

ProInformatik 3: Objektorientierte Programmierung Python (Teil 2)



SoSe 2018

Oliver Wiese

Bisher gesehen:

- Syntax und Semantik imperativer Programmiersprachen
 - Variablen, Ausdruck, Zuweisung, Kontrollfluss, Funktionen
 - Call-by-value vs. call-by-reference
- Besonderheiten in Python
 - Datentypen (int, float, String, Listen, Tuple, ...)
 - Operatoren
 - Schlüsselwörter
 - Programm in der Ausführung

Heute

- Wiederholung von Schleifen
- Einsatz von Funktionen
- Testen von Funktionen
- Schlüsselwörter
- Coding style
- Weitere Vorschläge?

Erinnerung: **for**-Schleife

```
for Ausdruck in Sequenz :  
    Anweisungen
```

Bei der **for**-Schleife in Python wird nicht über eine Folge von Zahlen, sondern über Elemente einer Sequenz iteriert.

```
text = input ( "text = " )
```

```
for i in text:  
    print(i)
```

```
text = abc  
a  
b  
c
```

for-Schleifen

```
for x in ['spam', 'bla', 'eggs']:  
    print (x)
```

```
>>>  
spam  
bla  
eggs
```

```
for x in [1, 2, 3, 4]:  
    print (x)
```

```
>>>  
1  
2  
3  
4
```

```
sum=0
```

```
for x in [1, 2, 3, 4, 5]:  
    sum = sum + x
```

```
print( sum)
```

```
>>>  
15
```

```
for i in range(1, 5):  
    sum += i
```

```
print( sum)
```

```
>>>  
25
```

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

```
for (x,y) in Dic:  
    print (x, y)
```

```
>>>  
1 a  
2 b  
3 c
```

Schrittweite von Schleifen:

```
n = 10
```

```
summe = 0
```

```
for i in range (0,n,2):
```

```
    print(i)
```

```
    summe += i
```

```
print('sum: ',summe)
```

Output:

```
#####  
0  
2  
4  
6  
8  
sum: 20  
>>>
```

Listen-Generatoren

Python:

```
[ x*x for x in range (5) ]
```

Eine Liste mit den Quadratzahlen von 0 bis 4 wird generiert.

```
>>> [ x%3 for x in [1, 5, -3, -6, 4, 7, 6, 0] if (x>0) ]
```

```
[1, 2, 1, 1, 0]
```

Verschachtelung

Programm:

```
n = 3
```

```
print('i j')
```

```
for i in range(0,n):
```

```
    for j in range(0,n):
```

```
        print(i, j)
```

```
    print('increase outer loop')
```

Output:

```
i j
0 0
0 1
0 2
increase outer loop
1 0
1 1
1 2
increase outer loop
2 0
2 1
2 2
increase outer loop
>>>
```


Unvollständiger Lösungsvorschlag

```
for guessesTaken in range(3):  
    print('Take a guess.')  
    guess = int(input())  
    if guess < number:  
        print('Your guess is too low.')  
    if guess > number:  
        print('your guess is too high.')  
    if guess == number:  
        break  
if guess == number:  
    print('Good job! You are right')
```

Mit der **break**-Anweisung wird die Ausführung einer Schleife (**for** und **while**) vorzeitig beendet

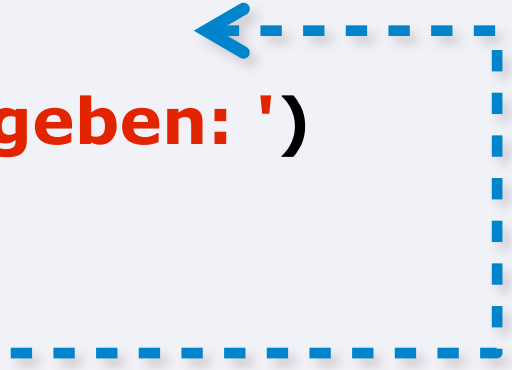
Output

```
Take a guess.  
2  
Your guess is too low.  
Take a guess.  
5  
Good job! You are right  
>>>
```

continue-Anweisung

Die **continue**-Anweisung wird verwendet, um die restlichen Anweisungen der aktuellen Schleife zu überspringen und direkt mit dem nächsten Schleifen-Durchlauf fortzufahren.

```
while True:
    s = input('Text eingeben: ')
    if s == 'kein print':
        continue
    print ('Die Laenge des Texts ist', len(s))
```



Zusammenfassung:

- Unterscheidung zwischen **for** und **while**
- Wichtige Schlüsselbegriffe: **break**, **continue**
- Beachte **range** und Verwendung als Listengenerator
- Rechenzeit intensiv

Funktionen

Funktionen

Funktionen sind das **wichtigste Konzept** in der Welt der höheren Programmiersprachen.



Funktionen sind ein grundlegendes Hilfsmittel, um Probleme in kleinere Teilaufgaben zerlegen zu können.

