HowTo:Python

Paua-App // Louis Kobras

5. Mai 2016

Syntax

- “runnable pseudo code”
- Wie Java, ohne Klammern
- Indentation-empfindlich
- ASCII-Encoding
- Kommentar: #

Syntax

- “runnable pseudo code”
- Wie Java, ohne Klammern
- Indentation-empfindlich
- ASCII-Encoding
- Kommentar: #

```
# This is a comment
print "Hello World."
name = raw_input("What is your name?")
print "Hello, %s" % name
```

Kontrollstrukturen

- if-(elif)*-(else)?[1]
- while
- foreach

[1] Man beachte die RegEx-Syntax

Kontrollstrukturen

- if-(elif)*-(else)?[1]
- while
- foreach

```
if condition:
    doStuff()
elif foo:
    bar()
elif spam:
    eggs()
else:
    # todo
```

[1] Man beachte die RegEx-Syntax

Daten

- Everything is data (even functions)
- dynamisch getypt
- Funktionen können Tupel zurückgeben
- Default-Parameter möglich

Daten

- Everything is data (even functions)
- dynamisch getypt
- Funktionen können Tupel zurückgeben
- Default-Parameter möglich

```
# alles gueltige Deklarationen
a = "String"
b = 0
c = 0.7
d = a
a += 1
```

#Demo

OOP 1

- Klassen als Schablonen
- Magic Methods für Objektverhalten
- Private Methoden und Felder möglich, jedoch ungenutzt

OOP 1

- Klassen als Schablonen
- Magic Methods für Objektverhalten
- Private Methoden und Felder möglich, jedoch ungenutzt

Magic Methods:

- init (Konstruktor)
- add, sub, mul, div . . . (Arithmetik)
- eq, ne, lt, gt, le, ge (Vergleich)
- pos, neg, floor, ceil . . . (Numerik)
- viele, viele mehr (googlen)

OOP 2

```
class foo:
lst = []
    def __init__(self, lst):
        self.lst = lst
    def __add__(self, lst):
        self.lst.append(lst)
        return self.lst
```

Konstruktoraufruf: bar = foo([a,b,c])
Übergibt die Liste [a,b,c] an init

OOP 3 - Vererbung

- Mutterklassen in Klammern
- implizites Override von Methoden und Feldern
- transitiv
- aufrufen von Eigenschaften der Superklasse mit `super().spam`

OOP 3 - Vererbung

- Mutterklassen in Klammern
- implizites Override von Methoden und Feldern
- transitiv
- aufrufen von Eigenschaften der Superklasse mit `super().spam`

```
class animal:
    ...
    def bark():
        print "woof"

class wolf(animal):
    ...


>>> wolf.bark()
"woof"
```

Funktionale Programmierung

- Rekursion und Tail Recursion implementierbar
- Funktionen höherer Ordnung vorhanden und baubar
- `lambda`-Funktion:

```
\pause
>>> def square(lst):
...     o = []
...     for e in lst:
...         a = lambda x: x*x
...         o.append(a(e))
...     return o
>>> square([1,2,3])
[1, 4, 9]
```

$\Rightarrow$ `lambda`-Funktion an den Buchstaben 'a' gebunden

Module

- Importe mit `import »module«` oder `from »module« import »function«`
- Importe können benannt werden (`... as »nick«`)
- Bedingte Importe möglich:

Module

- Importe mit `import »module«` oder `from »module« import »function«`
- Importe können benannt werden (`... as »nick«`)
- Bedingte Importe möglich:

```
from math import factorial as fac
if os.name == "WinDoof":
    import thatOneLib as lib
elif os.name == "Linux":
    import thatOtherLib as lib
else:
    import thatStandartLib as lib
```

Grafik Stuff

- drölf verschiedene Frameworks
- Auswahl je nach gewünschter Funktionalität und Zielplattform(en)

About - Random Facts and Stuff

- PEPs
- Zen of Python (`import this`)
- BDFL