FRIEDRICH-SCHILLER-
UNIVERSITÄT
JENA

Einführung in Python

2. Vorlesung

Kontrollstrukturen

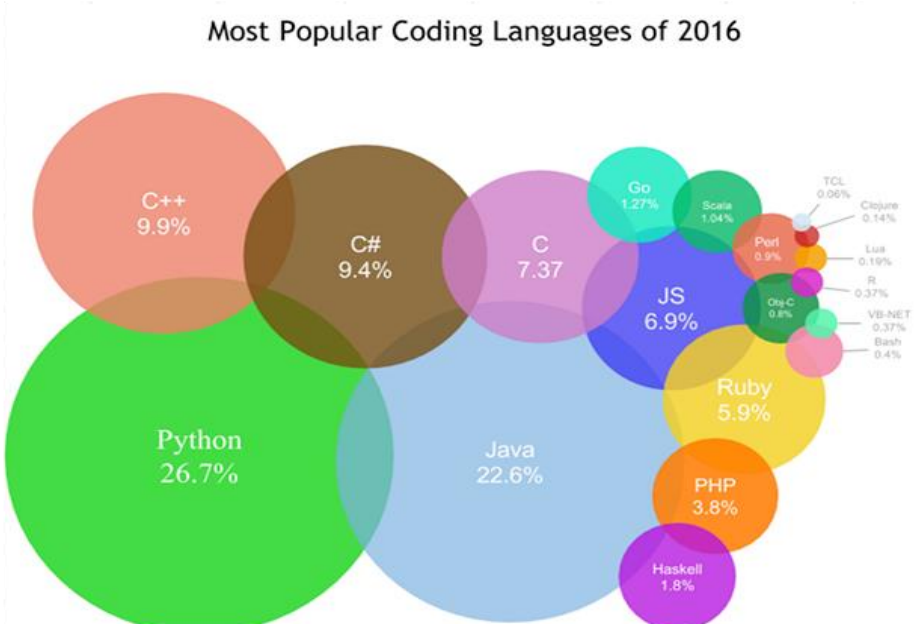
Funktionen

Wiederholung letztes Mal

- Was ist Python: Eine offene, minimalistische, dynamische, funktionale, interaktive, umfassend objektorientierte Skriptsprache

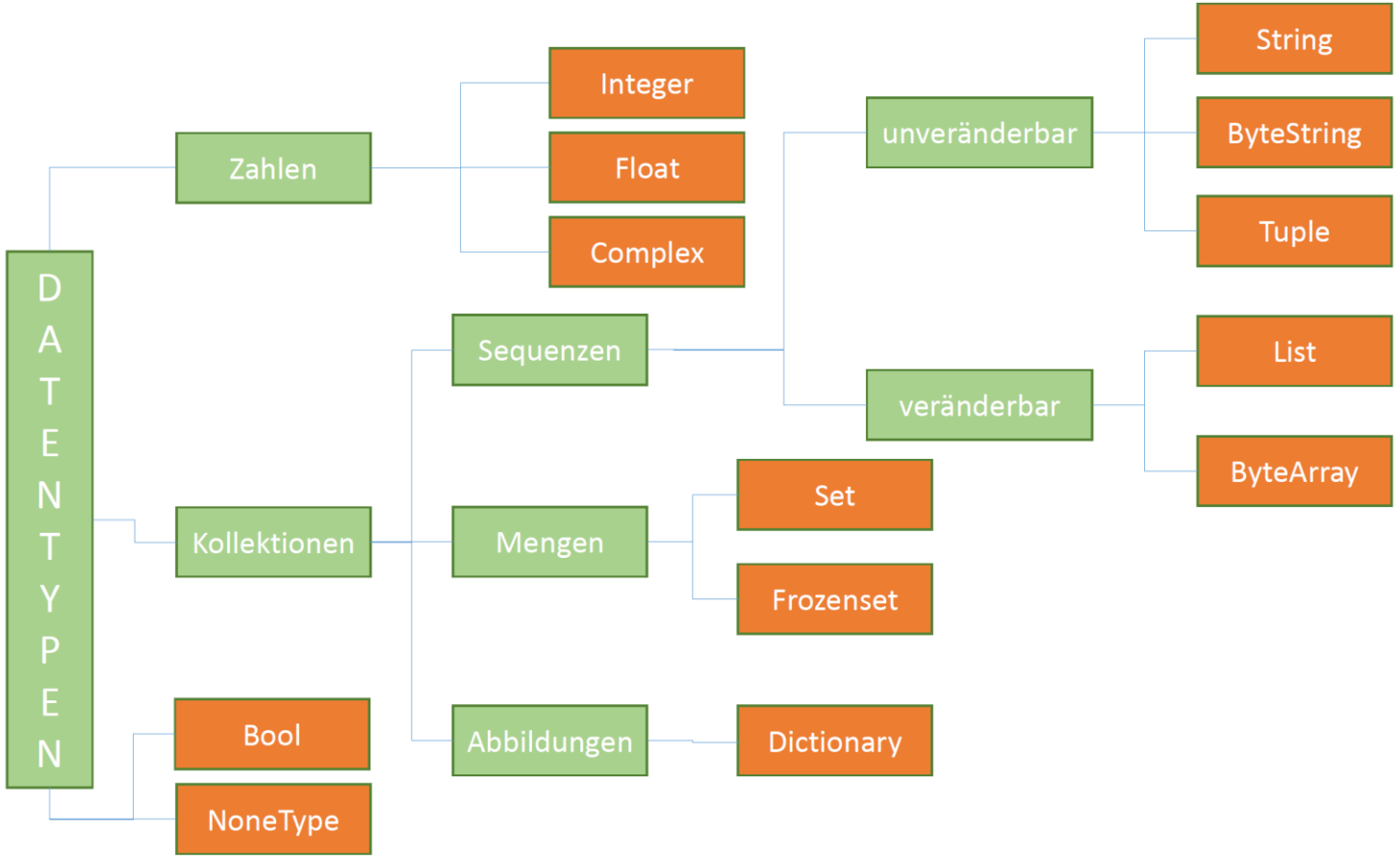
- Äußerst beliebt und vielseitig benutzbar

- Python 3 ist cooler als Python 2
- (und alles ist cooler als Perl)



