2. Prozedurale Programmierung

Prozedurale Programmierung

- Strukturierte **Datentypen**
- Was ist eine **Funktion**
- **Wie schreibt man Funktionen in Python**

Strukturierte Datentypen

- weitere Beispiele
- Listen
- Tupel
- Dictionaries

Listen

Operation	Erklärung
`s in x`	prüft, ob s in x ist
`s not in x`	prüft, ob s nicht in x ist
`x + y`	Verkettung von x und y
`x[n]`	liefert das n-te Element von x
`x[n:m]`	liefert eine Teilsequenz von n bis m
`x[n:m:k]`	liefert eine Teilsequenz von n bis m, aber nur jedes k-te Element wird berücksichtigt
`len(x)`	liefert die Anzahl von Elementen
`min(x)`	liefert das kleinste Element
`max(n)`	liefert das größte Element

Listen

```
1   myList = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
2   print(myList[:2])
3   print(myList[2:])
4   myList[5:] = ['a', 'b', 'c']
5   print(myList)
6
7   myList = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
8   myList[1:9] = 'x'
9   print(myList)
10
11
```

Tupel

```
1   tup = 1, 2, 'a'
2   print(tup)
3   print(tup[1])
4
5   for e in tup:
6       print(e)
7
8   '''
9       Was ist die Ausgabe, wenn man diese Zeile auskommentiert?
10  '''
11  #tup[1] = 'x'
12
13
```

Dictionaries

```
1   d = {'num':1,'den':2}
2   print(d)
3   print(d['num'])
4   d['num'] = 99
5   print(d['num'])
6
7   if 'num' in d:
8     print('We have num!')
9
10  del d['num']
11
12  if 'num' in d:
13    print('We have num!')
14
15
```

Zustand, Verhalten, Identität

Python: alle sind Objekte

Ein Objekt:

- Zustand (state)
- Verhalten (behavior)
- Identität

- unveränderlichen Grund-Datentypen (Zahlen, Strings, Tupel)...
- und veränderlichen Objekte Listen, Dictionaries...

- `id(objekt)`
- `type(objekt)`
- `isinstance(objekt,typ)`

Zustand, Verhalten, Identität

- in der realen Welt
- wir verwenden täglich viele verschiedene Objekte

Ein Objekt: Laptop

- Zustand (state): Dell (Hersteller), 14” (Bildschirm), Intel (CPU)
  - Eigenschaften
- Verhalten (behavior): anschalten, reset
  - Methoden
- Identität: 8FG89W2 (Serial Number)

Zustand, Verhalten, Identität - in Python

```
1   l = [1,2,3]
2
3   print (id(l)) # zB: 4566092872
4
5   v = [1,2,3,4]
6   print (id(l)) # zB: 4566829256
7
8   for el in l:
9     print(el) # 1,2,3
10
11  l.append(33)
12  l.pop()
13
14
```

Identität

Zustand

verhalten

unveränderlichen und veränderlichen Objekte


```
1   s = "abc"
2   print(id(s)) #4566030184
3
4   s = s + "d"
5   print(id(s)) #4566832720
6
7
8   l = [1,2,3]
9   print(id(l)) #4566832720
10
11
12  l.append(4)
13  print(id(l)) #4566832720
14
```

unveränderlichen und veränderlichen Objekte


```
1   myList = [1, 2, 3]
2   print(myList)
3   print(myList[1])
4
5   print('Die Liste enthält', len(myList), 'Elemente')
6   print('Das erste Element ist ', myList[0], 'und das letzte ist ', myList[len(myList) - 1])
7
8   x = myList
9   print(myList , x)
10
11  '''
12      Das output?
13  '''
14  x[1] = '?'
15  print(myList , x)
16
17
```

unveränderlichen und veränderlichen Objekte

```
1   myList = [1, 2, 3]
2   print(myList)
3   print(myList[1])
4
5   print('Die Liste enthält', len(myList), 'Elemente')
6   print('Das erste Element ist ', myList[0], 'und das letzte ist ', myList[len(myList) - 1])
7
8   x = myList
9   print(myList , x)
10
11  '''
12      Das output?
13  '''
14  x[1] = '?'
15  print(myList , x)
16
17
```

