

ProInformatik 3: Objektorientierte Programmierung Python (Teil 2)



SoSe 2018

Oliver Wiese

Grundlegende Elemente von imperativen Programmen

- Variablen
- Zuweisungen
- Ausdrücke
- Definitionen von Datentypen
- Deklarationen von Variablen unter Verwendung vordefinierter Datentypen
- **Anweisungen für den Kontrollfluss innerhalb des Programms**
- Gültigkeitsbereich von Variablen (*locality of reference*)
- Definition von Prozeduren und Funktionen

Vergleichsoperatoren

alle binär

$<$	Kleiner
$>$	Größer
$<=$	Kleiner oder gleich
$>=$	Größer oder gleich
$==$	Gleichheit
$!=$	Ungleichheit

Fallunterscheidung

if-else-Anweisung

```
a = int(input( "Zahl = " ))  
  
if a<0:  
    print ( "a ist negativ" )  
else:  
    print ( "a ist positiv" )
```

```
a = int(input( "Zahl = " ))  
  
if a<0:  
    print ( "a ist negativ" )  
elif a==0:  
    print ( "a ist gleich 0" )  
else:  
    print ( "a ist positiv" )
```

Einrücken anstatt Klammern



if-else-Anweisung

```
bit = True
```

```
if bit:
```

```
    print( "Hi!" ) -----> Hi!
```

```
else:
```

```
    print( "No!" )
```

```
bit = 0
```

```
if bit:
```

```
    print(True)
```

```
else:
```

```
    print(False) -----> False
```

```
bit = "Hi"
```

```
if bit:
```

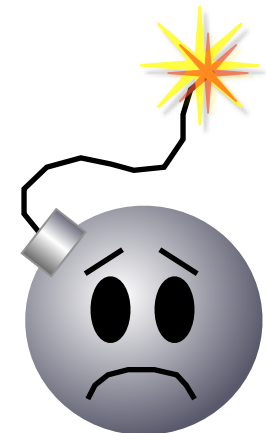
```
    print(True) -----> True
```

```
else:
```

```
    print(False)
```



Zahlen und andere Datentypen
werden wie in **C** als Wahrheitswerte
interpretiert



for-Schleifen

```
for x in ['spam', 'bla', 'eggs']:  
    print (x)
```

```
>>>  
spam  
bla  
eggs
```

```
for x in [1, 2, 3, 4]:  
    print (x)
```

```
>>>  
1  
2  
3  
4
```

```
sum=0
```

```
for x in [1, 2, 3, 4, 5]:  
    sum = sum + x
```

```
print( sum)
```

```
>>>  
15
```

```
for i in range(1, 5):  
    sum += i
```

```
print( sum)
```

```
>>>  
25
```

```
Dic = [(1,'a'), (2,'b'),(3,'c')]
```

```
for (x,y) in Dic:  
    print (x, y)
```

```
>>>  
1 a  
2 b  
3 c
```

for-Schleife

```
for Ausdruck in Sequenz :  
    Anweisungen
```

Bei der **for**-Schleife in Python wird nicht über eine Folge von Zahlen, sondern über Elemente einer Sequenz iteriert.

Erzeugt alle Kombinationen von zwei Zeichen aus *text*

```
text = input ( "text = " )
```

```
for i in text:  
    for j in text:  
        print ( i+j )
```

```
text = abc  
aa  
ab  
ac  
ba  
bb  
bc  
ca  
cb  
cc
```

Logische Operatoren

Operator		Beschreibung
not	unär	logische Negation
or	binär	logisches Oder
and	binär	logisches Und

Bit-Operatoren

Operator		Beschreibung
~	unär	bitweise Inversion (Negation)
<<	binär	nach Links schieben
>>	binär	nach Rechts schieben
&	binär	bitweise UND
 	binär	bitweise ODER
^	binär	bitweise Exklusives Oder

while-Anweisung

while *Ausdruck* :
Anweisungen

Berechnet alle Quadratzahlen bis n

```
n = int(input( "n = " ))
```

```
zaehler = 0
```

```
while zaehler<=n:
```

```
    print (zaehler*zaehler)
```

```
    zaehler = zaehler + 1
```

for- vs. while-Schleifen

```
summe = 0

for i in range(1, 100):
    summe += i # summe = summe + i
print( summe )
```



```
summe = 0

i = 1
while i < 100:
    summe += i # summe = summe + i
    i += 1    # i = i + 1
print( summe )
```

```
n = int(input('n= '))

y = n+1
while (not isPrime(y)):
    y = y + 1
print('next prime > ', n, 'is: ', y)
```



?

Python Programm ohne Funktionen

```
n = int(input( 'Integer number: ' ))
```

```
result = 0
while result**3 < abs(n):
    result = result+1
```

```
if result**3 != abs(n):
    print(n, 'is not a perfect cube ')
else:
    if n<0:
        result = -result
    print ('The cube root is', result)
```

```
n = int(input( 'Integer number: ' ))
```

```
for result in range(0, abs(n)+1):
    if result**3 == abs(n):
        break
```

```
if result**3 != abs(n):
    print(n, 'is not a perfect cube')
else:
    if n<0:
        result = -result
    print ('The cube root is', result)
```

Imperatives Programmieren

Grundlegende Operation: die **Zuweisung**

- Speicherinhalte werden verändert und damit der Zustand der gesamten Maschine.

Kontrollfluss-Anweisungen

- bedingte Sprünge:

if-then-else-Anweisung und **Loop**-Anweisungen (**for**- und **while**-Schleifen).

- unbedingte Sprünge:

GOTO-, **break**-, **return**-Anweisung usw.

