

Einführungskurs Python

Tom Hanika

24. April 2018



Programmieren mit Python

Ziele des Abschnitts

Ziele

- ▶ This space intentionally left blank.

Ziele des Abschnitts

Ziele

- ▶ This space intentionally left blank.

Es gibt viele Implementierungen

Ziele des Abschnitts

Ziele

- ▶ This space intentionally left blank.

Es gibt viele Implementierungen

- ▶ CPython
- ▶ PyPy
- ▶ Jython

Ziele des Abschnitts

Ziele

- ▶ This space intentionally left blank.

Es gibt viele Implementierungen

- ▶ CPython
- ▶ PyPy
- ▶ Jython

Es gibt viele IDEs!

- ▶ Emacs mit `e1py`
- ▶ Sicher kann man Python auch in Eclipse programmieren
- ▶ Notebook-Programming:
 - ▶ IPython (früher)
 - ▶ Jupyter (heute)
 - ▶ Emacs org-babel

There is ONE way to do it!? (Best Practise)

Es gibt für Python einen Style guide:

<https://www.python.org/dev/peps/pep-0008/>

Beispiel (Indentation)

„ Use 4 spaces per indentation level.

Continuation lines should align wrapped elements either vertically using Python's implicit line joining inside parentheses, brackets and braces, or using a hanging indent [7]. When using a hanging indent the following should be considered; there should be no arguments on the first line and further indentation should be used to clearly distinguish itself as a continuation line.“

The Zen of Python, by Tim Peters I

```
1 >>> import this
```

- ▶ Beautiful is better than ugly.
- ▶ Explicit is better than implicit.
- ▶ Simple is better than complex.
- ▶ Complex is better than complicated.
- ▶ Flat is better than nested.
- ▶ Sparse is better than dense.
- ▶ Readability counts.
- ▶ Special cases aren't special enough to break the rules.
- ▶ Although practicality beats purity.
- ▶ Errors should never pass silently.
- ▶ Unless explicitly silenced.
- ▶ In the face of ambiguity, refuse the temptation to guess.

The Zen of Python, by Tim Peters II

- ▶ There should be one— and preferably only one —obvious way to do it.
- ▶ Although that way may not be obvious at first unless you're Dutch.
- ▶ Now is better than never.
- ▶ Although never is often better than **right** now.
- ▶ If the implementation is hard to explain, it's a bad idea.
- ▶ If the implementation is easy to explain, it may be a good idea.
- ▶ Namespaces are one honking great idea – let's do more of those!

WTF: Wer ist Tim Peters?

Langzeit Python Entwickler und „Erfinder“ von Timsort.

Programmieren mit Python

Funktionen und Objekte

Klassen

- ▶ Konstruktoren
- ▶ Klassen-/Instanzvariablen

Klassen

- ▶ Konstruktoren
- ▶ Klassen-/Instanzvariablen

```
1 class Spam():  
2     eggs = []  
3     def __init__(self, bacon):  
4         self.bacon = bacon
```

Klassen

- ▶ Konstruktoren
- ▶ Klassen-/Instanzvariablen

```
1 class Spam():  
2     eggs = []  
3     def __init__(self, bacon):  
4         self.bacon = bacon
```

Python 2:

```
1 class Spam(object):  
2     eggs = []  
3     def __init__(self, bacon):  
4         self.bacon = bacon
```



Klassen

- ▶ Konstruktoren
- ▶ Klassen-/Instanzvariablen

Klassen

- ▶ Konstruktoren
- ▶ Klassen-/Instanzvariablen

```
1 class Spam():  
2     eggs = []  
3     def __init__(self, bacon):  
4         self.bacon = bacon
```

Klassen

- ▶ Konstruktoren
- ▶ Klassen-/Instanzvariablen

```
1 class Spam():
2     eggs = []
3     def __init__(self, bacon):
4         self.bacon = bacon
```

```
1 >>> x = Spam(23)
2 >>> y = Spam(42)
3 >>> x.eggs, x.bacon
4 ([], 23)
5 >>> x.eggs.append(42)
6 >>> x.bacon += 1
7 >>> y.eggs, x.bacon, y.bacon
8 ([42], 24, 42)
```


Klassenvariablen

- ▶ Konstruktoren
- ▶ Klassen-/Instanzvariablen

Klassenvariablen

- ▶ Konstruktoren
- ▶ Klassen-/Instanzvariablen

```
1 class Spam():  
2     eggs = 99  
3     def __init__(self, bacon):  
4         self.bacon = bacon
```

