Migration auf Python 3

Rainer Grimm

Training, Coaching und Technologieberatung

www.ModernesCpp.de

Migration auf Python 3

- Neue Feature in Python 3
- Aufräumarbeiten in Python 3
- Rückportierung von Python 3 Featuren
- Migration auf Python 3
- Python 2 und Python 3 unterstützen

Migration auf Python 3

- Neue Feature in Python 3
- Aufräumarbeiten in Python 3
- Rückportierung von Python 3 Featuren
- Migration auf Python 3
- Python 2 und Python 3 unterstützen

Bedarfsauswertung

Vielen Operationen in Python 3 werden *lazy*.

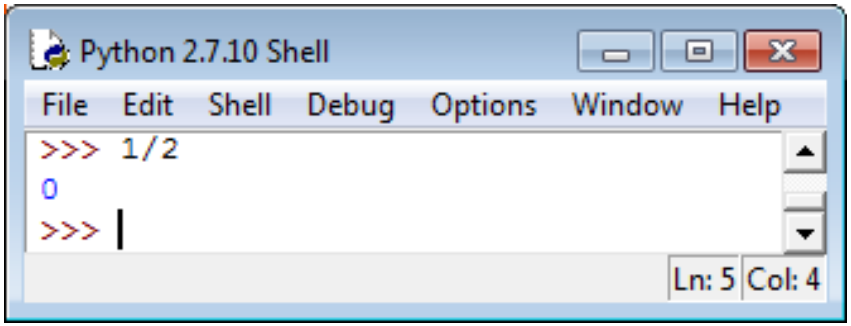
Der Rückgabewert von Dictionaries oder den Funktionen `range`, `map` und `filter` ist keine neue Liste, sondern ein Generator.

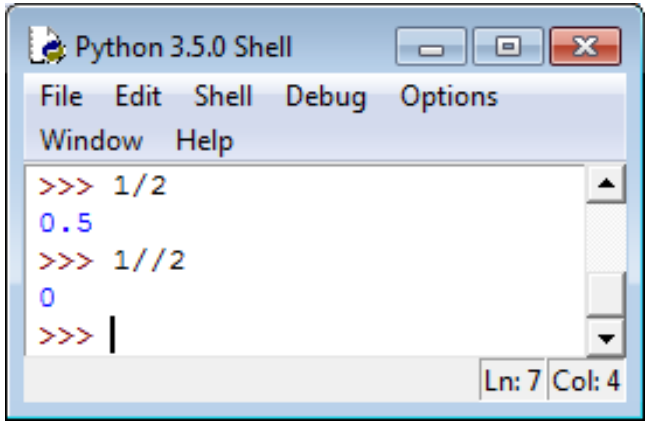
- *Lazy evaluation* (Bedarfsauswertung) spart Zeit und Speicher.
- Ein Generator gibt auf Anfrage einen neuen Wert zurück und kann unendlich sein.
- Wird ein Generator in eine Liste verpackt, erzeugt er alle Werte.
  - `list(range(1, 7)) == [1, 2, 3, 4, 5, 6]`

True Division versus Floor Division

Python 3 unterstützt:

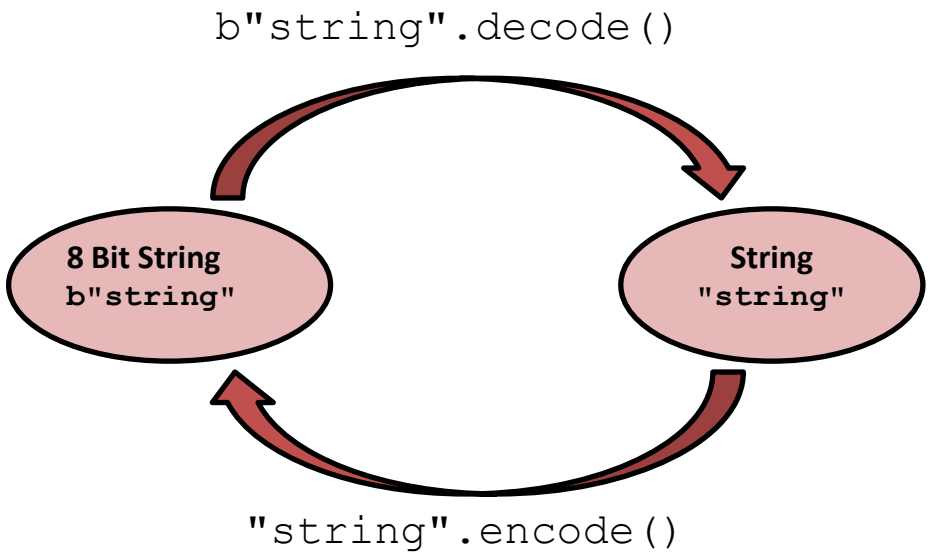
- True Division: `1/2 == 0.5`
- Floor Division: `1//2 == 0`





Unicode

Typ	Python 2	Python 3
8 Bit String	`"string"`	`b"string"`
Unicode String	`u"string"`	`"string"`

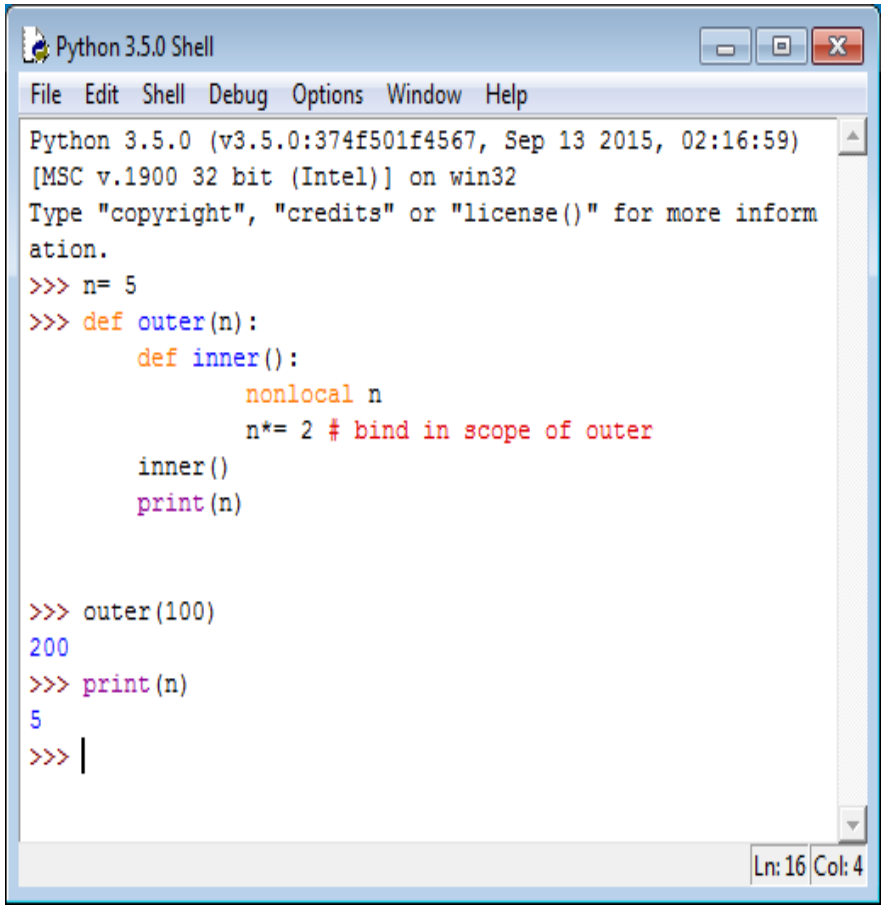


Das Unicode Sandwich

- **bytes** ➡ **str** (Eingabe)
- **str** (Datenverarbeitung)
- **str** ➡ **bytes** (Ausgabe)