Grundlegende Elemente von imperativen Programmen

- Definitionen von Datentypen
- Deklarationen von Variablen unter Verwendung vordefinierter Datentypen
- Zuweisungen
- Ausdrücke
- Anweisungen für den Kontrollfluss innerhalb des Programms

- Jetzt
- **Gültigkeitsbereich von Variablen (*locality of reference*)**
 - **Definition von Prozeduren, Subroutinen und Funktionen**
 - **Parameter-Übergabe**

Funktionen

Funktionen sind das **wichtigste Konzept** in der Welt der höheren Programmiersprachen.



Funktionen sind ein grundlegendes Hilfsmittel, um Probleme in kleinere Teilaufgaben zerlegen zu können.

Sie ermöglichen damit eine **bessere Strukturierung** eines Programms sowie die Wiederverwertbarkeit des Programmcodes.

Gut strukturierte Programme bestehen typischerweise aus **vielen kleinen**, nicht aus wenigen großen Funktionen.

Funktionen in Python

Eine Funktionsdefinition startet mit dem **def**-Schlüsselwort

Funktionsname

Argumente

$$\pi = 4 \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1}$$

```
def pi_leibniz ( k ):
    sum = 0
    for n in range( 0, k ):
        sum = sum + ((-1)**n)/(2.0*n+1)
    return sum*4
```

Das Einrücken entscheidet, was zur Funktion gehört bzw. wann die Funktionsdefinition zu Ende ist.

Die **return**-Anweisung gibt das Ergebnis der Funktion zurück

Python-Funktionen

etwas genauer:

```
def Funktionsname ( Arg1, Arg2, ... ) :  
    Anweisung1  
    Anweisung2  
    ...  
    Anweisungn
```

Eine Anweisung der Form:

return *Ergebniswert*

befindet sich an beliebiger Stelle und beliebig oft in dem Funktionsrumpf; sie beendet die Ausführung der Funktion mit der Rückgabe eines Ergebniswertes.

Formale und aktuelle Parameter

Die formalen Parameter einer Funktionsdefinition sind **Platzhalter**.

Beim Aufruf der Funktion werden die formalen Parameter durch reale Variablen ersetzt, die den gleichen Typ wie die formalen Parameter haben müssen.

Formale und aktuelle Parameter

Funktionsbeispiel:

```
def teiler (   :  
    return False )
```

Funktionsaufruf:

```
>>> print ( teiler ( 7 7 3 3 )
```

```
>>> False
```

```
>>>
```

Funktionen in Python

Funktionen

```
def quadrat( zahl ):  
    return zahl*zahl
```

```
def teiler( a, b ):  
    return a%b == 0
```

keine saubere Funktion

```
def test_funktionen():  
    a = int( input('a='))  
    b = int( input('b='))  
    print( teiler(quadrat(a), quadrat(b)) )
```

```
test_funktionen()
```

Anwendung
innerhalb eines
Ausdrucks

Funktionen

Funktionen verdienen ihre Namen, wenn:

- diese keine Seiteneffekte beinhalten
- die **Eingabe** nur durch die **Argumente** erfolgt
- die **Ausgabe** nur mit Hilfe von **return**-Anweisungen stattfindet
- und **zwischendurch keinerlei Ein-/Ausgabe** verwendet wird.

Funktionen

Funktionen sollen möglichst nur **lokale** Variablen benutzen.

Gut definierte Funktionen können innerhalb von Ausdrücken angewendet werden.

Sonst sollen sie **Subroutinen**, **Prozeduren** oder **Methoden** heißen.

Funktionen in C



Funktionen

Der Funktionsbegriff wird innerhalb vieler Programmiersprachen sehr **unpräzise** verwendet.

In **C**, **Python** und vielen Programmiersprachen spricht man von Funktionen, obwohl sie oft keine Funktionen im mathematischen Sinn sind.

In einigen Programmiersprachen unterscheidet man zwischen Funktionen und Prozeduren (*Subroutines*) wie z.B. VB (VBA)

In Python muss weder der Datentyp des Rückgabewertes noch der Datentyp der Argumente deklariert werden.

Geltungsbereich und Lebenszeit von Variablen

- Der **Geltungsbereich** einer Variablen ist der Bereich innerhalb des Programms, in dem diese sichtbar ist.
- **Lebenszeit** ist die Zeit, die eine Variable im Speicher existiert.

Ein **Modul** ist eine Datei, die Python-Definitionen und -Anweisungen beinhaltet.

Geltungsbereich von Variablen in Python

```
def foo(x, y):  
    print(x, y)  
  
def foo2(a, b):  
    print(a, b, x, y)
```

```
x = 100  
y = 200
```

```
foo(1, 2)  
foo2(1, 2)
```

Ausgabe:

>>> 1 2

>>> 1 2 100 200

Geltungsbereich von Variablen in Python

global-Spezifizierer

```
def foo3():  
    global x  
    global y  
    x = 0  
    y = 0
```

```
x = 100  
y = 200  
print (x, y)  
foo3()  
print (x, y)
```

Ausgabe:

>>>

100 200

0 0

>>>

global-Spezifizierer

Beispiel:

```
def func(x, y):
    global g
    g = 3
    v = 6
    x, y = y, x
    print(g, v, x, y)
```

```
g, v, x, y = 10, 20, 30, 40
```

```
func(0, 1)
print(g, v, x, y)
```

```
func(x, y)
print(g, v, x, y)
```

Mit dem `global`-Spezifizierer werden
Bezeichner dem globalen Namensraum
zugeordnet.

guter Programmierstil?
sinnvolle Anwendung?

