Einführung in Python

3. Vorlesung

Spezielle Funktionen Datei Ein-/Ausgaben Systemfunktionen

Wiederholung letztes Mal

- Bedingungen und logische Opertoren

```
>>> not (2 > 3) and (1 != 1)
False

>>> not ((2 > 3) and (1 != 1))
True
```

- Bedingte Anweisungen & Schleifen

```
if a > b:
    print('a ist größer als b')
elif a < b:
    print('a ist kleiner als b')
else:
    print('a und b sind gleich groß')
```

```
for i in range(5):
    print(i*i, end=' ')
0 1 4 9 16
```

```
while a > 1:
    a *= 2
```

Wiederholung letztes Mal

- Abfangen von Laufzeitfehlern

```
try:
    zahl = int(input('Bitte gib eine ganze Zahl ein: '))
    print('Danke für die Zahl')
except ValueError:
    print('Eingabe nicht in Ordnung.')
```

- Funktionen

Funktionskopf

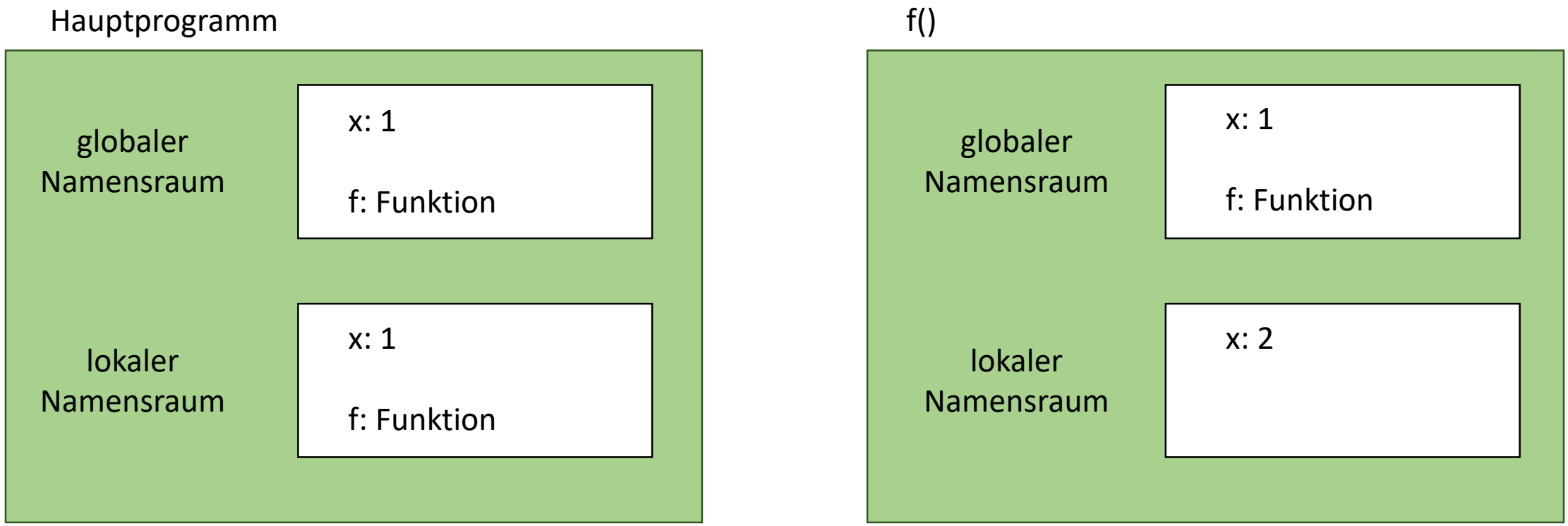
def *funktionsname* (*parameterliste*)**:**

