# Einführungskurs Python

Tom Hanika

19. November 2017

## Organisation und eine kurze Einführung in Python

# Organisation und eine kurze Einführung in Python

## Wer bin ich ?



## Tom Hanika

- Wissenschaftlicher Mitarbeiter am FG Wissensverarbeitung
- Diplom Mathematik
- passionierter Programmierer (C/C++/Python/Haskell/Clojure ...manchmal auch Java ;) )
- hat schon mal bei jemandem zugeschaut der Python programmiert
- darf nicht laut sagen, dass er Fortran 2003 ganz ok findet

# Organisation und eine kurze Einführung in Python

## Warum ist Tom hier?



### Tom ...

- denkt, dieser Kurs hat euch gerade noch gefehlt!
- meint, Python ist eine gute Ergänzung zum Studium in Kassel
- hofft, Studenten im KDD-Praktikum eine Hilfe zu geben
- weiss, dass Python in Wissenschaft wie Industrie sehr beliebt ist

Und woher kommt dieser Kurs?

- Fragment aus LLP-Kurs (**L**inux**L**aTeX**P**ython) 2014
- ...gehalten an der TU Dresden
- ...von Dr. Daniel Borchmann, Maximilian Marx und Tom

# Organisation und eine kurze Einführung in Python

## Warum seid Ihr hier? (Ab jetzt wird es interessant.)



### Ihr ...

- seid nicht hier, weil Ihr Creditpoints erwartet.
- seid hier, um mehr, oder überhaupt etwas, über Python zu erfahren.
- habt schon mal von Python gehört? (Jetzt nicken!)

# Organisation und eine kurze Einführung in Python

Inhalt, Ablauf, Termine



## Inhalt, Ablauf, Termine

Termine:
16.11 (Raum 0446) / 20.11 (Raum 0446) / 21.11 (Raum -1418)

Inhalt:

- Python-Grundlagen
- Programmieren mit Python
- Wissenschaftliches Arbeiten mit Python (aka, tolle Libs)

Für Fragen, Kursmaterial und Anregungen:

- `https://www.kde.cs.uni-kassel.de/lehre/ws2017-18/pythonkurs`
- `hanika@cs.uni-kassel.de`

## Organisation und eine kurze Einführung in Python

Fragen bis hierher?

## Organisation und eine kurze Einführung in Python

Eine (sehr) kurze Geschichte von Python

# Short history of Python (*...as in Monty Python!!*)

- Februar 1991: Guido van Rossum veröffentlicht Python 0.9 auf alt.sources. (Zusammengebastelt aus *ABC* und *Modula-3*)
- Januar 1994: Python 1.0 erscheint. Features: *lambda, map, filter and reduce*
- Oktober 2000: Python 2.0 erscheint. DAS FEATURE: *List Comprehensions*
- Dezember 2008: Python 3.0 erscheint, ein **redesign**⇒ Nicht Abwärtskompatibel!
- Dezember 2016: Python 3.6

⇒ Python 2 Support bis 2020

⇒ Wir lernen Python 3

http://www.flickr.com/people/52614599@N00

## Organisation und eine kurze Einführung in Python

Python in Buzzwords

# Bingo anyone?

- Multiparadigmensprache?!:
  - Imperativ (Anweisung nach Anweisung)
  - Strukturiert (Subroutine/Blöcke/etc)
  - Objekt-orientiert (wenn man will, sehr gut unterstützt)
  - Funktional („OK“: Kein automatisches Currying, tail recursion opt. …)
- Typsystem:
  - stark (Es gibt Typen für Integer, Reals, Strings, usw.)
  - dynamisch (Laufzeit entscheidet ob Typ passt)
  - Duck-Typing (Wenn ich es addieren kann, ist es sowas wie ne Zahl)
- More buzzwords:
  - Python wird *compiliert* zu Python-Bytecode
  - Python ist (manchmal) langsam!?

