werden, wie z. B. Netzwerkzugriff, recht effektiv sein:

```
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
```

Obwohl jeder `process` zwei Sekunden für die Ausführung benötigte, konnten die vier Prozesse zusammen effektiv ausgeführt werden, was insgesamt zwei Sekunden dauerte.

Multithreading führt jedoch in Fällen, in denen intensive Berechnungen in Python-Code ausgeführt

werden, beispielsweise bei vielen Berechnungen, zu keiner großen Verbesserung und kann sogar langsamer sein als das parallele Ausführen:

```
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
```

Im letzteren Fall kann Multiprocessing effektiv sein, da mehrere Prozesse natürlich mehrere Anweisungen gleichzeitig ausführen können:

```
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
```

Ausführen in mehreren Threads

Verwenden Sie `threading.Thread` , um eine Funktion in einem anderen Thread auszuführen.

```
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
```

Ausführen in mehreren Prozessen

Verwenden Sie `multiprocessing.Process` , um eine Funktion in einem anderen Prozess auszuführen. Die Schnittstelle ähnelt `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
```

Status zwischen Threads teilen

Da alle Threads in demselben Prozess ausgeführt werden, haben alle Threads Zugriff auf dieselben Daten.

Der gleichzeitige Zugriff auf gemeinsam genutzte Daten sollte jedoch durch eine Sperre geschützt werden, um Synchronisierungsprobleme zu vermeiden.

```
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}
```

Status zwischen Prozessen teilen

Code, der in verschiedenen Prozessen ausgeführt wird, verwendet standardmäßig nicht die gleichen Daten. Das `multiprocessing` Modul enthält jedoch Grundelemente, die das gemeinsame Nutzen von Werten über mehrere Prozesse unterstützen.

```
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
```

Prozesse und Threads online lesen: https://riptutorial.com/de/python/topic/4110/prozesse-und-threads

Kapitel 125: py.test

Examples

Py.test einrichten

`py.test` ist eine von mehreren Test-Bibliotheken von Drittanbietern , die für Python verfügbar sind. Es kann mit `pip` installiert werden

```
pip install pytest
```

Der zu testende Code

`projectroot/module/code.py` wir an, wir testen eine Zusatzfunktion in `projectroot/module/code.py` :

```
# projectroot/module/code.py
def add(a, b):
    return a + b
```

Der Prüfcode

Wir erstellen eine Testdatei in `projectroot/tests/test_code.py` . Die Datei **muss mit `test_` beginnen** , um als `test_` erkannt zu werden.

```
# projectroot/tests/test_code.py
from module import code


def test_add():
    assert code.add(1, 2) == 3
```

Den Test ausführen

Von `projectroot` wir einfach `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
================================================
```

Fehlgeschlagene Tests

Ein fehlgeschlagener Test liefert hilfreiche Informationen darüber, was falsch gelaufen ist:

```
# projectroot/tests/test_code.py
from module import code


def test_add__failing():
    assert code.add(10, 11) == 33
```

Ergebnisse:

```
$ 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
================================================
```

Einführung in Test-Fixtures

Für kompliziertere Tests müssen manchmal einige Dinge eingerichtet werden, bevor Sie den Code ausführen, den Sie testen möchten. Es ist möglich, dies in der Testfunktion selbst auszuführen, aber am Ende müssen Sie mit großen Testfunktionen so viel tun, dass es schwierig ist zu sagen, wo das Setup stoppt und der Test beginnt. Sie können auch eine Menge doppelten Setup-Codes zwischen Ihren verschiedenen Testfunktionen erhalten.

Unsere Code-Datei:

```
# projectroot/module/stuff.py
class Stuff(object):
    def prep(self):
        self.foo = 1
```

```
        self.bar = 2
```

Unsere Testdatei:

```
# 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
```

Dies sind ziemlich einfache Beispiele, aber wenn für unser `Stuff` Objekt mehr Setup erforderlich wäre, würde es schwerfällig werden. Wir haben festgestellt, dass zwischen unseren Testfällen ein paar doppelte Codes vorhanden sind. Lassen Sie uns den Code zuerst in eine separate Funktion umwandeln.

```
# 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
```

Das sieht zwar besser aus, aber wir haben immer noch den `my_stuff = get_prepped_stuff()` , der unsere Testfunktionen `my_stuff = get_prepped_stuff()` .

py.test Geräte zur Rettung!

Fixtures sind wesentlich leistungsfähigere und flexiblere Versionen von Testeinrichtungsfunktionen. Sie können viel mehr, als wir hier nutzen, aber wir gehen Schritt für Schritt vor.

Zuerst ändern wir `get_prepped_stuff` in ein Fixture namens `prepped_stuff` . Sie möchten Ihre Geräte mit Substantiven anstatt mit Verben benennen, da die Geräte später in den Testfunktionen selbst verwendet werden. Das `@pytest.fixture` gibt an, dass diese spezifische Funktion als Fixture und nicht als reguläre Funktion behandelt werden soll.

```
@pytest.fixture
def prepped_stuff():
    my_stuff = stuff.Stuff()
    my_stuff.prep()
    return my_stuff
```

Jetzt sollten wir die Testfunktionen aktualisieren, damit sie das Gerät verwenden. Dazu fügen Sie ihrer Definition einen Parameter hinzu, der genau mit dem Namen des Geräts übereinstimmt. Wenn py.test ausgeführt wird, wird das Gerät vor dem Test ausgeführt und der Rückgabewert des Geräts wird durch diesen Parameter an die Testfunktion übergeben. (Beachten Sie, dass Vorrichtungen **brauchen** keinen Wert zurückgeben, sie anderen Setup Dinge tun können , statt wie eine externe Ressource aufrufen, die Dinge auf dem Dateisystem Arrangieren, setzen Werte in einer Datenbank, was die Tests müssen für die Einrichtung)

```
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
```

Jetzt können Sie sehen, warum wir es mit einem Nomen benannt haben. aber die `my_stuff = prepped_stuff` Zeile ist ziemlich nutzlos, also verwenden `prepped_stuff` stattdessen direkt `prepped_stuff` .

```
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
```

Jetzt benutzen wir Fixtures! Wir können weitergehen, indem wir den Umfang des Geräts ändern (dh es wird nur einmal pro Testmodul oder Testsuite-Ausführungssitzung statt nur einmal pro Testfunktion ausgeführt), indem Geräte erstellt werden, die andere Geräte verwenden, und das Gerät (und damit alle Geräte) Tests mit diesem Fixture werden mehrmals ausgeführt (einmal für jeden Parameter, der dem Fixture übergeben wird), Fixtures, die Werte aus dem Modul lesen, das sie aufruft ... Wie bereits erwähnt, haben Fixtures viel mehr Leistung und Flexibilität als eine normale Setup-Funktion.

Nach den Tests aufräumen.

Nehmen wir an, unser Code ist gewachsen und unser Stuff-Objekt muss jetzt besonders aufgeräumt werden.

```
# 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
```

Wir könnten etwas Code hinzufügen, um die Aufräumaktion am Ende jeder Testfunktion aufzurufen. Fixtures bieten jedoch eine bessere Möglichkeit, dies zu tun. Wenn Sie dem Gerät eine Funktion hinzufügen und es als **Finalizer** registrieren, wird der Code in der Finalizer-Funktion aufgerufen, nachdem der Test mit dem Gerät durchgeführt wurde. Wenn der Umfang des Fixtures größer ist als eine einzelne Funktion (wie ein Modul oder eine Sitzung), wird der Finalizer ausgeführt, nachdem alle Tests im Umfang abgeschlossen sind. Nach Abschluss des Moduls oder am Ende der gesamten Testlaufsitzung .

```
@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
```

Die Verwendung der Finalizer-Funktion innerhalb einer Funktion kann auf den ersten Blick etwas schwer zu verstehen sein, insbesondere bei komplizierteren Geräten. Sie können stattdessen ein **Ertrags-Fixture verwenden** , um dasselbe mit einem besser lesbaren Ausführungsablauf zu tun. Der einzige wirkliche Unterschied besteht darin, dass wir anstelle von `return` eine `yield` für den Teil des Geräts verwenden, in dem das Setup durchgeführt wird und die Steuerung zu einer Testfunktion gehen sollte, und dann den gesamten Bereinigungscode nach der `yield` hinzufügen. Wir dekorieren es auch als `yield_fixture` damit py.test damit umgehen kann.

```
@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()
```

Und damit ist das Intro zu Test Fixtures abgeschlossen!

Weitere Informationen finden Sie in der offiziellen Dokumentation des py.test-Geräts und in der Dokumentation des offiziellen Ertragsgeräts

py.test online lesen: https://riptutorial.com/de/python/topic/7054/py-test

Kapitel 126: pyaudio

Einführung

PyAudio bietet Python-Bindungen für PortAudio, die plattformübergreifende Audio-E / A-Bibliothek. Mit PyAudio können Sie Python problemlos verwenden, um Audio auf verschiedenen Plattformen abzuspielen und aufzunehmen. PyAudio ist inspiriert von:

1.pyPortAudio / fastaudio: Python-Bindungen für die PortAudio v18-API.

2.tkSnack: Cross-Plattform-Sound-Toolkit für Tcl / Tk und Python.

Bemerkungen

Hinweis: stream_callback wird in einem separaten Thread (vom Hauptthread) aufgerufen. Ausnahmen, die im stream_callback auftreten, werden:
1 .print eine Rückverfolgung auf dem Standardfehlerdebugging zu unterstützen,
2 .queue die Ausnahme, die (zu einem bestimmten Zeitpunkt) im Hauptthread geworfen werden soll, und
3. Bringen Sie paAbort zu PortAudio zurück, um den Stream zu stoppen.
Hinweis: Rufen Sie nicht Stream.read () oder Stream.write () auf, wenn Sie einen nicht blockierenden Vorgang verwenden.
Weitere Informationen finden Sie unter PortAudio-Rückrufsignatur:
http://portaudio.com/docs/v19-doxydocs/portaudio_8h.html#a8a60fb2a5ec9cbade3f54a9c978e2710

Examples

Callback-Modus Audio-E / A

```
"""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()
```

Im Callback-Modus ruft PyAudio eine angegebene Callback-Funktion (2) auf, wenn neue Audiodaten benötigt werden (zum Abspielen) und / oder wenn neue (aufgezeichnete) Audiodaten verfügbar sind. Beachten Sie, dass PyAudio die Rückruffunktion in einem separaten Thread aufruft. Die Funktion hat den folgenden Signatur- `callback(<input_data>, <frame_count>, <time_info>, <status_flag>)` und muss ein Tupel zurückgeben, das `frame_count` Frames mit Audiodaten und ein Flag enthält, das `frame_count` , ob weitere Frames zum Abspielen / Aufnehmen vorhanden sind.

Starten Sie die Verarbeitung des Audiostreams mit **pyaudio.Stream.start_stream ()** (4), die die Callback-Funktion wiederholt **aufruft,** bis diese Funktion **pyaudio.paComplete zurückgibt** .

Um den Stream aktiv zu halten, darf der Haupt-Thread nicht beendet werden, z. B. durch Schlafen (5).

Audio-E / A im Blockierungsmodus

"" "PyAudio-Beispiel: Abspielen einer Wave-Datei." ""

```
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()
```

Um PyAudio zu verwenden, instanziieren Sie zuerst PyAudio mit **pyaudio.PyAudio ()** (1), wodurch das Portaudio-System eingerichtet wird.

Um Audio aufzunehmen oder abzuspielen, öffnen Sie mit **pyaudio.PyAudio.open ()** (2) einen Stream mit den gewünschten **Audioparametern** auf dem gewünschten Gerät. Dadurch wird ein **pyaudio.Stream** zur Wiedergabe oder Aufnahme von Audio eingerichtet.

Sie können Audiodaten **abspielen** , indem Sie Audiodaten mit **pyaudio.Stream.write ()** in den Stream **schreiben** oder Audiodaten mit **pyaudio.Stream.read ()** aus dem Stream **lesen** . (3)

Beachten Sie, dass im " *Blockierungsmodus* " jeder **pyaudio.Stream.write ()** oder **pyaudio.Stream.read ()** blockiert, bis alle angegebenen / angeforderten Frames abgespielt / aufgezeichnet wurden. Alternativ können Sie Audiodaten im laufenden Betrieb erzeugen oder aufgezeichnete Audiodaten sofort verarbeiten, indem Sie den „Rückrufmodus“ verwenden (*siehe Beispiel zum Rückrufmodus*).

Verwenden Sie pyaudio.Stream.stop_stream (), um die Wiedergabe / Aufnahme **anzuhalten** , und **pyaudio.Stream.close (),** um den Stream zu beenden. (4)

Beenden Sie schließlich die Portaudio-Sitzung mit **pyaudio.PyAudio.terminate ()** (5).

pyaudio online lesen: https://riptutorial.com/de/python/topic/10627/pyaudio

Kapitel 127: Pyautogui-Modul

Einführung

pyautogui ist ein Modul zur Steuerung von Maus und Tastatur. Dieses Modul wird im Wesentlichen zur Automatisierung von Mausklick- und Tastendruckaufgaben verwendet. Bei der Maus beginnen die Koordinaten des Bildschirms (0,0) in der oberen linken Ecke. Wenn Sie außer Kontrolle geraten, bewegen Sie den Mauszeiger schnell nach oben links. Die Maus und die Tastatur werden vom Python gesteuert und geben Sie an Sie zurück.

Examples

Mausfunktionen

Dies sind einige nützliche Mausfunktionen zur Steuerung der Maus.

```
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.
```

Tastaturfunktionen

Dies sind einige nützliche Tastaturfunktionen zum Automatisieren des Tastendrucks.

```
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 und Bilderkennung

Diese Funktion hilft Ihnen, den Screenshot aufzunehmen und das Bild mit dem Bildschirmteil abzugleichen.

```
.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.
```

Pyautogui-Modul online lesen: https://riptutorial.com/de/python/topic/9432/pyautogui-modul

Kapitel 128: Pygame

Einführung

Pygame ist die Go-To-Bibliothek zum Erstellen von Multimedia-Anwendungen, insbesondere Spielen, in Python. Die offizielle Website ist http://www.pygame.org/ .

Syntax

- pygame.mixer.init (Frequenz = 22050, Größe = -16, Kanäle = 2, Puffer = 4096)
- pygame.mixer.pre_init (Frequenz, Größe, Kanäle, Puffer)
- pygame.mixer.quit ()
- pygame.mixer.get_init ()
- pygame.mixer.stop ()
- pygame.mixer.pause ()
- pygame.mixer.unpause ()
- pygame.mixer.fadeout (Zeit)
- pygame.mixer.set_num_channels (count)
- pygame.mixer.get_num_channels ()
- pygame.mixer.set_reserved (count)
- pygame.mixer.find_channel (force)
- pygame.mixer.get_busy ()

Parameter

Parameter	Einzelheiten
Anzahl	Eine positive ganze Zahl, die etwa der Anzahl der zu reservierenden Kanäle entspricht.
Macht	Ein boolescher Wert (`False` oder `True`), der bestimmt, ob `find_channel()` einen Kanal (inaktiv oder nicht) mit `True` oder nicht (wenn keine inaktiven Kanäle vorhanden sind) mit `False`

Examples

Pygame installieren

Mit `pip` :

```
pip install pygame
```

Mit `conda` :

```
conda install -c tlatorre pygame=1.9.2
```

Direkter Download von der Website: http://www.pygame.org/download.shtml

Sie finden die geeigneten Installationsprogramme für Windows und andere Betriebssysteme.

Projekte finden Sie auch unter http://www.pygame.org/

Pygame's Mischermodul

Das Modul `pygame.mixer` hilft bei der Steuerung der in `pygame` Programmen verwendeten Musik. Ab sofort gibt es 15 verschiedene Funktionen für das `mixer` .

Initialisierung

Ähnlich wie Sie `pygame` mit `pygame.init()` initialisieren müssen, müssen Sie auch `pygame.mixer` initialisieren.

Mit der ersten Option initialisieren wir das Modul mit den Standardwerten. Sie können diese Standardoptionen jedoch überschreiben. Mit der zweiten Option können wir das Modul mit den von uns manuell eingegebenen Werten initialisieren. Standardwerte:

```
pygame.mixer.init(frequency=22050, size=-16, channels=2, buffer=4096)
```

Um zu überprüfen, ob wir es initialisiert haben oder nicht, können wir `pygame.mixer.get_init()` , das `True` zurückgibt, wenn es ist, und `False` wenn nicht. Um die Initialisierung zu beenden / rückgängig zu machen, verwenden Sie einfach `pygame.mixer.quit()` . Wenn Sie mit dem Modul weitere Sounds abspielen möchten, müssen Sie das Modul möglicherweise neu initialisieren.

Mögliche Aktionen

Während der Sound abgespielt wird, können Sie ihn mit `pygame.mixer.pause()` vorübergehend `pygame.mixer.pause()` . Um die Wiedergabe Ihrer Sounds `pygame.mixer.unpause()` , verwenden Sie einfach `pygame.mixer.unpause()` . Sie können das Ende des Sounds auch mit `pygame.mixer.fadeout()` . Es braucht ein Argument, nämlich die Anzahl der Millisekunden, die zum Ausblenden der Musik benötigt werden.

Channels

Sie können so viele Songs wie nötig abspielen, solange genügend freie Kanäle zur Unterstützung vorhanden sind. Standardmäßig gibt es 8 Kanäle. Um die Anzahl der Kanäle zu ändern, verwenden Sie `pygame.mixer.set_num_channels()` . Das Argument ist eine nicht negative ganze Zahl. Wenn die Anzahl der Kanäle verringert wird, werden alle auf den entfernten Kanälen wiedergegebenen Sounds sofort beendet.

Um herauszufinden, wie viele Kanäle aktuell verwendet werden, rufen Sie `pygame.mixer.get_channels(count)` . Die Ausgabe ist die Anzahl der Kanäle, die derzeit nicht geöffnet sind. Sie können Kanäle auch für Sounds reservieren, die mit `pygame.mixer.set_reserved(count)` abgespielt werden `pygame.mixer.set_reserved(count)` . Das Argument ist auch eine nicht negative ganze Zahl. Alle Sounds, die auf den neu reservierten Kanälen abgespielt werden, werden nicht angehalten.

Sie können auch herausfinden, welcher Kanal nicht verwendet wird, indem Sie `pygame.mixer.find_channel(force)` . Ihr Argument ist ein bool: entweder Wahr oder Falsch. Wenn es keine Kanäle gibt, die frei sind und `force` False ist, wird `None` . Wenn `force` Wert true hat, wird der Kanal zurückgegeben, der die längste Zeit gespielt hat.

Pygame online lesen: https://riptutorial.com/de/python/topic/8761/pygame

Kapitel 129: Pyglet

Einführung

Pyglet ist ein Python-Modul für Visuals und Sound. Es besteht keine Abhängigkeit von anderen Modulen. Die offiziellen Informationen finden Sie unter [pyglet.org] [1]. [1]: http://pyglet.org

Examples

Hallo Welt in Pyglet

```
import pyglet
window = pyglet.window.Window()
label = pyglet.text.Label('Hello, world',
                      font_name='Times New Roman',
                      font_size=36,
                      x=window.width//2, y=window.height//2,
                      anchor_x='center', anchor_y='center')
@window.event
def on_draw():
    window.clear()
    label.draw()
pyglet.app.run()
```

Installation von Pyglet

Installieren Sie Python, gehen Sie in die Befehlszeile und geben Sie Folgendes ein:

Python 2:

```
pip install pyglet
```

Python 3:

```
pip3 install pyglet
```

Sound in Pyglet abspielen

```
sound = pyglet.media.load(sound.wav)
sound.play()
```

Pyglet für OpenGL verwenden

```
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()
```

Zeichnungspunkte mit Pyglet und 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
```

Ersetzen `GL_POINTS` zum Verbinden der Punkte `GL_POINTS` durch `GL_LINE_LOOP` .

Pyglet online lesen: https://riptutorial.com/de/python/topic/8208/pyglet

Kapitel 130: PyInstaller - Verteilen von Python-Code

Syntax

- pyinstaller [Optionen] Skript [Skript ...] | Specfile

Bemerkungen

PyInstaller ist ein Modul, mit dem Python-Apps in einem Paket zusammen mit allen Abhängigkeiten gebündelt werden. Der Benutzer kann dann die Paket-App ohne einen Python-Interpreter oder irgendwelche Module ausführen. Es bündelt viele wichtige Pakete wie Numpy, Django, OpenCv und andere.

Einige wichtige Punkte zum Erinnern:

- Pyinstaller unterstützt Python 2.7 und Python 3.3+
- Pyinstaller wurde gegen Windows, Linux und Mac OS X getestet.
- Es ist **kein** Cross Compiler. (Eine Windows-App kann nicht in Linux gepackt werden. Sie müssen PyInstaller in Windows ausführen, um eine App für Windows zu bündeln.)

Homepage Offizielle Dokumente

Examples

Installation und Einrichtung

Pyinstaller ist ein normales Python-Paket. Es kann mit pip installiert werden:

```
pip install pyinstaller
```

Installation unter Windows
Für Windows ist pywin32 oder pypiwin32 eine Voraussetzung. Letzteres wird automatisch installiert, wenn pyinstaller mit pip installiert wird.

Installation unter Mac OS X
PyInstaller arbeitet mit dem Standard-Python 2.7, der im aktuellen Mac OS X enthalten ist. Wenn spätere Versionen von Python verwendet werden sollen oder wenn Hauptpakete wie PyQT, Numpy, Matplotlib und dergleichen verwendet werden, wird empfohlen, sie mit zu installieren entweder MacPorts oder Homebrew .

Installation aus dem Archiv
Wenn pip nicht verfügbar ist, laden Sie das komprimierte Archiv von PyPI herunter .
Laden Sie zum Testen der Entwicklungsversion das komprimierte Archiv vom *Entwicklungszweig*

der PyInstaller-Downloadseite herunter .

Erweitern Sie das Archiv und suchen Sie das Skript `setup.py` . `python setup.py install` mit Administratorrechten aus, um PyInstaller zu installieren oder zu aktualisieren.

Installation überprüfen
Der Befehl `pyinstaller` sollte nach erfolgreicher Installation für alle Plattformen im `pyinstaller` vorhanden sein.
Überprüfen Sie dies, indem `pyinstaller --version` in der Befehlszeile `pyinstaller --version` eingeben. Dadurch wird die aktuelle Version von pyinstaller gedruckt.

Pyinstaller verwenden

Navigieren Sie im einfachsten Anwendungsfall einfach zu dem Verzeichnis, in dem sich Ihre Datei befindet, und geben Sie Folgendes ein:

```
pyinstaller myfile.py
```

Pyinstaller analysiert die Datei und erstellt:

- Eine Datei **myfile.spec** im selben Verzeichnis wie `myfile.py`
- Ein **Build-** Ordner im selben Verzeichnis wie `myfile.py`
- Ein **dist-** Ordner im selben Verzeichnis wie `myfile.py`
- Protokolldateien im **Erstellungsordner**

Die mitgelieferte App befindet sich im Ordner **dist**

Optionen
Es gibt verschiedene Optionen, die mit dem pyinstaller verwendet werden können. Eine vollständige Liste der Optionen finden Sie hier .

Einmal gebündelt kann Ihre App ausgeführt werden, indem Sie 'dist \ myfile \ myfile.exe' öffnen.

In einem Ordner bündeln

Wenn PyInstaller ohne Optionen zum `myscript.py` , ist die Standardausgabe ein einzelner Ordner (namens `myscript`), der eine ausführbare Datei namens `myscript` (in Windows `myscript.exe`) mit allen erforderlichen Abhängigkeiten enthält.
Die App kann verteilt werden, indem der Ordner in eine ZIP-Datei komprimiert wird.

Ein Ordnermodus kann explizit mit der Option `-D` oder `--onedir`

```
pyinstaller myscript.py -D
```

Vorteile:

Einer der Hauptvorteile der Bündelung in einem einzigen Ordner besteht darin, dass Probleme leichter zu debuggen sind. Wenn ein Modul nicht importiert werden kann, kann dies durch

Überprüfen des Ordners überprüft werden.
Ein weiterer Vorteil ist bei Updates zu spüren. Wenn der Code einige Änderungen enthält, die verwendeten Abhängigkeiten jedoch *genau* gleich sind, können Verteiler die ausführbare Datei (die normalerweise kleiner als der gesamte Ordner ist) versenden.

Nachteile

Der einzige Nachteil dieser Methode besteht darin, dass die Benutzer unter einer großen Anzahl von Dateien nach der ausführbaren Datei suchen müssen.
Benutzer können auch andere Dateien löschen / ändern, was dazu führen kann, dass die App nicht ordnungsgemäß funktioniert.

Bündeln in eine einzelne Datei

```
pyinstaller myscript.py -F
```

Die Optionen zum Generieren einer einzelnen Datei sind `-F` oder `--onefile` . Dadurch wird das Programm in einer einzigen `myscript.exe` Datei zusammengefasst.

Eine einzelne Datei ist langsamer als das Ein-Ordner-Paket. Sie sind auch schwieriger zu debuggen.

PyInstaller - Verteilen von Python-Code online lesen:
https://riptutorial.com/de/python/topic/2289/pyinstaller---verteilen-von-python-code

Kapitel 131: Python aus C # aufrufen

Einführung

Die Dokumentation enthält eine Beispielimplementierung der Kommunikation zwischen C # - und Python-Skripts.

Bemerkungen

Beachten Sie, dass die Daten im obigen Beispiel mit der **MongoDB.Bson-** Bibliothek serialisiert werden, die über den NuGet Manager installiert werden kann.

Ansonsten können Sie eine beliebige JSON-Serialisierungsbibliothek Ihrer Wahl verwenden.

Nachfolgend sind Implementierungsschritte für die Kommunikation zwischen Prozessen aufgeführt:

- Eingabeargumente werden in eine JSON-Zeichenfolge serialisiert und in einer temporären Textdatei gespeichert:

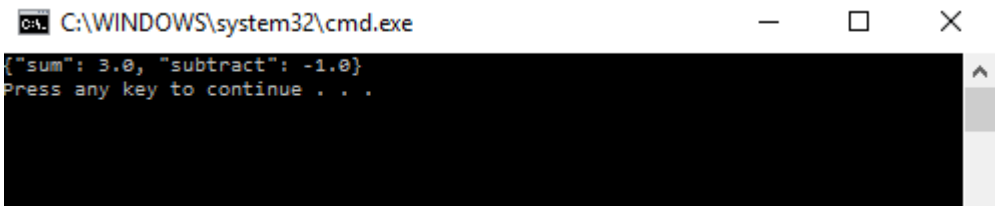
```
 BsonDocument argsBson = BsonDocument.Parse("{ 'x' : '1', 'y' : '2' }");
 string argsFile = string.Format("{0}\\{1}.txt", Path.GetDirectoryName(pyScriptPath),
Guid.NewGuid());
```

- Der Python-Interpreter python.exe führt das Python-Skript aus, das die JSON-Zeichenfolge aus einer temporären Textdatei liest und Eingabeargumente zurücksetzt:

```
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' ] ]
```

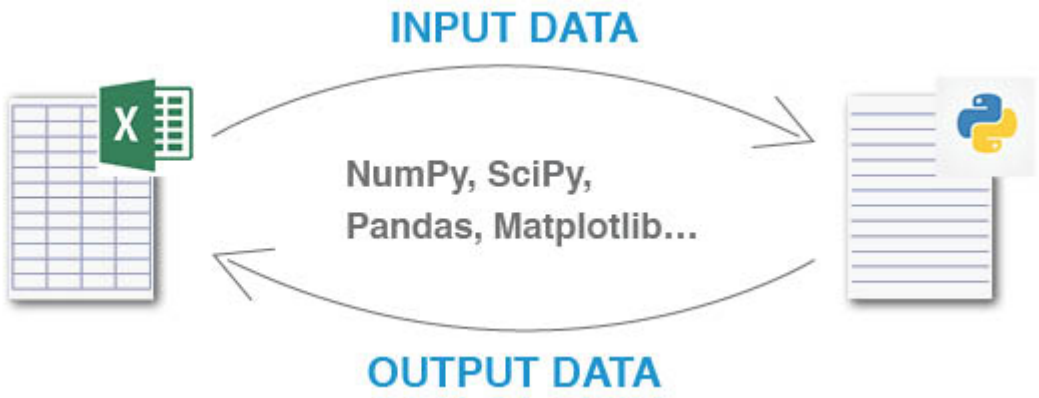
- Das Python-Skript wird ausgeführt und das Ausgabewörterbuch in eine JSON-Zeichenfolge serialisiert und im Befehlsfenster gedruckt:

```
print json.dumps( { 'sum' : x + y , 'subtract' : x - y } )
```



- Lesen Sie den Ausgabe-JSON-String aus der C # -Anwendung:

```
using (StreamReader myStreamReader = process.StandardOutput)
{
   outputString = myStreamReader.ReadLine();
   process.WaitForExit();
}
```



Ich verwende in einem meiner Projekte die Kommunikation zwischen C # - und Python-Skripts, die das direkte Aufrufen von Python-Skripts aus Excel-Tabellen ermöglicht.

Das Projekt verwendet das ExcelDNA-Add-In für die C # - Excel-Bindung.

Der Quellcode wird im GitHub- Repository gespeichert.

Nachfolgend finden Sie Links zu Wiki-Seiten, die einen Überblick über das Projekt bieten und in 4 einfachen Schritten zum Einstieg beitragen.

- Fertig machen
- Übersicht über die Implementierung
- Beispiele
- Objekt-Assistent
- Funktionen

Ich hoffe, Sie finden das Beispiel und das Projekt nützlich.

Examples

Python-Skript, das von der C # -Anwendung aufgerufen werden soll

```
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 } )
```

C # -Code, der ein Python-Skript aufruft

```
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);
        }
    }
}
```

Python aus C # aufrufen online lesen: https://riptutorial.com/de/python/topic/10759/python-aus-c-sharp-aufrufen

Kapitel 132: Python Lex-Yacc

Einführung

PLY ist eine reine Python-Implementierung der beliebten Compiler-Konstruktionswerkzeuge Lex und Yacc.

Bemerkungen

Zusätzliche links:

1. Offizielle Dokumente
2. Github

Examples

Erste Schritte mit PLY

Um PLY auf Ihrem Rechner für Python2 / 3 zu installieren, führen Sie die folgenden Schritte aus:

1. Laden Sie den Quellcode hier herunter.
2. Entpacken Sie die heruntergeladene ZIP-Datei
3. Navigieren Sie in den entpackten Ordner `ply-3.10`
4. Führen Sie den folgenden Befehl in Ihrem Terminal aus: `python setup.py install`

Wenn Sie alle oben genannten Schritte ausgeführt haben, sollten Sie jetzt das PLY-Modul verwenden können. Sie können es testen, indem Sie einen Python-Interpreter öffnen und `import ply.lex`.

Hinweis: Verwenden Sie *keine* `pip` PLY zu installieren, wird es eine gebrochene Verteilung auf Ihrem Rechner installieren.

Das "Hallo, Welt!" of PLY - Ein einfacher Rechner

Lassen Sie uns die Leistungsfähigkeit von PLY anhand eines einfachen Beispiels demonstrieren: Dieses Programm nimmt einen arithmetischen Ausdruck als Zeichenfolgeingabe und versucht, ihn zu lösen.

Öffnen Sie Ihren bevorzugten Editor und kopieren Sie den folgenden Code:

```
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)
```

Speichern Sie diese Datei als `calc.py` und führen Sie sie aus.

Ausgabe:

```
-8
```

Welches ist die richtige Antwort für `-4 * - (3 - 5)` .

Teil 1: Tokenisierung mit Lex

Es gibt zwei Schritte, die der Code aus Beispiel 1 ausführte: *Erstens wurde* die Eingabe mit einem *Token versehen* , das heißt, es wurde nach Symbolen gesucht, die den arithmetischen Ausdruck bilden, und der zweite Schritt war das *Parsing* , bei dem die extrahierten Token analysiert und das Ergebnis bewertet werden.

Dieser Abschnitt enthält ein einfaches Beispiel für die *Tokensierung von* Benutzereingaben und unterteilt sie dann Zeile für Zeile.

```
    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)
```

Speichern Sie diese Datei als `calclex.py` . Wir werden dies beim Bau unseres Yacc-Parsers verwenden.

Nervenzusammenbruch

1. Importieren Sie das Modul mit `import ply.lex`
2. Alle Lexer müssen eine Liste namens `tokens` bereitstellen, die alle möglichen Token-Namen definiert, die vom Lexer erzeugt werden können. Diese Liste ist immer erforderlich.

```
tokens = [
   'NUMBER',
   'PLUS',
   'MINUS',
   'TIMES',
   'DIVIDE',
   'LPAREN',
   'RPAREN',
]
```

`tokens` können auch ein Tupel von Strings (und nicht ein String) sein, wobei jeder String ein Token wie zuvor bezeichnet.

3. Die Regex-Regel für jeden String kann entweder als String oder als Funktion definiert werden. In beiden Fällen sollte dem Variablennamen t_ vorangestellt werden, um anzugeben, dass dies eine Regel für übereinstimmende Token ist.

    - Für einfache Token kann der reguläre Ausdruck als Zeichenfolge angegeben werden: `t_PLUS = r'\+'`
    - Wenn eine Aktion ausgeführt werden muss, kann eine Tokenregel als Funktion angegeben werden.

    ```
    def t_NUMBER(t):
        r'\d+'
        t.value = int(t.value)
        return t
    ```

    Beachten Sie, dass die Regel als Dokumentzeichenfolge innerhalb der Funktion angegeben wird. Die Funktion akzeptiert ein Argument, das eine Instanz von `LexToken` , führt eine Aktion aus und gibt das Argument zurück.

    Wenn Sie einen externen String als Regex-Regel für die Funktion verwenden möchten, anstatt einen Doc-String anzugeben, betrachten Sie das folgende Beispiel:

    ```
    @TOKEN(identifier)          # identifier is a string holding the regex
    def t_ID(t):
        ...      # actions
    ```

    - Eine Instanz des `LexToken` Objekts (nennen wir dieses Objekt `t`) hat die folgenden Attribute:

        1. `t.type` ist der Token-Typ (als Zeichenfolge) (z. B. `'NUMBER'` , `'PLUS'` usw.). Standardmäßig ist `t.type` nach dem Präfix `t_` auf den Namen gesetzt.
        2. `t.value` was das Lexem ist (der tatsächliche Text, der angeglichen wird)
        3. `t.lineno` ist die aktuelle Zeilennummer (diese wird nicht automatisch aktualisiert, da der Lexer nichts von Zeilennummern weiß). Aktualisieren Sie Leineno mit einer Funktion namens `t_newline` .

        ```
        def t_newline(t):
            r'\n+'
            t.lexer.lineno += len(t.value)
        ```

        4. `t.lexpos` ist die Position des Tokens relativ zum Anfang des Eingabetextes.

    - Wenn von einer Regex-Regelfunktion nichts zurückgegeben wird, wird das Token verworfen. Wenn Sie ein Token verwerfen möchten, können Sie alternativ zu einer Regex-Regelvariablen das Präfix t_ignore_ hinzufügen, anstatt eine Funktion für dieselbe Regel zu definieren.

    ```
    def t_COMMENT(t):
        r'\#.*'
    ```

```
        pass
        # No return value. Token discarded
```

...Ist das gleiche wie:

```
    t_ignore_COMMENT = r'\#.*'
```

Dies ist natürlich ungültig, wenn Sie eine Aktion ausführen, wenn Sie einen Kommentar sehen. Verwenden Sie in diesem Fall eine Funktion, um die Regex-Regel zu definieren.

Wenn Sie für einige Zeichen kein Token definiert haben, es aber dennoch ignorieren möchten, verwenden Sie `t_ignore = "<characters to ignore>"` (diese Präfixe sind erforderlich):

```
    t_ignore_COMMENT = r'\#.*'
    t_ignore  = ' \t'    # ignores spaces and tabs
```

- Beim Erstellen des Master-Regex fügt lex die in der Datei angegebenen Regexes wie folgt hinzu:

    1. Von Funktionen definierte Token werden in derselben Reihenfolge hinzugefügt, in der sie in der Datei angezeigt werden.
    2. Von Zeichenfolgen definierte Token werden in abnehmender Reihenfolge der Zeichenfolgenlänge der Zeichenfolge hinzugefügt, die den regulären Ausdruck für dieses Token definiert.

    Wenn Sie `==` und `=` in derselben Datei finden, nutzen Sie diese Regeln.

- Literale sind Marken, die zurückgegeben werden, wie sie sind. Sowohl `t.type` als auch `t.value` werden auf das Zeichen selbst gesetzt. Definieren Sie eine Liste von Literalen als solche:

```
literals = [ '+', '-', '*', '/' ]
```

oder,

```
literals = "+-*/"
```

Es ist möglich, Token-Funktionen zu schreiben, die zusätzliche Aktionen ausführen, wenn Literale abgeglichen werden. Sie müssen jedoch den Tokentyp entsprechend festlegen. Zum Beispiel:

```
literals = [ '{', '}' ]

def t_lbrace(t):
    r'\{'
    t.type = '{'  # Set token type to the expected literal (ABSOLUTE MUST if this
is a literal)
```

```
    return t
```

- Behandeln Sie Fehler mit der Funktion 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)
```

Im Allgemeinen überspringt `t.lexer.skip(n)` n Zeichen in der Eingabezeichenfolge.

4. Letzte Vorbereitungen:

Erstellen Sie den Lexer mit `lexer = lex.lex()` .

Sie können auch alles in eine Klasse einfügen und die use-Instanz der Klasse aufrufen, um den Lexer zu definieren. Z.B:

```
 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")     #
```

Geben Sie die Eingabe mithilfe von `lexer.input(data)` wobei data eine Zeichenfolge ist

Um die Token zu erhalten, verwenden Sie `lexer.token()` das übereinstimmende Token zurückgibt. Sie können in einer Schleife über Lexer iterieren wie in:

```
for i in lexer:
    print(i)
```

Teil 2: Analyse von getakteten Eingaben mit Yacc

In diesem Abschnitt wird erläutert, wie die mit Token versehenen Eingaben aus Teil 1 verarbeitet werden. Dies geschieht mithilfe von Context Free Grammars (CFGs). Die Grammatik muss angegeben werden und die Token werden gemäß der Grammatik verarbeitet. Unter der Haube verwendet der Parser einen LALR-Parser.

```
# 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)
```

Nervenzusammenbruch

- Jede Grammatikregel wird durch eine Funktion definiert, bei der der Docstring für diese Funktion die entsprechende kontextfreie Grammatikspezifikation enthält. Die Anweisungen, aus denen der Funktionskörper besteht, implementieren die semantischen Aktionen der Regel. Jede Funktion akzeptiert ein einzelnes Argument p, das eine Folge ist, die die Werte jedes Grammatiksymbols in der entsprechenden Regel enthält. Die Werte von `p[i]` werden wie hier gezeigt auf Grammatiksymbole abgebildet:

```
def p_expression_plus(p):
    'expression : expression PLUS term'
    #   ^            ^        ^    ^
    #  p[0]         p[1]     p[2] p[3]

    p[0] = p[1] + p[3]
```

- Bei Tokens entspricht der "Wert" des entsprechenden `p[i]` dem im Lexer-Modul zugewiesenen `p.value` Attribut. `PLUS` hat also den Wert `+` .

- Für Nichtterminals wird der Wert durch das bestimmt, was in `p[0]` . Wenn nichts platziert wird, lautet der Wert None. Außerdem ist `p[-1]` nicht dasselbe wie `p[3]` , da `p` keine einfache Liste ist (`p[-1]` kann eingebettete Aktionen angeben (hier nicht erläutert)).

Beachten Sie, dass die Funktion einen beliebigen Namen haben kann, solange `p_` vorangestellt ist.

- Die `p_error(p)` -Regel wird definiert, um Syntaxfehler `yyerror` (wie `yyerror` in yacc / bison).

- Mehrere Grammatikregeln können zu einer einzigen Funktion kombiniert werden. Dies ist eine gute Idee, wenn Produktionen eine ähnliche Struktur haben.

```
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]
```

- Zeichenliterale können anstelle von Token verwendet werden.

```
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]
```

Natürlich müssen die Literale im Lexer-Modul angegeben werden.

- Leere Produktionen haben das `'''symbol : '''`
- Um das Startsymbol explizit zu setzen, verwenden Sie `start = 'foo'` , wobei `foo` ein Nicht-Terminal ist.
- Das Festlegen von Priorität und Assoziativität kann mithilfe der Prioritätsvariablen erfolgen.

```
precedence = (
    ('nonassoc', 'LESSTHAN', 'GREATERTHAN'),  # Nonassociative operators
    ('left', 'PLUS', 'MINUS'),
    ('left', 'TIMES', 'DIVIDE'),
    ('right', 'UMINUS'),            # Unary minus operator
)
```

Token werden vom niedrigsten bis zum höchsten Rang angeordnet. `nonassoc` bedeutet, dass diese Tokens nicht assoziieren. Dies bedeutet, dass so etwas wie `a < b < c` illegal ist, während `a < b` noch legal ist.

- `parser.out` ist eine Debugging-Datei, die erstellt wird, wenn das Programm yacc zum ersten Mal ausgeführt wird. Immer wenn ein Verschiebungs- / Reduzierungskonflikt auftritt, verschiebt sich der Parser immer.

Python Lex-Yacc online lesen: https://riptutorial.com/de/python/topic/10510/python-lex-yacc

Kapitel 133: Python mit SQL Server verbinden

Examples

Verbindung zum Server herstellen, Tabelle erstellen, Abfragedaten

Installieren Sie das Paket:

```
$ 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()
```

Sie können alles tun, wenn Ihre Arbeit mit SQL-Ausdrücken zusammenhängt. Übergeben Sie diese Ausdrücke einfach an die Ausführungsmethode (CRUD-Operationen).

Für mit aufruf gespeicherte Prozedur, Fehlerbehandlung oder weitere Beispielprüfung: pymssql.org

Python mit SQL Server verbinden online lesen: https://riptutorial.com/de/python/topic/7985/python-mit-sql-server-verbinden

Kapitel 134: Python Parallelität

Bemerkungen

Die Python-Entwickler stellten sicher, dass die API zwischen `threading` und `multiprocessing` ähnlich ist, sodass der Wechsel zwischen den beiden Varianten für Programmierer einfacher ist.

Examples

Das Einfädelmodul

```
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()
```

Bei bestimmten Implementierungen von Python wie CPython ist wahr Parallelität nicht wegen der Verwendung , was als die GIL bekannt ist die Verwendung von Threads erreicht, oder **G** LOBALE **I** nterpreter **L** ock.

Hier ist ein hervorragender Überblick über die Parallelität von Python:

Python-Parallelität von David Beazley (YouTube)

Das Multiprocessing-Modul

```
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()
```

Hier wird jede Funktion in einem neuen Prozess ausgeführt. Da eine neue Instanz von Python VM den Code `GIL` , gibt es keine `GIL` und Sie können Parallelität auf mehreren Kernen ausführen.

Die `Process.start` Methode startet diesen neuen Prozess und führen Sie die Funktion in dem übergebenen `target` Argumente mit den Argumenten `args` . Die `Process.join` Methode wartet auf das Ende der Ausführung der Prozesse `p1` und `p2` .

Die neuen Prozesse werden gestartet unterschiedlich , je nach Version von Python und der entsprechenden Plattform , auf die der Code zum *Beispiel* ausgeführt wird :

- Windows verwendet `spawn` , um den neuen Prozess zu erstellen.
- Bei Unix-Systemen und Versionen vor 3.3 werden die Prozesse mithilfe einer `fork` . Beachten Sie, dass diese Methode die POSIX-Verwendung von Fork nicht beachtet und daher zu unerwartetem Verhalten führt, insbesondere wenn Sie mit anderen Multiprocessing-Bibliotheken interagieren.
- Mit Unix-System und Version 3.4 und höher können Sie die neuen Prozesse entweder mit `fork` , `forkserver` oder `spawn` Verwendung von `multiprocessing.set_start_method` am Anfang Ihres Programms starten. `forkserver` und `spawn` methoden sind langsamer als forking, vermeiden jedoch unerwartetes Verhalten.

POSIX-Gabelverwendung :

> Nach einer Abzweigung in einem Multithread-Programm kann das Kind nur sicherheitsgerichtete asynchrone Funktionen aufrufen, bis es ausgeführt wird.
> (siehe)

Mit Fork wird ein neuer Prozess mit demselben Status für den gesamten aktuellen Mutex gestartet, aber nur der `MainThread` wird gestartet. Dies ist unsicher, da dies zu Rennbedingungen führen kann, *z* .

- Wenn Sie eine `Lock` in `MainThread` und sie an einen anderen Thread übergeben, der sie an einem bestimmten Punkt sperren soll. Wenn die `fork` gleichzeitig occures, wird das neue Verfahren mit einem gesperrten Schloss beginnen, die nie als der zweite Thread freigegeben wird nicht in diesem neuen Verfahren existiert.

Eigentlich sollte diese Art von Verhalten nicht in reinem Python auftreten, da `multiprocessing` dies richtig handhabt. Wenn Sie jedoch mit anderen Bibliotheken interagieren, kann diese Art von Verhalten auftreten, was zum Absturz Ihres Systems führen kann (z. B. mit numpy / beschleunigt auf macOS).

Übergabe von Daten zwischen Multiprozessoren

Da die Daten zwischen zwei Threads vertraulich behandelt werden (wenn man annimmt, dass gleichzeitiges Lesen und gleichzeitiges Schreiben Konflikte verursachen können, was zu Race-Bedingungen führt), wurde eine Reihe von eindeutigen Objekten erstellt, um das Weiterleiten von Daten zwischen Threads zu erleichtern. Jede wirklich atomare Operation kann zwischen Threads

verwendet werden, aber es ist immer sicher, bei Queue zu bleiben.

```
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
```

Die meisten Leute schlagen vor, bei der Verwendung der Warteschlange immer die Warteschlangendaten in einem try: zu platzieren, außer: blockieren statt leer zu verwenden. Bei Anwendungen, bei denen es nicht darauf ankommt, einen Scan-Zyklus zu überspringen (Daten können in die Warteschlange gestellt werden, während der `queue.Empty==True` aus der Warteschlange `queue.Empty==True` in die Warteschlange. `queue.Empty==False`), ist es in der Regel besser, read zu platzieren und Schreibzugriff auf das, was ich als Iftry-Block bezeichne, da eine 'if'-Anweisung technisch performanter ist als die Ausnahme.

```
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
```

Python Parallelität online lesen: https://riptutorial.com/de/python/topic/3357/python-parallelitat

Kapitel 135: Python Requests Post

Einführung

Dokumentation für das Python-Requests-Modul im Kontext der HTTP-POST-Methode und der entsprechenden Requests-Funktion

Examples

Einfacher Beitrag

```
from requests import post

foo = post('http://httpbin.org/post', data = {'key':'value'})
```

Führt eine einfache HTTP-POST-Operation aus. Gebuchte Daten können in fast allen Formaten vorliegen, jedoch sind Schlüsselwertpaare am häufigsten.

Kopfzeilen

Header können angezeigt werden:

```
print(foo.headers)
```

Eine Beispielantwort:

```
{'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'}
```

Header können auch vor dem Post vorbereitet werden:

```
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'})
```

Codierung

Die Kodierung kann auf die gleiche Weise festgelegt und angezeigt werden:

```
 print(foo.encoding)

'utf-8'

foo.encoding = 'ISO-8859-1'
```

SSL-Überprüfung

Bei Anfragen werden standardmäßig SSL-Zertifikate von Domänen überprüft. Dies kann überschrieben werden:

```
foo = post('http://httpbin.org/post', data = {'key':'value'}, verify=False)
```

Umleitung

Bei jeder Umleitung (zB http zu https) kann dies ebenfalls geändert werden:

```
foo = post('http://httpbin.org/post', data = {'key':'value'}, allow_redirects=False)
```

Wenn die Nachoperation umgeleitet wurde, kann auf diesen Wert zugegriffen werden:

```
print(foo.url)
```

Eine vollständige Geschichte von Weiterleitungen kann angezeigt werden:

```
print(foo.history)
```

Formularcodierte Daten

```
from requests import post

payload = {'key1' : 'value1',
           'key2' : 'value2'
           }

foo = post('http://httpbin.org/post', data=payload)
```

Um formcodierte Daten mit der Nachoperation zu übergeben, müssen Daten als Wörterbuch strukturiert und als Datenparameter bereitgestellt werden.

Wenn die Daten nicht formcodiert werden sollen, übergeben Sie einfach eine Zeichenfolge oder eine Ganzzahl an den Datenparameter.

Geben Sie das Wörterbuch an den Parameter json für Requests, um die Daten automatisch zu formatieren:

```
from requests import post

payload = {'key1' : 'value1', 'key2' : 'value2'}
```

```
foo = post('http://httpbin.org/post', json=payload)
```

Datei-Upload

Mit dem Requests-Modul muss lediglich ein `.read()` im Gegensatz zu den mit `.read()` abgerufenen `.read()` :

```
from requests import post

files = {'file' : open('data.txt', 'rb')}

foo = post('http://http.org/post', files=files)
```

Dateiname, Inhaltstyp und Kopfzeilen können ebenfalls festgelegt werden:

```
files = {'file': ('report.xls', open('report.xls', 'rb'), 'application/vnd.ms-excel',
{'Expires': '0'})}

foo = requests.post('http://httpbin.org/post', files=files)
```

Strings können auch als Datei gesendet werden, sofern sie als `files` Parameter angegeben werden.

Mehrere Dateien

Mehrere Dateien können ähnlich wie eine Datei bereitgestellt werden:

```
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)
```

Antworten

Antwortcodes können in einer Nachoperation angezeigt werden:

```
from requests import post

foo = post('http://httpbin.org/post', data={'data' : 'value'})
print(foo.status_code)
```

Zurückgegebene Daten

Zugriff auf zurückgegebene Daten:

```
foo = post('http://httpbin.org/post', data={'data' : 'value'})
print(foo.text)
```

Rohe Antworten

In den Fällen, in denen Sie auf das zugrunde liegende urllib3 response.HTTPResponse-Objekt zugreifen müssen, kann dies folgendermaßen geschehen:

```
foo = post('http://httpbin.org/post', data={'data' : 'value'})
res = foo.raw

print(res.read())
```

Authentifizierung

Einfache HTTP-Authentifizierung

Die einfache HTTP-Authentifizierung kann mit folgendem erreicht werden:

```
from requests import post

foo = post('http://natas0.natas.labs.overthewire.org', auth=('natas0', 'natas0'))
```

Dies ist eine technisch kurze Hand für Folgendes:

```
from requests import post
from requests.auth import HTTPBasicAuth

foo = post('http://natas0.natas.labs.overthewire.org', auth=HTTPBasicAuth('natas0', 'natas0'))
```

HTTP-Digest-Authentifizierung

Die HTTP-Digest-Authentifizierung erfolgt auf sehr ähnliche Weise. Requests bietet hierfür ein anderes Objekt:

```
from requests import post
from requests.auth import HTTPDigestAuth

foo = post('http://natas0.natas.labs.overthewire.org', auth=HTTPDigestAuth('natas0',
'natas0'))
```

Benutzerdefinierte Authentifizierung

In einigen Fällen reichen die integrierten Authentifizierungsmechanismen möglicherweise nicht aus. Stellen Sie sich folgendes Beispiel vor:

Ein Server ist so konfiguriert, dass er die Authentifizierung akzeptiert, wenn der Absender über die richtige Benutzeragentenzeichenfolge und einen bestimmten Header-Wert verfügt und die richtigen Anmeldeinformationen über die HTTP-Basisauthentifizierung bereitstellt. Um dies zu erreichen, muss eine benutzerdefinierte Authentifizierungsklasse erstellt werden, die AuthBase, die Basis für die Implementierung der Anforderungsauthentifizierung, darstellt:

```
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
```

Dies kann dann mit dem folgenden Code verwendet werden:

```
foo = get('http://test.com/admin', auth=CustomAuth('SecretHeader', 'CustomUserAgent', 'user',
'password' ))
```

Proxies

Jeder Anforderungs-POST-Vorgang kann für die Verwendung von Netzwerk-Proxys konfiguriert werden

HTTP / S-Proxies

```
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)
```

Die HTTP-Basisauthentifizierung kann auf folgende Weise bereitgestellt werden:

```
proxies = {'http': 'http://user:pass@192.168.0.128:312'}
foo = requests.post('http://httpbin.org/post', proxies=proxies)
```

SOCKS Proxies

Die Verwendung von Socken-Proxys erfordert Abhängigkeiten von Drittanbietern `requests[socks]` . Sobald Socken-Proxies installiert sind, werden sie auf ähnliche Weise wie HTTPBasicAuth verwendet:

```
proxies = {
'http': 'socks5://user:pass@host:port',
'https': 'socks5://user:pass@host:port'
}
```

```
foo = requests.post('http://httpbin.org/post', proxies=proxies)
```

Python Requests Post online lesen: https://riptutorial.com/de/python/topic/10021/python-requests-post

Kapitel 136: Python serielle Kommunikation (pyserial)

Syntax

- ser.read (Größe = 1)
- ser.readline ()
- ser.write ()

Parameter

Parameter	Einzelheiten
Hafen	Gerätename zB / dev / ttyUSB0 unter GNU / Linux oder COM3 unter Windows.
Baudrate	Baudratentyp: int default: 9600 Standardwerte: 50, 75, 110, 134, 150, 200, 300, 600, 1200, 1800, 2400, 4800, 9600, 19200, 38400, 57600, 115200

Bemerkungen

Weitere Informationen finden Sie in der Dokumentation zu Ihrem System

Examples

Initialisieren Sie das serielle Gerät

```
import serial
#Serial takes these two parameters: serial device and baudrate
ser = serial.Serial('/dev/ttyUSB0', 9600)
```

Vom seriellen Port lesen

Initialisieren Sie das serielle Gerät

```
import serial
#Serial takes two parameters: serial device and baudrate
ser = serial.Serial('/dev/ttyUSB0', 9600)
```

Einzelbyte vom seriellen Gerät lesen

```
data = ser.read()
```

um die angegebene Anzahl von Bytes vom seriellen Gerät zu lesen

```
data = ser.read(size=5)
```

eine Zeile vom seriellen Gerät lesen.

```
data = ser.readline()
```

um die Daten vom seriellen Gerät zu lesen, während etwas darüber geschrieben wird.

```
#for python2.7
data = ser.read(ser.inWaiting())

#for python3
ser.read(ser.inWaiting)
```

Prüfen Sie, welche seriellen Anschlüsse auf Ihrem Computer verfügbar sind

Um eine Liste der verfügbaren seriellen Anschlüsse zu erhalten, verwenden Sie

```
python -m serial.tools.list_ports
```

an einer Eingabeaufforderung oder

```
from serial.tools import list_ports
list_ports.comports()  # Outputs list of available serial ports
```

aus der Python-Shell.

Python serielle Kommunikation (pyserial) online lesen:
https://riptutorial.com/de/python/topic/5744/python-serielle-kommunikation--pyserial-

Kapitel 137: Python Server - Gesendete Ereignisse

Einführung

Server Sent Events (SSE) ist eine unidirektionale Verbindung zwischen einem Server und einem Client (normalerweise ein Webbrowser), über die der Server Informationen an den Client "pushen" kann. Es ist wie Websockets und lange Abfragen. Der Hauptunterschied zwischen SSE und Websockets besteht darin, dass SSE unidirektional ist. Nur der Server kann Informationen an den Client senden, wobei wie bei Websockets beide Informationen an einander senden können. SSE wird in der Regel als wesentlich einfacher zu verwenden / zu implementieren als Websockets.

Examples

Flasche 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

In diesem Beispiel wird die asyncio-SSE-Bibliothek verwendet:
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()
```

Python Server - Gesendete Ereignisse online lesen:
https://riptutorial.com/de/python/topic/9100/python-server---gesendete-ereignisse

Kapitel 138: Python und Excel

Examples

Geben Sie Listendaten in eine Excel-Datei ein.

```
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 ist ein Modul zum Bearbeiten und Erstellen von `xlsx/xlsm/xltx/xltm` im Arbeitsspeicher.

Bearbeiten und Lesen einer vorhandenen Arbeitsmappe:

```
import openpyxl as opx
#To change an existing wookbook we located it by referencing its path
workbook = opx.load_workbook(workbook_path)
```

`load_workbook()` enthält den Parameter `read_only` hier `True` `read_only` , wird die Arbeitsmappe als read_only geladen. Dies ist hilfreich, wenn Sie größere `xlsx` Dateien `xlsx` :

```
workbook = opx.load_workbook(workbook_path, read_only=True)
```

Nachdem Sie die Arbeitsmappe in den Arbeitsspeicher geladen haben, können Sie mithilfe von `workbook.sheets` auf die einzelnen `workbook.sheets` zugreifen

```
first_sheet = workbook.worksheets[0]
```

Wenn Sie den Namen eines verfügbaren `workbook.get_sheet_names()` angeben möchten, können Sie `workbook.get_sheet_names()` .