- [1, '?', 3] [1, '?', 3]
- die beiden Listen wurden geändert
- myList und x sind unterschiedliche Name für das gleiche Objekt

unveränderlichen und veränderlichen Objekte

- Unveränderliche Objekte können nach der Erstellung nicht mehr geändert werden
  - d.h. jede Änderung erzeugt ein neues Objekt

- Zugriff auf unveränderliche ist im Prinzip schneller
- Veränderlichen Objekte sind nützlich, wenn die Größe des Objekts geändert werden muss
- Unveränderliche Objekte werden verwendet, wenn man sicherstellen muss, dass das Objekt immer unverändert bleibt

- Unveränderliche Objekte sind grundsätzlich teuer zu „ändern“, da dazu eine Kopie erstellt werden muss.
- Das Ändern veränderlicher Objekte ist billig.

Prozedurale Programmierung

Ein **Programmierparadigma** = ein fundamentaler Programmierstil

Imperative Programmierung: das Programm wird als eine Reihe von Anweisungen geschrieben, die den Zustand des Programms ändern.

Zuweisung: `a = 10`

Prozedurale Programmierung: Programme werden aus eine oder mehreren Prozeduren bzw. Funktionen aufgebaut

Prozedurale Programmierung

Gemäß des **prozeduralen Paradigmas**

- wird der Zustand eines Programms mit Variablen beschrieben
- werden die möglichen Systemabläufe algorithmisch formuliert
- bilden Prozeduren/Funktionen das zentrale Strukturierungs- und Abstraktionsmittel

Prozedurale Programmierung. warum?

- man kann nicht verstehen, wo etwas endet oder etwas anderes beginnt
- schwer zu verstehen, zu erweitern
- ziemlich traurig

- jedes Teil hat ein klar definiertes Ziel
- leicht zu erweitern
- man kann alles verstehen
- Lasagna ist einfach besser :)

Funktionen

Funktion: etwas, das einen oder mehrere Werte nimmt und einen oder mehrere Werte zurückgibt

- Hat einen Namen
- Kann eine Liste von (formalen) Parametern haben
- Kann einen Rückgabewert
- Hat eine Spezifikation

Syntax

```
def <name>(P1, ..., Pn):
    #anweisungen
    return <ergebnis>
```

- Definition mit dem Keyword `def`
- man muss mit dem `()-Operator` die Funktion aufrufen
- `return` gibt den Wert zurück

Funktionen - Beispiel

```
def absolute_value(num):
    """Diese Funktion gibt den absoluten Wert
         der eingegebenen Zahl zurück"""

    if num >= 0:
        return num
    else:
        return -num

def main():
   print(absolute_value(2))
   print(absolute_value(-4))

main()
```

eine Funktion ohne Spezifikation ist nicht vollständig

```
1   def f (k):
2       v = 2
3       while v < k and k%v:
4           v += 1
5       return v>=k
```

- Könnt ihr bestimmen, was der Code ausgibt?
- Hat es länger als ein paar Sekunden gedauert?
- Jede Funktion hat eine Spezifikation, die besteht aus:
  - Eine kurze Beschreibung
  - Typ und Beschreibung aller Parameter
  - Bedingungen für Eingabeparameter
  - Typ und Beschreibung für den Rückgabewert
  - Bedingungen, die nach der Ausführung erfüllt sein müssen
  - Ausnahmen

Funktionen


```
1   def maximum (x,y ):
2     """
3     Gibt das Maximum von zwei Werten zurück
4     input: x,y - die Parameter
5     output: der größte der Parameter
6     Error: TypeError die Parameter dürfen nicht vergleichen werden
7     """
8     if x>y:
9       return x
10    return y
11
```

Funktionen

Jede Funktion **muss** enthalten:

- sinnvolle Namen (für Parameter und Namen)
- Kommentare
- Eine spezifikation

Testen!