**Python-Grundlagen**

# Ziele des Abschnitts

## Ziele

- „Python als Taschenrechner“
- Grundlagen: Syntax, Datentypen

## Interaktiv!

Python installieren:

- `https://www.python.org/downloads/`
- hier: Python 3.6+

Hinweis für KDD-„Praktikanten“:

- Wir empfehlen für das Praktikum die Anaconda Distribution
- `https://www.anaconda.com/download/`
- Warum? Gutes Package-Management, viele relevante Pakete vorinstalliert, konzipiert für Data-Science.
- Allerdings: Nicht da hängen bleiben in der Zukunft ;)

# Hallo.

```
1 print("Hello, world!")
```

## Python 2:

```
1 from __future__ import print_function
```

# Rechnen mit Python

- Arithmetik mit +, -, *, /, **, %
- Ganzzahlige Division mit //

```
>>> 23 + 42
65
>>> 2.3 * 5 - 9 / 6 ** -3
-1932.5
>>> 3 / 2
1.5
>>> 3 // 2
1
>>> 42**2**2**2
937537496836987504768450 56
```

Python 2:

```
from __future__ import division
```



# Zahlentypen

- Ganzzahlen, beliebig genau
- Binäre Fließkommazahlen
- Komplexe Zahlen
- Rationale Zahlen (Modul `fractions`)
- Dezimale Fließkommazahlen (Modul `decimal`)

```
>>> complex(0, 1)
1j
>>> 3 - 1j
(3-1j)
>>> (3-1j).conjugate()
(3+1j)
>>> (3 - 1j).real
3.0
>>> (3 - 1j).imag
-1.0
```

# Zeichenketten

- In Python 3 alles Unicode! :)
- unveränderbar, jede Änderung erzeugt eine neue Zeichenkette

```
>>> "spam" 'eggs'
'spameggs'
>>> s = 'spam'
>>> s * 3 + 'eggs'
'spamspamspameggs'
>>> print('spam\neggs\nbacon')
spam
eggs
bacon
>>> print(r'spam\neggs\nbacon')
spam\neggs\nbacon
```



# Listen

- veränderbare Folge von Elementen

```
>>> l = list(range(1, 10))
>>> l
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> l[2]
3
>>> l[2:]
[3, 4, 5, 6, 7, 8, 9]
>>> l[:2]
[1, 2]
>>> l[::2]
[1, 3, 5, 7, 9]
>>> l[2:7:3]
[3, 6]
```

# Mehr Listen

```
>>> l = list(range(1, 10))
>>> l
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> l[2] = 23
>>> l
[1, 2, 23, 4, 5, 6, 7, 8, 9]
>>> list(reversed([1, 3, 2]))
[2, 3, 1]
>>> sorted([2, 7, 1, 8, 2, 8])
[1, 2, 2, 7, 8, 8]
>>> ", ".join(['eggs', 'bacon', 'spam'])
'eggs, bacon, spam'
```



# Tupel

- unveränderbare Folge von Elementen

```
>>> a, b, c = (1, 2, 3)
>>> a
1
>>> (b, c)
(2, 3)
>>> c, a, b = a, b, c
>>> (a, b, c)
(2, 3, 1)
>>> (1, 2, 3)[2]
2
```

# Sets

- veränderbare Menge von unveränderbaren Elementen⇒?

```
>>> c = {'Mathe',5,True,2.812}
>>> c
{'Mathe', True, 2.812, 5}
>>> a = set(range(1, 10))
>>> b = set(range(3, 9))
>>> a
{1, 2, 3, 4, 5, 6, 7, 8, 9}
>>> a.union(b)
{1, 2, 3, 4, 5, 6, 7, 8, 9}
>>> a.issuperset(b)
True
>>> a.difference(b)
{1, 2, 9}
```



# Dictionaries

- veränderbare Zuordnung von unveränderbaren Schlüsseln zu Werten

```
>>> d = {0: 'eggs', 1: 'bacon', 'spam': 'spam'}
>>> d
{0: 'eggs', 1: 'bacon', 'spam': 'spam'}
>>> d[1]
'bacon'
>>> d['spam']
'spam'
>>> d['eggs'] = 'spam'
>>> d
{0: 'eggs', 1: 'bacon', 'eggs': 'spam', 'spam': 'spam'}
```

# Python-Grundlagen

## Funktionen



## Funktionen

- Einrückung als Syntaxelement
- Funktionen sind *first-class*

```
>>> def f():
...     "prints␣'hello,␣world!'"
...     print('hello,␣world!')
...
>>> f()
'hello,␣world!'
>>> help(f)
Help on function f in module __main__:

f()
    prints 'hello,␣world'
```

# Mehr Funktionen

- Funktionen mit Argumenten
- Lambdas als anonyme Funktionen

```
>>> def g(x, y):
...     return complex(y, x).conjugate()
...
>>> g(23, 42)
(42-23j)
>>> g(y=23, x=42)
(23-42j)
>>> h = lambda x, y: complex(y, x).conjugate()
>>> h(23, 42)
(42-23j)
```