```
sheet = workbook.get_sheet_by_name('Sheet Name')
```

Zum Schluss kann auf die Zeilen des `sheet.rows` Hilfe von `sheet.rows` zugegriffen werden. Um die Zeilen in einem Arbeitsblatt zu durchlaufen, verwenden Sie:

```
for row in sheet.rows:
    print row[0].value
```

Da jede `row` in `rows` eine Liste von `Cell` s ist, verwenden Sie `Cell.value` , um den Inhalt der Cell `Cell.value` .

Erstellen einer neuen Arbeitsmappe im Arbeitsspeicher:

```
#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
```

Mehrere Eigenschaften von Registerkarten können durch openpyxl geändert werden, z. B. `tabColor` :

```
ws.sheet_properties.tabColor = 'FFC0CB'
```

Um unsere erstellte Arbeitsmappe zu speichern, beenden wir mit:

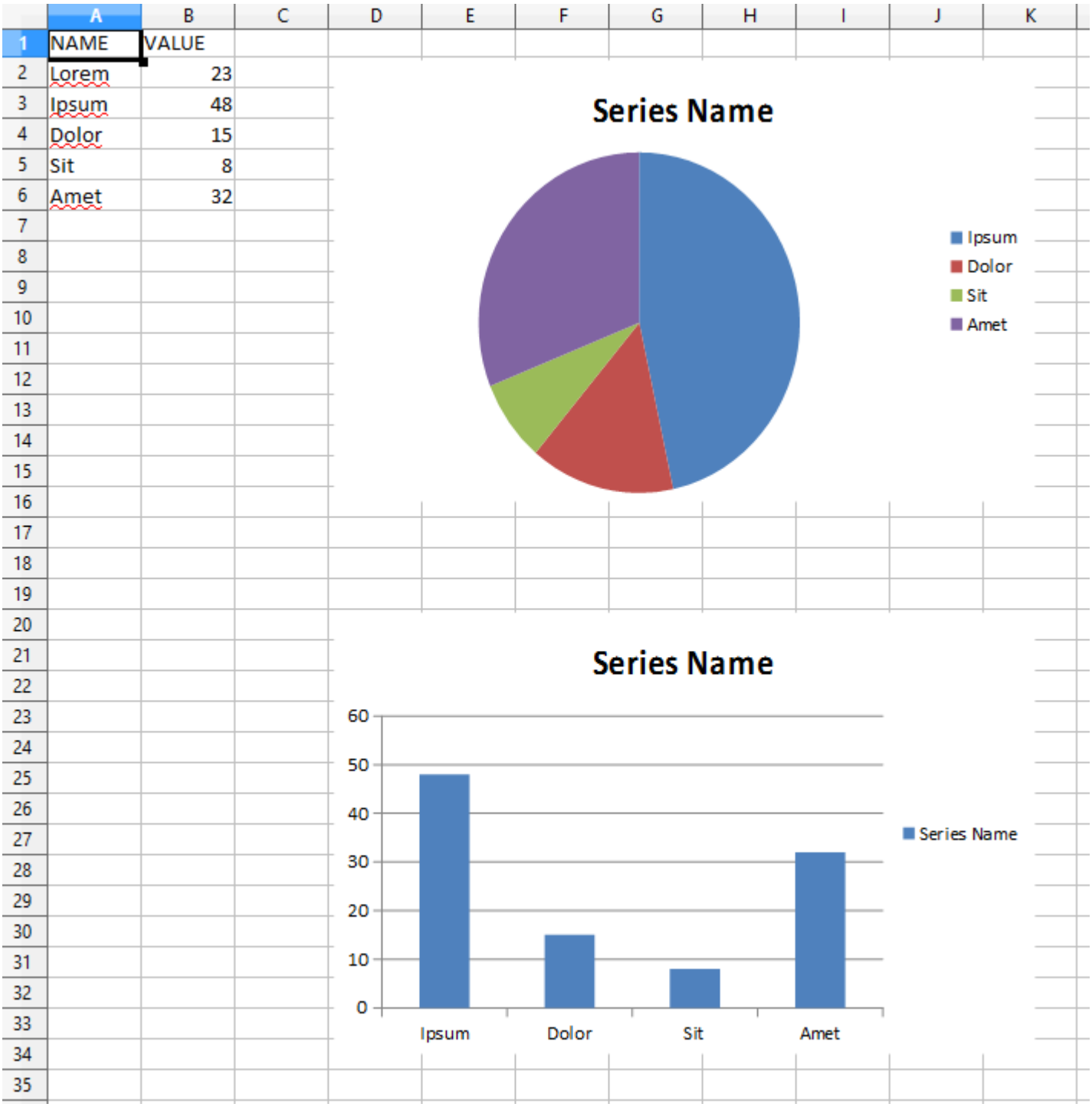
```
wb.save('filename.xlsx')
```

Erstellen Sie mit xlsxwriter Excel-Diagramme

```
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()
```

Ergebnis:



Lesen Sie die Excel-Daten mit dem xlrd-Modul

Die Python xlrd-Bibliothek dient zum Extrahieren von Daten aus Microsoft Excel (tm) -Tabellen.

Installation:-

```
pip install xlrd
```

Oder Sie können die Datei setup.py von pypi verwenden

https://pypi.python.org/pypi/xlrd

Lesen einer Excel- Tabelle **: -** Importieren Sie das xlrd-Modul und öffnen Sie die Excel-Datei mit der Methode open_workbook ().

```
import xlrd
book=xlrd.open_workbook('sample.xlsx')
```

Überprüfen Sie die Anzahl der Blätter in Excel

```
print book.nsheets
```

Drucken Sie die Blattnamen

```
print book.sheet_names()
```

Holen Sie sich das Blatt basierend auf Index

```
sheet=book.sheet_by_index(1)
```

Lesen Sie den Inhalt einer Zelle

```
cell = sheet.cell(row,col) #where row=row number and col=column number
print cell.value #to print the cell contents
```

Rufen Sie die Anzahl der Zeilen und die Anzahl der Spalten in einem Excel-Arbeitsblatt ab

```
num_rows=sheet.nrows
num_col=sheet.ncols
```

Holen Sie sich Excel-Tabelle nach Namen

```
sheets = book.sheet_names()
cur_sheet = book.sheet_by_name(sheets[0])
```

Formatieren Sie Excel-Dateien mit 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()
```

Python und Excel online lesen: https://riptutorial.com/de/python/topic/2986/python-und-excel

Kapitel 139: Python-Anti-Patterns

Examples

Übereifrig mit Ausnahme der Klausel

Ausnahmen sind mächtig, aber eine einzige übereifrige Ausnahme-Klausel kann alles in einer einzigen Zeile wegnehmen.

```
try:
    res = get_result()
    res = res[0]
    log('got result: %r' % res)
except:
    if not res:
        res = ''
    print('got exception')
```

Dieses Beispiel zeigt 3 Symptome des Antipattern:

1. Die `except` ohne Ausnahmetyp (Zeile 5) fängt sogar fehlerhafte Ausnahmen auf, einschließlich `KeyboardInterrupt` . Dadurch wird das Programm in einigen Fällen möglicherweise nicht beendet.
2. Der Ausnahme-Block erhöht den Fehler nicht erneut, was bedeutet, dass wir nicht feststellen können, ob die Ausnahme innerhalb von `get_result` oder weil `res` eine leere Liste war.
3. Am schlimmsten: Wenn wir uns Sorgen machten, dass das Ergebnis leer wäre, haben wir etwas Schlimmeres verursacht. Wenn `get_result` fehlschlägt, bleibt `res` vollständig unset und der Verweis auf `res` im außer -Block `NameError` , dass `NameError` und der ursprüngliche Fehler vollständig maskiert wird.

Denken Sie immer an die Art der Ausnahme, mit der Sie umgehen möchten. Lesen Sie die Ausnahmeseite und erfahren Sie, welche grundlegenden Ausnahmen bestehen.

Hier ist eine feste Version des obigen Beispiels:

```
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)
```

Wir stellen spezifischere Ausnahmen fest und reraisen wenn nötig. Ein paar Zeilen mehr, aber

unendlich viel korrekter.

Schauen Sie, bevor Sie mit einer prozessorintensiven Funktion springen

Ein Programm kann leicht Zeit verschwenden, indem eine prozessorintensive Funktion mehrmals aufgerufen wird.

Nehmen wir zum Beispiel eine Funktion , die wie folgt aussieht: es eine ganze Zahl zurück , wenn der `value` ein, sonst produzieren kann `None` :

```
def intensive_f(value): # int -> Optional[int]
   # complex, and time-consuming code
   if process_has_failed:
       return None
   return integer_output
```

Und es könnte auf folgende Weise verwendet werden:

```
x = 5
if intensive_f(x) is not None:
    print(intensive_f(x) / 2)
else:
    print(x, "could not be processed")

print(x)
```

Während dies funktioniert, hat es das Problem, `intensive_f` aufzurufen, wodurch sich die Zeitdauer für die Ausführung des Codes verdoppelt. Eine bessere Lösung wäre, den Rückgabewert der Funktion vorher zu erhalten.

```
x = 5
result = intensive_f(x)
if result is not None:
    print(result / 2)
else:
    print(x, "could not be processed")
```

Ein klarer und möglicherweise pythonischerer Weg ist jedoch die Verwendung von Ausnahmen, zum Beispiel:

```
x = 5
try:
    print(intensive_f(x) / 2)
except TypeError: # The exception raised if None + 1 is attempted
    print(x, "could not be processed")
```

Hier wird keine temporäre Variable benötigt. Es kann oft vorzuziehen sein, eine `assert` Anweisung zu verwenden und stattdessen `AssertionError` abzufangen.

Wörterbuchschlüssel

Ein häufiges Beispiel dafür ist der Zugriff auf Wörterbuchschlüssel. Zum Beispiel vergleichen Sie:

```
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)
```

mit:

```
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)
```

Im ersten Beispiel muss das Wörterbuch zweimal durchgesehen werden. Da es sich um ein langes Wörterbuch handelt, kann dies jedes Mal sehr lange dauern. Die zweite erfordert nur eine Suche durch das Wörterbuch und spart somit viel Prozessorzeit.

Eine Alternative dazu ist die Verwendung von `dict.get(key, default)` *. In vielen Fällen können jedoch komplexere Vorgänge erforderlich sein, wenn der Schlüssel nicht vorhanden ist*

Python-Anti-Patterns online lesen: https://riptutorial.com/de/python/topic/4700/python-anti-patterns

Kapitel 140: Python-Datentypen

Einführung

Datentypen sind nichts anderes als Variablen, mit denen Sie etwas Speicherplatz reservieren. Python-Variablen benötigen keine explizite Deklaration, um Speicherplatz zu reservieren. Die Deklaration erfolgt automatisch, wenn Sie einer Variablen einen Wert zuweisen.

Examples

Datentyp Zahlen

Zahlen haben vier Arten in Python. Int, Float, komplex und lang.

```
int_num = 10    #int value
float_num = 10.2    #float value
complex_num = 3.14j    #complex value
long_num = 1234567L    #long value
```

String-Datentyp

Zeichenfolge wird als eine zusammenhängende Menge von Zeichen identifiziert, die in den Anführungszeichen dargestellt werden. Python erlaubt entweder einfache oder doppelte Anführungszeichen. Strings sind unveränderliche Sequenzdatentypen, dh bei jeder Änderung eines Strings wird ein völlig neues String-Objekt erstellt.

```
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
```

List Datentyp

Eine Liste enthält Elemente, die durch Kommas getrennt und in eckige Klammern [] eingeschlossen sind. Listen ähneln den Arrays in C. Ein Unterschied besteht darin, dass alle Elemente, die zu einer Liste gehören, einen anderen Datentyp haben können.

```
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']
```

Tupel-Datentyp

Listen sind in Klammern [] eingeschlossen und ihre Elemente und Größe können geändert werden, während Tupel in Klammern () eingeschlossen sind und nicht aktualisiert werden können. Tupel sind unveränderlich.

```
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.
```

Wörterbuch-Datentyp

Das Wörterbuch besteht aus Schlüssel-Wert-Paaren. Es wird von geschweiften Klammern {} eingeschlossen und die Werte können mit eckigen Klammern [] zugewiesen und aufgerufen werden.

```
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']
```

Datentypen festlegen

Sets sind ungeordnete Sammlungen eindeutiger Objekte. Es gibt zwei Arten von Sets:

1. Sets - Sie sind veränderbar, und neue Elemente können hinzugefügt werden, sobald Sets definiert sind

   ```
   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. Gefrorene Sets - Sie sind unveränderlich und neue Elemente können nach ihrer Definition nicht hinzugefügt werden.

   ```
   b = frozenset('asdfagsa')
   print(b)
   > frozenset({'f', 'g', 'd', 'a', 's'})
   cities = frozenset(["Frankfurt", "Basel","Freiburg"])
   print(cities)
   > frozenset({'Frankfurt', 'Basel', 'Freiburg'})
   ```

Python-Datentypen online lesen: https://riptutorial.com/de/python/topic/9366/python-datentypen

Kapitel 141: Python-Geschwindigkeit des Programms

Examples

Notation

Die Grundidee

Die Notation, die zur Beschreibung der Geschwindigkeit Ihres Python-Programms verwendet wird, wird als Big-O-Notation bezeichnet. Nehmen wir an, Sie haben eine Funktion:

```
def list_check(to_check, the_list):
    for item in the_list:
        if to_check == item:
          return True
    return False
```

Dies ist eine einfache Funktion, um zu überprüfen, ob sich ein Element in einer Liste befindet. Um die Komplexität dieser Funktion zu beschreiben, sagen Sie O (n). Dies bedeutet "Order of n", da die O-Funktion als Order-Funktion bezeichnet wird.

O (n) - im Allgemeinen ist n die Anzahl der Artikel im Container

O (k) - im Allgemeinen ist k der Wert des Parameters oder die Anzahl der Elemente im Parameter

Operationen auflisten

Operations: Average Case (setzt voraus, dass die Parameter zufällig generiert werden)

Anhängen: O (1)

Kopie: O (n)

Del Slice: O (n)

Element löschen: O (n)

Einfügen: O (n)

Gegenstand erhalten: O (1)

Setzposten: O (1)

Iteration: O (n)

Slice erhalten: O (k)

Slice einstellen: O (n + k)

Erweitern: O (k)

Sortierung: O (n log n)

Multiplizieren Sie: O (nk)

x in s: O (n)

min (s), max (s): O (n)

Länge erhalten: O (1)

Deque-Operationen

Eine Deque ist eine doppelseitige Warteschlange.

```
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)
```

Operations: Average Case (setzt voraus, dass die Parameter zufällig generiert werden)

Anhängen: O (1)

Anhang: O (1)

Kopie: O (n)

Erweitern: O (k)

Linkshöhe: O (k)

Pop: O (1)

Popleft: O (1)

Entfernen: O (n)

Drehen: O (k)

Operationen einstellen

Operation: Average Case (unterstellt, dass die Parameter zufällig generiert werden): Worst case

x in s: O (1)

Unterschied s - t: O (len (s))

Schnittpunkt s & t: O (min (len (s), len (t))): O (len (s) * len (t)

Mehrere Schnittpunkte s1 & s2 & s3 & ... & sn:: (n-1) * O (l) wobei l max ist (len (s1), ..., len (sn))

abweichungsupdate (t): O (len (t)): O (len (t) * len (s))

s.symetric_difference_update (t): O (len (t))

Symmetrische Differenz s ^ t: O (len (s)): O (len (s) * len (t))

Union s | t: O (len (s) + len (t))

Algorithmische Notationen ...

Es gibt bestimmte Prinzipien, die für die Optimierung in jeder Computersprache gelten, und Python ist keine Ausnahme. **Optimieren Sie nicht wie Sie** : Schreiben Sie Ihr Programm ohne Berücksichtigung möglicher Optimierungen und konzentrieren Sie sich stattdessen darauf, dass der Code sauber, korrekt und verständlich ist. Wenn es zu groß oder zu langsam ist, können Sie es in Betracht ziehen, es zu optimieren.

Denken Sie an die 80/20-Regel : In vielen Bereichen können Sie mit 20% des Aufwands 80% des Ergebnisses erzielen (auch als 90/10-Regel bezeichnet - abhängig davon, mit wem Sie sprechen). Wann immer Sie Code optimieren möchten, verwenden Sie die Profilerstellung, um herauszufinden, wo 80% der Ausführungszeit liegen. So wissen Sie, wo Sie Ihre Anstrengungen konzentrieren müssen.

Führen Sie immer "vor" und "nach" Benchmarks aus : Wie wissen Sie sonst noch, dass Ihre Optimierungen tatsächlich einen Unterschied gemacht haben? Wenn sich herausstellt, dass Ihr optimierter Code nur geringfügig schneller oder kleiner als die Originalversion ist, machen Sie Ihre Änderungen rückgängig und kehren Sie zum ursprünglichen Code zurück.

Verwenden Sie die richtigen Algorithmen und Datenstrukturen: Verwenden Sie keinen O (n2) - Blasensortieralgorithmus, um tausend Elemente zu sortieren, wenn ein O (n log n) - Quicksort verfügbar ist. Speichern Sie auf keinen Fall eintausend Elemente in einem Array, für das eine O (n) -Suche erforderlich ist, wenn Sie einen binären O (log n) -Baum oder eine O (1) -Python-Hashtabelle verwenden könnten.

Für weitere Informationen besuchen Sie den folgenden Link ... Python Speed Up

Die folgenden 3 asymptotischen Notationen werden meist zur Darstellung der zeitlichen Komplexität von Algorithmen verwendet.

1. **Θ Notation** : Die Theta-Notation begrenzt Funktionen von oben und unten und definiert somit genau asymptotisches Verhalten. Eine einfache Möglichkeit, die Theta-Notation eines Ausdrucks zu erhalten, besteht darin, niederwertige Terme zu löschen und führende Konstanten zu ignorieren. Betrachten Sie zum Beispiel den folgenden Ausdruck. 3n3 + 6n2 + 6000 = Θ (n3) Das Ablegen von Termen niedrigerer Ordnung ist immer in Ordnung, da es immer ein n0 gibt, nach dem Θ (n3) unabhängig von den beteiligten Konstanten höhere Werte als Θn2 hat. Für eine gegebene Funktion g (n) bezeichnen wir Θ (g (n)) als Funktionssatz. (G (n)) = {f (n): Es gibt positive Konstanten c1, c2 und n0, so dass 0 <= c1 *g (n)* <= *f (n)* <= *c2* g (n) für alle n> ist = n0} Die obige Definition bedeutet, wenn f (n) theta von g (n) ist, dann liegt der Wert f (n) immer zwischen c1 *g (n) und c2* g (n) für große Werte von n (n> = n0). Die Definition von Theta erfordert auch, dass f (n) für Werte von n größer als n0 nicht negativ sein darf.

2. **Big-O-Notation** : Die Big-O-Notation definiert eine obere Grenze eines Algorithmus, sie begrenzt eine Funktion nur von oben. Betrachten Sie beispielsweise den Fall Einfügungssortierung. Im besten Fall dauert es linear und im schlechtesten Fall quadratisch. Wir können mit Sicherheit sagen, dass die zeitliche Komplexität der Einfügungssortierung O (n ^ 2) ist. Beachten Sie, dass O (n ^ 2) auch die lineare Zeit abdeckt. Wenn wir die complexity-Notation verwenden, um die zeitliche Komplexität der Einfügungssortierung darzustellen, müssen wir zwei Anweisungen für den besten und den schlechtesten Fall verwenden:

1. Die ungünstigste zeitliche Komplexität von Insertion Sort ist Θ (n ^ 2).
2. Die beste zeitliche Komplexität von Insertion Sort ist Θ (n).

Die Big-O-Notation ist nützlich, wenn die Komplexität eines Algorithmus nur mit der Zeit begrenzt ist. Oft finden wir leicht eine obere Grenze, indem wir einfach den Algorithmus betrachten. O (g (n)) = {f (n): Es gibt positive Konstanten c und n0, so dass 0 <= f (n) <= cg (n) für alle n> = n0}

3. **Ω-Notation** : So wie die Big-O-Notation eine asymptotische Obergrenze für eine Funktion bereitstellt, stellt die Ω-Notation eine asymptotische Untergrenze bereit. Ω Notation <kann nützlich sein, wenn die Komplexität eines Algorithmus niedriger ist. Wie im vorherigen Post diskutiert, ist die beste Leistung eines Algorithmus im Allgemeinen nicht nützlich. Die Omega-Notation ist die am wenigsten verwendete Notation unter allen drei. Für eine gegebene Funktion g (n) bezeichnen wir die Menge der Funktionen mit Ω (g (n)). Ω (g (n)) = {f (n): Es gibt positive Konstanten c und n0, so dass 0 <= cg (n) <= f (n) für alle n> = n0} ist. Betrachten wir hier dasselbe Einfügungssortierbeispiel. Die zeitliche Komplexität der Einfügungssortierung kann als Ω (n) geschrieben werden, ist jedoch keine sehr nützliche Information über die Einfügungssortierung, da wir uns im Allgemeinen für den ungünstigsten Fall und manchmal für den Durchschnittsfall interessieren.

Python-Geschwindigkeit des Programms online lesen:
https://riptutorial.com/de/python/topic/9185/python-geschwindigkeit-des-programms

Kapitel 142: Python-HTTP-Server

Examples

Einen einfachen HTTP-Server ausführen

Python 2.x 2.3

```
python -m SimpleHTTPServer 9000
```

Python 3.x 3.0

```
python -m http.server 9000
```

Wenn Sie diesen Befehl ausführen, werden die Dateien des aktuellen Verzeichnisses an Port `9000` .

Wenn kein Argument als Portnummer angegeben ist, wird der Server auf dem Standardport `8000` .

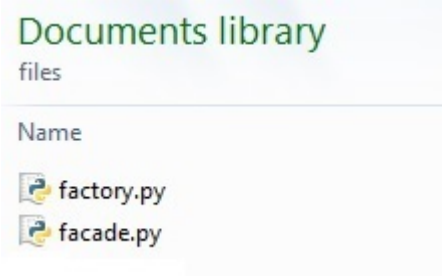
Das `-m` Flag durchsucht `sys.path` nach der entsprechenden `.py` Datei, die als Modul ausgeführt werden soll.

Wenn Sie nur auf localhost dienen möchten, müssen Sie ein benutzerdefiniertes Python-Programm schreiben, z.

```
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()
```

Dateien bereitstellen

Angenommen, Sie haben das folgende Dateiverzeichnis:



Sie können einen Webserver für die Bereitstellung dieser Dateien wie folgt einrichten:

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()
```

Das `SocketServer` Modul stellt die Klassen und Funktionen zum Einrichten eines Netzwerkservers bereit.

`SocketServer` -Klasse von `TCPServer` richtet einen Server mit dem TCP-Protokoll ein. Der Konstruktor akzeptiert ein Tupel, das die Adresse des Servers (dh die IP-Adresse und den Port) und die Klasse darstellt, die die Serveranforderungen verarbeitet.

Mit der `SimpleHTTPRequestHandler` Klasse des `SimpleHTTPServer` Moduls können die Dateien im aktuellen Verzeichnis `SimpleHTTPServer` werden.

Speichern Sie das Skript im selben Verzeichnis und führen Sie es aus.

Führen Sie den HTTP-Server aus:

Python 2.x 2.3

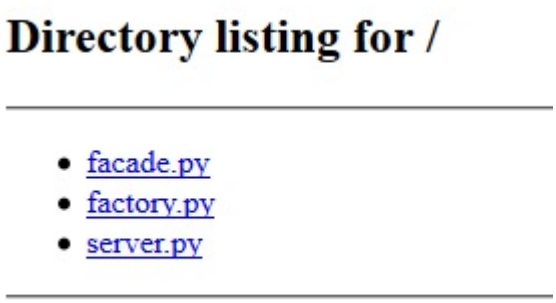
python -m SimpleHTTPServer 8000

Python 3.x 3.0

Python -m http.server 8000

Das '-m' Flag sucht nach 'sys.path' nach der entsprechenden '.py'-Datei, um sie als Modul auszuführen.

Öffnen Sie localhost: 8000 im Browser. Sie erhalten folgende Informationen:



Programmatische API von SimpleHTTPServer

Was passiert, wenn wir `python -m SimpleHTTPServer 9000` ausführen?

Um diese Frage zu beantworten, sollten wir das Konstrukt von SimpleHTTPServer (https://hg.python.org/cpython/file/2.7/Lib/SimpleHTTPServer.py) und BaseHTTPServer (https://hg.python.org/cpython/file) verstehen /2.7/Lib/BaseHTTPServer.py) .

Zunächst ruft Python das `SimpleHTTPServer` Modul mit `9000` als Argument auf. Nun den SimpleHTTPServer-Code beobachten,

```
def test(HandlerClass = SimpleHTTPRequestHandler,
         ServerClass = BaseHTTPServer.HTTPServer):
    BaseHTTPServer.test(HandlerClass, ServerClass)


if __name__ == '__main__':
    test()
```

Die Testfunktion wird nach Anforderungshandlern und ServerClass aufgerufen. Jetzt wird BaseHTTPServer.test aufgerufen

```
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()
```

Daher wird hier die Portnummer, die der Benutzer als Argument übergeben hat, analysiert und an die Hostadresse gebunden. Weitere grundlegende Schritte der Socket-Programmierung mit gegebenem Port und Protokoll werden ausgeführt. Schließlich wird der Socket-Server gestartet.

Dies ist eine grundlegende Übersicht über die Vererbung von SocketServer-Klassen an andere Klassen:

```
    +------------+
    | BaseServer |
    +------------+
          |
          v
    +-----------+        +------------------+
    | TCPServer |------->| UnixStreamServer |
    +-----------+        +------------------+
          |
          v
    +-----------+        +--------------------+
    | UDPServer |------->| UnixDatagramServer |
    +-----------+        +--------------------+
```

Die Links https://hg.python.org/cpython/file/2.7/Lib/BaseHTTPServer.py und https://hg.python.org/cpython/file/2.7/Lib/SocketServer.py sind hilfreich für die weitere Suche Information.

Grundlegende Handhabung von GET, POST, PUT mit 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()
```

Beispielausgabe mit `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
```

Python-HTTP-Server online lesen: https://riptutorial.com/de/python/topic/4247/python-http-server

Kapitel 143: Python-Netzwerk

Bemerkungen

(Sehr) einfaches Beispiel für Python-Client-Sockets

Examples

Das einfachste Python-Socket-Client-Server-Beispiel

Serverseite:

```
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
```

Kundenseite:

```
import socket

clientsocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
clientsocket.connect(('localhost', 8089))
clientsocket.send('hello')
```

Führen Sie zuerst SocketServer.py aus und stellen Sie sicher, dass der Server zum Abhören / Empfangen von sth bereit ist. Dann sendet der Client Informationen an den Server. Nachdem der Server etw erhalten hat, wird er beendet

Einen einfachen HTTP-Server erstellen

Um Dateien gemeinsam zu nutzen oder einfache Websites (http und Javascript) in Ihrem lokalen Netzwerk zu hosten, können Sie das integrierte SimpleHTTPServer-Modul von Python verwenden. Python sollte in Ihrer Path-Variable stehen. Wechseln Sie in den Ordner, in dem sich Ihre Dateien befinden, und geben Sie Folgendes ein:

Für `python 2` :

```
$ python -m SimpleHTTPServer <portnumber>
```

Für `python 3` :

```
$ python3 -m http.server <portnumber>
```

Wenn keine Portnummer angegeben ist, ist `8000` der Standardport. Die Ausgabe wird also sein:

> HTTP wird auf 0.0.0.0 Port 8000 bereitgestellt ...

Sie können über jedes mit dem lokalen Netzwerk verbundene Gerät auf Ihre Dateien zugreifen, indem Sie `http://hostipaddress:8000/` .

`hostipaddress` ist Ihre lokale IP-Adresse, die wahrscheinlich mit `192.168.xx` beginnt.

Um das Modul zu beenden, drücken `ctrl+c`. einfach `ctrl+c`.

TCP-Server erstellen

Sie können einen TCP-Server mithilfe der `socketserver` Bibliothek `socketserver` . Hier ist ein einfacher Echoserver.

Serverseite

```
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()
```

Client-Seite

```
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` können relativ einfach einfache TCP-Server erstellt werden. Sie sollten sich jedoch bewusst sein, dass die Server standardmäßig einen einzelnen Thread haben und jeweils nur einen Client bedienen können. Wenn Sie mit mehreren Clients arbeiten möchten, instanziieren Sie stattdessen einen `ThreadingTCPServer` .

```
from socketserver import ThreadingTCPServer
...
```

```
if __name__ == '__main__':
    server = ThreadingTCPServer(('', 5000), EchoHandler)
    server.serve_forever()
```

Erstellen eines UDP-Servers

Ein UDP-Server kann einfach mit der `socketserver` Bibliothek erstellt werden.

ein einfacher Zeitserver:

```
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()
```

Testen:

```
>>> 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))
```

Starten Sie Simple HttpServer in einem Thread und öffnen Sie den Browser

Nützlich, wenn Ihr Programm unterwegs Webseiten ausgibt.

```
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))
```

Python-Netzwerk online lesen: https://riptutorial.com/de/python/topic/1309/python-netzwerk

Kapitel 144: Python-Pakete erstellen

Bemerkungen

Das pypa-Beispielprojekt enthält ein komplettes, leicht zu `setup.py` Template `setup.py` , das eine Vielzahl von Möglichkeiten zeigt, die Setup-Tools bieten.

Examples

Einführung

Für jedes Paket ist eine `setup.py` Datei erforderlich, die das Paket beschreibt.

Betrachten Sie die folgende Verzeichnisstruktur für ein einfaches Paket:

```
+-- package_name
|       |
|       +-- __init__.py
|
+-- setup.py
```

Die `__init__.py` enthält nur die Zeile `def foo(): return 100` .

Die folgende `setup.py` definiert das Paket:

```
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.
)
```

Mit virtualenv können Sie Paketinstallationen testen, ohne Ihre anderen Python-Umgebungen zu ändern :

```
$ 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
```

Hochladen in PyPI

Sobald Ihre `setup.py` voll funktionsfähig ist (siehe Einführung), ist es sehr einfach, Ihr Paket auf PyPI hochzuladen.

Richten Sie eine .pypirc-Datei ein

Diese Datei speichert Anmeldungen und Kennwörter, um Ihre Konten zu authentifizieren. Sie wird normalerweise in Ihrem Heimatverzeichnis gespeichert.

```
# .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
```

Es ist sicherer , `twine` zum Hochladen von Paketen zu verwenden. `twine` Sie daher sicher, dass die Installation erfolgt.

```
$ pip install twine
```

Registrieren und Hochladen zu testpypi (optional)

Hinweis : PyPI erlaubt das Überschreiben von hochgeladenen Paketen nicht. Daher ist es ratsam , die Bereitstellung zunächst auf einem dedizierten Testserver (z. B. testpypi) zu testen. Diese Option wird diskutiert. Betrachten Sie vor dem Hochladen ein Versionsschema für Ihr Paket, z. B. Kalenderversionierung oder semantische Versionierung .

Melden Sie sich an oder erstellen Sie ein neues Konto bei testpypi . Eine Registrierung ist nur beim ersten Mal erforderlich, obwohl eine mehrfache Registrierung nicht schädlich ist.

```
$ python setup.py register -r pypitest
```

Im Stammverzeichnis des Pakets:

```
$ twine upload dist/* -r pypitest
```

Ihr Paket sollte jetzt über Ihr Konto zugänglich sein.

Testen

Erstellen Sie eine virtuelle Testumgebung. Versuchen Sie, Ihr Paket entweder von testpypi oder PyPI zu `pip install` .

```
# 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
```

Bei Erfolg ist Ihr Paket am wenigsten importierbar. Sie können Ihre API auch testen, bevor Sie sie endgültig in PyPI hochladen. Wenn das Paket während des Tests fehlgeschlagen ist, machen Sie sich keine Sorgen. Sie können das Problem weiterhin beheben, erneut auf testpypi hochladen und erneut testen.

Registrieren und Hochladen auf PyPI

Stellen Sie sicher, dass das `twine` installiert ist:

```
$ pip install twine
```

Melden Sie sich an oder erstellen Sie ein neues Konto bei PyPI .

```
$ python setup.py register -r pypi
```

```
$ twine upload dist/*
```

Das ist es! Ihr Paket ist jetzt live .

Wenn Sie einen Fehler entdecken, laden Sie einfach eine neue Version Ihres Pakets hoch.

Dokumentation

Vergessen Sie nicht, mindestens eine Art Dokumentation für Ihr Paket beizulegen. PyPi verwendet reStructuredText als Standardformatierungssprache.

Readme

Wenn Ihr Paket keine umfangreiche Dokumentation enthält, geben Sie an, was anderen Benutzern in der Datei `README.rst` helfen kann. Wenn die Datei fertig ist, wird eine weitere benötigt, um PyPi mitzuteilen, dass die Datei angezeigt werden soll.

Erstellen `setup.cfg` Datei `setup.cfg` und `setup.cfg` diese beiden Zeilen ein:

```
[metadata]
description-file = README.rst
```

Wenn Sie versuchen, eine Markdown- Datei in Ihr Paket aufzunehmen, wird sie von PyPi als reine Textdatei ohne Formatierung gelesen.

Lizenzierung

Es ist oft mehr als willkommen, eine `LICENSE.txt` Datei mit einer der OpenSource-Lizenzen in Ihr Paket aufzunehmen, um den Benutzern mitzuteilen, ob sie Ihr Paket beispielsweise in kommerziellen Projekten verwenden können oder ob Ihr Code mit ihrer Lizenz verwendet werden kann.

In besser lesbarer Weise werden einige Lizenzen bei TL erläutert.

Paket ausführbar machen

Wenn Ihr Paket nicht nur eine Bibliothek ist, sondern einen Code enthält, der bei der Installation des Pakets entweder als Showcase oder als eigenständige Anwendung verwendet werden kann, legen Sie diesen Code in `__main__.py` Datei `__main__.py`

Legen Sie die `__main__.py` `package_name` Ordner `package_name` . Auf diese Weise können Sie es direkt von der Konsole aus ausführen:

```
python -m package_name
```

Wenn keine `__main__.py` Datei verfügbar ist, wird das Paket nicht mit diesem Befehl ausgeführt, und dieser Fehler wird gedruckt:

> Python: Kein Modul mit dem Namen `package_name.__main__`; 'package_name' ist ein Paket und kann nicht direkt ausgeführt werden.

Python-Pakete erstellen online lesen: https://riptutorial.com/de/python/topic/1381/python-pakete-erstellen

Kapitel 145: Python-Persistenz

Syntax

- pickle.dump (obj, file, protocol = keine, *, fix_imports = true)
- pickle.load (file, *, fix_imports = True, Kodierung = "ASCII", Fehler = "streng")

Parameter

Parameter	Einzelheiten
obj	Pickle-Darstellung von obj in der Objektdatei der geöffneten Datei
Protokoll	eine ganze Zahl, weist den Pickler an, das angegebene Protokoll zu verwenden, `0` -ASCII, `1` - altes Binärformat
Datei	Das Dateiargument muss über eine write () - Methode `wb` für die *dump*- Methode und das Laden der read () - Methode `rb` verfügen

Examples

Python-Persistenz

Objekte wie Zahlen, Listen, Wörterbücher, verschachtelte Strukturen und Klasseninstanzobjekte befinden sich im Speicher Ihres Computers und gehen verloren, sobald das Skript beendet ist.

Pickle speichert Daten dauerhaft in einer separaten Datei.

gebeizt Darstellung eines Objekt ist immer ein Byte Objekt in allen Fällen so eine Datei in öffnen muß `wb` Daten und speichern `rb` Daten aus Beize zu laden.

die Daten können von irgendeiner Art sein, zum Beispiel

```
data={'a':'some_value',
     'b':[9,4,7],
     'c':['some_str','another_str','spam','ham'],
     'd':{'key':'nested_dictionary'},
     }
```

Daten speichern

```
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
```

Lade Daten

```
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']}
```

Folgende Typen können gebeizt werden

1. Keine, Richtig und Falsch
2. Ganzzahlen, Fließkommazahlen, komplexe Zahlen
3. Zeichenfolgen, Bytes, Byearrays
4. Tupel, Listen, Sets und Wörterbücher, die nur wählbare Objekte enthalten
5. Funktionen, die auf der obersten Ebene eines Moduls definiert sind (mit def, nicht mit Lambda)
6. integrierte Funktionen, die auf der obersten Ebene eines Moduls definiert sind
7. Klassen, die auf der obersten Ebene eines Moduls definiert sind
8. Instanzen solcher Klassen, deren **Dikt** oder das Ergebnis des Aufrufs von **getstate** ()

Funktionsprogramm zum Speichern und Laden

Daten in und aus Datei speichern

```
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'
```

Python-Persistenz online lesen: https://riptutorial.com/de/python/topic/7810/python-persistenz

Kapitel 146: Redewendungen

Examples

Wörterbuch-Schlüsselinitialisierungen

`dict.get` Methode `dict.get` , wenn Sie nicht sicher sind, ob der Schlüssel vorhanden ist. Sie können einen Standardwert zurückgeben, wenn der Schlüssel nicht gefunden wird. Die traditionelle Methode `dict[key]` würde eine `KeyError` Ausnahme `KeyError` .

Anstatt zu tun

```
def add_student():
    try:
        students['count'] += 1
    except KeyError:
        students['count'] = 1
```

Tun

```
def add_student():
        students['count'] = students.get('count', 0) + 1
```

Variablen wechseln

Um den Wert von zwei Variablen umzuschalten, können Sie das Tupel-Auspacken verwenden.

```
x = True
y = False
x, y = y, x
x
# False
y
# True
```

Verwenden Sie die Wahrheitswertprüfung

Python konvertiert jedes Objekt implizit zu Testzwecken in einen booleschen Wert. Verwenden Sie es daher nach Möglichkeit.

```
# 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
```

Dies führt im Allgemeinen zu besser lesbarem Code und ist im Umgang mit unerwarteten Typen in der Regel viel sicherer.

Klicken Sie hier, um eine Liste der `False` Werte `False` .

Testen Sie "__main__", um unerwartete Codeausführung zu vermeiden

Es `__name__` Variable `__name__` des aufrufenden Programms zu `__name__` , bevor Sie den Code ausführen.

```
import sys

def main():
    # Your code starts here

    # Don't forget to provide a return code
    return 0

if __name__ == "__main__":
    sys.exit(main())
```

Durch die Verwendung dieses Musters wird sichergestellt, dass Ihr Code nur dann ausgeführt wird, wenn Sie dies erwarten. Wenn Sie beispielsweise Ihre Datei explizit ausführen:

```
python my_program.py
```

Der Vorteil besteht jedoch, wenn Sie sich dafür entscheiden, Ihre Datei in ein anderes Programm zu `import` (z. B. wenn Sie sie als Teil einer Bibliothek schreiben). Sie können dann Ihre Datei `import` , und der `__main__` Trap stellt sicher, dass kein Code unerwartet ausgeführt wird:

```
# 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()
```

Redewendungen online lesen: https://riptutorial.com/de/python/topic/3070/redewendungen

Kapitel 147: Reduzieren

Syntax

- reduzieren (Funktion, iterable [, Initialisierung])

Parameter

Parameter	Einzelheiten
Funktion	Funktion, die zur Reduzierung des Iterierbaren verwendet wird (muss zwei Argumente annehmen) (nur *positionell*)
iterable	iterable das wird reduziert werden. (nur *positionell*)
Initialisierer	Startwert der Reduktion. (*optional* , nur *positionell*)

Bemerkungen

`reduce` möglicherweise nicht immer die effizienteste Funktion. Für einige Typen gibt es gleichwertige Funktionen oder Methoden:

- `sum()` für die Summe einer Sequenz, die *addierbare* Elemente (keine Strings) enthält:

```
sum([1,2,3])                                    # = 6
```

- `str.join` für die Verkettung von Strings:

```
''.join(['Hello', ',', ' World'])               # = 'Hello, World'
```

- `next` zusammen mit einem Generator könnte man eine Kurzschlussvariante im Vergleich dazu `reduce` :

```
# First falsy item:
next((i for i in [100, [], 20, 0] if not i)) # = []
```

Examples

Überblick

```
# 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` reduziert eine Iteration, indem eine Funktion wiederholt auf das nächste Element einer `iterable` und das kumulative Ergebnis `iterable` .

```
def add(s1, s2):
    return s1 + s2

asequence = [1, 2, 3]

reduce(add, asequence)  # equivalent to: add(add(1,2),3)
# Out: 6
```

In diesem Beispiel haben wir unsere eigene `add` Funktion definiert. Python verfügt jedoch über eine standardmäßige äquivalente Funktion im `operator` :

```
import operator
reduce(operator.add, asequence)
# Out: 6
```

`reduce` kann auch ein Startwert übergeben werden:

```
reduce(add, asequence, 10)
# Out: 16
```

Verwenden Sie reduzieren

```
def multiply(s1, s2):
    print('{arg1} * {arg2} = {res}'.format(arg1=s1,
                                           arg2=s2,
                                           res=s1*s2))
    return s1 * s2

asequence = [1, 2, 3]
```

Bei einem `initializer` die Funktion durch Anwenden auf den Initialisierer und das erste iterierbare Element gestartet:

```
cumprod = reduce(multiply, asequence, 5)
# Out: 5 * 1 = 5
#      5 * 2 = 10
#      10 * 3 = 30
print(cumprod)
# Out: 30
```

Ohne `initializer` beginnt die `reduce` mit der Anwendung der Funktion auf die ersten beiden Listenelemente:

```
cumprod = reduce(multiply, asequence)
# Out: 1 * 2 = 2
#      2 * 3 = 6
print(cumprod)
# Out: 6
```

Kumulatives Produkt

```
import operator
reduce(operator.mul, [10, 5, -3])
# Out: -150
```

Nicht-Kurzschlussvariante von any / all

`reduce` wird die Iteration nicht beenden , bevor die `iterable` completly iteriert wurde , so dass es einen nicht kurzschluß zu schaffen verwendet werden kann `any()` oder `all()` Funktion:

```
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
```

Erstes Wahrheits- / Falsches Element einer Sequenz (oder letztes Element, wenn es keine gibt)

```
# 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
```

Anstatt eine `lambda` Funktion zu erstellen, wird im Allgemeinen empfohlen, eine benannte Funktion zu erstellen:

```
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])                   # = []
```

Reduzieren online lesen: https://riptutorial.com/de/python/topic/328/reduzieren

Kapitel 148: Reguläre Ausdrücke (Regex)

Einführung

Python macht reguläre Ausdrücke über das `re` Modul verfügbar.

Reguläre Ausdrücke sind Kombinationen von Zeichen, die als Regeln für übereinstimmende Teilzeichenfolgen interpretiert werden. Zum Beispiel kann der Ausdruck `'amount\D+\d+'` werde jede Saite durch das Wort zusammengesetzt übereinstimmen `amount` zuzüglich einer ganzen Zahl, getrennt durch eine oder mehr nicht-Ziffern, wie beispielsweise: `amount=100` , `amount is 3` , `amount is equal to: 33` usw.

Syntax

- **Direkte reguläre Ausdrücke**
- re.match (Muster, Zeichenfolge, Flag = 0) # Out: Suchmuster am Anfang der Zeichenfolge oder Keine
- re.search (Muster, Zeichenfolge, Flag = 0) # Out: Suchmuster in Zeichenfolge oder Keine
- re.findall (Muster, Zeichenfolge, Flag = 0) # Out: Liste aller Übereinstimmungen von Muster in Zeichenfolge oder []
- re.finditer (pattern, string, flag = 0) # Out: wie re.findall, gibt jedoch das Iterator-Objekt zurück
- re.sub (Muster, Ersetzung, Zeichenfolge, Flag = 0) # Out: Zeichenfolge mit Ersetzung (Zeichenfolge oder Funktion) anstelle von Muster
- **Vorkompilierte reguläre Ausdrücke**
- precompiled_pattern = re.compile (Muster, Flag = 0)
- precompiled_pattern.match (string) # Out: Übereinstimmung am Anfang von String oder None
- precompiled_pattern.search (string) # Out: Entsprechung an beliebiger Stelle in String oder None
- precompiled_pattern.findall (string) # Out: Liste aller übereinstimmenden Teilstrings
- precompiled_pattern.sub (string / pattern / function, string) # Out: ersetzter String

Examples

Den Anfang einer Zeichenfolge abgleichen

Das erste Argument von `re.match()` ist der reguläre Ausdruck, das zweite ist der zu `re.match()` String:

```
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'
```

Möglicherweise stellen Sie fest, dass die Mustervariable eine Zeichenfolge mit dem Präfix `r` , was darauf hinweist, dass es sich bei der Zeichenfolge um ein *reines Zeichenkettenliteral handelt* .

Ein Roh-String-Literal hat eine etwas andere Syntax als ein String-Literal, nämlich ein Backslash `\` in einem Roh-String-Literal bedeutet "nur ein Backslash", und es ist nicht notwendig, Backlashes zu verdoppeln, um "Escape-Sequenzen" wie Zeilenumbrüche (`\n`) zu vermeiden. , Tabs (`\t`), Backspaces (`\`), Formular-Feeds (`\r`) und so weiter. Bei normalen String-Literalen muss jeder Backslash verdoppelt werden, um nicht als Beginn einer Escape-Sequenz zu gelten.

Daher ist `r"\n"` eine Zeichenfolge von 2 Zeichen: `\` und `n` . Regex-Muster verwenden auch umgekehrte Schrägstriche, z. B. bezieht sich `\d` auf ein beliebiges Zeichen. Wir können vermeiden, dass wir unsere Strings (`"\\d"`) mit rohen Strings (`r"\d"`) doppelt sichern müssen.

Zum Beispiel:

```
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'
```

Der Abgleich erfolgt nur am Anfang der Zeichenfolge. Wenn Sie irgendwo `re.search` möchten, verwenden `re.search` stattdessen `re.search` :

```
match = re.match(r"(123)", "a123zzb")

match is None
# Out: True

match = re.search(r"(123)", "a123zzb")

match.group()
```

```
# Out: '123'
```

Suchen

```
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.'
```

Die Suche wird im Gegensatz zu `re.match` beliebigen Stelle in der Zeichenfolge `re.match` . Sie können auch `re.findall` .

Sie können auch am Anfang der Zeichenfolge suchen (verwenden Sie `^`).

```
match = re.search(r"^123", "123zzb")
match.group(0)
# Out: '123'

match = re.search(r"^123", "a123zzb")
match is None
# Out: True
```

am Ende des Strings (verwende `$`),

```
match = re.search(r"123$", "zzb123")
match.group(0)
# Out: '123'

match = re.search(r"123$", "123zzb")
match is None
# Out: True
```

oder beides (verwende sowohl `^` als auch `$`):

```
match = re.search(r"^123$", "123")
match.group(0)
# Out: '123'
```

Gruppierung

Die Gruppierung erfolgt in Klammern. Beim Aufruf von `group()` eine Zeichenfolge zurückgegeben, die aus den übereinstimmenden, in Klammern stehenden Untergruppen besteht.

```
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'
```

Für `group()` können auch Argumente bereitgestellt werden, um eine bestimmte Untergruppe abzurufen.

Aus den Dokumenten :

> Wenn es ein einzelnes Argument gibt, ist das Ergebnis eine einzelne Zeichenfolge. Wenn es mehrere Argumente gibt, ist das Ergebnis ein Tupel mit einem Element pro Argument.

Wenn Sie `groups()` aufrufen, wird dagegen eine Liste von Tupeln zurückgegeben, die die Untergruppen enthalten.

```
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')
```

Benannte Gruppen

```
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'
```

Erstellt eine Capture-Gruppe, auf die sowohl nach Name als auch nach Index verwiesen werden kann.

Nicht einfangende Gruppen

Mit `(?:)` wird eine Gruppe erstellt, die Gruppe wird jedoch nicht erfasst. Das bedeutet, dass Sie es als Gruppe verwenden können, aber Ihren "Gruppenbereich" nicht verschmutzen.

```
re.match(r'(\d+)(\+(\d+))?', '11+22').groups()
# Out: ('11', '+22', '22')

re.match(r'(\d+)(?:\+(\d+))?', '11+22').groups()
# Out: ('11', '22')
```

Dieses Beispiel entspricht `11+22` oder `11` , jedoch nicht `11+` . Das ist , da das `+` Zeichen und der zweite Ausdruck gruppiert sind. Auf der anderen Seite, die `+` ist Zeichen nicht erfaßt.

Sonderzeichen entkommen

Sonderzeichen (wie die Zeichenklassen `[` und `]` unten) werden nicht wörtlich abgeglichen:

```
match = re.search(r'[b]', 'a[b]c')
match.group()
# Out: 'b'
```

Durch das Umgehen der Sonderzeichen können sie wörtlich abgeglichen werden:

```
match = re.search(r'\[b\]', 'a[b]c')
match.group()
# Out: '[b]'
```

Die Funktion `re.escape()` kann für Sie verwendet werden:

```
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'
```

Die Funktion `re.escape()` entgeht allen Sonderzeichen. Sie können also einen regulären Ausdruck basierend auf Benutzereingaben erstellen:

```
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.!']
```

Ersetzen

`re.sub` können Saiten mit `re.sub` .

Zeichenketten ersetzen

```
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'
```

Gruppenreferenzen verwenden

Ersetzungen mit einer kleinen Anzahl von Gruppen können wie folgt durchgeführt werden:

```
re.sub(r"t([0-9])([0-9])", r"t\2\1", "t13 t19 t81 t25")
# Out: 't31 t91 t18 t52'
```

Wenn Sie jedoch eine Gruppen-ID wie '10' erstellen, funktioniert dies nicht : `\10` wird als 'ID-Nummer 1 gefolgt von 0' gelesen. Sie müssen also genauer sein und die Schreibweise `\g<i>` verwenden:

```
re.sub(r"t([0-9])([0-9])", r"t\g<2>\g<1>", "t13 t19 t81 t25")
# Out: 't31 t91 t18 t52'
```

Ersatzfunktion verwenden

```
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'
```

Alle nicht überlappenden Übereinstimmungen suchen

```
re.findall(r"[0-9]{2,3}", "some 1 text 12 is 945 here 4445588899")
# Out: ['12', '945', '444', '558', '889']
```

Beachten Sie, dass das `r` vor `"[0-9]{2,3}"` Python anweist, den String so zu interpretieren, wie er ist; als "rohe" Zeichenfolge.

Sie können auch `re.finditer()` das auf dieselbe Weise wie `re.findall()` jedoch einen Iterator mit `SRE_Match` Objekten anstelle einer Liste von Strings `SRE_Match` :

```
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
'''
```

Vorkompilierte Muster

```
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
```

Durch das Kompilieren eines Musters kann es später in einem Programm wiederverwendet werden. Beachten Sie jedoch, dass Python kürzlich verwendete Ausdrücke (docs , SO answer) zwischenspeichert, so dass *"Programme, die jeweils nur wenige reguläre Ausdrücke verwenden, sich keine Sorgen um das Kompilieren regulärer Ausdrücke machen müssen"* .

```
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
```

Es kann mit re.match () verwendet werden.

Auf zulässige Zeichen prüfen

Wenn Sie überprüfen möchten, dass eine Zeichenfolge nur eine bestimmte Menge von Zeichen enthält, in diesem Fall az, AZ und 0-9, können Sie dies folgendermaßen tun:

```
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'
```

Sie können die Ausdruckszeile auch von `[^a-zA-Z0-9.]` `[^a-z0-9.]` , um beispielsweise Großbuchstaben zu verbieten.

Teilgutschrift: http://stackoverflow.com/a/1325265/2697955

Zeichenfolge mit regulären Ausdrücken aufteilen

Sie können auch reguläre Ausdrücke verwenden, um eine Zeichenfolge zu teilen. Zum Beispiel,

```
import re
data = re.split(r'\s+', 'James 94 Samantha 417 Scarlett 74')
print( data )
# Output: ['James', '94', 'Samantha', '417', 'Scarlett', '74']
```

Flaggen

In einigen speziellen Fällen müssen wir das Verhalten des regulären Ausdrucks ändern. Dies geschieht mithilfe von Flags. Flags können auf zwei Arten gesetzt werden, über das Schlüsselwort `flags` oder direkt im Ausdruck.

Flaggen-Schlüsselwort

Nachfolgend ein Beispiel für `re.search` aber es funktioniert für die meisten Funktionen im Modul `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'
```

Gemeinsame Flaggen

Flagge	kurze Beschreibung
`re.IGNORECASE` , `re.I`	Lässt das Muster den Fall ignorieren
`re.DOTALL` , `re.S`	Macht `.` alles zusammenbringen, einschließlich Newlines
`re.MULTILINE` , `re.M`	Lässt `^` den Anfang einer Zeile und `$` das Ende einer Zeile anpassen
`re.DEBUG`	Aktiviert Debug-Informationen

Die vollständige Liste aller verfügbaren Flags finden Sie in den Dokumenten

Inline-Flags

Aus den Dokumenten :

> `(?iLmsux)` (Ein oder mehrere Buchstaben aus der Gruppe 'i', 'L', 'm', 's', 'u', 'x'.)
>
> Die Gruppe stimmt mit der leeren Zeichenfolge überein. die Buchstaben setzen die entsprechenden Flags: re.I (Groß- / Kleinschreibung ignorieren), re.L (abhängig vom Gebietsschema), re.M (mehrzeilig), re.S (Punkt trifft auf alle zu), re.U (abhängig von Unicode) und re.X (verbose) für den gesamten regulären Ausdruck. Dies ist nützlich, wenn Sie die Flags als Teil des regulären Ausdrucks einschließen möchten, anstatt ein Flagargument an die Funktion re.compile () zu übergeben.
>
> Beachten Sie, dass das Flag (? X) ändert, wie der Ausdruck analysiert wird. Es sollte zuerst in der Ausdruckszeichenfolge oder nach einem oder mehreren Leerzeichen verwendet werden. Wenn sich vor dem Flag Zeichen ohne Leerzeichen befinden, sind die Ergebnisse undefiniert.

Iteration über Matches mit `re.finditer`

Sie können `re.finditer` , um alle Übereinstimmungen in einer Zeichenfolge zu `re.finditer` . Dies gibt Ihnen (im Vergleich zu `re.findall` zusätzliche Informationen, z. B. Informationen über den Übereinstimmungsort in der Zeichenfolge (Indizes):

```
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))
```

Ergebnis:

```
Match "an" found at: [5,7]
Match "an" found at: [20,22]
Match "ant" found at: [23,26]
```

Stimmen Sie einen Ausdruck nur an bestimmten Orten ab

Häufig möchten Sie einen Ausdruck nur an *bestimmten* Stellen zuordnen (dh, er bleibt an anderen Stellen unberührt). Betrachten Sie den folgenden Satz:

```
An apple a day keeps the doctor away (I eat an apple everyday).
```

Hier kommt der "Apfel" zweimal vor, was mit sogenannten *Backtracking-Kontrollverben* gelöst werden kann, die vom neueren `regex` Modul unterstützt werden. Die Idee ist:

```
forget_this | or this | and this as well | (but keep this)
```

Bei unserem Apfelbeispiel wäre dies:

```
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
```

Dies entspricht "Apfel" nur, wenn es außerhalb der Klammern gefunden werden kann.

So funktioniert das:

- Während von **links nach rechts** schauen, die Regex - Engine alles links verbraucht, `(*SKIP)` fungiert als „immer wahr Behauptung“. Danach fällt es bei `(*FAIL)` und Backtracks korrekt aus.
- Nun kommt es zum Punkt `(*SKIP)` **von rechts nach links** (aka beim Backtracking), wo es verboten ist, weiter nach links zu gehen. Stattdessen wird der Engine befohlen, alles nach links wegzuwerfen und zu dem Punkt zu springen, an dem `(*SKIP)` aufgerufen wurde.

Reguläre Ausdrücke (Regex) online lesen: https://riptutorial.com/de/python/topic/632/regulare-ausdrucke--regex-

Kapitel 149: Rekursion

Bemerkungen

Rekursion benötigt eine `stopCondition` um die Rekursion zu `stopCondition` .

Die ursprüngliche Variable muss an die rekursive Funktion übergeben werden, damit sie gespeichert wird.

Examples

Summe der Zahlen von 1 bis n

Wenn ich die Summe der Zahlen von `1` bis `n` herausfinden wollte, wobei `n` eine natürliche Zahl ist, kann ich `1 + 2 + 3 + 4 + ... + (several hours later) + n` . Alternativ könnte ich eine `for` Schleife schreiben:

```
n = 0
for i in range (1, n+1):
    n += i
```

Oder ich könnte eine als Rekursion bekannte Technik verwenden:

```
def recursion(n):
    if n == 1:
        return 1
    return n + recursion(n - 1)
```

Rekursion hat Vorteile gegenüber den beiden obigen Methoden. Rekursion benötigt weniger Zeit als das Herausschreiben von `1 + 2 + 3` für eine Summe von 1 bis 3. Bei `recursion(4)` kann Rekursion zum Rückwärtsarbeiten verwendet werden:

Funktionsaufrufe: (4 -> 4 + 3 -> 4 + 3 + 2 -> 4 + 3 + 2 + 1 -> 10)

Während die `for` Schleife streng vorwärts arbeitet: (1 -> 1 + 2 -> 1 + 2 + 3 -> 1 + 2 + 3 + 4 -> 10). Manchmal ist die rekursive Lösung einfacher als die iterative Lösung. Dies ist offensichtlich, wenn eine Stornierung einer verknüpften Liste implementiert wird.

Was, Wie und Wann der Rekursion

Rekursion tritt auf, wenn ein Funktionsaufruf bewirkt, dass dieselbe Funktion erneut aufgerufen wird, bevor der ursprüngliche Funktionsaufruf beendet wird. Betrachten Sie zum Beispiel den bekannten mathematischen Ausdruck `x!` (dh die faktorielle Operation). Die faktorielle Operation wird für alle nicht negativen Ganzzahlen wie folgt definiert:

- Wenn die Zahl 0 ist, lautet die Antwort 1.

- Andernfalls lautet die Antwort: Diese Zahl multipliziert mit der Fakultät einer Zahl, die kleiner als diese Zahl ist.

In Python kann eine naive Implementierung der faktoriellen Operation als Funktion wie folgt definiert werden:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

Rekursionsfunktionen sind manchmal schwer zu verstehen, gehen wir also Schritt für Schritt durch. Betrachten Sie die Ausdrucksfaktor `factorial(3)` . Dies und *alle* Funktionsaufrufe schaffen eine neue **Umgebung** . Eine Umgebung ist im Grunde nur eine Tabelle, die Bezeichner (z. B. `n` , `factorial` , `print` usw.) ihren entsprechenden Werten zuordnet. Sie können jederzeit mit `locals()` auf die aktuelle Umgebung zugreifen. Beim ersten Funktionsaufruf wird als einzige lokale Variable `n = 3` . Daher würde das Drucken von `locals() {'n': 3}` . Da `n == 3` , wird der Rückgabewert zu `n * factorial(n - 1)` .

In diesem nächsten Schritt können die Dinge etwas verwirrend werden. Wenn wir unseren neuen Ausdruck betrachten, wissen wir bereits, was `n` ist. Wir wissen jedoch noch nicht, was `factorial(n - 1)` ist. Zuerst wird `n - 1` zu `2` ausgewertet. Dann wird `2` als Wert für `n` an `factorial` . Da dies ein neuer Funktionsaufruf ist, wird eine zweite Umgebung zum Speichern dieses neuen `n` . Sei *A* die erste Umgebung und *B* die zweite Umgebung. *A* existiert immer noch und ist gleich `{'n': 3}` , jedoch ist *B* (was gleich `{'n': 2}`) die aktuelle Umgebung. Betrachtet man den Funktionsrumpf, so ist der Rückgabewert wiederum `n * factorial(n - 1)` . Ohne diesen Ausdruck auszuwerten, setzen wir ihn in den ursprünglichen Rückgabeausdruck ein. Auf diese Weise verwerfen wir *B* geistig. Erinnern Sie sich also daran, `n` entsprechend zu ersetzen (dh Verweise auf *B* 's `n` werden durch `n - 1` , das *A* ' s `n`). Nun wird der ursprüngliche Rückgabeausdruck zu `n * ((n - 1) * factorial((n - 1) - 1))` . Nehmen Sie sich eine Sekunde Zeit, um sich zu vergewissern, warum dies so ist.

Nun wollen wir den `factorial((n - 1) - 1))` bewerten. Da *A* `n == 3` , übergeben wir `1` in `factorial` . Daher erstellen wir eine neue Umgebung *C,* die gleich `{'n': 1}` . Der Rückgabewert ist wiederum `n * factorial(n - 1)` . Ersetzen Sie also die `factorial((n - 1) - 1))` des "ursprünglichen" Rückgabeausdrucks ähnlich wie zuvor den ursprünglichen Rückgabeausdruck angepasst. Der "ursprüngliche" Ausdruck ist jetzt `n * ((n - 1) * ((n - 2) * factorial((n - 2) - 1)))` .

Fast fertig. Nun müssen wir die `factorial((n - 2) - 1)` bewerten. Diesmal übergeben wir `0` . Daher wird dies mit `1` bewertet. Lassen Sie uns nun unsere letzte Ersetzung durchführen. Der "ursprüngliche" Rückkehrausdruck ist jetzt `n * ((n - 1) * ((n - 2) * 1))` . Unter Hinweis darauf, dass der ursprüngliche Rückgabeausdruck unter *A* ausgewertet wird, wird der Ausdruck zu `3 * ((3 - 1) * ((3 - 2) * 1))` . Dies wird natürlich mit 6 bewertet. Um zu bestätigen, dass dies die richtige Antwort ist, erinnern Sie sich an `3! == 3 * 2 * 1 == 6` . Vergewissern Sie sich vor dem Lesen, dass Sie das Konzept der Umgebungen und deren Anwendung für die Rekursion vollständig verstehen.

Die Anweisung `if n == 0: return 1` wird als Basisfall bezeichnet. Dies liegt daran, dass es keine Rekursion zeigt. Ein Basisfall ist unbedingt erforderlich. Ohne eins stoßen Sie auf unendliche Rekursion. Solange Sie mindestens einen Basisfall haben, können Sie so viele Fälle haben, wie Sie möchten. Zum Beispiel könnten wir eine gleichwertige `factorial` wie folgt schreiben:

```
def factorial(n):
    if n == 0:
        return 1
    elif n == 1:
        return 1
    else:
        return n * factorial(n - 1)
```

Sie können auch mehrere Rekursionsfälle haben, aber wir werden nicht darauf eingehen, da dies relativ ungewöhnlich ist und es oft schwierig ist, sie mental zu verarbeiten.

Sie können auch "parallele" rekursive Funktionsaufrufe durchführen. Betrachten Sie zum Beispiel die Fibonacci-Sequenz, die wie folgt definiert ist:

- Wenn die Zahl 0 ist, lautet die Antwort 0.
- Wenn die Nummer 1 ist, lautet die Antwort 1.
- Ansonsten ist die Antwort die Summe der vorherigen zwei Fibonacci-Zahlen.

Wir können dies wie folgt definieren:

```
def fib(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib(n - 2) + fib(n - 1)
```

Ich werde diese Funktion nicht so gründlich durchlaufen wie bei `factorial(3)` , aber der endgültige Rückgabewert von `fib(5)` entspricht dem folgenden (*syntaktisch* ungültigen) Ausdruck:

```
(
  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)
    )
  )
)
```

Dies wird zu `(1 + (0 + 1)) + ((0 + 1) + (1 + (0 + 1)))` was natürlich mit `5` bewertet wird.

Lassen Sie uns nun ein paar weitere Vokabelthemen behandeln:

- Ein **Abbruchaufruf** ist einfach ein rekursiver Funktionsaufruf, bei dem es sich um die zuletzt ausgeführte Operation handelt, bevor ein Wert zurückgegeben wird. Um klar zu sein, ist `return foo(n - 1)` ein Rückruf, aber `return foo(n - 1) + 1` nicht (da der Zusatz die letzte Operation ist).
- **Die Rückrufoptimierung** (TCO) ist eine Möglichkeit, Rekursion in rekursiven Funktionen automatisch zu reduzieren.
- **Tail Call Elimination** (TCE) ist die Reduzierung eines Tail Call auf einen Ausdruck, der ohne Rekursion ausgewertet werden kann. TCE ist eine Art TCO.

Die Optimierung von Rückrufen ist aus mehreren Gründen hilfreich:

- Der Interpreter kann den von Umgebungen belegten Speicherplatz minimieren. Da kein Computer über unbegrenzten Speicher verfügt, würden übermäßige rekursive Funktionsaufrufe zu einem **Stapelüberlauf führen** .
- Der Interpreter kann die Anzahl der **Stack-Frame-** Switches reduzieren.

Python hat aus verschiedenen Gründen keine TCO-Form implementiert. Daher sind andere Techniken erforderlich, um diese Einschränkung zu umgehen. Die Methode der Wahl hängt vom Anwendungsfall ab. Mit einiger Intuition können die Definitionen von `factorial` und `fib` relativ leicht in iterativen Code wie folgt konvertiert werden:

```
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
```

Dies ist in der Regel die effizienteste Möglichkeit, Rekursion manuell zu eliminieren, bei komplexeren Funktionen kann dies jedoch sehr schwierig werden.

Ein weiteres nützliches Werkzeug ist der lru_cache- Dekorator von Python, mit dem die Anzahl der redundanten Berechnungen reduziert werden kann.

Sie haben jetzt eine Idee, wie Sie Rekursion in Python vermeiden können, aber wann *sollten* Sie Rekursion verwenden? Die Antwort lautet "nicht oft". Alle rekursiven Funktionen können iterativ implementiert werden. Es geht einfach darum, herauszufinden, wie das geht. In seltenen Fällen ist Rekursion jedoch in Ordnung. Rekursion ist in Python üblich, wenn die erwarteten Eingaben keine signifikante Anzahl rekursiver Funktionsaufrufe verursachen würden.

Wenn Rekursion ein Thema ist, das Sie interessiert, bitte ich Sie, funktionale Sprachen wie Scheme oder Haskell zu lernen. In solchen Sprachen ist Rekursion viel nützlicher.

Bitte beachten Sie, dass das obige Beispiel für die Fibonacci-Sequenz zwar gut geeignet ist, um zu zeigen, wie die Definition in Python angewendet wird und später der Lru-Cache verwendet wird, eine ineffiziente Laufzeit aufweist, da für jeden Nicht-Basisfall zwei rekursive Aufrufe erfolgen. Die Anzahl der Aufrufe der Funktion steigt exponentiell auf `n` .
Nicht intuitiv würde eine effizientere Implementierung eine lineare Rekursion verwenden:

```
def fib(n):
    if n <= 1:
        return (n,0)
    else:
        (a, b) = fib(n - 1)
        return (a + b, a)
```

Aber dieser hat die Ausgabe eines *Zahlenpaares* . Dies unterstreicht, dass einige Funktionen von Rekursion wirklich nicht viel profitieren.

Baumerkundung mit Rekursion

Sagen wir, wir haben den folgenden Baum:

```
root
- A
  - AA
  - AB
- B
  - BA
  - BB
    - BBA
```

Wenn wir nun alle Namen der Elemente auflisten möchten, können Sie dies mit einer einfachen for-Schleife tun. Wir nehmen an, dass es eine Funktion `get_name()` , um eine Zeichenfolge des Namens eines Knotens zurückzugeben, eine Funktion `get_children()` , um eine Liste aller `get_children()` eines gegebenen Knotens in der Baumstruktur zurückzugeben, und eine Funktion `get_root()` an Holen Sie sich den Wurzelknoten.