Wiederholung letztes Mal

- Fundamentale Datentypen in Python



Wiederholung letztes Mal

- Typumwandlung (Variable Casting) in Python

```
>>> st = ‘Ritter der Kokosnuss‘
>>> st = “Flying Circus“
>>> st = str()

>>> i = 5
>>> i = int()

>>> f = 19.0
>>> f = float()

>>> se = {36,69,119}
>>> se = set()

>>> l = [‘Python‘, 3.5, 2.7]
>>> l = list()

>>> t = (x,y,z)
>>> t = tupel()
```

- Variablen sind grundsätzlich dynamisch getypt
- Ausdrücke sind **stark** getypt → “Explicit is better than implicit.”

```
>>> 5 + int(‘14‘)
19

>>> str(5) + ‘14’
‘514’
```

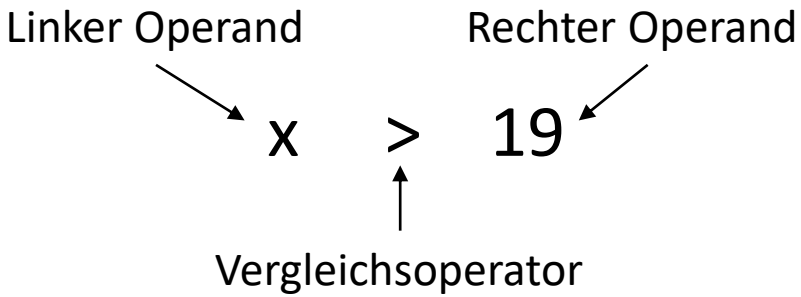
Kontrollstrukturen

- Programmverzweigungen
- Schleifen
- Sprünge
- Aufruf von Funktionen und Methoden
- Abfangen von Fehlern

Bedingungen

Bedingungen

- Vergleiche



Operator	Erklärung
<	Kleiner
<=	Kleiner gleich
>	Größer
>=	Größer gleich
==	Gleich
!=	Ungleich
is	Identisch
is not	Nicht identisch

Bedingungen

- Vergleiche

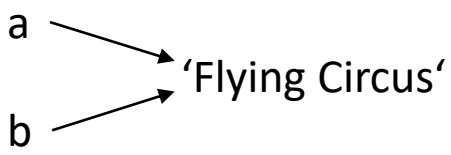
```
>>> 5 < 19                          #Objekte gleichen Typs
True

>>> 36 == 36.0                      #Gleiche Zahlen unterschiedlichen Typs
True

>>> 119 == 'Monty'                  #Objekte unterschiedlichen Typs
False

>>> 1 < 2 >= 5                      #Verkettung von Vergleichsoperatoren
False

>>> 'perl' < 'python'               #Vergleich von Strings
True

>>> 'a' > 'B'
True
```

Bedingungen

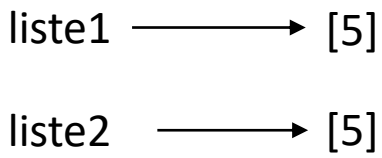
FRIEDRICH-SCHILLER-
UNIVERSITÄT
JENA

- Jedes Objekt in Python hat eine eigene Identität

```
>>>  149 is 149        #Zwei gleiche Literale besitzen immer die selbe Identität
True

>>>  id(149)          #Die Funktion id() gibt die Identität jedes Objekts wieder
1486774320

>>>  a = ‘Flying Circus‘
>>>  b = ‘Flying Circus‘
>>>  a is b
True

>>>  liste1 = [5]         #Alle Datentypen außer Zahlen, Strings, Booleans und None
>>>  liste2 = [5]         #erzeugen neue Objekte
>>>  liste1 is liste2
False
```



Ein (identisches) Objekt mit zwei Namen



Zwei gleiche, aber nicht identische Objekte

Bedingungen

- Zugehörigkeit zu einer Kollektion mit *in* oder *not in*

```
>>>  cooleSkriptsprachen = [‘Python‘ , ‘Ruby‘ , ‘Javascript‘ , ‘PHP‘]
>>>  ‘Python‘ in cooleSkriptsprachen
True
>>>  ‘Perl‘ not in cooleSkriptsprachen
True
```

- Alle Objekte und Ausdrücke haben immer einen Wahrheitswert

```
>>>  bool(119)                  #Numerische Werte ungleich null sind Wahr
True
>>>  bool(0.0)
False
>>>  bool(‘Nichts‘)             #Alle nicht-leeren Kollektionen sind Wahr
True
>>>  bool(‘‘)                   #und alle leeren sind Falsch
False
>>>  bool([])
False
```

Logische Operatoren

- Helfen um einfache Bedingungen zu komplexeren zusammen zubauen
  - *and :* Konjunktion
  - *or :* Disjunktion (inklusives oder)
  - *not :* Negation