- **man muss jede non-UI Funktion testen (kommt später)**

Übung

Gegeben sei eine Liste L mit n ≥ 2 positiven Zahlen. Alle Zahlen in der Liste seien unterschiedlich. Schreiben Sie eine Python-Funktion, die das zweitgrößte Element der Liste ausgibt.

Beispiel:

```
L = [1, 3, 23, 7, 5, 4, 11, 20]

second_best(l) -> 11
```

Optionale Parameter

```
1   def test(param = 'Hallo'):
2       print (param)
3
4
5   def main():
6       test()
7       test('World!')
8
9   main()
10
11  '''
12  output:
13  Hallo
14  World!
15  '''
16
```

Sichtbarkeit und Blöcke. Teil II

- **Block:** ein Programmabschnitt, der als eine Einheit ausgeführt wird
- Blöcke sind durch einen Einrückungslevel definiert bzw. markiert
- eine Funktion ist ein Block
- ein Block wird innerhalb eines Execution Frame (Aufrufrahmen) ausgeführt
- Wenn eine Funktion aufgerufen wird, wird ein neuer Execution Frame erstellt

Aufrufen einer Funktion (Execution Frame)

Ein Execution Frame enthält:

- Einige administrative Informationen (zum Debugging verwendet)
- Informationen über, wo und wie die Ausführung fortgesetzt wird
- Definiert zwei Namespaces, den lokalen und den globalen Namespace, die sich auf die Ausführung des Codeblocks auswirken
- Ein Namespace ist eine Zuordnung von Namen zu Objekten.
- Ein bestimmter Namespace kann von mehr als einem Execution Frame referenziert werden

Aufrufen einer Funktion (Execution Frame)

```
1   global_name = 10
2
3   def funktion ():
4       local_name = 100
5
6       print (global_name)
7       print (local_name)
8
9       print (locals(), globals())
10
11  funktion()
12
13
```

```
10
100
{'local_name': 100} {'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <_frozen_importlib_external.SourceFileLoader object at 0x7f700c0e7970>, '__spec__': None, '__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>, '__file__': 'main.py', '__cached__': None, 'global_name': 10, 'funktion': <function funktion at 0x7f700c0cc3a0>}
```

global vs local

```python
global_name = 10

def funktion ():
    local_name = 100

    global global_name

    global_name = 101

    print (global_name)
    print (local_name)



funktion()
print (global_name)

'''
Output

101
100
101
'''
```

```python
global_name = 10

def funktion ():
    local_name = 100

    global_name = 101

    print (global_name)
    print (local_name)



funktion()
print (global_name)

'''
Output

101
100
10
'''
```

Argumentübergabe

- Formale Parameter
  - ein Name für einen Eingabeparameter einer Funktion
  - Jeder Aufruf der Funktion muss für jeden obligatorischen Parameter einen entsprechenden Wert übergeben
- Tatsächliche Parameter
  - ein Wert, den der Aufrufer der Funktion für einen formalen Parameter bereitstellt
- Die tatsächlichen Parameter werden beim Aufruf in die lokale Symboltabelle der Funktion eingefügt