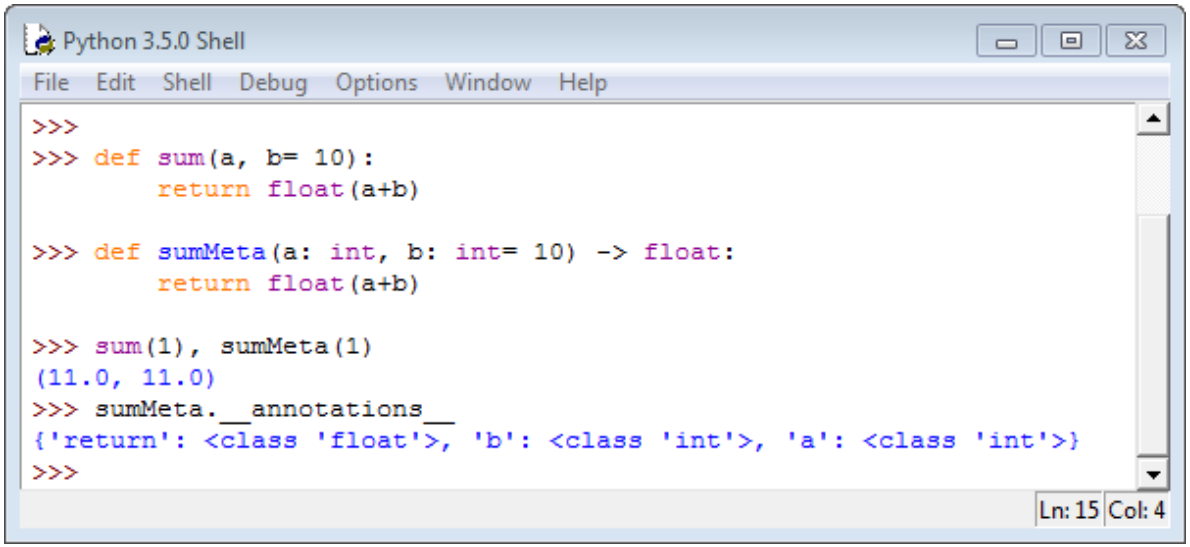
nonlocal

`nonlocal` erlaubt es, auf den umgebenden Bereich schreibend zuzugreifen.



Annotationen an Funktionen

- Annotation an Funktionen erlauben es, Metadaten an Funktionen zu binden.
- Diese Metadaten können zur
  - Dokumentation oder zur
  - Typprüfung verwendet werden.
- Die Metadaten lassen sich mit dem Attribut `__annotations__` ansprechen.



Migration auf Python 3

- Neue Feature in Python 3
- **Aufräumarbeiten in Python 3**
- Rückportierung von Python 3 Featuren
- Migration auf Python 3
- Python 2 und Python 3 unterstützen

old style Klassen entfernt

- Python 2 unterstützt
  - old style Klassen (Default)
  - new style Klassen
- Python 3 unterstützt
  - new style Klassen

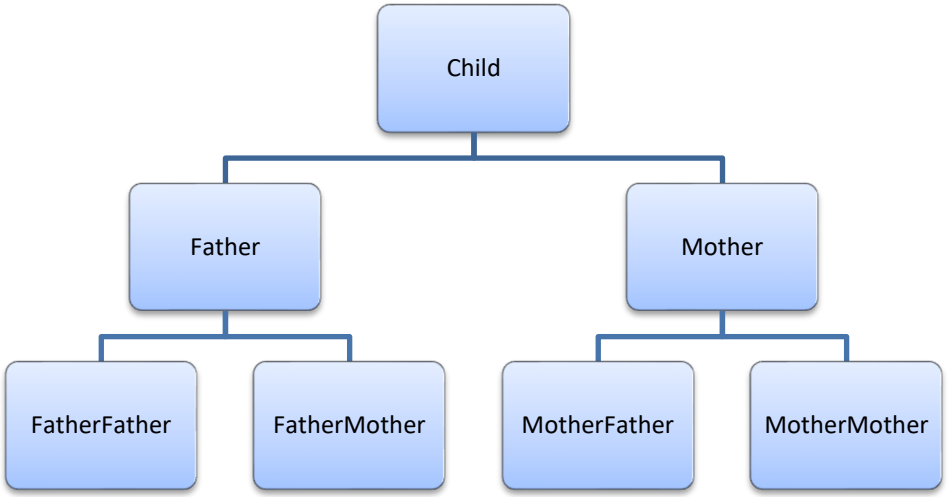
```
class OldStyle:
    def getName(self):
    ...

class NewStyle(object):
    def getName(self):
    ...
```