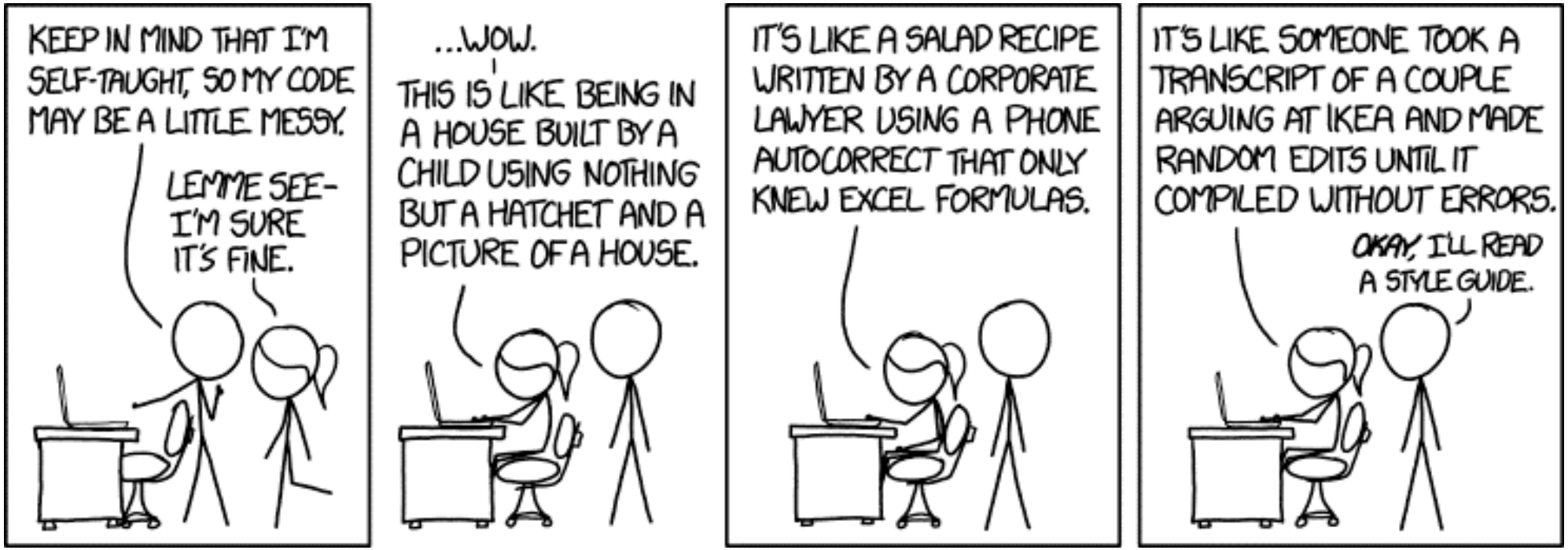
(Negation) <Operand 1> ***Konjunktion/Disjunktion*** <Operand 2>

```
>>> 1 or b == 'T'

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

- Logische Operatoren binden immer schwächer als Vergleichsoperatoren und sind *lazy*

FRIEDRICH-SCHILLER-
UNIVERSITÄT
JENA

Good Programming Practice: Lektion III



Good Programming Practice: Lektion III

- Klammern verbessern die Lesbarkeit (komplexer) logischer Ausdrücke

→ erleichtern das Finden logischer Fehler

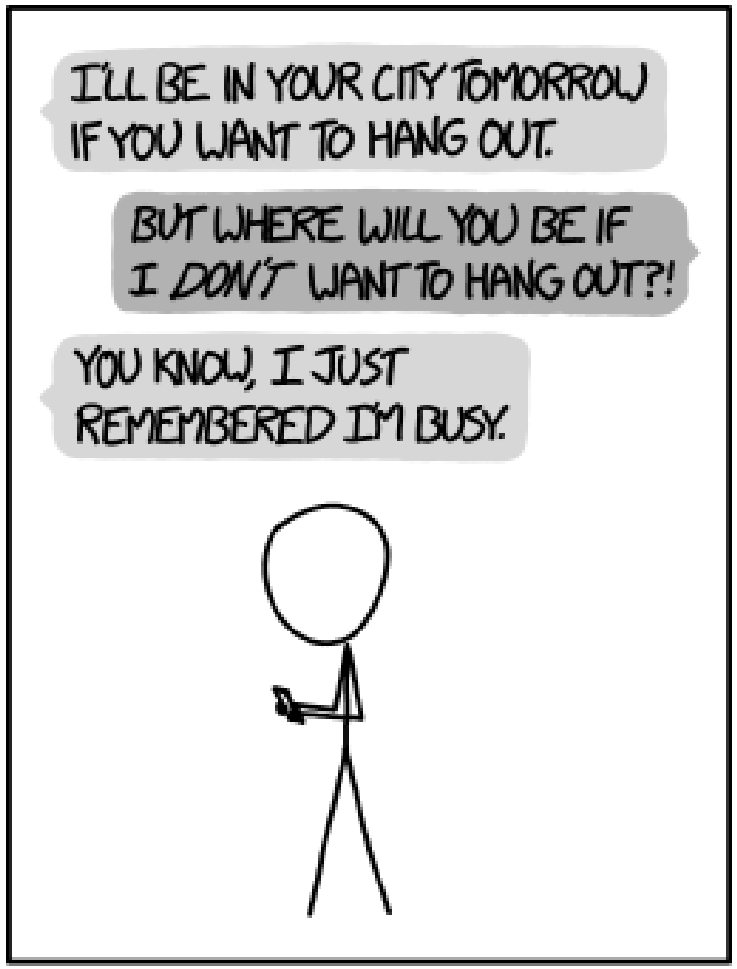
a > b → (a > b)

a > b or c < d → (a > b) or (c < d)

a < b < c < d → (a < b < c < d)

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

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

Bedingte Anweisungen



Bedingte Anweisungen

- Verzweigungen werden durch *if*-Anweisungen realisiert
- Einseitig (*if*), Zweiseitig (*if-else*), Mehrseitig (*if-elif*)

if **Bedingung:**

(Einrückung →) Anweisungblock

```
>>>  if a > b:
       print('a ist größer als b')
```

Bedingte Anweisungen

- Verzweigungen werden durch *if*-Anweisungen realisiert
- Einseitig (*if*), Zweiseitig (*if-else*), Mehrseitig (*if-elif*)

if **Bedingung:**

(Einrückung →) Anweisungblock

else **Bedingung:**

(Einrückung →) Anweisungblock

```
>>>  if a > b:
         print('a ist größer als b')
     else:
         print('a ist kleiner als b')
```

Bedingte Anweisungen

if **Bedingung:**

(Einrückung →) Anweisungblock

elif **Bedingung:**

(Einrückung →) Anweisungblock

else **Bedingung:**

(Einrückung →) Anweisungblock