Klassenvariablen

- ▶ Konstruktoren
- ▶ Klassen-/Instanzvariablen

```
1 class Spam():
2     eggs = 99
3     def __init__(self, bacon):
4         self.bacon = bacon
```

```
1 >>> x = Spam(23)
2 >>> y = Spam(42)
3 >>> (x.eggs,y.eggs)
4 (99,99)
5 >>> Spam.eggs=98
6 >>> (x.eggs,y.eggs)
7 (98,98)
```

Komische Klassenvariablen

- ▶ Konstruktoren
- ▶ Klassen-/Instanzvariablen

Komische Klassenvariablen

- ▶ Konstruktoren
- ▶ Klassen-/Instanzvariablen

```
1 class Spam():  
2     eggs = 99  
3     def __init__(self, bacon):  
4         self.bacon = bacon
```

Komische Klassenvariablen

- ▶ Konstruktoren
- ▶ Klassen-/Instanzvariablen

```
1 class Spam():  
2     eggs = 99  
3     def __init__(self, bacon):  
4         self.bacon = bacon
```

```
1 >>> x = Spam(23)  
2 >>> y = Spam(42)  
3 >>> x.eggs=123  
4 >>> (x.eggs,y.eggs)  
5 (123,99)  
6 >>> Spam.eggs=98  
7 >>> (x.eggs,y.eggs)  
8 (123,98)
```

Methoden

- ▶ Funktionen in Klassen
- ▶ Klasseninstanz als (explizites) erstes Argument

Methoden

- ▶ Funktionen in Klassen
- ▶ Klasseninstanz als (explizites) erstes Argument

```
1 class Spam():  
2     def __init__(self, bacon):  
3         self.bacon = bacon  
4     def spam(self):  
5         print(self.bacon)
```


Methoden

- ▶ Funktionen in Klassen
- ▶ Klasseninstanz als (explizites) erstes Argument

```
1 class Spam():
2     def __init__(self, bacon):
3         self.bacon = bacon
4     def spam(self):
5         print(self.bacon)
```

```
1 >>> x = Spam('eggs')
2 >>> y = Spam('bacon')
3 >>> x.spam()
4 eggs
5 >>> x.bacon
6 eggs
7 >>> y.spam()
8 bacon
```

Klassen Methoden

- ▶ Funktionen in Klassen
- ▶ Belfist nur ein Name

Klassen Methoden

- ▶ Funktionen in Klassen
- ▶ Belfist nur ein Name

```
1 class Spam():
2     eggs = 99
3     def __init__(self, bacon):
4         self.bacon = bacon
5     @classmethod
6     def spam(self):
7         print(self.eggs)
```

Klassen Methoden

- ▶ Funktionen in Klassen
- ▶ Belfist nur ein Name

```
1 class Spam():
2     eggs = 99
3     def __init__(self, bacon):
4         self.bacon = bacon
5     @classmethod
6     def spam(self):
7         print(self.eggs)
```

```
1 >>> x = Spam('Suppe')
2 >>> x.spam()
3 99
```

Einschub: Many faces of `_`

- ▶ Im Interpreter enthält `_` den letzten Ausdruck

```
1 >>> [1,2,3]
2 [1,2,3]
3 >>> sum(_)
4 6
```

- ▶ Protected Variablen (in Klassen) `_` variable. Bsp: class `_Bingo()`:

```
1 _versteckte_loesung = 42
2 def __init__(self,value):
3     self._value = value
```

- ▶ Trennen von Ziffern in Literalkonstanten

```
1 >>> print(123_456_789)
2 123456789
```

- ▶ Und noch viel mehr...

Eigenschaften

- ▶ Nicht-Funktionen statt Funktionen

Eigenschaften

- ▶ Nicht-Funktionen statt Funktionen

```
1 class Spam():
2     def __init__(self):
3         self._eggs = 0
4     def eggsInc(self):
5         self._eggs += 1
6         print('{} _eggs'.format(self._eggs))
7     eggs = property(eggsInc)
```

Eigenschaften

- ▶ Nicht-Funktionen statt Funktionen

```
1 class Spam():
2     def __init__(self):
3         self._eggs = 0
4     def eggsInc(self):
5         self._eggs += 1
6         print('{} eggs'.format(self._eggs))
7     eggs = property(eggsInc)
```

```
1 >>> x = Spam()
2 >>> x.eggs
3 1 eggs
4 >>> x.eggs
5 2 eggs
```


Gibt's das auch mit Zucker?