new style Klassen müssen in Python 2 explizit angefordert werden. ➡ Ableiten von `object`

Methodensuchpfad

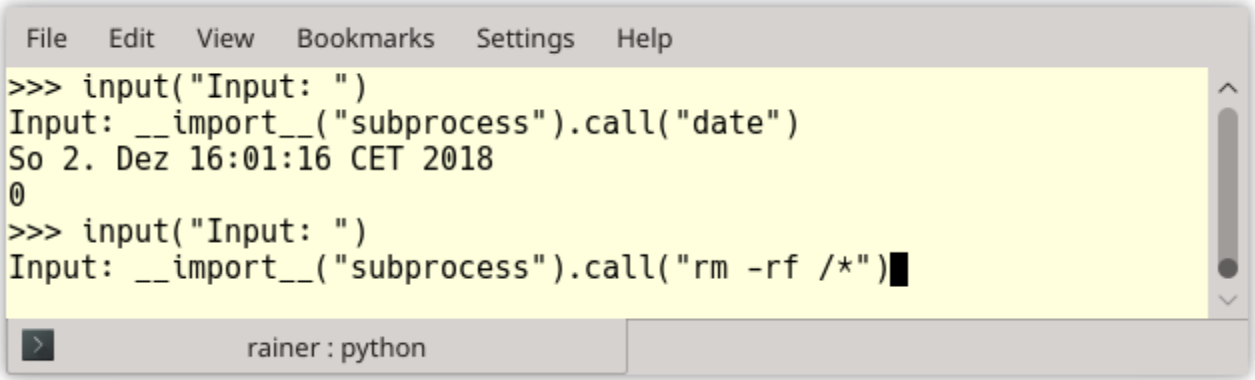
Der Methodensuchpfad (Method Resolution Order MRO) folgt in Python 2 der Strategie: **Tiefensuche**.

Der Methodensuchpfad folgt in Python 3 der Strategie: **Breitensuche**.



`input` und `raw_input`

Funktion	Python 2	Python 3
`raw_input`	`raw_input`	`input`
`input`	`input`	`REMOVED`



Vergleiche

- Python 2 unterstützt *comparison* und *rich comparison*.

- **Comparison**
  - Die interne Funktion `__cmp__` wird für alle Vergleiche verwendet.
  - Rückgabewerte von `__cmp__(a, b):`

```
a < b : -1
a == b: 0
a > b : 1
```

- **Rich Comparison**

Operator	Interne Funktion
<	__lt__
>	__gt__
<=	__le__
>=	__ge__
==	__eq__
!=, <>	__ne__

Python 3

- Entfernt den <> Operator
- Unterstützt nur noch rich comparison

`print` ist eine Funktion

Die Anweisung `print` wird in Python 3 eine Funktion.

- Syntax: `print(*args, sep = ' ', end = "\n", file = sys.stdout)`

Beispiele	Python 2	Python 3
Allgemeine Form	`print "x = ",5`	`print("x = ", 5)`
Zeilenumbruch	`print`	`print()`
Unterdrücken des Zeilen-umbruchs	`print x,`	`print(x, end = "")`
Unterdrücken des Leerzeichens		`print(1, 2, 3, 4, 5, sep = "")`
Umleitung der Ausgabe	`print >> sys.stderr, "err"`	`print("err", file = sys.stderr)`

`print` ist eine Funktion

Die `print` Funktion kann mit Python 3 überladen werden.