Ausgabe: >>>

→ 3 6 1 0

→ 3 20 30 40

→ 3 6 40 30

→ 3 20 30 40

Funktionsargumente können sehr flexibel angegeben werden.

""" Argumente von Funktionen """

```
def fun( a=1, b=3, c=7 ):
    print ('a=', a, 'b=', b, 'c=', c)
```

```
fun( 30, 70 )
```

```
fun(20, c=100)
```

```
fun(c=50, a=100)
```

```
fun(20)
```

```
fun(c=30)
```

```
fun()
```

Ausgabe:

```
>>>
```

```
a= 30 b= 70 c= 7
```

```
a= 20 b= 3 c= 100
```

```
a= 100 b= 3 c= 50
```

```
a= 20 b= 3 c= 7
```

```
a= 1 b= 3 c= 30
```

```
a= 1 b= 3 c= 7
```

Die Reihenfolge der Argumente kann verändert werden.

Nur die Argumente, die benötigt werden, können angegeben werden.

Geltungsbereich und Variablen in Python

Beispiel:

```
x = 100
z = 7
def fun( a=1, b=3, c=7 ):
    x = 6
    print('a=', a, 'b=', b, 'c=', c)
    while ( x>3 ):
        y = 4
        print( y, x )
        print( z )
        x = x - 1
        print ( y, x )

print( x )
fun()
print( y )
```

Modul Variablen

Lokale Variablen
innerhalb der
Funktionsdefinition

Geltungsbereich von Variablen in Python

Beispiel:

```
x = 100
z = 7
def fun( a=1, b=3, c=7 ):
    x = 6
    print('a=', a, 'b=', b, 'c=', c)
    while ( x>3 ):
        y = 4
        print( y, x )
        print( z )
        x = x - 1
    print ( y, x )

print( x )
fun()
print( y ) # Laufzeitfehler
```

Ausgabe:

```
>>>
100
a=1 b=3 c=7
4 6
7
4 5
7
4 4
7
4 3
Traceback (most recent call last):
  File "example.py", line 16, in
<module> print(y)
NameError: name 'y' is not defined
```

Gültigkeitsbereich von Variablen in Python

Lokaler

Variablennamen sind innerhalb einer Methode oder Funktion definiert.

Modul-Variablen

Die Variablen sind innerhalb eines Moduls (Skriptdatei).

Eingebauter Geltungsbereich

Innerhalb der Python-Interpreter vordefinierte Namen, die immer gültig sind.

Verschachtelte Funktionen

Funktionen können innerhalb anderer Funktionen definiert werden.

```
def percent (a, b, c):
    def pc(x): return (x*100.0) / (a+b+c)
    return (pc(a), pc(b), pc(c))

print (percent (2, 4, 4))
print (percent (1, 1, 1))
```

```
>>>
(20.0, 40.0, 40.0)
(33.333333333333, 33.333333333333, 33.333333333333)
>>>
```


Parameter-Übergabe in Funktionen

call-by-value

- Ausdrücke werden zuerst ausgewertet und dann nur der Ergebniswert an die Funktionen übergeben.
- Einzelne Variablen werden kopiert und nur eine Kopie als Parameter weitergegeben.
- Der Inhalt der originalen Variablen des Aufrufers bleibt unverändert.

Parameter-Übergabe in Python

call-by-value

Beim Aufruf einer Funktion wird in Python nur eine Kopie der **Referenzen** der jeweiligen Parameter-Objekte übergeben.

Innerhalb der Funktionen werden die Objekte mittels ihrer **Referenzen** für die Berechnungen verwendet.

Zuweisungen auf **nicht** veränderbare Variablen verursachen das Erzeugen von neuen Objekten.

Zuweisungen auf veränderbare Variablen haben Auswirkung auf die originalen Variablen des aufrufenden Programmteils.

Typsystem von Python

Python ist eine objektorientierte Programmiersprache im weiten Sinn.

In Python wird alles durch Objekte repräsentiert. Jedes Objekt besitzt eine **Identität**.

Die Identität eines Objekts kann mit der Standardfunktion **id()** abgefragt werden.

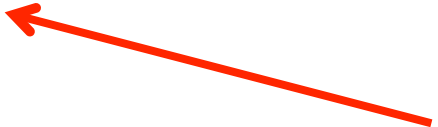
Wert- vs. Referenz-Semantik

- Wert-Semantik

Ein Ausdruck wird ausgewertet und das Ergebnis direkt in eine Variablen-Adresse gespeichert.

- Referenz-Semantik

Ein Ausdruck wird zu einem Objekt ausgewertet, dessen Speicheradresse in einer Variablen-Adresse gespeichert wird.

 Python

Dynamisches Typsystem von Python

nur die halbe Wahrheit!