```
>>>  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')
```

Bedingte Anweisungen

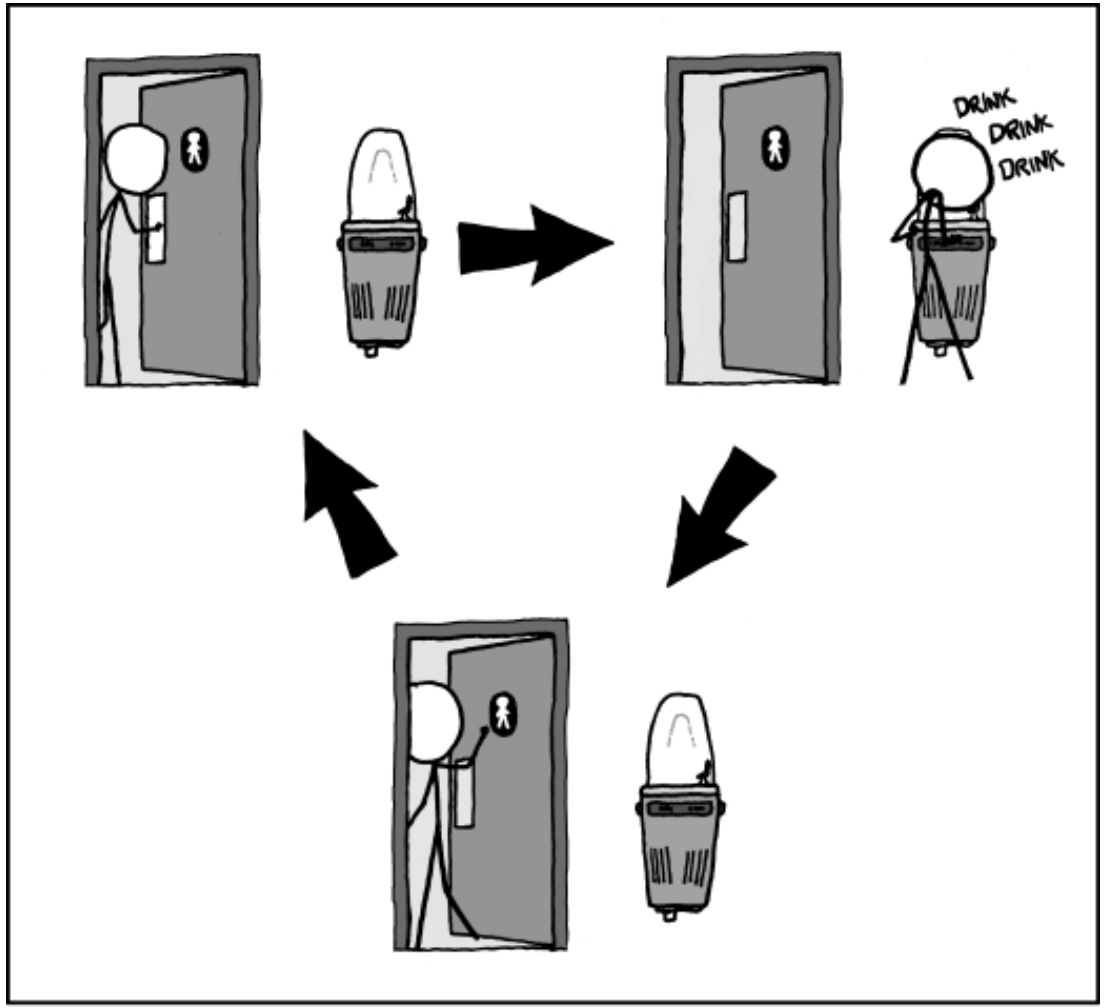
```
>>>  if a > 5:
         print('a ist größer als 5')
     elif a > 19:
         print('a ist größer als 19')
     elif a > 36:
         print('a ist größer als 36')
     elif a > 69:
         print('a ist größer als 69')
     elif a > 119:
         print('a ist größer als 119')
     elif a > 149:
         print('a ist größer als 149')
     else:
         print('a ist echt rießig')
```

Bedingte Ausdrücke

- Für sehr einfache Fallunterscheidungen kann man *if-else* Anweisungen auf einen Ausdruck reduzieren

```
>>> if a == True:
        text = ‘Fall A‘
    else:
        text = ‘Nicht Fall A‘


>>> text = ‘Fall A‘ if a == True else ‘Nicht Fall A‘

 #Bei bedingten Ausdrücken werden keine ‘:‘ benötigt
```

Schleifen


DRINK
DRINK
DRINK
I AVOID DRINKING FOUNTAINS OUTSIDE BATHROOMS
BECAUSE I'M AFRAID OF GETTING TRAPPED IN A LOOP.

Bedingte Wiederholung

- Anweisungen innerhalb einer *while*-Schleifen werden wiederholt so lange die Schleifenbedingung erfüllt ist

while **Bedingung:**

(Einrückung →) Anweisungsblock

```
>>> a = 2
>>> while a <= 256:     #Solange a kleiner als 256 verdopple a
        a = a*2


>>> while a > 1:        #Endlosschleife
        a *= 2

#Mit Strg+C löst man ein KeyboardInterupt aus und bricht jedes Programm ab
```

Iteration über eine Kollektion

- Anweisungen innerhalb einer *for*-Schleife werden eine definierte Anzahl an Schritten wiederholt (Länge der Kollektion)

for **Element** in **Kollektion:**

(Einrückung →) Anweisungsblock

```
>>> farben = ('grün', 'hellgrün', 'dunkelgrün')
>>> for farbe in farben:                #Gehe über jedes Element des Tupels
        print(farbe)
grün
hellgrün
dunkelgrün

>>> for i in [1, 2, 3, 4, 5]:           #Die Variable i nimmt nacheinander die
     print(i*i, end=' ')                #Werte der gegebenen Liste an
1 4 9 16 25
```

Zählschleife

- Anweisungen innerhalb einer *for*-Schleife werden eine definierte Anzahl an Schritten wiederholt
- Die Zählvariable ist (meist) ein Integer

for **Zählvariable** in *range*(**X**)**:**

(Einrückung →) Anweisungsblock

```
>>> for i in range(5):        #Die range-Funktion löst n Wiederholungen aus
      print(i, end=‘ ‘)
0 1 2 3 4
```

Zählschleife

- Die *range*-Funktion erzeugt ein spezielles Objekt das eine Folge ganzer Zahlen repräsentiert

range(start=0, stop=x, step=1)

```
>>> for i in range(5):                  #Nur der Stopp wird übergeben
     print(i, end=‘ ‘)
0 1 2 3 4

>>> for i in range(1, 5):               #Start und Stopp werden übergeben
     print(i, end=‘ ‘)
1 2 3 4

>>> for i in range(1, 5, 2):            #Start, Stopp und Schrittweise übergeben
     print(i, end=‘ ‘)
1 3
```

Zählschleife

```
#Die ersten 20 Zahlen der Fibonacci-Folge

>>> fib1 = 1
>>> fib2 = 1

>>> for i in range(18):
        fibx = fib1 + fib2
        print(fibx, end=‘ ‘)
        fib1 = fib2
        fib2 = fibx

2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765
```

Abbruch einer Schleife oder eines Durchlaufs

- Mit ***break*** wird die Ausführung einer Schleife komplett beendet

```
#Überprüfe ob n eine Primzahl ist

>>> n = int(input())

>>> for i in range(2, n):
        if n%i == 0:
            print(str(n) +‘ ist durch ‘ +str(i) +‘ teilbar‘)
            break

    if i == n-1:
        print(str(n) +‘ ist eine Primzahl‘)
```

Abbruch einer Schleife oder eines Durchlaufs

- Mit ***continue*** wird der aktuelle Schleifendurchlauf abgebrochen und ein neuer Durchlauf begonnen, sofern das Schleifenende nicht erreicht ist