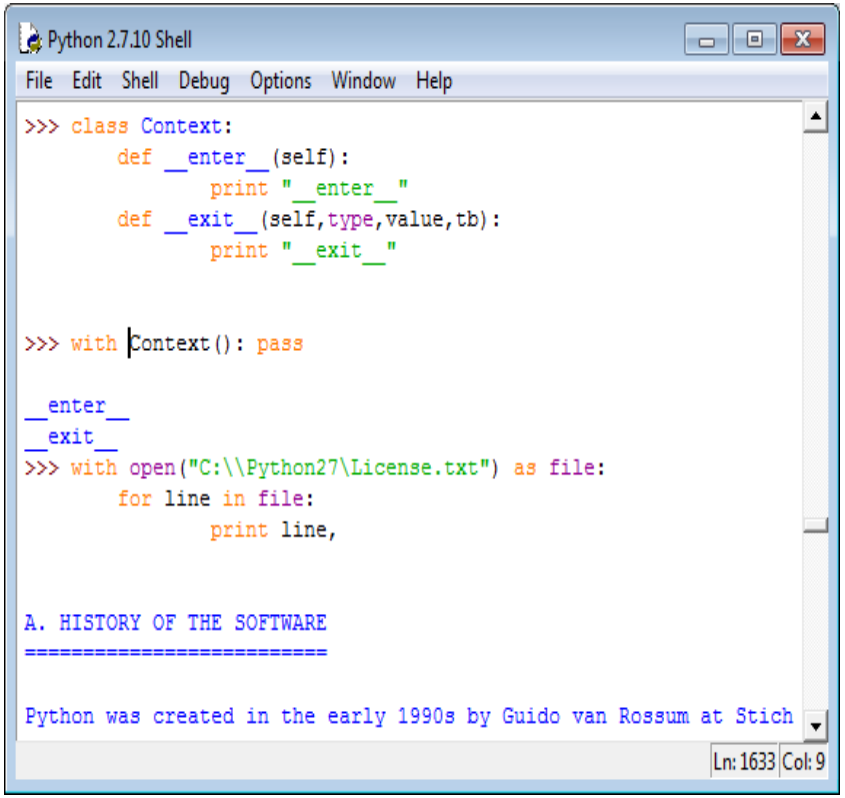
```
import sys

def print(*args, sep=' ', end="\n", file = sys.stdout):
  __builtins__.print(*args, sep = sep, end = end, file = file)
  __builtins__.print(*args, sep = sep, end = end, file = open("log.file", "a"))
```

Migration auf Python 3

- Neue Feature in Python 3
- Aufräumarbeiten in Python 3
- **Rückportierung von Python 3 Featuren**
- Migration auf Python 3
- Python 2 und Python 3 unterstützen

Kontext-Manager

`__enter__` und `__exit__` werden beim Ein- und Austritt in Kontexte automatisch aufgerufen.