Sie ermöglichen damit eine **bessere Strukturierung** eines Programms sowie die Wiederverwertbarkeit des Programmcodes.

Gut strukturierte Programme bestehen typischerweise aus **vielen kleinen**, nicht aus wenigen großen Funktionen.

Funktionen in Python

Eine Funktionsdefinition startet mit dem **def**-Schlüsselwort

Funktionsname

Argumente

$$\pi = 4 \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1}$$

```
def pi_leibniz ( k ):
    sum = 0
    for n in range( 0, k ):
        sum = sum + ((-1)**n)/(2.0*n+1)
    return sum*4
```

Das Einrücken entscheidet, was zur Funktion gehört bzw. wann die Funktionsdefinition zu Ende ist.

Die **return**-Anweisung gibt das Ergebnis der Funktion zurück

Rekursive Funktionen

Funktionen können rekursiv definiert werden.

Beispiel:

```
def fact (n) :  
    if n==0:  
        return 1  
    else:  
        return n*fact(n-1)
```

Anwendung:

```
>>> print( fact(5) )  
>>> 120
```

Rekursive Funktionen

```
10! = 3628800
```

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

```
  File "/Volumes/Nidaba HD/gitolite/lehre/SS18/ProInf3/VL3_examples/myFunctions.py",  
line 8, in <module>
```

```
    print('1000! = ',fact(1000))
```

```
  File "/Volumes/Nidaba HD/gitolite/lehre/SS18/ProInf3/VL3_examples/myFunctions.py",  
line 5, in fact
```

```
    return n*fact(n-1)
```

```
  File "/Volumes/Nidaba HD/gitolite/lehre/SS18/ProInf3/VL3_examples/myFunctions.py",  
line 5, in fact
```

```
    return n*fact(n-1)
```

```
  File "/Volumes/Nidaba HD/gitolite/lehre/SS18/ProInf3/VL3_examples/myFunctions.py",  
line 5, in fact
```

```
    return n*fact(n-1)
```

```
[Previous line repeated 989 more times]
```

```
  File "/Volumes/Nidaba HD/gitolite/lehre/SS18/ProInf3/VL3_examples/myFunctions.py",  
line 2, in fact
```

```
    if n == 0:
```

```
RecursionError: maximum recursion depth exceeded in comparison
```

Warum?

```
>>>
```


Verschachtelte Funktionen

Funktionen können innerhalb anderer Funktionen definiert werden.

```
def percent (a, b, c):
    def pc(x): return (x*100.0) / (a+b+c)
    return (pc(a), pc(b), pc(c))

print (percent (2, 4, 4))
print (percent (1, 1, 1))
```

```
>>>
(20.0, 40.0, 40.0)
(33.333333333333, 33.333333333333, 33.333333333333)
>>>
```

Funktionen als Objekte

- In Python sind Funktionen Objekte (**first-class objects**)
- haben ein Datentyp

```
>>> type ( factorial )
```

```
>>> <type 'function'>
```

- ermöglicht Meta-Programmierung

higher-order functions

aus FP:

Eine Funktion wird als **Funktion höherer Ordnung** bezeichnet, wenn **Funktionen als Argumente** verwendet werden oder wenn eine **Funktion als Ergebnis** zurück gegeben wird.

Funktionen als Objekte

```
def myMap(ls, f):  
    for i in range(len(ls)):  
        ls[i] = f(ls[i])
```

```
list1 = [2, 3, 4, 5, 0, 1]  
myMap(list1, factorial)  
print ( list1 )
```

Ausgabe:

```
>>>
```

```
[2, 6, 24, 120, 1, 1]
```

Funktionen höherer Ordnung

```
def myMap (f, xs):  
    result = []  
    for x in xs:  
        result.append(f(x))  
    return result  
  
nums = [2,3,4,5,0,1]  
result_list = myMap (factorial, nums)  
print(nums)  
print(result_list)
```

Ausgabe:

```
>>>  
[2, 3, 4, 5, 0, 1]  
[2, 6, 24, 120, 1, 1]
```

yield-Anweisung

Die **yield**-Anweisung innerhalb einer Funktion **f** verursacht einen Rücksprung in die aufrufende Funktion und der Wert hinter der **yield**-Anweisung wird als Ergebnis zurückgegeben.

Im Unterschied zur **return**-Anweisung werden die aktuelle Position innerhalb der Funktion **f** und ihre lokalen Variablen zwischengespeichert.

Beim nächsten Aufruf der Funktion **f** springt Python hinter dem zuletzt ausgeführten **yield** weiter und kann wieder auf die alten lokalen Variablen von **f** zugreifen.

