



Python Entwicklung

Johannes Hellrich & Tobias Kolditz

Jena University Language & Information Engineering (JULIE) Lab
Friedrich Schiller University Jena,
Jena, Germany

<http://www.julielab.de>

DISCLAIMER:

Basiert teilweise auf Recherche, nicht auf Erfahrung!

Alles Cpython 3.6

Mehr Beispiele in Quellcode

Gliederung

- Syntax
- Testing
- Bibliotheken

Input / Output

- Gilt für `open()` und `Path.open()`
- Dateien werden per default als text behandelt (ausschaltbar per `mode=b`) und Inhalte je nach Vorgabe en-/decoded
- Buffering ist per default aus, an mit `buffering=size`
- `encoding` nutzt Plattformdefault bei lesen/schreiben, besser setzen mit z.B. `encoding='utf8'` (Offsetfehler!)
- `newline` ist per default `None` und damit “universal”, also liest/schreibt Plattform, zeigt intern aber ‘\n’ (Offsetfehler!)
- <https://docs.python.org/3/library/functions.html#open>

with

- Automatisches Schließen von Ressourcen

```
with open('foo') as foo_file:  
    for line in foo_file:  
        print(line)
```

- Ressourcen müssen Context Manager Methoden haben
(`__enter__()` & `__exit__()`, typisch in Bibliotheken)
- Geht auch für mehrere Kontexte

```
with open('foo') as foo, open('bar') as bar:
```

__init__.py

- Packages (Ordner) mit Modulen (filename.py) brauchen eine __init__.py Datei
- Unter bestimmten Bedingungen weglassbar, aber kann zu Problemen führen <https://www.python.org/dev/peps/pep-0420/>
- Kann selber z.B. import (kürzere effektive Namespaces) oder Konfiguration (Logger) beinhalten

Comprehensions

- Kompakt und **performant**
- Für verschiedene Datenstrukturen (list, set, dict, generator)
- Können Kontrollstrukturen beinhalten
- Beispiel:

```
[x+1 for x in [1, 2, 3]] == [2, 3, 4]
```

List-Comprehension Vergleich

Neue Elemente Iterable

Funktionale Programmierung

- Comprehensions sind eine Form und bevorzugt
- Funktionen als Objekt & Lambdas:

```
addOne = lambda x: x+1
```

- Test der Elemente von Iterables auf Wahrheit: any/all
- itertools erlauben z.B. Kombinationen/Permutationen
- Siehe <https://docs.python.org/3.6/howto/functional.html>

Klassen

- Definiert über **class**
 - Syntax: `class Beispiel:`
- Vererbung durch Überklassen in Klammern
 - Syntax: `class Beispiel(Mama):`
- Multiple Vererbung unterstützt, Auflösung links->rechts
 - Syntax: `class Beispiel(Mama, Papa):`
- Keine Interfaces, aber abstrakte Klassen über **abc**
- Siehe <https://docs.python.org/3.6/tutorial/classes.html>

Methoden

- Funktionen die innerhalb einer Klasse definiert werden gehören zu dieser
- 1. Argument muss `cls/self` sein und wird automatisch übergeben
- Dekoratoren:
 - `@classmethod` für alternative Konstruktoren (vererbbar!)
 - `@staticmethod` als Alternative zu Funktion in Modul Skopus

Datenklassen

- Ziel: Daten strukturiert speichern/abrufen mit „angenehmer“ Syntax (vgl. C struct)
- “normale“ Klassen mit automatischer `__init__` usw.
 - Attr
 - Dataclasses (nur 3.7+)
- Tupel mit klassenartigem Interface
 - namedtuple
 - NamedTuple

Underscores

- Underscores am Anfang von Namen sind ein schwaches Gegenstück zu Javas `private`
- `_ein_underscore` für modul-interne Funktionen und Variablen, wird bei `import *` nicht importiert
- `--zwei_underscores` für Attribute und Methoden einer Klasse die bei Vererbung nicht überschrieben werden (Namen werden von Kompiler modifiziert)