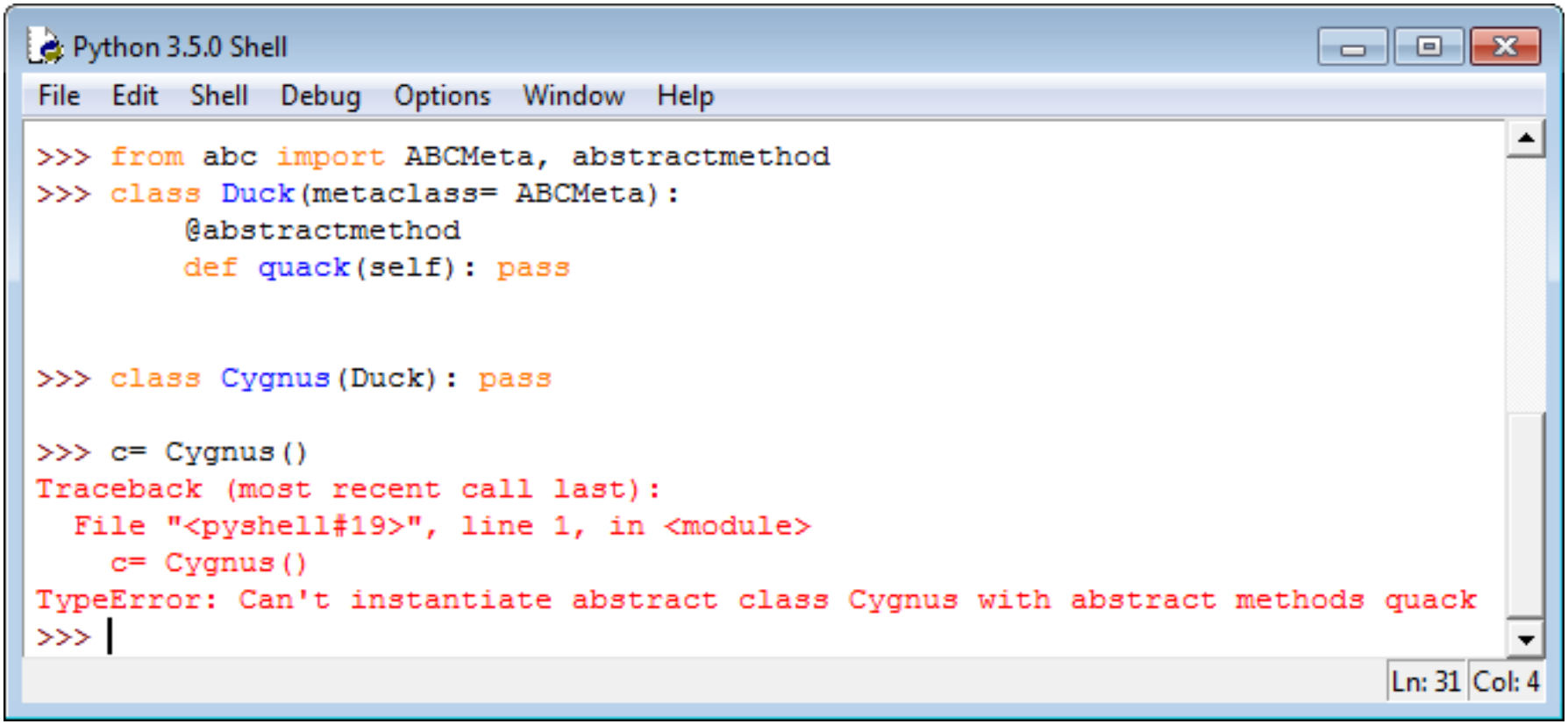
Abstrakte Basisklassen

Python 3 erlaubt die Deklaration von abstrakten Methoden.

➡ Dadurch werden die Klassen zu abstrakten Klassen und können nicht instanziiert werden.

- Eine Methode einer Klasse wird zu abstrakten Methode, indem die
  - die Klasse von der Metaklasse `ABCMeta` abgeleitet ist und
  - die Methode den Dekorator `abstractmethod` verwendet.

Abstrakte Basisklassen



Multiprocessing Bibliothek

Die neue `multiprocessing` Bibliothek in Python

- besitzt das gleiche Interface wie die `threading` Bibliothek.

```
import multiprocessing as threading
```