(Einrückung →) Anweisungsblock

Funktionskörper

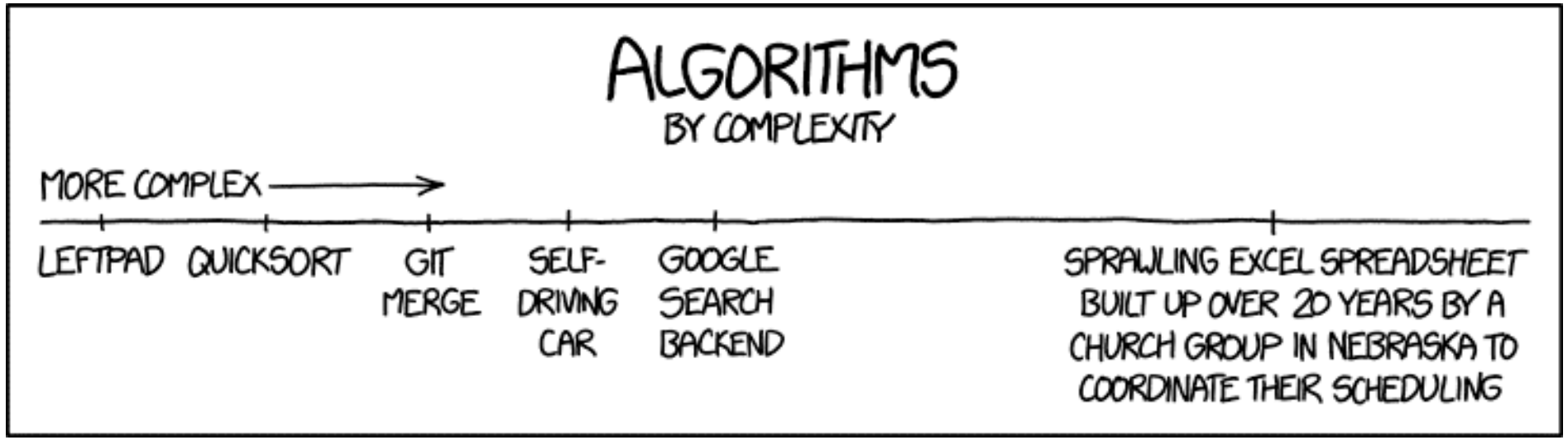
Wiederholung letztes Mal

- Namensräume und Seiteneffekte

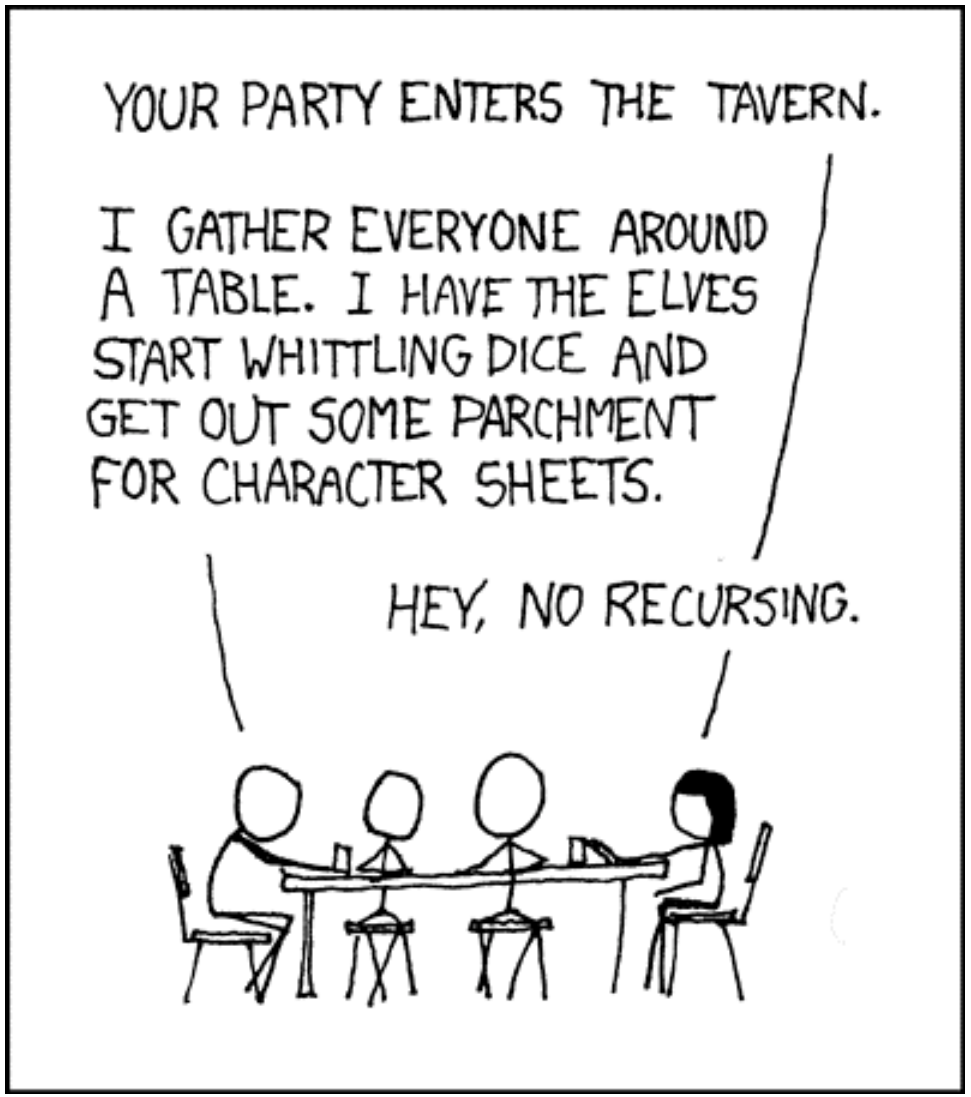


Spezielle Funktionen

- Rekursive Funktionen
- Lokale Funktionen
- Lambda Funktionen
- Generatoren



Rekursive Funktionen



Rekursive Funktionen

- Rekursive Funktionen sind Funktionen die sich selbst aufrufen

- Bestehen immer aus dem *Elementarfall* und dem *Rekursionsaufruf*

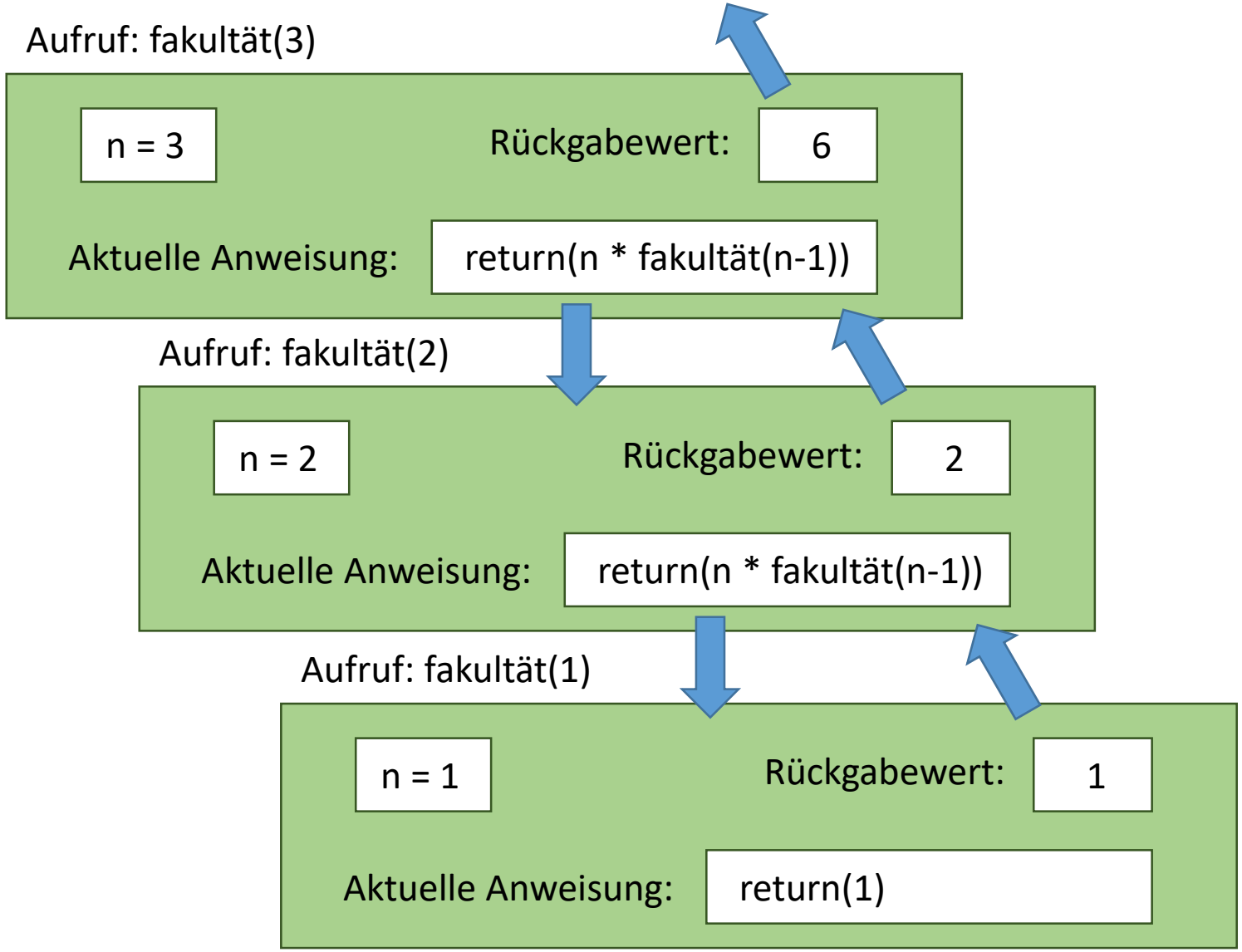
```
>>>def fakultät(n):
            if n <= 1:                          #Elementarfall
                return(1)
            else:
                return(n*fakultät(n-1))     #rekursiver Aufruf


>>> print(fakultät(19))
121645100408832000
```

Rekursive Funktionen – Execution Frames

- *Execution Frame*: ein Objekt, das den augenblicklichen Zustand der Ausführung einer Funktion repräsentiert

- Enthält immer folgende Informationen:
  - Aktuelle Anweisung, die gerade ausgeführt wird
  - Ein Verweis auf den übergeordneten Execution Frame, in dem die Funktion aufgerufen wurde
  - Lokaler Namensraum mit den aktuellen Werten der lokalen Variablen

Rekursive Funktionen – Execution Frames



Rekursive Funktionen – Rekursionstiefe

- Rekursive Funktionen benötigen viel Speicherplatz um Execution Frames zu speichern
- Arbeiten häufig ineffizient

→ Der Python-Interpreter hat eine Obergrenze für Rekursionstiefen

```
>>> i = 0
>>> def f():
        global i
        i += 1
        f()

>>> f()
RecursionError: maximum recursion depth exceeded
>>> i
995
```