```
#Zählen der Konsonanten in einem Wort

>>> wort = input('Wort: ')
>>> anzahl = 0
>>> for b in wort:
        if b in 'aeiouAEIOU':
            continue
        anzahl = anzahl + 1
>>> print('Das Wort enthält ' +str(anzahl) +' Konsonanten.')
```

Spezialfall: *for… else*

```
>>> for ele in [1, 3, 5, 7]:
        if ele % 2 == 0:
            print('Enthält eine gerade Zahl.')
            break
    else:
        print('Enthält nur ungerade Zahlen.')
```

- Der Anweisungsblock unter *else* wird nur dann ausgeführt, wenn die *for*-Schleife normal endet und nicht durch ein *break* unterbrochen wurde

List comprehension

- Listen können in Python auch auf abstrakte Weise generiert werden:

liste = [*ausdruck* **for** *element* **in** *andereListe*]

```
>>> quadZahlen = [i**2 for i in [0, 1, 2, 3, 4]]
[0, 1, 4, 9, 16]
```

- Listendefinitionen können durch eine Bedingung erweitert werden

```
>>> siebenTeiler = [i for i in range(50) if i%7 == 0]
[0, 7, 14, 21, 28, 35, 42, 49]

>>> A = [1, 2, 3]
>>> B = ['a', 'b', 'c']
>>> C = [(i,j) for i in A for j in B]
[(1,'a'), (1,'b'), (1,'c'), (2,'a'), (2,'b'), (2,'c'), (3,'a'), (3,'b'),
(3,'c')]
```

Exceptions

```
>>> zahl = int(input('Bitte gib eine ganze Zahl ein: '))
Bitte gib eine ganze Zahl ein: neunzehn

ValueError: invalid literal for int() with base 10: 'neunzehn'
```

Abfangen von Fehlern

- Viele Fehler treten in Python meist erst zur Laufzeit auf und führen dann zum Programmabbruch
  - Variablen falsch oder überhaupt nicht gecastet (umgewandelt)
  - Ein Ausdruck kann nicht ausgewertet werden
  - Zugriff auf ein nicht existierendes Element
  - Zugriff auf Dateien die nicht existieren oder die Rechte nicht gesetzt sind
  - …

- Programmabbruch verhindern und Fehler behandeln: *try…except*
- Programm nach einem Fehler gezielt beenden: *try…finally*

try… except

- Format:

try:

Anweisungblock 1

except **Ausnahmetyp**:

Anweisungblock 2

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

try… except

- Laufzeitfehler zwischen *try* und *except* werden abgefangen und müssen behandelt werden, danach geht das Programm weiter
- Der *except* Block wird nur aufgerufen wenn ein (spezifischer) Fehler auftritt
- Mit dem Schlüsselwort *pass* kann ein leerer Anweisungsblock erstellt werden*

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

*"Errors should never pass silently." – The Zen of Python

try… except

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

```
Bitte gib eine ganze Zahl ein: eins
Eingabe nicht in ganze Zahl umwandelbar.

Bitte gib eine ganze Zahl ein: 36j
Eingabe nicht in Ordnung.

Bitte gib eine ganze Zahl ein: 5
Danke für die Zahl.
```

try… finally

- Format:

try:

Anweisungblock 1

finally:

Anweisungblock 2

```
try:
    file = open('/Monty_Python/Filme/Der_Sinn_des_Kokosnuss/Kritik.txt','w')
    file.write(daten)
    file.close()
finally:
    file = open('/temp/backup.txt','w')
    file.write(daten)
    file.close()
    print('Pfad existiert nicht. Daten gespeicht in /temp/backup.txt')
```

try… finally

- Laufzeitfehler zwischen *try* und *finally* werden nicht abgefangen, das Programm endet nach der Ausführung des *finally* Blocks
- Der *finally* Block wird immer aufgerufen, selbst wenn kein Fehler auftritt

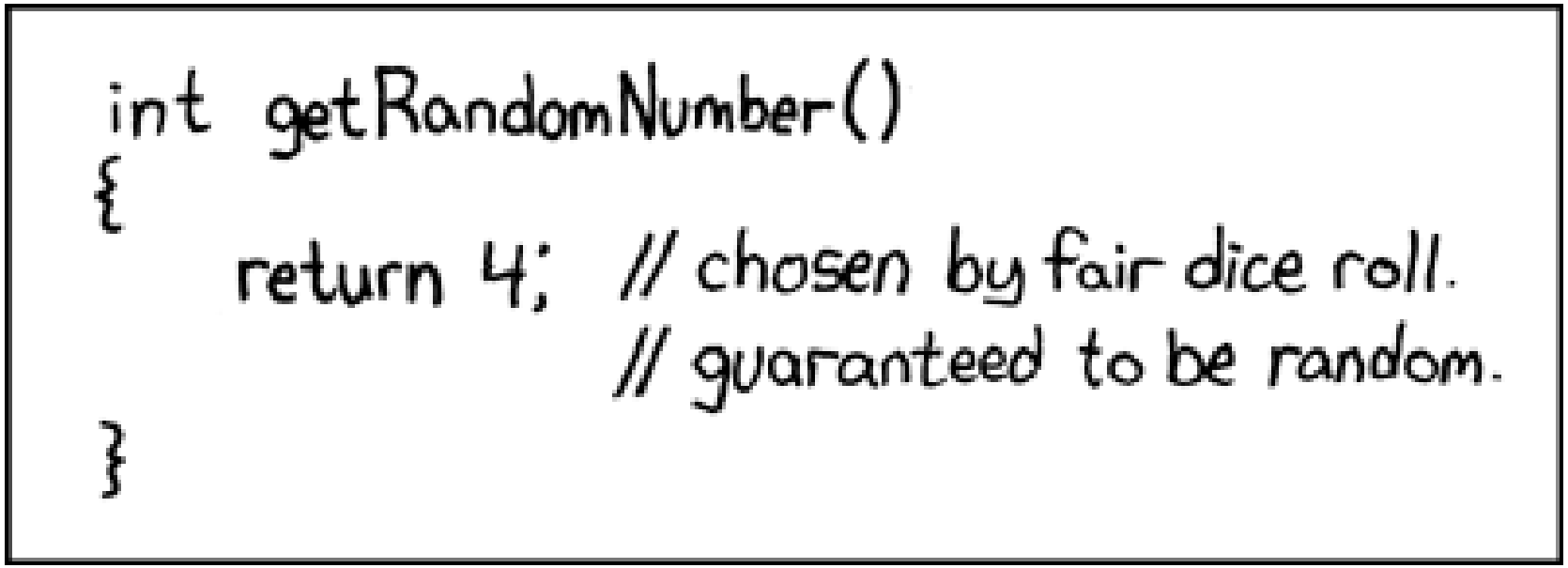
```
try:
    file = open('/Monty_Python/Filme/Der_Sinn_des_Kokosnuss/Kritik.txt','w')
    file.write(daten)
    file.close()
finally:
    file = open('/temp/backup.txt','w')
    file.write(daten)
    file.close()
    print('Pfad existiert nicht. Daten gespeicht in /temp/backup.txt')
```

try… except… finally

- Wenn der *except* Block genau den spezifischen Fehler abfängt, endet das Programm nicht nach dem *finally* Block