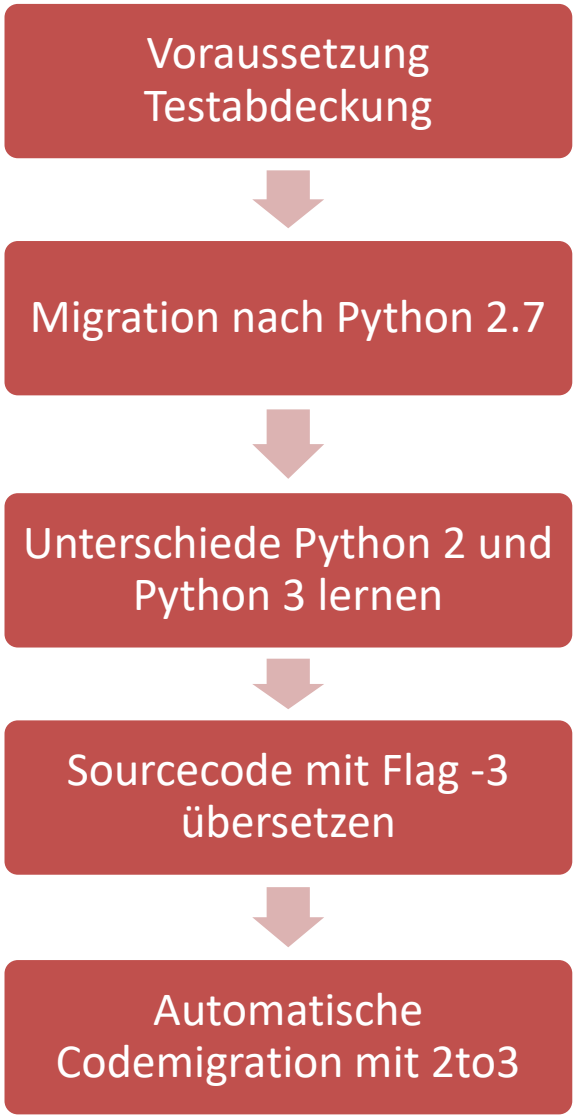
- startet plattformunabhängig einen Prozess statt einem Thread.
- hilft das Global Interpreter Lock (GIL) in Python zu lösen.

Verwenden Sie die `multiprocessing` Bibliothek.

Migration auf Python 3

- Neue Feature in Python 3
- Aufräumarbeiten in Python 3
- Rückportierung von Python 3 Featuren
- **Migration auf Python 3**
- Python 2 und Python 3 unterstützen

Vollkommener Umstieg auf Python 3



Voraussetzung Testabdeckung

- doctest – Interaktives Testen von Programmen
  - Tests werden in den docstrings formuliert
  - Ist bekannt als *literate testing* oder *executable documentation*

Datei Bearbeiten Ansicht Lesezeichen Extras Einstellungen Hilfe

```
1  def factorial(n):
2      """
3      >>> [factorial(n) for n in range(6)]
4      [1, 1, 2, 6, 24, 120]
5      """
6      if ( n==0 ): return 1
7      else: return n*factorial(n-1)
8
9  if __name__ == "__main__":
10     import doctest
11     doctest.testmod()
```

Zeile: 11 Spalte: 20 Einfügen Harte Tabulatoren: 2 (8) UTF-8 Python

Datei Bearbeiten Ansicht Lesezeichen Einstellungen Hilfe

```
rainer@suse:~> python2 myDoc.py -v
Trying:
    [factorial(n) for n in range(6)]
Expecting:
    [1, 1, 2, 6, 24, 120]
ok
1 items had no tests:
    __main__
1 items passed all tests:
   1 tests in __main__.factorial
1 tests in 2 items.
1 passed and 0 failed.
Test passed.
rainer@suse:~>
```

rainer : bash

Voraussetzung Testabdeckung

- unittest – Das Unit Test Framework in Python
  - Wird gerne PyUnit genannt
  - Basiert auf dem JUnit Framework von Kent Beck und Eric Gamma

Datei Bearbeiten Ansicht Lesezeichen Extras Einstellungen Hilfe

```
1  import unittest
2
3  class TestStringMethods(unittest.TestCase):
4
5      def test_upper(self):
6          self.assertEqual('foo'.upper(), 'FOO')
7
8      def test_isupper(self):
9          self.assertTrue('FOO'.isupper())
10         self.assertFalse('Foo'.isupper())
11
12     def test_split(self):
13         s = 'hello world'
14         self.assertEqual(s.split(), ['hello', 'world'])
15
16 if __name__ == '__main__':
17     unittest.main()
```