```
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
```

Dies funktioniert gut und schnell, aber was ist, wenn die Unterknoten eigene Unterknoten bekommen? Und diese Unterknoten könnten weitere Unterknoten haben ... Was ist, wenn Sie nicht vorher wissen, wie viele es gibt? Eine Methode, um dieses Problem zu lösen, ist die Verwendung von Rekursion.

```
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
```

Vielleicht möchten Sie nicht drucken, sondern eine einfache Liste aller Knotennamen zurückgeben. Dies kann durch Übergeben einer Rolling List als Parameter erfolgen.

```
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']
```

Maximale Rekursionstiefe erhöhen

Die Tiefe der möglichen Rekursion ist begrenzt, abhängig von der Python-Implementierung. Wenn der Grenzwert erreicht ist, wird eine RuntimeError-Ausnahme ausgelöst:

```
RuntimeError: Maximum Recursion Depth Exceeded
```

Hier ist ein Beispiel eines Programms, das diesen Fehler verursachen würde:

```
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!
```

Sie können die Rekursionstiefenbegrenzung mit ändern

```
sys.setrecursionlimit(limit)
```

Sie können die aktuellen Parameter des Grenzwerts überprüfen, indem Sie Folgendes ausführen:

```
sys.getrecursionlimit()
```

Die gleiche Methode wie oben beschrieben mit unserem neuen Limit erhalten wir

```
sys.setrecursionlimit(2000)
cursing(0)
# Out: I recursed 1997 times!
```

Bei Python 3.5 ist die Ausnahme ein RecursionError, der von RuntimeError abgeleitet wird.

Schwanzrekursion - schlechte Praxis

Wenn das einzige, was von einer Funktion zurückgegeben wird, ein rekursiver Aufruf ist, wird dies als rekursive Rekursion bezeichnet.

Hier ein Beispiel für einen Countdown, der mit der Rekursion des Endes geschrieben wurde:

```
def countdown(n):
    if n == 0:
        print "Blastoff!"
    else:
        print n
        countdown(n-1)
```

Jede Berechnung, die unter Verwendung der Iteration durchgeführt werden kann, kann auch unter Verwendung der Rekursion durchgeführt werden. Hier ist eine Version von find_max, die mit Tail-Rekursion geschrieben wurde:

```
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)
```

Die Tail-Rekursion wird in Python als schlechte Praxis betrachtet, da der Python-Compiler die Optimierung für rekursive Tails-Aufrufe nicht behandelt. Die rekursive Lösung verwendet in solchen Fällen mehr Systemressourcen als die entsprechende iterative Lösung.

Rekursionsoptimierung durch Stack-Introspection

Standardmäßig darf der Rekursionsstapel von Python 1000 Frames nicht überschreiten. Sie können dies ändern, indem Sie `sys.setrecursionlimit(15000)` setzen. `sys.setrecursionlimit(15000)` ist jedoch schneller, da diese Methode mehr Speicher verbraucht. Stattdessen können wir das Problem der Tail-Rekursion auch durch Stack-Introspection lösen.

```
#!/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
```

Um die rekursiven Funktionen zu optimieren, können wir den `@tail_call_optimized` Dekorator verwenden, um unsere Funktion aufzurufen. Hier einige der häufigsten Rekursionsbeispiele, die den oben beschriebenen Dekorator verwenden:

Fakultätsbeispiel:

```
@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.
```

Fibonacci-Beispiel:

```
@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.
```

Rekursion online lesen: https://riptutorial.com/de/python/topic/1716/rekursion

Kapitel 150: Sammlungen-Modul

Einführung

Das integrierte `collections` Paket bietet verschiedene spezialisierte, flexible Collection-Typen, die sowohl eine hohe Leistung bieten als auch Alternativen zu den allgemeinen Collection-Typen `dict` , `list` , `tuple` und `set` bieten. Das Modul definiert auch abstrakte Basisklassen, die verschiedene Arten von Auflistungsfunktionen beschreiben (z. B. `MutableSet` und `ItemsView`).

Bemerkungen

Im **Kollektionsmodul** stehen drei weitere Typen zur Verfügung:

1. UserDict
2. Benutzerliste
3. UserString

Sie fungieren jeweils als Wrapper für das gebundene Objekt, z. B. *UserDict* als Wrapper für ein *Diktierobjekt* . In jedem Fall simuliert die Klasse ihren benannten Typ. Der Inhalt der Instanz wird in einem regulären Typobjekt gehalten, auf das über das Datenattribut der Wrapper-Instanz zugegriffen werden kann. In jedem dieser drei Fälle wurde die Notwendigkeit für diese Typen teilweise durch die Fähigkeit ersetzt, direkt vom Basistyp zu subklassieren. Es kann jedoch einfacher sein, mit der Wrapper-Klasse zu arbeiten, da der zugrunde liegende Typ als Attribut verfügbar ist.

Examples

Sammlungen. Zähler

Counter ist eine Diktierunterklasse, mit der Sie Objekte leicht zählen können. Es gibt nützliche Methoden zum Arbeiten mit den Häufigkeiten der Objekte, die Sie zählen.

```
import collections
counts = collections.Counter([1,2,3])
```

Der obige Code erstellt ein Objekt mit der Anzahl aller Elemente, die an den Konstruktor übergeben werden. Dieses Beispiel hat den Wert `Counter({1: 1, 2: 1, 3: 1})`

Konstruktionsbeispiele

Briefzähler

```
>>> 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})
```

Wortzähler

```
>>> 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})
```

Rezepte

```
>>> c = collections.Counter({'a': 4, 'b': 2, 'c': -2, 'd': 0})
```

Zählen Sie die einzelnen Elemente

```
>>> c['a']
4
```

Anzahl der einzelnen Elemente festlegen

```
>>> c['c'] = -3
>>> c
Counter({'a': 4, 'b': 2, 'd': 0, 'c': -3})
```

Gesamtzahl der Elemente in Zähler ermitteln (4 + 2 + 0 - 3)

```
>>> sum(c.itervalues())  # negative numbers are counted!
3
```

Elemente abrufen (nur diejenigen mit einem positiven Zähler werden beibehalten)

```
>>> list(c.elements())
['a', 'a', 'a', 'a', 'b', 'b']
```

Schlüssel mit 0 oder negativem Wert entfernen

```
>>> c - collections.Counter()
Counter({'a': 4, 'b': 2})
```

Alles entfernen

```
>>> c.clear()
>>> c
Counter()
```

Hinzufügen, einzelne Elemente entfernen

```
>>> 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})
```

collection.defaultdict

Collections.defaultdict (default_factory) gibt eine Unterklasse von `dict` , die über einen Standardwert für fehlende Schlüssel verfügt. Das Argument sollte eine Funktion sein, die den Standardwert zurückgibt, wenn sie ohne Argumente aufgerufen wird. Wenn nichts übergeben wird, wird standardmäßig " `None` .

```
>>> state_capitals = collections.defaultdict(str)
>>> state_capitals
defaultdict(<class 'str'>, {})
```

Gibt einen Verweis auf ein defaultdict zurück, das ein String-Objekt mit seiner default_factory-Methode erstellt.

In der `defaultdict` verwendet `defaultdict` einen der eingebauten Typen wie `str` , `int` , `list` oder `dict` als default_factory, da diese leere Typen zurückgeben, wenn sie ohne Argumente aufgerufen werden:

```
>>> str()
''
>>> int()
0
>>> list
[]
```

Das Aufrufen des defaultdict mit einer nicht vorhandenen Taste führt nicht zu einem Fehler wie in einem normalen Wörterbuch.

```
>>> state_capitals['Alaska']
''
>>> state_capitals
defaultdict(<class 'str'>, {'Alaska': ''})
```

Ein weiteres Beispiel mit `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})
```

Normale Wörterbuchmethoden arbeiten mit dem Standardwörterbuch

```
>>> state_capitals['Alabama'] = 'Montgomery'
>>> state_capitals
```

```
defaultdict(<class 'str'>, {'Alabama': 'Montgomery', 'Alaska': ''})
```

Wenn Sie `list` als default_factory verwenden, wird für jeden neuen Schlüssel eine Liste erstellt.

```
>>> 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

Die Reihenfolge der Schlüssel in Python-Wörterbüchern ist beliebig: Sie werden nicht von der Reihenfolge bestimmt, in der Sie sie hinzufügen.

Zum Beispiel:

```
>>> 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}
```

(Die oben angegebene willkürliche Reihenfolge bedeutet, dass Sie mit dem obigen Code andere Ergebnisse erzielen können als hier gezeigt.)

Die Reihenfolge, in der die Schlüssel erscheinen, ist die Reihenfolge, in der sie wiederholt werden würden, z. B. mithilfe einer `for` Schleife.

Die `collections.OrderedDict` Klasse stellt Wörterbuchobjekte bereit, die die Reihenfolge der Schlüssel beibehalten. `OrderedDict` s können wie unten gezeigt mit einer Reihe von bestellten Artikeln erstellt werden (hier eine Liste von Tupel-Schlüssel-Wert-Paaren):

```
>>> 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)])
```

Oder wir erstellen ein leeres `OrderedDict` und fügen dann Elemente hinzu:

```
>>> o = OrderedDict()
>>> o['key1'] = "value1"
>>> o['key2'] = "value2"
>>> print(o)
OrderedDict([('key1', 'value1'), ('key2', 'value2')])
```

Durch Iteration durch ein `OrderedDict` wird der Schlüsselzugriff in der Reihenfolge ermöglicht, in der sie hinzugefügt wurden.

Was passiert, wenn wir einem vorhandenen Schlüssel einen neuen Wert zuweisen?

```
>>> d['foo'] = 4
>>> print(d)
OrderedDict([('foo', 4), ('bar', 6), ('baz', 7), ('foobar', 8)])
```

Der Schlüssel behält seinen ursprünglichen Platz im `OrderedDict` .

collection.namedtuple

Definieren Sie einen neuen Typ `Person` mit `namedtuple` wie `namedtuple` :

```
Person = namedtuple('Person', ['age', 'height', 'name'])
```

Das zweite Argument ist die Liste der Attribute, die das Tupel haben wird. Sie können diese Attribute auch als durch Leerzeichen oder durch Kommas getrennte Zeichenfolge auflisten:

```
Person = namedtuple('Person', 'age, height, name')
```

oder

```
Person = namedtuple('Person', 'age height name')
```

Nach der Definition kann ein benanntes Tupel durch Aufrufen des Objekts mit den erforderlichen Parametern instanziiert werden, z.

```
dave = Person(30, 178, 'Dave')
```

Benannte Argumente können auch verwendet werden:

```
jack = Person(age=30, height=178, name='Jack S.')
```

Jetzt können Sie auf die Attribute des namedtuple zugreifen:

```
print(jack.age)   # 30
print(jack.name)  # 'Jack S.'
```

Das erste Argument für den namedTuple-Konstruktor (in unserem Beispiel `'Person'`) ist der `typename` . Es ist üblich, dasselbe Wort für den Konstruktor und den Typnamen zu verwenden, sie können jedoch unterschiedlich sein:

```
Human = namedtuple('Person',  'age, height, name')
dave = Human(30, 178, 'Dave')
print(dave)  # yields: Person(age=30, height=178, name='Dave')
```

Collections.deque

Gibt ein neues `deque` Objekt zurück, das von links nach rechts (mithilfe von append ()) mit den Daten von iterable initialisiert wurde. Wenn iterable nicht angegeben ist, ist der neue `deque` leer.

Deques sind eine Verallgemeinerung von Stapeln und Warteschlangen (der Name wird als "Deck" ausgesprochen und steht für "Double-Ended-Queue"). Deques unterstützen Thread-sichere, speichereffiziente Anhänge und Pop-ups von beiden Seiten der `deque` mit ungefähr derselben O (1) -Leistung in beiden Richtungen.

Obwohl Listenobjekte ähnliche Operationen unterstützen, sind sie für schnelle Operationen mit fester Länge optimiert und verursachen O (n) Speicherbewegungskosten für Pop (0) - und Insert-Operationen (0, v), die sowohl die Größe als auch die Position der zugrunde liegenden Datendarstellung ändern .

Neu in Version 2.4.

Wenn `maxlen` nicht angegeben ist oder `None` , können Deques beliebig lang werden. Ansonsten ist der `deque` an die angegebene maximale Länge gebunden. Wenn eine begrenzte `deque` voll ist und neue Elemente hinzugefügt werden, wird eine entsprechende Anzahl von Elementen vom gegenüberliegenden Ende entfernt. Begrenzte Längenverschiebungen bieten ähnliche Funktionen wie der Endfilter in Unix. Sie sind auch nützlich, um Transaktionen und andere Datenpools zu verfolgen, bei denen nur die letzte Aktivität von Interesse ist.

In Version 2.6 geändert: Maxlen-Parameter hinzugefügt.

```
>>> 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'])
```

Quelle: https://docs.python.org/2/library/collections.html

Collections.ChainMap

`ChainMap` ist neu in **Version 3.3**

Gibt ein neues `ChainMap` Objekt bei einer Anzahl von `maps` . Dieses Objekt fasst mehrere Diktate oder andere Zuordnungen zusammen, um eine einzelne, aktualisierbare Ansicht zu erstellen.

`ChainMap` sind nützlich, um verschachtelte Kontexte und Überlagerungen zu verwalten. Ein Beispiel in der Python-Welt ist die Implementierung der `Context` Klasse in Djangos Template-Engine. Es ist nützlich, um eine Reihe von Zuordnungen schnell zu verknüpfen, damit das Ergebnis als eine Einheit behandelt werden kann. Dies ist häufig viel schneller als das Erstellen eines neuen Wörterbuchs und das Ausführen mehrerer Aufrufe von `update()` .

Immer wenn einer eine Kette von Nachschlagewerten hat, kann es für `ChainMap` einen Fall `ChainMap` . Ein Beispiel enthält sowohl vom Benutzer angegebene Werte als auch ein Wörterbuch mit Standardwerten. Ein anderes Beispiel sind die `POST` und `GET` Parameterkarten, die bei der Verwendung im Web gefunden werden, z. B. Django oder Flask. Durch die Verwendung von `ChainMap` eine kombinierte Ansicht von zwei verschiedenen Wörterbüchern zurückgegeben.

Die `maps` Parameterliste wird von zuerst gesucht bis zuletzt durchsucht. Suchvorgänge

durchsuchen die zugrunde liegenden Zuordnungen nacheinander, bis ein Schlüssel gefunden wird. Im Gegensatz dazu werden Schreibvorgänge, Aktualisierungen und Löschvorgänge nur für das erste Mapping ausgeführt.

```
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)
```

Beachten Sie die Auswirkungen der Reihenfolge, auf die der Wert bei der nachfolgenden Suche zuerst gefunden wird

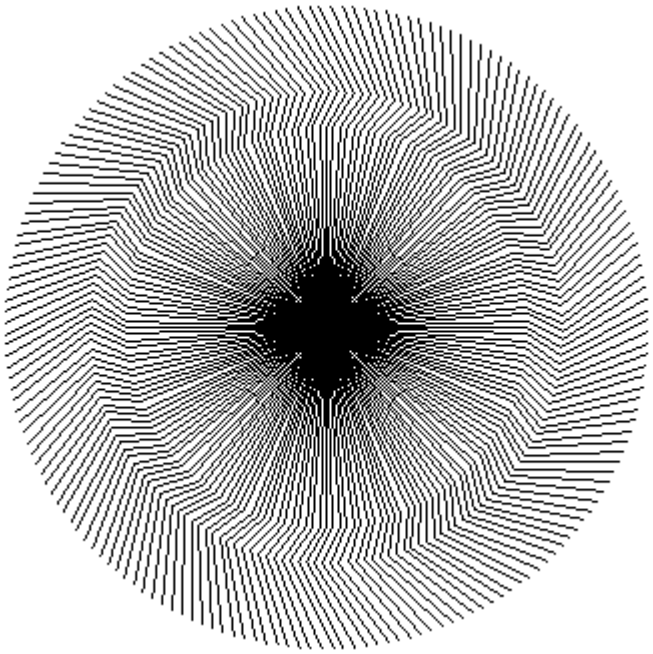
```
for k, v in combined_dict.items():
    print(k, v)

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
```

Sammlungen-Modul online lesen: https://riptutorial.com/de/python/topic/498/sammlungen-modul

Kapitel 151: Schildkröte-Grafiken

Examples

Ninja Twist (Schildkrötengrafik)



Hier ein Ninja Twist von Turtle Graphics:

```
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()
```

Schildkröte-Grafiken online lesen: https://riptutorial.com/de/python/topic/7915/schildkrote-grafiken

Kapitel 152: Schleifen

Einführung

Als eine der grundlegendsten Funktionen beim Programmieren sind Schleifen in fast jeder Programmiersprache ein wichtiges Element. Mithilfe von Schleifen können Entwickler bestimmte Teile ihres Codes so einstellen, dass sie durch eine Reihe von Schleifen wiederholt werden, die als Iterationen bezeichnet werden. In diesem Thema werden verschiedene Arten von Schleifen und Anwendungen von Schleifen in Python behandelt.

Syntax

- Während <boolescher Ausdruck>:
- für <variable> in <iterable>:
- für <Variable> im Bereich (<Nummer>):
- für <variable> im Bereich (<start_number>, <end_number>):
- für <Variable> im Bereich (<Startnummer>, <Endnummer>, <Schrittgröße>):
- für i, <variable> in Aufzählung (<iterable>): # mit Index i
- für <variable1>, <variable2> in zip (<iterable1>, <iterable2>):

Parameter

Parameter	Einzelheiten
Boolescher Ausdruck	Ausdruck, der in einem booleschen Kontext ausgewertet werden kann, z. B. `x < 10`
Variable	Variablenname für das aktuelle Element aus dem `iterable` Element
iterable	alles, was Iterationen implementiert

Examples

Iterieren über Listen

Iterieren über eine Liste , die Sie verwenden können , `for` :

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

Dadurch werden die Elemente der Liste ausgedruckt:

```
one
two
```

```
three
four
```

Die `range` Funktion generiert Zahlen, die auch häufig in einer for-Schleife verwendet werden.

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

Das Ergebnis ist ein spezieller Bereichssequenztyp in Python> = 3 und eine Liste in Python <= 2. Beide können mit der for-Schleife durchgeschleift werden.

```
1
2
3
4
5
```

Wenn Sie beide Elemente einer Liste durchlaufen *und* auch einen Index für die Elemente haben möchten, können Sie die `enumerate` Python verwenden:

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

`enumerate` generiert Tupel, die in `index` (eine ganze Zahl) und `item` (den tatsächlichen Wert aus der Liste) entpackt werden. Die obige Schleife wird gedruckt

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

Durchlaufen Sie eine Liste mit Wertmanipulation mit `map` und `lambda` , dh wenden Sie für jedes Element in der Liste die Lambda-Funktion an:

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

Ausgabe:

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

Hinweis: In Python 3.x gibt `map` einen Iterator anstelle einer Liste zurück. Wenn Sie also eine Liste benötigen, müssen Sie das Ergebnis `print(list(x))` (siehe http://www.riptutorial.com/python/Beispiel / 8186 / map-- in http://www.riptutorial.com/python/topic/809/incompatibilities-moving-from-python-2-to-python-3).

Für Schleifen

`for` Schleifen durchlaufen eine Auflistung von Elementen, z. B. `list` oder `dict` , und führen einen

Codeblock mit jedem Element aus der Auflistung aus.

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

Die obige `for` Schleife durchläuft eine Liste mit Zahlen.

Bei jeder Iteration wird der Wert von `i` auf das nächste Element der Liste gesetzt. Also zuerst `0` , dann `1` , dann `2` usw. Die Ausgabe lautet wie folgt:

```
0
1
2
3
4
```

`range` ist eine Funktion, die eine Reihe von Zahlen in einer iterierbaren Form zurückgibt. Daher kann sie `for` Schleifen verwendet werden:

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

gibt das gleiche Ergebnis wie die erste `for` Schleife. Beachten Sie, dass `5` nicht gedruckt wird, da der Bereich hier die ersten fünf Zahlen sind, die von `0` .

Iterierbare Objekte und Iteratoren

`for` Schleife kann ein beliebiges iterierbares Objekt `__getitem__` dem es sich um ein Objekt handelt, das eine `__getitem__` oder eine `__iter__` Funktion definiert. Die Funktion `__iter__` gibt einen Iterator zurück. `__iter__` handelt es sich um ein Objekt mit einer `next` Funktion, mit der auf das nächste Element des iterierbaren Elements zugegriffen wird.

Brechen Sie in Loops und fahren Sie fort

Anweisung `break`

Wenn eine `break` Anweisung innerhalb einer Schleife ausgeführt wird, "bricht" der Steuerungsfluss sofort aus der Schleife heraus:

```
i = 0
while i < 7:
    print(i)
    if i == 4:
        print("Breaking from loop")
        break
    i += 1
```

Die Schleifenbedingung wird nach der Ausführung der `break` Anweisung nicht ausgewertet. Beachten Sie, dass `break` Anweisungen nur *innerhalb von Schleifen* syntaktisch zulässig *sind* . Eine `break` Anweisung innerhalb einer Funktion kann nicht zum Beenden von Schleifen verwendet werden, die diese Funktion aufgerufen haben.

Beim Ausführen des folgenden Befehls wird jede Ziffer bis Nummer `4` gedruckt, wenn die `break` Anweisung erfüllt ist und die Schleife stoppt:

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

`break` Anweisungen können auch innerhalb `for` Schleifen verwendet werden, dem anderen von Python bereitgestellten Schleifenkonstrukt:

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

Das Ausführen dieser Schleife wird jetzt gedruckt:

```
0
1
2
```

Beachten Sie, dass 3 und 4 nicht gedruckt werden, da die Schleife beendet ist.

Wenn eine Schleife eine `else` Klausel hat , wird sie nicht ausgeführt, wenn die Schleife durch eine `break` Anweisung beendet wird.

`continue` Aussage

Eine `continue` Anweisung springt zur nächsten Iteration der Schleife, wobei der Rest des aktuellen Blocks umgangen wird, die Schleife jedoch fortgesetzt wird. Wie bei `break` kann `continue` nur innerhalb von Schleifen erscheinen:

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

0
1
3
5
```

Beachten Sie, dass `2` und `4` nicht gedruckt werden. Dies liegt daran, dass `continue` zur nächsten Iteration geht, anstatt mit `print(i) continue` , wenn `i == 2` oder `i == 4` .

Verschachtelte Loops

`break` und `continue` nur auf einer Ebene der Schleife arbeiten. Das folgende Beispiel bricht nur aus der inneren `for` Schleife aus, nicht aus der äußeren `while` Schleife:

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

Python ist nicht in der Lage, mehrere Loop-Ebenen gleichzeitig zu durchbrechen. Wenn dieses Verhalten erwünscht ist, kann das Umgestalten einer oder mehrerer Loops in eine Funktion und das Ersetzen von `break` durch `return` möglicherweise der Weg sein.

Verwenden Sie `return` aus einer Funktion als `break`

Die `return` Anweisung verlässt eine Funktion, ohne den Code auszuführen, der danach kommt.

Wenn Sie eine Schleife innerhalb einer Funktion verwenden, ist die Verwendung von `return` von innerhalb dieser Schleife gleichbedeutend mit einer `break` da der Rest des Codes der Schleife nicht ausgeführt wird (*beachten Sie, dass nach der Schleife auch kein Code ausgeführt wird*):

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

Wenn Sie geschachtelte Schleifen haben, unterbricht die `return` Anweisung alle Schleifen:

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

wird ausgeben:

```
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
```

Loops mit einer "else" -Klausel

Die Anweisungen `for` und `while` zusammengesetzte Anweisungen (Schleifen) können optional eine `else` Klausel enthalten (in der Praxis ist diese Verwendung eher selten).

Die `else` Klausel wird nur ausgeführt, nachdem eine `for` Schleife durch Iteration bis zum Abschluss beendet wurde, oder nach einer `while` Schleife, wenn der bedingte Ausdruck falsch wird.

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

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

Ausgabe:

```
 0
 1
 2
 done
```

Die `else` Klausel wird *nicht* ausgeführt, wenn die Schleife auf andere Weise beendet wird (durch eine `break` Anweisung oder durch Auslösen einer Ausnahme):

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

Ausgabe:

```
0
1
```

Die meisten anderen Programmiersprachen verfügen nicht über diese optionale `else` Klausel von Schleifen. Die Verwendung des Schlüsselworts `else` insbesondere wird oft verwirrend betrachtet.

Das ursprüngliche Konzept für eine solche Klausel geht auf Donald Knuth zurück und die Bedeutung des Schlüsselworts `else` wird deutlich, wenn wir eine Schleife in Form von `if` Anweisungen und `goto` Anweisungen aus früheren Tagen vor der strukturierten Programmierung oder aus einer Assemblersprache einer niedrigeren Ebene schreiben.

Zum Beispiel:

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

ist äquivalent zu:

```
# pseudocode

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

<<end>>:
```

Diese bleiben gleichwertig, wenn wir jedem eine `else` Klausel hinzufügen.

Zum Beispiel:

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

ist äquivalent zu:

```
# pseudocode

<<start>>:
if loop_condition():
    ...
    if break_condition():
        goto <<end>>
    ...
    goto <<start>>
else:
    print('done')

<<end>>:
```

Eine `for` Schleife mit einer `else` Klausel kann auf dieselbe Weise verstanden werden. Konzeptionell gibt es eine Schleifenbedingung, die wahr bleibt, solange das iterierbare Objekt oder die Sequenz noch einige Elemente enthält.

Warum sollte man dieses seltsame Konstrukt verwenden?

Der Hauptanwendungsfall für das Konstrukt `for...else` ist eine präzise Implementierung der Suche, wie zum Beispiel:

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

Um den `else` in diesem Konstrukt weniger verwirrend zu machen, kann man es sich als " *wenn nicht brechen* " oder " *wenn nicht gefunden* " *vorstellen* .

Einige Diskussionen dazu finden Sie in [Python-ideas] Summary von for ... else-Threads . Warum verwendet Python 'else' nach und while-Schleifen? und Else-Klauseln zu Schleifenanweisungen

Wörterbücher iterieren

Betrachten Sie das folgende Wörterbuch:

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

Zum Durchlaufen der Schlüssel können Sie Folgendes verwenden:

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

Ausgabe:

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

Das ist äquivalent zu:

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

oder in Python 2:

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

Um durch seine Werte zu iterieren, verwenden Sie:

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

Ausgabe:

```
1
2
3
```

Um durch seine Schlüssel und Werte zu iterieren, verwenden Sie:

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

Ausgabe:

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

Man beachte , dass in 2 Python, `.keys()` , `.values()` und `.items()` eine Rückkehr `list` Objekt. Wenn Sie lediglich das Ergebnis `.iterkeys()` , können Sie die entsprechenden `.iterkeys()` , `.itervalues()` und `.iteritems()` .

Der Unterschied zwischen `.keys()` und `.iterkeys()` , `.values()` und `.itervalues()` , `.items()` und `.iteritems()` besteht darin, dass die `iter*` -Methoden Generatoren sind. Somit werden die Elemente innerhalb des Wörterbuchs bei der Auswertung nacheinander ausgegeben. Wenn eine `list` Objekt zurückgegeben wird, werden alle Elemente in eine Liste gepackt und kehrte dann zur weiteren Auswertung.

> Beachten Sie auch, dass in Python 3 die Reihenfolge der auf diese Weise gedruckten Elemente keiner Reihenfolge entspricht.

While-Schleife

Eine `while` Schleife bewirkt, dass die Schleifenanweisungen ausgeführt werden, bis die Schleifenbedingung falsch ist . Der folgende Code führt die Schleifenanweisungen insgesamt viermal aus.

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

Die obige Schleife kann leicht in eine elegantere `for` Schleife übersetzt werden, `while` Schleifen nützlich sind, um zu überprüfen, ob eine Bedingung erfüllt ist. Die folgende Schleife wird solange ausgeführt, bis `myObject` fertig ist.

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

`while` Schleifen können auch ohne Bedingung ausgeführt werden, indem Zahlen (komplex oder reell) oder `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
```

Wenn die Bedingung immer wahr ist, wird die while-Schleife für immer ausgeführt (Endlosschleife), wenn sie nicht durch eine break- oder return-Anweisung oder eine Ausnahme beendet wird.

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

Die Pass-Erklärung

`pass` ist eine Nullanweisung, wenn eine Python-Syntax eine Anweisung erfordert (z. B. im Rumpf einer `for` oder `while` Schleife), jedoch keine Aktion vom Programmierer erforderlich oder gewünscht ist. Dies kann als Platzhalter für Code dienen, der noch geschrieben werden muss.

```
for x in range(10):
    pass #we don't want to do anything, or are not ready to do anything here, so we'll pass
```

In diesem Beispiel wird nichts passieren. Die `for` Schleife wird ohne Fehler abgeschlossen, es werden jedoch keine Befehle oder Codes ausgeführt. `pass` können wir unseren Code erfolgreich ausführen, ohne dass alle Befehle und Aktionen vollständig implementiert sind.

In ähnlicher Weise kann `pass` in `while` Schleifen sowie in Auswahl- und Funktionsdefinitionen usw. verwendet werden.

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

Verschiedene Teile einer Liste mit unterschiedlicher Schrittweite iterieren

Angenommen, Sie haben eine lange Liste von Elementen und interessieren sich nur für jedes andere Element der Liste. Vielleicht möchten Sie nur das erste oder letzte Element oder einen bestimmten Bereich von Einträgen in Ihrer Liste untersuchen. Python verfügt über integrierte Indexierungsfunktionen. Hier einige Beispiele, wie Sie diese Szenarien erreichen können.

Hier ist eine einfache Liste, die in den Beispielen verwendet wird:

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

Iteration über die ganze Liste

Um jedes Element in der Liste zu durchlaufen, kann eine `for` Schleife wie folgt verwendet werden:

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

Die `for` Schleife weist jedem Element von `lst` s zu. Dies wird drucken:

```
a
b
c
d
e
```

Häufig benötigen Sie sowohl das Element als auch den Index dieses Elements. Das `enumerate` Schlüsselwort führt diese Aufgabe aus.

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

Der Index `idx` beginnt mit Null und erhöht sich für jede Iteration, während die `s` das zu bearbeitende Element enthalten. Der vorherige Ausschnitt wird ausgegeben:

```
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
```

Iteriere über Unterliste

Wenn Sie über einen Bereich iterieren möchten (wenn Sie sich daran erinnern, dass Python die nullbasierte Indizierung verwendet), verwenden Sie das Schlüsselwort `range` .

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

Dies würde ausgeben:

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

Die Liste kann auch in Scheiben geschnitten werden. Die folgende Slice-Notation geht vom

Element an Index 1 bis zum Ende mit einem Schritt von 2. Die beiden `for` Schleifen führen zum gleichen Ergebnis.

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

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

Die obigen Snippet-Ausgaben:

```
bravo
delta
```

Das Indizieren und Schneiden ist ein eigenes Thema.

Die "halbe Schleife" machen

Im Gegensatz zu anderen Sprachen verfügt Python nicht über ein do-until oder ein do-while-Konstrukt (dadurch kann der Code einmal ausgeführt werden, bevor die Bedingung getestet wird). Sie können jedoch eine `while True` mit einer `break` kombinieren, um denselben Zweck zu erreichen.

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

Dies wird drucken:

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

Schleifen und Auspacken

Wenn Sie beispielsweise eine Liste von Tupeln durchlaufen möchten:

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

anstatt etwas zu tun:

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

```
    # logic
```

oder sowas:

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

Sie können dies einfach tun:

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

Dies funktioniert auch für die *meisten* Arten von Iterables, nicht nur für Tupel.

Schleifen online lesen: https://riptutorial.com/de/python/topic/237/schleifen

Kapitel 153: setup.py

Parameter

Parameter	Verwendungszweck
`name`	Name Ihrer Distribution
`version`	Versionszeichenfolge Ihrer Distribution.
`packages`	Liste der Python-Pakete (dh Verzeichnisse mit Modulen), die eingeschlossen werden sollen. Dies kann manuell angegeben werden. `setuptools.find_packages()` wird stattdessen ein Aufruf von `setuptools.find_packages()` verwendet.
`py_modules`	Liste der Python-Module der obersten Ebene (`.py` einzelne `.py` Dateien), die eingeschlossen werden sollen.

Bemerkungen

Weitere Informationen zur Python-Verpackung finden Sie unter:

Einführung

Für das Schreiben offizieller Pakete gibt es eine Gebrauchsanweisung für Verpackungen .

Examples

Zweck von setup.py

Das Setup-Skript ist das Zentrum aller Aktivitäten beim Erstellen, Verteilen und Installieren von Modulen mithilfe der Distutils. Zweck ist die korrekte Installation der Software.

Wenn Sie nur ein Modul namens foo verteilen möchten, das in einer Datei foo.py enthalten ist, kann Ihr Setup-Skript so einfach sein:

```
from distutils.core import setup

setup(name='foo',
      version='1.0',
      py_modules=['foo'],
      )
```

Um eine Quelldistribution für dieses Modul zu erstellen, erstellen Sie ein Setupskript, setup.py, das den obigen Code enthält, und führen diesen Befehl von einem Terminal aus aus:

```
python setup.py sdist
```

sdist erstellt eine Archivdatei (z. B. Tarball unter Unix, ZIP-Datei unter Windows) mit Ihrem Setup-Skript setup.py und Ihrem Modul foo.py. Die Archivdatei heißt foo-1.0.tar.gz (oder .zip) und wird in ein Verzeichnis foo-1.0 entpackt.

Wenn ein Endbenutzer Ihr foo-Modul installieren möchte, muss er lediglich foo-1.0.tar.gz (oder .zip) herunterladen, entpacken und aus dem foo-1.0-Verzeichnis ausführen

```
python setup.py install
```

Hinzufügen von Befehlszeilenskripten zu Ihrem Python-Paket

Befehlszeilenskripts innerhalb von Python-Paketen sind üblich. Sie können Ihr Paket so organisieren, dass das Skript, wenn ein Benutzer das Paket installiert, unter seinem Pfad verfügbar ist.

Wenn Sie das `greetings` , das das Befehlszeilenskript `hello_world.py` .

```
greetings/
   greetings/
      __init__.py
      hello_world.py
```

Sie können dieses Skript ausführen, indem Sie Folgendes ausführen:

```
python greetings/greetings/hello_world.py
```

Wenn Sie es jedoch gerne so ausführen möchten:

```
hello_world.py
```

Sie können dies erreichen , indem das Hinzufügen `scripts` zu Ihrem `setup()` in `setup.py` wie folgt aus :

```
from setuptools import setup
setup(
  name='greetings',
  scripts=['hello_world.py']
)
```

Wenn Sie jetzt das Begrüßungspaket installieren, wird `hello_world.py` zu Ihrem Pfad hinzugefügt.

Eine andere Möglichkeit wäre, einen Einstiegspunkt hinzuzufügen:

```
entry_points={'console_scripts': ['greetings=greetings.hello_world:main']}
```

Auf diese Weise müssen Sie es nur so ausführen:

```
greetings
```

Verwenden der Quellcodeverwaltungsmetadaten in setup.py

`setuptools_scm` ist ein offiziell gesegnetes Paket, das Git oder Mercurial-Metadaten verwenden kann, um die Versionsnummer Ihres Pakets zu ermitteln und Python-Pakete und -Daten zu finden, die darin enthalten sind.

```
from setuptools import setup, find_packages

setup(
    setup_requires=['setuptools_scm'],
    use_scm_version=True,
    packages=find_packages(),
    include_package_data=True,
)
```

In diesem Beispiel werden beide Funktionen verwendet. Um nur SCM-Metadaten für die Version zu verwenden, ersetzen Sie den Aufruf von `find_packages()` durch Ihre manuelle Paketliste, oder entfernen Sie `use_scm_version=True` , um nur den `use_scm_version=True` .

Installationsoptionen hinzufügen

Wie in den vorherigen Beispielen gezeigt, ist die grundlegende Verwendung dieses Skripts:

```
python setup.py install
```

Es gibt jedoch noch mehr Optionen, beispielsweise die Installation des Pakets und die Möglichkeit, den Code zu ändern und zu testen, ohne ihn erneut installieren zu müssen. Dies geschieht mit:

```
python setup.py develop
```

Wenn Sie bestimmte Aktionen ausführen *möchten* , beispielsweise das Erstellen einer *Sphinx*-Dokumentation oder das *Erstellen von Fortran*- Code, können Sie eine eigene Option wie *folgt* erstellen:

```
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` und `finalize_options` werden vor und nach der `run` Funktion ausgeführt, wie ihre Namen es vermuten lassen.

Danach können Sie Ihre Option aufrufen:

```
python setup.py build_sphinx
```

setup.py online lesen: https://riptutorial.com/de/python/topic/1444/setup-py

Kapitel 154: Sichere Shell-Verbindung in Python

Parameter

Parameter	Verwendungszweck
Hostname	Dieser Parameter teilt dem Host mit, zu dem die Verbindung hergestellt werden muss
Nutzername	Benutzername erforderlich, um auf den Host zugreifen zu können
Hafen	Host-Port
Passwort	Passwort für das Konto

Examples

SSH-Verbindung

```
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
```

Sichere Shell-Verbindung in Python online lesen:
https://riptutorial.com/de/python/topic/5709/sichere-shell-verbindung-in-python

Kapitel 155: Sicherheit und Kryptographie

Einführung

Python ist eine der beliebtesten Sprachen in der Computer- und Netzwerksicherheit und hat ein großes Potenzial in Sicherheit und Kryptographie. Dieses Thema behandelt die kryptografischen Funktionen und Implementierungen in Python von der Verwendung in der Computer- und Netzwerksicherheit bis hin zu Hash- und Verschlüsselungs- / Entschlüsselungsalgorithmen.

Syntax

- hashlib.new (name)
- hashlib.pbkdf2_hmac (Name, Passwort, Salt, Runden, Dklen = Keine)

Bemerkungen

Bei vielen der Methoden in `hashlib` müssen Sie Werte übergeben, die als Puffer von Bytes und nicht als Zeichenfolgen interpretierbar sind. Dies ist der Fall für `hashlib.new().update()` sowie `hashlib.pbkdf2_hmac` . Wenn Sie eine Zeichenfolge haben, können Sie sie in einen Bytepuffer konvertieren, indem Sie das Zeichen `b` vor den Anfang der Zeichenfolge setzen:

```
  "This is a string"
 b"This is a buffer of bytes"
```

Examples

Berechnen eines Message-Digest

Das `hashlib` Modul ermöglicht das Erstellen von Message Digest-Generatoren mit der `new` Methode. Diese Generatoren verwandeln einen beliebigen String in einen Digest mit fester Länge:

```
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-
='
```

Beachten Sie, dass Sie eine beliebige Anzahl von `update` aufrufen können, bevor Sie `digest` aufrufen. `digest` ist nützlich, wenn Sie einen großen Datei-Chunk von Chunk durchgehen möchten. Sie können den Digest auch im Hexadezimalformat `hexdigest` indem Sie `hexdigest` :

```
h.hexdigest()
```

```
# ==> '2edfdada56525b1290ff16fb1744cfb482dd2914ffbcb649790c0e589e462d3d'
```

Verfügbare Hash-Algorithmen

`hashlib.new` erfordert den Namen eines Algorithmus, wenn Sie ihn zur Erzeugung eines Generators aufrufen. Um herauszufinden, welche Algorithmen im aktuellen Python-Interpreter verfügbar sind, verwenden Sie `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'}
```

Die zurückgegebene Liste variiert je nach Plattform und Interpret. Stellen Sie sicher, dass Ihr Algorithmus verfügbar ist.

Es gibt auch einige Algorithmen , die auf allen Plattformen und Dolmetscher zur Verfügung stehen sind *garantiert,* die unter Verwendung verfügbar sind `hashlib.algorithms_guaranteed` :

```
hashlib.algorithms_guaranteed
# ==> {'sha256', 'sha384', 'sha1', 'sha224', 'md5', 'sha512'}
```

Sicheres Passwort-Hashing

Der vom `hashlib` Modul `hashlib` PBKDF2-Algorithmus kann verwendet werden, um sicheres Passwort-Hashing durchzuführen. Dieser Algorithmus kann zwar keine Brute-Force-Angriffe verhindern, um das ursprüngliche Kennwort aus dem gespeicherten Hash wiederherzustellen, macht jedoch solche Angriffe sehr teuer.

```
import hashlib
import os

salt = os.urandom(16)
hash = hashlib.pbkdf2_hmac('sha256', b'password', salt, 100000)
```

PBKDF2 kann mit jedem Digest-Algorithmus arbeiten. Das obige Beispiel verwendet SHA256, was normalerweise empfohlen wird. Das zufällige Salt sollte zusammen mit dem Hash-Passwort gespeichert werden. Sie benötigen es erneut, um ein eingegebenes Passwort mit dem gespeicherten Hash zu vergleichen. Es ist wichtig, dass jedes Passwort mit einem anderen Salt gehashed wird. Es wird empfohlen, die Anzahl der Runden für Ihre Anwendung so hoch wie möglich einzustellen.

Wenn das Ergebnis hexadezimal sein soll, können Sie das `binascii` Modul verwenden:

```
import binascii
hexhash = binascii.hexlify(hash)
```

Hinweis : Während PBKDF2 nicht schlecht ist, gelten bcrypt und insbesondere scrypt als stärker

gegen Brute-Force-Angriffe. Zur Zeit ist kein Teil der Python-Standardbibliothek.

Hashing von Dateien

Ein Hash ist eine Funktion, die eine Folge von Bytes mit variabler Länge in eine Folge mit fester Länge konvertiert. Das Hashing von Dateien kann aus vielen Gründen vorteilhaft sein. Mithilfe von Hashes können Sie überprüfen, ob zwei Dateien identisch sind, oder überprüfen, ob der Inhalt einer Datei nicht beschädigt oder geändert wurde.

Sie können `hashlib` , um einen Hash für eine Datei zu generieren:

```
import hashlib

hasher = hashlib.new('sha256')
with open('myfile', 'r') as f:
    contents = f.read()
    hasher.update(contents)

print hasher.hexdigest()
```

Bei größeren Dateien kann ein Puffer mit fester Länge verwendet werden:

```
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())
```

Symmetrische Verschlüsselung mit Pycrypto

Die in Python integrierte Kryptofunktionalität ist derzeit auf das Hashing beschränkt. Für die Verschlüsselung ist ein Drittanbieter-Modul wie pycrypto erforderlich . Beispielsweise bietet es den AES-Algorithmus, der als Stand der Technik für die symmetrische Verschlüsselung gilt. Der folgende Code verschlüsselt eine bestimmte Nachricht mit einer Passphrase:

```
import hashlib
import math
import os

from Crypto.Cipher import AES

IV_SIZE = 16    # 128 bit, fixed for the AES algorithm
KEY_SIZE = 32   # 256 bit meaning AES-256, can also be 128 or 192 bits
SALT_SIZE = 16  # This size is arbitrary

cleartext = b'Lorem ipsum'
password = b'highly secure encryption password'
salt = os.urandom(SALT_SIZE)
derived = hashlib.pbkdf2_hmac('sha256', password, salt, 100000,
```

```
                              dklen=IV_SIZE + KEY_SIZE)
iv = derived[0:IV_SIZE]
key = derived[IV_SIZE:]

encrypted = salt + AES.new(key, AES.MODE_CFB, iv).encrypt(cleartext)
```

Der AES-Algorithmus benötigt drei Parameter: Verschlüsselungsschlüssel, Initialisierungsvektor (IV) und die eigentliche zu verschlüsselnde Nachricht. Wenn Sie einen zufällig generierten AES-Schlüssel haben, können Sie diesen direkt verwenden und lediglich einen zufälligen Initialisierungsvektor generieren. Eine Passphrase hat jedoch nicht die richtige Größe, und es ist auch nicht empfehlenswert, sie direkt zu verwenden, da sie nicht wirklich zufällig ist und daher vergleichsweise wenig Entropie aufweist. Stattdessen verwenden wir die integrierte Implementierung des PBKDF2-Algorithmus , um einen 128-Bit-Initialisierungsvektor und einen 256-Bit-Verschlüsselungsschlüssel aus dem Kennwort zu generieren.

Beachten Sie den zufälligen Salt-Wert, der für jede verschlüsselte Nachricht einen anderen Initialisierungsvektor und einen anderen Schlüssel benötigt. Dies stellt insbesondere sicher, dass zwei gleiche Nachrichten nicht zu identischem verschlüsseltem Text führen, es verhindert jedoch auch, dass Angreifer die Arbeit für das Erraten einer Passphrase für mit einer anderen Passphrase verschlüsselte Nachrichten erneut verwenden. Dieses Salt muss zusammen mit der verschlüsselten Nachricht gespeichert werden, um denselben Initialisierungsvektor und denselben Schlüssel für die Entschlüsselung abzuleiten.

Der folgende Code entschlüsselt unsere Nachricht erneut:

```
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:])
```

Generierung von RSA-Signaturen mit pycrypto

RSA kann zum Erstellen einer Nachrichtensignatur verwendet werden. Eine gültige Signatur kann nur mit Zugriff auf den privaten RSA-Schlüssel generiert werden, die Validierung ist dagegen nur mit dem entsprechenden öffentlichen Schlüssel möglich. Solange die andere Seite Ihren öffentlichen Schlüssel kennt, kann sie die von Ihnen zu signierende und unveränderte Nachricht überprüfen - ein Ansatz, der zum Beispiel für E-Mails verwendet wird. Derzeit ist für diese Funktionalität ein Drittanbieter-Modul wie pycrypto erforderlich.

```
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)
```

Das Überprüfen der Signatur funktioniert ähnlich, verwendet jedoch den öffentlichen Schlüssel anstelle des privaten Schlüssels:

```
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')
```

Hinweis : Die obigen Beispiele verwenden den Signaturalgorithmus PKCS # 1 v1.5, der sehr häufig vorkommt. pycrypto implementiert auch den neueren PKCS # 1-PSS-Algorithmus. `PKCS1_v1_5` durch `PKCS1_PSS` In den Beispielen sollte dies funktionieren, wenn Sie diesen verwenden möchten. Momentan scheint es jedoch wenig Grund zu geben, es zu benutzen .

Asymmetrische RSA-Verschlüsselung mit Pycrypto

Die asymmetrische Verschlüsselung hat den Vorteil, dass eine Nachricht verschlüsselt werden kann, ohne dass ein geheimer Schlüssel mit dem Empfänger der Nachricht ausgetauscht wird. Der Absender muss lediglich den öffentlichen Schlüssel des Empfängers kennen, wodurch die Nachricht so verschlüsselt werden kann, dass nur der angegebene Empfänger (der den entsprechenden privaten Schlüssel besitzt) die Nachricht entschlüsseln kann. Derzeit ist für diese Funktionalität ein Drittanbieter-Modul wie pycrypto erforderlich.

```
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)
```

Der Empfänger kann die Nachricht dann entschlüsseln, wenn er den richtigen privaten Schlüssel hat:

```
with open('privkey.pem', 'rb') as f:
    key = RSA.importKey(f.read())
cipher = PKCS1_OAEP.new(key)
decrypted = cipher.decrypt(encrypted)
```

Hinweis : Die obigen Beispiele verwenden das PKCS # 1-OAEP-Verschlüsselungsschema. pycrypto implementiert auch das Verschlüsselungsschema PKCS # 1 v1.5. Dieses Protokoll wird jedoch für neue Protokolle aufgrund bekannter Vorbehalte nicht empfohlen.

Sicherheit und Kryptographie online lesen: https://riptutorial.com/de/python/topic/2598/sicherheit-und-kryptographie

Kapitel 156: Sockets und Nachrichtenverschlüsselung / Entschlüsselung zwischen Client und Server

Einführung

Kryptographie wird aus Sicherheitsgründen verwendet. Es gibt nicht viele Beispiele für die Verschlüsselung / Entschlüsselung in Python mit der IDEA-Verschlüsselung MODE CTR. **Ziel dieser Dokumentation:**

Erweiterung und Implementierung des RSA Digital Signature-Schemas in der Stationskommunikation. Das Verwenden von Hashing für die Integrität einer Nachricht ist SHA-1. Erstellen Sie ein einfaches Schlüsseltransportprotokoll. Schlüssel mit IDEA-Verschlüsselung verschlüsseln. Der Blockcipher-Modus ist der Counter-Modus

Bemerkungen

Verwendete Sprache: Python 2.7 (Download-Link: https://www.python.org/downloads/)

Verwendete Bibliothek:

* **PyCrypto** (Download-Link: https://pypi.python.org/pypi/pycrypto)

* **PyCryptoPlus** (Download-Link: https://github.com/doegox/python-cryptoplus)

Bibliotheksinstallation:

PyCrypto: Entpacken Sie die Datei. Wechseln Sie in das Verzeichnis, und öffnen Sie das Terminal für Linux (Alt + Strg + T) und CMD (Umschalt + Rechtsklick + Eingabeaufforderung hier öffnen) für Windows. Danach schreiben Sie python setup.py install (Stellen Sie sicher, dass die Python-Umgebung im Windows-Betriebssystem richtig eingestellt ist).

PyCryptoPlus: Entspricht der letzten Bibliothek.

Aufgaben-Implementierung: Die Aufgabe ist in zwei Teile aufgeteilt. Einer ist ein Handshake-Prozess und ein anderer ist ein Kommunikationsprozess. Socket-Setup:

- Da das Erstellen öffentlicher und privater Schlüssel sowie das Hashing des öffentlichen Schlüssels erforderlich ist, müssen wir den Socket jetzt einrichten. Um das Socket einzurichten, müssen wir ein anderes Modul mit "Socket importieren" importieren und (für den Client) verbinden oder (für den Server) die IP-Adresse und den Port mit dem Socket verbinden, der vom Benutzer abgerufen wird.

  ---------- Client-Seite ----------

```
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))
```

---------- Serverseite ---------

```
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-----"
```

Mit "socket.AF_INET, socket.SOCK_STREAM" können wir die Funktion accept () und Messaging-Grundlagen verwenden. Anstelle dessen können wir auch "socket.AF_INET, socket.SOCK_DGRAM" verwenden, aber zu diesem Zeitpunkt müssen wir setblocking (value) verwenden.

Handshake-Prozess:

- (KUNDEN) Die erste Aufgabe besteht darin, einen öffentlichen und einen privaten Schlüssel zu erstellen. Um den privaten und den öffentlichen Schlüssel zu erstellen, müssen wir einige Module importieren. Sie sind: aus Crypto-Import Random und aus Crypto.PublicKey-Import RSA. Um die Schlüssel zu erstellen, müssen wir einige einfache Codezeilen schreiben:

```
random_generator = Random.new().read
        key = RSA.generate(1024,random_generator)
        public = key.publickey().exportKey()
```

random_generator wird vom Modul " **Crypto import Random** " abgeleitet. Der Schlüssel wird von " **from Crypto.PublicKey import RSA** " abgeleitet, der einen privaten Schlüssel mit der Größe 1024 erstellt, indem zufällige Zeichen generiert werden. Public exportiert den öffentlichen Schlüssel aus einem zuvor generierten privaten Schlüssel.

- (KUNDEN) Nachdem Sie den öffentlichen und den privaten Schlüssel erstellt haben, müssen Sie den öffentlichen Schlüssel mit einem SHA-1-Hash an den Server senden. Um den SHA-1-Hash verwenden zu können, müssen Sie ein anderes Modul importieren, indem Sie „import hashlib“ schreiben. Um den öffentlichen Schlüssel zu haschen, müssen wir zwei Codezeilen schreiben:

```
hash_object = hashlib.sha1(public)
hex_digest = hash_object.hexdigest()
```

Hier ist hash_object und hex_digest unsere Variable. Danach sendet der Client hex_digest und public an den Server, und der Server überprüft sie, indem er den vom Client erhaltenen Hash und den neuen Hash des öffentlichen Schlüssels vergleicht. Wenn der neue Hash und der Hash vom Client übereinstimmen, wird zur nächsten Prozedur übergegangen. Da das vom Client gesendete Publikum in Form einer Zeichenfolge vorliegt, kann es nicht als Schlüssel auf der Serverseite

verwendet werden. Um dies zu verhindern und den öffentlichen String des öffentlichen Zeichens in einen öffentlichen rsa-Schlüssel zu konvertieren, müssen Sie `server_public_key = RSA.importKey(getpbk)` schreiben. Hier ist getpbk der öffentliche Schlüssel des Clients.

- (SERVER) Als nächsten Schritt erstellen Sie einen Sitzungsschlüssel. Hier habe ich mit dem "os" -Modul einen zufälligen Schlüssel "key = os.urandom (16)" erstellt, der uns einen 16-Bit-langen Schlüssel geben wird. Danach habe ich diesen Schlüssel in "AES.MODE_CTR" verschlüsselt und ihn erneut hashiert mit 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()
```

Der en_digest wird also unser Sitzungsschlüssel sein.