# Noch mal ausführlich: Lambda / filter / reduce / map

```
>>> tolle_liste = [2, 24, 9, 23, 19, 18, 8, 6, 27]
>>> print(list(filter(lambda x: x % 3 == 0, tolle_liste)))
[24, 9, 18, 6, 27]
>>> print(list(map(lambda x: x**2, tolle_liste)))
[4, 576, 81, 529, 361, 324, 64, 36, 729]
>>> from functools import *
>>> print(reduce(lambda x,y: x+y,tolle_liste))
136
```

# Python-Grundlagen

## Kontrollfluss



## Vergleiche

- Vergleiche mit <, <=, ==, !=, >=, >
- == vergleich Wert, nicht Referenz
- `is` vergleicht Referenz

```
>>> 1 == 1
True
>>> 1 < 2 <= 3 != 4 >= 3 > 2
True
>>> 1 < 2 <= 3 == 4 >= 3 > 2
False
>>> [1, 2, 3] == [1, 2, 3]
True
>>> [1, 2, 3] is [1, 2, 3]
False
```

# Logische Ausdrücke

- `and`, `or`
- Kurzschlussauswertung

```
>>> False and True
False
>>> False and 0 / 0 < 1
False
>>> 1 < 2 or False or 3 < 4
True
```



# Bedingungen

```
def f(x):
    if -3 <= x <= 3:
        print(r'x␣\in␣(-3,␣3)')
    elif x < 0:
        print('x␣<␣-3')
    else:
        print('x␣>␣3)
```

```
>>> f(0)
x \in (-3, 3)
>>> f(4)
x < -3
>>> f(-4)
x > 3
```

# Schleifen

- for und while

```
>>> for i in range(1, 4):
...     print(i)
...
1
2
3
>>> i = 1
>>> while i < 4:
...     print(i)
...     i += 1
...
1
2
3
```



# Mehr Schleifen

```
ne_liste = ["Haus", 42, True]
for i in ne_liste:
  print(i)
  if i == "Haus":
    print("Yay")
  else:
    print('Kein Haus')
```

```
while ne_liste !=[] :
  print(ne_liste.pop())
```

```
noch_ne_liste = []
while len(noch_ne_liste) < 3 :
  noch_ne_liste.append("ding")

print(noch_ne_liste)
```

# Noch mehr Schleifen

- Iteration durch beliebige Sequenzen

```
>>> for c in 'spam':
...     print(c)
...
s
p
a
m
>>> for x in {'eggs', 23, 'bacon', 'spam'}:
...     print(x)
...
eggs
spam
bacon
23
```



Fragen soweit?

# Programmieren mit Python

## Ziele des Abschnitts

### Ziele

- This space intentionally left blank.

### Es gibt viele IDEs!

- Emacs mit `elpy`
- 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.

# 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
```

Funktionen und Objekte

# 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)
```

# Methoden

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

```
class Spam():
    def __init__(self, bacon):
        self.bacon = bacon
    def spam(self):
        print(self.bacon)
```

```
>>> x = Spam('eggs')
>>> y = Spam('bacon')
>>> x.spam()
eggs
>>> x.bacon
eggs
>>> y.spam()
bacon
```



# Einschub: Many faces of _

- Im Interpreter enthält _ den letzten Ausdruck

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

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

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

- Trennen von Ziffern in Literalkonstanten

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

- Und noch viel mehr...

# Eigenschaften

- Nicht-Funktionen statt Funktionen

```
class Spam():
    def __init__(self):
        self._eggs = 0
    def eggs(self):
        self._eggs += 1
        print('{} eggs'.format(self._eggs))
    eggs = property(eggs)
```

```
>>> x = Spam()
>>> x.eggs
1 eggs
>>> x.eggs
2 eggs
```



# Gibt's das auch mit Zucker?

- Dekoratoren als Kurzschreibweise

```
class Spam():
    def __init__(self):
        self._eggs = 0
    @property
    def eggs(self):
        self._eggs += 1
        print('{} eggs'.format(self._eggs))
    @eggs.setter
    def eggs(self, n):
        self._eggs = n
```

```
>>> x = Spam(); x.eggs = 41
>>> x.eggs
42 eggs
```