↓

```

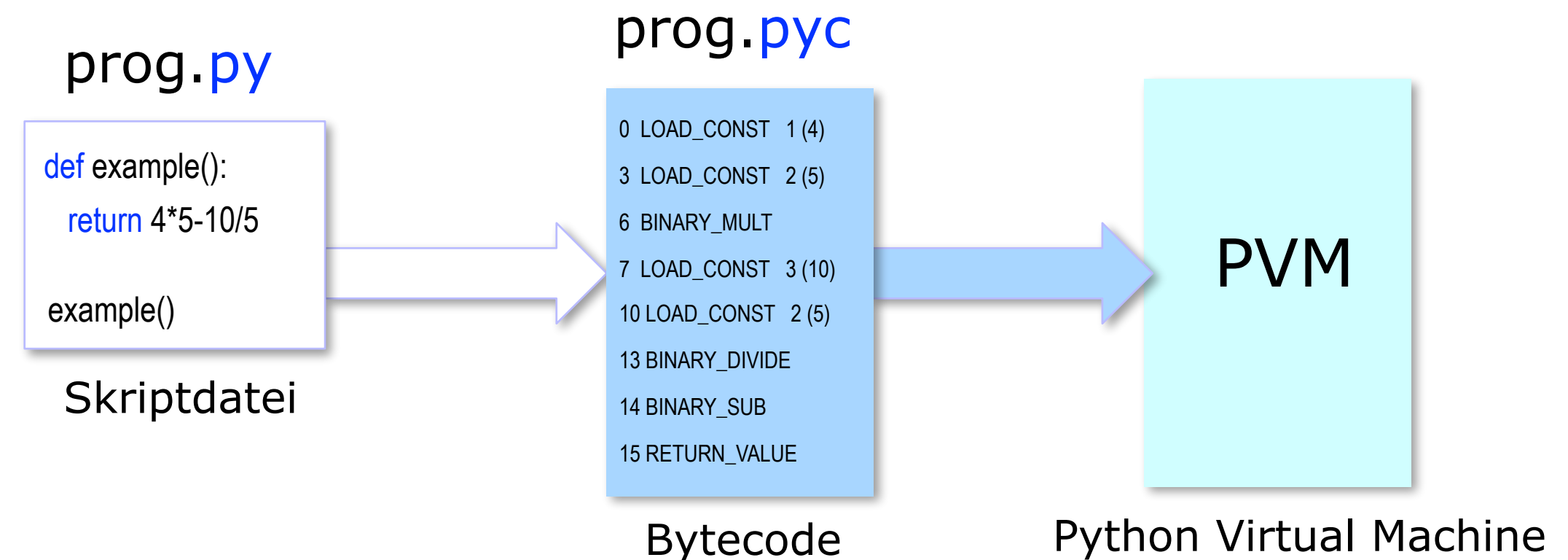
y = 1
x = 2
a = 10.5
sum = y+a
mult = a*x
    
```

Virtuelle Maschine

a	10.5
x	2
y	1
sum	11.5
mult	21

Speicher

Python Virtual Machine



Cpython

Standard aus www.python.org

PVM

Jython

Übersetzung auf Java-Bytecode

JVM

IronPython

für Microsoft .Net Framework

CLR

Programm in Ausführung (Prozess)

Ausführungsstapel

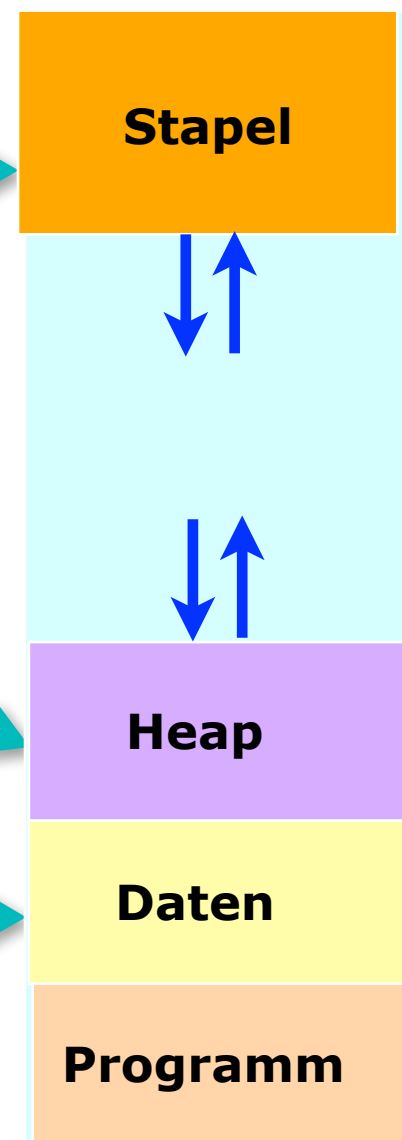
Parameter von Funktionen, *return*-Adressen und lokale Variablen werden hier gespeichert.

Daten, die während der Programmausführung dynamisch erzeugt werden, werden hier verlagert.

Statische Datenstrukturen, die am Anfang der Programmausführung erzeugt werden.

