Einführung in die Programmierung

Ronja Düffel
WS2018/19

05. Oktober 2018

Rückblick

- Datentypen
  - `bool`
  - Zahlen (`int` und `float`)
  - `string`
- Variablen
- Kontrollstrukturen
  - Verzweigungen (`if...:` und `if...else:`)
  - Schleifen (`while...:` und `for...:`)

Aggregierte Datentypen

- Datentyp: Zusammenfassung von Objektmengen und der darauf definierten Operationen
  z.B. die elementaren Datentypen `bool`, `int`, `float`
- Aus anderen (elementare oder aggregierte) Datentypen zusammengesetzter Datentyp.
- um mehrere Daten, die zusammengehören zu verwalten
- in Python gibt es vordefinierte aggregierte Datentypen
  z.B. `set`, `tupel`, `dictionary`, `list`

Listen

- mehrere Werte unter einem Variablennamen zusammengefasst
- werden durch eckige Klammern [ ] angezeigt, Listenelemente werden durch Komma getrennt.

```
>>> liste_1 = [2,3,4,6,10]
>>>
>>> liste_2 = ['Joe', 'Jack', 'Alice']
>>>
>>> liste_3 = [0.5, 3, 'Blumen', 7.34]
```

- auf einzelne Werte kann über den Index zugegriffen werden

```
>>> liste_1[0]
2
>>> liste_3[-2]
'Blumen'
>>> liste_1[1]
3
>>>
```

Listen

Operator/ Funktion	Beschreibung
`<list>[x]`	Zugriff auf Element mit Index `x`
`<list>[x:y]`	Zugriff auf Teilliste von Index `x` bis `y`−1 (!!!)
`<list> + <list>`	zusammenfügen von Listen
`<list>.append(x)`	hinzufügen von x

```
>>> liste_1[1:4]
[3, 4, 6]
>>> liste_2 + liste_3
['Joe', 'Jack', 'Alice', 0.5, 3, 'Blumen', 7.34]
>>> liste_1.append('Bob')
>>> liste_1
[2, 3, 4, 6, 10, 'Bob']
>>>
```

Listen

`del <list>[x]`	löschen von Element mit Index `x`
`<list>.remove(x)`	löschen von Element x
`len(<list>)`	Länge der Liste

```
>>> liste_4=[1,5,3,2,5]
>>> del liste_4[3]
>>> liste_4
[1, 5, 3, 5]
>>> liste_4.remove(5)
>>> liste_4
[1, 3, 5]
>>> len(liste_4)
3
>>>
```

Funktionen

für Operationen die immer wieder gebraucht werden

+ Wiederverwertbarkeit
+ leichte Wartbarkeit
+ nur einmal schreiben
+ leicht auszutauschen
+ Übersichtlichkeit

Funktionen

- werden mit `def`-Anweisung definiert, Übergabeparameter in runden Klammern `( )` dahinter
- Funktionsrumpf muss eingerückt sein
- Ende der Funktion durch beenden der Einrückung
- Schlüsselwort `return` beendet die Funktion und veranlasst Zuweisung des Rückgabewerts

```
1 def add(a,b):
2     '''Addiere die Zahlen a und b'''
3     return a+b
```

- `return`-Anweisung ist optional. Hat die Funktion keinen Rückgabewert, so wird das Objekt `None` zurückgegeben.

Datentyp `None`

- hat nur einen einzigen Wert: `None`
- Schlüsselwort, dient als Platzhalter für Variablen, die keinen Wert haben oder deren Wert noch nicht bekannt ist
- Bei Auswertung von Ausdrücken in der Python-Shell wird nur etwas ausgegeben, wenn der Rückgabewert nicht `None` ist

Beispiel None

```
>>> empty = None
>>> empty # keine Ausgabe!
>>> print(empty)
None
>>> if empty is None:
        print("empty is: None")


empty is: None
>>> if empty == None:
        print("empty is: None")


empty is: None
>>>
```

Funktionen

- Funktionsdefinition muss im Code (lexikalisch) **vor** dem Aufruf erfolgen
- Übergabeparameter müssen beim Aufruf in der richtigen Reihenfolge angegeben werden

Beispiel:Parameterübergabe

```
1 def getNewBalance (now, spent, name):
2     amount = now - spent
3     owner = name
4     return ((owner, amount))
5
6 balance = getNewBalance('Bob', 500, 150)
7 print(balance)
```

```
Traceback (most recent call last):
  File "/media/ronja/Elements/WS1718/Material/Folien/prameter.py", line 6, in <m
odule>
    balance = getNewBalance('Bob', 500, 150)
  File "/media/ronja/Elements/WS1718/Material/Folien/prameter.py", line 2, in ge
tNewBalance
    amount = now - spent
TypeError: unsupported operand type(s) for -: 'str' and 'int'
>>>
```

Parameter benennen

```
1 def getNewBalance (now, spent, name):
2     amount = now - spent
3     owner = name
4     return ((owner, amount))
5
6 balance = getNewBalance(name='Bob', now=500, spent
      =150)
7 print(balance)
```

```
('Bob', 350)
>>>
```

Fehlersuche

- mit `type()` kann man sich den Datentyp einer Variablen ausgeben lassen

```
>>> zahl = '5'
>>> type(zahl)
<class 'str'>
>>>
```

