Python Einführungskurs

Einführung in die Numerik
Sommersemester 2022

27. April 2022

2. Übung

Programme und Debugging

Python-Skripte

1. Programm speichern (z. B. `file.py`)
2. Ausführen im Terminal:

```
python3 myprogram.py
```

3. im Interpreter:

```
>>> import myprogram
```

Alternativ:

```
from ... import ... [as ...]
import ... [ as ...]
```

Beispiel: `script.py`

Wiederholung

- Variablen `x = 1`
- Variablen-Typen
  - Int,float,bool
  - Strings
- Container
  - Listen `a = [1,2,3]`
  - Tupel `a = (1,2,3)`
  - Dictionaries `a = {'a':1,'b':2}`
- `if-else`-statements
- loops
  - `for`-loop
  - `while`-loop

Übungen

1. Erstellen Sie eine Liste mit 20 Einträgen, die bei 0^3 beginnt und mit 19^3 endet.
2. Printen Sie jeden Eintrag in der Liste.
3. Printen Sie jeden *zweiten* Eintrag in der Liste
4. Printen Sie jeden Eintrag, der durch 4 Teilbar ist und kleiner als 200 ist
5. Printen Sie die *laufende* Summe über die Liste

Lösungen

```
# a)
our_list = [i**3 for i in range(20)]
# b)
for val in our_list:
  print(val)
index = 0
# c)
while index < len(our_list):
  if index % 2 ==0:
    print(our_list[index])
  index += 1
```

Lösungen 2

```
# d)
our_list = [i**3 for i in range(20)]
summ = 0
  for val in our_list:
    summ += val
    print(summ)
```

Funktionen

Funktion definieren mit `def`
Quadrierere Zahl `x`:

```
>>> def square(x):
...     y = x**2
...     print("x**2 = {}".format(y))
```

Aufrufen der Funktion

```
>>> square(15)
```

return

Rückgabe mit `return`

```
>>> def square(x):
...     y = x**2
...     print("x**2 = {}".format(y))
...     return y
```

Funktion aufrufen und einer Variablen zuweisen

```
>>> y_res = square(15)
```

Default Arguments

Falls Parameter nicht übergeben wird:

```
>>> def square(x=1):
...     y = x**2
...     return y
>>> y_res = square()
>>> z_res = square(5)
```

Mehrere Eingangsvariablen

Berechnung der Potenz x^n

```
>>> def power(x, n=2):
...     if n == 0:
...         return 1
...     y = x
...     for i in range(n-1):
...         y *= x
...     return y
...
```

Funktion aufrufen

```
>>> x = power(2, 10)
```

Modulare Programmierung

- Programme systematisch in logische Teilblöcke aufspalten

```
from ... import ... [as ...]
import ... [ as ...]
```

- Code erneut verwendbar ($\rightarrow$ große packages)

mytools.py

```
def print_matrix(A):
    for i in range(len(A)):
        for j in range(len(A[i])):
            print(A[i][j], end=' ')
            if j == len(A)-1:
                print()
M = [[1, 2, 3],
     [4, 5, 6],
     [7, 8, 9]]
print_matrix(M)
```

main.py

```
from mytools import print_matrix

A = [[1, 2, 3],
     [4, 5, 6],
     [7, 8, 9]]

print_matrix(A)
```

main2.py

```
import mytools

A = [[1, 2, 3],
     [4, 5, 6],
     [7, 8, 9]]

mytools.print_matrix(A)
```

main3.py

```
from mytools import print_matrix as pm

A = [[1, 2, 3],
     [4, 5, 6],
     [7, 8, 9]]

pm(A)
```

main4.py

```
from mytools import *

A = [[1, 2, 3],
     [4, 5, 6],
     [7, 8, 9]]

print_matrix(A)
```

docstrings

```
>>> def square(x):
...     """ prints the square value of x

        """
...     y = x**2
...     print("x**2 = {}".format(y))
```

dann:

```
>>> help(square)
```

mit "q" verlassen

Coding Style

- selbsterklärende Variablennamen
- Kommentare erklären Hintergrund

```
>>> k = 1 # setting k to 1
>>> k = 1 # iteration counter
```

- Einrückungen helfen
- Code ggf. in Module aufteilen

Debugging

- Wo klappt es noch?
- `breakpoint()` startet Interpreter
- Beispielcode zum Debuggen:

```
x = input("Input value to be squared:")
y = x**2
print("x**2 = "+ y)
```

Fibonacci

Fibonacci-Folge a_n ist definiert durch:

$$a_n = \begin{cases} 1, & n = 0, 1, \\ a_{n-1} + a_{n-2}, & \text{else.} \end{cases}$$

Aufgabe

Implementiere eine Funktion `fib(n)` die die n-te Fibonacci-Zahl a_n zurückgibt. Implementiere eine Funktion `fib(n)` die die n-te Fibonacci-Zahl a_n zurückgibt.

Rekursion

Fibonacci-Folge:

```
>>> def fib(n=10):
...     if n == 0 or n==1:
...         return 1
...     else:
...         return fib(n-1) + fib(n-2)
```

Fibonacci-loop

```
>>> def fib(n=10):
        fibo = [1,1]
...     for i in range(2,n+1):
...       fibo += fibo[i-1] + fibo[i-2]
...     return fibo[-1] # fibo[n]
```

Fibonacci-loop verbessert

```
>>> def fib(n=10):
        a = 1
        b = 1
...     for i in range(2,n):
...       a,b = b,a+b
...     return b
```