Virtuelle Maschine

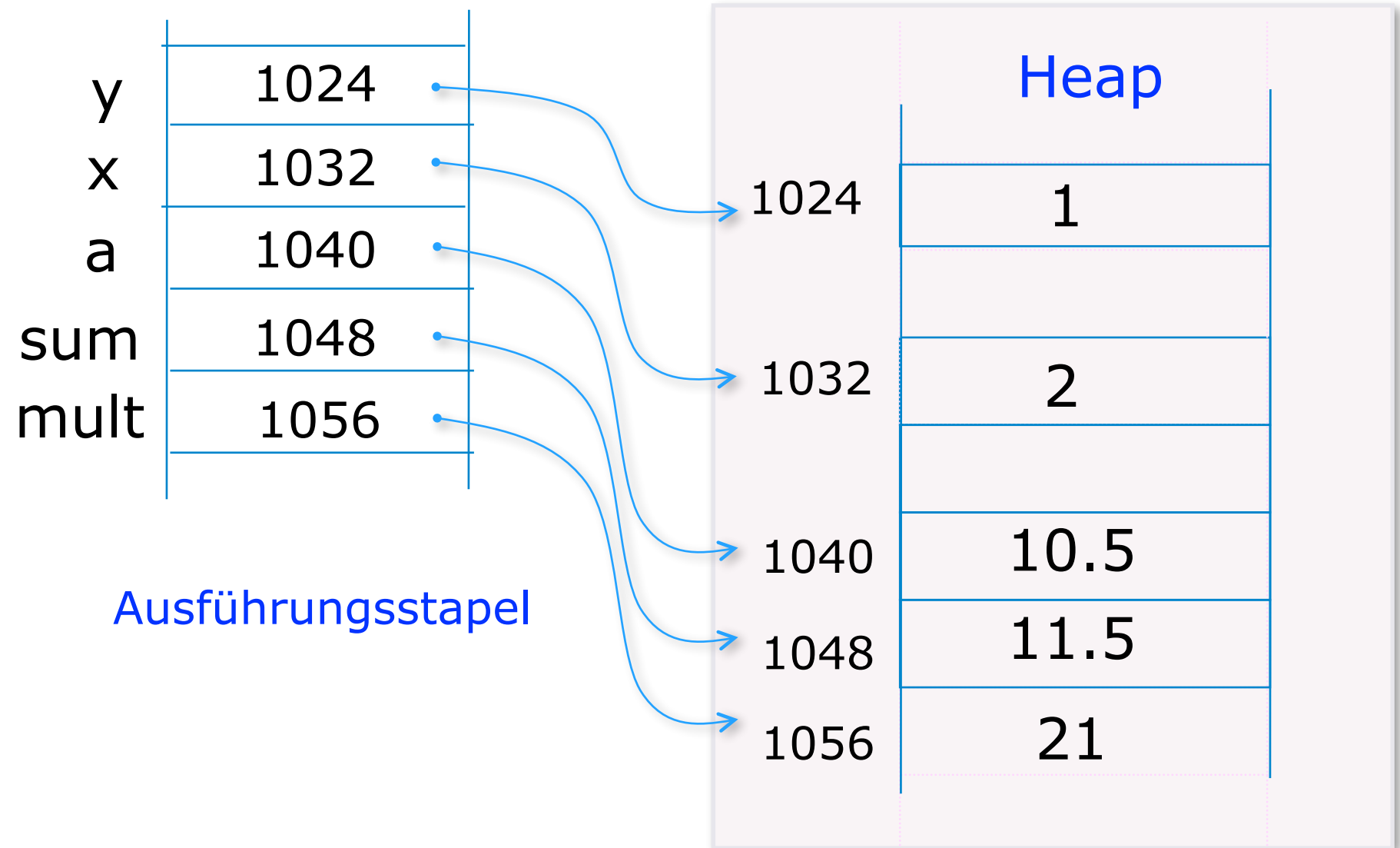
Prozessabbild



Dynamisches Typsystem von Python

Python arbeitet nur mit Referenzen

```
y = 1
x = 2
a = 10.5
sum = y+a
mult = a*x
```



Python arbeitet nur mit Referenzen

Änderbare (*mutable*)

- Listen
- Dictionary

Unveränderbar (*immutable*)

- Integer
- Boolean
- Complex
- Float
- String
- Tupel

Dynamisches Typsystem von Python

Python arbeitet nur mit Referenzen

```
x = 1
y = 1
z = x
print (id(x))
print (id(y))
print (id(z))

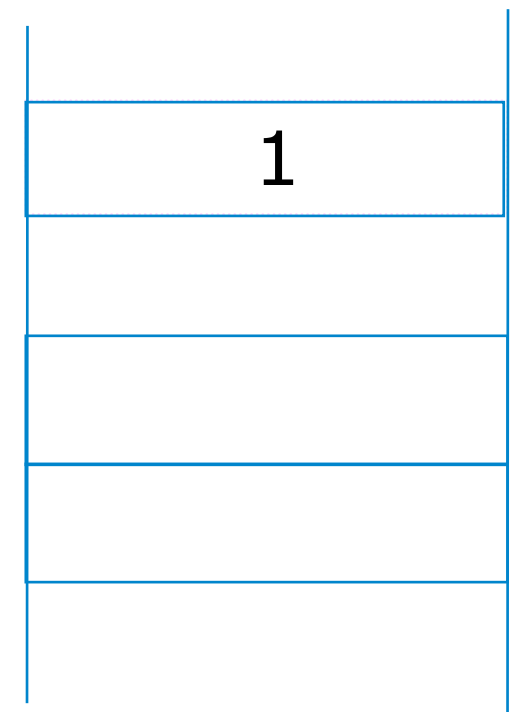
x = 3
print (id(x))
print (id(y))
print (id(z))
```

Ausführungsstapel



```
>>>
```

Heap



Dynamisches Typsystem von Python

Python arbeitet nur mit Referenzen

```
x = 1
```

```
y = 1
```

```
z = x
```

```
print (id(x))
```

```
print (id(y))
```

```
print (id(z))
```