Magische Attribute

- Grundlage vieler APIs
- Teilweise nicht in Instanzen, sondern Klassen definiert, Zugriff über Klasse.X oder type(Instanz).X
- Beispiele:
 - __name__ Name einer Klasse/Funktion (nicht in Instanz)
 - __dict__ Dictionary mit Attributen/Methoden
 - __doc__ Eingebettete Dokumentation
- Siehe auch z.B. <https://docs.python.org/3.6/reference/datamodel.html>

Magische Methoden

- Grundlage vieler APIs, in Klassen per `def` definiert
- Beispiele:
 - `__init__`(`self`) Konstruktor
 - `__str__`(`self`) entspricht etwa Java `toString()`
 - `__repr__`(`self`) entspricht etwa Java `toString()`
 - `__cmp__`(`self, other`) für Vergleiche
 - `__hash__`(`self`) für Hashing
 - `__getitem__`(`self, key`) Zugriff per `instanz[key]`
 - Überladung aller typischen Operatoren, z.B. `__add__` für “+“
- Siehe auch z.B. <https://rszalski.github.io/magicmethods/> oder <https://docs.python.org/3.6/reference/datamodel.html>

Dekoratoren

- Entsprechen Java Annotationen
- Syntaktischer Zucker über Wrapperfunktionen/-Klassen
- Siehe auch z.B. <https://www.thecodeship.com/patterns/guide-to-python-function-decorators/>

Gliederung

- Syntax
- Testing
- Bibliotheken

Generell

- Beispiele zeigen Aufruf der Tests als Skripte, in Praxis über `python -m TESTFRAMEWORK GETESTET.py`
- `assert` im eigentlichen Code ist kein Test, sondern garantiert Validität von Input und kann per Kompilerflag ignoriert werden: `python -O`

Doctest

- In die Dokumentation eingebettete Beispiele werden als Test genutzt
- Teil der Standardbibliothek
- <https://docs.python.org/3.6/library/doctest.html>

Unittest

- Teil der Standardbibliothek
- JUnit inspiriert → Klassen mit Test Methoden
- Tests über `self.assertEqual` (usw.)
- <https://docs.python.org/3.6/library/unittest.html>

Pytest

- Eleganter als unittest, keine Klassen nötig
- Greift in Kompilierung ein um assert zu modifizieren, gibt sinnvolle Fehlermeldungen:

```
assert result == expected
```

- <https://docs.pytest.org/en/latest/>

Gliederung

- Syntax
- Testing
- Bibliotheken

Geschwindigkeits Profiling

- Hier mit cProfile (Alternative profile langsamer)
- Funktionsgenau
- tottime = ohne Unterfunktionen
- cumtime = mit Unterfunktionen
- <https://docs.python.org/3.6/library/profile.html>

Speicherverbrauch Profiling

- Hier mit `memory_profiler`
- Braucht Dekorator an zu testenden Methoden
- Zeilengenau
- Aber: Garbagecollection begrenzt Auflösung
- <https://docs.pytest.org/en/latest/>

Argumente parsen mit docopt

- <http://docopt.org>
- Auch als Java (usw.) Port verfügbar!
- Dokumentationsstring der gleichzeitig CLI Parsing beschreibt

```
args = docopt("""
```

Usage:

```
    coha_converter.py [options] <files>...
```

Options:

```
        --lemma Output lemmata""")
```

```
in_files = args["<files>"]
```

```
lemma = args["--lemma"] #True wenn Flag gesetzt, sonst False
```

Path

- Objektorientierte API für Dateioperationen

```
from pathlib import Path  
current = Path('.').  
test = current / 'testdemo' / 'tested.py'  
test.stem == 'tested'  
test.read_text()
```

- <https://docs.python.org/3.6/library/pathlib.html>

Glob

- Dateien mit Wildcards suchen wie in (ba)sh

```
from pathlib import Path  
  
>>> pyfiles = Path(".").glob("*py")  
  
>>> " ".join([str(x) for x in pyfiles])  
'comprehensiondemo.py memorydemo.py functionaldemo.py  
decorator.py speedtest.py'
```

- Früher per `glob.glob` (Ergebnis: Strings), jetzt mit `Path.glob` (Ergebnis: Path Objekte)

- Rekursion mit `**` (erforderte Flag bei `glob.glob`)

```
pyfiles = Path(".").glob("**/*py")
```

Bonus: Concurrency Quellen von Tobias

- Vorträge von David Beazley:

<https://www.youtube.com/watch?v=MCs5OvhV9S4>

<https://www.youtube.com/watch?v=E-1Y4kSsAFc&t=2849s>

<https://www.youtube.com/watch?v=xOyJiN3yGfU>

- Vortrag von Raymond Hettinger:

<https://www.youtube.com/watch?v=9zinZmE3Ogk>

Bonus: Implementierungen & Co.