- (SERVER) Der letzte Teil des Handshake-Prozesses besteht darin, den vom Client erhaltenen öffentlichen Schlüssel und den auf dem Server erstellten Sitzungsschlüssel zu verschlüsseln.

```
 #encrypting session key and public key
 E = server_public_key.encrypt(encrypto,16)
```

Nach der Verschlüsselung sendet der Server den Schlüssel als Zeichenfolge an den Client.

- (CLIENT) Nachdem der verschlüsselte String (public und session key) vom Server abgerufen wurde, entschlüsselt der Client diese mit dem zuvor mit dem öffentlichen Schlüssel erstellten privaten Schlüssel. Da der verschlüsselte (öffentliche und Sitzungsschlüssel) in Form einer Zeichenfolge vorliegt, müssen wir ihn jetzt mithilfe von eval () als Schlüssel zurückholen. Wenn die Entschlüsselung abgeschlossen ist, ist der Handshake-Vorgang abgeschlossen, da beide Seiten bestätigen, dass sie dieselben Schlüssel verwenden. Zu entschlüsseln:

```
 en = eval(msg)
 decrypt = key.decrypt(en)
 # hashing sha1
 en_object = hashlib.sha1(decrypt) en_digest = en_object.hexdigest()
```

Ich habe den SHA-1 hier verwendet, damit er in der Ausgabe lesbar ist.

Kommunikationsprozess:

Für den Kommunikationsprozess müssen wir den Sitzungsschlüssel von beiden Seiten als Schlüssel für die IDEA-Verschlüsselung MODE_CTR verwenden. Beide Seiten verschlüsseln und entschlüsseln Nachrichten mit IDEA.MODE_CTR unter Verwendung des Sitzungsschlüssels.

- (Verschlüsselung) Für die IDEA-Verschlüsselung benötigen wir einen 16bit großen Schlüssel und einen Zähler, der aufrufbar ist. Zähler ist in MODE_CTR obligatorisch. Der von uns

verschlüsselte und gehashte Sitzungsschlüssel hat jetzt eine Größe von 40, was den Grenzwert der IDEA-Verschlüsselung überschreitet. Daher müssen wir die Größe des Sitzungsschlüssels reduzieren. Zur Reduzierung können wir den in Python eingebauten Funktionsstring [value: value] verwenden. Dabei kann der Wert je nach Wahl des Benutzers ein beliebiger Wert sein. In unserem Fall habe ich "key [: 16]" vorgenommen, wobei der Schlüssel aus 0 bis 16 Werten genommen wird. Diese Konvertierung kann auf viele Arten durchgeführt werden, z. B. Schlüssel [1:17] oder Schlüssel [16:]. Der nächste Teil ist das Erstellen einer neuen IDEA-Verschlüsselungsfunktion, indem IDEA.new () geschrieben wird, das 3 Argumente für die Verarbeitung benötigt. Das erste Argument ist KEY, das zweite Argument ist der Modus der IDEA-Verschlüsselung (in unserem Fall IDEA.MODE_CTR) und das dritte Argument ist der counter =, der eine aufrufbare Funktion ist. Der Zähler = enthält eine Stringgröße, die von der Funktion zurückgegeben wird. Um den Zähler zu definieren, müssen wir vernünftige Werte verwenden. In diesem Fall habe ich die Größe des Schlüssels verwendet, indem ich Lambda definiert habe. Anstatt Lambda zu verwenden, können wir Counter.Util verwenden, das einen Zufallswert für counter = generiert. Um Counter.Util verwenden zu können, müssen wir das Counter-Modul aus Crypto importieren. Daher lautet der Code:

```
ideaEncrypt = IDEA.new(key, IDEA.MODE_CTR, counter=lambda : key)
```

Nachdem wir "ideaEncrypt" als IDEA-Verschlüsselungsvariable definiert haben, können wir die integrierte Verschlüsselungsfunktion verwenden, um jede Nachricht zu verschlüsseln.

```
eMsg = ideaEncrypt.encrypt(whole)
#converting the encrypted message to HEXADECIMAL to readable eMsg =
eMsg.encode("hex").upper()
```

In diesem Codesegment ist "Ganz" die zu verschlüsselnde Nachricht und "eMsg" die verschlüsselte Nachricht. Nach dem Verschlüsseln der Nachricht habe ich sie in HEXADECIMAL konvertiert, um sie lesbar zu machen, und upper () ist die eingebaute Funktion, um die Zeichen in Großbuchstaben zu schreiben. Danach wird diese verschlüsselte Nachricht zur Entschlüsselung an die Gegenstation gesendet.

- **(Entschlüsselung)**

Um die verschlüsselten Nachrichten zu entschlüsseln, müssen Sie eine weitere Verschlüsselungsvariable erstellen, indem Sie dieselben Argumente und denselben Schlüssel verwenden. Diesmal entschlüsselt die Variable jedoch die verschlüsselten Nachrichten. Der Code hierfür entspricht dem letzten Mal. Vor dem Entschlüsseln der Nachrichten müssen wir die Nachricht jedoch hexadezimal decodieren, da in unserem Verschlüsselungsteil die verschlüsselte Nachricht hexadezimal codiert wurde, um lesbar zu sein. Daher wird der gesamte Code sein:

```
decoded = newmess.decode("hex")
ideaDecrypt = IDEA.new(key, IDEA.MODE_CTR, counter=lambda: key)
dMsg = ideaDecrypt.decrypt(decoded)
```

Diese Prozesse werden sowohl auf der Server- als auch auf der Clientseite zum Verschlüsseln und Entschlüsseln ausgeführt.

Examples

Serverseitige Implementierung

```
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")
```

Client-seitige Implementierung

```
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()
```

Sockets und Nachrichtenverschlüsselung / Entschlüsselung zwischen Client und Server online lesen: https://riptutorial.com/de/python/topic/8710/sockets-und-nachrichtenverschlusselung---entschlusselung-zwischen-client-und-server

Kapitel 157: Sortierung, Minimum und Maximum

Examples

Holen Sie sich das Minimum oder Maximum von mehreren Werten

```
min(7,2,1,5)
# Output: 1

max(7,2,1,5)
# Output: 7
```

Verwenden Sie das Schlüsselargument

Das Finden des Minimums / Maximums einer Sequenz von Sequenzen ist möglich:

```
list_of_tuples = [(0, 10), (1, 15), (2, 8)]
min(list_of_tuples)
# Output: (0, 10)
```

Wenn Sie jedoch in jeder Sequenz nach einem bestimmten Element sortieren möchten, verwenden Sie das `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)]
```

Default Argument auf max, min

Sie können keine leere Sequenz in `max` oder `min` :

```
min([])
```

ValueError: min () arg ist eine leere Sequenz

Mit Python 3 können Sie jedoch das Schlüsselwortargument `default` mit einem Wert übergeben, der zurückgegeben wird, wenn die Sequenz leer ist, anstatt eine Ausnahme auszulösen:

```
max([], default=42)
# Output: 42
max([], default=0)
# Output: 0
```

Sonderfall: Wörterbücher

Das Ermitteln des Minimums oder Maximums oder die Verwendung der `sorted` hängt von den Iterationen des Objekts ab. Bei `dict` erfolgt die Iteration nur über den Tasten:

```
adict = {'a': 3, 'b': 5, 'c': 1}
min(adict)
# Output: 'a'
max(adict)
# Output: 'c'
sorted(adict)
# Output: ['a', 'b', 'c']
```

Um die Wörterbuchstruktur `.items()` , müssen Sie die `.items()` :

```
min(adict.items())
# Output: ('a', 3)
max(adict.items())
# Output: ('c', 1)
sorted(adict.items())
# Output: [('a', 3), ('b', 5), ('c', 1)]
```

Für `sorted` , können Sie eine erstellen `OrderedDict` die Sortierung zu halten , während eine mit `dict` -ähnlichen Struktur:

```
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
```

Nach Wert

Dies ist wiederum mit dem `key` möglich:

```
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)]
```

Eine sortierte Reihenfolge erhalten

Verwendung **einer** Sequenz:

```
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']
```

Das Ergebnis ist immer eine neue `list` . Die ursprünglichen Daten bleiben unverändert.

Minimum und Maximum einer Sequenz

Das Minimum einer Sequenz (iterierbar) zu erhalten, entspricht dem Zugriff auf das erste Element einer `sorted` Sequenz:

```
min([2, 7, 5])
# Output: 2
sorted([2, 7, 5])[0]
# Output: 2
```

Das Maximum ist etwas komplizierter, weil `sorted` die Reihenfolge hält und `max` den zuerst gefundenen Wert zurückgibt. Falls keine Duplikate vorhanden sind, entspricht das Maximum dem letzten Element der sortierten Rückgabe:

```
max([2, 7, 5])
# Output: 7
sorted([2, 7, 5])[-1]
# Output: 7
```

Nicht jedoch, wenn mehrere Elemente mit dem Maximalwert bewertet werden:

```
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
```

Jede Iteration, die Elemente enthält, die `<` oder `>` Operationen unterstützen, ist zulässig.

Machen Sie benutzerdefinierte Klassen bestellbar

`min` , `max` und `sorted` alle Objekte bestellbar sein. Um korrekt bestellbar zu sein, muss die Klasse alle 6 Methoden `__lt__` , `__gt__` , `__ge__` , `__le__` , `__ne__` und `__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
```

Die Implementierung all dieser Methoden erscheint zwar unnötig, aber einige davon wegzulassen, führt dazu, dass Ihr Code anfällig für Fehler wird .

Beispiele:

```
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)]
```

mit `reverse=True` `sorted` `reverse=True` verwendet auch `__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)]
```

`sorted` kann `__gt__` stattdessen `__gt__` verwenden, wenn der Standard nicht implementiert ist:

```
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)
```

Sortiermethoden `TypeError` einen `TypeError` wenn weder `__lt__` noch `__gt__` implementiert sind:

```
del IntegerContainer.__gt__   # The IntegerContainer no longer implements "greater then"

res = min(alist)
```

TypeError: nicht anordnungsfähige Typen: IntegerContainer () <IntegerContainer ()

`functools.total_ordering` Dekorator `functools.total_ordering` kann verwendet werden, um die Erstellung dieser umfangreichen Vergleichsmethoden zu vereinfachen. Wenn Sie Ihre Klasse mit `total_ordering` , müssen Sie `__eq__` , `__ne__` und nur eines von `__lt__` , `__le__` , `__ge__` oder `__gt__` . Der Dekorateur füllt den Rest aus:

```
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
```

Beachten Sie, wie die `>` (*größer als*) jetzt die Methode *less als* und in einigen Fällen sogar die Methode `__eq__` . Das bedeutet auch, dass, wenn Geschwindigkeit von großer Bedeutung ist, Sie jede ausführliche Vergleichsmethode selbst implementieren sollten.

Extrahieren von N größten oder N kleinsten Elementen aus einer iterierbaren

Um eine bestimmte Anzahl (mehr als einen) der größten oder kleinsten Werte einer Iteration zu finden, können Sie die `nlargest` und `nsmallest` des `heapq` Moduls verwenden:

```
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]
```

Dies ist viel effizienter, als das gesamte Iterable zu sortieren und dann am Ende oder Anfang zu schneiden. Intern verwenden diese Funktionen die Datenstruktur der binären Heap-Prioritätswarteschlange , die für diesen Anwendungsfall sehr effizient ist.

Wie `min` , `max` und `sorted` akzeptieren diese Funktionen das optionale `key` , das eine Funktion sein muss, die bei einem Element seinen Sortierschlüssel zurückgibt.

Hier ist ein Programm, das 1000 längste Zeilen aus einer Datei extrahiert:

```
import heapq
with open(filename) as f:
    longest_lines = heapq.nlargest(1000, f, key=len)
```

Hier öffnen wir die Datei und übergeben das `nlargest f` an `nlargest` . Durch die Iteration der Datei wird jede Zeile der Datei als separate Zeichenfolge angezeigt. `nlargest` dann jedes Element (oder jede Zeile) an die Funktion `len` , um den Sortierschlüssel zu bestimmen. `len` Angabe einer Zeichenfolge gibt `len` die Länge der Zeile in Zeichen zurück.

Dies erfordert nur Speicherplatz für eine Liste von 1000 größten Zeilen, die mit denen verglichen werden können

```
longest_lines = sorted(f, key=len)[1000:]
```

die muss *die gesamte Datei im Speicher halten* .

Sortierung, Minimum und Maximum online lesen:
https://riptutorial.com/de/python/topic/252/sortierung--minimum-und-maximum

Kapitel 158: Sqlite3-Modul

Examples

Sqlite3 - Kein separater Serverprozess erforderlich.

Das Modul sqlite3 wurde von Gerhard Häring geschrieben. Um das Modul verwenden zu können, müssen Sie zuerst ein Connection-Objekt erstellen, das die Datenbank darstellt. Hier werden die Daten in der Datei example.db gespeichert:

```
import sqlite3
conn = sqlite3.connect('example.db')
```

Sie können auch den speziellen Namen angeben: memory: Zum Erstellen einer Datenbank im RAM. Sobald Sie eine Verbindung haben, können Sie ein Cursor-Objekt erstellen und seine Methode execute () aufrufen, um SQL-Befehle auszuführen:

```
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()
```

Abrufen der Werte aus der Datenbank und Fehlerbehandlung

Abrufen der Werte aus der SQLite3-Datenbank.

Zeilenwerte drucken, die von Auswahlabfragen zurückgegeben werden

```
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
```

Fetchone () -Methode abrufen

```
print c.fetchone()
```

Verwenden Sie für mehrere Zeilen die Methode fetchall ()

```
a=c.fetchall() #which is similar to list(cursor) method used previously
for row in a:
    print row
```

Fehlerbehandlung kann mithilfe der integrierten Funktion sqlite3.Error ausgeführt werden

```
try:
    #SQL Code
except sqlite3.Error as e:
    print "An error occurred:", e.args[0]
```

Sqlite3-Modul online lesen: https://riptutorial.com/de/python/topic/7754/sqlite3-modul

Kapitel 159: Stapel

Einführung

Ein Stapel ist ein Container mit Objekten, die nach dem LIFO-Prinzip (Last-in-First-Out) eingefügt und entfernt werden. In den Pushdown-Stapeln sind nur zwei Vorgänge zulässig: **Schieben Sie den Gegenstand in den Stapel und ziehen Sie den Gegenstand aus dem Stapel** . Ein Stapel ist eine begrenzte Zugriffsdatenstruktur - **Elemente können nur oben hinzugefügt und aus dem Stapel entfernt werden** . Hier ist eine strukturelle Definition eines Stapels: Ein Stapel ist entweder leer oder besteht aus einer Oberseite und dem Rest, der ein Stapel ist.

Syntax

- stack = [] # Erstellen Sie den Stapel
- stack.append (object) # Fügt ein Objekt am oberen Rand des Stapels hinzu
- stack.pop () -> object # Gibt das oberste Objekt aus dem Stapel zurück und entfernt es auch
- list [-1] -> object # Sehen Sie das oberste Objekt an, ohne es zu entfernen

Bemerkungen

Von Wikipedia :

> In der Informatik ist ein *Stapel* ein abstrakter Datentyp, der als Auflistung von Elementen dient, mit zwei Hauptoperationen: *Push* (Hinzufügen), um ein Element zur Auflistung hinzuzufügen, und *Pop* (*Pop)* , um das zuletzt hinzugefügte Element zu entfernen, das noch nicht entfernt wurde.

Aufgrund des Zugriffs auf ihre Elemente werden Stapel auch als *Last-In-, First-Out-* (*LIFO*) - *Stapel bezeichnet* .

In Python kann man Listen als Stapel verwenden, mit `append()` als Push und `pop()` als Popoperationen. Beide Operationen laufen in konstanter Zeit O (1).

Die `deque` Datenstruktur des Python kann auch als Stapel verwendet werden. Im Vergleich zu Listen ermöglichen `deque` s Push- und Pop-Operationen mit konstanter zeitlicher Komplexität von beiden Seiten.

Examples

Erstellen einer Stack-Klasse mit einem Listenobjekt

Ein Verwenden `list` Objekt können Sie einen voll funktionsfähigen generic Stapel mit Hilfsmethoden wie spähen und die Überprüfung erstellen , wenn der Stapel leer ist. Schauen Sie sich die offiziellen Python-Dokumente für die Verwendung der `list` als `Stack` hier an .

```
#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
```

Ein Beispiellauf:

```
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())
```

Ausgabe:

```
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
```

Parsen von Klammern

Stapel werden häufig zum Parsen verwendet. Eine einfache Parsing-Aufgabe besteht darin, zu prüfen, ob eine Klammerfolge übereinstimmt.

Zum Beispiel stimmt die Zeichenfolge `([])` überein, da die äußeren und inneren Klammern Paare bilden. `()<>)` stimmt nicht überein, da der letzte `)` keinen Partner hat. `([)]` stimmt auch nicht überein, da Paare entweder vollständig innerhalb oder außerhalb anderer Paare sein müssen.

```
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()
```

Stapel online lesen: https://riptutorial.com/de/python/topic/3807/stapel

Kapitel 160: Steckdosen

Einführung

Viele Programmiersprachen verwenden Sockets für die Kommunikation zwischen Prozessen oder zwischen Geräten. In diesem Thema wird die Verwendung des Sockets-Moduls in Python erläutert, um das Senden und Empfangen von Daten über gängige Netzwerkprotokolle zu vereinfachen.

Parameter

Parameter	Beschreibung
socket.AF_UNIX	UNIX-Socket
socket.AF_INET	IPv4
socket.AF_INET6	IPv6
socket.SOCK_STREAM	TCP
socket.SOCK_DGRAM	UDP

Examples

Senden von Daten über UDP

UDP ist ein verbindungsloses Protokoll. Nachrichten an andere Prozesse oder Computer werden gesendet, ohne dass irgendeine Verbindung hergestellt wird. Es erfolgt keine automatische Bestätigung, wenn Ihre Nachricht empfangen wurde. UDP wird normalerweise in latenzempfindlichen Anwendungen oder in Anwendungen verwendet, die Broadcasts im gesamten Netzwerk senden.

Mit dem folgenden Code wird eine Nachricht an einen Prozess gesendet, der mit UDP auf dem Localhost-Port 6667 überwacht wird

Beachten Sie, dass der Socket nach dem Senden nicht "geschlossen" werden muss, da UDP verbindungslos ist .

```
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))
```

Daten über UDP empfangen

UDP ist ein verbindungsloses Protokoll. Dies bedeutet, dass Peers, die Nachrichten senden, vor dem Senden von Nachrichten keine Verbindung herstellen müssen. `socket.recvfrom` also ein Tupel zurück (`msg` [die Nachricht, die der Socket empfangen hat], `addr` [die Adresse des Absenders])

Ein UDP-Server, der ausschließlich das `socket` Modul verwendet:

```
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))
```

Nachfolgend finden Sie eine alternative Implementierung mit `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()
```

Standardmäßig blockieren `sockets` . Dies bedeutet, dass die Ausführung des Skripts wartet, bis der Socket Daten empfängt.

Senden von Daten über TCP

Der Versand von Daten über das Internet wird durch mehrere Module ermöglicht. Das Sockets-Modul ermöglicht einen einfachen Zugriff auf die zugrunde liegenden Betriebssystemvorgänge, die für das Senden oder Empfangen von Daten von anderen Computern oder Prozessen verantwortlich sind.

Der folgende Code sendet die `b'Hello'` an einen TCP-Server, der Port 6667 auf dem Host localhost `b'Hello'` und schließt die Verbindung, wenn er fertig ist:

```
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()
```

Die Socket-Ausgabe blockiert standardmäßig, dh das Programm wartet auf die Verbindung und sendet Aufrufe, bis die Aktion abgeschlossen ist. Für connect bedeutet dies, dass der Server die Verbindung tatsächlich annimmt. Für das Senden bedeutet dies nur, dass das Betriebssystem

über genügend Pufferplatz verfügt, um die später zu sendenden Daten in die Warteschlange zu stellen.

Steckdosen sollten nach Gebrauch immer geschlossen sein.

TCP-Socket-Server mit mehreren Threads

Wenn es ohne Argumente ausgeführt wird, startet dieses Programm einen TCP-Socket-Server, der auf Port `5000` Verbindungen zu `127.0.0.1 5000` . Der Server behandelt jede Verbindung in einem separaten Thread.

Wenn dieses Programm mit dem Argument `-c` , stellt es eine Verbindung zum Server her, liest die Clientliste und druckt es aus. Die Clientliste wird als JSON-String übertragen. Der Clientname kann angegeben werden, indem das Argument `-n` . Durch die Übergabe verschiedener Namen kann der Effekt auf die Kundenliste beobachtet werden.

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()
```

Server-Ausgabe

```
$ python client_list.py
Starting server...
```

Client-Ausgabe

```
$ python client_list.py -c -n name1
{
    "name1": {
        "address": "127.0.0.1",
        "port": 62210,
        "name": "name1"
    }
}
```

Die Empfangspuffer sind auf 1024 Byte begrenzt. Wenn die JSON-Zeichenfolgendarstellung der Clientliste diese Größe überschreitet, wird sie abgeschnitten. Dadurch wird die folgende Ausnahme ausgelöst:

```
ValueError: Unterminated string starting at: line 1 column 1023 (char 1022)
```

Raw Sockets unter Linux

Deaktivieren Sie zunächst die automatische Prüfsumme Ihrer Netzwerkkarte:

```
sudo ethtool -K eth1 tx off
```

Senden Sie dann Ihr Paket mit einem SOCK_RAW-Socket:

```
#!/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)
```

Steckdosen online lesen: https://riptutorial.com/de/python/topic/1530/steckdosen

Kapitel 161: String-Formatierung

Einführung

Beim Speichern und Umwandeln von Daten, die der Mensch sehen kann, kann die Formatierung von Strings sehr wichtig werden. Python bietet eine Vielzahl von String-Formatierungsmethoden, die in diesem Thema beschrieben werden.

Syntax

- "{}". format (42) ==> "42"
- "{0}". Format (42) ==> "42"
- "{0: .2f}". Format (42) ==> "42.00"
- "{0: .0f}". Format (42.1234) ==> "42"
- "{answer}". format (no_answer = 41, antwort = 42) ==> "42"
- "{answer: .2f}". format (no_answer = 41, antwort = 42) ==> "42.00"
- "{[key]}". format ({'key': 'value'}) ==> "value"
- "{[1]}". Format (['null', 'eins', 'zwei']) ==> "eins"
- "{Antwort} = {Antwort}". Format (Antwort = 42) ==> "42 = 42"
- " .join (['stack', 'overflow']) ==> "stack overflow"

Bemerkungen

- Schauen Sie sich PyFormat.info an, um eine sehr gründliche und sanfte Einführung / Erklärung zu erhalten.

Examples

Grundlagen der String-Formatierung

```
foo = 1
bar = 'bar'
baz = 3.14
```

Sie können `str.format` um die Ausgabe zu formatieren. Klammerpaare werden in der Reihenfolge, in der die Argumente übergeben werden, durch Argumente ersetzt:

```
print('{}, {} and {}'.format(foo, bar, baz))
# Out: "1, bar and 3.14"
```

In den Klammern können auch Indizes angegeben werden. Die Zahlen entsprechen den Indizes der Argumente, die an die Funktion `str.format` (0-basiert) übergeben wurden.

```
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
```

Benannte Argumente können auch verwendet werden:

```
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."
```

`str.format` können bei der `str.format` an `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"
```

Wörterbuchschlüssel können ebenfalls verwendet werden:

```
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"
```

Gleiches gilt für Listen- und Tupelindizes:

```
my_list = ['zero', 'one', 'two']
print("2nd element is: {0[2]}".format(my_list))  # "0" is optional
# Out: "2nd element is: two"
```

> Hinweis: Zusätzlich zu `str.format` bietet Python den Modulo-Operator `%` - auch als *String-Formatierungs-* oder *Interpolationsoperator* (siehe PEP 3101) - zum Formatieren von Strings an. `str.format` ist ein Nachfolger von `%` und bietet mehr Flexibilität, beispielsweise durch die Erleichterung mehrerer Substitutionen.

Neben den Argumentindizes können Sie auch eine *Formatangabe* in die geschweiften Klammern einfügen. Dies ist ein Ausdruck, die besonderen Regeln folgt und muss einen Doppelpunkt vorangestellt werden (`:`). Eine vollständige Beschreibung der Formatangaben finden Sie in den Dokumenten . Ein Beispiel für eine Formatspezifikation ist die Ausrichtungsanweisung `:~^20` (`^` steht für Zentrumsausrichtung, Gesamtbreite 20, mit ~ Zeichen füllen):

```
'{:~^20}'.format('centered')
# Out: '~~~~~~centered~~~~~~'
```

`format` erlaubt Verhalten, das mit `%` nicht möglich ist, z. B. Wiederholung von Argumenten:

```
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
```

Da `format` eine Funktion ist, kann es als Argument in anderen Funktionen verwendet werden:

```
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
```

Ausrichtung und Polsterung

Python 2.x 2.6

Die `format()` -Methode kann verwendet werden, um die Ausrichtung der Zeichenfolge zu ändern. Sie müssen dies mit einem `:[fill_char][align_operator][width]` des Formulars tun `:[fill_char][align_operator][width]` Dabei steht `align_operator` für `align_operator` :

- `<` Erzwingt, dass das Feld innerhalb der `width` nach links ausgerichtet wird.
- `>` zwingt das Feld innerhalb der `width` nach rechts auszurichten.
- `^` zwingt das Feld innerhalb der `width` zu zentrieren.
- `=` zwingt die Auffüllung nach dem Zeichen (nur numerische Typen).

`fill_char` (wenn nicht angegeben, ist der Whitespace-Standardwert) das Zeichen, das für die Auffüllung verwendet wird.

```
'{:~<9s}, World'.format('Hello')
# 'Hello~~~~, World'

'{:~>9s}, World'.format('Hello')
# '~~~~Hello, World'

'{:~^9s}'.format('Hello')
# '~~Hello~~'

'{:0=6d}'.format(-123)
# '-00123'
```

Anmerkung: Sie können die gleichen Ergebnisse mit den String-Funktionen `ljust()` , `rjust()` , `center()` , `zfill()` Diese Funktionen werden jedoch seit Version 2.5 nicht mehr unterstützt.

Format Literale (F-String)

Zeichenketten im Literalformat wurden in PEP 498 (Python3.6 und höher) eingeführt, sodass Sie `f` am Anfang eines Zeichenfolgenlitals voranstellen können, um das `.format` effektiv auf alle Variablen im aktuellen Gültigkeitsbereich anzuwenden.

```
>>> foo = 'bar'
>>> f'Foo is {foo}'
'Foo is bar'
```

Dies funktioniert auch mit erweiterten Formatzeichenfolgen, einschließlich Ausrichtung und Punktnotation.

```
>>> f'{foo:^7s}'
'  bar  '
```

Hinweis: Das `f''` bezeichnet keinen bestimmten Typ wie `b''` für `bytes` oder `u''` für `unicode` in Python2. Die Formatierung wird sofort angewendet, was zu einer normalen Stabilisierung führt.

Die Formatstrings können auch *geschachtelt sein* :

```
>>> price = 478.23
>>> f"{f'${price:0.2f}':*>20s}"
'*************$478.23'
```

Die Ausdrücke in einem F-String werden in der Reihenfolge von links nach rechts ausgewertet. Dies ist nur erkennbar, wenn die Ausdrücke Nebenwirkungen haben:

```
>>> 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]
```

String-Formatierung mit datetime

Jede Klasse kann ihre eigene String-Formatierungssyntax mit der `__format__` Methode `__format__` . Ein Typ in der Python-Standardbibliothek, der dies praktisch nutzt, ist der `datetime` Typ, bei dem `datetime strftime` Formatierungscodes direkt in `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.'
```

Eine vollständige Liste der Liste der Datumsformatierungsformate finden Sie in der offiziellen Dokumentation .

Formatieren Sie mit Getitem und Getattr

Für jede Datenstruktur, die `__getitem__` unterstützt, kann die verschachtelte Struktur formatiert werden:

```
person = {'first': 'Arthur', 'last': 'Dent'}
'{p[first]} {p[last]}'.format(p=person)
# 'Arthur Dent'
```

Auf `getattr()` kann mit `getattr()` zugegriffen werden:

```
class Person(object):
    first = 'Zaphod'
    last = 'Beeblebrox'

'{p.first} {p.last}'.format(p=Person())
# 'Zaphod Beeblebrox'
```

Float-Formatierung

```
>>> '{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'
```

Gleiches gilt für andere Verweise:

```
>>> '{:.3f}'.format(42.12345)
'42.123'

>>> '{answer:.3f}'.format(answer=42.12345)
'42.123'
```

Fließkommazahlen können auch in wissenschaftlicher Notation oder als Prozentsatz formatiert werden:

```
>>> '{0:.3e}'.format(42.12345)
'4.212e+01'

>>> '{0:.0%}'.format(42.12345)
'4212%'
```

Sie können auch die Notizen `{0}` und `{name}` kombinieren. Dies ist besonders nützlich, wenn Sie

alle Variablen mit einer *Deklaration* auf eine vordefinierte Anzahl von Dezimalstellen runden möchten:

```
>>> 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'
```

Numerische Werte formatieren

Die `.format()` -Methode kann eine Zahl in verschiedenen Formaten interpretieren, z.

```
>>> '{:c}'.format(65)    # Unicode character
'A'

>>> '{:d}'.format(0x0a)  # base 10
'10'

>>> '{:n}'.format(0x0a)  # base 10 using current locale for separators
'10'
```

Formatieren von Ganzzahlen auf verschiedene Basen (Hex, Okt, Binär)

```
>>> '{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'
```

Verwenden Sie die Formatierung, um ein RGB-Float-Tupel in einen Farb-Hex-String zu konvertieren:

```
>>> r, g, b = (1.0, 0.4, 0.0)
>>> '#{:02X}{:02X}{:02X}'.format(int(255 * r), int(255 * g), int(255 * b))
'#FF6600'
```

Nur ganze Zahlen können konvertiert werden:

```
>>> '{: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'
```

Benutzerdefinierte Formatierung für eine Klasse

Hinweis:

> Alles gilt unten an der `str.format` Methode sowie die `format` - Funktion. Im folgenden Text sind die beiden austauschbar.

Für jeden Wert, der den übergeben `format` Funktion sucht Python für eine `__format__` Methode für dieses Argument. Ihre eigene benutzerdefinierte Klasse kann daher ihre eigene hat `__format__` Methode , um zu bestimmen , wie die `format` - Funktion wird angezeigt und Ihre Klasse formatiert und es ist Attribut.

Dies unterscheidet sich von der `__str__` Methode, da Sie bei der `__format__` Methode die Formatierungssprache, einschließlich Ausrichtung, `__format__` usw., berücksichtigen können und sogar (wenn Sie möchten) eigene Formatbezeichner und Ihre eigenen Formatierungssprachenerweiterungen implementieren. 1

```
object.__format__(self, format_spec)
```

Zum Beispiel :

```
# 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.
```

Hinweis:

> Wenn Ihre benutzerdefinierte Klasse nicht über eine benutzerdefinierte `__format__`

Methode verfügt und eine Instanz der Klasse an die `format` Funktion **übergeben** wird, verwendet **Python2** immer den Rückgabewert der `__str__` Methode oder der `__str__` `__repr__` Methode, um zu bestimmen, was gedruckt werden soll Standard `repr` verwendet), und Sie werden das verwenden müssen `s` Formatbezeichner formatiert werden diese. Mit **Python3,** Ihre benutzerdefinierten Klasse zur passieren `format` - Funktion, müssen Sie definieren `__format__` Methode auf Ihrer benutzerdefinierte Klasse.

Verschachtelte Formatierung

Einige Formate können zusätzliche Parameter enthalten, z. B. die Breite der formatierten Zeichenfolge oder die Ausrichtung:

```
>>> '{:.>10}'.format('foo')
'.......foo'
```

Diese können auch als Parameter zum `format` bereitgestellt werden, indem Sie mehr `{}` in `{}` verschachteln:

```
>>> '{:.>{}}'.format('foo', 10)
'.......foo'
'{:{}{}{}}'.format('foo', '*', '^', 15)
'******foo******'
```

Im letzteren Beispiel wird die Formatzeichenfolge `'{:{}{}{}}'` in `'{:*^15}'` (dh "center und pad mit * bis Gesamtlänge 15") geändert, bevor sie auf das Format angewendet wird tatsächliche Zeichenfolge `'foo'` , um auf diese Weise formatiert zu werden.

Dies kann nützlich sein, wenn Parameter vorher nicht bekannt sind, z. B. beim Ausrichten von Tabellendaten:

```
>>> data = ["a", "bbbbbbb", "ccc"]
>>> m = max(map(len, data))
>>> for d in data:
...     print('{:>{}}'.format(d, m))
      a
bbbbbbb
    ccc
```

Saiten auffüllen und abschneiden, kombiniert

Angenommen, Sie möchten Variablen in einer Spalte mit 3 Zeichen drucken.

Hinweis: Das Verdoppeln `{` und `}` entgeht ihnen.

```
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))
```

Ausgabe:

```
pad
{:3}             :1  :

truncate
{:.3}            :555:

combined
{:>3.3}          :  1:
{:3.3}           :1  :
{:3.3}           :333:
{:3.3}           :555:
```

Benannte Platzhalter

Formatstrings können benannte Platzhalter enthalten, die zum `format` mit Schlüsselwortargumenten interpoliert werden.

Wörterbuch verwenden (Python 2.x)

```
>>> data = {'first': 'Hodor', 'last': 'Hodor!'}
>>> '{first} {last}'.format(**data)
'Hodor Hodor!'
```

Wörterbuch verwenden (Python 3.2+)

```
>>> '{first} {last}'.format_map(data)
'Hodor Hodor!'
```

`str.format_map` können Wörterbücher verwendet werden, ohne sie vorher entpacken zu müssen. Anstelle eines neu gefüllten `dict` auch die `data` (möglicherweise ein benutzerdefinierter Typ) verwendet.

Ohne Wörterbuch:

```
>>> '{first} {last}'.format(first='Hodor', last='Hodor!')
'Hodor Hodor!'
```

String-Formatierung online lesen: https://riptutorial.com/de/python/topic/1019/string-formatierung

Kapitel 162: String-Methoden

Syntax

- str.capitalize () -> str
- str.casefold () -> str [nur für Python> 3.3]
- str.center (width [, fillchar]) -> str
- str.count (sub [, start [, end]]) -> int
- str.decode (coding = "utf-8" [, Fehler]) -> Unicode [nur in Python 2.x]
- str.encode (Kodierung = "utf-8", Fehler = "streng") -> Bytes
- str.endswith (Suffix [, Anfang [, Ende]]) -> 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 ([Zeichen]) -> str
- statische Str.maketrans (x [, y [, z]])
- str.partition (sep) -> (kopf, sep, schwanz)
- str.replace (alt, neu [, Anzahl]) -> str
- str.rfind (sub [, start [, end]]) -> int
- str.rindex (sub [, start [, end]]) -> int
- str.just (width [, fillchar]) -> str
- str.rpartition (sep) -> (kopf, sep, schwanz)
- str.rsplit (sep = None, maxsplit = -1) -> Liste der Strings
- str.rstrip ([Zeichen]) -> str
- str.split (sep = None, maxsplit = -1) -> Liste der Strings
- str.splitlines ([keepends]) -> Liste der Strings
- str.startswith (Präfix [, Start [, Ende]]) -> book
- str.strip ([Zeichen]) -> str

- str.swapcase () -> str
- str.title () -> str
- str.translate (table) -> str
- str.upper () -> str
- str.zfill (width) -> str

Bemerkungen

String-Objekte sind unveränderlich, was bedeutet, dass sie nicht so geändert werden können, wie dies mit einer Liste möglich ist. Aus diesem Grund geben Methoden des integrierten Typs `str` immer ein **neues** `str` Objekt zurück, das das Ergebnis des Methodenaufrufs enthält.

Examples

Ändern der Großschreibung einer Zeichenfolge

Der String-Typ von Python bietet viele Funktionen, die sich auf die Großschreibung einer Zeichenfolge beziehen. Diese schließen ein :

- `str.casefold`
- `str.upper`
- `str.lower`
- `str.capitalize`
- `str.title`
- `str.swapcase`

Bei Unicode-Zeichenfolgen (der Standardeinstellung in Python 3) sind diese Vorgänge **nicht** 1: 1-Zuordnungen oder umkehrbar. Die meisten dieser Operationen sind eher zur Anzeige als zur Normalisierung gedacht.

Python 3.x 3.3

`str.casefold()`

`str.casefold` erstellt eine Zeichenfolge in Kleinbuchstaben, die für Vergleiche ohne `str.casefold` der `str.casefold` und Kleinschreibung geeignet ist. Dies ist aggressiver als `str.lower` und kann Zeichenfolgen ändern, die bereits in Kleinbuchstaben geschrieben sind oder die Länge der Zeichenfolgen `str.lower` , und ist nicht für Anzeigezwecke gedacht.

```
"XßΣ".casefold()
# 'xssσ'

"XßΣ".lower()
# 'xßς'
```

Die unter Casefolding stattfindenden Umwandlungen werden vom Unicode-Konsortium in der Datei CaseFolding.txt auf ihrer Website definiert.

`str.upper()`

`str.upper` nimmt jedes Zeichen in einer Zeichenfolge und konvertiert es in seine entsprechenden Großbuchstaben. Beispiel:

```
"This is a 'string'.".upper()
# "THIS IS A 'STRING'."
```

`str.lower()`

`str.lower` macht das Gegenteil; Es nimmt jedes Zeichen in einer Zeichenfolge und konvertiert es in sein Kleinbuchstaben:

```
"This IS a 'string'.".lower()
# "this is a 'string'."
```

`str.capitalize()`

`str.capitalize` gibt eine `str.capitalize` Version der Zeichenfolge zurück, `str.capitalize` , das erste Zeichen wird in Großbuchstaben geschrieben, der Rest darunter:

```
"this Is A 'String'.".capitalize() # Capitalizes the first character and lowercases all others
# "This is a 'string'."
```

`str.title()`

`str.title` gibt die Titelversion des Strings zurück, d. h. jeder Buchstabe am Anfang eines Wortes wird in Großbuchstaben und alle anderen in Kleinbuchstaben geschrieben:

```
"this Is a 'String'".title()
# "This Is A 'String'"
```

`str.swapcase()`

`str.swapcase` gibt ein neues String-Objekt zurück, in dem alle Kleinbuchstaben in Großbuchstaben und alle Großbuchstaben in Kleinbuchstaben umgewandelt werden:

```
"this iS A STRiNG".swapcase() #Swaps case of each character
# "THIS Is a strIng"
```

Verwendung als `str` Klassenmethoden

Es sei darauf hingewiesen, dass diese Methoden entweder für String-Objekte (wie oben gezeigt) oder als Klassenmethode der Klasse `str` (mit einem expliziten Aufruf von `str.upper` usw.) `str.upper` können.

```
str.upper("This is a 'string'")
# "THIS IS A 'STRING'"
```

Dies ist besonders nützlich, wenn eine dieser Methoden auf viele Zeichenfolgen gleichzeitig angewendet wird, beispielsweise in einer *map* Funktion.

```
map(str.upper,["These","are","some","'strings'"])
# ['THESE', 'ARE', 'SOME', "'STRINGS'"]
```

Teilen Sie eine Zeichenfolge basierend auf einem Trennzeichen in eine Liste von Zeichenfolgen

str.split(sep=None, maxsplit=-1)

`str.split` einen String und gibt eine Liste der Teilstrings des ursprünglichen Strings zurück. Das Verhalten hängt davon ab, ob das `sep` Argument angegeben oder weggelassen wird.

Wenn " `sep` nicht angegeben ist oder " `None` , erfolgt die Aufteilung überall dort, wo Leerzeichen vorhanden sind. Führende und nachgestellte Leerzeichen werden jedoch ignoriert, und mehrere aufeinanderfolgende Leerzeichen werden wie ein einzelnes Leerzeichen behandelt:

```
>>> "This is a sentence.".split()
['This', 'is', 'a', 'sentence.']

>>> " This is    a sentence.  ".split()
['This', 'is', 'a', 'sentence.']

>>> "            ".split()
[]
```

Mit dem Parameter `sep` kann eine Trennzeichenfolge definiert werden. Die ursprüngliche Zeichenfolge wird dort aufgeteilt, wo die Trennzeichenfolge auftritt, und das Trennzeichen selbst wird verworfen. Mehrere aufeinanderfolgende Trennzeichen werden *nicht* wie ein einzelnes Vorkommen behandelt, sondern es werden leere Zeichenfolgen erstellt.

```
>>> "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.']
```

Standardmäßig wird bei *jedem* Vorkommen des Trennzeichens eine `maxsplit` Parameter `maxsplit` begrenzt jedoch die Anzahl der auftretenden Aufteilungen. Der Standardwert von `-1` bedeutet kein

Limit:

```
>>> "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") unterscheidet sich von `str.split` ("left split"), wenn `maxsplit` angegeben wird. Die Aufteilung beginnt am Ende der Zeichenfolge und nicht am Anfang:

```
>>> "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', '.']
```

Hinweis : Python gibt die maximale Anzahl der durchgeführten *Splits an* , während die meisten anderen Programmiersprachen die maximale Anzahl der erstellten *Teilzeichenfolgen* angeben. Dies kann zu Verwirrung beim Portieren oder Vergleichen von Code führen.

Ersetzen Sie alle Vorkommen einer Teilzeichenfolge durch eine andere Teilzeichenfolge

Der `str` Typ von Python verfügt auch über eine Methode, um Vorkommen einer Unterzeichenfolge durch eine andere Unterzeichenfolge in einer gegebenen Zeichenfolge zu ersetzen. Für anspruchsvollere Fälle kann `re.sub` .

`str.replace(old, new[, count])` :

`str.replace` zwei `old` und `new` Argumente, die die `old` Teilzeichenfolge enthalten, die durch die `new` Teilzeichenfolge ersetzt werden soll. Das optionale Argument `count` gibt die Anzahl der durchzuführenden Ersetzungen an:

Um beispielsweise `'foo'` durch `'spam'` in der folgenden Zeichenfolge zu ersetzen, können wir `str.replace` mit `old = 'foo'` und `new = 'spam'` aufrufen:

```
>>> "Make sure to foo your sentence.".replace('foo', 'spam')
"Make sure to spam your sentence."
```

Wenn die angegebene Zeichenfolge mehrere Beispiele enthält, die dem `old` Argument

entsprechen, werden **alle** Vorkommen durch den in `new` Wert ersetzt:

```
>>> "It can foo multiple examples of foo if you want.".replace('foo', 'spam')
"It can spam multiple examples of spam if you want."
```

es sei denn, wir liefern natürlich einen Wert für die `count` . In diesem Fall `count` gehen Vorkommen ersetzt werden:

```
>>> """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 und f-strings: Formatiert Werte in einen String

Python bietet String-Interpolations- und Formatierungsfunktionen über die in Version 2.6 eingeführte Funktion `str.format` und in Version 3.6 eingeführte F-Strings.

Angesichts der folgenden Variablen:

```
i = 10
f = 1.5
s = "foo"
l = ['a', 1, 2]
d = {'a': 1, 2: 'foo'}
```

Die folgenden Aussagen sind alle gleichwertig

```
"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}"
```

Als Referenz unterstützt Python auch C-Stil-Qualifizierer für die Formatierung von Zeichenfolgen. Die folgenden Beispiele entsprechen den oben genannten, die `str.format` Versionen werden jedoch aufgrund der Vorteile `str.format` Flexibilität, Konsistenz der Notation und Erweiterbarkeit bevorzugt:

```
"%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)
```

Die für die Interpolation in `str.format` verwendeten `str.format` können auch durchnummeriert werden, um die Duplizierung beim Formatieren von Strings zu reduzieren. Zum Beispiel sind die folgenden gleichwertig:

```
"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")
```

Während die offizielle Python-Dokumentation wie immer gründlich genug ist, bietet pyformat.info eine Reihe von Beispielen mit detaillierten Erklärungen.

Darüber hinaus können die Zeichen `{` und `}` mit doppelten Klammern geschützt werden:

```
"{'a': 5, 'b': 6}"
```

```
>>> "{{'{}': {}, '{}': {}}}".format("a", 5, "b", 6)

>>> f"{{'{'a'}': {5}, '{'b'}': {6}}"
```

Weitere Informationen finden Sie unter String-Formatierung . `str.format()` wurde in PEP 3101 und f-strings in PEP 498 vorgeschlagen .

Zählt, wie oft ein Teilstring in einem String erscheint

Es gibt eine Methode, um die Anzahl der Vorkommen einer Unterzeichenfolge in einer anderen Zeichenfolge, `str.count` , zu `str.count` .

```
str.count(sub[, start[, end]])
```

`str.count` gibt ein `int` , das die Anzahl der nicht überlappenden Vorkommen des Sub-String- `sub` in einem anderen String angibt. Die optionalen Argumente `start` und `end` geben den Anfang und das Ende der Suche an. Standardmäßig sind `start = 0` und `end = len(str)` dh der gesamte String wird durchsucht:

```
>>> s = "She sells seashells by the seashore."
>>> s.count("sh")
2
>>> s.count("se")
3
>>> s.count("sea")
2
>>> s.count("seashells")
1
```

Durch die Angabe eines anderen Werts für `start` , `end` wir eine lokalisierte Suche und Zählung

erhalten. Wenn `start` gleich `13` der Aufruf an:

```
>>> s.count("sea", start)
1
```

ist äquivalent zu:

```
>>> t = s[start:]
>>> t.count("sea")
1
```

Testen Sie die Start- und Endzeichen einer Zeichenfolge

Um den Anfang und das Ende eines bestimmten Strings in Python zu testen, können die Methoden `str.startswith()` und `str.endswith()` .

`str.startswith(prefix[, start[, end]])`

Wie der Name schon sagt, wird mit `str.startswith` getestet, ob eine angegebene Zeichenfolge mit den angegebenen Zeichen im `prefix` beginnt.

```
>>> s = "This is a test string"
>>> s.startswith("T")
True
>>> s.startswith("Thi")
True
>>> s.startswith("thi")
False
```

Die optionalen Argumente `start` und `end` geben die Start- und Endpunkte an, von denen aus der Test gestartet und beendet wird. Im folgenden Beispiel wird durch Angabe eines Startwerts von `2` unsere Zeichenfolge ab Position `2` und danach gesucht:

```
>>> s.startswith("is", 2)
True
```

Dies ergibt `True` da `s[2] == 'i'` und `s[3] == 's'` .

Sie können ein `tuple` auch verwenden, um zu überprüfen, ob es mit einer Reihe von Zeichenfolgen beginnt

```
>>> s.startswith(('This', 'That'))
True
>>> s.startswith(('ab', 'bc'))
False
```

`str.endswith(prefix[, start[, end]])`

`str.endswith` ähnelt `str.endswith` genau, mit dem einzigen Unterschied, dass nach Endzeichen und

nicht nach `str.startswith` `str.endswith` wird. Um beispielsweise zu testen, ob eine Zeichenfolge in einem Punkt endet, kann man schreiben:

```
>>> s = "this ends in a full stop."
>>> s.endswith('.')
True
>>> s.endswith('!')
False
```

wie bei `startswith` mehr als ein Zeichen als Endsequenz verwendet werden:

```
>>> s.endswith('stop.')
True
>>> s.endswith('Stop.')
False
```

Sie können ein `tuple` auch verwenden, um zu überprüfen, ob es mit einem String-Satz endet

```
>>> s.endswith(('.', 'something'))
True
>>> s.endswith(('ab', 'bc'))
False
```

Testen, woraus eine Zeichenfolge besteht

Der `str` Typ von Python bietet auch eine Reihe von Methoden, mit denen der Inhalt eines Strings ausgewertet werden kann. Dies sind `str.isalpha` , `str.isdigit` , `str.isalnum` , `str.isspace` . Die Kapitalisierung kann mit `str.isupper` , `str.islower` und `str.istitle` .

`str.isalpha`

`str.isalpha` keine Argumente und gibt '' `True` wenn alle Zeichen in einer angegebenen Zeichenfolge alphabetisch sind. Beispiel:

```
>>> "Hello World".isalpha()  # contains a space
False
>>> "Hello2World".isalpha()  # contains a number
False
>>> "HelloWorld!".isalpha()  # contains punctuation
False
>>> "HelloWorld".isalpha()
True
```

Als `"".isalpha()` wird der leere String bei Verwendung von `"".isalpha()` als `"".isalpha()` `False` `"".isalpha()` .

`str.isupper` , `str.islower` , `str.istitle`

Diese Methoden testen die Großschreibung in einer bestimmten Zeichenfolge.

`str.isupper` ist eine Methode, die `True` zurückgibt, wenn alle Zeichen in einer angegebenen Zeichenfolge Großbuchstaben sind, andernfalls `False` .

```
>>> "HeLLO WORLD".isupper()
False
>>> "HELLO WORLD".isupper()
True
>>> "".isupper()
False
```

Umgekehrt ist `str.islower` eine Methode, die `True` zurückgibt, wenn alle Zeichen in einer angegebenen Zeichenfolge Kleinbuchstaben und andernfalls `False` sind.

```
>>> "Hello world".islower()
False
>>> "hello world".islower()
True
>>> "".islower()
False
```

`str.istitle` gibt `True` wenn die angegebene Zeichenfolge mit einem Titel versehen ist. Das heißt, jedes Wort beginnt mit einem Großbuchstaben gefolgt von Kleinbuchstaben.

```
>>> "hello world".istitle()
False
>>> "Hello world".istitle()
False
>>> "Hello World".istitle()
True
>>> "".istitle()
False
```

`str.isdecimal` , `str.isdigit` , `str.isnumeric`

`str.isdecimal` zurück, ob es sich bei der Zeichenfolge um eine Folge von Dezimalstellen handelt, die zur Darstellung einer Dezimalzahl geeignet ist.

`str.isdigit` enthält Ziffern, die sich nicht in einer für die Darstellung einer Dezimalzahl geeigneten Form befinden, wie z. B. hochgestellte Ziffern.

`str.isnumeric` enthält beliebige Zahlenwerte, auch wenn es sich nicht um Ziffern handelt, z. B. Werte außerhalb des Bereichs 0-9.

```
            isdecimal    isdigit   isnumeric

12345         True         True        True
٢٣٤٥          True         True        True
①²³₅        False        True        True
⑩             False        False       True
Five          False        False       False
```

Bytestrings (`bytes` in Python 3, `str` in Python 2) unterstützen nur `isdigit` , wobei nur ASCII-

`isdigit` .

Wie bei `str.isalpha` wird der leere String zu `False` ausgewertet.

`str.isalnum`

Dies ist eine Kombination von `str.isalpha` und `str.isnumeric` , speziell es wertet `True` , wenn alle Zeichen in der angegebenen String - **alphanumerische** sind, das heißt, sie bestehen aus alphabetischen *oder* numerischen Zeichen:

```
>>> "Hello2World".isalnum()
True
>>> "HelloWorld".isalnum()
True
>>> "2016".isalnum()
True
>>> "Hello World".isalnum()  # contains whitespace
False
```

`str.isspace`

Wird als " `True` ausgewertet, wenn die Zeichenfolge nur Leerzeichen enthält.

```
>>> "\t\r\n".isspace()
True
>>> " ".isspace()
True
```

Manchmal sieht eine Zeichenfolge „leer“ aus, aber wir wissen nicht, ob sie nur Leerzeichen oder überhaupt keine Zeichen enthält

```
>>> "".isspace()
False
```

Um diesen Fall abzudecken, benötigen wir einen zusätzlichen Test

```
>>> my_str = ''
>>> my_str.isspace()
False
>>> my_str.isspace() or not my_str
True
```

Der kürzeste Weg, um zu testen, ob eine Zeichenfolge leer ist oder nur Leerzeichen enthält, ist `strip` (ohne Argumente werden alle führenden und nachgestellten Leerzeichen entfernt).

```
>>> not my_str.strip()
True
```

str.translate: Zeichen in einer Zeichenfolge übersetzen

Python unterstützt eine `translate` für den `str` Typ, mit der Sie die (für die Ersetzung verwendete) Übersetzungstabelle sowie alle Zeichen angeben können, die im Prozess gelöscht werden sollen.

```
str.translate(table[, deletechars])
```

Parameter	Beschreibung
`table`	Es ist eine Nachschlagetabelle, die die Zuordnung von einem Zeichen zu einem anderen definiert.
`deletechars`	Eine Liste der Zeichen, die aus der Zeichenfolge entfernt werden sollen.

Mit der Methode `maketrans` (`str.maketrans` in Python 3 und `string.maketrans` in Python 2) können Sie eine Übersetzungstabelle `string.maketrans` .

```
>>> translation_table = str.maketrans("aeiou", "12345")
>>> my_string = "This is a string!"
>>> translated = my_string.translate(translation_table)
'Th3s 3s 1 str3ng!'
```

Die `translate` gibt einen String zurück, der eine übersetzte Kopie des Originalstrings ist.

Sie können das `table` auf `None` wenn Sie nur Zeichen löschen müssen.

```
>>> 'this syntax is very useful'.translate(None, 'aeiou')
'ths syntx s vry sfl'
```

Entfernen Sie unerwünschte führende / nachgestellte Zeichen aus einer Zeichenfolge

Es stehen drei Methoden zur Verfügung, mit denen führende und nachgestellte Zeichen aus einem String entfernt werden können: `str.strip` , `str.rstrip` und `str.lstrip` . Alle drei Methoden haben dieselbe Signatur und alle drei geben ein neues String-Objekt zurück, wobei unerwünschte Zeichen entfernt werden.

```
str.strip([chars])
```

`str.strip` wirkt auf eine gegebene Zeichenkette und entfernt (Streifen) irgendwelche führenden oder nachfolgenden im Argument enthaltenen Zeichen `chars` ; Wenn `chars` nicht angegeben sind oder `None` , werden standardmäßig alle Leerzeichen entfernt. Zum Beispiel:

```
>>> "    a line with leading and trailing space     ".strip()
'a line with leading and trailing space'
```

Wenn `chars` , werden alle darin enthaltenen Zeichen aus der zurückgegebenen Zeichenfolge entfernt. Zum Beispiel:

```
>>> ">>> a Python prompt".strip('> ')  # strips '>' character and space character
'a Python prompt'
```

`str.rstrip([chars])` und `str.lstrip([chars])`

Diese Methoden haben mit `str.strip()` ähnliche Semantiken und Argumente. Ihr Unterschied liegt in der Richtung, aus der sie beginnen. `str.rstrip()` beginnt am Ende des Strings, während `str.lstrip()` vom Anfang des Strings getrennt wird.

Verwenden Sie beispielsweise `str.rstrip` :

```
>>> "     spacious string      ".rstrip()
'     spacious string'
```

Während mit `str.lstrip` :

```
>>> "     spacious string      ".rstrip()
'spacious string      '
```

String-Vergleiche ohne Berücksichtigung von Groß- und Kleinschreibung

Ein Vergleich der Zeichenfolge ohne Berücksichtigung der Groß- und Kleinschreibung scheint etwas Triviales zu sein, ist es aber nicht. Dieser Abschnitt berücksichtigt nur Unicode-Zeichenfolgen (die Standardeinstellung in Python 3). Beachten Sie, dass Python 2 im Vergleich zu Python 3 subtile Schwächen aufweisen kann - die spätere Unicode-Verarbeitung ist viel vollständiger.

Das erste, was zu beachten ist, dass Konvertierungen, bei denen die Groß- / Kleinschreibung entfernt wird, nicht trivial sind. Es gibt Text, für den `text.lower() != text.upper().lower()` , beispielsweise `"ß"` :

```
>>> "ß".lower()
'ß'

>>> "ß".upper().lower()
'ss'
```

Aber sagen wir mal, Sie wollten `"BUSSE"` und `"Buße"` . Heck, du willst wahrscheinlich auch `"BUSSE"` und `"BU□E"` gleich vergleichen - das ist die neuere Kapitalform. Die empfohlene Methode ist die Verwendung von `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.
"""
```

Verwenden Sie nicht einfach `lower` . Wenn die `casefold` nicht verfügbar ist, hilft das Ausführen von `.upper().lower()` (jedoch nur etwas).