- ▶ Dekoratoren als Kurzschreibweise

Gibt's das auch mit Zucker?

- Dekoratoren als Kurzschreibweise

```
1 class Spam():
2     def __init__(self):
3         self._eggs = 0
4     @property
5     def eggs(self):
6         self._eggs += 1
7         print('{} _eggs'.format(self._eggs))
8     @eggs.setter
9     def eggs(self, n):
10        self._eggs = n - 1
```

Gib't's das auch mit Zucker?

- Dekoratoren als Kurzschreibweise

```
1 class Spam():
2     def __init__(self):
3         self._eggs = 0
4     @property
5     def eggs(self):
6         self._eggs += 1
7         print('{} eggs'.format(self._eggs))
8     @eggs.setter
9     def eggs(self, n):
10        self._eggs = n - 1
```

```
1 >>> x = Spam(); x.eggs = 42
2 >>> x.eggs
3 42 eggs
```

Programmieren mit Python

Comprehensions

Listen/Mengen

$$A = \{x + 1 \mid x \in [1, 10)\}$$

$$B = \{x + 1 \mid x \in A, x = 2 \pmod{3}\}$$

Listen/Mengen

$$A = \{x + 1 \mid x \in [1, 10)\}$$

$$B = \{x + 1 \mid x \in A, x = 2 \pmod{3}\}$$

```

1 >>> a = {x + 1 for x in range(1, 10)}
2 >>> b = {x + 1 for x in a if x % 3 == 2}
3 >>> a
4 {2, 3, 4, 5, 6, 7, 8, 9, 10}
5 >>> b
6 {3, 6, 9}

```

Verschachtelung ist möglich:

```

1 >>> [{x for x in range(1,y+1) if x <= y} for y in range(1,6)]
2 [{1}, {1, 2}, {1, 2, 3}, {1, 2, 3, 4}, {1, 2, 3, 4, 5}]

```

Sequenzen

- ▶ Auch für Sets und Dictionaries
- ▶ Faulheit: Generator-Ausdrücke

Sequenzen

- ▶ Auch für Sets und Dictionaries
- ▶ Faulheit: Generator-Ausdrücke

```
1 >>> {x % 3 for x in range(1, 10)}
2 {0, 1, 2}
3 >>> {w: len(w) for w in ['eggs', 'bacon', 'spam']}
4 {'spam': 4, 'eggs': 4, 'bacon': 5}
5 >>> (x % 3 for x in range(1, 1000))
6 <generator object <genexpr> at 0x7fe12484be10>
7 >>> len(list(_))
8 999
```


Programmieren mit Python

Module

Import

```
1 >>> import math
2 >>> math.sin(0)
3 0.0
4 >>> import math as m
5 >>> m.sin(0)
6 0.0
7 >>> from math import sin, cos
8 >>> sin(0)
9 0.0
10 >>> from math import sin as s
11 >>> s(0)
12 0.0
13 >>> from math import *
14 >>> tan(0)
15 0.0
```

Batteries included

Nützliche Module in der Standardbibliothek

- ▶ Textverarbeitung: `re`, `string`, `textwrap`
- ▶ Mathematik: `decimal`, `fractions`, `math`, `random`, `statistics`
- ▶ Funktionen: `functools`, `itertools`, `operator`
- ▶ Sonstiges: `argparse`, `logging`, `multiprocessing`, `os`
- ▶ Datascience: `NumPy`, `SciPy`, `Pandas`, `Matplotlib`, `Seaborn`, `SciKit-Learn`,...
- ▶ Ostereier: `antigravity`, `this`
- ▶ ...

Programmieren mit Python

Exceptions

Fehlerbehandlung

```
1 try:
2     print(1 // int(input('> i = ')))
3 except ZeroDivisionError:
4     print('Division by zero')
5 except Exception as e:
6     print(repr(e))
7 else:
8     print('no error')
```

Fehlerbehandlung

```
1 try:
2     print(1 // int(input('> i = ')))
3 except ZeroDivisionError:
4     print('Division by zero')
5 except Exception as e:
6     print(repr(e))
7 else:
8     print('no error')
```

```
1 > i = 0
2 Division by zero
3 > i = a
4 ValueError("invalid literal for int() with base 10: 'a',")
5 > i = 1
6 1
7 no error
```

Programmieren mit Python

10

Lesen

- ▶ `open` liefert `File`-Instanz
- ▶ Automatisches Aufräumen

Lesen

- ▶ open liefert File-Instanz
- ▶ Automatisches Aufräumen

```
1 >>> f = open('/etc/issue.net', 'r')
2 ...     print(f.read())
3 ...
4 Debian GNU/Linux buster/sid
```