Rekursive Funktionen – Rekursionstiefe

```
>>> i = 0
>>> def fibo(n):
        global i
        i += 1
        print('Aufruf Nr. ' +str(i) +'   fibo(' +str(n) +')')
        if n in [1,2]:
            return(1)
        else:
            return(fibo(n-2) + fibo(n-1))
>>> fibo(5)
Aufruf Nr. 1   fibo(5)
Aufruf Nr. 2   fibo(3)
Aufruf Nr. 3   fibo(1)
Aufruf Nr. 4   fibo(2)
Aufruf Nr. 5   fibo(4)
Aufruf Nr. 6   fibo(2)
Aufruf Nr. 7   fibo(3)
Aufruf Nr. 8   fibo(1)
Aufruf Nr. 9   fibo(2)
5
```

Lokale Funktionen

- Lokale Funktionen werden innerhalb einer anderen Funktion definiert
  → sind von außen nicht erreichbar
- Achtung: Dadurch entstehen auch verschachtelte Namensräume

```
>>> def äussere_funktion():
        def innere_funktion():
            print(x)

        x = 2
        innere_funktion()

>>> x = 3
>>> äussere_funktion()
2

>>> innere_funktion()
NameError: name 'innere_funktion' is not defined
```

Lokale Funktionen - Beispiel

```
>>> def fakultät(n):
        #Fehlerbehandlung
        if not isinstance(n, int):
            raise TypeError('n muss ein Integer sein!')
        if not n >= 0:
            raise ValueError('n muss 0 oder positiv sein')

        def innere_funktion(n):
            if n <= 1:
                return(1)
            return(n*innere_funktion(n-1))
        return innere_funktion(n)


>>> print(fakultät(4))
24
```

Lokale Funktionen - Beispiel

```
>>> def closure():
      container = [0]                    #Diese Liste existiert nur im lokalen
                                         #Namensraum der closure()-Funktion
      def increase():
            container[0] += 1
      def get():
            return(container[0])
      return(increase, get)

>>> i, g = closure()
>>> g()
0
>>> i()
>>> i()                                  #Da i() und g() Referenzen auf die lokalen
>>> g()                                  #increase()- und get()-Funktionen sind
2                                        #haben sie Zugriff auf die Variable container
```

Lokale Funktionen - Beispiel

```
>>> def generiere_potenz(basis):          #factory function
        def nte_potenz(exponent):
            return(basis ** exponent)
    return(nte_potenz)


>>> zweite_potenz = generiere_potenz(2)
>>> dritte_potenz = generiere_potenz(3)
>>> vierte_potenz = generiere_potenz(4)

>>> zweite_potenz(4)
16
>>> dritte_potenz(3)
27
>>> vierte_potenz(2)
16
```

Lambda-Funktionen

- Lambda-Funktionen sind anonyme Funktionen (haben keinen Namen), mit einer beliebigen Anzahl an Argumenten, **einem** Ausdruck und den Wert des Ausdrucks als Rückgabewert

lambda x, y: x + y

```
>>> (lambda x,y: x+y) (5,14)
19

>>> summe = lambda x,y: x+y
>>> summe(5, 14)
19
```

Lambda-Funktionen

map(*funktion*, *sequenz*)

```
>>> celsius = [39.2, 36.5, 37.3, 37.8]
>>> fahrenheit = map(lambda x: (9/5)*x+32, celsius)
>>> list(fahrenheit)
[102.56, 97.7, 99.14, 100.03999999999999]


>>> a = [1,2,3,4]
>>> b = [17,12,11,10]
>>> c = [-1, -4, 5, 9]
>>> list(map(lambda x,y,z: x+y-z, a,b,c))
[19, 18, 9, 5]
```

Lambda-Funktionen

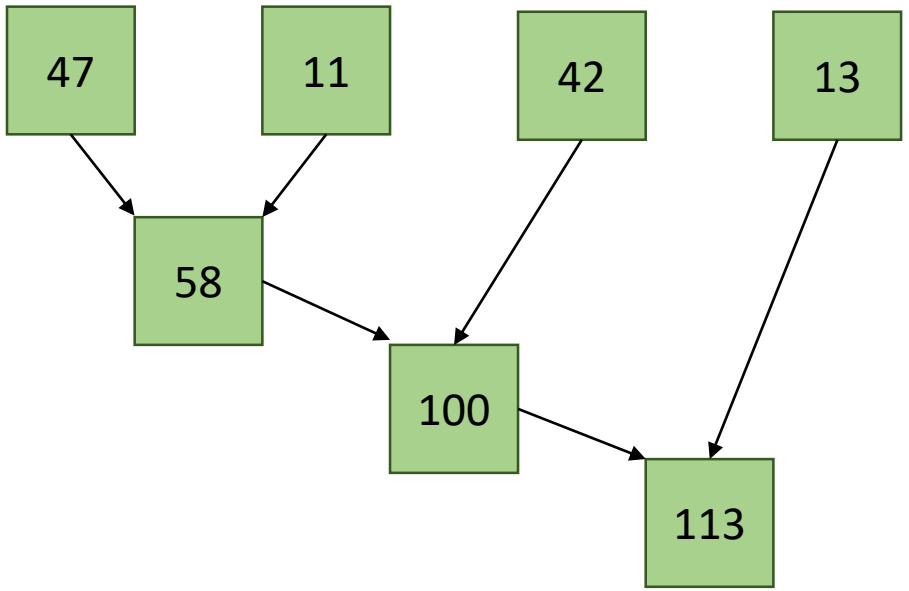
filter(*funktion*, *liste*)

```
>>> fib = [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
>>> result = filter(lambda x: x % 2, fib)
>>> list(result)
[1, 1, 3, 5, 13, 21, 55]


>>> result = filter(lambda x: x % 2 == 0, fib)
>>> list(result)
[0, 2, 8, 34]
```

Lambda-Funktionen

reduce(*funktion*, *sequenz*)

```
>>> from functools import reduce

>>> reduce(lambda x,y: x+y, [47, 11, 42, 13])
113
```