```
filePfad = '/Monty_Python/Filme/Der_Sinn_des_Kokosnuss/Kritik.txt'
try:
    file = open(filePfad,'w')
    file.write(daten)
    file.close()
except IOError:
    filePfad = '/temp/backup,text'
finally:
    file = open('/temp/backup.txt','w')
    file.write(daten)
    file.close()
    print('Pfad existiert nicht. Daten gespeicht in /temp/backup.txt')
```

Funktionen


int getRandomNumber()
{
return 4; // chosen by fair dice roll.
// guaranteed to be random.
}

Funktionen

- Funktionen sind Objekte (in Python ist alles ein Objekt), die bestimmte Teilaufgaben eines Programms lösen sollen

- Wichtig um effizient und übersichtlich zu programmieren (schrittweise Verfeinerung des Codes)

- Python besitzt viele Standardfunktionen (built-in functions) und noch mehr Module mit spezielleren Funktionen

funktionsname (*parameterliste*)

Funktionsargumente

- In der Regel akzeptieren Funktionen immer nur eine bestimmte Anzahl an Argumenten die einem bestimmten Typ angehören müssen
- Argumente können konkrete Werte (Literale), Variablen oder Ausdrücke sein

```
>>> len('eins')                    #Ein Literal als Argument
4

>>> a = 'zwei'
>>> len(a, 'drei')                 #Die Funktion len() akzeptiert nur ein Argument
TypeError: len() takes exactly one argument (2 given)

>>> len(a + 'drei')                #Ausdrücke deren Auswertung ein Wert ausgeben
8                                  #können auch als Argument benutzt werden
```

Funktionsargumente

```
#Manche Funktionen benötigen auch keine Argumente
>>> globals()
{'__spec__': None, '__package__': None, '__loader__': <class
'_frozen_importlib.BuiltinImporter'>, '__doc__': None, '__name__':
'__main__', '__builtins__': <module 'builtins' (built-in)>}

#Manche Funktionen akzeptieren eine variable Anzahl an Argumenten
>>> min(19, 5, 119, 69, 149, 36)
5
>>> min([69, 149, 36])
36

#Manche Funktionen haben optionale Argumente
>>> int('100')
100
>>> int('100',2)
4
>>> int('100',10)
100
```

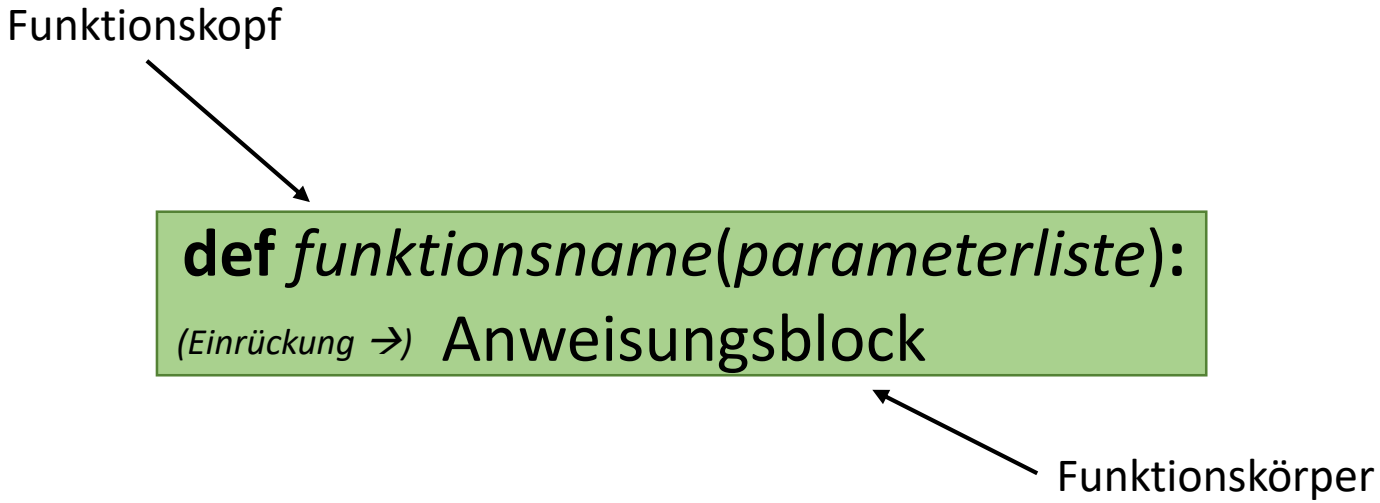
Funktionsargumente

- Auch Funktionen können als Argumente an andere Funktionen übergeben werden

map(*function, kollektion***)**

```
‘‘‘Die map()-Funktion sorgt dafür das eine Funktion auf alle Elemente
einer Kollektion angewandt wird‘‘‘
>>> result = map(len, [‘Chapman‘,‘Palin‘,‘Cleese‘,‘Jones‘])
>>> result
<map object at 0x00000000038BEE80>

>>> list(result)
[7, 5, 6, 5]
```

Definition von Funktionen



- Benötigt die Funktion keine Argumente, lässt man die Parameterliste einfach leer
- Enthält der Funktionskörper eine *return*-Anweisung, so gibt die Funktion einen bestimmten Wert zurück

Definition von Funktionen

```
def primzahl(zahl):
    if zahl <= 2:
        prim = False
    else:
        for i in range(2, zahl//2):
            if zahl % i == 0:           #Teiler gefunden
                prim = False
                break
        else:
            prim = True                 #Kein Teiler gefunden
    return(prim)                #return-Anweisung gibt den Wert von prim zurück
```

```
>>> primzahl(131071)
True

>>> primzahl(4)
False
```

Ausführung von Funktionen - Namensräume

```
#Wichtige Funktion f
def f():
    x = 2
    print('x: ' +str(x))