Dann sollten Sie Akzente berücksichtigen. Wenn Ihr Font-Renderer gut ist, denken Sie wahrscheinlich `"ê" == "ê "` - aber es funktioniert nicht:

```
>>> "ê" == "ê "
False
```

Das ist, weil sie tatsächlich sind

```
>>> 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']
```

Der einfachste Weg, um damit umzugehen, ist `unicodedata.normalize` . Sie möchten möglicherweise die **NFKD-** Normalisierung verwenden, können jedoch die Dokumentation überprüfen. Dann tut man es

```
>>> unicodedata.normalize("NFKD", "ê") == unicodedata.normalize("NFKD", "ê ")
True
```

Zum Abschluss wird dies hier in Funktionen ausgedrückt:

```
import unicodedata

def normalize_caseless(text):
    return unicodedata.normalize("NFKD", text.casefold())

def caseless_equal(left, right):
    return normalize_caseless(left) == normalize_caseless(right)
```

Verknüpfen Sie eine Liste von Zeichenfolgen in einer Zeichenfolge

Eine Zeichenfolge kann als Trennzeichen verwendet werden, um eine Liste von Zeichenfolgen mithilfe der `join()` Methode zu einer einzelnen Zeichenfolge zusammenzufügen. Sie können beispielsweise eine Zeichenfolge erstellen, bei der jedes Element in einer Liste durch ein Leerzeichen getrennt ist.

```
>>> " ".join(["once","upon","a","time"])
"once upon a time"
```

Im folgenden Beispiel werden die Zeichenfolgenelemente durch drei Bindestriche getrennt.

```
>>> "---".join(["once", "upon", "a", "time"])
"once---upon---a---time"
```

Nützliche Konstanten des String-Moduls

Das `string` Modul von Python stellt Konstanten für string-bezogene Operationen bereit. Um sie zu verwenden, importieren Sie das `string` Modul:

```
>>> import string
```

`string.ascii_letters` :

Verkettung von `ascii_lowercase` und `ascii_uppercase` :

```
>>> string.ascii_letters
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

`string.ascii_lowercase` :

Enthält alle ASCII-Kleinbuchstaben:

```
>>> string.ascii_lowercase
'abcdefghijklmnopqrstuvwxyz'
```

`string.ascii_uppercase` :

Enthält alle ASCII-Großbuchstaben:

```
>>> string.ascii_uppercase
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

`string.digits` :

Enthält alle Dezimalziffern:

```
>>> string.digits
'0123456789'
```

`string.hexdigits` :

Enthält alle hexadezimalen Zeichen:

```
>>> string.hexdigits
'0123456789abcdefABCDEF'
```

`string.octaldigits` :

Enthält alle Oktalzeichen:

```
>>> string.octaldigits
'01234567'
```

`string.punctuation` :

Enthält alle Zeichen, die im `C` Gebietsschema als Interpunktion gelten:

```
>>> string.punctuation
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

`string.whitespace` :

Enthält alle ASCII-Zeichen, die als Leerzeichen gelten:

```
>>> string.whitespace
' \t\n\r\x0b\x0c'
```

Im Skriptmodus werden mit `print(string.whitespace)` die tatsächlichen Zeichen gedruckt. Verwenden Sie `str` , um den oben angegebenen String `print(string.whitespace)` .

`string.printable` :

Enthält alle Zeichen, die als druckbar gelten. eine Kombination aus `string.digits` , `string.ascii_letters` , `string.punctuation` und `string.whitespace` .

```
>>> string.printable
'0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&\'()*+,-
./:;<=>?@[\\]^_`{|}~ \t\n\r\x0b\x0c'
```

Einen String umkehren

Eine Zeichenfolge kann mit der integrierten Funktion `reversed()` umgekehrt werden. Diese Funktion nimmt eine Zeichenfolge und gibt einen Iterator in umgekehrter Reihenfolge zurück.

```
>>> reversed('hello')
<reversed object at 0x0000000000000000>
>>> [char for char in reversed('hello')]
['o', 'l', 'l', 'e', 'h']
```

`reversed()` kann in einen Aufruf von `''.join()` , um aus dem Iterator eine Zeichenfolge zu `''.join()` .

```
>>> ''.join(reversed('hello'))
'olleh'
```

Während `reversed()` für ungeübte Python-Benutzer möglicherweise lesbarer ist, ist die Verwendung von erweitertem Slicing mit einem Schritt von `-1` schneller und übersichtlicher. Versuchen Sie es hier als Funktion zu implementieren:

```
>>> def reversed_string(main_string):
...     return main_string[::-1]
...
>>> reversed_string('hello')
'olleh'
```

Zeichenfolgen rechtfertigen

Python bietet Funktionen zum Ausrichten von Zeichenfolgen, wodurch das Auffüllen mit Text die Ausrichtung verschiedener Zeichenfolgen erheblich vereinfacht.

Unten ist ein Beispiel für `str.ljust` und `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` und `rjust` sind sehr ähnlich. Beide haben einen `width` Parameter und einen optionalen `fillchar` Parameter. Jeder von diesen Funktionen erstellte String ist mindestens so lang wie der `width` Parameter, der an die Funktion übergeben wurde. Wenn die Zeichenfolge länger als die `width` ist, wird sie nicht abgeschnitten. Das Argument `fillchar` , das standardmäßig auf das Leerzeichen `' '` muss ein einzelnes Zeichen und keine Zeichenfolge mit `fillchar` Zeichen sein.

Die `ljust` Funktion `ljust` das Ende des aufgerufenen Strings mit dem `fillchar` bis es `width` Zeichen lang ist. Die `rjust` Funktion `rjust` den Anfang der Saite auf ähnliche Weise auf. Daher beziehen sich `l` und `r` in den Namen dieser Funktionen auf die Seite, auf der die Originalzeichenfolge und *nicht die* `fillchar` in der Ausgabezeichenfolge positioniert ist.

Konvertierung zwischen Str. Oder Byte-Daten und Unicode-Zeichen

Der Inhalt von Dateien und Netzwerknachrichten kann verschlüsselte Zeichen darstellen. Sie müssen häufig in Unicode konvertiert werden, um eine korrekte Anzeige zu ermöglichen.

In Python 2 müssen Sie möglicherweise str-Daten in Unicode-Zeichen konvertieren. Der Standardwert (`''` , `""` usw.) ist eine ASCII-Zeichenfolge, wobei alle Werte außerhalb des ASCII-Bereichs als Escape-Werte angezeigt werden. Unicode-Zeichenfolgen sind `u''` (oder `u""` usw.).

Python 2.x 2.3

```
# You get "© abc" encoded in UTF-8 from a file, network, or other data source

s = '\xc2\xa9 abc'  # s is a byte array, not a string of characters
                    # Doesn't know the original was UTF-8
                    # Default form of string literals in Python 2
s[0]                # '\xc2' - meaningless byte (without context such as an encoding)
type(s)             # str - even though it's not a useful one w/o having a known encoding

u = s.decode('utf-8')  # u'\xa9 abc'
                       # Now we have a Unicode string, which can be read as UTF-8 and printed
properly
                       # In Python 2, Unicode string literals need a leading u
                       # str.decode converts a string which may contain escaped bytes to a
Unicode string
u[0]                # u'\xa9' - Unicode Character 'COPYRIGHT SIGN' (U+00A9) '©'
type(u)             # unicode

u.encode('utf-8')   # '\xc2\xa9 abc'
                    # unicode.encode produces a string with escaped bytes for non-ASCII
characters
```

In Python 3 müssen Sie möglicherweise Byte-Arrays (als Byte-Literal bezeichnet) in Zeichenfolgen aus Unicode-Zeichen konvertieren. Der Standardwert ist jetzt eine Unicode-Zeichenfolge, und Bytestring-Literale müssen jetzt als `b''` , `b""` usw. eingegeben werden. Ein Byte-Literal gibt `True` an `isinstance(some_val, byte)` , wobei `some_val` , dass `some_val` eine möglicherweise codierte Zeichenfolge ist als Bytes.

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.
```

String enthält

Python macht es extrem intuitiv zu überprüfen, ob eine Zeichenfolge eine gegebene Teilzeichenfolge enthält. Verwenden Sie einfach den `in` Operator:

```
>>> "foo" in "foo.baz.bar"
True
```

Hinweis: Das Testen einer leeren Zeichenfolge führt immer zu " `True` :

```
>>> "" in "test"
True
```

String-Methoden online lesen: https://riptutorial.com/de/python/topic/278/string-methoden

Kapitel 163: Subprozess-Bibliothek

Syntax

- subprocess.call (args, *, stdin = keine, stdout = keine, stderr = keine, shell = False, timeout = keine)
- subprozess.Popen (args, bufsize = -1, ausführbare Datei = keine, stdin = keine, stdout = keine, stderr = keine, preexec_fn = keine, close_fds = true, shell = false, cwd = none, keine; , startupinfo = Keine, creationflags = 0, restore_signals = True, start_new_session = False, pass_fds = ())

Parameter

Parameter	Einzelheiten
`args`	Eine einzelne ausführbare Datei oder eine Folge von ausführbaren Dateien und Argumenten - `'ls'` , `['ls', '-la']`
`shell`	Unter einer Shell laufen? Die Standard-Shell für `/bin/sh` unter POSIX.
`cwd`	Arbeitsverzeichnis des untergeordneten Prozesses.

Examples

Externe Befehle aufrufen

Der einfachste Anwendungsfall ist die Verwendung der Funktion `subprocess.call` . Es akzeptiert eine Liste als erstes Argument. Das erste Element in der Liste sollte die externe Anwendung sein, die Sie anrufen möchten. Die anderen Elemente in der Liste sind Argumente, die an diese Anwendung übergeben werden.

```
subprocess.call([r'C:\path\to\app.exe', 'arg1', '--flag', 'arg'])
```

Setzen Sie für `shell=True` und geben Sie den Befehl als String anstelle einer Liste an.

```
subprocess.call('echo "Hello, world"', shell=True)
```

Beachten Sie, dass die beiden obigen Befehle nur den `exit status` des Unterprozesses zurückgeben. Beachten Sie außerdem bei der Verwendung von `shell=True` da dies Sicherheitsprobleme verursacht (siehe hier).

Wenn Sie die Standardausgabe des Subprozesses abrufen möchten, ersetzen Sie `subprocess.call` durch `subprocess.check_output` . Weitere Informationen finden Sie hier .

Mehr Flexibilität mit Popen

Mit `subprocess.Popen` können Sie gestartete Prozesse genauer steuern als `subprocess.call` .

Subprozess starten

```
process = subprocess.Popen([r'C:\path\to\app.exe', 'arg1', '--flag', 'arg'])
```

Die Signatur für `Popen` ist sehr ähnlich wie die `call` `Popen` kehrt jedoch sofort zurück, anstatt auf den Abschluss des Unterprozesses zu warten, wie dies bei einem `call` Fall ist.

Warten auf den Abschluss eines Unterprozesses

```
process = subprocess.Popen([r'C:\path\to\app.exe', 'arg1', '--flag', 'arg'])
process.wait()
```

Ausgabe aus einem Unterprozess lesen

```
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
```

Interaktiver Zugriff auf laufende Unterprozesse

Sie können `stdin` und `stdout` auch dann lesen und schreiben, wenn der Unterprozess noch nicht abgeschlossen ist. Dies kann nützlich sein, wenn Sie die Funktionalität in einem anderen Programm automatisieren.

In einen Teilprozess schreiben

```
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.
```

Wenn Sie jedoch nur einen Satz von Eingaben und Ausgaben benötigen, anstatt dynamischer Interaktion, sollten Sie `communicate()` anstatt direkt auf `stdin` und `stdout` zuzugreifen.

Einen Stream aus einem Unterprozess lesen

Wenn Sie die Ausgabe eines Unterprozesses zeilenweise anzeigen möchten, können Sie den folgenden Ausschnitt verwenden:

```
process = subprocess.Popen(<your_command>, stdout=subprocess.PIPE)
while process.poll() is None:
    output_line = process.stdout.readline()
```

Falls die Unterbefehlsausgabe kein EOL-Zeichen hat, funktioniert das obige Snippet nicht. Sie können die Ausgabe zeichenweise wie folgt lesen:

```
process = subprocess.Popen(<your_command>, stdout=subprocess.PIPE)
while process.poll() is None:
    output_line = process.stdout.read(1)
```

Die als Argument für die `read` angegebene `1` teilt dem Lesevorgang mit, 1 Zeichen zu lesen. Sie können festlegen, dass beliebig viele Zeichen mit einer anderen Nummer gelesen werden. Eine negative Zahl oder 0 gibt an, dass `read` bis eine EOF gefunden wird (siehe hier).

In beiden der obigen Ausschnitte ist `process.poll() None` bis der Unterprozess abgeschlossen ist. Dies wird verwendet, um die Schleife zu beenden, wenn keine weitere Ausgabe mehr zum Lesen vorhanden ist.

Dasselbe Verfahren könnte auf den `stderr` des Unterprozesses angewendet werden.

So erstellen Sie das Befehlslistenargument

Die Unterprozessmethode, die das Ausführen von Befehlen zulässt, erfordert den Befehl in Form einer Liste (zumindest mit `shell_mode=True`).

Die Regeln zum Erstellen der Liste sind nicht immer einfach zu befolgen, insbesondere bei komplexen Befehlen. Glücklicherweise gibt es ein sehr hilfreiches Werkzeug, das dies ermöglicht: `shlex` . Die einfachste Methode zum Erstellen der Liste, die als Befehl verwendet werden soll, ist die folgende:

```
import shlex
cmd_to_subprocess = shlex.split(command_used_in_the_shell)
```

Ein einfaches Beispiel:

```
import shlex
shlex.split('ls --color -l -t -r')

out: ['ls', '--color', '-l', '-t', '-r']
```

Subprozess-Bibliothek online lesen: https://riptutorial.com/de/python/topic/1393/subprozess-bibliothek

Kapitel 164: Suchen

Bemerkungen

Alle Suchalgorithmen auf iterierbaren Elementen, die `n` Elemente enthalten, weisen eine `O(n)` - Komplexität auf. Nur spezialisierte Algorithmen wie `bisect.bisect_left()` können mit `O(log(n))` `bisect.bisect_left()` schneller sein.

Examples

Den Index für Strings abrufen: str.index (), str.rindex () und str.find (), str.rfind ()

`String` auch eine `index` , aber auch erweiterte Optionen und die zusätzliche `str.find` . Für beide gibt es eine komplementäre *umgekehrte* Methode.

```
astring = 'Hello on StackOverflow'
astring.index('o')  # 4
astring.rindex('o') # 20

astring.find('o')   # 4
astring.rfind('o')  # 20
```

Der Unterschied zwischen `index` / `rindex` und `find` / `rfind` besteht darin, was passiert, wenn die Teilzeichenfolge nicht in der Zeichenfolge gefunden wird:

```
astring.index('q') # ValueError: substring not found
astring.find('q')  # -1
```

Alle diese Methoden ermöglichen einen Start- und Endindex:

```
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: Unterzeichenfolge nicht gefunden

```
astring.rindex('o', 20) # 20
astring.rindex('o', 19) # 20 - still from left to right

astring.rindex('o', 4, 7) # 6
```

Nach einem Element suchen

Alle integrierten Sammlungen in Python implementieren eine Art und Weise Element Mitgliedschaft zu überprüfen Verwendung `in` .

Liste

```
alist = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
5 in alist   # True
10 in alist  # False
```

Tupel

```
atuple = ('0', '1', '2', '3', '4')
4 in atuple    # False
'4' in atuple  # True
```

String

```
astring = 'i am a string'
'a' in astring   # True
'am' in astring  # True
'I' in astring   # False
```

einstellen

```
aset = {(10, 10), (20, 20), (30, 30)}
(10, 10) in aset  # True
10 in aset        # False
```

Dikt

`dict` ist etwas Besonderes: die normalen `in` nur die *Schlüssel* überprüft. Wenn Sie nach *Werten* suchen möchten, müssen Sie diese angeben. Dasselbe, wenn Sie nach *Schlüsselwertpaaren* suchen möchten.

```
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
```

Abrufen der Indexliste und der Tupel: list.index (), tuple.index ()

`list` und `tuple` haben eine `index` Methode, um die Position des Elements zu ermitteln:

```
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 ist nicht in der Liste

Gibt jedoch nur die Position des ersten gefundenen Elements zurück:

```
atuple = (10, 16, 26, 5, 2, 19, 105, 26)
atuple.index(26)   # 2
atuple[2]          # 26
atuple[7]          # 26 - is also 26!
```

Suchschlüssel nach einem Wert in dict

`dict` hat keine integrierte Methode zum Suchen eines Werts oder Schlüssels, da *Wörterbücher* ungeordnet sind. Sie können eine Funktion erstellen, die den Schlüssel (oder die Schlüssel) für einen angegebenen Wert abruft:

```
def getKeysForValue(dictionary, value):
    foundkeys = []
    for keys in dictionary:
        if dictionary[key] == value:
            foundkeys.append(key)
    return foundkeys
```

Dies könnte auch als äquivalentes Listenverständnis geschrieben werden:

```
def getKeysForValueComp(dictionary, value):
    return [key for key in dictionary if dictionary[key] == value]
```

Wenn Sie nur einen gefundenen Schlüssel interessieren:

```
def getOneKeyForValue(dictionary, value):
    return next(key for key in dictionary if dictionary[key] == value)
```

Die ersten beiden Funktionen geben eine `list` aller `keys` mit dem angegebenen Wert zurück:

```
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) # []
```

Der andere gibt nur einen Schlüssel zurück:

```
getOneKeyForValue(adict, 10)   # 'c'  - depending on the circumstances this could also be 'a'
getOneKeyForValue(adict, 20)   # 'b'
```

und wirft eine `StopIteration` - `Exception` , wenn der Wert nicht in dem ist `dict` :

```
getOneKeyForValue(adict, 25)
```

StopIteration

Den Index für sortierte Sequenzen abrufen: bisect.bisect_left ()

Sortierte Sequenzen ermöglichen die Verwendung schneller Suchalgorithmen: `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

Bei sehr großen **sortierten Sequenzen kann** der Geschwindigkeitszuwachs sehr hoch sein. Für die erste Suche ungefähr 500 mal so schnell:

```
%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
```

Es ist zwar etwas langsamer, wenn das Element eines der ersten ist:

```
%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
```

Verschachtelte Sequenzen durchsuchen

Das Suchen in verschachtelten Sequenzen wie eine `list` von `tuple` erfordert einen Ansatz wie das Durchsuchen der Schlüssel nach Werten in `dict` , erfordert jedoch angepasste Funktionen.

Der Index der äußersten Sequenz, wenn der Wert in der Sequenz gefunden wurde:

```
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
```

oder der Index der äußeren und inneren Sequenz:

```
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
```

Im Allgemeinen (*nicht immer*) ist die Verwendung von `next` und eines **Generatorausdrucks** mit Bedingungen zum Auffinden des ersten Werts des gesuchten Werts der effizienteste Ansatz.

Suchen in benutzerdefinierten Klassen: __contains__ und __iter__

Um die Verwendung von `in` für benutzerdefinierte Klassen zuzulassen, muss die Klasse entweder die magische Methode `__contains__` oder, `__iter__` eine `__iter__` `__contains__` Methode, `__iter__` .

Angenommen, Sie haben eine Klasse, die eine `list` mit `list` s enthält:

```
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))
```

Mitgliedschaft Prüfung zu verwenden ist möglich mit `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__.
```

auch nach dem Löschen der `__contains__` Methode:

```
del ListList.__contains__
5 in a     # True
# Prints: Using __iter__.
```

Hinweis: Die Schleife `in` (wie in `for i in a`) verwendet immer `__iter__` auch wenn die Klasse eine implementiert `__contains__` Methode.

Suchen online lesen: https://riptutorial.com/de/python/topic/350/suchen

Kapitel 165: sys

Einführung

Das **sys-** Modul bietet Zugriff auf Funktionen und Werte, die die Laufzeitumgebung des Programms betreffen, wie z. B. die Befehlszeilenparameter in `sys.argv` oder die Funktion `sys.exit()` , um den aktuellen Prozess an einem beliebigen Punkt im Programmablauf zu beenden.

Obwohl es sauber in ein Modul unterteilt ist, ist es tatsächlich eingebaut und steht unter normalen Umständen immer zur Verfügung.

Syntax

- Importieren Sie das sys-Modul und machen Sie es im aktuellen Namespace verfügbar:

```
import sys
```

- Importieren Sie eine bestimmte Funktion aus dem sys-Modul direkt in den aktuellen Namespace:

```
from sys import exit
```

Bemerkungen

Einzelheiten zu allen Mitgliedern des **sys-** Moduls finden Sie in der offiziellen Dokumentation .

Examples

Kommandozeilenargumente

```
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.
```

Beachten Sie, dass Sie in größeren und ausgefeilteren Programmen Module verwenden, z. B. klicken Sie , um Befehlszeilenargumente zu behandeln, anstatt es selbst zu tun.

Skriptname

```
# 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
```

Standardfehlerstrom

```
# 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)
```

Den Prozess vorzeitig beenden und einen Beendigungscode zurückgeben

```
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()
```

sys online lesen: https://riptutorial.com/de/python/topic/9847/sys

Kapitel 166: Teilfunktionen

Einführung

Wie Sie wahrscheinlich wissen, wenn Sie aus der OOP-Schule kamen, ist die Spezialisierung einer abstrakten Klasse eine Übung, die Sie beim Schreiben Ihres Codes berücksichtigen sollten.

Was wäre, wenn Sie eine abstrakte Funktion definieren und spezialisieren könnten, um verschiedene Versionen davon zu erstellen? Dient als eine Art *Funktionsvererbung,* bei der Sie bestimmte Parameter binden, um sie für ein bestimmtes Szenario zuverlässig zu machen.

Syntax

- partiell (Funktion, ** params_you_want_fix)

Parameter

Param	Einzelheiten
x	die Zahl, die erhöht werden soll
y	der Exponent
erziehen	die Funktion, spezialisiert zu sein

Bemerkungen

Wie in Python doc angegeben, die *functools.partial* :

> Gibt ein neues Teilobjekt zurück, das sich beim Aufruf wie func verhält, das mit den Schlüsselwörtern positional arguments args und keyword arguments aufgerufen wird. Wenn mehr Argumente an den Aufruf übergeben werden, werden sie an args angehängt. Wenn zusätzliche Schlüsselwortargumente angegeben werden, erweitern und überschreiben sie Schlüsselwörter.

Überprüfen Sie diesen Link , um zu sehen, wie *partiell* implementiert werden kann.

Examples

Erhöhen Sie die Kraft

Nehmen wir an, wir wollen *x* auf eine Zahl *y* erhöhen.

Sie würden dies schreiben als:

```
def raise_power(x, y):
    return x**y
```

Was ist, wenn Ihr *y*- Wert eine begrenzte Menge von Werten annehmen kann?

Nehmen wir an, *y* kann eine von *[3,4,5] sein,* und nehmen wir an, Sie möchten dem Endbenutzer nicht die Möglichkeit bieten, eine solche Funktion zu verwenden, da sie sehr rechenintensiv ist. In der Tat würden Sie prüfen, ob *y* einen gültigen Wert annimmt, und Ihre Funktion neu schreiben als:

```
def raise(x, y):
    if y in (3,4,5):
        return x**y
    raise NumberNotInRangeException("You should provide a valid exponent")
```

Unordentlich? Verwenden wir die abstrakte Form und spezialisieren Sie sie auf alle drei Fälle: Lassen Sie uns sie **teilweise** implementieren.

```
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)
```

was geschieht hier? Wir haben die *y*- Parameter korrigiert und drei verschiedene Funktionen definiert.

Sie brauchen die oben definierte abstrakte Funktion nicht zu verwenden (Sie könnten sie als *privat definieren*), aber Sie können **teilweise angewendete** Funktionen verwenden, um das Erhöhen einer Zahl auf einen festen Wert zu behandeln.

Teilfunktionen online lesen: https://riptutorial.com/de/python/topic/9383/teilfunktionen

Kapitel 167: tempfile NamedTemporaryFile

Parameter

param	Beschreibung
Modus	Modus zum Öffnen der Datei, Standard = w + b
löschen	Um die Datei beim Schließen zu löschen, ist default = True
Suffix	Dateinamensuffix, default = ''
Präfix	Dateinamenpräfix, default = 'tmp'
dir	dirname, um tempfile zu platzieren, default = None
Buffsize	default = -1, (Standardeinstellung des Betriebssystems)

Examples

Erstellen (und Schreiben) einer bekannten, permanenten temporären Datei

Sie können temporäre Dateien erstellen , die einen sichtbaren Namen auf dem Dateisystem hat , das über die zugegriffen werden kann `name` - Eigenschaft. Die Datei kann auf Unix-Systemen so konfiguriert werden, dass sie beim Schließen gelöscht wird (durch `delete` param festgelegt, der Standardwert ist True) oder kann später erneut geöffnet werden.

Im Folgenden wird eine benannte temporäre Datei erstellt und geöffnet und "Hello World!" zu dieser Datei. Der Dateipfad der temporären Datei kann über zugegriffen werden `name` , in diesem Beispiel wird in der Variable gespeichert `path` und für den Benutzer bedruckt. Die Datei wird dann nach dem Schließen der Datei erneut geöffnet und der Inhalt der Tempfile wird gelesen und für den Benutzer gedruckt.

```
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()
```

Ausgabe:

```
/tmp/tmp6pireJ
Hello World!
```

tempfile NamedTemporaryFile online lesen: https://riptutorial.com/de/python/topic/3666/tempfile-namedtemporaryfile

Kapitel 168: tkinter

Einführung

In Tkinter veröffentlicht ist die populärste GUI-Bibliothek (Graphical User Interface) von Python. In diesem Thema wird die ordnungsgemäße Verwendung dieser Bibliothek und ihrer Funktionen erläutert.

Bemerkungen

Die Großschreibung des Tkinter-Moduls unterscheidet sich zwischen Python 2 und 3. Für Python 2 verwenden Sie Folgendes:

```
from Tkinter import *  # Capitalized
```

Für Python 3 verwenden Sie Folgendes:

```
from tkinter import *  # Lowercase
```

Für Code, der mit Python 2 und 3 funktioniert, können Sie dies tun

```
try:
    from Tkinter import *
except ImportError:
    from tkinter import *
```

oder

```
from sys import version_info
if version_info.major == 2:
    from Tkinter import *
elif version_info.major == 3:
    from tkinter import *
```

> Weitere Informationen finden Sie in der tkinter-Dokumentation

Examples

Eine minimale Tkinter-Anwendung

`tkinter` ist ein GUI-Toolkit, das einen Wrapper um die Tk / Tcl-GUI-Bibliothek bereitstellt und in Python enthalten ist. Mit dem folgenden Code wird ein neues Fenster mit `tkinter` und Text in den Fensterkörper `tkinter` .

> Anmerkung: In Python 2 kann die Großschreibung etwas abweichen, siehe Abschnitt "Anmerkungen".

```
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()
```

Geometrie-Manager

Tkinter verfügt über drei Mechanismen für das Geometriemanagement: `place` , `pack` und `grid` .

Der `place` Manager verwendet absolute Pixelkoordinaten.

Der `pack` platziert Widgets auf einer von vier Seiten. Neue Widgets werden neben vorhandenen Widgets platziert.

Der `grid` Manager platziert Widgets in einem Raster, das einer Tabelle mit dynamischer Größenänderung ähnelt.

Platz

Die häufigsten Schlüsselwortargumente für `widget.place` lauten wie folgt:

- `x` , die absolute x-Koordinate des Widgets
- `y` , die absolute y-Koordinate des Widgets
- `height` , die absolute Höhe des Widgets
- `width` , die absolute Breite des Widgets

Ein Codebeispiel mit `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()
```

Pack

`widget.pack` kann die folgenden Schlüsselwortargumente `widget.pack` :

- `expand` , ob der von Eltern hinterlassene Platz gefüllt werden soll oder nicht
- `fill` , ob erweitert werden soll, um den gesamten Bereich zu füllen (NONE (Standard), X, Y oder BEIDE)
- `side` , die Seite, gegen die gepackt werden soll (TOP (Standard), UNTEN, LINKS oder RECHTS)

Gitter

Die am häufigsten verwendeten Schlüsselwortargumente von `widget.grid` lauten wie folgt:

- `row` , die Zeile des Widgets (standardmäßig kleinste unbesetzte)
- `rowspan` , die Anzahl der Spalten eines Widgets (Standardeinstellung 1)
- `column` , die Spalte des Widget (Standardwert 0)
- `columnspan` , die Anzahl der Spalten eines Widgets (Standard 1)
- `sticky` , wo das Widget platziert werden soll, wenn die Gitterzelle größer ist (Kombination aus N, NE, E, SE, S, SW, W, NW)

Die Zeilen und Spalten sind nullindiziert. Zeilen steigen nach unten und Spalten nach rechts.

Ein Codebeispiel mit `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()
```

Mischen Sie niemals `pack` und `grid` innerhalb eines Rahmens! Dies führt zu einem Deadlock der Anwendung!

tkinter online lesen: https://riptutorial.com/de/python/topic/7574/tkinter

Kapitel 169: Tupel

Einführung

Ein Tupel ist eine unveränderliche Liste von Werten. Tupel sind eine der einfachsten und häufigsten Sammlungstypen von Python und können mit dem Kommaoperator erstellt werden (`value = 1, 2, 3`).

Syntax

- (1, a, "hallo") # a muss eine Variable sein
- () # ein leeres Tupel
- (1,) # ein Tupel mit 1 Elementen. `(1)` ist kein Tupel.
- 1, 2, 3 # das 3-Element-Tupel (1, 2, 3)

Bemerkungen

Klammern werden nur für leere Tupel oder bei Verwendung in einem Funktionsaufruf benötigt.

Ein Tupel ist eine Folge von Werten. Die Werte können von jedem Typ sein und werden durch ganze Zahlen indiziert. In dieser Hinsicht sind Tupel sehr ähnlich wie Listen. Der wichtige Unterschied ist, dass Tupel unveränderlich und hashierbar sind, sodass sie in Sets und Karten verwendet werden können

Examples

Indexierung von Tupeln

```
x = (1, 2, 3)
x[0]  # 1
x[1]  # 2
x[2]  # 3
x[3]  # IndexError: tuple index out of range
```

Die Indizierung mit negativen Zahlen beginnt mit dem letzten Element als -1:

```
x[-1] # 3
x[-2] # 2
x[-3] # 1
x[-4] # IndexError: tuple index out of range
```

Indizierung einer Reihe von Elementen

```
print(x[:-1])   # (1, 2)
print(x[-1:])   # (3,)
print(x[1:3])   # (2, 3)
```

Tupel sind unveränderlich

Einer der Hauptunterschiede zwischen `list` und `tuple` in Python ist, dass Tupel unveränderlich sind, dh, dass Elemente nicht hinzugefügt oder geändert werden können, wenn das Tupel initialisiert ist. Zum Beispiel:

```
>>> 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
```

In ähnlicher Weise haben Tupel keine `.append` und `.extend` Methoden wie `list` . Die Verwendung von `+=` ist möglich, ändert jedoch die Bindung der Variablen und nicht das Tupel selbst:

```
>>> t = (1, 2)
>>> q = t
>>> t += (3, 4)
>>> t
(1, 2, 3, 4)
>>> q
(1, 2)
```

Seien Sie vorsichtig, wenn Sie veränderliche Objekte (z. B. `lists`) in Tupeln platzieren. Dies kann zu sehr verwirrenden Ergebnissen führen, wenn sie geändert werden. Zum Beispiel:

```
>>> t = (1, 2, 3, [1, 2, 3])
(1, 2, 3, [1, 2, 3])
>>> t[3] += [4, 5]
```

Wird ***sowohl*** ein Fehler ausgegeben als auch der Inhalt der Liste innerhalb des Tupels geändert:

```
TypeError: 'tuple' object does not support item assignment
>>> t
(1, 2, 3, [1, 2, 3, 4, 5])
```

Sie können den Operator `+=` , um an ein Tupel "anzuhängen". Dies funktioniert, indem Sie ein neues Tupel mit dem neuen Element erstellen, das Sie "angehängt" haben, und es der aktuellen Variablen zuweisen. Das alte Tupel wird nicht verändert, sondern ersetzt!

Dies vermeidet das Konvertieren in eine und aus einer Liste. Dies ist jedoch langsam und eine schlechte Praxis, insbesondere wenn Sie mehrmals anhängen.

Tuple sind elementweise Hashfähig und gleichwertig

```
hash( (1, 2) )  # ok
```

```
hash( ([], {"hello"})  # not ok, since lists and sets are not hashabe
```

Daher kann ein Tupel nur dann in ein `set` oder als Schlüssel in einem `dict` , wenn jedes seiner Elemente dies kann.

```
{ (1, 2) } #  ok
{ ([], {"hello"}) ) # not ok
```

Tupel

Syntaktisch ist ein Tupel eine durch Kommas getrennte Liste von Werten:

```
t = 'a', 'b', 'c', 'd', 'e'
```

Obwohl dies nicht notwendig ist, werden Tupel häufig in Klammern gesetzt:

```
t = ('a', 'b', 'c', 'd', 'e')
```

Erstellen Sie ein leeres Tupel mit Klammern:

```
t0 = ()
type(t0)            # <type 'tuple'>
```

Um ein Tupel mit einem einzelnen Element zu erstellen, müssen Sie ein abschließendes Komma angeben:

```
t1 = 'a',
type(t1)             # <type 'tuple'>
```

Beachten Sie, dass ein einzelner Wert in Klammern kein Tupel ist:

```
t2 = ('a')
type(t2)             # <type 'str'>
```

Um ein Singleton-Tupel zu erstellen, muss ein nachfolgendes Komma verwendet werden.

```
t2 = ('a',)
type(t2)             # <type 'tuple'>
```

Beachten Sie, dass es für Singleton-Tupel empfohlen wird, Klammern zu verwenden (siehe PEP8 in den nachfolgenden Kommas). Kein Leerzeichen nach dem nachgestellten Komma (siehe PEP8 zu Whitespaces)

```
t2 = ('a',)          # PEP8-compliant
t2 = 'a',            # this notation is not recommended by PEP8
t2 = ('a', )         # this notation is not recommended by PEP8
```

Eine andere Möglichkeit , ein Tupel zu erstellen , ist die integrierte Funktion `tuple` .

```
t = tuple('lupins')
print(t)              # ('l', 'u', 'p', 'i', 'n', 's')
t = tuple(range(3))
print(t)              # (0, 1, 2)
```

Diese Beispiele basieren auf Material aus dem Buch **Think Python** von Allen B. Downey .

Verpacken und Auspacken von Tupeln

Tupel in Python sind durch Kommas getrennte Werte. Das Einschließen von Klammern zur Eingabe von Tupeln ist optional, also die beiden Zuweisungen

```
a = 1, 2, 3   # a is the tuple (1, 2, 3)
```

und

```
a = (1, 2, 3) # a is the tuple (1, 2, 3)
```

sind gleichwertig. Die Zuordnung `a = 1, 2, 3` wird auch als *Packung bezeichnet,* da Werte in einem Tupel zusammengefasst werden.

Beachten Sie, dass ein einwertiges Tupel auch ein Tupel ist. Um Python mitzuteilen, dass eine Variable ein Tupel und kein einzelner Wert ist, können Sie ein nachfolgendes Komma verwenden

```
a = 1  # a is the value 1
a = 1, # a is the tuple (1,)
```

Ein Komma ist auch erforderlich, wenn Sie Klammern verwenden

```
a = (1,) # a is the tuple (1,)
a = (1)  # a is the value 1 and not a tuple
```

Um Werte aus einem Tupel zu entpacken und mehrere Zuweisungen auszuführen, verwenden Sie diese Option

```
# unpacking AKA multiple assignment
x, y, z = (1, 2, 3)
# x == 1
# y == 2
# z == 3
```

Das Symbol _ kann als Name einer verfügbaren Variablen verwendet werden, wenn nur einige Elemente eines Tupels als Platzhalter benötigt werden:

```
a = 1, 2, 3, 4
_, x, y, _ = a
# x == 2
# y == 3
```

Einzelelement-Tupel:

```
x, = 1,  # x is the value 1
x  = 1,  # x is the tuple (1,)
```

In Python 3 kann eine Zielvariable mit einem * -Prefix als *Catch-All-* Variable verwendet werden (siehe Iterables auspacken):

Python 3.x 3.0

```
first, *more, last = (1, 2, 3, 4, 5)
# first == 1
# more == [2, 3, 4]
# last == 5
```

Elemente umkehren

Elemente innerhalb eines Tupels umkehren

```
colors = "red", "green", "blue"
rev = colors[::-1]
# rev: ("blue", "green", "red")
colors = rev
# colors: ("blue", "green", "red")
```

Oder mit reversed (reversed ergibt eine Iterable, die in ein Tupel umgewandelt wird):

```
rev = tuple(reversed(colors))
# rev: ("blue", "green", "red")
colors = rev
# colors: ("blue", "green", "red")
```

Eingebaute Tupel-Funktionen

Tupel unterstützen die folgenden integrierten Funktionen

Vergleich

Wenn Elemente vom gleichen Typ sind, führt Python den Vergleich durch und gibt das Ergebnis zurück. Wenn Elemente unterschiedliche Typen sind, wird geprüft, ob es sich um Zahlen handelt.

- Bei Zahlen Vergleich durchführen.
- Wenn eines der Elemente eine Zahl ist, wird das andere Element zurückgegeben.
- Andernfalls werden Typen alphabetisch sortiert.

Wenn wir das Ende einer der Listen erreicht haben, ist die längere Liste "größer". Wenn beide Listen gleich sind, wird 0 zurückgegeben.

```
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
```

Tupel-Länge

Die Funktion `len` gibt die Gesamtlänge des Tupels zurück

```
len(tuple1)
Out: 5
```

Max eines Tupels

Die Funktion `max` gibt das Element mit dem Maximalwert aus dem Tupel zurück

```
max(tuple1)
Out: 'e'

max(tuple2)
Out: '3'
```

Min von einem Tupel

Die Funktion min gibt den Gegenstand aus dem Tupel mit dem Mindestwert zurück

```
min(tuple1)
Out: 'a'

min(tuple2)
Out: '1'
```

Konvertieren Sie eine Liste in ein Tupel

Die integrierte Funktion `tuple` wandelt eine Liste in ein Tupel.

```
list = [1,2,3,4,5]
tuple(list)
```

```
Out: (1, 2, 3, 4, 5)
```

Tuple-Verkettung

Verwenden Sie `+` um zwei Tupel zu verketten

```
tuple1 + tuple2
Out: ('a', 'b', 'c', 'd', 'e', '1', '2', '3')
```

Tupel online lesen: https://riptutorial.com/de/python/topic/927/tupel

Kapitel 170: Typ Hinweise

Syntax

- typ.Callable [[int, str], Keine] -> def func (a: int, b: str) -> Keine
- typing.Mapping [str, int] -> {"a": 1, "b": 2, "c": 3}
- typing.List [int] -> [1, 2, 3]
- typing.Set [int] -> {1, 2, 3}
- Eingabe.Optional [int] -> Keine oder int
- typ..Sequence [int] -> [1, 2, 3] oder (1, 2, 3)
- Typ.Any -> Beliebiger Typ
- tippen.Union [int, str] -> 1 oder "1"
- T = typing.TypeVar ('T') -> Generischer Typ

Bemerkungen

Type Hinting, wie in `PEP 484` , ist eine formalisierte Lösung zur statischen Angabe des Wertetyps für Python-Code. Neben dem `typing` werden mit Hilfe von Typhinweisen Python-Benutzern die Möglichkeit gegeben, ihren Code zu kommentieren, wodurch Typprüfer unterstützt werden und der Code indirekt mit weiteren Informationen dokumentiert wird.

Examples

Generische Typen

Die `typing.TypeVar` ist eine generische `typing.TypeVar` . Sein Hauptziel ist es, als Parameter / Platzhalter für generische Funktions- / Klassen- / Methodenanmerkungen zu dienen:

```
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]
```

Hinzufügen von Typen zu einer Funktion

Nehmen wir ein Beispiel für eine Funktion, die zwei Argumente empfängt und einen Wert zurückgibt, der ihre Summe angibt:

```
def two_sum(a, b):
    return a + b
```

Wenn Sie sich diesen Code `two_sum` , können Sie nicht sicher und ohne Zweifel den Typ der

Argumente für die Funktion `two_sum` . Es funktioniert beides, wenn es mit `int` Werten versorgt wird:

```
print(two_sum(2, 1))  # result: 3
```

und mit strings:

```
print(two_sum("a", "b"))  # result: "ab"
```

und mit anderen Werten, wie `list` s, `tuple` s et cetera.

Aufgrund dieser dynamischen Natur von Python-Typen, bei denen viele für eine bestimmte Operation anwendbar sind, kann ein Typprüfer nicht vernünftigerweise feststellen, ob ein Aufruf dieser Funktion zulässig ist oder nicht.

Zur Unterstützung unseres Typprüfers können wir jetzt in der Funktionsdefinition Typhinweise dazu angeben, die den zulässigen Typ angeben.

Um anzuzeigen, dass wir nur `int` Typen zulassen möchten, können wir unsere Funktionsdefinition folgendermaßen ändern:

```
def two_sum(a: int, b: int):
    return a + b
```

Anmerkungen folgen dem Argumentnamen und werden durch ein Zeichen `:` getrennt.

Um anzugeben, dass nur `str` Typen zulässig sind, ändern wir unsere Funktion, um sie anzugeben:

```
def two_sum(a: str, b: str):
    return a + b
```

Neben der Angabe des Typs der Argumente könnte auch der Rückgabewert eines Funktionsaufrufs angegeben werden. Dazu fügen Sie das Zeichen `->` gefolgt von dem Typ nach der schließenden Klammer in der Argumentliste, *aber* vor dem `:` am Ende der Funktionsdeklaration ein:

```
def two_sum(a: int, b: int) -> int:
    return a + b
```

Nun haben wir angegeben, dass der Rückgabewert beim Aufruf von `two_sum` vom Typ `int` . Ähnlich können wir geeignete Werte für `str` , `float` , `list` , `set` und andere definieren.

Obwohl Typhinweise hauptsächlich von Typenprüfern und IDEs verwendet werden, müssen Sie sie manchmal abrufen. Dies kann mit dem Spezialattribut `__annotations__` :

```
two_sum.__annotations__
# {'a': <class 'int'>, 'b': <class 'int'>, 'return': <class 'int'>}
```

Klassenmitglieder und Methoden

```
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))
```

Die Vorwärtsreferenz der aktuellen Klasse ist erforderlich, da die Anmerkungen bei der Definition der Funktion ausgewertet werden. Weiterleitungsreferenzen können auch verwendet werden, wenn auf eine Klasse verwiesen wird, die beim Import einen zirkulären Import verursachen würde.

Variablen und Attribute

Variablen werden mit Kommentaren versehen:

```
x = 3  # type: int
x = negate(x)
x = 'a type-checker might catch this error'
```

Python 3.x 3.6

Ab Python 3.6 gibt es auch eine neue Syntax für Variablenanmerkungen . Der Code oben könnte das Formular verwenden

```
x: int = 3
```

Im Gegensatz zu Kommentaren ist es auch möglich, einer Variablen, die noch nicht deklariert wurde, einen Typhinweis hinzuzufügen, ohne einen Wert festzulegen:

```
y: int
```

Wenn diese im Modul oder auf Klassenebene verwendet werden, können die `typing.get_type_hints(class_or_module)` mithilfe von `typing.get_type_hints(class_or_module)` abgerufen werden:

```
class Foo:
    x: int
    y: str = 'abc'

print(typing.get_type_hints(Foo))
# ChainMap({'x': <class 'int'>, 'y': <class 'str'>}, {})
```

Alternativ kann auf sie mit der speziellen Variablen oder dem Attribut `__annotations__` zugegriffen werden:

```
x: int
print(__annotations__)
# {'x': <class 'int'>}

class C:
    s: str
print(C.__annotations__)
# {'s': <class 'str'>}
```

NamedTuple

Das Erstellen eines namedtuple mit Typhinweisen erfolgt mit der Funktion NamedTuple aus dem `typing` :

```
import typing
Point = typing.NamedTuple('Point', [('x', int), ('y', int)])
```

Beachten Sie, dass der Name des resultierenden Typs das erste Argument der Funktion ist. Sie sollte jedoch einer Variablen mit demselben Namen zugewiesen werden, um die Arbeit der Typprüfer zu erleichtern.

Geben Sie Hinweise für Schlüsselwortargumente ein

```
def hello_world(greeting: str = 'Hello'):
    print(greeting + ' world!')
```

Beachten Sie die Leerzeichen um das Gleichheitszeichen und nicht wie die Schlüsselwortargumente normalerweise gestaltet werden.

Typ Hinweise online lesen: https://riptutorial.com/de/python/topic/1766/typ-hinweise

Kapitel 171: Überlastung

Examples

Magie / Dunder-Methoden

Magic-Methoden (auch als Dunder als Abkürzung für doppelten Unterstrich bezeichnet) dienen in Python einem ähnlichen Zweck wie das Überladen von Operatoren in anderen Sprachen. Sie erlauben einer Klasse, ihr Verhalten zu definieren, wenn sie als Operand in unären oder binären Operatorausdrücken verwendet wird. Sie dienen auch als Implementierungen, die von einigen integrierten Funktionen aufgerufen werden.

Betrachten Sie diese Implementierung von zweidimensionalen Vektoren.

```
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)
```

Nun ist es natürlich möglich, Instanzen der `Vector` Klasse in verschiedenen Ausdrücken zu verwenden.

```
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
```

Container- und Sequenztypen

Es ist möglich, Containertypen zu emulieren, die den Zugriff auf Werte nach Schlüssel oder Index unterstützen.

Betrachten Sie diese naive Implementierung einer spärlichen Liste, die nur die von Null verschiedenen Elemente speichert, um Speicherplatz zu sparen.

```
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
```

Dann können wir eine `sparselist` ähnlich einer regulären `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
```

Aufrufbare Typen

```
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
```

Umgang mit nicht implementiertem Verhalten

Wenn Ihre Klasse keinen bestimmten überladenen Operator für die angegebenen Argumenttypen implementiert, sollte `return NotImplemented` (**beachten Sie,** dass dies eine spezielle Konstante ist , die nicht mit `NotImplementedError`). Dadurch kann Python auf andere Methoden zurückgreifen, um die Operation zum Laufen zu bringen:

> Wenn `NotImplemented` zurückgegeben wird, versucht der Interpreter je nach Operator die reflektierte Operation mit dem anderen Typ oder einen anderen Fallback. Wenn alle versuchten Operationen `NotImplemented` , `NotImplemented` der Interpreter eine entsprechende Ausnahme aus.

Wenn Sie beispielsweise `x + y` angeben, wenn `x.__add__(y)` unimplementiert zurückgegeben wird, wird stattdessen `y.__radd__(x)` versucht.

```
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__
```

Da dies die *reflektierte* Methode ist, müssen wir `__add__` **und** `__radd__` implementieren, um das erwartete Verhalten in allen Fällen zu erhalten. Zum Glück können wir, da beide in diesem einfachen Beispiel dasselbe tun, eine Abkürzung nehmen.

In Benutzung:

```
>>> 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
```

Überlastung des Bedieners

Nachfolgend sind die Operatoren aufgeführt, die in Klassen überladen werden können, zusammen mit den erforderlichen Methodendefinitionen sowie ein Beispiel für den in einem Ausdruck verwendeten Operator.

Anmerkung: Die Verwendung `other` als Variablenname ist nicht obligatorisch, wird jedoch als Norm betrachtet.

Operator	Methode	Ausdruck
`+` Zusatz	`__add__(self, other)`	`a1 + a2`
`-` Subtraktion	`__sub__(self, other)`	`a1 - a2`
`*` Multiplikation	`__mul__(self, other)`	`a1 * a2`
`@` Matrixmultiplikation	`__matmul__(self, other)`	`a1 @ a2` (*Python 3.5*)
`/` Abteilung	`__div__(self, other)`	`a1 / a2` (nur *Python 2*)
`/` Abteilung	`__truediv__(self, other)`	`a1 / a2` (*Python 3*)
`//` Floor Division	`__floordiv__(self, other)`	`a1 // a2`
`%` Modulo / Rest	`__mod__(self, other)`	`a1 % a2`
`**` Power	`__pow__(self, other[, modulo])`	`a1 ** a2`
`<<` Bitweise Linksverschiebung	`__lshift__(self, other)`	`a1 << a2`

Operator	Methode	Ausdruck
`>>` Bitweise Rechtsverschiebung	`__rshift__(self, other)`	`a1 >> a2`
`&` Bitweises AND	`__and__(self, other)`	`a1 & a2`
`^` Bitweises XOR	`__xor__(self, other)`	`a1 ^ a2`
`|` (Bitweises ODER)	`__or__(self, other)`	`a1 | a2`
`-` Negation (Arithmetik)	`__neg__(self)`	`-a1`
`+` Positiv	`__pos__(self)`	`+a1`
`~` Bitweise NICHT	`__invert__(self)`	`~a1`
`<` Weniger als	`__lt__(self, other)`	`a1 < a2`
`<=` Weniger als oder gleich	`__le__(self, other)`	`a1 <= a2`
`==` Gleich	`__eq__(self, other)`	`a1 == a2`
`!=` Nicht gleich	`__ne__(self, other)`	`a1 != a2`
`>` Größer als	`__gt__(self, other)`	`a1 > a2`
`>=` Größer als oder gleich	`__ge__(self, other)`	`a1 >= a2`
`[index]` Indexoperator	`__getitem__(self, index)`	`a1[index]`
`in` in operator	`__contains__(self, other)`	`a2 in a1`
`(*args, ...)` Anrufen	`__call__(self, *args, **kwargs)`	`a1(*args, **kwargs)`

Der optionale Parameter `modulo` für `__pow__` wird nur von der integrierten `pow` -Funktion verwendet.

Jede der Methoden, die einem *binären* Operator entsprechen, hat eine entsprechende "rechte" Methode, die mit `__r` , beispielsweise `__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
```

sowie eine entsprechende Inplace-Version, beginnend mit `__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
```

Da diese Methoden nichts Besonderes sind, fügen viele andere Teile der Sprache, Teile der Standardbibliothek und sogar Module von Drittanbietern selbst magische Methoden hinzu, beispielsweise Methoden, um ein Objekt in einen Typ umzuwandeln oder die Eigenschaften des Objekts zu überprüfen. Die eingebaute Funktion `str()` ruft beispielsweise die __str `__str__` Methode des Objekts auf, sofern vorhanden. Einige dieser Anwendungen sind nachfolgend aufgeführt.

Funktion	Methode	Ausdruck
Casting auf `int`	`__int__(self)`	`int(a1)`
Absolute Funktion	`__abs__(self)`	`abs(a1)`
Casting auf `str`	`__str__(self)`	`str(a1)`
Casting zu `unicode`	`__unicode__(self)`	`unicode(a1)` (nur Python 2)
String-Darstellung	`__repr__(self)`	`repr(a1)`
Casting zu `bool`	`__nonzero__(self)`	`bool(a1)`
String-Formatierung	`__format__(self, formatstr)`	`"Hi {:abc}".format(a1)`
Hashing	`__hash__(self)`	`hash(a1)`
Länge	`__len__(self)`	`len(a1)`
Rückgängig gemacht	`__reversed__(self)`	`reversed(a1)`
Fußboden	`__floor__(self)`	`math.floor(a1)`
Decke	`__ceil__(self)`	`math.ceil(a1)`

Es gibt auch die speziellen Methoden `__enter__` und `__exit__` für Kontextmanager und viele andere.

Überlastung online lesen: https://riptutorial.com/de/python/topic/2063/uberlastung

Kapitel 172: Überprüfen der Pfadexistenz und der Berechtigungen

Parameter

Parameter	Einzelheiten
os.F_OK	Wert, der als Modusparameter von access () übergeben wird, um das Vorhandensein von Pfad zu testen.
os.R_OK	Wert, der in den Modusparameter von access () eingeschlossen werden soll, um die Lesbarkeit des Pfads zu testen.
os.W_OK	Wert, der in den Modusparameter von access () eingeschlossen werden soll, um die Schreibbarkeit des Pfads zu testen.
os.X_OK	Wert, der in den Modusparameter von access () eingeschlossen werden soll, um zu bestimmen, ob der Pfad ausgeführt werden kann.

Examples

Überprüfen Sie mit os.access

`os.access` ist eine viel bessere Lösung, um zu prüfen, ob ein Verzeichnis existiert, und es ist für Lesen und Schreiben zugänglich.

```
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)
```

Es ist auch möglich, alle Prüfungen zusammen auszuführen

```
os.access(path, os.F_OK & os.R_OK & os.W_OK & os.E_OK)
```

Alle oben genannten Werte geben " `True` wenn der Zugriff zulässig ist, und " `False` wenn dies nicht

zulässig ist. Diese sind für Unix und Windows verfügbar.

Überprüfen der Pfadexistenz und der Berechtigungen online lesen:
https://riptutorial.com/de/python/topic/1262/uberprufen-der-pfadexistenz-und-der-berechtigungen

Kapitel 173: Überschreiben der Methode

Examples

Grundlegende Methode überschreiben

Hier ist ein Beispiel für die grundlegende Überschreibung in Python (aus Gründen der Klarheit und Kompatibilität mit Python 2 und 3, unter Verwendung der neuen Stilklasse und `print` mit `()`):

```
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
```

Wenn die `Child` Klasse erstellt wird, erbt sie die Methoden der `Parent` Klasse. Dies bedeutet, dass alle Methoden, die die übergeordnete Klasse hat, auch die untergeordnete Klasse haben. Im Beispiel ist die `introduce` für die `Child` Klasse definiert, da sie für `Parent` definiert ist, obwohl sie nicht explizit in der Klassendefinition von `Child` .

In diesem Beispiel tritt das Überschreiben auf, wenn `Child` seine eigene Methode `print_name` definiert. Wenn diese Methode nicht deklariert wurde, hätte `c.print_name()` `"Parent"` gedruckt. Allerdings `Child` hat die außer Kraft gesetzt `Parent` ,s Definition von `print_name` , und so jetzt beim Aufruf `c.print_name()` , das Wort `"Child"` wird gedruckt.

Überschreiben der Methode online lesen:
https://riptutorial.com/de/python/topic/3131/uberschreiben-der-methode

Kapitel 174: Umgang mit der Global Interpreter Lock (GIL)

Bemerkungen

Warum gibt es eine GIL?

Die GIL gibt es seit der Gründung von Python-Threads im Jahr 1992 in CPython. Sie soll Thread-Sicherheit beim Ausführen von Python-Code gewährleisten. Mit einer GIL geschriebene Python-Interpreter verhindern, dass mehrere native Threads Python-Bytecodes gleichzeitig ausführen. Dies macht es Plugins leicht, sicherzustellen, dass ihr Code Thread-sicher ist: Sperren Sie einfach die GIL, und nur Ihr aktiver Thread kann ausgeführt werden, sodass Ihr Code automatisch Thread-sicher ist.

Kurze Version: Die GIL stellt sicher, dass unabhängig von der Anzahl der Prozessoren und Threads *nur ein Thread eines Python-Interpreters gleichzeitig ausgeführt wird.*

Dies hat viele Vorteile bei der Benutzerfreundlichkeit, aber auch viele negative Vorteile.

Beachten Sie, dass eine GIL keine Anforderung der Python-Sprache ist. Daher können Sie nicht direkt vom Standard-Python-Code auf die GIL zugreifen. Nicht alle Implementierungen von Python verwenden eine GIL.

Dolmetscher, die eine GIL haben: CPython, PyPy, Cython (aber Sie können die GIL mit `nogil` deaktivieren)

Interpreter ohne GIL: Jython, IronPython

Details zur Funktionsweise der GIL:

Wenn ein Thread läuft, sperrt er die GIL. Wenn ein Thread ausgeführt werden soll, fordert er die GIL an und wartet, bis sie verfügbar ist. In CPython, vor Version 3.2, prüfte der laufende Thread nach einer bestimmten Anzahl von Python-Anweisungen, ob anderer Code die Sperre wünschte (d. H. Er gab die Sperre auf und forderte sie erneut auf). Diese Methode neigte dazu, einen Thread-Hunger zu verursachen, hauptsächlich, weil der Thread, der die Sperre aufgehoben hatte, ihn erneut erwerben würde, bevor die wartenden Threads eine Chance hatten, aufzuwachen. Seit 3.2 warten Threads, die möchten, dass die GIL für einige Zeit auf die Sperre wartet, und setzen danach eine gemeinsam genutzte Variable, die den laufenden Thread zwingt, nachzugeben. Dies kann jedoch zu drastisch längeren Ausführungszeiten führen. Weitere Informationen finden Sie unter den Links von dabeaz.com (im Abschnitt mit den Referenzen).

CPython gibt die GIL automatisch frei, wenn ein Thread eine E / A-Operation ausführt. Bildbearbeitungsbibliotheken und numerische Anzahlprozeduren geben die GIL vor der

Verarbeitung frei.

Vorteile der GIL

Für Dolmetscher, die die GIL verwenden, ist die GIL systemisch. Es wird verwendet, um den Status der Anwendung zu erhalten. Vorteile umfassen:

- Garbage Collection - Thread-sichere Referenzzählungen müssen geändert werden, während die GIL gesperrt ist. *In CPython ist die gesamte Garbarge-Sammlung an die GIL gebunden.* Dies ist eine große Sache. In dem Wiki-Artikel zu Python.org über die GIL (aufgeführt in Referenzen unten) finden Sie Details dazu, was noch funktionieren muss, wenn Sie die GIL entfernen möchten.
- Einfach für Programmierer, die sich mit der GIL befassen - alles zu sperren ist einfach, aber leicht zu programmieren
- Erleichtert den Import von Modulen aus anderen Sprachen

Folgen der GIL

Die GIL erlaubt es nur einem Thread, jeweils Python-Code innerhalb des Python-Interpreters auszuführen. Das bedeutet, dass das Multithreading von Prozessen, die strengen Python-Code ausführen, einfach nicht funktioniert. Wenn Sie Threads gegen die GIL verwenden, haben Sie mit den Threads wahrscheinlich eine schlechtere Leistung als bei einem einzelnen Thread.

Verweise:

https://wiki.python.org/moin/GlobalInterpreterLock - eine kurze Zusammenfassung der Funktionsweise sowie genaue Angaben zu allen Vorteilen

http://programmers.stackexchange.com/questions/186889/why-was-python-wrched-with-the-gil - übersichtlich geschriebene Zusammenfassung

http://www.dabeaz.com/python/UnderstandingGIL.pdf - wie die GIL funktioniert und warum sie auf mehreren Kernen langsamer wird

http://www.dabeaz.com/GIL/gilvis/index.html - Visualisierung der Daten, die zeigen, wie die GIL Threads blockiert

http://jeffknupp.com/blog/2012/03/31/pythons-hardest-problem/ - Geschichte des GIL-Problems einfach zu verstehen

https://jeffknupp.com/blog/2013/06/30/pythons-hardest-problem-revisited/ - Details zur Lösung der Einschränkungen der GIL

Examples

Multiprocessing.Pool

Die einfache Antwort auf die Frage, wie Threads in Python verwendet werden sollen, lautet: "Nicht. Verwenden Sie stattdessen Prozesse." Mit dem Multiprocessing-Modul können Sie Prozesse mit einer ähnlichen Syntax wie das Erstellen von Threads erstellen, aber ich bevorzuge die Verwendung ihres praktischen Pool-Objekts.