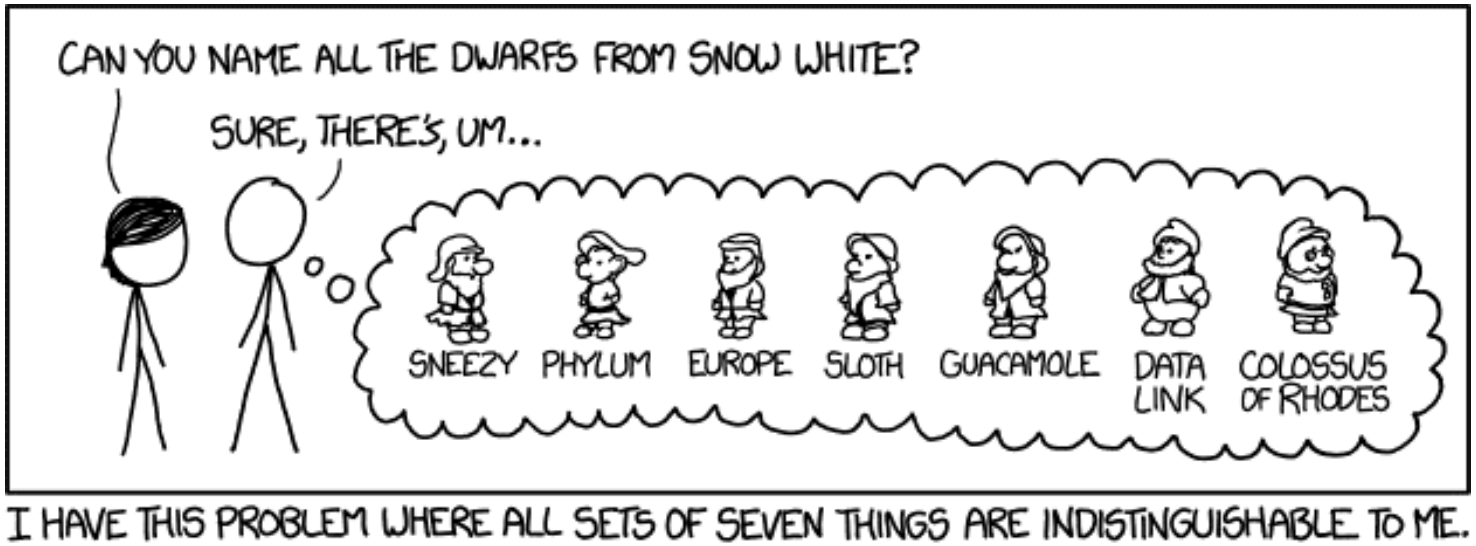


Lambda-Funktionen

sort(*key*)

```
>>> l = [(2,‘C‘),(3,‘A‘),(1,‘B‘)]
>>> l.sort()
>>> l
[(1, 'B'), (2, 'C'), (3, 'A')]


>>> l.sort(key= lambda x: x[1])
>>> l
[(3, 'A'), (1, 'B'), (2, 'C')]
```

Generatoren & Iteratoren



Generatoren

```
>>> a = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81] #Schreibarbeit, belegt viel Speicher

>>> b = [i*i for i in range(10)]              #belegt immer noch viel Speicher
>>> b
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

>>> c = (i*i for i in range(10))              #Generatorausdruck
>>> c
<generator object <genexpr> at 0x034311E0>    #belegt nur wenig Speicher

>>> for n in c:
        print(n, end=' ')
0 1 4 9 16 25 36 49 64 81
```

Generatoren

- Ein Generator ist eine abstrakte Kollektion von Objekten, bei der die Objekte nicht explizit aufgezählt werden
- Eine *Konstruktionsvorschrift* gibt an wie bei Bedarf jedes Element generiert werden kann
- Generatorfunktionen erzeugen **Generatorobjekte**

```
>>> def generiere_zahlen(n):
        for i in range(n):
            yield(i*i)                  #statt return ist yield das Schlüsselwort

>>> g = generiere_zahlen(10)
>>> g
<generator object generiere_zahlen at 0x034311E0>
```

Generatoren

- Generatoren merken sich immer nur den **aktuellen** Zustand des zur Funktion gehörenden Prozesses → Generatoren lassen sich nur einmalig benutzen

```
>>> g = generiere_zahlen(3)
>>> next(g)
0
>>> next(g)
1
>>> next(g)
4
>>> next(g)                          #der Generator ist leer
StopIteration
```

Generatoren

- Mit Generatoren lassen sich unendlich große (abstrakte) Kollektionen erzeugen

```
>>> def generiere_zahlen():
        i = 1
        while True:                    #Endlosschleife
            yield(i*i)
            i += 1

>>> g = generiere_zahlen()
>>> for n in g:                        #Hier hängt sich das Programm auf
        print(n)
```

Iteratoren

- Iteratoren sind spezielle Generatoren, die den Zugriff auf die Elemente einer Kollektion kontrollieren
- Ein Iterator zu einer Kollektion gibt **einmal nach und nach** die Elemente der Kollektion wieder

```
>>> s = [1, 2, 3, 4]
>>> i1 = iter(s)
>>> i2 = iter(s)
>>> next(i1)
1
>>> next(i1)
2
>>> next(i2)
1
```

```
>>> def i1():
        yield(‘1‘)
        yield(‘2‘)
        yield(‘3‘)
        yield(‘4‘)
>>> next(i1)
1
>>> next(i1)
2
```

Good programming practice: Lektion V

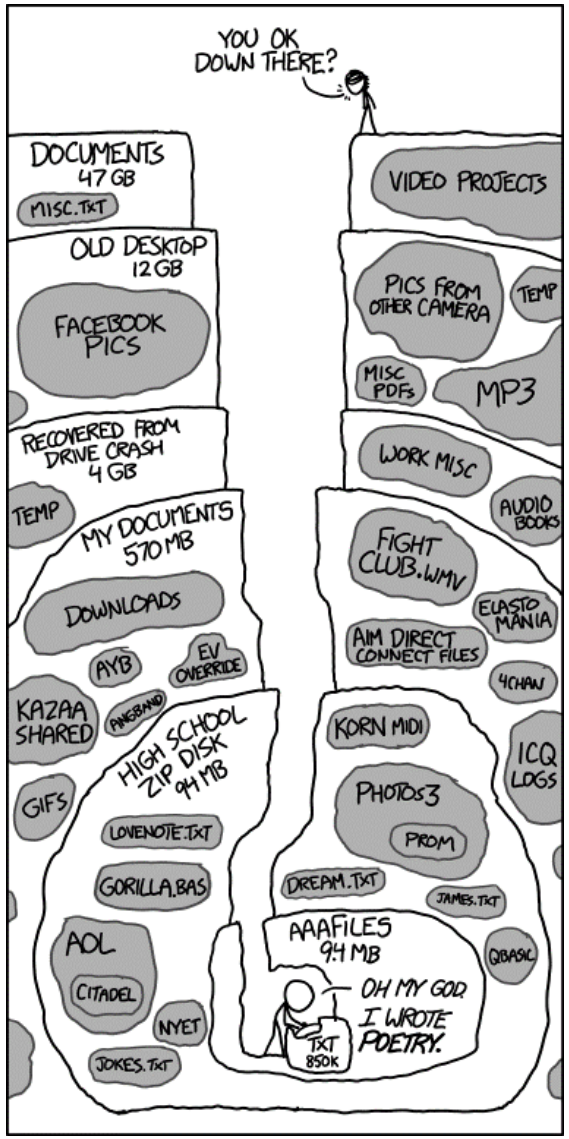
Good programming practice: Lektion V

- Benutzt *Docstrings* zur Beschreibung eurer Funktionen
- Inhalt: Knappe Beschreibung der Aufgabe der Funktion, Vorbedingungen, Nachbedingung, globale Variablen

```
>>> def coole_funktion():
        """ Diese coole Funktion macht nichts """
        pass

>>> help(coole_funktion)
Help on function coole_funktion in module __main__:

coole_funktion()
    Diese coole Funktion macht nichts

>>> coole_funktion.__doc__
' Diese coole Funktion macht nichts '
```

Ein- und Ausgabe

- File-Objekte
- Files lesen und schreiben
- Beliebige Daten speichern mit *pickle*
- Pseudofiles



File-Objekte

- Alle Ein-/Ausgaben verlaufen in Python über Objekte des Typs *File*
- Zu speichernde Daten werden zuerst in ein File-Objekt geschrieben, danach mit Hilfe des Betriebssystems (physisch) gespeichert
- Auch eingelesene Daten werden in ein File-Objekt geschrieben und können danach verarbeitet werden

File-Objekte

- File-Objekte werden folgendermaßen erzeugt:

open(*filename*, modus ,[Kodierung])

Modus	Erklärung
r	Die Datei wird ausschließlich zum Lesen geöffnet. Falls die Datei nicht existiert erhält man beim Öffnen, eine Fehlermeldung
w	Falls eine Datei mit dem gleichen Namen schon existiert, wird deren Inhalt gelöscht und neu beschrieben. Andernfalls wird eine neue Datei zum Schreiben angelegt.
a	Die Datei ist zum anhängen neuer Daten bestimmt. Der bisherige Inhalt wird nicht gelöscht, es wird am Ende weiter geschrieben.
rb	Die Datei wird im Binärmodus zum Lesen geöffnet. Das heißt, beim Lesen werden Bytestrings zurückgegeben.
wb	Die Datei wird im Binärmodus zum Lesen geöffnet. Das heißt, es können nur Bytestrings (und keine normalen Strings) in das File geschrieben werden.