Beispiel type()

```
1 side = input('Seitenlänge in cm: ')
2 area = side**2
3 print('Flächeninhalt:',area)
```

```
1 side = input('Seitenlänge in cm: ')
2 print('debug:',side, type(side))
3 area = side**2
4 print('Flächeninhalt:',area)
```

```
Seitenlänge in cm: 6
debug: 6 <class 'str'>
Traceback (most recent call last):
  File "/home/ronja/Uni/Lernzentrum/Vorkurs/WS1718/Material/Folie
n/typeErrorFind1.py", line 3, in <module>
    area = side**2
TypeError: unsupported operand type(s) for ** or pow(): 'str' and
'int'
>>>
```

Gültigkeitsbereiche

```
>>> def sowas():
        x = 0
        print("Wert von x:", x)
        return

>>> x = 5
>>> sowas()
Wert von x: 0
>>> 5/x
1.0
>>>
```

Gültigkeitsbereiche

- Variablenname ist in dem Anweisungsblock gültig, in dem er definiert wird.
- unterscheide zwischen *lokalen* (innerhalb Block/Funktion) und *globalen* (auch außerhalb) Variablen
- Verwendung globaler Variablen innerhalb von Funktionen mit `global`

Beispiel: global

```
>>> def sowas():
        global x
        x = 0
        print("Wert von x:", x)
        return

>>> x = 5
>>> sowas()
Wert von x: 0
>>> 5/x
Traceback (most recent call last):
  File "<pyshell#205>", line 1, in <module>
    5/x
ZeroDivisionError: division by zero
>>>
```

Beispiel: Gültigkeitsbereich

```
1 def varFunction():
2     var_2 = 5
3     print('varFunction')
4     return
5
6 var_1=5
7 varFunction()
8 var_3 = var_1 + var_2
9 print(var_2,var_3,var_1)
```

```
varFunction
Traceback (most recent call last):
  File "/media/ronja/Elements/WS1718/Material/Folien/nameError.py", line 8, in <
module>
    var_3 = var_1 + var_2
NameError: name 'var_2' is not defined
>>>
```

Module

Wiederverwendung von Funktionen in anderen Programmen :

- `import`
  - `import <Modulname>` (Dateiname ohne `.py`)
    Verwendung durch `<Modulname>.<Funktionsname>`
    (kein Namenskonflikt)
  - `from <Modulname> import <Funktionsname(n)>`
    Verwendung durch `<Funktionsname>`
    (!gleichnamige Funktionen werden überschrieben!)
  - `from <Modulname> import *`
    Alles wird importiert, gefährlich aber “bequem”

Beispiel import

Datei: sums.py

```
1  def add(a,b):
2      '''Addiere die Zahlen a und b'''
3      return a+b
4
5  def sum(n):
6      '''berechnet die Summe der ersten n natürlichen
           Zahlen'''
7      ergebnis = 0
8      for i in range(n+1):
9          ergebnis += i
10     return ergebnis
```

Beispiel import

Programm, mit dem der Nutzer die Summe der ersten *n* natürlichen Zahlen berechnen lassen kann

```
1  import sums  # importiere Modul sums
2
3  while True:
4      a = input("Geben Sie eine Ganzzahl > 0 ein: ")
5      if a.isdigit():
6          break
7
8  a_int = int(a)
9  result = sums.sum(a_int)
10 print("Die Summe von 1 bis", a ,"ist:",result)
```

Dateien lesen und schreiben

open() : öffnet eine Datei in angegebenem Modus

- `'r'`: Lesemodus
- `'w'`: Schreibmodus !Datei wird überschreiben !
- `'a'`: Schreibmodus, neue Daten werden am Ende hinzugefügt

read() : Lese den Inhalt der Datei; komplett, oder die angegebene Anzahl an Bytes

write() : Schreibt Daten in Datei. Zeilenumbruch muss explizit angegeben werden

close() : schließt Datei.

Beispiel

Datei `zahlen.txt`

```
3
8
19
2
5
23
7
```

Beispiel

```
1  # Programm das Zahlen aus einer Datei einliest
2  #  und sortiert wieder zurückschreibt.
3
4  file_name = input("Bitte Dateinamen eingeben: ")
5
6  numbers = [] # Liste für Zahlen anlegen
7  file = open(file_name, 'r')
8  for line in file:
9      numbers.append(int(line.replace('\n',"")))
10 file.close()
11 numbers.sort()
12 print(numbers)
13 file = open(file_name, 'w')
14 for i in numbers:
15     file.write(str(i) + '\n')
16 file.close()
```

Beispiel

```
Bitte Dateinamen eingeben: zahlen.txt
[2, 3, 5, 7, 8, 19, 23]
>>>
```

Datei `zahlen.txt`

```
2
3
5
7
8
19
23
```

Rekursion

Um Rekursion zu verstehen,
muss man erstmal Rekursion verstehen

- Methode etwas durch sich selbst zu definieren

Beispiel (Summe)

Die Funktion $f : \mathbb{N} \to \mathbb{N}$ *sei gegeben durch*

$$f(n) := \begin{cases} 0, & \text{falls } n = 0 \quad \text{Rekursionsanfang} \\ n + f(n-1), & \text{sonst.} \quad \text{Rekursionsschritt} \end{cases}$$

rekursive Programmierung

- Funktionen die sich selbst aufrufen (auch verschachtelt)
- Abbruchbedingung muss auch erreicht werden (Gefahr der Endlosschleife)

```
1 # Summe rekursiv
2
3 def sum_rek(n):
4     if n==0:
5         return 0
6     else:
7         return n + sum_rek(n-1)
```

Beispiel Summe

```python
# Summe iterativ

def sum_iter(n):
    result = 0
    for i in range(n+1):
        result += i
    return result
```

```python
# Summe rekursiv

def sum_rek(n):
    if n==0:
        return 0
    else:
        return n + sum_rek(n-1)
```

Fragen?