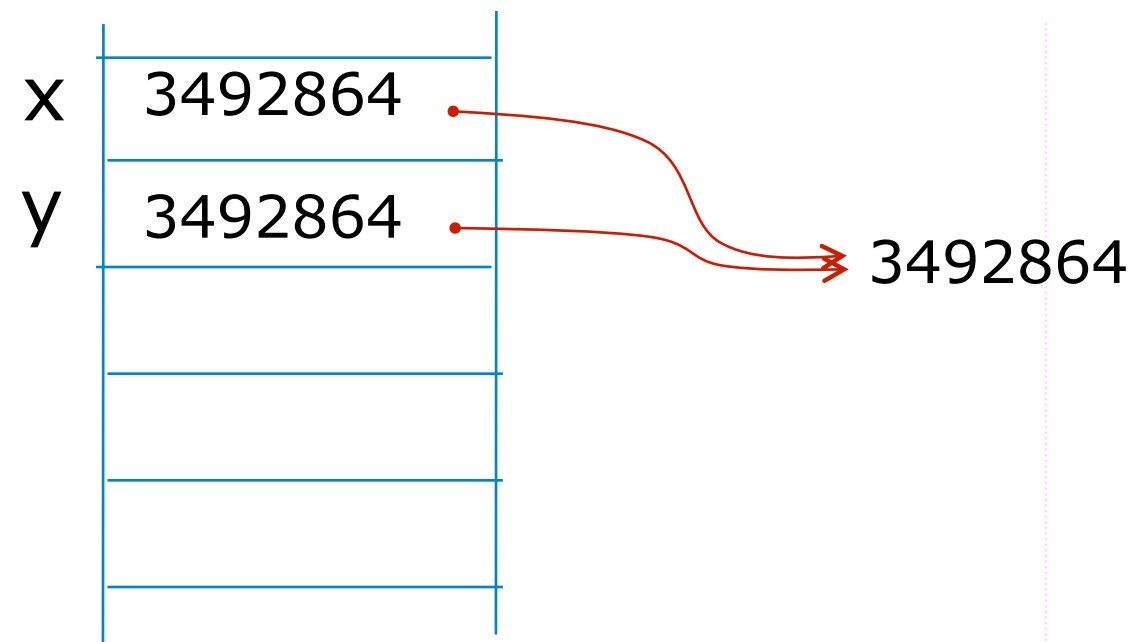
```
x = 3
```

```
print (id(x))
```

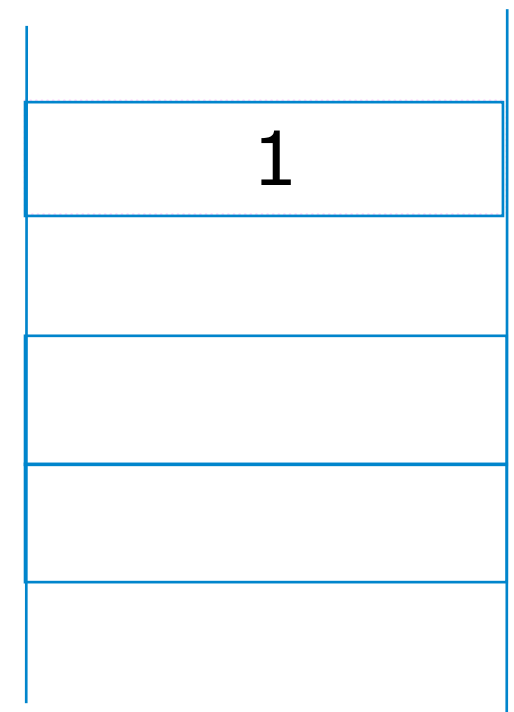
```
print (id(y))
```

```
print (id(z))
```

Ausführungsstapel



Heap



```
>>>
```

Dynamisches Typsystem von Python

Python arbeitet nur mit Referenzen

```
x = 1
```

```
y = 1
```

```
z = x
```

```
print (id(x))
```

```
print (id(y))
```

```
print (id(z))
```