Besser aber mit with:

```
1 >>> with open('/etc/issue.net', 'r') as issue:
2 ...     print(issue.read())
3 ...
4 Debian GNU/Linux buster/sid
```

Schreiben

- ▶ Ausgabe analog

Schreiben

► Ausgabe analog

```
1 >>> with open('spam.txt', mode='w') as spam:
2     ...     for i in range(1, 5):
3     ...         spam.write('{}\n'.format(i))
4     ...
5     2
6     2
7     2
8     2
```

Programmieren mit Python

Operatorüberladung

Am Beispiel I

```
1 class Point:
2     def __init__(self, x = 0, y = 0):
3         self.x = x
4         self.y = y
5
6     def __str__(self):
7         return "{0},{1}".format(self.x,self.y)
8
9     def __add__(self,other):
10        x = self.x + other.x
11        y = self.y + other.y
12        return Point(x,y)
```

```
1 >>> p1 = Point(3,4); p2 = Point(-2,3)
2 >>> print(p1 + p2)
3 (1,7)
```

Am Beispiel II

```
1 class Point:
2     def __init__(self, x = 0, y = 0):
3         self.x = x
4         self.y = y
5
6     def __lt__(self, other):
7         return self.x < other.x
```

```
1 >>> p1 = Point(3,4); p2 = Point(-2,3)
2 >>> print(p1 < p2)
3 (1,7)
```

Programmieren mit Python

Doing things Parallel

How and why?

- ▶ Keine (wirkliche) Thread-Parallelität wegen GIL (Global Interpreter Lock)

How and why?

- ▶ Keine (wirkliche) Thread-Parallelität wegen GIL (Global Interpreter Lock)
- ▶ \Rightarrow Kein Multithreading!

How and why?

- ▶ Keine (wirkliche) Thread-Parallelität wegen GIL (Global Interpreter Lock)
- ▶ \Rightarrow Kein Multithreading!
- ▶ ABER: Außerhalb von GIL kann „jeder machen was gewollt“

How and why?

- ▶ Keine (wirkliche) Thread-Parallelität wegen GIL (Global Interpreter Lock)
- ▶ ⇒ Kein Multithreading!
- ▶ ABER: Außerhalb von GIL kann „jeder machen was gewollt“
- ▶ ⇒ Process-Parallelität

How and why?

- ▶ Keine (wirkliche) Thread-Parallelität wegen GIL (Global Interpreter Lock)
- ▶ ⇒ Kein Multithreading!
- ▶ ABER: Außerhalb von GIL kann „jeder machen was gewollt“
- ▶ ⇒ Process-Parallelität
- ▶ Zum Beispiel NumPy tut das!

How and why?

- ▶ Keine (wirkliche) Thread-Parallelität wegen GIL (Global Interpreter Lock)
- ▶ ⇒ Kein Multithreading!
- ▶ ABER: Außerhalb von GIL kann „jeder machen was gewollt“
- ▶ ⇒ Process-Parallelität
- ▶ Zum Beispiel NumPy tut das!
- ▶ ABER: Nur Sinnvoll ab Mindestgröße

How and why?

- ▶ Keine (wirkliche) Thread-Parallelität wegen GIL (Global Interpreter Lock)
- ▶ ⇒ Kein Multithreading!
- ▶ ABER: Außerhalb von GIL kann „jeder machen was gewollt“
- ▶ ⇒ Process-Parallelität
- ▶ Zum Beispiel NumPy tut das!
- ▶ ABER: Nur Sinnvoll ab Mindestgröße

How and why?

- ▶ Keine (wirkliche) Thread-Parallelität wegen GIL (Global Interpreter Lock)
- ▶ ⇒ Kein Multithreading!
- ▶ ABER: Außerhalb von GIL kann „jeder machen was gewollt“
- ▶ ⇒ Process-Parallelität
- ▶ Zum Beispiel NumPy tut das!
- ▶ ABER: Nur Sinnvoll ab Mindestgröße

Process

no shared memory
spawning expensive
memory sync not needed

Threads

shared memory
spawning less expensive
memory sync needed

einfaches Beispiel (aus Python-doc)

```
1 from multiprocessing import Pool, TimeoutError
2 import time
3 import os
4
5 def f(x):
6     return x*x
7 with Pool(processes=4) as pool:
8     # print "[0, 1, 4, ..., 81]"
9     print(pool.map(f, range(10)))
```