

Skriptsprachen: Python

Funktionen

Jan Krüger, Alexander Sczyrba



Technische Fakultät
Universität Bielefeld

30. September 2016

Python

—

Funktionen

Funktionen

- Warum Funktionen/Methoden

Funktionen

- Warum Funktionen/Methoden
 - keine Code-Duplikation
 - Faktorisierung
 - Strukturierung
 - keine Spaghetti-Code

Funktionen

- Warum Funktionen/Methoden
 - keine Code-Duplikation
 - Faktorisierung
 - Strukturierung
 - keine Spaghetti-Code
- in Python: Funktionen

Funktionen

- Warum Funktionen/Methoden
 - keine Code-Duplikation
 - Faktorisierung
 - Strukturierung
 - keine Spaghetti-Code
- in Python: Funktionen
- **Funktionen sind Objekte !!!**

Schreiben einer Funktion

- Eine Funktion muss
 - einen Namen
 - eine Schnittstelle (mit keinem, einem oder mehreren Parametern)
 - einen Rueckgabewert (Default: None)haben.
- Schluesselwort `def`

```
def Funktionsname (p1, p2, ... pn):  
    Anweisung1  
    Anweisung2  
    ...  
    return Rueckgabewert
```

Aufgabe

Schreibe eine Funktion 'reverse', welche einen String umgekehrt wieder zurueckgibt.

Positional vs. KeyWord Parameter

bisher positional Parameter
Reihenfolge ist wichtig

alternativ Keyword Parameter

- Reihenfolge ist unwichtig
- Parameter durch das Keyword festgelegt

```
def El_Mamun(calif, sons, baggage_camels, costs, additional_costs):  
    return (calif+sons)*baggage_camels*costs+additional_costs
```

```
El_Mamun(1,2,3,4,5)
```

```
El_Mamun(sons=2, calif=1, baggage_camels=3, additional_costs=5, costs=4)
```

Optionale Parameter

- ipython: `?range`

Welche Erkenntnis kannst Du aus der Online Hilfe ziehen ?

Optionale Parameter

- ipython: ?range

Welche Erkenntnis kannst Du aus der Online Hilfe ziehen ?

- ```
def hoch(x,y=2):
 return x ** y
```

# Optionale Parameter

---

- ipython: `?range`  
Welche Erkenntnis kannst Du aus der Online Hilfe ziehen ?
- ```
def hoch(x,y=2):  
    return x ** y
```
- Optionale Parameter muessen immer am Ende der Deklaration stehen, Warum ?

Beliebige Anzahl von Parametern

Python erlaubt es, fuer die beiden Formen der Parameteruebergaben (positional und keyword) eine nicht festgelegte Anzahl von Parameter verarbeiten

- 'positional' Parameter

```
def fkt_1(*param):  
    print param
```

- 'keyword' Parameter

```
def fkt_2 (**param):  
    print param
```

- Teste beide Funktionen in der python Shell, Was faellt Dir auf ?

Dokumentation

- Python bietet eine einfache Möglichkeit selbst geschriebene Funktionen zu dokumentieren (Docstring)
- wird von der Online Hilfe von der [i]python Shell genutzt

```
def test():  
    "Die Funktion 'test' ist ein Beispiel..."
```

```
?test()
```

Mehrere Rueckgabewerte

In vielen Sprachen problematisch, fuer Python kein Problem.

```
def hoch(n):  
    return n*n, n*n*n  
  
2_hoch_2, 2_hoch_3 = hoch(2)  
3_hoch_2_AND_3_hoch_3 = hoch(3)
```

Aufgabe

Schreibe eine Funktion, die eine Sequenz im FASTA Format ausgibt, wobei die Breite der Sequenz in der Ausgabe als Parameter uebergeben werden soll.

- Was ist eine FASTA Sequenz ? Wie ist Sie definiert ?
- 'Positional' oder 'KeyWords' Parameter - Was ist die bessere Wahl ?

Lexical Scoping [1]

- lokale vs. globale Variablen

```
def f(a):  
    return a
```

```
a = 100  
f(111)  
a
```

- lesender Zugriff auf globale Variable

```
def g():  
    print s  
s = 1  
g()
```

Lexical Scoping [2]

- Probiere folgendes aus :

```
def h():  
    s = 2  
    print s  
s = 1  
h()  
s
```

Lexical Scoping [2]

- Probiere folgendes aus :

```
def h():  
    s = 2  
    print s  
s = 1  
h()  
s
```

- schreibender Zugriff auf globale Variable

```
def h():  
    global s  
    s = 2  
    print s
```

Lokale Funktionen

- Funktionen in einer Funktion heissen 'lokale' Funktionen
- Gueltigkeit nur innerhalb des Namensraum einer Funktion

```
def globale_funktion(n):  
    def lokale_funktion(n):  
        return n*n  
  
    return lokale_funktion
```

Anonyme Funktionen

- Schluesselwort `lambda` - λ Funktion
- Vorteile einer 'echten' Funktion aber einfach zu definieren

```
> hoch = lambda x,y:x**y  
> hoch(2,2)  
> hoch(2,3)
```

- keine Kontrollstrukturen
- effizienter als 'echte' Funktionen
- direkt aufrufen :

```
> (lambda x,y:x**y)(2,2)
```

Aufgabe

Schreibe eine λ Funktion die eine Liste von Zahlen als Argument bekommt und die Summe dieser zurueckgibt !