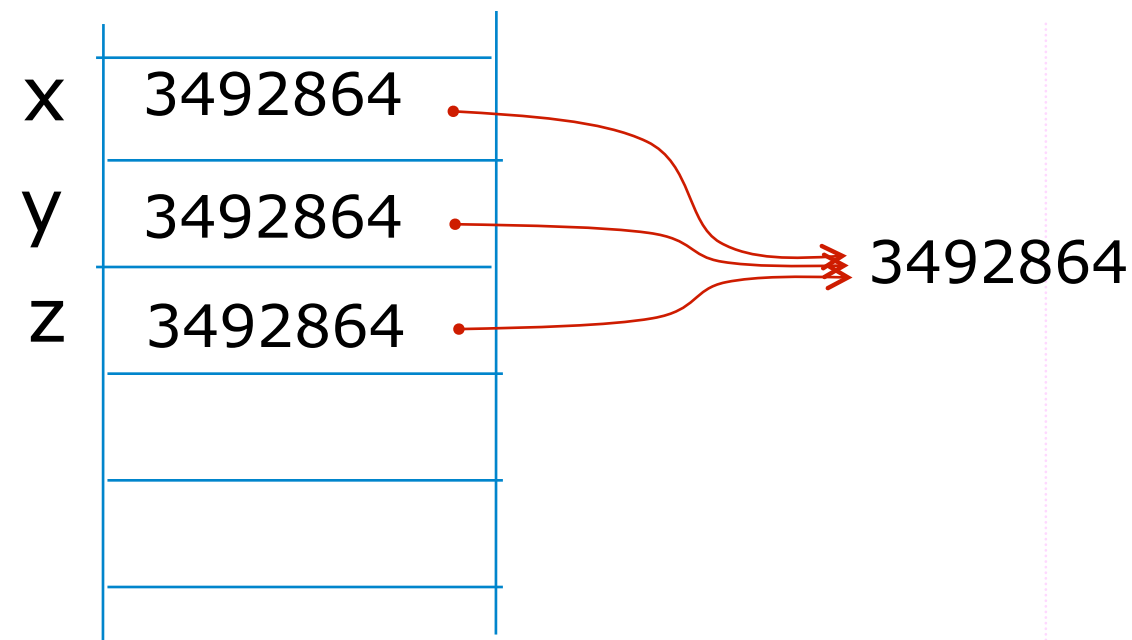
```
x = 3
```

```
print (id(x))
```

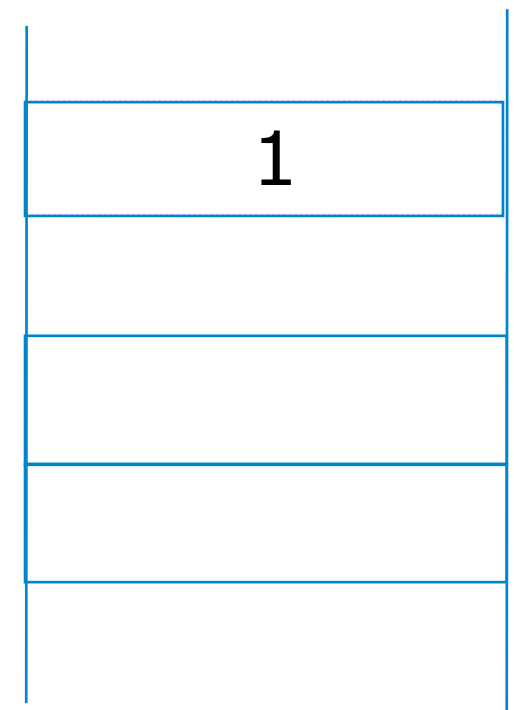
```
print (id(y))
```

```
print (id(z))
```

Ausführungsstapel



Heap



```
>>>
```

Dynamisches Typsystem von Python

Python arbeitet nur mit Referenzen

```
x = 1
y = 1
z = x
```

```
print (id(x))
print (id(y))
print (id(z))
```

```
x = 3
print (id(x))
print (id(y))
print (id(z))
```

Ausführungsstapel

x	3492864
y	3492864
z	3492864

```
>>>
3492864
```

Heap

1

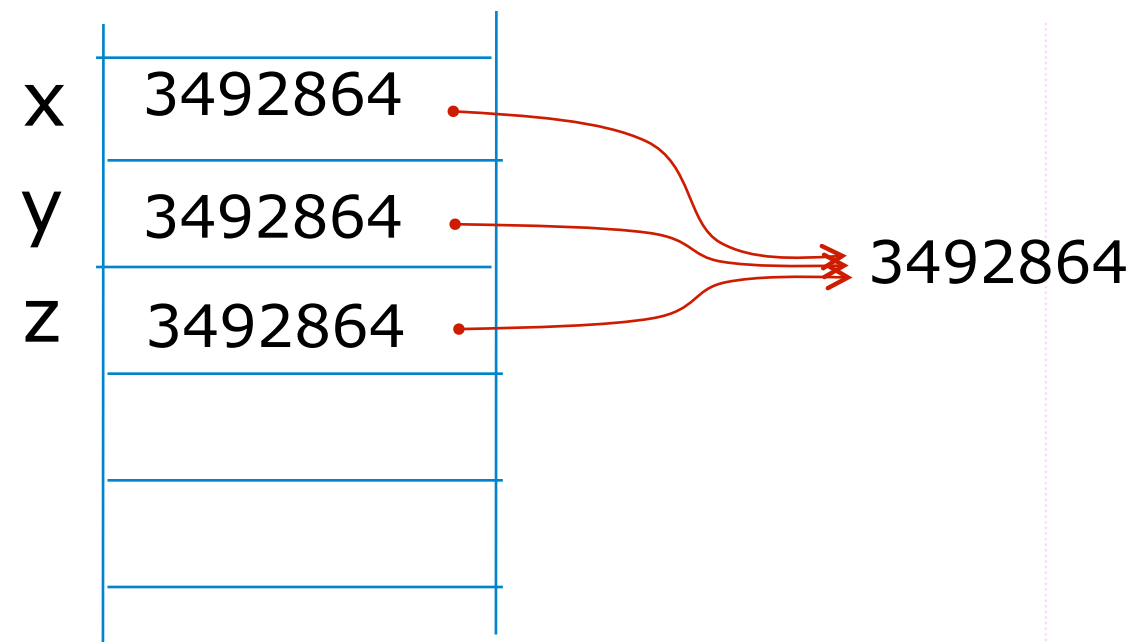
Dynamisches Typsystem von Python

Python arbeitet nur mit Referenzen

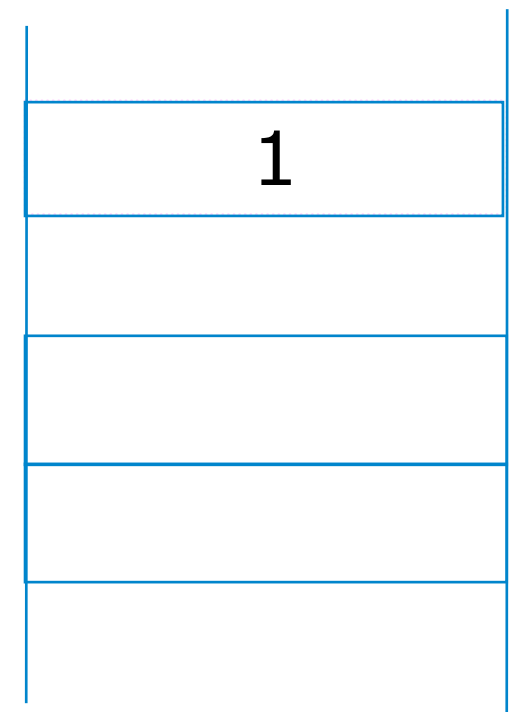
```
x = 1
y = 1
z = x
print (id(x))
print (id(y))
print (id(z))

x = 3
print (id(x))
print (id(y))
print (id(z))
```