Wenn das Ende der Funktion **f** erreicht wird, wird diese endgültig beendet.

yield-Anweisung

```
def myRange(n):  
    i = 0  
    while (i < n):  
        yield i  
        i = i + 1  
  
for i in myRange(5):  
    print(i)
```

```
>>>  
0  
1  
2  
3  
4  
>>>
```

```
def genConstants():  
    yield 3  
    yield 5  
    yield 11  
  
def testMyRange():  
    for x in genConstants():  
        print(x)  
  
testMyRange()
```

```
>>>  
3  
5  
11  
>>>
```

yield-Anweisung

```
def fibonacci():  
    a, b = 0, 1  
    while True:  
        yield a  
        a, b = b, a + b  
  
def getFibonacci(n):  
    counter = 0  
    for x in fibonacci():  
        counter += 1  
        if (counter > n):  
            break  
    return x  
  
print(getFibonacci(77))
```

```
>>>  
5527939700884757
```

Funktionsdokumentation

Funktionen können einen Dokumentationstext beinhalten, der als Blockkommentar in der **ersten Zeile der Funktion** geschrieben werden muss.

```
def fact (n) :  
    """ Berechnet die Fakultätsfunktion der Zahl n """  
    if n==0:  
        return 1  
    else:  
        return n*fact(n-1)
```

```
>>> help (fact)  
Help on function fact in module __main__:  
fact(n)  
    Berechnet die Fakultätsfunktion der Zahl n
```


gute imperative Funktionen

- vermeiden Rekursion
- sind dokumentiert
- können andere Funktionen als Argument beinhalten
- Eingabe erfolgt nur mittels Argumenten (kein **input**, keine globale Variable)
- Ausgabe erfolgt nur mittels **return** (kein print)

Testen von Funktionen

```
def teiler (a,b):  
    return a%b == 0
```

```
def testTeilerManuell():
```

```
    print(teiler(5,2) == False)
```

```
    print(teiler(4,2) == True)
```

```
    print(teiler(2,2) == True)
```

```
    print(teiler(25,5) == True)
```

```
    print(teiler(24,5) == False)
```

```
    print(teiler(25,1) == True)
```

```
    print(teiler(1,25) == False)
```

Output:

```
True  
True  
True  
True  
True  
True  
True  
>>>
```

Testen von Funktionen

```
def teiler (a,b):  
    return a%b == 0
```

```
def testTeiler (a,b):  
    val = teiler(a,b)  
    for i in range(1,a):  
        if i*b == a and not val:  
            return False  
        elif i*b == a and val:  
            return True  
    if val:  
        return False  
    return True
```

Dilemma? Ist meine Testfunktion korrekt?

Testen von Funktionen

Andere Typen und Sonderfälle?

```
def teiler (a,b):  
    return a%b == 0
```

b = 0?

a = "Hi" b = "Hello"?

Traceback (most recent call last):

File "/Volumes/Nidaba HD/gitolite/lehre/SS18/ProInf3/VL3_examples/myFunctions.py",
line 32, in <module>

printTeiler(2,0)

File "/Volumes/Nidaba HD/gitolite/lehre/SS18/ProInf3/VL3_examples/myFunctions.py",
line 29, in printTeiler

print(a, '%', b, ' == ', teiler(a,b))

File "/Volumes/Nidaba HD/gitolite/lehre/SS18/ProInf3/VL3_examples/myFunctions.py",
line 12, in teiler

return a%b == 0

ZeroDivisionError: integer division or modulo by zero

>>>

Testen von Funktionen

- Funktionen sollten getestet werden
- Kein Beweis der Korrektheit eines Programmes!
- Testen von Randbedingungen, Sonderfällen, Typen von Eingaben, Wertebereiche...
- Testfälle dokumentieren und in eigener Funktion (Wiederverwendbarkeit!)
- Wer kontrolliert den Testfall? Eigenes Auge oder der Computer?

Reservierte Wörter in Python

help> keywords

and	else	import	raise
assert	except	in	return
break	exec	is	try
class	finally	lambda	while
continue	for	not	yield
def	from	or	False
del	global	pass	True
elif	if	print	...

assert-Anweisung in Python

Syntax:

```
assert condition1
```

```
assert condition1 [, expression2]
```

Beispiele:

```
>>> assert 1 == True
>>> assert 1 == False, "That can't be right."
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: That can't be right.
```

assert-Anweisung in Python

Beispiel:

```
myList = [1,2]
```

```
assert len(myList)>0
print(myList.pop())
```

```
assert len(myList)>0
print(myList.pop())
```

```
assert len(myList)>0
myList.pop()
```

```
>>>
```

```
2
```

```
1
```

Traceback (most recent call last):

File "assert_stm.py", line 11, in <module>

assert len(myList)>0

AssertionError

```
>>>
```

```
$ python -O assert_stm.py
```

```
2
```

```
1
```

Traceback (most recent call last):

File "assert_stm.py", line 12, in <module>

myList.pop()

IndexError: pop from empty list

CodingStyle

- Sinnvolle Benennung von Variablen und Funktionen
- Code mit Funktionen strukturieren
- Sinnvolle Kommentierung: Entscheidungen, Annahmen und Ansatz dokumentieren
- Mehr: <https://www.python.org/dev/peps/pep-0008/>

Fragen?
(Morgen mehr Tafel und weniger Folien!)