File-Objekte

```
>>> daten = open(‘/home/python/tolles_file.txt‘, ‘r‘)
>>> daten
<_io.TextIOWrapper name='/home/python/tolles_file.txt' mode='r,
encoding='cp1252'>

>>> daten.close()

>>> daten = open(‘tolles_file.txt‘)
>>> daten
<_io.TextIOWrapper name='/home/tolles_file.txt' mode='r, encoding='cp1252'>

>>> daten.close()
```

Zugriff auf Files

Attribute und Methoden	Erklärung
close()	Die Datei wird geschlossen und gespeichert.
closed	Das Attribut hat den Wert **True**, falls die Datei geschlossen ist und sonst **False**.
flush()	Die Datei wird gespeichert, aber nicht geschlossen.
mode	Das Attribut enthält den I/O Modus. (r, w, a, rw, rb)
read(*[bytes]*)	Der Inhalt der Datei wird gelesen und als String zurückgegeben (höchstens ein Stück der Länge *bytes*, falls dieser Parameter angegeben ist).
readline()	Die nächste Zeile wird eingelesen und als String wiedergegeben.
readlines()	Die Datei wird zeilenweise gelesen und als eine Liste von Strings zurückgegeben.
seek(*pos*)	Der Cursor wird auf die angegebene Position gesetzt.
tell()	Die Methode liefert die aktuelle Cursorposition.
write(*str*)	Die Zeichenkette *str* wird in die Datei geschrieben.

Zugriff auf Files

```
>>> output = open('/home/python/out_file.txt', 'w')  #File-Objekt zum Schreiben
>>> text = input('Bitte gib deinen Namen ein: ')
>>> output.write(text +'\n')                  #Schreibe den String ins File-Objekt
>>> output.close()              #Schließe das File-Objekt und schreibe alle Daten

>>> output = open('/home/python/out_file.txt', 'a')
>>> output.write('Die fabelhafte Welt\n')
>>> output.flush()                #Schreibe die Daten ohne das File zu schließen
>>> output.closed
False
>>> output.write('der Pythonprogrammierung!\n')
>>> output.close()

>>> output.closed                    #Attribute ob das File geschlossen ist
True

>>> output.mode              #Attribute in welchem Modus das File geöffnet wurde
'a'
```

Zugriff auf Files

```
>>> infile = open('/home/python/out_file.txt')  #File-Objekt zum Lesen
>>> type(infile)
<class '_io.TextIOWrapper'>
>>> for line in infile :
        print(line)
Monty
Die fabelhafte Welt
der Pythonprogramierung!
>>> infile.close()

>>> infile = open('/home/python/out_file.txt')
>>> infile.readline()                          #Lesen der nächsten Zeile als str
Monty
>>> infile.readline()
Die fabelhafte Welt
>>> infile.readline()
der Pythonprogramierung!
>>> infile.readline()
''
```

Zugriff auf Files

```
>>> infile = open(‘/home/python/out_file.txt‘).readlines() #File Inhalt als list
>>> type(infile)
<class 'list'>
>>> infile
[‘Monty\n‘, ‘Die fabelhafte Welt\n‘, ‘der Pythonprogramierung!\n‘, ‘‘]


>>> infile = open(‘/home/python/out_file.txt‘).read()      #File Inhalt als str
>>> type(infile)
<class 'str'>
>>> infile
‘Monty\nDie fabelhafte Welt\nder Pythonprogramierung!\n‘

>>> infile.close()                                  #File ist bereits geschlossen
AttributeError: 'str' object has no attribute 'close'
```

Zugriff auf Files

```
>>> output = open('/home/python/out_file.txt', 'a')
>>> output.tell()                                  #Gib die Position des Cursors wieder
50

>>> output.write('onty')
>>> output.tell()
54

>>> output.seek(50)                              #Setze Cursors auf Position 50
>>> output.write('M')
>>> output.seek(0, 2)                      #Setze Cursors relativ zum File Ende auf Pos 0
>>> output.close()
```

seek(x, 0) → Setze Cursor x Positionen rechts von aktueller Position

seek(x, 1) → Setze Cursor x Positionen rechts vom Start

seek(x, 2) → Setze Cursor x Positionen rechts vom Ende

Zuverlässigkeit beim Lesen und Schreiben

- Sollte der Pfad zu einer zu lesenden oder schreibenden Datei ungültig sein gibt es einen Laufzeitfehler
- Mit *try…catch* oder *try… finally* Anweisungen kann man entsprechenden Code sichern

```
>>> try:
        e = open('/pfad/existiert/nicht/file_auch_nicht.dat')
    except:
        print('Das File konnte nicht geöffnet werden.')
```

Zuverlässigkeit beim Lesen und Schreiben

- Sollte der Pfad zu einer zu lesenden oder schreibenden Datei ungültig sein gibt es einen Laufzeitfehler
- Mit *try…catch* oder *try… finally* Anweisungen kann man entsprechenden Code sichern

```
>>> daten = ‘Sehr wichtige Daten‘
>>> try:
        e = open(‘/pfad/existiert/nicht/file_auch_nicht.dat‘, ‘w‘)
        e.write(daten)
        e.close()
    finally:
        f = open(‘/tmp/backup.dat‘, ‘w‘)
        f.write(daten)
        f.close()
```

Zuverlässigkeit beim Lesen und Schreiben

- Bestimmte Objekte besitzen die Methoden *__enter__()* und *__exit__()*
- Bei File-Objekten entspricht ersteres dem Öffnen und letzteres dem Schließen des Files
- Die ***with*-Anweisung** sorgt dafür das immer *__exit__()* ausgeführt wird wenn *__enter__()* erfolgreich war

```
>>> with open('/home/python/wichtige_daten.dat') as f:
        daten = f.readlines()
        [...]
```

Speichern beliebiger Daten auf Files

```
>>> wichtigeListe = [‘E=mc2‘, 19, 36.149]
>>> output = open(‘/home/python/wichtige_daten.dat‘, ‘w‘)
>>> output.write(str(wichtige_liste))
>>> output.close()

>>> wichtigeListe = open(‘/home/python/wichtige_daten.dat‘).read()
>>> wichtigeListe
“[‘E=mc2‘, 19, 36.149]“
>>> list(wichtigeListe)
['[', "'", 'E', '=', 'm', 'c', '2', "'", ',', ' ', '1', '9', ',', ' ', '3', '6',
'.', '1', '4', '9', ']']


>>> from ownHTML import OwnHTML
>>> a = OwnHTML()
>>> str(a)
‘<ownHTML.OwnHTML object at 0x7f2b8a663898>‘
```

Speichern beliebiger Daten auf Files

- Das Modul *pickle* erlaubt es (fast) beliebige Objekte zu speichern und auch wieder zu laden:
  - Alle (beliebig verschachtelten) Standarddatentypen
  - Funktionen
  - Instanzen von Standardmodule
  - Objekte von selbsterstellten Klassen

- Zum speichern: **dump**(*Objekt, File*)
- Zum laden: **load**(*File*)