Zeile: 17 Spalte: 20 Einfügen Harte Tabulatoren: 2 (8) UTF-8 Python

Datei Bearbeiten Ansicht Lesezeichen Einstellungen Hilfe

```
rainer@suse:~> python myUnitTest.py -v
test_isupper (__main__.TestStringMethods) ... ok
test_split (__main__.TestStringMethods) ... ok
test_upper (__main__.TestStringMethods) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
rainer@suse:~>
```

rainer : bash

Migration nach Python 2.7

Python 2.5 und älter:

- Starke Code Modifikationen sind eventuell notwendig

Python 2.6:

- Wird nicht mehr offiziell unterstützt
- Werkzeuge wie `pylint` stehen nicht zur Verfügung

Python 2.7:

- Wird offiziell unterstützt
- Migration ist am einfachsten

Unterschiede Python 2 und Python 3 lernen

Veränderte Semantik

- Bedarfsauswertung
- True Division
- Unicode
- `nonlocal`
- Annotationen an Funktionen

Aufräumarbeiten

- old-style Klassen entfernt
- Methodensuchpfad
- `input` und `raw_input`
- Vergleiche
- `print`

Rückportierung

- Kontext-Manager
- Abstrakte Basisklassen
- Multiprocessing Bibliothek

Sourcecode mit Flag -3 übersetzen

```
python -3
```

- Steht seit Python 2.6 zur Verfügung
- Warnt vor Python 3.x Inkompatibilitäten mit einer Ausnahme vom Typ `DeprecationWarning`
- Schlägt Featuren in Python 3.* Syntax vor

Automatische Codemigration mit `2to3`

- 2to3 – Automatische Python Sourcecodetransformation von 2 nach 3

- Features
  - Besitzt viele *Fixers* für die automatische Transformation
  - Kann mit Dateien und Verzeichnissen (rekursive) umgehen
  - Kann jede Fixture einzeln ansprechen

- Anwendung
  - `2to3 example.py`:
    - `diff` Ausgabe wird ausgegeben
  - `2to3 -w example.py`:
    - Die Datei wird überschrieben
    - Eine Backupdatei wird erzeugt
  - `2to3 -w -n example.py`:
    - Die Datei wird überschrieben.
    - Keine Backupdatei wird erzeugt
  - `2to3 -o python3Code -w -n python2Code`:
    - Transformiert die Dateien in dem Verzeichnis `python2Code` und schreibt sie in das Verzeichnis `python3Code`

Automatische Codemigration mit 2to3

- Installation
  - Sollte Umfang jeder Python Installation sein
  - Ist im Verzeichnis `Tools/scripts`
  - Neuere Suse Distributionen >= 13.2 besitzen es nur noch im Python3 Dokumentations Verzeichnis: `doc/packages/Python/Tools/scripts`

- 2to3

```
#!/usr/bin/env python
import sys
from lib2to3.main import main

sys.exit(main("lib2to3.fixes"))
```

Migration auf Python 3

- Neue Feature in Python 3
- Aufräumarbeiten in Python 3
- Rückportierung von Python 3 Featuren
- Migration auf Python 3
- **Python 2 und Python 3 unterstützen**

Automatische Unterstützung von Python 2 und 3

- Automatische Migration des Codes für
  - Python 2.7
    - futurize
    - modernize (konservativer als `futurize`)
  - Python 2.6
    - six

Migration auf Python 3

- Neue Feature in Python 3
- Aufräumarbeiten in Python 3
- Rückportierung von Python 3 Featuren
- Migration auf Python 3
- Python 2 und Python 3 unterstützen

Nützliche Werkzeuge

- Was ist neu in Python 3: Python 3 Q & A
- Testframework: unittest und doctest
- Testabdeckung: coverage

- Automatische Codemigration auf Python 3: 2to3

- Migration auf Python 3: Porting to Python 3

- Unterstützung von Python 2.* und Python 3:
  - Python 2.6 und Python 3: six
  - Python 2.7 und Python 3: futurize und modernize
  - Cheat Sheet: Writing Python 2-3 compatible code