- Cpython: default
- Jython: 2.7 auf JVM
- PyPy: 2.7/3.5, reines Python mit JIT
- Cython: Python zu C Kompiler für 2.7/3.3+, kann eingebettetes C (z.B. in SciPy genutzt)

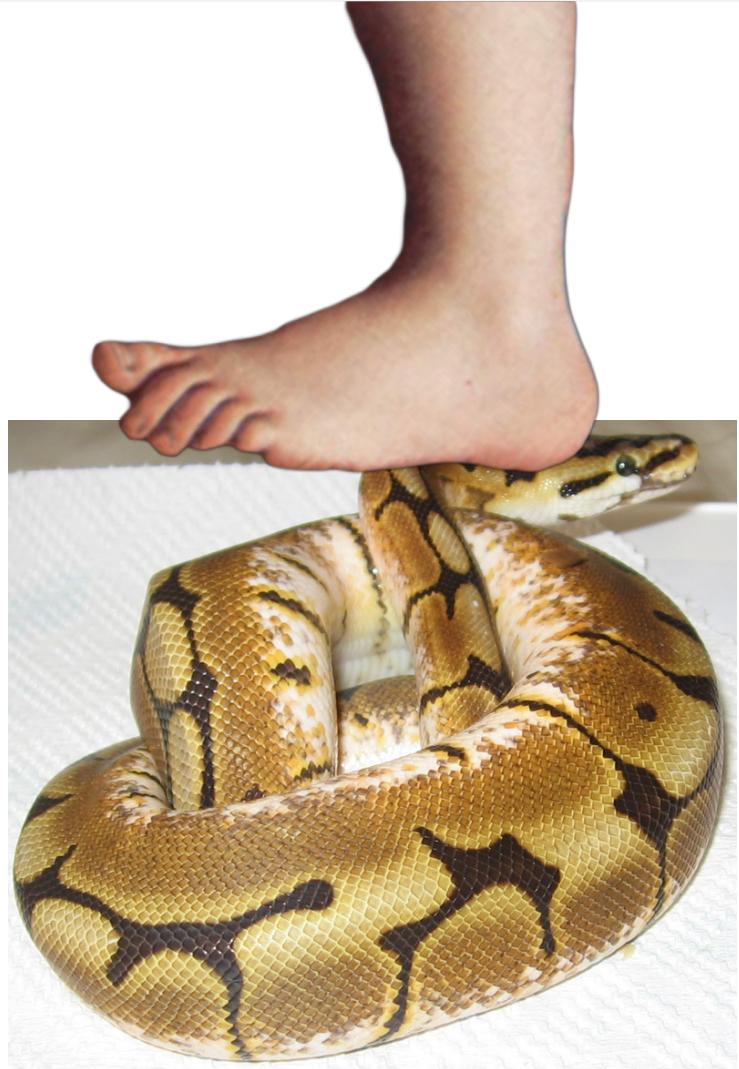
Bonus: Nicht erwähnt

- Metaklassen: Dynamische Erzeugung von Klassen,
Einsatz meist unnötig <https://realpython.com/python-metaclasses/>
- Loggen: vermutlich relevanter ;)
<https://docs.python.org/3.6/library/logging.html>
- * und ** in Funktionsaufrufen (manchmal sinnvoll?)
<http://false.ekta.is/2013/03/esoteric-python-of-the-day-and/>
- Neue String Formattierungs API (wirkt sehr nützlich!)
<https://www.python.org/dev/peps/pep-0498/>

Das Wichtigste zum Schluss

By the way, the language is named after the BBC show “Monty Python’s Flying Circus” and has nothing to do with reptiles.

<https://docs.python.org/3/tutorial/appetite.html>



By WingedWolfPsion - Own work, CC BY-SA 3.0,
<https://commons.wikimedia.org/w/index.php?curid=4166830>