Mit dem Code, mit dem David Beazley zuerst die Gefahren von Threads gegen die GIL aufgezeigt hat , schreiben wir ihn mit Multiprocessing neu. Pool :

David Beazleys Code, der Probleme beim Einfädeln von GIL zeigte

```
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
```

Mit multiprocessing.Pool neu geschrieben:

```
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)
```

Anstatt Threads zu erstellen, werden neue Prozesse erstellt. Da jeder Prozess ein eigener Interpreter ist, gibt es keine GIL-Kollisionen. multiprocessing.Pool öffnet so viele Prozesse, wie Kerne auf der Maschine vorhanden sind, im obigen Beispiel wären jedoch nur zwei erforderlich. In

einem realen Szenario möchten Sie, dass Ihre Liste mindestens so lang ist, wie Prozessoren auf Ihrem Computer vorhanden sind. Der Pool führt die von Ihnen angegebene Funktion mit jedem Argument aus, bis zu der Anzahl der erstellten Prozesse. Wenn die Funktion abgeschlossen ist, werden alle verbleibenden Funktionen in der Liste für diesen Prozess ausgeführt.

Ich habe festgestellt, dass die Prozesse auch dann noch vorhanden sind, wenn Sie die `with` Anweisung verwenden. Um Ressourcen zu bereinigen, schließe ich immer meine Pools.

Cython Nogil:

Cython ist ein alternativer Python-Interpreter. Es verwendet die GIL, kann jedoch deaktiviert werden. Sehen Sie ihre Dokumentation

Als Beispiel verwenden wir den Code , mit dem David Beazley zuerst die Gefahren von Threads gegen die GIL aufzeigte, mit Nogil neu schreiben:

David Beazleys Code, der Probleme beim Einfädeln von GIL zeigte

```
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
```

Mit Nogil neu geschrieben (NUR FUNKTIONIERT IN CYTHON):

```
from threading import Thread
import time
def countdown(n):
    while n > 0:
        n -= 1

COUNT = 10000000

with nogil:
    t1 = Thread(target=countdown,args=(COUNT/2,))
```

```
    t2 = Thread(target=countdown,args=(COUNT/2,))
    start = time.time()
    t1.start();t2.start()
    t1.join();t2.join()

end = time.time()
print end-start
```

So einfach ist das, solange Sie Cython verwenden. Beachten Sie, dass Sie in der Dokumentation darauf hingewiesen werden, dass Sie keine Python-Objekte ändern müssen:

> Code im Rumpf der Anweisung darf Python-Objekte auf keine Weise manipulieren und darf nichts aufrufen, das Python-Objekte manipuliert, ohne die GIL erneut zu erwerben. Cython prüft dies derzeit nicht.

Umgang mit der Global Interpreter Lock (GIL) online lesen:
https://riptutorial.com/de/python/topic/4061/umgang-mit-der-global-interpreter-lock--gil-

Kapitel 175: Unicode

Examples

Kodierung und Dekodierung

Kodieren Sie immer von Unicode in Bytes. In dieser Richtung können **Sie die Kodierung auswählen** .

```
>>> u'🐍'.encode('utf-8')
'\xf0\x9f\x90\x8d'
```

Der andere Weg ist die *Decodierung* von Bytes in Unicode. In dieser Richtung müssen **Sie wissen, was die Kodierung ist** .

```
>>> b'\xf0\x9f\x90\x8d'.decode('utf-8')
u'\U0001f40d'
```

Unicode online lesen: https://riptutorial.com/de/python/topic/5618/unicode

Kapitel 176: Unicode und Bytes

Syntax

- str.encode (Kodierung, Fehler = 'streng')
- bytes.decode (Kodierung, Fehler = 'streng')
- offen (Dateiname, Modus, Kodierung = Keine)

Parameter

Parameter	Einzelheiten
Codierung	Die zu verwendende Kodierung, zB `'ascii'` , `'utf8'` usw.
Fehler	Der Fehlermodus, z. B. `'replace'` , um ungültige Zeichen durch Fragezeichen zu ersetzen, `'ignore'` , um ungültige Zeichen zu ignorieren usw.

Examples

Grundlagen

In Python 3 ist `str` der Typ für Unicode-aktivierte Zeichenfolgen, während `bytes` der Typ für Sequenzen von unformatierten Bytes ist.

```
type("f") == type(u"f")  # True, <class 'str'>
type(b"f")               # <class 'bytes'>
```

In Python 2 war eine zufällige Zeichenfolge standardmäßig eine Folge roher Bytes, und die Unicode-Zeichenfolge bestand aus jeder Zeichenfolge mit dem Präfix "u".

```
type("f") == type(b"f")  # True, <type 'str'>
type(u"f")               # <type 'unicode'>
```

Unicode in Bytes

Unicode-Zeichenfolgen können mit `.encode(encoding)` in Bytes konvertiert werden.

Python 3

```
>>> "£13.55".encode('utf8')
b'\xc2\xa313.55'
>>> "£13.55".encode('utf16')
```

```
b'\xff\xfe\xa3\x001\x003\x00.\x005\x005\x00'
```

Python 2

In py2 lautet die Standardkonsolencodierung `sys.getdefaultencoding() == 'ascii'` und nicht `utf-8` wie in py3. Daher ist das Drucken wie in dem vorherigen Beispiel nicht direkt möglich.

```
>>> 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
```

Wenn die Codierung die Zeichenfolge nicht verarbeiten kann, wird ein UnicodeEncodeError-Fehler ausgelöst:

```
>>> "£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)
```

Bytes bis Unicode

Bytes können mit `.decode(encoding)` in Unicode-Zeichenfolgen `.decode(encoding)` .

Eine Folge von Bytes kann nur mit der entsprechenden Kodierung in einen Unicode-String umgewandelt werden!

```
>>> b'\xc2\xa313.55'.decode('utf8')
'£13.55'
```

Wenn die Codierung die Zeichenfolge nicht verarbeiten kann, wird ein `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
```

Behandlung von Codierungs- / Decodierungsfehlern

`.encode` und `.decode` beide `.decode` .

Der Standardwert ist `'strict'` , was bei Fehlern zu Ausnahmen führt. Andere Modi sind verzeihender.

Codierung

```
>>> "£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'
```

Dekodierung

```
>>> 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'
```

Moral

Aus dem Vorstehenden ist klar, dass es wichtig ist, Ihre Kodierungen beim Umgang mit Unicode und Bytes gerade zu halten.

Datei I / O

Dateien, die in einem nicht binären Modus geöffnet wurden (z. B. `'r'` oder `'w'`), behandeln Strings. Die Deafult-Kodierung lautet `'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")
```

Dateien, die in einem binären Modus geöffnet werden (z. B. `'rb'` oder `'wb'`), behandeln Bytes. Es kann kein Kodierungsargument angegeben werden, da keine Kodierung vorhanden ist.

```
open(fn, mode='wb')  # open file for writing bytes

# ERROR: cannot write string when bytes is expected:
open(fn, mode='wb').write("hi")
```

Unicode und Bytes online lesen: https://riptutorial.com/de/python/topic/1216/unicode-und-bytes

Kapitel 177: Unit Testing

Bemerkungen

Es gibt verschiedene Gerätetest-Tools für Python. Dieses Dokumentationsthema beschreibt das grundlegende `unittest` . Andere Testwerkzeuge umfassen `py.test` und `nosetests` . Diese Python-Dokumentation zum Testen vergleicht mehrere dieser Tools, ohne in die Tiefe zu gehen.

Examples

Ausnahmen testen

Programme werfen Fehler, wenn zum Beispiel eine falsche Eingabe erfolgt. Aus diesem Grund muss sichergestellt werden, dass ein Fehler ausgegeben wird, wenn eine falsche Eingabe erfolgt. Aus diesem Grund müssen wir nach einer genauen Ausnahme suchen. In diesem Beispiel werden wir die folgende Ausnahme verwenden:

```
class WrongInputException(Exception):
    pass
```

Diese Ausnahme wird ausgelöst, wenn falsche Eingaben gemacht werden, in dem folgenden Kontext, in dem immer eine Zahl als Texteingabe erwartet wird.

```
def convert2number(random_input):
    try:
        my_input = int(random_input)
    except ValueError:
        raise WrongInputException("Expected an integer!")
    return my_input
```

Um zu prüfen, ob eine Ausnahme `assertRaises` wurde, verwenden wir `assertRaises` , um diese Ausnahme zu überprüfen. `assertRaises` kann auf zwei Arten verwendet werden:

1. Verwenden des regulären Funktionsaufrufs. Das erste Argument hat den Ausnahmetyp, zweitens eine aufrufbare Funktion (normalerweise eine Funktion), und die übrigen Argumente werden an diese aufrufbare Komponente übergeben.
2. Verwenden Sie eine `with` Klausel, und geben Sie der Funktion nur den Ausnahmetyp an. Dies hat den Vorteil, dass mehr Code ausgeführt werden kann, der jedoch mit Vorsicht verwendet werden sollte, da mehrere Funktionen dieselbe Ausnahme verwenden können, was problematisch sein kann. Ein Beispiel: with self.assertRaises (WrongInputException): convert2number ("keine Zahl")

Dieser erste wurde in dem folgenden Testfall implementiert:

```
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()
```

Es kann auch erforderlich sein, nach einer Ausnahme zu suchen, die nicht hätte ausgelöst werden sollen. Ein Test schlägt jedoch automatisch fehl, wenn eine Ausnahme ausgelöst wird, und ist daher möglicherweise überhaupt nicht erforderlich. Um die Optionen zu zeigen, zeigt die zweite Testmethode einen Fall, wie geprüft werden kann, ob eine Ausnahme nicht ausgelöst wird. Grundsätzlich wird die Ausnahme erkannt und der Test mit der `fail` Methode `fail` .

Verspottungsfunktionen mit unittest.mock.create_autospec

Eine Möglichkeit zum `create_autospec` einer Funktion ist die Verwendung der Funktion `create_autospec` , mit der ein Objekt gemäß seinen Angaben verspottet wird. Mit Funktionen können wir dies nutzen, um sicherzustellen, dass sie entsprechend aufgerufen werden.

Mit einer Funktion in `custom_math.py` `multiply` :

```
def multiply(a, b):
    return a * b
```

Und eine Funktion `multiples_of` in `process_math.py` :

```
from custom_math import multiply


def multiples_of(integer, *args, num_multiples=0, **kwargs):
    """
    :rtype: list
    """
    multiples = []

    for x in range(1, num_multiples + 1):
        """
        Passing in args and kwargs here will only raise TypeError if values were
        passed to multiples_of function, otherwise they are ignored. This way we can
        test that multiples_of is used correctly. This is here for an illustration
        of how create_autospec works. Not recommended for production code.
        """
        multiple = multiply(integer,x, *args, **kwargs)
        multiples.append(multiple)

    return multiples
```

Wir können `multiples_of` alleine testen, `multiply` wir `multiply` . Im folgenden Beispiel wird die Python-Standardbibliothek unittest verwendet, dies kann jedoch auch mit anderen

Testframeworks, wie z.

```
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
```

Testen Sie Setup und Teardown innerhalb eines Tests. TestCase

Manchmal möchten wir einen Kontext für jeden Test erstellen, unter dem ausgeführt wird. Die `setUp` Methode wird vor jedem Test in der Klasse ausgeführt. `tearDown` wird am Ende jedes Tests ausgeführt. Diese Methoden sind optional. Denken Sie daran, dass Testfälle häufig in kooperativer Mehrfachvererbung verwendet werden. Sie sollten also immer den `super` - `setUp` in diesen Methoden `tearDown` , damit auch die Methoden `setUp` und `tearDown` der `setUp` aufgerufen werden. Die `TestCase` von `TestCase` stellt leere `setUp` und `tearDown` Methoden `tearDown` , sodass sie ohne Ausnahmen aufgerufen werden können:

```
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()
```

Beachten Sie, dass es in python2.7 + auch die `addCleanup` Methode gibt, die Funktionen registriert, die nach dem Ausführen des Tests aufgerufen werden sollen. Im Gegensatz zu `tearDown` das nur aufgerufen wird, wenn `setUp` erfolgreich ist, werden über `addCleanup` registrierte Funktionen auch im Fall einer nicht behandelten Ausnahme in `setUp` . Als ein konkretes Beispiel kann diese Methode häufig gesehen werden, wenn verschiedene Mocks entfernt werden, die während des Tests

registriert wurden:

```
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)
```

Ein weiterer Vorteil der Registrierung von Bereinigungen auf diese Weise ist, dass der Programmierer den Bereinigungscode neben den Setup-Code stellen kann und Sie `tearDown` falls ein Subclasser vergisst, `super` in `tearDown` .

Ausnahmen geltend machen

Sie können mit zwei integrierten Methoden testen, ob eine Funktion mit dem integrierten Test eine Ausnahme auslöst.

Verwenden eines Kontextmanagers

```
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)
```

Dadurch wird der Code im Kontext-Manager ausgeführt. Wenn er erfolgreich ist, schlägt der Test fehl, da die Ausnahme nicht ausgelöst wurde. Wenn der Code eine Ausnahme des richtigen Typs auslöst, wird der Test fortgesetzt.

Sie können auch den Inhalt der ausgelösten Ausnahme abrufen, wenn Sie weitere Zusicherungen dagegen ausführen möchten.

```
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')
```

Durch die Bereitstellung einer aufrufbaren Funktion

```
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)
```

Die zu prüfende Ausnahme muss der erste Parameter sein und eine aufrufbare Funktion muss als zweiter Parameter übergeben werden. Alle anderen angegebenen Parameter werden direkt an die aufgerufene Funktion übergeben, sodass Sie die Parameter angeben können, die die Ausnahme auslösen.

Assertions innerhalb Unittests auswählen

Während Python eine `assert` Anweisung hat , hat das Test-Framework für Python-Einheiten bessere Assertions für Tests: Sie sind informativer bei Fehlern und hängen nicht vom Debug-Modus der Ausführung ab.

Die einfachste Behauptung ist `assertTrue` , die folgendermaßen verwendet werden kann:

```
import unittest

class SimplisticTest(unittest.TestCase):
    def test_basic(self):
        self.assertTrue(1 + 1 == 2)
```

Dies wird gut funktionieren, aber die Zeile oben durch ersetzen

```
        self.assertTrue(1 + 1 == 3)
```

wird versagen.

Die `assertTrue` Behauptung ist wahrscheinlich die allgemeinste Behauptung, da alles, was getestet wurde, als boolesche Bedingung angesehen werden kann, aber oft gibt es bessere Alternativen. Wenn Sie wie oben auf Gleichheit prüfen, sollten Sie besser schreiben

```
        self.assertEqual(1 + 1, 3)
```

Wenn der Erstere fehlschlägt, wird die Nachricht angezeigt

```
======================================================================
```

```
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
```

Wenn letzteres fehlschlägt, wird die Nachricht angezeigt

```
======================================================================

FAIL: test (__main__.TruthTest)

----------------------------------------------------------------------

Traceback (most recent call last):

  File "stuff.py", line 6, in test

    self.assertEqual(1 + 1, 3)
AssertionError: 2 != 3
```

was informativer ist (es wertete tatsächlich das Ergebnis der linken Seite aus).

Die Liste der Zusicherungen finden Sie in der Standarddokumentation . Im Allgemeinen ist es eine gute Idee, die Assertion zu wählen, die der Bedingung am besten entspricht. Wie oben gezeigt, ist es daher besser, `assertEqual` zu verwenden als `assertTrue` , um zu behaupten, dass `1 + 1 == 2` ist. `assertIsNone` zu behaupten, dass `a is None` , ist es besser, `assertIsNone` zu verwenden als `assertEqual` .

Beachten Sie auch, dass die Aussagen negative Formen haben. So hat `assertEqual` seinen negativen Gegenwert `assertNotEqual` und `assertIsNone` seinen negativen Gegenwert `assertIsNotNone` . Die Verwendung der negativen Gegenstücke führt gegebenenfalls zu klareren Fehlermeldungen.

Unit-Tests mit Pytest

pytest installieren:

```
pip install pytest
```

Tests vorbereiten:

```
mkdir tests
touch tests/test_docker.py
```

Funktionen zum Testen in `docker_something/helpers.py` :

```
from subprocess import Popen, PIPE
# this Popen is monkeypatched with the fixture `all_popens`

def copy_file_to_docker(src, dest):
    try:
        result = Popen(['docker','cp', src, 'something_cont:{}'.format(dest)], stdout=PIPE,
stderr=PIPE)
        err = result.stderr.read()
        if err:
            raise Exception(err)
    except Exception as e:
        print(e)
    return result

def docker_exec_something(something_file_string):
    fl = Popen(["docker", "exec", "-i", "something_cont", "something"], stdin=PIPE,
stdout=PIPE, stderr=PIPE)
    fl.stdin.write(something_file_string)
    fl.stdin.close()
    err = fl.stderr.read()
    fl.stderr.close()
    if err:
        print(err)
        exit()
    result = fl.stdout.read()
    print(result)
```

Der Test importiert `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
```

Verspotten einer Datei wie eines Objekts in `test_docker.py` :

```
class MockBytes():
    '''Used to collect bytes
    '''
    all_read = []
    all_write = []
    all_close = []

    def read(self, *args, **kwargs):
        # print('read', args, kwargs, dir(self))
        self.all_read.append((self, args, kwargs))

    def write(self, *args, **kwargs):
        # print('wrote', args, kwargs)
        self.all_write.append((self, args, kwargs))

    def close(self, *args, **kwargs):
        # print('closed', self, args, kwargs)
        self.all_close.append((self, args, kwargs))
```

```
    def get_all_mock_bytes(self):
        return self.all_read, self.all_write, self.all_close
```

Affen-Patches mit pytest in `test_docker.py` :

```
@pytest.fixture
def all_popens(monkeypatch):
    '''This fixture overrides / mocks the builtin Popen
        and replaces stdin, stdout, stderr with a MockBytes object

        note: monkeypatch is magically imported
    '''
    all_popens = []

    class MockPopen(object):
        def __init__(self, args, stdout=None, stdin=None, stderr=None):
            all_popens.append(self)
            self.args = args
            self.byte_collection = MockBytes()
            self.stdin = self.byte_collection
            self.stdout = self.byte_collection
            self.stderr = self.byte_collection
            pass
    monkeypatch.setattr(helpers, 'Popen', MockPopen)

    return all_popens
```

Beispieltests müssen mit dem Präfix `test_` in der Datei `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])
```

Ausführen der Tests nacheinander:

```
py.test -k test_docker_install tests
```

```
py.test -k test_copy_file_to_docker tests
py.test -k test_docker_exec_something tests
```

alle Tests in den laufenden `tests` Ordner:

```
py.test -k test_ tests
```

Unit Testing online lesen: https://riptutorial.com/de/python/topic/631/unit-testing

Kapitel 178: Unterschied zwischen Modul und Paket

Bemerkungen

Es ist möglich, ein Python-Paket in eine ZIP-Datei zu packen und es auf diese Weise zu verwenden, wenn Sie diese Zeilen am Anfang Ihres Skripts hinzufügen:

```
import sys
sys.path.append("package.zip")
```

Examples

Module

Ein Modul ist eine einzelne Python-Datei, die importiert werden kann. Die Verwendung eines Moduls sieht folgendermaßen aus:

`module.py`

```
def hi():
    print("Hello world!")
```

`my_script.py`

```
import module
module.hi()
```

in einem Dolmetscher

```
>>> from module import hi
>>> hi()
# Hello world!
```

Pakete

Ein Paket besteht aus mehreren Python-Dateien (oder -Modulen) und kann sogar Bibliotheken enthalten, die in C oder C ++ geschrieben sind. Es handelt sich nicht um eine einzelne Datei, sondern um eine gesamte Ordnerstruktur, die wie folgt aussehen kann:

Ordner - `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!")
```

Alle Python-Pakete müssen eine `__init__.py` . `__init__.py` Datei enthalten. Wenn Sie ein Paket in Ihr Skript `import package` (`import package`), wird das Skript `__init__.py` ausgeführt, mit dem Sie auf alle Funktionen im Paket zugreifen können. In diesem Fall können Sie die Funktionen `package.hi` und `package.woof` verwenden.

Unterschied zwischen Modul und Paket online lesen:
https://riptutorial.com/de/python/topic/3142/unterschied-zwischen-modul-und-paket

Kapitel 179: Unveränderbare Datentypen (int, float, str, tuple und frozensets)

Examples

Einzelne Zeichen von Strings können nicht zugewiesen werden

```
foo = "bar"
foo[0] = "c" # Error
```

Der unveränderliche Variablenwert kann nicht geändert werden, sobald er erstellt wurde.

Die einzelnen Mitglieder von Tuple können nicht zugewiesen werden

```
foo = ("bar", 1, "Hello!",)
foo[1] = 2 # ERROR!!
```

Die zweite Zeile würde einen Fehler zurückgeben, da einmal erstellte Tupelmitglieder nicht zuweisbar sind. Wegen der Unveränderlichkeit des Tupels.

Frozensets sind unveränderlich und nicht zuordenbar

```
foo = frozenset(["bar", 1, "Hello!"])
foo[2] = 7 # ERROR
foo.add(3) # ERROR
```

Die zweite Zeile würde einen Fehler zurückgeben, da einmal erstellte frozenset-Mitglieder nicht zuweisbar sind. Die dritte Zeile würde einen Fehler zurückgeben, da Frozensets keine Funktionen unterstützen, die Mitglieder bearbeiten können.

Unveränderbare Datentypen (int, float, str, tuple und frozensets) online lesen: https://riptutorial.com/de/python/topic/4806/unveranderbare-datentypen--int--float--str--tuple-und-frozensets-

Kapitel 180: Urllib

Examples

HTTP GET

Python 2.x 2.7

Python 2

```
import urllib
response = urllib.urlopen('http://stackoverflow.com/documentation/')
```

Bei Verwendung von `urllib.urlopen()` wird ein `urllib.urlopen()` , das ähnlich wie eine Datei behandelt werden kann.

```
print response.code
# Prints: 200
```

Der `response.code` repräsentiert den http-Rückgabewert. 200 ist in Ordnung, 404 ist nicht gefunden usw.

```
print response.read()
'<!DOCTYPE html>\r\n<html>\r\n<head>\r\n\r\n<title>Documentation - Stack. etc'
```

`response.read()` und `response.readlines()` können verwendet werden, um die aus der Anforderung zurückgegebene tatsächliche HTML-Datei zu lesen. Diese Methoden funktionieren ähnlich wie `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>
```

Das Modul wurde für Python 3.x aktualisiert, die Anwendungsfälle bleiben jedoch im Wesentlichen

gleich. `urllib.request.urlopen` gibt ein ähnliches dateiähnliches Objekt zurück.

HTTP POST

Um POST-Daten zu übergeben, übergeben Sie die codierten Abfrageargumente als Daten an 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'
```

Empfangene Bytes nach Inhaltstypen codieren

Die empfangenen Bytes müssen mit der korrekten Zeichencodierung dekodiert werden, um als Text interpretiert zu werden:

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)
```

Urllib online lesen: https://riptutorial.com/de/python/topic/2645/urllib

Kapitel 181: Variabler Geltungsbereich und Bindung

Syntax

- global a, b, c
- nichtlokal a, b
- x = etwas # bindet x
- (x, y) = etwas # bindet x und y
- x + = etwas # bindet x. Ähnlich für alle anderen "op ="
- del x # bindet x
- für x in etwas: # bindet x
- mit etwas als x: # bindet x
- Ausnahme Ausnahme als Ex: # bindet Ex innerhalb des Blocks

Examples

Globale Variablen

In Python werden Variablen innerhalb von Funktionen genau dann als lokal betrachtet, wenn sie auf der linken Seite einer Zuweisungsanweisung oder einem anderen verbindlichen Vorkommen erscheinen. Andernfalls wird eine solche Bindung in umschließenden Funktionen bis zum globalen Umfang nachgeschlagen. Dies gilt auch dann, wenn die Zuweisungsanweisung nie ausgeführt wird.

```
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
```

Normalerweise spiegelt eine Zuweisung innerhalb eines Bereichs alle äußeren Variablen desselben Namens:

```
x = 'Hi'

def change_local_x():
    x = 'Bye'
    print(x)
change_local_x()  # prints Bye
print(x)  # prints Hi
```

Wenn Sie einen Namen `global` deklarieren, bedeutet dies, dass für den Rest des Bereichs alle Zuweisungen zu dem Namen auf der obersten Ebene des Moduls erfolgen:

```
x = 'Hi'

def change_global_x():
    global x
    x = 'Bye'
    print(x)

change_global_x()  # prints Bye
print(x)  # prints Bye
```

Das `global` Schlüsselwort bedeutet, dass Zuweisungen auf der obersten Ebene des Moduls stattfinden, nicht auf der obersten Ebene des Programms. Andere Module benötigen weiterhin den üblichen gepunkteten Zugriff auf Variablen innerhalb des Moduls.

Zusammenfassend: Um zu wissen, ob eine Variable `x` für eine Funktion lokal ist, sollten Sie die *gesamte* Funktion lesen:

1. Wenn Sie `global x` , ist `x` eine **globale** Variable
2. Wenn Sie `nonlocal x` , gehört `x` zu einer einschließenden Funktion und ist weder lokal noch global
3. Wenn Sie `x = 5` oder `for x in range(3)` oder eine andere Bindung gefunden haben, ist `x` eine **lokale** Variable
4. Ansonsten gehört `x` zu einem einschließenden Bereich (Funktionsbereich, globaler Bereich oder Builtins).

Lokale Variablen

Wenn ein Name *an* eine Funktion *gebunden* ist, kann er standardmäßig nur innerhalb der Funktion aufgerufen werden:

```
def foo():
    a = 5
    print(a) # ok

print(a) #  NameError: name 'a' is not defined
```

Kontrollfluss - Konstrukte haben keine Auswirkungen auf den Umfang (mit Ausnahme von der `except`

), aber variable Zugriff, der noch nicht zugewiesen wurde , ist ein Fehler:

```
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
```

Übliche Bindungsoperationen sind Zuweisungen, `for` Schleifen und erweiterte Zuordnungen wie `a += 5`

Nichtlokale Variablen

Python 3.x 3.0

Python 3 hat ein neues Schlüsselwort namens **nonlocal** hinzugefügt. Das nichtlokale Schlüsselwort fügt dem inneren Bereich eine Bereichsüberschreibung hinzu. In PEP 3104 können Sie alles darüber lesen. Dies wird am besten anhand einiger Codebeispiele veranschaulicht. Eines der häufigsten Beispiele ist das Erstellen einer Funktion, die inkrementiert werden kann:

```
def counter():
    num = 0
    def incrementer():
        num += 1
        return num
    return incrementer
```

Wenn Sie versuchen, diesen Code auszuführen, erhalten Sie einen **UnboundLocalError,** da die **num -Variable** referenziert wird, bevor sie in der innersten Funktion zugewiesen wird. Fügen wir dem Mix nichtlokal hinzu:

```
def counter():
    num = 0
    def incrementer():
        nonlocal num
        num += 1
        return num
    return incrementer

c = counter()
c() # = 1
c() # = 2
c() # = 3
```

Grundsätzlich erlaubt Ihnen `nonlocal` das Zuweisen von Variablen in einem äußeren, jedoch nicht in einem globalen Bereich. Sie können also `nonlocal` in unserer `counter` Funktion verwenden, da sie dann versuchen würde, einen globalen Gültigkeitsbereich zuzuweisen. `SyntaxError` Sie es aus und Sie erhalten schnell einen `SyntaxError` . Stattdessen müssen Sie `nonlocal` in einer

verschachtelten Funktion verwenden.

(Beachten Sie, dass die hier vorgestellte Funktionalität besser mit Generatoren implementiert wird.)

Verbindliches Vorkommen

```
x = 5
x += 7
for x in iterable: pass
```

Jede der obigen Aussagen ist *verbindlich* - `x` wird an das mit `5` bezeichnete Objekt gebunden. Wenn diese Anweisung in einer Funktion angezeigt wird, ist `x` standardmäßig funktionslokal. Eine Liste der Bindeanweisungen finden Sie im Abschnitt "Syntax".

Funktionen überspringen Klassenbereich beim Nachschlagen von Namen

Klassen haben während der Definition einen lokalen Gültigkeitsbereich, aber Funktionen innerhalb der Klasse verwenden diesen Gültigkeitsbereich nicht, wenn Sie nach Namen suchen. Da Lambdas Funktionen sind und Erkenntnisse mit dem Funktionsumfang implementiert werden, kann dies zu überraschendem Verhalten führen.

```
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
```

Benutzer, die mit der Funktionsweise dieses Bereichs nicht vertraut sind, erwarten möglicherweise eine `b class` von `b` , `c` und `e` .

Ab PEP 227 :

> Namen im Klassenbereich sind nicht zugänglich. Namen werden im innersten einschließenden Funktionsumfang aufgelöst. Wenn eine Klassendefinition in einer Kette verschachtelter Bereiche auftritt, überspringt der Auflösungsprozess

> Klassendefinitionen.

Aus der Python-Dokumentation zu Benennung und Bindung :

> Der Umfang der in einem Klassenblock definierten Namen ist auf den Klassenblock beschränkt. Sie erstreckt sich nicht auf die Codeblöcke von Methoden - dies schließt Verständnis und Generatorausdrücke ein, da sie mit einem Funktionsumfang implementiert werden. Dies bedeutet, dass Folgendes fehlschlägt:

```
class A:
    a = 42
    b = list(a + i for i in range(10))
```

In diesem Beispiel werden Referenzen dieser Antwort von Martijn Pieters verwendet, die eine detailliertere Analyse dieses Verhaltens enthalten.

Der Befehl del

Dieser Befehl hat mehrere verwandte, aber unterschiedliche Formen.

del v

Wenn `v` eine Variable ist, entfernt der Befehl `del v` die Variable aus ihrem Gültigkeitsbereich. Zum Beispiel:

```
x = 5
print(x) # out: 5
del x
print(x) # NameError: name 'f' is not defined
```

> Beachten Sie, dass `del` ein *verbindliches Vorkommen* ist. *Dies* bedeutet, dass, wenn nicht ausdrücklich anders angegeben (`nonlocal` oder `global`), `del v` `v` lokalen Bereich lokal wird. Wenn Sie `v` in einem äußeren Bereich löschen `nonlocal v` , verwenden Sie `nonlocal v` oder `global v` im gleichen Geltungsbereich der `del v` Anweisung.

In allen folgenden Fällen handelt es sich bei der Absicht eines Befehls um ein Standardverhalten, das jedoch nicht von der Sprache erzwungen wird. Eine Klasse kann so geschrieben werden, dass diese Absicht ungültig wird.

del v.name

Dieser Befehl löst einen Aufruf an `v.__delattr__(name)` .

Die Absicht ist , das Attribut zu machen `name` nicht verfügbar. Zum Beispiel:

```
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]

Dieser Befehl löst einen Aufruf an `v.__delitem__(item)` .

Die Absicht ist, dass das `item` nicht in das vom Objekt `v` implementierte Mapping gehört. Zum Beispiel:

```
x = {'a': 1, 'b': 2}
del x['a']
print(x) #  out: {'b': 2}
print(x['a']) # error: KeyError: 'a'
```

del v[a:b]

Dies nennt tatsächlich `v.__delslice__(a, b)` .

Die Intention ist der oben beschriebenen ähnlich, jedoch mit Slices - Artikelbereiche statt eines einzelnen Artikels. Zum Beispiel:

```
x = [0, 1, 2, 3, 4]
del x[1:3]
print(x) #  out: [0, 3, 4]
```

Siehe auch Garbage Collection # Der Befehl del .

Lokaler vs Globaler Geltungsbereich

Was ist lokal und global?

Alle Python-Variabes, auf die an einer bestimmten Stelle im Code zugegriffen werden kann, liegen entweder im *lokalen* oder im *globalen Bereich* .

Die Erklärung ist, dass der lokale Bereich alle Variablen umfasst, die in der aktuellen Funktion definiert sind, und der globale Bereich enthält Variablen, die außerhalb der aktuellen Funktion definiert sind.

```
foo = 1  # global

def func():
    bar = 2  # local
    print(foo)  # prints variable foo from global scope
    print(bar)  # prints variable bar from local scope
```

Man kann prüfen, welche Variablen in welchem Umfang sind. Integrierte Funktionen `locals()` und

`globals()` geben den gesamten Gültigkeitsbereich als Wörterbücher zurück.

```
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
```

Was passiert bei Namenskonflikten?

```
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
```

Verwenden Sie das Schlüsselwort `global` um eine globale Variable zu ändern:

```
foo = 1

def func():
    global foo
    foo = 2  # this modifies the global foo, rather than creating a local variable
```

Der Geltungsbereich ist für den gesamten Körper der Funktion definiert!

Das bedeutet, dass eine Variable für eine Hälfte der Funktion niemals global und danach lokal ist oder umgekehrt.

```
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)
```

Ebenso das Gegenteil:

```
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
```

Funktionen innerhalb von Funktionen

In Funktionen können viele Ebenen von Funktionen verschachtelt sein, aber innerhalb einer Funktion gibt es nur einen lokalen Gültigkeitsbereich für diese Funktion und den globalen Gültigkeitsbereich. Es gibt keine Zwischenbereiche.

```
foo = 1

def f1():
    bar = 1

    def f2():
        baz = 2
        # here, foo is a global variable, baz is a local variable
        # bar is not in either scope
        print(locals().keys())  # ['baz']
        print('bar' in locals())  # False
        print('bar' in globals())  # False

    def f3():
        baz = 3
        print(bar)  # bar from f1 is referenced so it enters local scope of f3 (closure)
        print(locals().keys())  # ['bar', 'baz']
        print('bar' in locals())  # True
        print('bar' in globals())  # False

    def f4():
        bar = 4  # a new local bar which hides bar from local scope of f1
        baz = 4
        print(bar)
        print(locals().keys())  # ['bar', 'baz']
        print('bar' in locals())  # True
        print('bar' in globals())  # False
```

`global` vs. `nonlocal` (nur Python 3)

Diese beiden Schlüsselwörter werden verwendet, um auf Variablen zuzugreifen, die für die aktuellen Funktionen nicht lokal sind.

Das `global` Schlüsselwort gibt an, dass ein Name als globale Variable behandelt werden soll.

```
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
```

`nonlocal` (siehe Nichtlokale Variablen), verfügbar in Python 3, nimmt eine *lokale* Variable aus einem umschließenden Bereich in den lokalen Bereich der aktuellen Funktion.

Aus der Python-Dokumentation zu `nonlocal` :

> Die nichtlokale Anweisung bewirkt, dass die aufgelisteten Bezeichner auf zuvor gebundene Variablen im nächsten umgebenden Bereich mit Ausnahme von Globals verweisen.

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!
```

Variabler Geltungsbereich und Bindung online lesen:
https://riptutorial.com/de/python/topic/263/variabler-geltungsbereich-und-bindung

Kapitel 182: Vergleiche

Syntax

- ! = - Ist nicht gleich
- == - Ist gleich
- `>` - größer als
- `<` - weniger als
- `>=` - größer als oder gleich
- `<=` - kleiner oder gleich
- `is` - test, ob Objekte genau das gleiche Objekt sind
- ist nicht = test, wenn Objekte nicht exakt das gleiche Objekt sind

Parameter

Parameter	Einzelheiten
x	Erster zu vergleichender Artikel
y	Zweiter zu vergleichender Punkt

Examples

Größer oder kleiner als

```
x > y
x < y
```

Diese Operatoren vergleichen zwei Arten von Werten. Sie sind weniger als und größer als Operatoren. Bei Zahlen werden die numerischen Werte einfach verglichen, um zu sehen, welche größer sind:

```
12 > 4
# True
12 < 4
# False
1 < 4
# True
```

Bei Strings werden sie lexikographisch verglichen, was der alphabetischen Reihenfolge ähnelt, jedoch nicht ganz dieselbe ist.

```
"alpha" < "beta"
# True
"gamma" > "beta"
# True
"gamma" < "OMEGA"
# False
```

In diesen Vergleichen werden Kleinbuchstaben als 'größer als' Großbuchstaben betrachtet, weshalb `"gamma" < "OMEGA"` falsch ist. Wenn sie alle in Großbuchstaben sind, wird das erwartete Ergebnis der alphabetischen Reihenfolge zurückgegeben:

```
"GAMMA" < "OMEGA"
# True
```

Jeder Typ definiert seine Berechnung mit den Operatoren `<` und `>` anders. Sie sollten daher vor der Verwendung untersuchen, was die Operatoren für einen bestimmten Typ bedeuten.

Nicht gleichzusetzen mit

```
x != y
```

Dies gibt `True` wenn `x` und `y` nicht gleich sind, und ansonsten `False` .

```
12 != 1
# True
12 != '12'
# True
'12' != '12'
# False
```

Gleich

```
x == y
```

Dieser Ausdruck wertet aus, ob `x` und `y` den gleichen Wert haben, und gibt das Ergebnis als booleschen Wert zurück. Im Allgemeinen müssen Typ und Wert übereinstimmen, daher ist int `12` nicht identisch mit der Zeichenfolge `'12'` .

```
12 == 12
# True
12 == 1
# False
'12' == '12'
# True
'spam' == 'spam'
# True
'spam' == 'spam '
# False
```

```
'12' == 12
# False
```

Beachten Sie, dass jeder Typ eine Funktion definieren muss, mit der bewertet wird, ob zwei Werte gleich sind. Bei eingebauten Typen verhalten sich diese Funktionen wie erwartet und bewerten nur die Werte, die auf demselben Wert basieren. Benutzerdefinierte Typen können Gleichheitstests jedoch als beliebig definieren, z. B. immer `True` oder `False` .

Kettenvergleiche

Sie können mehrere Elemente mit mehreren Vergleichsoperatoren mit Kettenvergleich vergleichen. Zum Beispiel

```
x > y > z
```

ist nur eine Kurzform von:

```
x > y and y > z
```

Dies `True` nur dann `True` , wenn beide Vergleiche `True` .

Die allgemeine Form ist

```
a OP b OP c OP d ...
```

Dabei steht `OP` für eine der zahlreichen Vergleichsoperationen, die Sie verwenden können, und die Buchstaben repräsentieren beliebige gültige Ausdrücke.

> Beachten Sie, dass `0 != 1 != 0` zu `True` ausgewertet wird, obwohl `0 != 0` `False` . Im Gegensatz zur üblichen mathematischen Notation, in der `x != y != z` bedeutet, haben `x` , `y` und `z` unterschiedliche Werte. Chaining `==` Operationen hat die natürliche Bedeutung in den meisten Fällen, da die Gleichstellung im Allgemeinen transitiv ist.

Stil

Es gibt keine theoretische Begrenzung für die Anzahl der Elemente und Vergleichsoperationen, die Sie verwenden, solange Sie die richtige Syntax verwenden:

```
1 > -1 < 2 > 0.5 < 100 != 24
```

Das Obige gibt `True` wenn jeder Vergleich `True` . Die Verwendung von gewundenen Ketten ist jedoch kein guter Stil. Eine gute Verkettung ist "direktional", nicht komplizierter als

```
1 > x > -4 > y != 8
```

Nebenwirkungen

Sobald ein Vergleich `False` , wird der Ausdruck sofort zu `False` ausgewertet, wobei alle verbleibenden Vergleiche übersprungen werden.

Beachten Sie, dass der Ausdruck `exp` in `a > exp > b` nur einmal ausgewertet wird, im Fall von

```
a > exp and exp > b
```

`exp` wird zweimal berechnet, wenn `a > exp` wahr ist.

Vergleich von `is` vs` == `

Ein häufiger Fehler ist verwirrend die Gleichheit Vergleichsoperator `is` und `==` .

`a == b` vergleicht den Wert von `a` und `b` .

`a is b` vergleicht die *Identitäten* von `a` und `b` .

Um zu zeigen:

```
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 [!!]
```

Im Grunde genommen `is` kann der als Kurzform gedacht werden `id(a) == id(b)` .

Darüber hinaus gibt es Macken der Laufzeitumgebung, die die Dinge noch komplizierter machen. Kurze Zeichenfolgen und kleine Ganzzahlen geben im Vergleich zu `is True` , da die Python-Maschine versucht, für identische Objekte weniger Speicher zu verwenden.

```
a = 'short'
b = 'short'
c = 5
d = 5
a is b # True
c is d # True
```

Längere Zeichenfolgen und größere Ganzzahlen werden jedoch separat gespeichert.

```
a = 'not so short'
b = 'not so short'
```

```
c = 1000
d = 1000
a is b # False
c is d # False
```

Sie sollten verwenden `is` , auf die getestet `None` :

```
if myvar is not None:
    # not None
    pass
if myvar is None:
    # None
    pass
```

Eine Verwendung von `is` ist das Testen eines "Sentinels" (dh eines eindeutigen Objekts).

```
sentinel = object()
def myfunc(var=sentinel):
    if var is sentinel:
        # value wasn’t provided
        pass
    else:
        # value was provided
        pass
```

Objekte vergleichen

Um die Gleichheit von benutzerdefinierten Klassen zu vergleichen, können Sie `==` und `!=` `__eq__` `__ne__` Methoden `__eq__` und `__ne__` definieren. Sie können auch `__lt__` (`<`), `__le__` (`<=`), `__gt__` (`>`) und `__ge__` (`>`) `__ge__` . Beachten Sie, dass Sie nur zwei Vergleichsmethoden überschreiben müssen und Python den Rest erledigen kann (`==` ist dasselbe wie `not <` und `not >` usw.).

```
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
```

Beachten Sie, dass bei diesem einfachen Vergleich davon ausgegangen wird, dass `other` Objekte (das Objekt, mit dem verglichen wird) denselben Objekttyp haben. Beim Vergleich mit einem anderen Typ wird ein Fehler ausgegeben:

```
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'
```

Das Überprüfen von `isinstance()` oder ähnlichem hilft, dies zu verhindern (falls gewünscht).

Common Gotcha: Python erzwingt keine Eingabe

In vielen anderen Sprachen, wenn Sie Folgendes ausführen (Java-Beispiel)

```
if("asgdsrf" == 0) {
    //do stuff
}
```

... Sie erhalten einen Fehler. Sie können nicht einfach Strings mit ganzen Zahlen vergleichen. In Python ist dies eine absolut rechtliche Aussage - es wird nur zu `False` .

Eine häufige Angelegenheit ist die folgende

```
myVariable = "1"
if 1 == myVariable:
    #do stuff
```

Dieser Vergleich wird jedes Mal ohne Fehler als `False` ausgewertet, `False` möglicherweise ein Fehler ausgeblendet oder eine Bedingung gebrochen wird.

Vergleiche online lesen: https://riptutorial.com/de/python/topic/248/vergleiche

Kapitel 183: Verknüpfte Listen

Einführung

Eine verknüpfte Liste ist eine Sammlung von Knoten, die jeweils aus einer Referenz und einem Wert bestehen. Knoten werden mit ihren Referenzen zu einer Sequenz zusammengefügt. Verknüpfte Listen können verwendet werden, um komplexere Datenstrukturen wie Listen, Stacks, Warteschlangen und assoziative Arrays zu implementieren.

Examples

Beispiel für eine einzelne verknüpfte Liste

In diesem Beispiel wird eine verknüpfte Liste mit vielen der gleichen Methoden implementiert wie das integrierte Listenobjekt.

```
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()
```

Die Verwendung funktioniert ähnlich wie die der integrierten Liste.

```
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
```

Verknüpfte Listen online lesen: https://riptutorial.com/de/python/topic/9299/verknupfte-listen

Kapitel 184: Verknüpfter Listenknoten

Examples

Schreiben Sie einen einfachen Linked List-Knoten in Python

Eine verknüpfte Liste ist entweder:

- die leere Liste, dargestellt durch Keine oder
- ein Knoten, der ein Frachtobjekt und einen Verweis auf eine verknüpfte Liste enthält.

```
#! /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")
```

Verknüpfter Listenknoten online lesen: https://riptutorial.com/de/python/topic/6916/verknupfter-listenknoten

Kapitel 185: Versteckte Funktionen

Examples

Überladung des Bedieners

Alles in Python ist ein Objekt. Jedes Objekt verfügt über spezielle interne Methoden, mit denen es mit anderen Objekten interagieren kann. Im Allgemeinen folgen diese Methoden der Namenskonvention `__action__` . Zusammen wird dies als Python-Datenmodell bezeichnet .

Sie können *jede* dieser Methoden überladen. Dies wird häufig beim Überladen von Operatoren in Python verwendet. Im Folgenden finden Sie ein Beispiel für die Überladung von Operatoren mit dem Python-Datenmodell. Die `Vector` Klasse erstellt einen einfachen Vektor aus zwei Variablen. Wir werden eine geeignete Unterstützung für mathematische Operationen von zwei Vektoren hinzufügen, wobei Operatorüberladung verwendet wird.

```
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)>
```

Das obige Beispiel zeigt die Überladung grundlegender numerischer Operatoren. Eine umfassende Liste finden Sie hier .

Versteckte Funktionen online lesen: https://riptutorial.com/de/python/topic/946/versteckte-funktionen

Kapitel 186: Verteilung

Examples

py2app

Um das py2app-Framework verwenden zu können, müssen Sie es zuerst installieren. Öffnen Sie dazu das Terminal und geben Sie den folgenden Befehl ein:

```
sudo easy_install -U py2app
```

Sie können die Pakete auch wie `pip` installieren:

```
pip install py2app
```

Dann erstellen Sie die Setup-Datei für Ihr Python-Skript:

```
py2applet --make-setup MyApplication.py
```

Bearbeiten Sie die Einstellungen der Setup-Datei nach Ihren Wünschen. Dies ist die Standardeinstellung:

```
"""
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'],
)
```

Um eine Symboldatei hinzuzufügen (diese Datei muss die Erweiterung .icns haben) oder Bilder als Referenz in Ihre Anwendung aufzunehmen, ändern Sie Ihre Optionen wie folgt:

```
DATA_FILES = ['myInsertedImage.jpg']
OPTIONS = {'argv_emulation': True, 'iconfile': 'myCoolIcon.icns'}
```

Zum Schluss geben Sie dies im Terminal ein:

```
python setup.py py2app
```

Das Skript sollte ausgeführt werden und Sie finden Ihre fertige Anwendung im Ordner dist.

Verwenden Sie die folgenden Optionen, um weitere Anpassungen vorzunehmen:

```
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

Installiere cx_Freeze von hier aus

Entpacken Sie den Ordner und führen Sie die folgenden Befehle aus diesem Verzeichnis aus:

```
python setup.py build
sudo python setup.py install
```

Erstellen Sie ein neues Verzeichnis für Ihr Python-Skript und erstellen Sie eine Datei **"setup.py"** in demselben Verzeichnis mit folgendem Inhalt:

```
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)])
```

Führen Sie nun Ihre setup.py vom Terminal aus:

```
python setup.py bdist_mac
```

HINWEIS: Auf El Capitan muss dieser als root ausgeführt werden, wobei der SIP-Modus deaktiviert ist.

Verteilung online lesen: https://riptutorial.com/de/python/topic/2026/verteilung

Kapitel 187: Vertiefung

Examples

Einrückungsfehler

Der Abstand sollte überall gleich und gleichmäßig sein. Ein falscher Einzug kann einen `IndentationError` verursachen oder bewirken, dass das Programm etwas Unerwartetes tut. Im folgenden Beispiel wird ein `IndentationError` :

```
a = 7
if a > 5:
  print "foo"
else:
  print "bar"
 print "done"
```

Wenn die Zeile nach einem Doppelpunkt nicht eingerückt ist, wird auch ein `IndentationError` :

```
if True:
print "true"
```

Wenn Sie Einrückungen hinzufügen, an denen sie nicht gehören, wird ein `IndentationError` :

```
if  True:
    a = 6
        b = 5
```

Wenn Sie vergessen, den Einzug aufzuheben, kann die Funktionalität verloren gehen. In diesem Beispiel wird statt des erwarteten `False` `None` zurückgegeben:

```
def isEven(a):
    if a%2 ==0:
        return True
        #this next line should be even with the if
        return False
print isEven(7)
```

Einfaches Beispiel

Für Python hat Guido van Rossum die Gruppierung von Aussagen auf Einrückung vorgenommen. Die Gründe dafür werden im ersten Abschnitt der "FAQ zu Design und Historie Python" erläutert. Doppelpunkte, `:` , werden verwendet , um einen eingerückten Codeblock zu deklarieren , wie im folgenden Beispiel:

```
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])
```

Leerzeichen oder Tabs?

Die empfohlene Einrückung besteht aus 4 Leerzeichen, aber Tabulatoren oder Leerzeichen können verwendet werden, solange sie konsistent sind. ***Mischen Sie keine Registerkarten und Leerzeichen in Python,*** da dies einen Fehler in Python 3 und in Python 2 Fehler verursachen kann.

Wie wird Einrückung analysiert?

Leerzeichen werden vom lexikalischen Analysator verarbeitet, bevor sie analysiert werden.

Der lexikalische Analysator verwendet einen Stapel zum Speichern von Eindruckebenen. Zu Beginn enthält der Stack nur den Wert 0, also die Position ganz links. Immer wenn ein verschachtelter Block beginnt, wird die neue Einrückungsebene in den Stapel verschoben, und ein Token "INDENT" wird in den Tokenstrom eingefügt, der an den Parser übergeben wird. Es kann niemals mehr als ein "INDENT" -Token in einer Zeile sein (`IndentationError`).

Wenn eine Zeile mit einer kleineren Einrückungsebene gefunden wird, werden die Werte aus dem Stapel entfernt, bis ein Wert oben angezeigt wird, der der neuen Einzugsebene entspricht (wenn keine gefunden wird, tritt ein Syntaxfehler auf). Für jeden gepoppten Wert wird ein "DEDENT" -Token generiert. Natürlich kann es mehrere "DEDENT" -Marker in einer Reihe geben.

Der lexikalische Analysator überspringt leere Zeilen (Zeilen, die nur Leerzeichen und möglicherweise Kommentare enthalten) und generiert niemals "INDENT" - oder "DEDENT" -Token für sie.

Am Ende des Quellcodes werden für jede auf dem Stapel verbleibende Einzugsebene "DEDENT" -Token generiert, bis nur noch die 0 übrig ist.

Zum Beispiel:

```
if foo:
    if bar:
        x = 42
else:
    print foo
```

wird analysiert als:

```
<if> <foo> <:>                    [0]
<INDENT> <if> <bar> <:>           [0, 4]
<INDENT> <x> <=> <42>             [0, 4, 8]
<DEDENT> <DEDENT> <else> <:>      [0]
<INDENT> <print> <foo>            [0, 2]
<DEDENT>
```

Der Parser behandelt dann die Token "INDENT" und "DEDENT" als Blockbegrenzer.

Vertiefung online lesen: https://riptutorial.com/de/python/topic/2597/vertiefung

Kapitel 188: Verwenden von Schleifen innerhalb von Funktionen

Einführung

In Python wird die Funktion zurückgegeben, sobald die Ausführung auf "return" stößt.

Examples

Anweisung innerhalb einer Schleife in einer Funktion zurückgeben

In diesem Beispiel kehrt die Funktion zurück, sobald der Wert var den Wert 1 hat

```
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])
```

Ausgabe

```
Got value 5
Still looping
Got value 3
Still looping
Got value 1
>>>> Got 1
```

Verwenden von Schleifen innerhalb von Funktionen online lesen:
https://riptutorial.com/de/python/topic/10883/verwenden-von-schleifen-innerhalb-von-funktionen

Kapitel 189: Verwendung des "pip" -Moduls: PyPI Package Manager

Einführung

Manchmal müssen Sie den pip package manager in python verwenden, z. Wenn einige Importe `ImportError` und Sie die Ausnahme behandeln möchten. Wenn Sie unter Windows die `Python_root/Scripts/pip.exe` entpacken, wird die Datei `__main__.py` gespeichert, in die die `main` aus dem `pip` Paket importiert wird. Das bedeutet, dass das Pip-Paket immer dann verwendet wird, wenn Sie eine ausführbare Pip-Datei verwenden. Zur Verwendung von pip als ausführbare Datei siehe: pip: PyPI Package Manager

Syntax

- pip. <Funktion | Attribut | Klasse> wobei Funktion eine der folgenden ist:
  - Autovervollständigung ()
    - Befehls- und Optionsvervollständigung für den Hauptoptionsparser (und die Optionen) und seine Unterbefehle (und Optionen). Aktivieren Sie diese Option, indem Sie eines der Completion-Shell-Skripts (bash, zsh oder fish) beziehen.
  - check_isolated (args)
    - param args {list}
    - gibt {boolean} zurück
  - create_main_parser ()
    - gibt ein {pip.baseparser.ConfigOptionParser-Objekt} zurück
  - main (args = keine)
    - param args {list}
    - Rückgabe {Ganzzahl} Wenn nicht fehlgeschlagen als 0 zurückgegeben wird
  - parseopts (args)
    - param args {list}
  - get_installed_distributions ()
    - gibt {list} zurück
  - get_similar_commands (name)
    - Befehlsname automatisch korrigiert
    - Parametername {Zeichenfolge}
    - gibt {boolean} zurück
  - get_summaries (order = True)
    - Ergibt sortierte (Befehlsname, Befehlszusammenfassung) Tupel.
  - get_prog ()
    - gibt {String} zurück
  - dist_is_editable (dist)
    - Ist die Distribution eine editierbare Installation?
    - param dist {object}
    - gibt {boolean} zurück

- command_dict
  - Attribut {Wörterbuch}

Examples

Beispiel für die Verwendung von Befehlen

```
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])
```

`pip.main(['freeze'])` nur erforderliche Parameter obligatorisch sind, sind sowohl `pip.main(['freeze'])` als auch `pip.main(['freeze', '', ''])` akzeptabel.

Batch-Installation

Es ist möglich, viele Paketnamen in einem Aufruf zu übergeben. Wenn jedoch eine Installation / ein Upgrade fehlschlägt, wird der gesamte Installationsvorgang angehalten und mit dem Status '1' beendet.

```
import pip

installed = pip.get_installed_distributions()
list = []
for i in installed:
    list.append(i.key)

pip.main(['install']+list+['--upgrade'])
```

Wenn Sie nicht anhalten möchten, wenn einige Installationen fehlschlagen, rufen Sie die Installation in einer Schleife auf.

```
for i in installed:
        pip.main(['install']+i.key+['--upgrade'])
```

Behandlung der ImportError-Ausnahme

Wenn Sie eine Python-Datei als Modul verwenden, müssen Sie nicht immer überprüfen, ob das Paket installiert ist, es ist jedoch für Skripte nützlich.

```
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
```

Installation erzwingen

Viele Pakete, zum Beispiel in Version 3.4, würden unter 3.6 problemlos laufen, aber wenn es keine Distributionen für eine bestimmte Plattform gibt, können sie nicht installiert werden, aber es gibt eine Problemumgehung. In WHL-Dateien (bekannt als Räder) entscheiden Namenskonventionen, ob Sie Pakete auf der angegebenen Plattform installieren können. Z.B. `scikit_learn-0.18.1-cp36-cp36m-win_amd64.whl` [Paketname] - [Version] - [Python-Interpreter] - [Python-Interpreter] - [Betriebssystem] .whl. Wenn der Name der Raddatei geändert wird und die Plattform nicht übereinstimmt, versucht pip, das Paket zu installieren, auch wenn die Plattform- oder Python-Version nicht übereinstimmt. Das Entfernen der Plattform oder des Interpreters aus dem Namen führt zu einem Fehler in der neuesten Version des Pip-Moduls. `kjhfkjdf.whl is not a valid wheel filename.` .

Alternativ kann die .whl-Datei mit einem Archivierer als 7-zip entpackt werden. - Normalerweise enthält es Meta-Ordner für die Verteilung und Ordner mit Quelldateien. Diese Quelldateien können einfach in das `site-packges` Verzeichnis entpackt werden, es sei denn, dieses Rad enthält ein Installationsskript. Wenn ja, muss es zuerst ausgeführt werden.

Verwendung des "pip" -Moduls: PyPI Package Manager online lesen: https://riptutorial.com/de/python/topic/10730/verwendung-des--pip---moduls--pypi-package-manager

Kapitel 190: Virtuelle Python-Umgebung - virtualenv

Einführung

Eine virtuelle Umgebung ("virtualenv") ist ein Werkzeug zum Erstellen isolierter Python-Umgebungen. Es hält die Abhängigkeiten, die von verschiedenen Projekten benötigt werden, an unterschiedlichen Orten, indem für sie eine virtuelle Python-Umgebung erstellt wird. Es löst das Problem „Projekt A hängt von Version 2.xxx ab, Projekt B benötigt jedoch das 2.xxx-Dilemma“ und hält Ihr globales Site-Packages-Verzeichnis sauber und verwaltbar.

"virtualenv" erstellt einen Ordner, der alle erforderlichen Bibliotheken und Ablagen enthält, um die Pakete zu verwenden, die ein Python-Projekt benötigt.

Examples

Installation

Installieren Sie virtualenv über pip / (apt-get):

```
pip install virtualenv
```

ODER

```
apt-get install python-virtualenv
```

Hinweis: Wenn Sie Probleme mit der Berechtigung erhalten, verwenden Sie sudo.

Verwendungszweck

```
$ cd test_proj
```

Erstellen Sie eine virtuelle Umgebung:

```
$ virtualenv test_proj
```

Um die virtuelle Umgebung nutzen zu können, muss sie aktiviert werden:

```
$ source test_project/bin/activate
```

Um Ihre Virtualenv zu beenden, geben Sie einfach "disable" ein:

```
$ deactivate
```

Installieren Sie ein Paket in Ihrer Virtualenv

Wenn Sie das bin-Verzeichnis in Ihrer virtualenv betrachten, sehen Sie easy_install, das geändert wurde, um Eier und Pakete in das Site-Packages-Verzeichnis der virtualenv zu legen. So installieren Sie eine App in Ihrer virtuellen Umgebung:

```
$ source test_project/bin/activate
$ pip install flask
```

Momentan müssen Sie sudo nicht verwenden, da die Dateien alle im lokalen Verzeichnis der virtualenv-Site-Packages installiert werden. Dies wurde als eigenes Benutzerkonto erstellt.

Andere nützliche Virtualenv-Befehle

lsvirtualenv : **Listet** alle Umgebungen auf.

cdvirtualenv : Navigieren Sie in das Verzeichnis der aktuell aktivierten virtuellen Umgebung, sodass Sie beispielsweise die Site-Pakete durchsuchen können.

cdsitepackages : Wie oben, jedoch direkt im Site-Packages-Verzeichnis.

lssitepackages : Zeigt den Inhalt des Site-Packages-Verzeichnisses an.

Virtuelle Python-Umgebung - virtualenv online lesen:
https://riptutorial.com/de/python/topic/9782/virtuelle-python-umgebung---virtualenv

Kapitel 191: virtuelle Umgebung mit Virtualenvwrapper

Einführung

Angenommen, Sie müssen an drei verschiedenen Projekten arbeiten. Projekt A, Projekt B und Projekt C. Projekt A und Projekt B benötigen Python 3 und einige erforderliche Bibliotheken. Für Projekt C benötigen Sie jedoch Python 2.7 und abhängige Bibliotheken.

Daher empfiehlt es sich, diese Projektumgebungen voneinander zu trennen. Um eine virtuelle Umgebung zu erstellen, können Sie folgende Technik verwenden:

Virtualenv, Virtualenvwrapper und Conda

Wir haben zwar mehrere Optionen für die virtuelle Umgebung, aber der Virtualenvwrapper wird am meisten empfohlen.

Examples

Erstellen Sie eine virtuelle Umgebung mit Virtualenvwrapper

Angenommen, Sie müssen an drei verschiedenen Projekten arbeiten. Projekt A, Projekt B und Projekt C. Projekt A und Projekt B benötigen Python 3 und einige erforderliche Bibliotheken. Für Projekt C benötigen Sie jedoch Python 2.7 und abhängige Bibliotheken.

Daher empfiehlt es sich, diese Projektumgebungen voneinander zu trennen. Um eine virtuelle Umgebung zu erstellen, können Sie folgende Technik verwenden:

Virtualenv, Virtualenvwrapper und Conda

Wir haben zwar mehrere Optionen für die virtuelle Umgebung, aber am meisten wird der Virtualenvwrapper empfohlen.

Wir haben zwar mehrere Optionen für die virtuelle Umgebung, aber ich bevorzuge einen virtualenvwrapper, weil er mehr Möglichkeiten hat als andere.