#Hautprogramm
x = 1
f()
print('x: ' +str(x))
```

```
x: 2
x: 1
```

- Erklärung: Es gibt im Hauptprogramm und im Funktionskörper zwei verschiedene unabhängige Variablen mit dem gleichen Namen

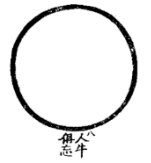
Ausführung von Funktionen - Namensräume

- Python erzeugt beim Aufruf jeder Funktion eine neue Umgebung aus *globalen* und *lokalen* Namensraum

→ lokale Variablen sind von der Außenwelt abgeschirmt und können nur innerhalb ihrer Umgebung genutzt werden

- Mit *globals()* und *locals()* kann man die jeweiligen Namensräume als dictionary abrufen

“Namespaces are one honking great idea -- let's do more of those!” – The Zen of Python



Ausführung von Funktionen - Namensräume

```
def f():
    x = 2
    print(globals())   #<---
    print(locals())    #<---

#Hautprogramm
x = 1
f()
print(globals())
print(locals())
```

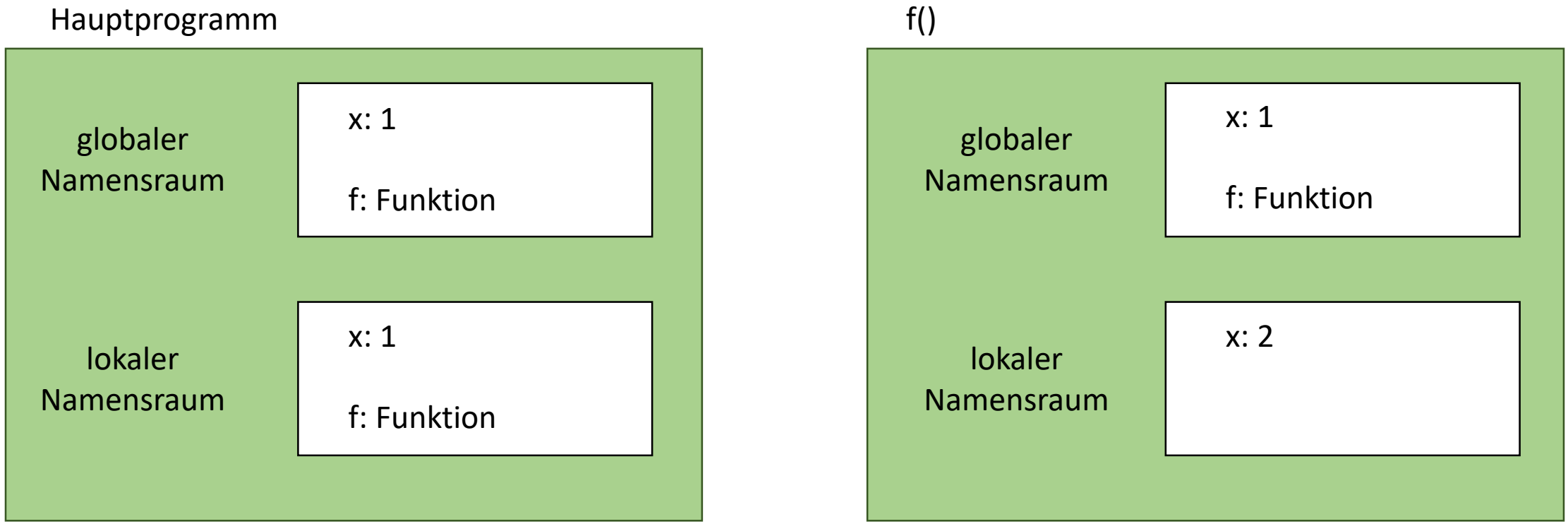
```
{'__spec__': None, '__package__': None, '__loader__': <class
'_frozen_importlib.BuiltinImporter'>, '__doc__': None, '__name__':
'__main__', '__builtins__': <module 'builtins' (built-in)>, ‘x‘: 1, ‘f‘:
<function f at 0x02C1A6F0}


{‘x‘: 2}
```

Ausführung von Funktionen - Namensräume

```
def f():
    x = 2
    print(globals())
    print(locals())

#Hautprogramm
x = 1
f()
print(globals())   #<---
print(locals())    #<---
```

```
{'__spec__': None, '__package__': None, '__loader__': <class
'_frozen_importlib.BuiltinImporter'>, '__doc__': None, '__name__':
'__main__', '__builtins__': <module 'builtins' (built-in)>, ‘x‘: 1, ‘f‘:
<function f at 0x02C1A6F0}
{'__spec__': None, '__package__': None, '__loader__': <class
'_frozen_importlib.BuiltinImporter'>, '__doc__': None, '__name__':
'__main__', '__builtins__': <module 'builtins' (built-in)>, ‘x‘: 1, ‘f‘:
<function f at 0x02C1A6F0}
```

Ausführung von Funktionen - Namensräume



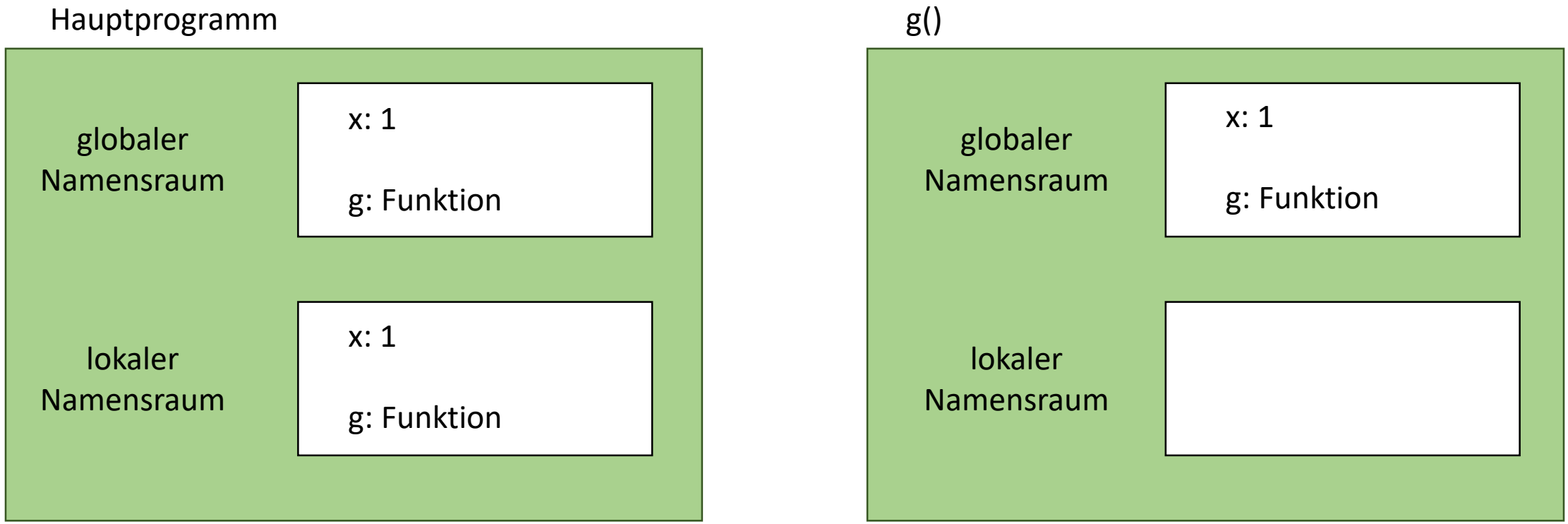
- Variablennamen werden immer zuerst im lokalen, dann im globalen Namensraum gesucht

Ausführung von Funktionen - Namensräume