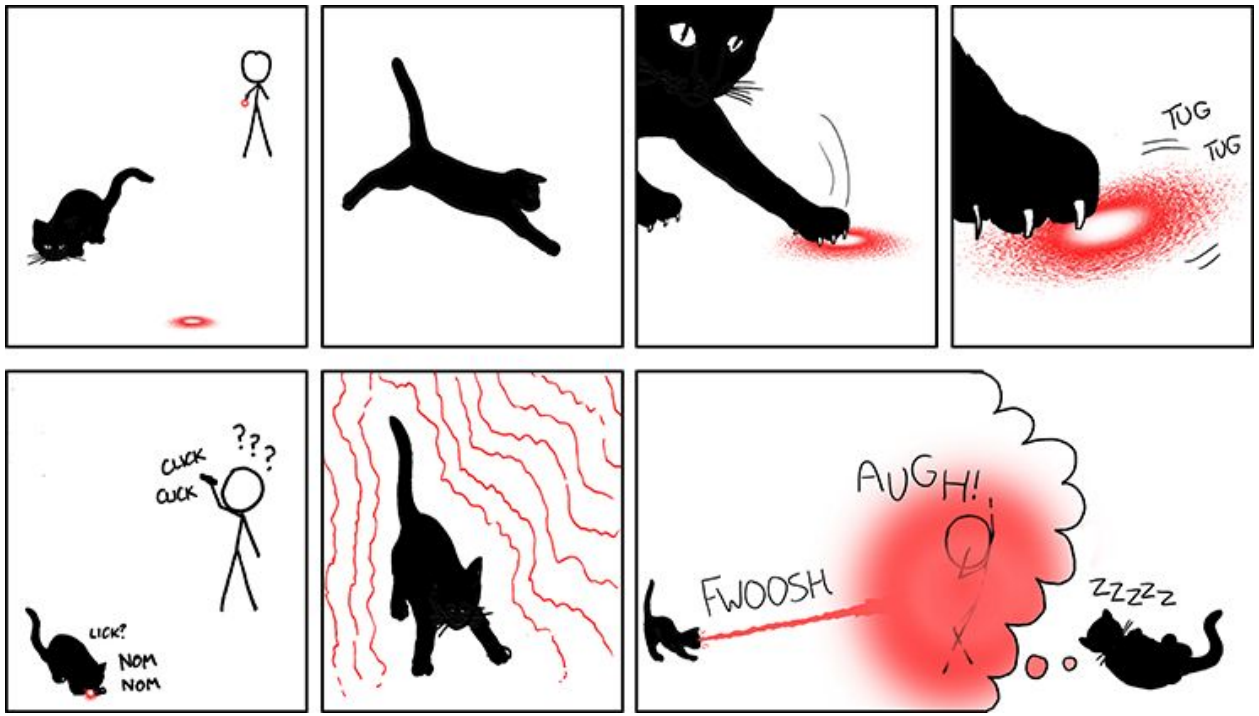
```
1  def test(param = 'Hallo'):
2      print (param)
3
4
5  def main():
6      test()
7      test('World!')
```

Formale Parameter

Tatsächliche Parameter

Argumentübergabe

- die Frage ist: sind Änderungen an einem Parameter für den Aufrufer sichtbar?
- in C++ oder Pascal war die Situation einfach
- man hat `var`, `&`, `*` und andere syntaktische Mechanismen
- aber in Python? leider nicht so einfach :)



http://xkcd.com/729/

Argumentübergabe

- Pass by Value - eine Kopie des Parameters wird an den formalen Parameter der Funktion gebunden
- Pass by Reference - Die Funktion erhält eine Referenz auf das eigentliche Argument statt eine Kopie
- Side Effect - Eine Funktion, die die Umgebung des Anrufers ändert (neben der Erzeugung eines Rückgabewerts), soll Side Effects haben

Argumentübergabe

```
1   def refDemo(x):
2       print("2. x=", x, " id=", id(x))
3       x = 42
4       print("3. x=", x, " id=", id(x))
5
6   x = 10
7   print("1. x=", x, " id=", id(x))
8
9   refDemo(x)
10  print("4. x=", x, " id=", id(x))
11
12
```

```
1. x= 10  id= 140197939083264
2. x= 10  id= 140197939083264
3. x= 42  id= 140197939084288
4. x= 10  id= 140197939083264
```

Argumentübergabe

```python
def noSideEffect(lst):
    print(lst)
    lst = [0, 1, 2, 3]
    print(lst)

def sideEffect(lst):
    print(lst)
    lst += [0, 1, 2, 3]
    print(lst)

fib = [0, 1, 1, 2, 3, 5, 8]
noSideEffect(fib)
# sideEffect(fib)
print(fib)
```

```
[0, 1, 1, 2, 3, 5, 8]
[0, 1, 2, 3]
[0, 1, 1, 2, 3, 5, 8]
```