```
$ 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
```

Jetzt können wir einige Software in die Umgebung installieren.

```
(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
```

Wir können das neue Paket mit lssitepackages sehen:

```
(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
```

Wir können mehrere virtuelle Umgebungen erstellen, wenn wir möchten.

Wechseln zwischen Umgebungen mit Workon:

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

Um die virtualenv zu verlassen

```
$ deactivate
```

virtuelle Umgebung mit Virtualenvwrapper online lesen:
https://riptutorial.com/de/python/topic/9983/virtuelle-umgebung-mit-virtualenvwrapper

Kapitel 192: Virtuelle Umgebungen

Einführung

Eine virtuelle Umgebung ist ein Werkzeug, um die Abhängigkeiten, die für verschiedene Projekte erforderlich sind, an unterschiedlichen Orten zu halten, indem virtuelle Python-Umgebungen für sie erstellt werden. Es löst das Problem "Project X hängt von Version 1.x ab, aber Project Y benötigt 4.x" und hält Ihr globales Site-Packages-Verzeichnis sauber und handhabbar.

Dadurch können Sie Ihre Umgebungen für verschiedene Projekte voneinander und von Ihren Systembibliotheken isolieren.

Bemerkungen

Virtuelle Umgebungen sind so nützlich, dass sie wahrscheinlich für jedes Projekt verwendet werden sollten. In virtuellen Umgebungen können Sie insbesondere:

1. Verwalten Sie Abhängigkeiten ohne Rootzugriff
2. Installieren Sie verschiedene Versionen derselben Abhängigkeit, beispielsweise wenn Sie an verschiedenen Projekten mit unterschiedlichen Anforderungen arbeiten
3. Arbeiten Sie mit verschiedenen Python-Versionen

Examples

Erstellen und Verwenden einer virtuellen Umgebung

`virtualenv` ist ein Tool zum Erstellen von isolierten Python-Umgebungen. Dieses Programm erstellt einen Ordner, der alle erforderlichen ausführbaren Dateien enthält, um die Pakete verwenden zu können, die ein Python-Projekt benötigt.

Das Virtualenv-Tool installieren

Dies ist nur einmal erforderlich. Das `virtualenv` Programm ist möglicherweise in Ihrer Distribution verfügbar. Bei Debian-ähnlichen Distributionen heißt das Paket `python-virtualenv` oder `python3-virtualenv` .

Alternativ können Sie `virtualenv` mit pip installieren:

```
$ pip install virtualenv
```

Erstellen einer neuen virtuellen Umgebung

Dies ist nur einmal pro Projekt erforderlich. Wenn Sie ein Projekt starten, für das Sie Abhängigkeiten isolieren möchten, können Sie eine neue virtuelle Umgebung für dieses Projekt einrichten:

```
$ virtualenv foo
```

Dadurch wird ein `foo` Ordner erstellt, der Tooling-Skripts und eine Kopie der `python` Binärdatei selbst enthält. Der Name des Ordners ist nicht relevant. Nachdem die virtuelle Umgebung erstellt wurde, ist sie in sich abgeschlossen und erfordert keine weitere Bearbeitung mit dem `virtualenv` Tool. Sie können jetzt die virtuelle Umgebung verwenden.

Aktivieren einer vorhandenen virtuellen Umgebung

Um eine virtuelle Umgebung zu *aktivieren* , ist etwas Shell-Zauber erforderlich, sodass Ihr Python derjenige ist, der sich in `foo` anstelle des Systems befindet. Dies ist der Zweck der `activate` , die Sie in Ihre aktuelle Shell eingeben müssen:

```
$ source foo/bin/activate
```

Windows-Benutzer sollten Folgendes eingeben:

```
$ foo\Scripts\activate.bat
```

Nachdem eine virtuelle Umgebung aktiviert wurde, sind die `python` und `pip` Binärdateien und alle von Drittanbietermodulen installierten Skripts die in `foo` . Insbesondere werden alle mit `pip` installierten Module in der virtuellen Umgebung bereitgestellt, sodass eine geschlossene Entwicklungsumgebung möglich ist. Durch die Aktivierung der virtuellen Umgebung sollte Ihrer Eingabeaufforderung auch ein Präfix hinzugefügt werden, wie in den folgenden Befehlen dargestellt.

```
# Installs 'requests' to foo only, not globally
(foo)$ pip install requests
```

Abhängigkeiten speichern und wiederherstellen

Um die über `pip` installierten Module zu speichern, können Sie alle diese Module (und die entsprechenden Versionen) mit dem Befehl `freeze` in einer Textdatei `freeze` . Dies ermöglicht anderen Benutzern, die für die Anwendung erforderlichen Python-Module mithilfe des Installationsbefehls schnell zu installieren. Der übliche Name für eine solche Datei lautet `requirements.txt` :

```
(foo)$ pip freeze > requirements.txt
(foo)$ pip install -r requirements.txt
```

Beachten Sie, dass `freeze` alle Module auflistet, einschließlich der von den von Ihnen installierten Top-Level-Modulen erforderlichen transitiven Abhängigkeiten. Daher können Sie die Datei „ `requirements.txt` möglicherweise von Hand erstellen , indem Sie nur die Module der obersten Ebene einsetzen, die Sie benötigen.

Beenden einer virtuellen Umgebung

Wenn Sie mit der virtuellen Umgebung fertig sind, können Sie sie deaktivieren, um zu Ihrer normalen Shell zurückzukehren:

```
(foo)$ deactivate
```

Verwenden einer virtuellen Umgebung in einem gemeinsam genutzten Host

Manchmal ist es nicht möglich, `$ source bin/activate` virtualenv zu `$ source bin/activate` , z. B. wenn Sie mod_wsgi auf einem gemeinsam genutzten Host verwenden oder wenn Sie keinen Zugriff auf ein Dateisystem haben, z. In diesen Fällen können Sie die installierten Bibliotheken in Ihrer lokalen virtualenv bereitstellen und Ihren `sys.path` .

Zum Glück wird Virtualenv mit einem Skript ausgeliefert, das sowohl Ihren `sys.path` als auch Ihren `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))
```

Sie sollten diese Zeilen am Anfang der Datei anhängen, die Ihr Server ausführt.

Dadurch wird die Datei `bin/activate_this.py` `virtualenv` , die von `virtualenv` im selben `virtualenv` erstellt wurde wie Sie. `virtualenv` Sie Ihre `lib/python2.7/site-packages` in `sys.path`

Wenn Sie schauen , um das verwenden `activate_this.py` Skript, erinnert mit einzusetzen, zumindest, das `bin` und `lib/python2.7/site-packages` Verzeichnisse und dessen Inhalt.

Python 3.x 3.3

Eingebaute virtuelle Umgebungen

Ab Python 3.3 erstellt das venv-Modul virtuelle Umgebungen. Der `pyvenv` Befehl muss nicht separat installiert werden:

```
$ pyvenv foo
$ source foo/bin/activate
```

oder

```
$ python3 -m venv foo
$ source foo/bin/activate
```

Pakete in einer virtuellen Umgebung installieren

Sobald Ihre virtuelle Umgebung aktiviert wurde, dass Sie jedes Paket jetzt installieren im installiert werden `virtualenv` & nicht global. Daher können neue Pakete ohne Root-Berechtigungen sein.

Um zu überprüfen, ob die Pakete in der `virtualenv` installiert werden, führen Sie den folgenden Befehl aus, um den Pfad der verwendeten ausführbaren Datei zu überprüfen:

```
(<Virtualenv Name) $ which python
/<Virtualenv Directory>/bin/python

(Virtualenv Name) $ which pip
/<Virtualenv Directory>/bin/pip
```

Jedes Paket, das dann mit pip installiert wird, wird in der `virtualenv` selbst im folgenden Verzeichnis installiert:

```
/<Virtualenv Directory>/lib/python2.7/site-packages/
```

Alternativ können Sie eine Datei erstellen, die die benötigten Pakete auflistet.

Anforderungen.txt :

```
requests==2.10.0
```

Ausführen:

```
# Install packages from requirements.txt
pip install -r requirements.txt
```

wird Version 2.10.0 der Paket installieren `requests` .

Sie können auch eine Liste der Pakete und ihrer aktuell in der aktiven virtuellen Umgebung installierten Versionen abrufen:

```
# 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
```

Alternativ müssen Sie Ihre virtuelle Umgebung nicht jedes Mal aktivieren, wenn Sie ein Paket installieren müssen. Sie können die ausführbare Pip-Datei direkt im virtuellen Umgebungsverzeichnis verwenden, um Pakete zu installieren.

```
$ /<Virtualenv Directory>/bin/pip install requests
```

Weitere Informationen zur Verwendung von pip finden Sie im PIP-Thema .

Da Sie ohne Root in einer virtuellen Umgebung installieren, ist dies *keine* globale Installation auf dem gesamten System. Das installierte Paket ist nur in der aktuellen virtuellen Umgebung verfügbar.

Erstellen einer virtuellen Umgebung für eine andere Python-Version

Wenn `python` und `python3` beide installiert sind, ist es möglich, eine virtuelle Umgebung für Python 3 zu erstellen, auch wenn `python3` nicht der Standard-Python ist:

```
virtualenv -p python3 foo
```

oder

```
virtualenv --python=python3 foo
```

oder

```
python3 -m venv foo
```

oder

```
pyvenv foo
```

Tatsächlich können Sie eine virtuelle Umgebung auf der Grundlage einer beliebigen Version des Arbeitspythons Ihres Systems erstellen. Sie können verschiedene Arbeitspythons unter `/usr/bin/` oder `/usr/local/bin/` (unter Linux) `/Library/Frameworks/Python.framework/Versions/XX/bin/` ODER in `/Library/Frameworks/Python.framework/Versions/XX/bin/` (OSX) und dann herausfinden benennen und verwenden Sie dies im Flag `--python` oder `-p` wenn Sie eine virtuelle Umgebung erstellen.

Verwalten mehrerer virtueller Umgebungen mit virtualenvwrapper

Das Dienstprogramm " `virtualenvwrapper` vereinfacht die Arbeit mit virtuellen Umgebungen und ist besonders nützlich, wenn Sie mit vielen virtuellen Umgebungen / Projekten arbeiten.

Anstatt sich mit den Verzeichnissen der virtuellen Umgebung selbst zu befassen, verwaltet `virtualenvwrapper` diese für Sie, indem alle virtuellen Umgebungen in einem zentralen Verzeichnis (standardmäßig `~/.virtualenvs`) `~/.virtualenvs` werden.

Installation

Installieren Sie den `virtualenvwrapper` mit dem Paketmanager Ihres Systems.

Debian / Ubuntu-basiert:

```
apt-get install virtualenvwrapper
```

Fedora / CentOS / RHEL:

```
yum install python-virtualenvrwapper
```

Arch Linux:

```
pacman -S python-virtualenvwrapper
```

Oder installieren Sie es von PyPI mit `pip` :

```
pip install virtualenvwrapper
```

Unter Windows können `virtualenvwrapper-powershell` stattdessen entweder `virtualenvwrapper-win` oder `virtualenvwrapper-powershell` verwenden.

Verwendungszweck

Virtuelle Umgebungen werden mit `mkvirtualenv` . Alle Argumente des ursprünglichen `virtualenv` werden ebenfalls akzeptiert.

```
mkvirtualenv my-project
```

oder z

```
mkvirtualenv --system-site-packages my-project
```

Die neue virtuelle Umgebung wird automatisch aktiviert. In neuen Shells können Sie die virtuelle Umgebung mit `workon`

```
workon my-project
```

Der Vorteil des `workon` Befehls gegenüber dem traditionellen . `path/to/my-env/bin/activate workon` ist, dass der `workon` Befehl in einem beliebigen Verzeichnis funktioniert. Sie müssen sich nicht merken, in welchem Verzeichnis die bestimmte virtuelle Umgebung Ihres Projekts gespeichert ist.

Projektverzeichnisse

Sie können sogar während der Erstellung der virtuellen Umgebung ein Projektverzeichnis mit der Option `-a` oder später mit dem Befehl `setvirtualenvproject` .

```
mkvirtualenv -a /path/to/my-project my-project
```

oder

```
workon my-project
cd /path/to/my-project
setvirtualenvproject
```

Wenn Sie ein Projekt `workon` , wechselt der `workon` Befehl automatisch zum Projekt und aktiviert den Befehl `cdproject` , mit dem Sie in das Projektverzeichnis wechseln können.

Um eine Liste aller von virtualenvwrapper verwalteten virtualenvs anzuzeigen, verwenden Sie `lsvirtualenv` .

Um ein Virtualenv zu entfernen, verwenden Sie `rmvirtualenv` :

```
rmvirtualenv my-project
```

Jede von virtualenvwrapper verwaltete virtualenv enthält 4 leere Bash-Skripts: `preactivate` , `postactivate` , `predeactivate` und `postdeactivate` . Diese dienen als Hooks zur Ausführung von Bash-Befehlen an bestimmten Punkten im Lebenszyklus der virtuellen Umgebung. Beispielsweise werden alle Befehle im `postactivate` Skript unmittelbar nach der Aktivierung der virtualenv ausgeführt. Dies ist ein guter Ort, um spezielle Umgebungsvariablen, Aliasnamen oder andere relevante Werte festzulegen. Alle 4 Skripte befinden sich unter `.virtualenvs/<virtualenv_name>/bin/` .

Weitere Informationen finden Sie in der Dokumentation zum Virtualenvwrapper .

Ermitteln, welche virtuelle Umgebung Sie verwenden

Wenn Sie die Standard- `bash` Eingabeaufforderung unter Linux verwenden, sollte der Name der virtuellen Umgebung am Anfang Ihrer Eingabeaufforderung angezeigt werden.

```
(my-project-env) user@hostname:~$ which python
/home/user/my-project-env/bin/python
```

Festlegen einer bestimmten Python-Version zur Verwendung in Skripten unter Unix / Linux

Um festzulegen, welche Python-Version die Linux-Shell verwenden soll, kann die erste Zeile der Python-Skripts eine Shebang-Zeile sein, die mit `#!` beginnt `#!` :

```
#!/usr/bin/python
```

Wenn Sie sich in einer virtuellen Umgebung befinden, verwendet `python myscript.py` Python aus Ihrer virtuellen Umgebung, aber `./myscript.py` verwendet den Python-Interpreter in `#!` Linie. Um sicherzustellen, dass der Python der virtuellen Umgebung verwendet wird, ändern Sie die erste Zeile in:

```
#!/usr/bin/env python
```

Vergessen Sie nach der Angabe der Shebang-Linie nicht, dem Skript Ausführungsberechtigungen zu erteilen, indem Sie Folgendes tun:

```
chmod +x myscript.py
```

`./myscript.py` können Sie das Skript ausführen, indem Sie `./myscript.py` (oder den absoluten Pfad zum Skript) anstelle von `python myscript.py` oder `python3 myscript.py` .

Verwendung von Virtualenv mit Fischmuschel

Fish Shell ist freundlicher, aber Sie können Probleme mit `virtualenv` oder `virtualenvwrapper` . Alternativ `virtualfish` es `virtualfish` für die Rettung. Folgen Sie einfach der nachstehenden Reihenfolge, um die Fish Shell mit virtualenv zu verwenden.

- Installieren Sie Virtualfish im globalen Bereich

  ```
  sudo pip install virtualfish
  ```

- Laden Sie das Python-Modul virtualfish während des Starts der Fish Shell

  ```
  $ echo "eval (python -m virtualfish)" > ~/.config/fish/config.fish
  ```

- Bearbeiten Sie diese Funktion `fish_prompt` von `$ funced fish_prompt --editor vim` fügen Sie die folgenden Zeilen hinzu und schließen Sie den vim-Editor

  ```
  if set -q VIRTUAL_ENV
      echo -n -s (set_color -b blue white) "(" (basename "$VIRTUAL_ENV") ")" (set_color
  normal) " "
  end
  ```

  Hinweis: Wenn Sie mit vim nicht vertraut sind, geben Sie einfach Ihren bevorzugten Editor wie diesen `$ funced fish_prompt --editor nano` oder `$ funced fish_prompt --editor gedit`

- Speichern Sie die Änderungen mit `funcsave`

  ```
  funcsave fish_prompt
  ```

- Um eine neue virtuelle Umgebung zu erstellen, verwenden Sie `vf new`

  ```
  vf new my_new_env # Make sure $HOME/.virtualenv exists
  ```

- Wenn Sie eine neue python3-Umgebung erstellen möchten, geben Sie diese über das Flag `-p`

```
vf new -p python3 my_new_env
```

- Um zwischen virtuellen Umgebungen umzuschalten, verwenden Sie `vf deactivate` disable und `vf activate another_env`

Offizielle Links:

- https://github.com/adambrenecki/virtualfish
- http://virtualfish.readthedocs.io/de/latest/

Erstellen von virtuellen Umgebungen mit Anaconda

Eine leistungsstarke Alternative zu `virtualenv` ist Anaconda - ein plattformübergreifender, `pip` ähnlicher Paketmanager mit Funktionen zum schnellen Erstellen und Entfernen von virtuellen Umgebungen. Nach der Installation von Anaconda folgen hier einige Befehle, um loszulegen:

Erstellen Sie eine Umgebung

```
conda create --name <envname> python=<version>
```

Dabei ist `<envname>` in einem beliebigen Namen für Ihre virtuelle Umgebung und `<version>` eine bestimmte Python-Version, die Sie `<envname>` möchten.

Aktivieren und deaktivieren Sie Ihre Umgebung

```
# Linux, Mac
source activate <envname>
source deactivate
```

oder

```
# Windows
activate <envname>
deactivate
```

Zeigen Sie eine Liste der erstellten Umgebungen an

```
conda env list
```

Entfernen Sie eine Umgebung

```
conda env remove -n <envname>
```

Weitere Befehle und Funktionen finden Sie in der offiziellen Conda-Dokumentation .

Überprüfen, ob in einer virtuellen Umgebung ausgeführt wird

Manchmal zeigt die Shell-Eingabeaufforderung nicht den Namen der virtuellen Umgebung an, und Sie möchten sichergehen, ob Sie sich in einer virtuellen Umgebung befinden oder nicht.

Führen Sie den Python-Interpreter aus und versuchen Sie Folgendes:

```
import sys
sys.prefix
sys.real_prefix
```

- Außerhalb einer virtuellen Umgebung `sys.prefix` auf die System-Python-Installation, und `sys.real_prefix` ist nicht definiert.
- In einer virtuellen Umgebung `sys.prefix` auf die Installation der virtuellen Umgebung von python und `sys.real_prefix` auf die Installation von system python.

Für virtuelle Umgebungen, die mit dem Standardbibliothek venv-Modul erstellt wurden, gibt es keinen `sys.real_prefix` . Prüfen `sys.base_prefix` stattdessen, ob `sys.base_prefix` mit `sys.prefix` `sys.base_prefix` ist.

Virtuelle Umgebungen online lesen: https://riptutorial.com/de/python/topic/868/virtuelle-umgebungen

Kapitel 193: Vorlagen in Python

Examples

Einfaches Datenausgabeprogramm mit Vorlage

```
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))
```

Ausgabe:

```
Simon bought 2 candy for 8 dollar
```

Vorlagen unterstützen $ -basierte Substitutionen anstelle von% -basierten Substitutionen. **Substitute** (Mapping, Schlüsselwörter) führt die Ersetzung der Vorlage durch und gibt eine neue Zeichenfolge zurück.

Bei der Zuordnung handelt es sich um ein wörterbuchähnliches Objekt, dessen Schlüssel mit den Platzhaltern der Vorlage übereinstimmen. In diesem Beispiel sind Preis und Menge Platzhalter. Schlüsselwortargumente können auch als Platzhalter verwendet werden. Platzhalter aus Schlüsselwörtern haben Vorrang, wenn beide vorhanden sind.

Trennzeichen ändern

Sie können das Trennzeichen "$" in ein anderes Zeichen ändern. Das folgende Beispiel:

```
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))
```

Sie können de docs lesen hier

Vorlagen in Python online lesen: https://riptutorial.com/de/python/topic/6029/vorlagen-in-python

Kapitel 194: Vorrang des Bedieners

Einführung

Python-Operatoren haben eine bestimmte **Rangfolge** , die bestimmt, welche Operatoren in einem möglicherweise mehrdeutigen Ausdruck zuerst ausgewertet werden. In dem Ausdruck 3 * 2 + 7 wird zum Beispiel zuerst 3 mit 2 multipliziert, und das Ergebnis wird zu 7 addiert, wodurch sich 13 ergibt. Der Ausdruck wird nicht umgekehrt ausgewertet, da * eine höhere Priorität als + hat.

Nachfolgend finden Sie eine Liste der Operatoren nach Prioritäten und eine kurze Beschreibung ihrer (normalerweise) Vorgänge.

Bemerkungen

Aus der Python-Dokumentation:

> In der folgenden Tabelle werden die Operatorvorrangstufen in Python zusammengefasst, von der niedrigsten Priorität (geringste Bindung) bis zur höchsten Priorität (meist Bindung). Operatoren in derselben Box haben die gleiche Priorität. Wenn die Syntax nicht explizit angegeben ist, sind Operatoren binär. Operatoren in derselben Boxgruppe von links nach rechts (außer für Vergleiche, einschließlich Tests, die alle dieselbe Priorität und Kette von links nach rechts haben und die Potenzierung von rechts nach links gruppiert)

Operator	Beschreibung
Lambda	Lambda-Ausdruck
ansonsten	Bedingter Ausdruck
oder	Boolesches ODER
und	Boolean AND
nicht x	Boolean NICHT
in, nicht in, ist, ist nicht, <, <=,>,> =, <>,! =, ==	Vergleiche, einschließlich Mitgliedschaftstests und Identitätstests
|	Bitweises ODER
^	Bitweises XOR
&	Bitweises AND
<<, >>	Verschiebungen

Operator	Beschreibung
+, -	Addition und Subtraktion
*, /, //,%	Multiplikation, Division, Rest [8]
+ x, -x, ~ x	Positiv, negativ, bitweise NICHT
**	Potenzierung [9]
x [index], x [index: index], x (Argumente ...), x.attribute	Abonnement, Slicing, Aufruf, Attributreferenz
(Ausdrücke ...), [Ausdrücke ...], {Schlüssel: Wert ...}, Ausdrücke ...	Bindungs- oder Tupelanzeige, Listenanzeige, Wörterbuchanzeige, Zeichenfolgenkonvertierung

Examples

Beispiele für einfache Operator-Priorität in Python.

Python folgt der PEMDAS-Regel. PEMDAS steht für Klammern, Exponenten, Multiplikation und Division sowie Addition und Subtraktion.

Beispiel:

```
>>> 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
```

Extras: mathematische Regeln gelten, aber nicht immer :

```
>>> 300 / 300 * 200
200.0
>>> 300 * 200 / 300
200.0
>>> 1e300 / 1e300 * 1e200
1e+200
>>> 1e300 * 1e200 / 1e300
inf
```

Vorrang des Bedieners online lesen: https://riptutorial.com/de/python/topic/5040/vorrang-des-bedieners

Kapitel 195: Warteschlangenmodul

Einführung

Das Warteschlangenmodul implementiert Warteschlangen mit mehreren Herstellern und mehreren Verbrauchern. Es ist besonders nützlich bei der Thread-Programmierung, wenn Informationen zwischen mehreren Threads sicher ausgetauscht werden müssen. Es gibt drei Arten von Warteschlangen, die vom Warteschlangenmodul bereitgestellt werden. Diese sind folgende: 1. Warteschlange 2. LifoQueue 3. PriorityQueue Ausnahme, die kommen könnte: 1. Full (Warteschlangenüberlauf) 2. Leer (Warteschlangenunterlauf)

Examples

Einfaches Beispiel

```
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))
```

Ausgabe:

```
('key', 1)
('key', 2)
('key', 3)
('key', 4)
('key', 5)
('key', 6)
('key', 7)
('key', 8)
('key', 9)
```

Warteschlangenmodul online lesen:
https://riptutorial.com/de/python/topic/8339/warteschlangenmodul

Kapitel 196: Webbrowser-Modul

Einführung

Gemäß der Standarddokumentation von Python bietet das Webbrowser-Modul eine übergeordnete Schnittstelle, über die Benutzer webbasierte Dokumente angezeigt werden können. In diesem Thema wird die ordnungsgemäße Verwendung des Webbrowser-Moduls beschrieben und veranschaulicht.

Syntax

- `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)`

Parameter

Parameter	Einzelheiten
`webbrowser.open()`	
URL	die URL, die im Webbrowser geöffnet werden soll
Neu	0 öffnet die URL in der vorhandenen Registerkarte, 1 öffnet sich in einem neuen Fenster, 2 öffnet sich in einer neuen Registerkarte
automatisieren	Bei Einstellung auf True wird das Fenster über den anderen Fenstern verschoben
`webbrowser.open_new()`	
URL	die URL, die im Webbrowser geöffnet werden soll
`webbrowser.open_new_tab()`	
URL	die URL, die im Webbrowser geöffnet werden soll
`webbrowser.get()`	
mit	den zu verwendenden Browser
`webbrowser.register()`	
URL	Name des Browsers
Konstrukteur	Pfad zum ausführbaren Browser (Hilfe)

Parameter	Einzelheiten
Beispiel	Eine Instanz eines Webbrowsers, die von der `webbrowser.get()` -Methode zurückgegeben wurde

Bemerkungen

In der folgenden Tabelle sind vordefinierte Browsertypen aufgeführt. Die linke Spalte enthält Namen, die an die `webbrowser.get()` Methode übergeben werden können. In der rechten Spalte werden die Klassennamen für jeden Browsertyp aufgeführt.

Modellname	Klassenname
`'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

URL mit dem Standardbrowser öffnen

Um einfach eine URL zu öffnen, verwenden Sie die `webbrowser.open()` -Methode:

```
import webbrowser
webbrowser.open("http://stackoverflow.com")
```

Wenn gerade ein Browserfenster geöffnet ist, öffnet die Methode eine neue Registerkarte unter der angegebenen URL. Wenn kein Fenster geöffnet ist, öffnet die Methode den Standardbrowser des Betriebssystems und navigiert zu der URL im Parameter. Die offene Methode unterstützt die folgenden Parameter:

- `url` - die im Webbrowser zu öffnende URL (Zeichenfolge) **[erforderlich]**
- `new` - 0 wird in vorhandenem Tab geöffnet, 1 öffnet neues Fenster, 2 öffnet neuer Tab (Ganzzahl) **[Standardeinstellung 0]**
- `autoraise` - Bei Einstellung auf True wird das Fenster über den Fenstern anderer Anwendungen verschoben (Boolean) **[default False]**

Beachten Sie, dass die `new` und `autoraise` Argumente nur selten funktionieren, da die meisten modernen Browser diese Befehle ablehnen.

Webbrowser kann auch versuchen, URLs mit der Methode `open_new` in neuen Fenstern zu `open_new` :

```
import webbrowser
webbrowser.open_new("http://stackoverflow.com")
```

Diese Methode wird von modernen Browsern häufig ignoriert und die URL wird normalerweise auf einer neuen Registerkarte geöffnet. Das Öffnen eines neuen Tabs kann vom Modul mit der Methode `open_new_tab` versucht werden:

```
import webbrowser
webbrowser.open_new_tab("http://stackoverflow.com")
```

URL mit verschiedenen Browsern öffnen

Das Webbrowser-Modul unterstützt auch verschiedene Browser mit den Methoden `register()` und `get()` . Die Get-Methode wird zum Erstellen eines Browser-Controllers unter Verwendung des Pfads einer bestimmten ausführbaren Datei verwendet. Mit der Registermethode werden diese ausführbaren Dateien zur zukünftigen Verwendung an voreingestellte Browsertypen angehängt, normalerweise bei Verwendung mehrerer Browsertypen.

```
import webbrowser
ff_path = webbrowser.get("C:/Program Files/Mozilla Firefox/firefox.exe")
ff = webbrowser.get(ff_path)
ff.open("http://stackoverflow.com/")
```

Einen Browsertyp registrieren:

```
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/")
```

Webbrowser-Modul online lesen: https://riptutorial.com/de/python/topic/8676/webbrowser-modul

Kapitel 197: Web-Scraping mit Python

Einführung

Web-Scraping ist ein automatisierter, programmatischer Prozess, durch den Daten ständig von Webseiten "abgeschabt" werden können. Web Scraping, auch als Screen Scraping oder Web Harvesting bekannt, kann sofortige Daten von jeder öffentlich zugänglichen Webseite bereitstellen. Auf einigen Websites ist das Scraping von Websites möglicherweise illegal.

Bemerkungen

Nützliche Python-Pakete für das Web-Scraping (alphabetische Reihenfolge)

Anfragen stellen und Daten sammeln

`requests`

Ein einfaches, aber leistungsfähiges Paket für HTTP-Anfragen.

`requests-cache`

Caching für `requests` ; Das Zwischenspeichern von Daten ist sehr nützlich. In der Entwicklung bedeutet dies, dass Sie vermeiden müssen, eine Website unnötig zu treffen. Wenn Sie eine echte Sammlung ausführen, bedeutet dies, dass wenn Ihr Scraper aus irgendeinem Grund abstürzt (möglicherweise haben Sie nicht mit ungewöhnlichen Inhalten auf der Website umgegangen ...? Vielleicht ist die Website ausgefallen ...?), Können Sie die Sammlung sehr schnell wiederholen von wo du aufgehört hast.

`scrapy`

Nützlich für das Erstellen von Webcrawlern, bei denen Sie etwas Stärkeres benötigen, als `requests` und Seiten zu durchlaufen.

`selenium`

Python-Bindungen für Selenium WebDriver zur Browser-Automatisierung. Mit `requests` an HTTP - Anfragen direkt ist oft einfacher , für das Abrufen von Webseiten zu machen. Dies ist jedoch ein nützliches Werkzeug, wenn es nicht möglich ist, das gewünschte Verhalten einer Website nur mit `requests` zu replizieren, insbesondere wenn JavaScript zum Rendern von Elementen auf einer Seite erforderlich ist.

HTML-Analyse

BeautifulSoup

Fragen Sie HTML- und XML-Dokumente mit einer Reihe verschiedener Parser ab (in Python integrierter HTML-Parser, `html5lib` , `lxml` oder `lxml.html`).

lxml

Verarbeitet HTML und XML. Kann verwendet werden, um Inhalte aus HTML-Dokumenten über CSS-Selektoren und XPath abzufragen und auszuwählen.

Examples

Ein einfaches Beispiel für die Verwendung von Anforderungen und lxml zum Abwischen einiger Daten

```
# 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()
```

Web-Scraping-Sitzung mit Anforderungen verwalten

Es ist eine gute Idee, eine Web-Scraping-Sitzung durchzuführen, um die Cookies und andere Parameter beizubehalten. Zusätzlich kann es zu einer *Leistungssteigerung* führen , da `requests.Session` die zugrunde liegende TCP - Verbindung zu einem Host wieder verwendet:

```
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)
```

Scraping mit dem Scrapy-Framework

Zuerst müssen Sie ein neues Scrapy-Projekt einrichten. Geben Sie ein Verzeichnis ein, in dem Sie Ihren Code speichern möchten, und führen Sie Folgendes aus:

```
scrapy startproject projectName
```

Zum Kratzen brauchen wir eine Spinne. Spider definieren, wie eine bestimmte Site abgestreift wird. Hier ist der Code für eine Spinne, die den Links zu den am häufigsten gestellten Fragen zu StackOverflow folgt und einige Daten von jeder Seite (Quelle) kratzt:

```
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,
        }
```

Speichern Sie Ihre Spider-Klassen im Verzeichnis `projectName\spiders` . In diesem Fall - `projectName\spiders\stackoverflow_spider.py` .

Jetzt kannst du deine Spinne benutzen. Versuchen Sie beispielsweise (im Verzeichnis des Projekts) auszuführen:

```
scrapy crawl stackoverflow
```

Ändern Sie den Scrapy-Benutzeragenten

Manchmal wird der standardmäßige Scrapy-Benutzeragent (`"Scrapy/VERSION (+http://scrapy.org)"`) vom Host blockiert. Um den Standardbenutzeragenten zu ändern, öffnen Sie die Datei **settings.py** , entfernen Sie das Kommentarzeichen und bearbeiten Sie die folgende Zeile in die gewünschte Zeile.

```
#USER_AGENT = 'projectName (+http://www.yourdomain.com)'
```

Zum Beispiel

```
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'
```

Kratzen mit 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
```

Kratzen mit Selenium WebDriver

Einige Websites mögen es nicht, abgeschabt zu werden. In diesen Fällen müssen Sie möglicherweise einen echten Benutzer simulieren, der mit einem Browser arbeitet. Selen startet und steuert einen Webbrowser.

```
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)
```

Selen kann viel mehr. Es kann Browser-Cookies ändern, Formulare ausfüllen, Mausklicks simulieren, Screenshots von Webseiten erstellen und benutzerdefiniertes JavaScript ausführen.

Einfacher Download von Webinhalten mit urllib.request

Das Standardbibliotheksmodul `urllib.request` kann zum Herunterladen von Webinhalten verwendet werden:

```
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)
```

Ein ähnliches Modul ist auch in Python 2 verfügbar.

Kratzen mit Locken

Importe:

```
from subprocess import Popen, PIPE
from lxml import etree
from io import StringIO
```

Wird heruntergeladen:

```
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` : stummer Download

`-A` : Benutzeragentenflag

Parsing:

```
tree = etree.parse(StringIO(result), etree.HTMLParser())
divs = tree.xpath('//div')
```

Web-Scraping mit Python online lesen: https://riptutorial.com/de/python/topic/1792/web-scraping-mit-python

Kapitel 198: Webserver-Gateway-Schnittstelle (WSGI)

Parameter

Parameter	Einzelheiten
start_response	Eine Funktion zum Verarbeiten des Starts

Examples

Serverobjekt (Methode)

Unser Serverobjekt erhält einen 'application'-Parameter, bei dem es sich um ein beliebiges aufrufbares Anwendungsobjekt handeln kann (siehe andere Beispiele). Es schreibt zuerst die Header und dann den Datenbestand, den unsere Anwendung an die Systemstandardausgabe zurückgibt.

```
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()'
```

Webserver-Gateway-Schnittstelle (WSGI) online lesen:
https://riptutorial.com/de/python/topic/5315/webserver-gateway-schnittstelle--wsgi-

Kapitel 199: Websockets

Examples

Einfaches Echo mit aiohttp

`aiohttp` stellt asynchrone `aiohttp` bereit.

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())
```

Wrapper-Klasse mit aiohttp

`aiohttp.ClientSession` kann als übergeordnetes `aiohttp.ClientSession` für eine benutzerdefinierte WebSocket-Klasse verwendet werden.

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()
```

Autobahn als Websocket-Fabrik nutzen

Das Autobahn-Paket kann für Python-Websocket-Serverfabriken verwendet werden.

Python Autobahnpaket-Dokumentation

Zur Installation wird normalerweise der Befehl terminal verwendet

(Für Linux):

```
sudo pip install autobahn
```

(Für Windows):

```
python -m pip install autobahn
```

Dann kann ein einfacher Echo-Server in einem Python-Skript erstellt werden:

```
from autobahn.asyncio.websocket import WebSocketServerProtocol
```

```
class MyServerProtocol(WebSocketServerProtocol):
    '''When creating server protocol, the
    user defined class inheriting the
    WebSocketServerProtocol needs to override
    the onMessage, onConnect, et-c events for
    user specified functionality, these events
    define your server's protocol, in essence'''
    def onMessage(self,payload,isBinary):
        '''The onMessage routine is called
        when the server receives a message.
        It has the required arguments payload
        and the bool isBinary. The payload is the
        actual contents of the "message" and isBinary
        is simply a flag to let the user know that
        the payload contains binary data. I typically
        elsewise assume that the payload is a string.
        In this example, the payload is returned to sender verbatim.'''
        self.sendMessage(payload,isBinary)
if__name__=='__main__':
    try:
        importasyncio
    except ImportError:
        '''Trollius = 0.3 was renamed'''
        import trollius as asyncio
    from autobahn.asyncio.websocketimportWebSocketServerFactory
    factory=WebSocketServerFactory()
    '''Initialize the websocket factory, and set the protocol to the
    above defined protocol(the class that inherits from
    autobahn.asyncio.websocket.WebSocketServerProtocol)'''
    factory.protocol=MyServerProtocol
    '''This above line can be thought of as "binding" the methods
    onConnect, onMessage, et-c that were described in the MyServerProtocol class
    to the server, setting the servers functionality, ie, protocol'''
    loop=asyncio.get_event_loop()
    coro=loop.create_server(factory,'127.0.0.1',9000)
    server=loop.run_until_complete(coro)
    '''Run the server in an infinite loop'''
    try:
        loop.run_forever()
    except KeyboardInterrupt:
        pass
    finally:
        server.close()
        loop.close()
```

In diesem Beispiel wird auf dem localhost (127.0.0.1) an Port 9000 ein Server erstellt. Dies ist die zu überwachende IP-Adresse und der Port. Dies ist eine wichtige Information, da Sie damit die LAN-Adresse Ihres Computers und die Weiterleitung des Ports von Ihrem Modem aus identifizieren können, unabhängig davon, welche Router Sie zum Computer haben. Anschließend können Sie mithilfe von Google Ihre WAN-IP-Adresse untersuchen. Sie können Ihre Website so einrichten, dass WebSocket-Nachrichten an Port 9000 (in diesem Beispiel) an Ihre WAN-IP-Adresse gesendet werden.

Es ist wichtig, dass Sie von Ihrem Modem aus einen Port-Forward-Vorgang durchführen, dh, wenn Sie Router an das Modem angeschlossen haben, geben Sie die Konfigurationseinstellungen des Modems ein, den Port-Forwarder vom Modem zum angeschlossenen Router usw. bis zum endgültigen Router Ihres Computers verbunden ist, wird die Information über den Modemport

9000 (in diesem Beispiel) an ihn weitergeleitet.

Websockets online lesen: https://riptutorial.com/de/python/topic/4751/websockets

Kapitel 200: Wörterbuch

Syntax

- mydict = {}
- meinDict [k] = Wert
- Wert = Mydict [k]
- value = meinedict.get (k)
- value = mydict.get (k, "default_value")

Parameter

Parameter	Einzelheiten
Schlüssel	Der gewünschte Schlüssel zum Nachschlagen
Wert	Der Wert, der gesetzt oder zurückgegeben werden soll

Bemerkungen

Hilfreiche Punkte, die Sie beim Erstellen eines Wörterbuchs beachten sollten:

- Jeder Schlüssel muss **eindeutig sein** (sonst wird er überschrieben)
- Jeder Schlüssel muss **hashbar sein** (kann mit der `hash` Funktion `TypeError` werden; andernfalls wird `TypeError` ausgelöst)
- Es gibt keine besondere Reihenfolge für die Schlüssel.

Examples

Zugriff auf Werte eines Wörterbuchs

```
dictionary = {"Hello": 1234, "World": 5678}
print(dictionary["Hello"])
```

Der obige Code wird `1234` drucken.

Die Zeichenfolge `"Hello"` in diesem Beispiel als *Schlüssel bezeichnet* . Es wird verwendet, um einen Wert im `dict` indem der Schlüssel in eckige Klammern gesetzt wird.

Die Nummer `1234` steht hinter dem jeweiligen Doppelpunkt in der `dict` . Dies ist der *Wert,* genannt `"Hello"` *Karten* in diesem `dict` .

Wenn Sie einen `KeyError` Wert mit einem nicht vorhandenen Schlüssel `KeyError` , wird eine `KeyError` Ausnahme `KeyError` , die die Ausführung `KeyError` , wenn sie nicht `KeyError` . Wenn wir auf

einen Wert zugreifen möchten, ohne einen `KeyError` zu riskieren, können wir die `dictionary.get` Methode verwenden. Wenn der Schlüssel nicht vorhanden ist, gibt die Methode standardmäßig `None` . Im Falle eines fehlgeschlagenen Suchvorgangs können Sie einen zweiten Wert übergeben, der anstelle von `None` .

```
w = dictionary.get("whatever")
x = dictionary.get("whatever", "nuh-uh")
```

In diesem Beispiel erhält `w` den Wert `None` und `x` den Wert `"nuh-uh"` .

Der dict () -Konstruktor

Der `dict()` kann zum Erstellen von Wörterbüchern aus Schlüsselwortargumenten oder aus einem einzigen iterierbaren Schlüsselpaar oder aus einem einzelnen Wörterbuch und Schlüsselwortargumenten verwendet werden.

```
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}
```

Vermeiden von KeyError-Ausnahmen

Bei der Verwendung von Wörterbüchern besteht häufig die Gefahr, auf einen nicht vorhandenen Schlüssel zuzugreifen. Dies führt normalerweise zu einer `KeyError` Ausnahme

```
mydict = {}
mydict['not there']
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'not there'
```

Eine Möglichkeit, Schlüsselfehler zu vermeiden, ist die Verwendung der Methode `dict.get` , mit der Sie einen Standardwert angeben können, der im Falle eines fehlenden Schlüssels zurückgegeben werden soll.

```
value = mydict.get(key, default_value)
```

`mydict[key]` falls vorhanden, ansonsten `default_value` . Beachten Sie, dass dies keinen `key` zu `mydict` hinzufügt. Also , wenn Sie diesen Schlüssel Wertpaar beibehalten möchten, sollten Sie `mydict.setdefault(key, default_value)` , die den Schlüssel Wertpaar speichert.

```
mydict = {}
print(mydict)
# {}
print(mydict.get("foo", "bar"))
# bar
print(mydict)
```

```
# {}
print(mydict.setdefault("foo", "bar"))
# bar
print(mydict)
# {'foo': 'bar'}
```

Eine alternative Möglichkeit, mit dem Problem umzugehen, ist die Ausnahme

```
try:
    value = mydict[key]
except KeyError:
    value = default_value
```

Sie können auch überprüfen, ob der Schlüssel `in` Wörterbuch enthalten ist.

```
if key in mydict:
    value = mydict[key]
else:
    value = default_value
```

Beachten Sie jedoch, dass es in Umgebungen mit mehreren Threads möglich ist, dass der Schlüssel nach der Überprüfung aus dem Wörterbuch entfernt wird, wodurch eine Race-Bedingung erstellt wird, bei der die Ausnahme immer noch ausgelöst werden kann.

Eine andere Option ist die Verwendung einer Unterklasse von dict, Collections.defaultdict, die über eine default_factory verfügt, um neue Einträge im dict zu erstellen, wenn ein neuer_key gegeben wird.

Zugriff auf Schlüssel und Werte

Wenn Sie mit Wörterbüchern arbeiten, ist es häufig erforderlich, auf alle Schlüssel und Werte im Wörterbuch zuzugreifen, entweder in einer `for` Schleife, einem Listenverständnis oder einfach als einfache Liste.

Gegeben ein Wörterbuch wie:

```
mydict = {
    'a': '1',
    'b': '2'
}
```

Sie können eine Liste der Schlüssel mit der `keys()` -Methode erhalten:

```
print(mydict.keys())
# Python2: ['a', 'b']
# Python3: dict_keys(['b', 'a'])
```

Wenn Sie stattdessen eine Liste mit Werten wünschen, verwenden Sie die Methode `values()` :

```
print(mydict.values())
# Python2: ['1', '2']
```

```
# Python3: dict_values(['2', '1'])
```

Wenn Sie sowohl mit dem Schlüssel als auch mit dem entsprechenden Wert arbeiten möchten, können Sie die Methode `items()` verwenden:

```
print(mydict.items())
# Python2: [('a', '1'), ('b', '2')]
# Python3: dict_items([('b', '2'), ('a', '1')])
```

HINWEIS: Da ein `dict` unsortiert ist, haben `keys()` , `values()` und `items()` keine Sortierreihenfolge. Verwenden Sie `sort()` , `OrderedDict sorted()` oder `OrderedDict` wenn Sie die Reihenfolge `OrderedDict` , die diese Methoden zurückgeben.

Python-2/3-Unterschied: In Python 3 geben diese Methoden spezielle iterierbare Objekte und keine Listen zurück. Sie entsprechen den Python-2- `iterkeys()` , `itervalues()` und `iteritems()` . Diese Objekte können größtenteils wie Listen verwendet werden, es gibt jedoch einige Unterschiede. Weitere Informationen finden Sie in PEP 3106 .

Einführung in das Wörterbuch

Ein Wörterbuch ist ein Beispiel für einen *Schlüsselwertspeicher, der* auch als *Mapping* in Python bezeichnet wird. Sie können Elemente speichern und abrufen, indem Sie auf einen Schlüssel verweisen. Da Wörterbücher über Schlüssel referenziert werden, können sie sehr schnell nachschlagen. Da sie hauptsächlich zur Referenzierung von Artikeln nach Schlüssel verwendet werden, werden sie nicht sortiert.

ein dikt erstellen

Wörterbücher können auf verschiedene Arten initiiert werden:

wörtliche Syntax

```
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}
```

Diktierverständnis

```
d = {k:v for k,v in [('key', 'value',)]}
```

siehe auch: Verständnis

eingebaute Klasse: `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)
```

ein dikt ändern

Um einem Wörterbuch Elemente hinzuzufügen, erstellen Sie einfach einen neuen Schlüssel mit einem Wert:

```
d['newkey'] = 42
```

Es ist auch möglich, `list` und `dictionary` als Wert hinzuzufügen:

```
d['new_list'] = [1, 2, 3]
d['new_dict'] = {'nested_dict': 1}
```

Um ein Element zu löschen, löschen Sie den Schlüssel aus dem Wörterbuch:

```
del d['newkey']
```

Wörterbuch mit Standardwerten

Verfügbar in der Standardbibliothek als `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'
```

[*] Wenn Sie die integrierte `dict` Klasse `using dict.setdefault()` müssen, können Sie alternativ bei `using dict.setdefault()` einen Standardwert erstellen, wenn Sie auf einen Schlüssel zugreifen, der zuvor nicht vorhanden war:

```
>>> d = {}
{}
>>> d.setdefault('Another_key', []).append("This worked!")
>>> d
{'Another_key': ['This worked!']}
```

Wenn Sie viele Werte hinzufügen müssen, erstellt `dict.setdefault()` bei `dict.setdefault()` eine neue Instanz des Anfangswerts (in diesem Beispiel ein `[]`). Dies kann zu unnötigen Workloads führen.

[*] *Python Cookbook, 3. Auflage, von David Beazley und Brian K. Jones (O'Reilly). Copyright 2013 David Beazley und Brian Jones, 978-1-449-34037-7.*

Ein geordnetes Wörterbuch erstellen

Sie können ein geordnetes Wörterbuch erstellen, das beim Iterieren über die Tasten im Wörterbuch einer festgelegten Reihenfolge folgt.

Verwenden Sie `OrderedDict` aus dem `collections` Modul. Dadurch werden die Wörterbuchelemente immer in der ursprünglichen Einfügereihenfolge zurückgegeben, wenn sie wiederholt werden.

```
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])
```

Wörterbücher mit dem Operator ** auspacken

Sie können den Operator zum Entschlüsseln des Schlüsselworts `**` , um die Schlüssel-Wert-Paare in einem Wörterbuch in die Argumente einer Funktion zu übermitteln. Ein vereinfachtes Beispiel aus der offiziellen Dokumentation :

```
>>>
>>> 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 !
```

Ab Python 3.5 können Sie diese Syntax auch verwenden, um eine beliebige Anzahl von `dict` Objekten zusammenzuführen.

```
>>> 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'}
```

Wie dieses Beispiel zeigt, werden Schlüssel auf den letzten Wert kopiert (z. B. überschreibt "Clifford" "Nemo").

Wörterbücher zusammenführen

Betrachten Sie die folgenden Wörterbücher:

```
>>> 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'}
```

Wie dieses Beispiel zeigt, werden Schlüssel auf den letzten Wert kopiert (z. B. überschreibt "Clifford" "Nemo").

Python 3.3+

```
>>> from collections import ChainMap
>>> dict(ChainMap(fish, dog))
{'hands': 'fins', 'color': 'red', 'special': 'gills', 'name': 'Nemo'}
```

Bei dieser Technik hat der vorderste Wert für einen bestimmten Schlüssel Vorrang vor dem letzten ("Clifford" wird zugunsten von "Nemo" verworfen).

Python 2.x, 3.x

```
>>> from itertools import chain
>>> dict(chain(fish.items(), dog.items()))
{'hands': 'paws', 'color': 'red', 'name': 'Clifford', 'special': 'gills'}
```

Dies verwendet den letzten Wert, wie bei der `**` -basierten Technik zum Zusammenführen ("Clifford" überschreibt "Nemo").

```
>>> fish.update(dog)
>>> fish
{'color': 'red', 'hands': 'paws', 'name': 'Clifford', 'special': 'gills'}
```

`dict.update` verwendet das letzte Diktat, um das vorherige zu überschreiben.

Das nachfolgende Komma

Wie Listen und Tupel können Sie in Ihr Wörterbuch ein nachfolgendes Komma einfügen.

```
role = {"By day": "A typical programmer",
        "By night": "Still a typical programmer", }
```

PEP 8 schreibt vor, dass zwischen dem nachgestellten Komma und der schließenden Klammer ein Leerzeichen stehen soll.

Alle Kombinationen von Wörterbuchwerten

```
options = {
    "x": ["a", "b"],
    "y": [10, 20, 30]
}
```

Angenommen, ein Wörterbuch wie das oben gezeigte, bei dem es eine Liste gibt, die einen Satz von Werten darstellt, um nach dem entsprechenden Schlüssel zu suchen. Angenommen, Sie möchten `"x"="a"` mit `"y"=10` erkunden, dann `"x"="a"` mit `"y"=10` usw., bis Sie alle möglichen Kombinationen untersucht haben.

Sie können eine Liste erstellen, die alle derartigen Wertekombinationen mithilfe des folgenden Codes zurückgibt.

```
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
```

Dies gibt uns die folgende Liste in den Variablen gespeichert `combinations` :

```
[{'x': 'a', 'y': 10},
 {'x': 'b', 'y': 10},
 {'x': 'a', 'y': 20},
 {'x': 'b', 'y': 20},
 {'x': 'a', 'y': 30},
 {'x': 'b', 'y': 30}]
```

Iteration über ein Wörterbuch

Wenn Sie ein Wörterbuch als Iterator verwenden (z. B. in einer `for` Anweisung), werden die **Schlüssel** des Wörterbuchs durchsucht. Zum Beispiel:

```
d = {'a': 1, 'b': 2, 'c':3}
for key in d:
    print(key, d[key])
# c 3
# b 2
# a 1
```

Dasselbe gilt, wenn es in einem Verständnis verwendet wird

```
print([key for key in d])
# ['c', 'b', 'a']
```

Python 3.x 3.0

Die `items()` -Methode kann verwendet werden, um sowohl den **Schlüssel** als auch den **Wert** gleichzeitig zu durchlaufen:

```
for key, value in d.items():
    print(key, value)
# c 3
# b 2
# a 1
```

Während die `values()` -Methode verwendet werden kann, um wie erwartet nur die Werte zu durchlaufen:

```
for key, value in d.values():
    print(key, value)
    # 3
    # 2
    # 1
```

Python 2.x 2.2

Hier geben die Methoden `keys()` , `values()` und `items()` Listen zurück, und es gibt die drei zusätzlichen Methoden `iterkeys()` `itervalues()` und `iteritems()` , um Iterater zurückzugeben.

Wörterbuch erstellen

Regeln zum Erstellen eines Wörterbuchs:

- Jeder Schlüssel muss **eindeutig sein** (sonst wird er überschrieben)
- Jeder Schlüssel muss **hashbar sein** (kann mit der `hash` Funktion `TypeError` werden; andernfalls wird `TypeError` ausgelöst)
- Es gibt keine besondere Reihenfolge für die Schlüssel.

```
# 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}
```

Wörterbücher Beispiel

Wörterbücher ordnen Schlüssel zu Werten zu.

```
car = {}
car["wheels"] = 4
car["color"] = "Red"
car["model"] = "Corvette"
```

Auf die Wörterbuchwerte kann über ihre Schlüssel zugegriffen werden.

```
print "Little " + car["color"] + " " + car["model"] + "!"
# This would print out "Little Red Corvette!"
```

Wörterbücher können auch in einem JSON-Stil erstellt werden:

```
car = {"wheels": 4, "color": "Red", "model": "Corvette"}
```

Wörterbuchwerte können wiederholt werden:

```
for key in car:
  print key + ": " + car[key]

# wheels: 4
# color: Red
# model: Corvette
```

Wörterbuch online lesen: https://riptutorial.com/de/python/topic/396/worterbuch

Kapitel 201: XML bearbeiten

Bemerkungen

Nicht alle Elemente der XML-Eingabe enden als Elemente des analysierten Baums. Derzeit werden in diesem Modul alle XML-Kommentare, Verarbeitungsanweisungen und Dokumenttypdeklarationen in der Eingabe übersprungen. Trotzdem können Bäume, die mit der API dieses Moduls erstellt wurden, anstatt aus XML-Text analysiert zu werden, Kommentare und Verarbeitungsanweisungen enthalten. Sie werden beim Generieren der XML-Ausgabe berücksichtigt.

Examples

Öffnen und Lesen mit einem ElementTree

Importieren Sie das ElementTree-Objekt, öffnen Sie die relevante XML-Datei und rufen Sie das Root-Tag ab:

```
import xml.etree.ElementTree as ET
tree = ET.parse("yourXMLfile.xml")
root = tree.getroot()
```

Es gibt mehrere Möglichkeiten, den Baum zu durchsuchen. Erstens durch Iteration:

```
for child in root:
    print(child.tag, child.attrib)
```

Ansonsten können Sie bestimmte Orte wie eine Liste referenzieren:

```
print(root[0][1].text)
```

Um nach bestimmten Tags nach Namen zu suchen, verwenden Sie `.find` oder `.findall` :

```
print(root.findall("myTag"))
print(root[0].find("myOtherTag"))
```

Ändern einer XML-Datei

Importieren Sie das Element Tree-Modul und öffnen Sie die XML-Datei, erhalten Sie ein XML-Element

```
import xml.etree.ElementTree as ET
tree = ET.parse('sample.xml')
root=tree.getroot()
element = root[0] #get first child of root element
```

Elementobjekte können durch Ändern der Felder, Hinzufügen und Ändern von Attributen sowie Hinzufügen und Entfernen von untergeordneten Elementen bearbeitet werden

```
element.set('attribute_name', 'attribute_value') #set the attribute to xml element
element.text="string_text"
```

Wenn Sie ein Element entfernen möchten, verwenden Sie die Element.remove () -Methode

```
root.remove(element)
```

ElementTree.write () - Methode zur Ausgabe des XML-Objekts in XML-Dateien.

```
tree.write('output.xml')
```

Erstellen und erstellen Sie XML-Dokumente

Importieren Sie das Element Tree-Modul

```
import xml.etree.ElementTree as ET
```

Die Funktion Element () wird zum Erstellen von XML-Elementen verwendet

```
p=ET.Element('parent')
```

SubElement () - Funktion zum Erstellen von Unterelementen zu einem gegebenen Element

```
c = ET.SubElement(p, 'child1')
```

Die Funktion dump () dient zum Ablegen von XML-Elementen.

```
ET.dump(p)
# Output will be like this
#<parent><child1 /></parent>
```

Wenn Sie in einer Datei speichern möchten, erstellen Sie einen XML-Baum mit der ElementTree () - Funktion und speichern Sie ihn in einer Datei

```
tree = ET.ElementTree(p)
tree.write("output.xml")
```

Die Comment () - Funktion wird verwendet, um Kommentare in die XML-Datei einzufügen.

```
comment = ET.Comment('user comment')
p.append(comment) #this comment will be appended to parent element
```

Öffnen und Lesen großer XML-Dateien mithilfe von iterparse (inkrementelles Parsing)

Manchmal möchten wir nicht die gesamte XML-Datei laden, um die benötigten Informationen zu erhalten. In diesen Fällen ist es nützlich, die relevanten Abschnitte inkrementell laden zu können und sie anschließend zu löschen, wenn wir fertig sind. Mit der iterparse-Funktion können Sie die Elementstruktur bearbeiten, die beim Analysieren der XML-Datei gespeichert wird.