Speichern beliebiger Daten auf Files

```
>>> import pickle
>>> wichtigeListe = [‘E=mc2‘, 19, 36.149]
>>> pickle.dump(wichtigeListe, open(‘/home/documents/il.pydmp‘, ‘wb‘))

>>> listeReloaded = pickle.load(open(‘/home/documents/il.pydmp‘, ‘rb‘))
>>> listeReloaded
[‘E=mc2‘, 19, 36.149]
>>> type(listeReloaded)
<class 'list'>

>>> s = pickle.dumps(wichtigeListe)
>>> s
b'\x80\x03]q\x00(X\x05\x00\x00\x00E=mc2q\x01K\x13G@B\x13\x12n\x97\x8dPe.‘

>>> s = pickle.loads(s)
>>> s
[‘E=mc2‘, 19, 36.149]
```

Pseudofiles

- Pseudofiles sind File-Objekte mit eingeschränkten Zugriffsmöglichkeiten für die Kommunikation mit dem Laufzeit-System

- Zugriff erhält man über das Modul *sys*

- Es gibt:
  - sys.stdin Standardeingabe, kann nur gelesen werden
  - sys.stdout Standardausgabe, kann nur beschrieben werden
  - sys.stderr Standardfehlerausgabe, kann nur beschrieben werden

Pseudofiles

```
>>> import sys
>>> print('Eingabe: ', end=' ')
>>> eingabe = sys.stdin.readline()                #Simulierung der input()-Funktion
>>> print('Folgende Eingabe wurde gemacht: ' +eingabe)
Folgende Eingabe wurde gemacht: Python

>>> sys.stdout.write('Das simuliert die print()-Funktion.')
Das simuliert die print()-Funktion.

>>> nonstdout = open('/home/documents/sys_log.text', 'w')
>>> sys.stdout = nonstdout
>>> print('Dieser Text landet nicht mehr auf der Konsole.')

>>> sys.stderr.write('Hilfe, ein Fehler!')
Hilfe, ein Fehler!
```

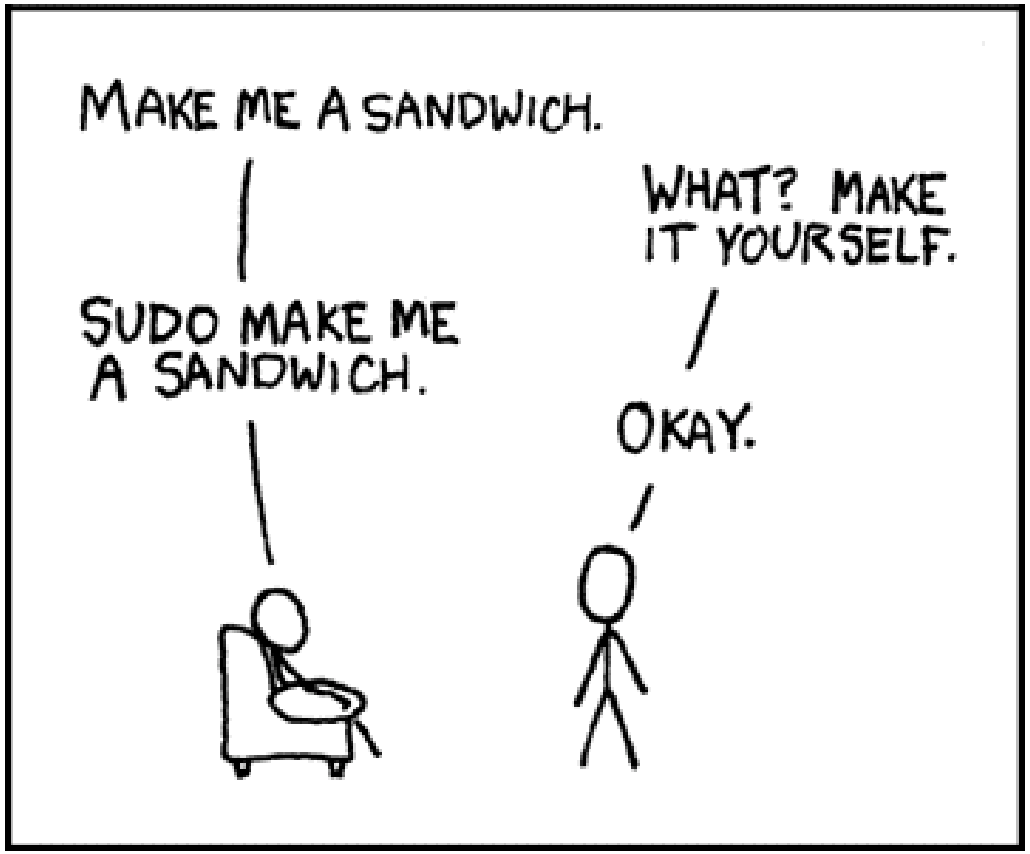
Pseudofiles

print(*obj*, …, *sep*=‘ ‘, *end*=‘\n‘, *file*=sys.stdout)

```
>>> montyPython = [John, Terry, Terry, Eric, Michael, Graham]
>>> for name in montyPython:
        print(name, sep=‘&‘, end=‘!‘, file=open(‘/home/mp_names.txt‘, ‘w‘))
```

```
John&Terry&Terry&Eric&Michael&Graham!
```

Systemfunktionen



Die Module sys, os und time

- Das Modul *sys* enthält die Schnittstelle zum Python-Interpreter und ermöglicht diesen zu beeinflussen
- Das Modul *os* ermöglicht es direkt Funktionen des Betriebssystems aufzurufen (z. B. Dateiverwaltung, Systemcalls)
- Das Modul *time* ermöglicht die Interaktion mit der Systemuhr (z. B. aktuelles Datum und Uhrzeit abrufen)

Systemfunktionen – Das Modul sys

Objekt	Erklärung
argv	Liste mit Kommandozeilen-Argumente.
platform	String mit Bezeichnung der aktuellen Platform. (z. B. ‘linux‘ oder ‘win32‘)
exc_info()	Liefert ein Tupel mit drei Werten, die Informationen über die Exception enthalten, die gerade behandelt wird.
exec_prefix	Eine Zeichenkette, die das Verzeichnis enthält, in dem die Platform-abhängigen Python-Dateien installiert sind. (z. B.: ‘/home/programs/python3/‘)
executable	Eine Zeichenkette, die den kompletten Pfad der ausführbaren Datei des Python-Interpreters enthält. (z. B.: ‘/home/programs/python3/bin/python3.5‘)
exit()	Beendet die Ausführung des Python-Scripts.
modules	Dictionary mit allen Standard-Modulen von Python.
path	Liste von Strings, die den Suchpfad von Modulen beschreiben.
stdin stdout stderr	File-Objekte für Standard-Eingabe, -Ausgabe, -Fehlerausgabe. Sie können durch andere Objekte ausgetauscht werden um so die Datenströme umzuleiten.
version	String mit Versionsbezeichnung des Pythoninterpreters.