```
#Noch wichtigere Funktion g
def g():
    print('x: ' +str(x))          #x ist nicht im Funktionskörper definiert

#Hautprogramm
x = 1
g()
```

```
x: 1
```

Ausführung von Funktionen - Namensräume



- Vorsicht: In Funktionen können globale Variablen zwar gelesen, aber meist nicht ohne Weiteres verändert werden!

Die *global*-Anweisung

```
>>> def verdopple():
        x = x * 2

>>> x = 2
>>> verdopple()

UnboundLocalError: local variable 'x' referenced before assignment
```

- Die Veränderung von x kann nicht durchgeführt werden, da x im lokalen Namensraum nicht bekannt ist → Verhinderung von **Seiteneffekten**

```
>>> def verdopple():
        global x                #mit global werden globale Variablen in den
        x = x * 2               #lokalen Namensraum geschrieben

>>> x = 2
>>> print(verdopple())
4
```

Parameterübergabe

- Prinzipiell zwei Arten von Parameterübergabe:
  - ***call-by-value***: In der Funktion wird mit Kopien der Übergeben Parameter gearbeitet → sicher aber schnell speicherlastig
  - ***call-by-reference***: In der Funktion wird mit Referenzen auf die im Hauptprogramm befindlichen Objekte gearbeitet → effizient aber unsicher

Parameterübergabe

- Prinzipiell zwei Arten von Parameterübergabe:

  - ***call-by-value***: In der Funktion wird mit Kopien der Übergeben Parameter gearbeitet → sicher aber schnell speicherlastig
    (alle unveränderlichen Datentypen: Zahlen, Strings, Tuple)

  - ***call-by-reference*:** In der Funktion wird mit Referenzen auf die im Hauptprogramm befindlichen Objekte gearbeitet → effizient aber unsicher
    (alle veränderlichen Datentypen: Listen, Mengen, Dictionaries)

Parameterübergabe

```
>>> def halbiere(zahl):
        zahl = zahl/2
        return(zahl)

>>> n = 5
>>> halbiere(n)
2.5
>>> n                          #n bleibt unverändert
5
```

```
>>> def quadriere(li):
        for i in range(len(li)):
            li[i] = li[i] * li[i]
            return(liste)

>>> liste = [5, 19, 36, 69, 119, 149]
>>> quadriere(liste) #liste wird verändert
[25, 361, 1296, 4761, 14161, 22201]
>>> liste
[25, 361, 1296, 4761, 14161, 22201]
```

Good programming practice: Lektion IV

- Vermeidet Seiteneffekte!

Voreingestellte Parameterwerte

- Um optionale Funktionsparameter zu ermöglichen, kann man Standard-Werte im Funktionskopf festlegen

```
>>> def funktionsname(arg1=default1, arg2=default2, ...):
```

```
>>> def quad_tabelle(anzahl=3, schritt=1):
        x = 1.0
        for i in range(anzahl):
            print(str(x) +'  ' +str(x*x))
            x += schritt

>>>quad()      >>>quad(4, 2)      >>>quad(2, 4)      >>>quad(schritt=2, anzahl=4)
1  1          1  1               1  1               1  1
2  4          3  9               5  25              3  9
3  9          5  25                                 5  25
              7  49                                 7  49
```

Voreingestellte Parameterwerte

- Vorsicht: Die Default-Werte werden nur **ein einziges Mal** (bei der Funktionsdefinition) eingesetzt und nicht bei jedem neuen Aufruf

```
>>> def anhaengen(a, liste=[]):
        liste += [a]
        return(liste)

>>> anhaengen(1)              #der Default-Parameter liste besteht auch nach
[1]                           #Funktionsende weiter
>>> anhaengen(2)
[1, 2]
>>> anhaengen(3)
[1, 2, 3]
```

- Dies gilt wieder nur für veränderbare Datentypen!

Voreingestellte Parameterwerte

- Vorsicht: Die Default-Werte werden nur **ein einziges Mal** (bei der Funktionsdefinition) eingesetzt und nicht bei jedem neuen Aufruf

```
>>> def anhaengen(a, liste=None):
        if(liste == None):
            liste = []
        liste += [a]
        return(liste)

>>> anhaengen(1)
[1]
>>> anhaengen(2)
[2]
>>> anhaengen(3)
[3]
```

Funktionen mit beliebiger Anzahl von Parametern

- Mit einem Stern vor einem Parameter im Funktionskopf erzeugt man ein Tuple von Argumenten

```
>>> def funktionsname(*args):
```

```
>>> def superquersumme(*zahlen):          #Summe von Quersummen
        supersumme = 0
        for ele in zahlen:
            summe = 0
            zahlenstring = str(ele)
            for b in zahlenstring:         #Berechnung der einzelnen Quersummen
                summe += int(b)
            supersumme += summe
        return(supersumme)

>>> superquersumme(1, 100, 123)
8
```


IMPORT OS
CUR_PATH = OS.GETCWD()
IGNORE_SET = SET(["__INIT__.PY",
"COUNT_SOURCELINES.PY"])
I DIDN'T KNOW HARRY SPOKE PYTHON.
YEAH, HE'S A PARSER-TONGUE.