Python

—

Errors und Exceptions

Semantik != Syntax

Selbst wenn Code semantisch korrekt ist, bedeutet das nicht, dass er auch syntaktisch korrekt ist. Umgekehrt gilt dies ebenso.

SyntaxError Semantisch evtl. korrekt, entspricht aber syntaktisch nicht den Python-Vorgaben

```
> while True print 'Hello_world'
File "<ipython-input-5-f4b9dbd125c8>", line 1
    while True print 'Hello_world'
                        ^
```

SyntaxError: invalid syntax

Syntax != Semantik

Exception Syntaktisch korrekt, entspricht aber semantisch keiner Logik bzw. Voraussetzung

```
> 10 * (1/0)
```

```
ZeroDivisionError: integer division or modulo  
by zero
```

```
> 4 + spam*3
```

```
NameError: name 'spam' is not defined
```

```
> '2' + 2
```

```
TypeError: cannot concatenate 'str' and  
'int' objects
```

Fehlerbehandlung

<code>try:</code>	<code>try</code> Unsicherer Block
<code> <ANWEISUNGEN></code>	
<code>except <FEHLER>:</code>	<code>except</code> Fehlerauffangender Teil. <FEHLER>:
<code> <ANWEISUNGEN></code>	<code>IOError as ioe</code> Fehler als Objekt im fehlerbehandelnden Teil verfügbar
<code>[except <FEHLER>:</code>	<code>(RuntimeError, TypeError)</code> behandelt alle Fehler des angegebenen Typs
<code> <ANWEISUNGEN></code>	<code>ARGUMENTLOS</code> behandelt jeden(!) Fehler
<code>.</code>	
<code>.</code>	
<code>.]</code>	
<code>[else:</code>	<code>else</code> Wird ausgeführt, wenn kein Fehler auftrat
<code> <ANWEISUNGEN>]</code>	

Fehlerbehandlung

```
>for arg in sys.argv[1:]:  
> try:  
>     f = open(arg, 'r')  
> except IOError:  
>     print 'cannot open', arg  
> else:  
>     print arg, 'has', len(f.readlines()), 'lines'  
>     f.close()
```

Besser nur die Codeteile in den try-Block schreiben, die wirklich Fehler verursachen können.

Fehler werfen

Mit `raise` können Fehler erzeugt werden.

```
> try:
>   raise NameError('HiThere')
> except NameError:
>   print '\nAn_exception_flew_by!\n'
>   raise
```

```
"An_exception_flew_by!"
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 2, in ?
```

```
NameError: HiThere
```

Erzwungene Ausführung bei jedem Zustand

Der `finally`-Block wird immer ausgeführt. Dieser Block wird auch als *Clean-up action* bezeichnet

```
> def divide(x, y):  
    try:  
        result = x / y  
    except ZeroDivisionError:  
        print "division by zero!"  
    else:  
        print "result is", result  
    finally:  
        print "THX for using me"
```

Übung – Fehlerhierarchie

```
BaseException
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
    +-- StopIteration
    +-- StandardError
        +-- BufferError
        +-- ArithmeticError
            +-- FloatingPointError
            +-- OverflowError
            +-- ZeroDivisionError
        +-- AssertionError
        +-- AttributeError
        +-- EnvironmentError
            +-- IOError
            +-- OSError
                +-- WindowsError (Windows)
                +-- VMSSError (VMS)
        +-- EOFError
        +-- ImportError
        +-- LookupError
            +-- IndexError
            +-- KeyError
        +-- MemoryError
        +-- NameError
            +-- UnboundLocalError
        +-- ReferenceError
        +-- RuntimeError
            +-- NotImplementedError
        +-- SyntaxError
            +-- IndentationError
            +-- TabError
        +-- SystemError
        +-- TypeError
        +-- ValueError
            +-- UnicodeError
                +-- UnicodeDecodeError
                +-- UnicodeEncodeError
                +-- UnicodeTranslateError
    +-- Warning
        +-- DeprecationWarning
        +-- PendingDeprecationWarning
        +-- RuntimeWarning
        +-- SyntaxWarning
        +-- UserWarning
        +-- FutureWarning
        +-- ImportWarning
        +-- UnicodeWarning
        +-- BytesWarning
```

Erstellt eine Funktion, die als Eingabe eine Liste und ein Objekt erwartet und True zurückliefert, wenn sich das Objekt in der Liste befindet und ansonsten einen LookupError mit entsprechender Fehlernachricht (Parameter) erzeugen soll.

Achtet auf korrektes *Exception handling* bei Verwendung der `index()`-Methode!