Importieren Sie das ElementTree-Objekt:

```
import xml.etree.ElementTree as ET
```

Öffnen Sie die XML-Datei und durchlaufen Sie alle Elemente:

```
for event, elem in ET.iterparse("yourXMLfile.xml"):
    ... do something ...
```

Alternativ können wir nur nach bestimmten Ereignissen suchen, z. B. Start- / End-Tags oder Namespaces. Wenn diese Option weggelassen wird (wie oben), werden nur "End" -Ereignisse zurückgegeben:

```
events=("start", "end", "start-ns", "end-ns")
for event, elem in ET.iterparse("yourXMLfile.xml", events=events):
    ... do something ...
```

Das vollständige Beispiel zeigt, wie Elemente aus dem In-Memory-Baum gelöscht werden, wenn wir damit fertig sind:

```
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 ...
```

Durchsuchen des XML mit XPath

Ab Version 2.7 `ElementTree` eine bessere Unterstützung für XPath-Abfragen. XPath ist eine Syntax, mit der Sie durch eine XML-Datei navigieren können, wie SQL zum Durchsuchen einer Datenbank verwendet wird. `findall` Funktionen `find` und `findall` unterstützen XPath. Die folgende XML wird für dieses Beispiel verwendet

```
 <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>
```

Suche nach allen Büchern:

```
import xml.etree.cElementTree as ET
tree = ET.parse('sample.xml')
tree.findall('Books/Book')
```

Suche nach dem Buch mit Titel = "Die Farbe der Magie":

```
tree.find("Books/Book[Title='The Colour of Magic']")
# always use '' in the right side of the comparison
```

Suche nach dem Buch mit id = 5:

```
tree.find("Books/Book[@id='5']")
# searches with xml attributes must have '@' before the name
```

Suche nach dem zweiten Buch:

```
tree.find("Books/Book[2]")
# indexes starts at 1, not 0
```

Suche nach dem letzten Buch:

```
tree.find("Books/Book[last()]")
# 'last' is the only xpath function allowed in ElementTree
```

Suche nach allen Autoren:

```
tree.findall(".//Author")
#searches with // must use a relative path
```

XML bearbeiten online lesen: https://riptutorial.com/de/python/topic/479/xml-bearbeiten

Kapitel 202: Zählen

Examples

Zählen aller Vorkommen aller Elemente in einer iterierbaren: collection.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
```

Der `collections.Counter` kann für jedes beliebige Element verwendet werden und zählt jedes Vorkommen für jedes Element.

Hinweis : Eine Ausnahme ist, wenn ein `dict` oder eine andere `collections.Mapping` Klasse angegeben ist. Dann werden sie nicht gezählt, sondern es wird ein Counter mit den folgenden Werten erstellt:

```
Counter({"e": 2})
# Out: Counter({"e": 2})

Counter({"e": "e"})        # warning Counter does not verify the values are int
# Out: Counter({"e": "e"})
```

Den häufigsten Wert (-s) ermitteln: Collections.Counter.most_common ()

Das Zählen der *Schlüssel* eines `Mapping` ist bei `collections.Counter` nicht möglich. Zähler, aber wir können die *Werte zählen* :

```
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})
```

Die häufigsten Elemente sind mit der `most_common` Methode `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)]
```

Zählen der Vorkommen eines Elements in einer Sequenz: list.count () und 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
```

Zählen der Vorkommen eines Teilstrings in einem String: str.count ()

```
astring = 'thisisashorttext'
astring.count('t')
# Out: 4
```

Dies funktioniert auch für Teilzeichenfolgen, die länger als ein Zeichen sind:

```
astring.count('th')
# Out: 1
astring.count('is')
# Out: 2
astring.count('text')
# Out: 1
```

was mit `collections.Counter` nicht möglich wäre. Zähler, der nur einzelne Zeichen zählt:

```
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})
```

Zählung von Vorkommen im numpy-Array

Zählen der Vorkommen eines Werts in einem numpy-Array. Das wird funktionieren:

```
>>> import numpy as np
>>> a=np.array([0,3,4,3,5,4,7])
>>> print np.sum(a==3)
2
```

Die Logik ist, dass die boolesche Anweisung ein Array erzeugt, in dem alle Vorkommen der angeforderten Werte 1 sind und alle anderen Null sind. Das Summieren dieser Werte ergibt also die Anzahl der Vorkommen. Dies funktioniert für Arrays mit beliebigen Formen oder Datentypen.

Ich verwende zwei Methoden, um Vorkommen aller eindeutigen Werte in numpy zu zählen. Einzigartig und zahlreich. Unique reduziert automatisch mehrdimensionale Arrays, während bincount nur bei 1d-Arrays funktioniert, die nur positive Ganzzahlen enthalten.

```
>>> 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]
```

Wenn Ihre Daten aus numpy-Arrays bestehen, ist es im Allgemeinen viel schneller, numpy-Methoden zu verwenden, als Ihre Daten in generische Methoden umzuwandeln.

Zählen online lesen: https://riptutorial.com/de/python/topic/476/zahlen

Kapitel 203: Zeichenfolgendarstellungen von Klasseninstanzen: __str__- und __repr__-Methoden

Bemerkungen

Ein Hinweis zum Implementieren beider Methoden

Wenn beide Methoden implementiert werden, ist es üblich, eine `__str__` Methode zu verwenden, die eine menschenfreundliche Darstellung (z. B. "Ace of Spaces") und `__repr__` eine `eval` freundliche Darstellung zurückgibt.

Tatsächlich beachten die Python-Dokumente für `repr()` genau das `repr()` :

> Bei vielen Typen versucht diese Funktion, eine Zeichenfolge zurückzugeben, die ein Objekt mit demselben Wert ergibt, wenn sie an eval () übergeben wird. Andernfalls handelt es sich bei der Darstellung um eine in spitze Klammern eingeschlossene Zeichenfolge, die den Namen des Objekttyps enthält mit zusätzlichen Informationen, die häufig den Namen und die Adresse des Objekts enthalten.

Das bedeutet, dass `__str__` so implementiert werden könnte, dass etwas wie "Ass der Räume" zurückgegeben wird. `__repr__` könnte so implementiert werden, dass stattdessen `Card('Spades', 1)`

Diese Zeichenfolge könnte in einem " `eval` " direkt an `eval` :

```
object -> string -> object
```

Ein Beispiel für eine Implementierung einer solchen Methode könnte sein:

```
def __repr__(self):
    return "Card(%s, %d)" % (self.suit, self.pips)
```

Anmerkungen

[1] Diese Ausgabe ist implementierungsspezifisch. Die angezeigte Zeichenfolge stammt von cpython.

[2] Möglicherweise haben Sie das Ergebnis dieser `str()` / `repr()` `str()` bereits gesehen und nicht erkannt. Wenn Strings, die Sonderzeichen wie Backslashes enthalten, über `str()` in Strings

konvertiert werden, erscheinen die Backslashes so, wie sie sind (sie erscheinen einmal). Wenn sie über `repr()` in Strings konvertiert werden (z. B. als Elemente einer angezeigten Liste), werden die Backslashes mit Escapezeichen versehen und erscheinen daher zweimal.

Examples

Motivation

Sie haben also gerade Ihre erste Klasse in Python erstellt, eine nette kleine Klasse, die eine Spielkarte enthält:

```
class Card:
    def __init__(self, suit, pips):
        self.suit = suit
        self.pips = pips
```

An anderer Stelle in Ihrem Code erstellen Sie einige Instanzen dieser Klasse:

```
ace_of_spades = Card('Spades', 1)
four_of_clubs = Card('Clubs',  4)
six_of_hearts = Card('Hearts', 6)
```

Sie haben sogar eine Liste von Karten erstellt, um eine "Hand" darzustellen:

```
my_hand = [ace_of_spades, four_of_clubs, six_of_hearts]
```

Beim Debuggen möchten Sie nun sehen, wie Ihre Hand aussieht, also tun Sie, was natürlich kommt und schreiben

```
print(my_hand)
```

Aber was Sie zurückbekommen, ist ein Haufen Kauderwelsch:

```
[<__main__.Card instance at 0x0000000002533788>,
 <__main__.Card instance at 0x00000000025B95C8>,
 <__main__.Card instance at 0x00000000025FF508>]
```

Verwirrt versuchen Sie, nur eine einzige Karte zu drucken:

```
print(ace_of_spades)
```

Und wieder erhalten Sie diese seltsame Ausgabe:

```
<__main__.Card instance at 0x0000000002533788>
```

Hab keine Angst. Wir sind dabei, das zu beheben.

Zunächst ist es jedoch wichtig zu verstehen, was hier vor sich geht. Wenn Sie schreibt

`print(ace_of_spades)` sagten Sie Python man es will Informationen über die drucken `Card` Instanz Code ruft `ace_of_spades` . Und um fair zu sein, tat es.

Diese Ausgabe besteht aus zwei wichtigen Bits: dem `type` des Objekts und der `id` des Objekts. Allein der zweite Teil (die Hexadezimalzahl) reicht aus, um das Objekt zum Zeitpunkt des `print` eindeutig zu identifizieren. [1]

Was wirklich los war, war, dass Sie Python gebeten haben, die Essenz dieses Objekts "in Worte zu fassen" und es Ihnen dann anzuzeigen. Eine explizitere Version derselben Maschine könnte sein:

```
string_of_card = str(ace_of_spades)
print(string_of_card)
```

In der ersten Zeile versuchen Sie, Ihre `Card` in eine Zeichenfolge umzuwandeln, und in der zweiten zeigen Sie sie an.

Das Problem

Das Problem, auf das Sie stoßen, ist darauf zurückzuführen, dass Sie Python zwar alles erzählten, was Sie über die `Card` wissen mussten, *um* Karten zu *erstellen* , *aber nicht* , wie Sie `Card` in Zeichenketten konvertieren wollten.

Und da es nicht wusste, als Sie (implizit) `str(ace_of_spades)` , hat es Ihnen das gegeben, was Sie gesehen haben, eine generische Darstellung der `Card` Instanz.

Die Lösung (Teil 1)

Wir können Python *jedoch* mitteilen, wie Instanzen unserer benutzerdefinierten Klassen in Strings konvertiert werden sollen. Und so machen wir das mit der `__str__` __- `__str__` "dunder" (für doppelten Unterstrich) oder "magic".

Immer, wenn Sie Python `__str__` , eine Zeichenfolge aus einer Klasseninstanz zu erstellen, sucht es nach einer `__str__` Methode für die Klasse und ruft sie auf.

Betrachten Sie die folgende aktualisierte Version unserer `Card` Klasse:

```
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)
```

Hier haben wir nun die `__str__` Methode in unserer `Card` Klasse definiert, die nach einem einfachen Nachschlagen des Wörterbuchs für Bildkarten eine formatierte Zeichenfolge **zurückgibt** , je nachdem, wie wir uns entscheiden.

(Beachten Sie, dass "Returns" hier fett gedruckt ist, um zu `str(ace_of_spades)` , wie wichtig es ist, einen String zurückzugeben und nicht einfach nur zu drucken. Das Drucken scheint zwar zu funktionieren, aber Sie haben die Karte dann gedruckt, wenn Sie etwas wie `str(ace_of_spades)` , ohne dass in Ihrem Hauptprogramm sogar eine Funktion zum Drucken aufgerufen wird. `__str__` sicher, dass `__str__` eine Zeichenfolge zurückgibt.

Die `__str__` Methode ist eine Methode. Das erste Argument ist also `self` und sollte weder zusätzliche Argumente annehmen noch übergeben.

Um auf das Problem zurückzukommen, die Karte benutzerfreundlicher darzustellen, wenn wir erneut ausführen:

```
ace_of_spades = Card('Spades', 1)
print(ace_of_spades)
```

Wir werden sehen, dass unsere Ausgabe viel besser ist:

```
Ace of Spades
```

So toll, wir sind fertig, richtig?

Um unsere Grundlagen zu verdeutlichen, überprüfen wir noch einmal, ob wir das erste Problem gelöst haben und die Liste der `Card` , der `hand` , gedruckt haben.

Also überprüfen wir den folgenden Code noch einmal:

```
my_hand = [ace_of_spades, four_of_clubs, six_of_hearts]
print(my_hand)
```

Und zu unserer Überraschung bekommen wir wieder diese lustigen Hex-Codes:

```
[<__main__.Card instance at 0x00000000026F95C8>,
 <__main__.Card instance at 0x000000000273F4C8>,
 <__main__.Card instance at 0x0000000002732E08>]
```

Was ist los? Wir erzählten Python, wie wir wollten, dass unsere `Card` angezeigt werden. Warum schien sie anscheinend zu vergessen?

Die Lösung (Teil 2)

Die Maschinerie hinter den Kulissen ist etwas anders, wenn Python die Zeichenfolgendarstellung von Elementen in einer Liste erhalten möchte. Es stellt sich heraus, dass sich Python für diesen Zweck nicht für `__str__` .

Stattdessen sucht es nach einer anderen Methode, `__repr__` , und wenn *diese* nicht gefunden wird, greift sie auf das "Hexadezimal-Ding" zurück. [2]

Sie sagen also, ich muss zwei Methoden entwickeln, um dasselbe zu tun? Eine, wenn ich meine Karte alleine `print` möchte, und eine andere, wenn sie sich in einem Container befindet?

Nein, aber zuerst schauen wir uns an, wie unsere Klasse aussehen *würde* , wenn wir beide `__str__` und `__repr__` Methoden implementieren `__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)
```

Hier ist die Implementierung der beiden Methoden `__str__` und `__repr__` genau gleich, mit der Ausnahme, dass zur Unterscheidung der beiden Methoden `(S)` zu den von `__str__` Zeichenfolgen `__str__` und `(R)` zu den von `__repr__` Zeichenfolgen `__repr__` .

Beachten Sie, dass genau wie unsere `__str__` Methode, `__repr__` keine Argumente akzeptiert und **gibt** einen String **zurück.**

Wir können jetzt sehen, welche Methode für jeden Fall verantwortlich ist:

```
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)
```

Wie bedeckt war, die `__str__` wurde Methode aufgerufen , wenn wir unsere übergeben `Card` Instanz `print` und die `__repr__` Methode aufgerufen wurde , wenn wir *eine Liste unserer Instanzen* weitergegeben zu `print` .

An dieser Stelle sei darauf hingewiesen, dass wir, genau wie zuvor mit `str()` explizit einen String aus einer benutzerdefinierten Klasseninstanz erstellen können, auch explizit eine **String-Darstellung** unserer Klasse mit einer eingebauten Funktion namens `repr()` erstellen können. `repr()` .

Zum Beispiel:

```
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)
```

Außerdem *könnten* wir, wenn definiert, die Methoden direkt aufrufen (obwohl dies etwas unklar und unnötig erscheint):

```
print(four_of_clubs.__str__())     # 4 of Clubs (S)

print(four_of_clubs.__repr__())    # 4 of Clubs (R)
```

Über diese duplizierten Funktionen ...

Python-Entwickler erkannten, dass in dem Fall, dass identische Zeichenfolgen von `str()` und `repr()` sollen, Methoden möglicherweise funktional dupliziert werden müssen - etwas, das niemandem gefällt.

Stattdessen gibt es einen Mechanismus, um die Notwendigkeit dafür zu beseitigen. Einer, an dem ich dich bis hierher geschlichen habe. Es stellt sich heraus, dass, wenn eine Klasse die `__repr__` Methode *aber nicht* die `__str__` `__repr__` Methode implementiert und Sie eine Instanz dieser Klasse an `str()` (ob implizit oder explizit), Python auf Ihre `__repr__` Implementierung `__repr__` und diese verwendet.

Um klar zu sein, betrachten Sie die folgende Version der `Card` Klasse:

```
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)
```

Beachten Sie, dass diese Version *nur* die `__repr__` Methode implementiert. Aufrufe von `str()` führen jedoch zu der benutzerfreundlichen Version:

```
print(six_of_hearts)            # 6 of Hearts  (implicit conversion)
print(str(six_of_hearts))       # 6 of Hearts  (explicit conversion)
```

wie Aufrufe von `repr()` :

```
print([six_of_hearts])          #[6 of Hearts] (implicit conversion)
print(repr(six_of_hearts))      # 6 of Hearts  (explicit conversion)
```

Zusammenfassung

Damit Sie Ihre Klasseninstanzen dazu befähigen können, sich auf benutzerfreundliche Weise "zu zeigen", sollten Sie mindestens die `__repr__` Methode Ihrer Klasse `__repr__` . Wenn das Gedächtnis während eines Gesprächs dient, sagte Raymond Hettinger, dass die Sicherstellung, dass `__repr__` für Klassen implementiert wird, eines der ersten Dinge ist, nach denen er bei Python-Code-Reviews sucht, und jetzt sollte klar sein, warum. Die Menge an Informationen, die Sie zum Debuggen von Anweisungen, Absturzberichten oder Protokolldateien mit einer einfachen Methode hinzugefügt haben *könnten* , ist überwältigend, wenn Sie sie mit dem bloßen vergleichen, und oft weniger nützliche Informationen (Typ, ID), die standardmäßig bereitgestellt werden.

Wenn Sie *unterschiedliche* Repräsentationen für beispielsweise innerhalb eines Containers wünschen, sollten Sie sowohl die Methoden `__repr__` als auch `__str__` implementieren. (Mehr darüber, wie Sie diese beiden Methoden auf unterschiedliche Weise verwenden können).

Beide Methoden implementiert, Eval-Round-Trip-Stil __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)
```

Zeichenfolgendarstellungen von Klasseninstanzen: __str__- und __repr__-Methoden online lesen: https://riptutorial.com/de/python/topic/4845/zeichenfolgendarstellungen-von-klasseninstanzen----str----und---repr---methoden

Kapitel 204: Zufälliges Modul

Syntax

- random.seed (a = None, Version = 2) (Version ist nur für Python 3.x verfügbar)
- random.getstate ()
- random.setstate (state)
- zufällig.randint (a, b)
- random.randrange (stop)
- Random.randrange (Start, Stop, Schritt = 1)
- random.choice (seq)
- random.shuffle (x, random = random.random)
- random.sample (bevölkerung, k)

Examples

Zufall und Sequenzen: Mischen, Auswahl und Probe

```
import random
```

Mischen()

Sie können `random.shuffle()` , um die Elemente in einer **veränderbaren und indizierbaren** Sequenz zu mischen bzw. zu randomisieren. Zum Beispiel eine `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!
```

Wahl()

Nimmt ein zufälliges Element aus einer beliebigen **Reihenfolge** :

```
print(random.choice(laughs))
# Out: He                  # Output may vary!
```

Probe()

Wie bei der `choice` zufällige Elemente aus einer beliebigen **Reihenfolge übernommen,** aber Sie können angeben, wie viele

```
#                    |--sequence--|--number--|
print(random.sample(    laughs    ,     1    ))  # Take one element
# Out: ['Ho']                     # Output may vary!
```

Es wird nicht zweimal dasselbe Element verwendet:

```
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: Stichprobe größer als Grundgesamtheit

Erstellen von zufälligen Ganzzahlen und Floats: Randint, Randrange, Random und Uniform

```
import random
```

randint ()

Gibt eine zufällige ganze Zahl zwischen `x` und `y` (einschließlich) zurück:

```
random.randint(x, y)
```

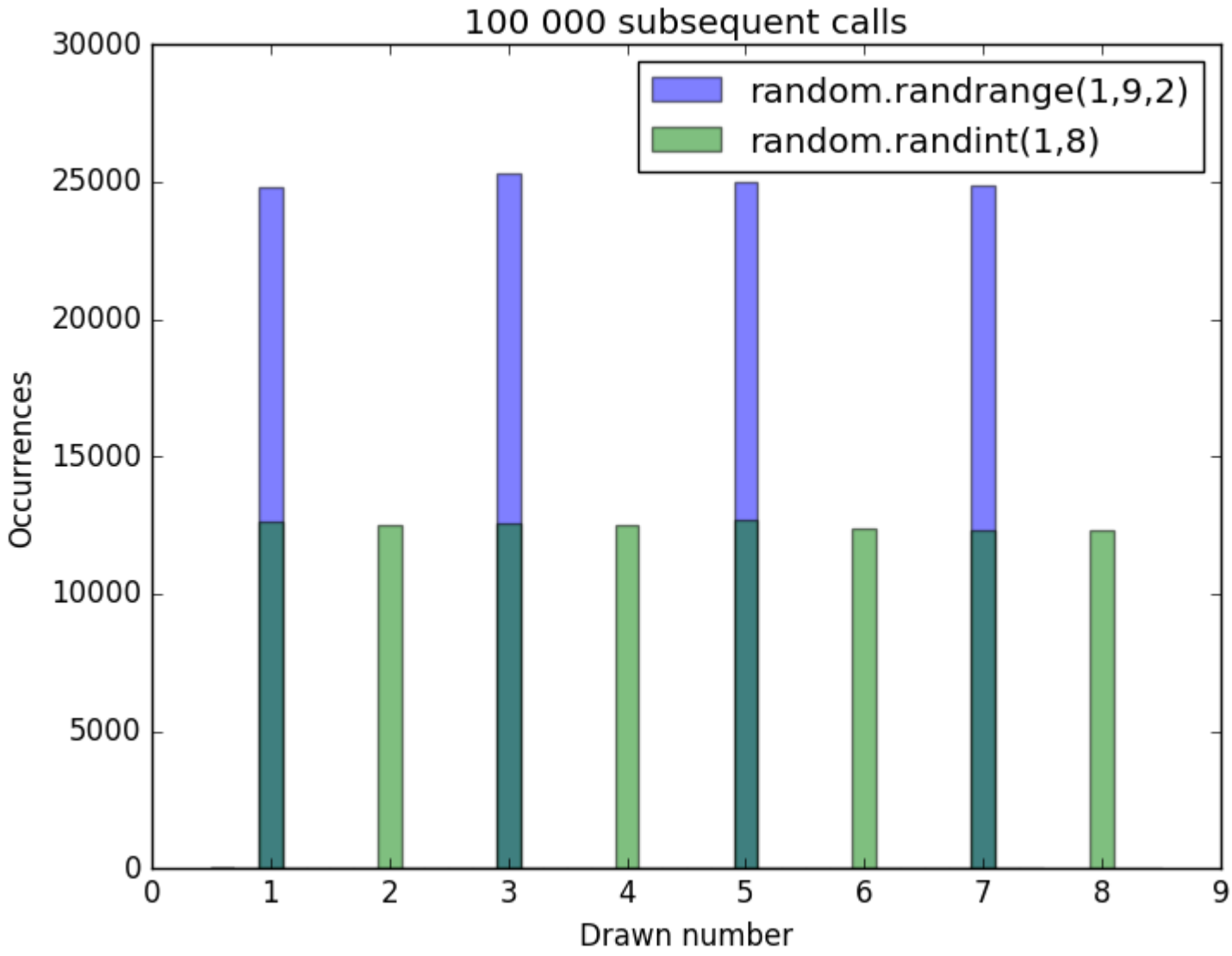
Zum Beispiel eine Zufallszahl zwischen `1` und `8` :

```
random.randint(1, 8) # Out: 8
```

randrange ()

`random.randrange` hat dieselbe Syntax wie `range` und anders als `random.randint` ist der letzte Wert **nicht** inklusive:

```
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)
```



zufällig

Gibt eine zufällige Gleitkommazahl zwischen 0 und 1 zurück:

```
random.random() # Out: 0.66486093215306317
```

Uniform

Gibt eine zufällige Gleitkommazahl zwischen `x` und `y` (einschließlich) zurück:

```
random.uniform(1, 8) # Out: 3.726062641730108
```

Reproduzierbare Zufallszahlen: Samen und Zustand

Durch das Festlegen eines bestimmten Seed wird eine feste Zufallszahlenreihe erstellt:

```
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
```

Durch das Zurücksetzen des Seed wird die gleiche "zufällige" Sequenz erneut erstellt:

```
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
```

Da der Samen fest ist, sind diese Ergebnisse immer `9` und `4` . Wenn es nicht erforderlich ist, bestimmte Zahlen zu verwenden, nur dass die Werte gleich sind, können Sie auch `getstate` und `setstate` , um den vorherigen Zustand wiederherzustellen:

```
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
```

Um die Sequenz erneut pseudozufällig zu machen, `seed` Sie mit `None` :

```
random.seed(None)
```

Oder rufen Sie die `seed` Methode ohne Argumente auf:

```
random.seed()
```

Erstellen Sie kryptografisch sichere Zufallszahlen

Das Python-Zufallsmodul verwendet standardmäßig das Mersenne Twister `PRNG` , um Zufallszahlen zu generieren, die zwar in Domänen wie Simulationen geeignet sind, aber den Sicherheitsanforderungen in anspruchsvolleren Umgebungen nicht genügen.

Um eine kryptographisch sichere Pseudozufallszahl zu erstellen, kann `SystemRandom` das mit `os.urandom` als kryptografisch sicherer Pseudozufallszahlengenerator (CPRNG) fungieren kann .

Der einfachste Weg zur Verwendung besteht darin, die `SystemRandom` Klasse zu initialisieren. Die zur Verfügung gestellten Methoden ähneln denen, die vom Zufallsmodul exportiert werden.

```
from random import SystemRandom
```

```
secure_rand_gen = SystemRandom()
```

Um eine zufällige Sequenz von 10 `int` s im Bereich `[0, 20]` zu erstellen, können Sie einfach `randrange()` aufrufen:

```
print([secure_rand_gen.randrange(10) for i in range(10)])
# [9, 6, 9, 2, 2, 3, 8, 0, 9, 9]
```

Um eine zufällige Ganzzahl in einem bestimmten Bereich zu erstellen, können Sie `randint` :

```
print(secure_rand_gen.randint(0, 20))
# 5
```

und entsprechend für alle anderen Methoden. Die Schnittstelle ist exakt gleich, die einzige Änderung ist der zugrunde liegende Zahlengenerator.

Sie können `os.urandom` direkt verwenden, um kryptografisch sichere Zufallsbytes zu erhalten.

Ein zufälliges Benutzerpasswort erstellen

Um ein zufälliges Benutzerpasswort zu erstellen, können wir die im `string` Modul enthaltenen Symbole verwenden. Speziell `punctuation` für Interpunktionssymbole, `ascii_letters` für Buchstaben und `digits` für Ziffern:

```
from string import punctuation, ascii_letters, digits
```

Wir können dann alle diese Symbole in einem Namen namens `symbols` kombinieren:

```
symbols = ascii_letters + digits + punctuation
```

Entfernen Sie eines dieser Elemente, um einen Pool von Symbolen mit weniger Elementen zu erstellen.

Danach können wir `random.SystemRandom` , um ein Passwort zu generieren. Für ein 10 langes Passwort:

```
secure_random = random.SystemRandom()
password = "".join(secure_random.choice(symbols) for i in range(10))
print(password)  # '^@g;J?]M6e'
```

Beachten Sie, **dass andere Routinen gemacht sofort verfügbar durch das `random` Modul - wie `random.choice` , `random.randint` usw. - sind *ungeeignet* für kryptographische Zwecke.**

Hinter diesen Vorhängen verwenden diese Routinen den Mersenne Twister PRNG , der die Anforderungen eines CSPRNG nicht erfüllt. Daher sollten Sie insbesondere keines davon verwenden, um Passwörter zu generieren, die Sie verwenden möchten. Verwenden Sie immer eine Instanz von `SystemRandom` wie oben gezeigt.

Python 3.x 3.6

Ab Python 3.6 steht das `secrets` Modul zur Verfügung, das kryptographisch sichere Funktionen bietet.

Zitieren Sie die offizielle Dokumentation , um *"ein* zehnstelliges *alphanumerisches Passwort mit mindestens einem Kleinbuchstaben, mindestens einem Großbuchstaben und mindestens drei Ziffern"* zu generieren.

```
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
```

Zufällige binäre Entscheidung

```
import random

probability = 0.3

if random.random() < probability:
    print("Decision with probability 0.3")
else:
    print("Decision with probability 0.7")
```

Zufälliges Modul online lesen: https://riptutorial.com/de/python/topic/239/zufalliges-modul

Kapitel 205: Zugriff auf Python-Quellcode und Bytecode

Examples

Zeigt den Bytecode einer Funktion an

Der Python-Interpreter kompiliert Code zu Bytecode, bevor er auf der virtuellen Maschine von Python ausgeführt wird (siehe auch Was ist Python-Bytecode?) .

So zeigen Sie den Bytecode einer Python-Funktion an

```
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)
```

Die Funktion `dis.dis` im dis-Modul gibt einen dekompilierten Bytecode der an sie übergebenen Funktion zurück.

Das Code-Objekt einer Funktion untersuchen

CPython ermöglicht den Zugriff auf das Codeobjekt für ein Funktionsobjekt.

Das `__code__` Objekt enthält den rohen Bytecode (`co_code`) der Funktion sowie andere Informationen wie Konstanten und Variablennamen.

```
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__)
```

Zeigen Sie den Quellcode eines Objekts an

Objekte, die nicht eingebaut sind

Um den Quellcode eines Python-Objekts zu drucken, verwenden Sie `inspect` . Beachten Sie, dass dies nicht für integrierte Objekte oder für interaktiv definierte Objekte funktioniert. Für diese benötigen Sie andere Methoden, die später erläutert werden.

So drucken Sie den Quellcode der Methode `randint` aus dem `random` :

```
import random
import inspect

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

So drucken Sie einfach die Dokumentationszeichenfolge

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

Vollständigen Pfad der Datei `random.randint` in der die Methode `random.randint` definiert ist:

```
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
```

Objekte interaktiv definiert

Wenn ein Objekt interaktiv definiert ist `inspect` kann `dill.source.getsource` den Quellcode nicht bereitstellen, stattdessen können Sie `dill.source.getsource` verwenden

```
# 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
```

Eingebaute Objekte

Der Quellcode für die in Python integrierten Funktionen ist in **c geschrieben** und kann nur über den Quellcode von Python (gehostet bei Mercurial oder unter https://www.python.org/downloads/source/) aufgerufen werden.

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

Zugriff auf Python-Quellcode und Bytecode online lesen:

https://riptutorial.com/de/python/topic/4351/zugriff-auf-python-quellcode-und-bytecode

Kapitel 206: zurückstellen

Einführung

Shelve ist ein Python-Modul zum Speichern von Objekten in einer Datei. Das Shelve-Modul implementiert einen permanenten Speicher für beliebige Python-Objekte, die mithilfe einer wörterbuchähnlichen API kommissioniert werden können. Das Shelve-Modul kann als einfache persistente Speicheroption für Python-Objekte verwendet werden, wenn eine relationale Datenbank überfordert ist. Das Regal ist wie bei einem Wörterbuch über Schlüssel zugänglich. Die Werte werden in eine Datenbank geschrieben, die von anydbm erstellt und verwaltet wird.

Bemerkungen

Hinweis: Verlassen Sie sich nicht darauf, dass das Regal automatisch geschlossen wird. Rufen Sie `close()` explizit auf, wenn Sie es nicht mehr benötigen, oder verwenden Sie `shelve.open()` als Kontextmanager:

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

Warnung:

Da das `shelve` mit `pickle` hinterlegt ist, ist es unsicher, ein Regal von einer nicht vertrauenswürdigen Quelle zu laden. Wie bei der Pickle kann das Laden eines Regals beliebigen Code ausführen.

Beschränkungen

1 Die Wahl des zu verwendenden Datenbankpakets (z. B. dbm.ndbm oder dbm.gnu) hängt von der verfügbaren Schnittstelle ab. Daher ist es nicht sicher, die Datenbank direkt mit dbm zu öffnen. Die Datenbank unterliegt (leider) den Einschränkungen von dbm, wenn sie verwendet wird. Dies bedeutet, dass die in der Datenbank gespeicherten Objekte (die angezeigte Darstellung von) relativ klein sein sollten. In seltenen Fällen kann es zu Kollisionen der Datenbank kommen Aktualisierungen ablehnen.

2 .Das Regalmodul unterstützt nicht den gleichzeitigen Lese- / Schreibzugriff auf abgelegte Objekte. (Mehrere gleichzeitige Lesezugriffe sind sicher.) Wenn ein Programm ein Regal zum Schreiben geöffnet hat, sollte es kein anderes Programm zum Lesen oder Schreiben geöffnet haben. Unix-Dateisperren können verwendet werden, um dieses Problem zu lösen. Dies ist jedoch bei den Unix-Versionen unterschiedlich und erfordert Kenntnisse über die verwendete Datenbankimplementierung.

Examples

Beispielcode für Regal

Um ein Objekt abzustellen, importieren Sie zuerst das Modul und weisen Sie den Objektwert folgendermaßen zu:

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

Um die Schnittstelle zusammenzufassen (Schlüssel ist eine Zeichenfolge, Daten ist ein beliebiges Objekt):

```
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
```

Ein neues Regal erstellen

Die einfachste Möglichkeit, Shelve zu verwenden, ist die Klasse **DbfilenameShelf** . Es verwendet anydbm, um die Daten zu speichern. Sie können die Klasse direkt verwenden oder einfach **shelve.open ()** aufrufen:

```
import shelve
```

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

Um erneut auf die Daten zuzugreifen, öffnen Sie das Regal und verwenden Sie es wie ein Wörterbuch:

```
    import shelve

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

print existing
```

Wenn Sie beide Beispielskripts ausführen, sollten Sie Folgendes sehen:

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

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

Das **dbm-** Modul unterstützt nicht mehrere Anwendungen, die gleichzeitig in dieselbe Datenbank schreiben. Wenn Sie wissen, dass Ihr Client das Regal nicht ändert, können Sie shelve anweisen, die Datenbank schreibgeschützt zu öffnen.

```
import shelve

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

print existing
```

Wenn Ihr Programm versucht, die Datenbank zu ändern, während es schreibgeschützt ist, wird eine Zugriffsfehlerausnahme generiert. Der Ausnahmetyp hängt vom Datenbankmodul ab, das von anydbm beim Erstellen der Datenbank ausgewählt wurde.

Schreib zurück

Regale verfolgen standardmäßig keine Änderungen an flüchtigen Objekten. Das heißt, wenn Sie den Inhalt eines Artikels ändern, der im Regal gespeichert ist, müssen Sie das Regal explizit aktualisieren, indem Sie den Artikel erneut speichern.

```
import shelve
```

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

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

In diesem Beispiel wird das Wörterbuch unter 'key1' nicht erneut gespeichert. Wenn das Regal erneut geöffnet wird, sind die Änderungen nicht erhalten.

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

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

Um Änderungen an flüchtigen Objekten, die im Regal gespeichert sind, automatisch abzufangen, öffnen Sie das Regal mit aktiviertem Rückschreiben. Das Rückschreib-Flag bewirkt, dass das Shelf alle von der Datenbank abgerufenen Objekte mit einem Cache-Speicher im Speicher speichert. Jedes Cache-Objekt wird auch in die Datenbank zurückgeschrieben, wenn das Shelf geschlossen wird.

```
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()
```

Obwohl es die Wahrscheinlichkeit von Programmierfehlern verringert und die Objektpersistenz transparenter machen kann, ist der Rückschreibmodus möglicherweise nicht in jeder Situation wünschenswert. Der Cache belegt zusätzlichen Speicher, während das Shelf geöffnet ist, und das Unterbrechen, um jedes zwischengespeicherte Objekt beim Schließen wieder in die Datenbank zu schreiben, kann zusätzliche Zeit in Anspruch nehmen. Da es keine Möglichkeit gibt, festzustellen, ob die zwischengespeicherten Objekte geändert wurden, werden sie alle zurückgeschrieben. Wenn Ihre Anwendung Daten mehr liest als schreibt, erhöht Writeback den Overhead, den Sie möglicherweise benötigen.

```
$ 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'}
```

zurückstellen online lesen: https://riptutorial.com/de/python/topic/10629/zuruckstellen

Credits

S. No	Kapitel	Contributors
1	Erste Schritte mit Python Language	A. Raza, Aaron Critchley, Abhishek Jain, AER, afeique, Akshay Kathpal, alejosocorro, Alessandro Trinca Tornidor, Alex Logan, ALinuxLover, Andrea, Andrii Abramov, Andy, Andy Hayden, angussidney, Ani Menon , Anthony Pham, Antoine Bolvy, Aquib Javed Khan, Ares , Arpit Solanki, B8vrede, Baaing Cow, baranskistad, Brian C, Bryan P, BSL-5, BusyAnt, Cbeb24404, ceruleus , ChaoticTwist, Charlie H, Chris Midgley, Christian Ternus, Claudiu, Clíodhna, CodenameLambda, 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, noɥʎd ʌzɐɹƆ, 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, Знаđошƒa, 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 und ** 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	2to3 Werkzeug	Alessandro Trinca Tornidor, Dartmouth, Firix, Kevin Brown, Naga2Raja, Stephen Leppik
4	Abstrakte Basisklassen (abc)	Akshat Mahajan, Alessandro Trinca Tornidor, JGreenwell, Kevin Brown, Mattew Whitt, mkrieger1, SashaZd, Stephen Leppik
5	Abstrakter Syntaxbaum	Teepeemm
6	Ähnlichkeiten in der Syntax, Bedeutungsunterschiede: Python vs. JavaScript	user2683246
7	Alternativen zum Wechseln von Anweisungen aus anderen Sprachen	davidism, J F, zmo, Валерий Павлов
8	ArcPy	Midavalo, PolyGeo, Zhanping Shi
9	Arrays	Andy, Pavan Nath, RamenChef, Vin
10	Asyncio-Modul	2Cubed, Alessandro Trinca Tornidor, Cimbali, hiro protagonist, obust, pylang, RamenChef, Seth M. Larson, Simplans, Stephen Leppik, Udi
11	Attribut-Zugriff	Elazar, SashaZd, SuperBiasedMan
12	Audio	blueberryfields, Comrade SparklePony, frankyjuang, jmunsch, orvi, qwertyuip9, Stephen Leppik, Thomas Gerot
13	Ausnahmen	Adrian Antunez, Alessandro Trinca Tornidor, Alfe, Andy, Benjamin Hodgson, Brian Rodriguez, BusyAnt, Claudiu, driax, Elazar, flazzarini, ghostarbeiter, Ilia Barahovski, J

		F, Marco Pashkov, muddyfish, noɥʎɹzɐɔ, Paul Weaver, Rahul Nair, RamenChef, Shawn Mehan, Shiven, Shkelqim Memolla, Simplans, Slickytail, Stephen Leppik, Sudip Bhandari, SuperBiasedMan, user2314737
14	Bedienmodul	MSeifert
15	Befehlszeilenargumente analysieren	amblina, Braiam, Claudiu, cledoux, Elazar, Gerard Roche, krato, loading..., Marco Pashkov, Or Duan, Pasha, RamenChef, rfkortekaas, Simplans, Thomas Gerot, Topperfalkon, zmo, zondo
16	Benutzerdefinierte Fehler / Ausnahmen auslösen	naren
17	Benutzerdefinierte Methoden	Alessandro Trinca Tornidor, Beall619, mnoronha, RamenChef, Stephen Leppik, Sun Qingyao
18	Binärdaten	Eleftheria, evuez, mnoronha
19	Bitweise Operatoren	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
20	Boolesche Operatoren	boboquack, Brett Cannon, Dair, Ffisegydd, John Zwinck, Severiano Jaramillo Quintanar, Steven Maude
21	ChemPy - Python-Paket	Biswa_9937
22	CLI-Unterbefehle mit präziser Hilfeausgabe	Alessandro Trinca Tornidor, anatoly techtonik, Darth Shadow
23	Codeblöcke, Ausführungsrahmen und Namespaces	Jeremy, Mohammed Salman
24	Commonwealth-Ausnahmen	Juan T, TemporalWolf
25	Conditionals	Andy Hayden, BusyAnt, Chris Larson, deepakkt, Delgan, Elazar, evuez, Ffisegydd, Geeklhem, Hannes Karppila, James, Kevin Brown, krato, Max Feng, noɥʎɹzɐɔ, rajah9, rrao, SashaZd, Simplans, Slayther, Soumendra Kumar Sahoo, Thomas Gerot, Trimax, Valentin Lorentz, Vinzee, wwii, xgord, Zack

26	configparser	Chinmay Hegde, Dunatotatos
27	CSV lesen und schreiben	Adam Matan, Franck Dernoncourt, Martin Valgur, mnoronha, ravigadila, Setu
28	ctypes	Or East
29	Das base64-Modul	Thomas Gerot
30	Das dis-Modul	muddyfish, user2314737
31	Das Ländereinstellungsmodul	Will, XonAether
32	Das os-Modul	Andy, Christian Ternus, JelmerS, JL Peyret, mnoronha, Vinzee
33	Dateien entpacken	andrew
34	Dateien und Ordner E / A	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
35	Daten kopieren	hashcode55, StuxCrystal
36	Datenbankzugriff	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
37	Datenserialisierung	Devesh Saini, Infinity, rfkortekaas
38	Datenserialisierung von Pickles	J F, Majid, Or East, RahulHP, rfkortekaas, zvone
39	Datenvisualisierung mit Python	Aquib Javed Khan, Arun, ChaoticTwist, cledoux, Ffisegydd, ifma
40	Datum und Uhrzeit	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

41	Datumsformatierung	surfthecity
42	Debuggen	Aldo, B8vrede, joel3000, Sardathrion, Sardorbek Imomaliev, Vlad Bezden
43	Dekorateure	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
44	Deque-Modul	Anthony Pham, BusyAnt, matsjoyce, ravigadila, Simplans, Thomas Ahle, user2314737
45	Der Dolmetscher (Befehlszeilenkonsole)	Aaron Christiansen, David, Elazar, Peter Shinners, ppperry
46	Designmuster	Charul, denvaar, djaszczurowski
47	Deskriptor	bbayles, cizixs, Nemo, pylang, SuperBiasedMan
48	Die Druckfunktion	Beall619, Frustrated, Justin, Leon Z., lukewrites, SuperBiasedMan, Valentin Lorentz
49	Die Pass-Anweisung	Anaphory
50	Die spezielle Variable __name__	Annonymous, BusyAnt, Christian Ternus, jonrsharpe, Lutz Prechelt, Steven Elliott
51	Django	code_geek, orvi
52	Dynamische Code-Ausführung mit "exec" und "eval"	Antti Haapala, Ilja Everilä
53	Eigenschaftsobjekte	Alessandro Trinca Tornidor, Darth Shadow, DhiaTN, J F, Jacques de Hooge, Leo, Martijn Pieters, mnoronha, Priya, RamenChef, Stephen Leppik
54	Einfache mathematische Operatoren	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
55	Einführung in RabbitMQ mit	eandersson

	AMQPStorm	
56	Einsatz	Gal Dreiman, Iancnorden, Wayne Werner
57	einstellen	Andrzej Pronobis, Andy Hayden, Bahrom, Cimbali, Cody Piersall, Conrad.Dean, Elazar, evuez, J F, James, Or East, pylang, RahulHP, RamenChef, Simplans, user2314737
58	Enum	Andy, Elazar, evuez, Martijn Pieters, techydesigner
59	Erste Schritte mit GZip	orvi
60	Erstellen eines Windows-Dienstes mit Python	Simon Hibbs
61	Erstellen Sie eine virtuelle Umgebung mit Virtualenvwrapper in Windows	Sirajus Salayhin
62	Erweiterungen schreiben	Dartmouth, J F, mattgathu, Nathan Osman, techydesigner, ygram
63	Externe Datendateien mit Pandas eingeben, unterteilen und ausgeben	Mark Miller
64	Filter	APerson, cfi, J Atkin, MSeifert, rajah9, SuperBiasedMan
65	Flasche	Stephen Leppik, Thomas Gerot
66	Functools-Modul	Alessandro Trinca Tornidor, enrico.bacis, flamenco, RamenChef, Shrey Gupta, Simplans, Stephen Leppik, StuxCrystal
67	Funktionale Programmierung in Python	Imran Bughio, mvis89, Rednivrug
68	Funktionen	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., noɥʎdʎzɐɔ, 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
69	Funktionen mit Listenargumenten definieren	zenlc2000
70	Generatoren	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
71	Graph-Werkzeug	xiaoyi
72	Grundflüche mit Python	4444, Guy, kollery, Vinzee
73	Grundlegende Eingabe und Ausgabe	Doraemon, GoatsWearHats, J F, JNat, Marco Pashkov, Mark Miller, Martijn Pieters, Nathaniel Ford, Nicolás, pcurry, pzp, SashaZd, SuperBiasedMan, Vilmar
74	gruppiere nach()	Parousia, Thomas Gerot
75	Hashlib	Mark Omo, xiaoyi
76	Häufige Fehler	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

77	Heapq	ettanany
78	HTML-Analyse	alecxe, talhasch
79	Ijson	Prem Narain
80	In CSV von String oder List schreiben	Hriddhi Dey, Thomas Crowley
81	Indizieren und Schneiden	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
82	Inkompatibilitäten von Python 2 zu 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, noɥʎdʎzɐƆ , 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
83	IoT-Programmierung mit Python und Himbeer-PI	dhimanta
84	Iterables und Iteratoren	4444, Conrad.Dean, demonplus, Ilia Barahovski, Pythonista
85	Itertools-Modul	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
86	JSON-Modul	Indradhanush Gupta, Leo, Martijn Pieters, pzp, theheadofabroom, Underyx, Wolfgang
87	Kartenfunktion	APerson, cfi, Igor Raush, Jon Ericson, Karl Knechtel, Marco Pashkov, MSeifert, noɥʞɟɑɟzɐɹƆ, Parousia, Simplans, SuperBiasedMan, tlama, user2314737
88	Kissen	Razik
89	kivy - Plattformübergreifendes Python-Framework für die NUI-Entwicklung	dhimanta
90	Klassen	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
91	Kommentare und	Ani Menon, FunkySayu, MattCorr, SuperBiasedMan,

	Dokumentation	TuringTux
92	Komplexe Mathematik	Adeel Ansari, Bosoneando, bpachev
93	Kontextmanager ("mit" - Anweisung)	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, λ user
94	Leistungsoptimierung	A. Ciclet, RamenChef, user2314737
95	List Destructuring (auch bekannt als Ein- und Auspacken)	J F, sth, zmo
96	Liste	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, noɥʌdʌzɐɹƆ, 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
97	Listen Sie Verständnis auf	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, noɥʎd ʎzɐɹƆ, 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
98	Listenaufteilung (Auswählen von Listenteilen)	Greg, JakeD
99	Listenverständnisse	3442, Akshit Soota, André Laszlo, Andy Hayden, Annonymous, Ari, Bhargav, Chris Mueller, Darth Shadow, Dartmouth, Delgan, enrico.bacis, Franck Dernoncourt, garg10may, intboolstring, Jeff Langemeier, Josh Caswell, JRodDynamite, justhalf, kdopen, Ken T, Kevin Brown, kiliantics, longyue0521, Martijn Pieters, Mattew Whitt, Moinuddin Quadri, MSeifert, muddyfish, noɥʎdʎzɐɹƆ, pktangyue, Pyth0nicPenguin, Rahul Nair, Riccardo Petraglia, SashaZd, shrishinde, Simplans, Slayther, sudo bangbang, theheadofabroom, then0rTh, Tim McNamara, Udi, Valentin Lorentz, Veedrac, Zags
100	Mathematik-Modul	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
101	Mehrdimensionale Arrays	boboquack, Buzz, rrao
102	Metaklassen	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
103	Mit ZIP-Archiven arbeiten	Chinmay Hegde, ghostarbeiter, Jeffrey Lin, SuperBiasedMan
104	Mixins	Doc, Rahul Nair, SashaZd
105	Module importieren	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
106	Müllsammlung	bogdanciobanu, Claudiu, Conrad.Dean, Elazar, FazeL, J F, James Elderfield, lukess, muddyfish, Sam Whited, SiggyF, Stephen Leppik, SuperBiasedMan, Xavier Combelle
107	Multiprocessing	Alon Alexander, Nander Speerstra, unutbu, Vinzee, Will
108	Multithreading	Alu, CLDSEED, juggernaut, Kevin Brown, Kristof, mattgathu , Nabeel Ahmed, nlsdfnbch, Rahul, Rahul Nair, Riccardo Petraglia, Thomas Gerot, Will, Yogendra Sharma
109	Mutable vs. Immutable (und Hashable) in Python	Cilyan
110	Neo4j und Cypher mit Py2Neo	Wingston Sharon
111	Nicht offizielle Python-Implementierungen	Jacques de Hooge, Squidward
112	Optische Zeichenerkennung	rassar

113	os.path	Claudiu, Fábio Perez, girish946, Jmills, Szabolcs Dombi, VJ Magar
114	Pandas-Transformation: Vorformung von Operationen in Gruppen und Verketten der Ergebnisse	Dee
115	Parallele Berechnung	Akshat Mahajan, Dair, Franck Dernoncourt, J F, Mahdi, nlsdfnbch, Ryan Smith, Vinzee, Xavier Combelle
116	pip: PyPI-Paketmanager	Andy, Arpit Solanki, Community, InitializeSahib, JNat, Mahdi, Majid, Matt Giltaji, Nathaniel Ford, Rápli András, SerialDev, Simplans, Steve Barnes, StuxCrystal, tlo
117	Plotten mit Matplotlib	Arun, user2314737
118	Plugin- und Erweiterungsklassen	2Cubed, proprefenetre, pylang, rrao, Simon Hibbs, Simplans
119	Polymorphismus	Benedict Bunting, DeepSpace, depperm, Simplans, skrrgwasme, Vinzee
120	PostgreSQL	Alessandro Trinca Tornidor, RamenChef, Stephen Leppik, user2027202827
121	Potenzierung	Anthony Pham, intboolstring, jtbandes, Luke Taylor, MSeifert, Pasha, supersam654
122	Profilierung	J F, keiv.fly, SashaZd
123	Protokollierung	Gal Dreiman, Jörn Hees, sxnwlfkk
124	Prozesse und Threads	Claudiu, Thomas Gerot
125	py.test	Andy, Claudiu, Ffisegydd, Kinifwyne, Matt Giltaji
126	pyaudio	Biswa_9937
127	Pyautogui-Modul	Damien, Rednivrug
128	Pygame	Anthony Pham, Aryaman Arora, Pavan Nath
129	Pyglet	Comrade SparklePony, Stephen Leppik
130	PyInstaller - Verteilen von Python-Code	ChaoticTwist, Eric, mnoronha
131	Python aus C # aufrufen	Julij Jegorov

132	Python Lex-Yacc	CLDSEED
133	Python mit SQL Server verbinden	metmirr
134	Python Parallelität	David Heyman, Faiz Halde, Iván Rodríguez Torres, J F, Thomas Moreau, Tyler Gubala
135	Python Requests Post	Ken Y-N, RandomHash
136	Python serielle Kommunikation (pyserial)	Alessandro Trinca Tornidor, Ani Menon, girish946, mnoronha, Saranjith, user2314737
137	Python Server - Gesendete Ereignisse	Nick Humrich
138	Python und Excel	bee-sting, Chinmay Hegde, GiantsLoveDeathMetal, hackvan, Majid, talhasch, user2314737, Will
139	Python-Anti-Patterns	Alessandro Trinca Tornidor, Annonymous, eenblam, Mahmoud Hashemi, RamenChef, Stephen Leppik
140	Python-Datentypen	Gavin, lorenzofeliz, Pike D., Rednivrug
141	Python-Geschwindigkeit des Programms	ADITYA, Antonio, Elodin, Neil A., Vinzee
142	Python-HTTP-Server	Arpit Solanki, J F, jmunsch, Justin Chadwell, Mark, MervS, orvi, quantummind, Raghav, RamenChef, Sachin Kalkur, Simplans, techydesigner
143	Python-Netzwerk	atayenel, ChaoticTwist, David, Geeklhem, mattgathu, mnoronha, thsecmaniac
144	Python-Pakete erstellen	Claudiu, KeyWeeUsr, Marco Pashkov, pylang, SuperBiasedMan, Thtu
145	Python-Persistenz	RamenChef, user2728397
146	Redewendungen	Benjamin Hodgson, Elazar, Faiz Halde, J F, Lee Netherton, loading..., Mister Mister
147	Reduzieren	APerson, Igor Raush, Martijn Pieters, MSeifert
148	Reguläre Ausdrücke (Regex)	Aidan, alejosocorro, andandandand, Andy Hayden, ashes999, B8vrede, Claudiu, Darth Shadow, driax, Fermi paradox, ganesh gadila, goodmami, Jan, Jeffrey Lin, jonrsharpe, Julien Spronck, Kevin Brown, Md.Sifatul Islam, Michael M., mnoronha, Nander Speerstra, nrusch, Or East, orvi, regnarg, sarvajeetsuman, Simplans, SN

		Ravichandran KR, SuperBiasedMan, user2314737, zondo
149	Rekursion	Bastian, japborst, JGreenwell, Jossie Calderon, mbomb007, SashaZd, Tyler Crompton
150	Sammlungen-Modul	asmeurer, Community, Elazar, jmunsch, kon psych, Marco Pashkov, MSeifert, RamenChef, Shawn Mehan, Simplans, Steven Maude, Symmitchry, void, XCoder Real
151	Schildkröte-Grafiken	Luca Van Oort, Stephen Leppik
152	Schleifen	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
153	setup.py	Adam Brenecki, amblina, JNat, ravigadila, strpeter, user2027202827, Y0da
154	Sichere Shell-Verbindung in Python	mnoronha, Shijo
155	Sicherheit und Kryptographie	adeora, ArtOfCode, BSL-5, Kevin Brown, matsjoyce, SuperBiasedMan, Thomas Gerot, Wladimir Palant, wrwrwr
156	Sockets und Nachrichtenverschlüsselung / Entschlüsselung zwischen Client und Server	Mohammad Julfikar
157	Sortierung, Minimum und Maximum	Antti Haapala, APerson, GoatsWearHats, Mirec Miskuf, MSeifert, RamenChef, Simplans, Valentin Lorentz
158	Sqlite3-Modul	Chinmay Hegde, Simplans
159	Stapel	ADITYA, boboquack, Chromium, cjds, depperm, Hannes Karppila, JGreenwell, Jonatan, kdopen, OliPro007, orvi,

		SashaZd, Snađoшfa, textshell, Thomas Ahle, user2314737
160	Steckdosen	David Cullen, Dev, MattCorr, nlsdfnbch, Rob H, StuxCrystal, textshell, Thomas Gerot, Will
161	String-Formatierung	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
162	String-Methoden	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
163	Subprozess-Bibliothek	Adam Matan, Andrew Schade, Brendan Abel, jfs, jmunsch, Riccardo Petraglia
164	Suchen	Dan Sanderson, Igor Raush, MSeifert
165	sys	blubberdiblub
166	Teilfunktionen	FrankBr
167	tempfile NamedTemporaryFile	Alessandro Trinca Tornidor, amblina, Kevin Brown, Stephen Leppik
168	tkinter	Dartmouth, rlee827, Thomas Gerot, TidB
169	Tupel	Anthony Pham, Antoine Bolvy, BusyAnt, Community,

		Elazar, James, Jim Fasarakis Hilliard, Joab Mendes, Majid, Md.Sifatul Islam, Mechanic, mezzode, nlsdfnbch, noɥʎdʎzɐɔ, Selcuk, Simplans, textshell, tobias_k, Tony Suffolk 66, user2314737
170	Typ Hinweise	alecxe, Annonymous, Antti Haapala, Elazar, Jim Fasarakis Hilliard, Jonatan, RamenChef, Seth M. Larson , Simplans, Stephen Leppik
171	Überlastung	Andy Hayden, Darth Shadow, ericmarkmartin, Ffisegydd , Igor Raush, Jonas S, jonrsharpe, L3viathan, Majid, RamenChef, Simplans, Valentin Lorentz
172	Überprüfen der Pfadexistenz und der Berechtigungen	Esteis, Marlon Abeykoon, mnoronha, PYPL
173	Überschreiben der Methode	DeepSpace, James
174	Umgang mit der Global Interpreter Lock (GIL)	Scott Mermelstein
175	Unicode	wim
176	Unicode und Bytes	Claudiu, KeyWeeUsr
177	Unit Testing	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
178	Unterschied zwischen Modul und Paket	DeepSpace, Simplans, tjohnson
179	Unveränderbare Datentypen (int, float, str, tuple und frozensets)	Alessandro Trinca Tornidor, FazeL, Ganesh K, RamenChef, Stephen Leppik
180	Urllib	Amitay Stern, ravigadila, sth, Will
181	Variabler Geltungsbereich und Bindung	Anthony Pham, davidism, Elazar, Esteis, Mike Driscoll, SuperBiasedMan, user2314737, zvone
182	Vergleiche	Anthony Pham, Ares, Elazar, J F, MSeifert, Shawn Mehan, SuperBiasedMan, Will, Xavier Combelle
183	Verknüpfte Listen	Nemo
184	Verknüpfter Listenknoten	orvi

185	Versteckte Funktionen	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
186	Verteilung	Alessandro Trinca Tornidor, JGreenwell, metahost, Pigman168, RamenChef, Stephen Leppik
187	Vertiefung	Alessandro Trinca Tornidor, depperm, J F, JGreenwell, Matt Giltaji, Pasha, RamenChef, Stephen Leppik
188	Verwenden von Schleifen innerhalb von Funktionen	naren
189	Verwendung des "pip" - Moduls: PyPI Package Manager	Zydnar
190	Virtuelle Python-Umgebung - virtualenv	Vikash Kumar Jain
191	virtuelle Umgebung mit Virtualenvwrapper	Sirajus Salayhin
192	Virtuelle Umgebungen	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
193	Vorlagen in Python	4444, Alessandro Trinca Tornidor, Fred Barclay, RamenChef, Ricardo, Stephen Leppik
194	Vorrang des Bedieners	HoverHell, JGreenwell, MathSquared, SashaZd, Shreyash S Sarnayak
195	Warteschlangenmodul	Prem Narain
196	Webbrowser-Modul	Thomas Gerot
197	Web-Scraping mit Python	alecxe, Amitay Stern, jmunsch, mrtuovinen, Ni., RamenChef, Saiful Azad, Saqib Shamsi, Simplans,

		Steven Maude, sth, sytech, talhasch, Thomas Gerot
198	Webserver-Gateway-Schnittstelle (WSGI)	David Heyman, Kevin Brown, Preston, techydesigner
199	Websockets	2Cubed, Stephen Leppik, Tyler Gubala
200	Wörterbuch	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, no ɥʞʎɑʎzɐɹƆ, ronrest, Roy Iacob, Shawn Mehan, Simplans, SuperBiasedMan, TehTris, Valentin Lorentz, viveksyngh , Xavier Combelle
201	XML bearbeiten	4444, Brad Larson, Chinmay Hegde, Francisco Guimaraes, greuze, heyhey2k, Rob Murray
202	Zählen	Andy Hayden, MSeifert, Peter Mølgaard Pallesen, pylang
203	Zeichenfolgendarstellungen von Klasseninstanzen: __str__- und __repr__-Methoden	Alessandro Trinca Tornidor, jedwards, JelmerS, RamenChef, Stephen Leppik
204	Zufälliges Modul	Alex Gaynor, Andrzej Pronobis, Anthony Pham, Community, David Robinson, Delgan, giucal, Jim Fasarakis Hilliard, michaelrbock, MSeifert, Nobilis, ppperry, RamenChef, Simplans, SuperBiasedMan
205	Zugriff auf Python-Quellcode und Bytecode	muddyfish, StuxCrystal, user2314737
206	zurückstellen	Biswa_9937