Systemfunktionen – Das Modul sys

```
>>> import sys
>>> sys.path
['/home/programs/python3/libs/', '/home/programs/python2/site-packages/']
>>> sys.path.append('/home/scripts/')
>>> sys.path
['/home/programs/python3/libs/', '/home/programs/python2/site-packages/',
'/home/scripts/']

>>> sys.version
3.5.1 (default, Mar  10 2015, 12:06:10) [MSC v.1600 64 bit (AMD64)]
>>> if(sys.version[0] != '3'):
        sys.exit('Das Script läuft unter Python2 nicht und wird nun beendet!')

>>> try:
        e = open('').read()
    except:
        print(sys.exc_info())
(<class 'TypeError'>, TypeError("Required argument 'file' (pos 1) not found",),
<traceback object at 0x00000000038EE788>)
```

Kommandozeilen-Argumente

```
user@pc:~$ mein_python_script.py arg1 arg2 arg3 ...
```

```
#!/home/programs/python3/bin/python3.5

import sys

print(sys.argv)

with open(sys.argv[1], 'w') as e:
    e.write(sys.argv[2] + sys.argv[3])

f = open(sys.argv[1], 'r').read()
print(f)
```

```
user@pc:~$ mein_python_script.py /home/test.txt Monty Python
['/home/mein_python_script.py', '/home/test.txt', 'Monty', 'Python']
MontyPython
```

Systemfunktionen – Das Modul os

- Das Modul bietet u. A. einen einheitlichen Satz an Funktionen und Variablen um auf das Dateiverzeichnis zu zugreifen, unabhängig vom Betriebssystem

- Es erlaubt die Umgebungsvariablen des Betriebssystems zu ändern

- Es erlaubt Systemfunktionen aus der Python-Laufzeitumgebung heraus aufzurufen und auszuführen

Das Modul os – Dateien und Verzeichnisse

- Dateien und Verzeichnisse suchen

Funktion	Erklärung
chdir(*path*)	Wechselt das Arbeitsverzeichnis, *path* ist ein String mit einem absoluten oder relativen Pfad.
getcwd()	Gibt das aktuelle Arbeitsverzeichnis als String zurück.
listdir(*path*)	Liefert eine Liste von Strings mit Datei- und Verzeichnisnamen im Verzeichnis *path*.
path.isdir(*path)*	Liefert True, wenn *path* ein Verzeichnis ist, sonst False.
path.isfile(*path*)	Liefert True, wenn *path* eine Datei ist, sonst False.

- Zugriffsrechte abfragen und ändern

Funktion	Erkärung
access(*path*, *mode*)	Prüft die Zugriffsrechte oder Existenz einer Datei.
chmod(*path*, *mode*)	Ändert die Zugriffsrechte.

Das Modul os – Dateien und Verzeichnisse

```
>>> import os
>>> os.getcwd()                              #Fragt das aktuelle Arbeitsverzeichnis ab
'/home/programs/python/'
>>> os.chdir('/home/scripts/')               #Wechselt das Arbeitsverzeichnis

>>> os.listdir(os.getcwd())
['eigene_module', 'mein_python_script.py', 'Readme.txt']

>>> os.path.isdir('./eigene_module')                  #Relativer Pfad
True
>>> os.path.isfile('/home/scripts/Readme.txt')    #Absoluter Pfad
True
```

Das Modul os – Dateien und Verzeichnisse

- Für jede Datei und jedes Verzeichnis sind Zugriffsrechte definiert
- Zugriffsrechte können durch eine dreistellige Oktalzahl codiert werden, jede Zahl repräsentiert ein Zugriffsrecht für eine bestimmte Personengruppe

Zahl	Zugriffsrecht	Personengruppe
400	Lesen	Besitzer (*owner*)
200	Schreiben	Besitzer (*owner*)
100	Ausführen	Besitzer (*owner*)
40	Lesen	Gruppe (*group*)
20	Schreiben	Gruppe (*group*)
10	Ausführen	Gruppe (*group*)
4	Lesen	andere (*others*)
2	Schreiben	andere (*others*)
1	Ausführen	andere (*others*)

Das Modul os – Dateien und Verzeichnisse

```
>>> import os
>>> os.access('/home/scripts/Readme.txt', os.R_OK and os.W_OK and os.X_OK)
True

>>> os.chmod('/home/scripts/Readme.txt', 0o754)
#Owner: Lesen, Schreiben, Ausführen
#Group: Lesen, Ausführen
#Other: Lesen
```

Modus	Erklärung
R_OK	Testen, ob Leserecht gegeben ist.
W_OK	Testen, ob Schreibrecht gegeben ist.
X_OK	Testen, ob Ausführungsrecht gegeben ist.
F_OK	Testen, ob Pfad existiert.

Das Modul os – Dateien und Verzeichnisse

- Dateien und Verzeichnisse anlegen und ändern

Funktion	Erklärung
mkdir(*path*)	Neues Verzeichnis *path* erstellen.
makedirs(*path*)	Neues Verzeichnis *path* mit allen Zwischenverzeichnissen erstellen.
remove(*path*)	Die Datei mit dem Pfad *path* wird gelöscht.
rmdir(*path*)	Leeres Verzeichnis *path löschen.*
removedirs(*path*)	Leere Verzeichnisse löschen.
rename(*old, new*)	*old* enthält als String den (existierenden) Pfad einer Datei oder eines Verzeichnisses und wird in *new* umbenannt.
renames(*old, new)*	Der gesamte Pfad *old* mit allen Zwischenverzeichnissen wird in *new* umbenannt.

- Existenz Dateien und Verzeichnisse

path.exists(*path*)	Testet ob Pfad existiert.

Das Modul os – Dateien und Verzeichnisse

```
>>> import os
>>> os.mkdir('/home/new/')   #Neuer Ordner new wird im Verzeichnis home angelegt

>>> os.makedirs('/home/new/newer/newest/')          #Mehrere neue Ordner anlegen
>>> os.mkdir('/home/new/newer/newest/')       #Fehler da Ordner schon exisitiert
FileExistsError

#Datei umbenennen
>>> os.rename('/home/scripts/Readme.txt', '/home/scripts/DontReadme.txt')
#Datei verschieben
>>> os.rename('/home/scripts/DontReadme.txt', '/home/new/DontReadme.txt')

>>> os.remove('/home/new/DontReadme.txt')                    #Datei wird gelöscht
>>> os.rmdir('/home/new/newer/newest/')              #Fehler da Ordner nicht leer
OSError

>>> os.removedirs('/home/new/newer/newest/') #Alle leeren Ordner im Pfad löschen
```

Das Modul os – Dateien und Verzeichnisse

- Mit der Funktion **os.walk**(*path*) kann ein Verzeichnispfad komplett durchlaufen werden

- Die Funktion erzeugt ein Generator-Objekt, das eine Folge von 3er-Tupeln repräsentiert, die folgendermaßen aufgebaut sind:

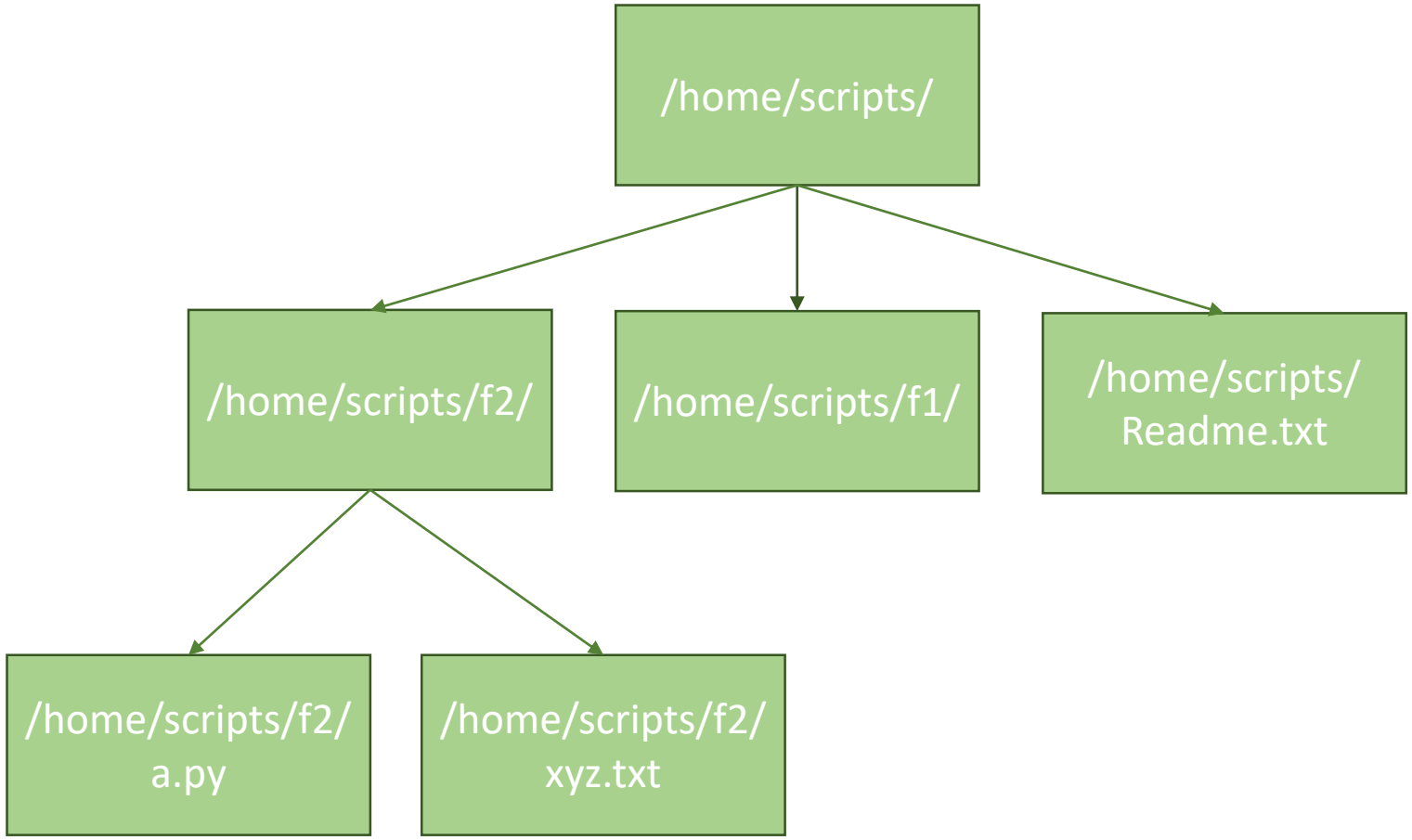
(Pfad, Unterverzeichnisse, Dateien)

String des Pfades des aktuell besuchten Verzeichnisses (z. B. ‘/home/scripts/‘)

Liste mit den Namen aller Unterverzeichnisse im aktuellen Verzeichnis (z. B. [‘Ordner1‘, ‘Ordner2‘])

Liste mit den Namen aller Dateien im aktuellen Verzeichnis (z. B. [‘File1‘, ‘File2‘])

Das Modul os – Dateien und Verzeichnisse



os.walk('/home/scripts/')

('/home/scripts/', ['f2','f1'], ['Readme.txt'])

('/home/scripts/f2/', [], ['a.py','xyz.txt'])

('/home/scripts/f1/', [], [])

Das Modul os – Systemaufrufe

- Mit der Funktion **os.system**(*command*) lassen sich Befehle auf der Shell bzw. Konsole direkt aus Python heraus ausführen
- *command* ist dabei ein String, welcher den auszuführenden Befehl repräsentiert

```
>>> import os
>>> os.chdir('/home/scripts/')
>>> os.system('ls')        #Shell-Befehl zum anzeigen des aktuellen Ordnerinhalts
eigene_module        mein_python_script.py        Readme.txt
0
>>> x = os.system('ls')    #Auffangen des Rückgabewerts des Shell-Befehls
>>> x
0                          #Konsolenausgaben sind (meist) nicht der Rückgabewert

>>> os.system('python anderes_python_script.py')
#Es lassen sich beliebige andere von der Shell ausführbaren Programme starten
```

Das Modul os – Systemaufrufe

- Die Funktion **os.popen**(*command*, *mode*) öffnet eine Verbindung zur Shell in Form eines File-Objektes

```
>>> import os
>>> os.chdir('/home/scripts/')
>>> prozess = os.popen('ls', 'r')          #Öffnen und ausführen eines Prozesses
>>> ergebnis = prozess.read()              #Abrufen der Ausgabe des Shell-Befehls
>>> prozess.close()                        #Prozess wieder schließen
>>> ergebnis
eigene_module        mein_python_script.py        Readme.txt
```

- Allerdings gibt es für Prozesshandling seit Python 2.6 ein neues überarbeitetes Modul *subprocess*

```
>>> subprocess.Popen('ls', stdout=subprocess.PIPE, shell=True).stdout.read()
```

Das Modul time

- Das Modul *time* enthält verschiedene zeitbezogene Funktionen, wobei Python drei verschiedene Formate verwendet:
  - Anzahl der Sekunden seit dem 1. Januar 1970 um 00:00 Uhr (Epoche/Unixzeit)
  - Ein Tupel mit den Einträgen für Jahr, Monat, Tag, Stunden, Minuten, Sekunden, …
  - Einen String aus 24 Zeichen (z. B. ‘Tue Apr 10 19:05:36 1990‘)

Funktion	Erklärung
asctime(*[tuple]*)	Wandelt einen Zeit-Tupel in einen 24 Zeichen Zeit-String, wenn das optionale Zeit-Tupel Argument fehlt, wird die aktuelle Lokalzeit verwendet.
ctime(*[secs]*)	Wandelt Unixzeit in einen 24-Zeichen-String um, wenn das optionale *secs* Argument fehlt wird die aktuelle Lokalzeit verwendet.
gmtime(*[secs]*)	Wandelt Unixzeit in ein Zeit-Tupel in UTC (Weltzeit) um, wenn das optionale *secs* Argument fehlt wird die aktuelle Lokalzeit verwendet.
localtime(*[secs]*)	Wie *gmtime()* liefert aber lokal Zeit.
mktime(*tupel*)	Gibt zu einem Zeit-Tupel die Sekunden seit Beginn der Epoche.
sleep(*n*)	Das Programm wird für *n* Sekunden pausiert.
time()	Liefert die Anzahl der Sekunden seit Beginn der Epoche.

Das Modul time

```
>>> import time
>>> time.asctime()
'Fri Apr 22 08:10:54 2016'

>>> time.gmtime()
time.struct_time(tm_year=2016, tm_mon=4, tm_mday=22, tm_hour=6, tm_min=15,
tm_sec=6, tm_wday=4, tm_yday=113, tm_isdst=0)

>>> time.localtime()
time.struct_time(tm_year=2016, tm_mon=4, tm_mday=22, tm_hour=8, tm_min=16,
tm_sec=30, tm_wday=4, tm_yday=113, tm_isdst=1)

>>> time.time()
1461306077.055702

>>> t = time.time()
    time.sleep(1)                          #Pausiert das Programm für eine Sekunde
    print(time.time() - t)
1.00099992752
```

"I SPEND A LOT OF TIME ON THIS TASK.
I SHOULD WRITE A PROGRAM AUTOMATING IT!"