Ausführungsstapel



Heap



```
>>>
3492864
3492864
```

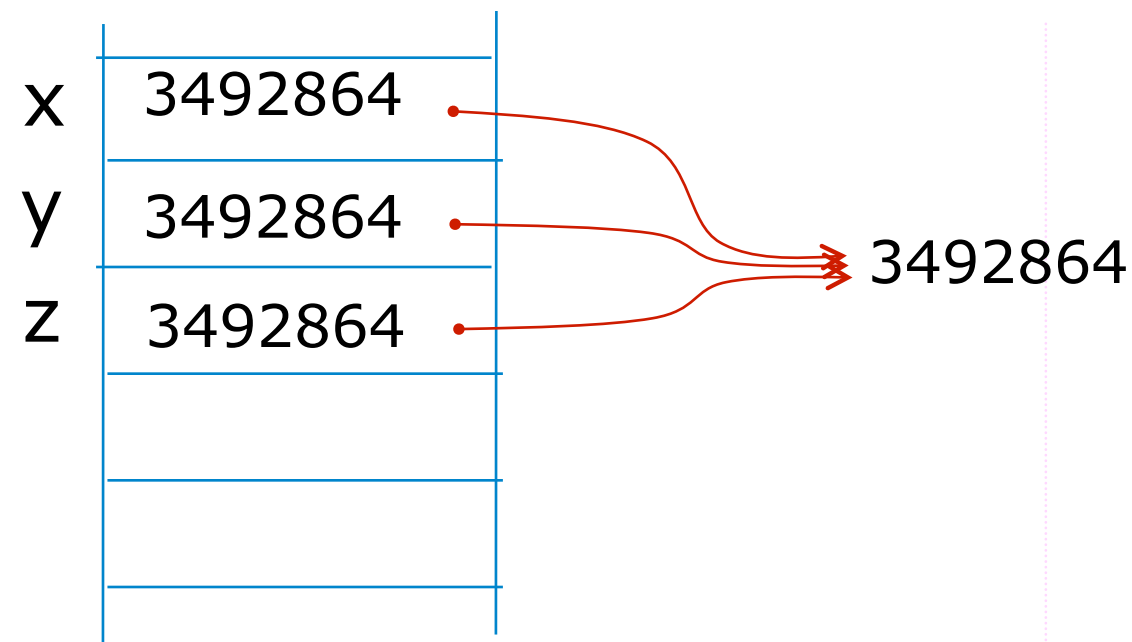
Dynamisches Typsystem von Python

Python arbeitet nur mit Referenzen

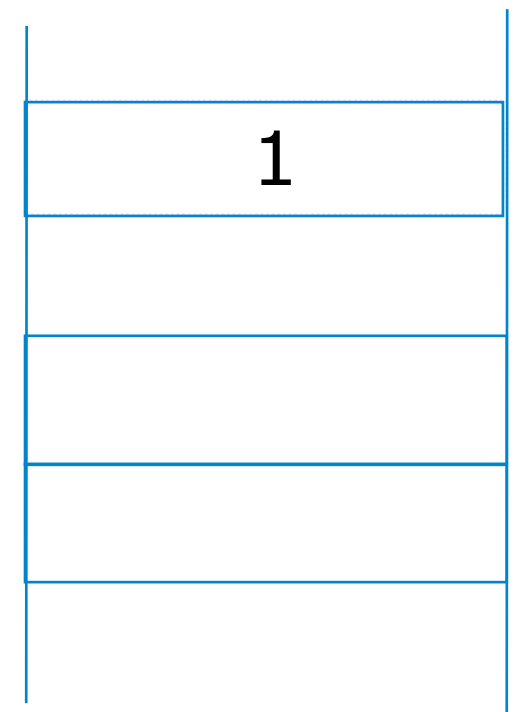
```
x = 1
y = 1
z = x
print (id(x))
print (id(y))
print (id(z))
```

```
x = 3
print (id(x))
print (id(y))
print (id(z))
```

Ausführungsstapel



Heap



```
>>>
3492864
3492864
3492864
```

Dynamisches Typsystem von Python

Python arbeitet nur mit Referenzen

```
x = 1
y = 1
z = x
print (id(x))
print (id(y))
print (id(z))
```

```
x = 3
print (id(x))
print (id(y))
print (id(z))
```

Ausführungsstapel

X	3492896
Y	3492864
Z	3492864

3492864

3492896

Heap

1
⋮
3

```
>>>
3492864
3492864
3492864
```


Dynamisches Typsystem von Python

Python arbeitet nur mit Referenzen

```
x = 1
y = 1
z = x
print (id(x))
print (id(y))
print (id(z))

x = 3
print (id(x))
print (id(y))
print (id(z))
```

Ausführungsstapel

X	3492896
Y	3492864
Z	3492864

```
>>>
3492896
3492864
3492864
```

Heap

1
⋮
3

Python arbeitet nur mit Referenzen

Änderbare (*mutable*)

- Listen
- Dictionary

Unveränderbar (*immutable*)

- Integer
- Boolean
- Complex
- Float
- String
- Tupel

Änderbare und unveränderbare Objekte

```
x = 1
y = 2
m = [[x, y], [x, y]]
print (m)
```

[[1, 2], [1, 2]]

```
x = 7
y = 10
print (m)
```

[[1, 2], [1, 2]]

```
m = [[x, y]*2]
print (m)
```

[[7, 10, 7, 10]]

```
r = [2, 3, 4, 5, 6]
print(r)
print(id(r))
```

[2, 3, 4, 5, 6]

20047992

```
r.append(10)
print(r)
print(id(r))
```

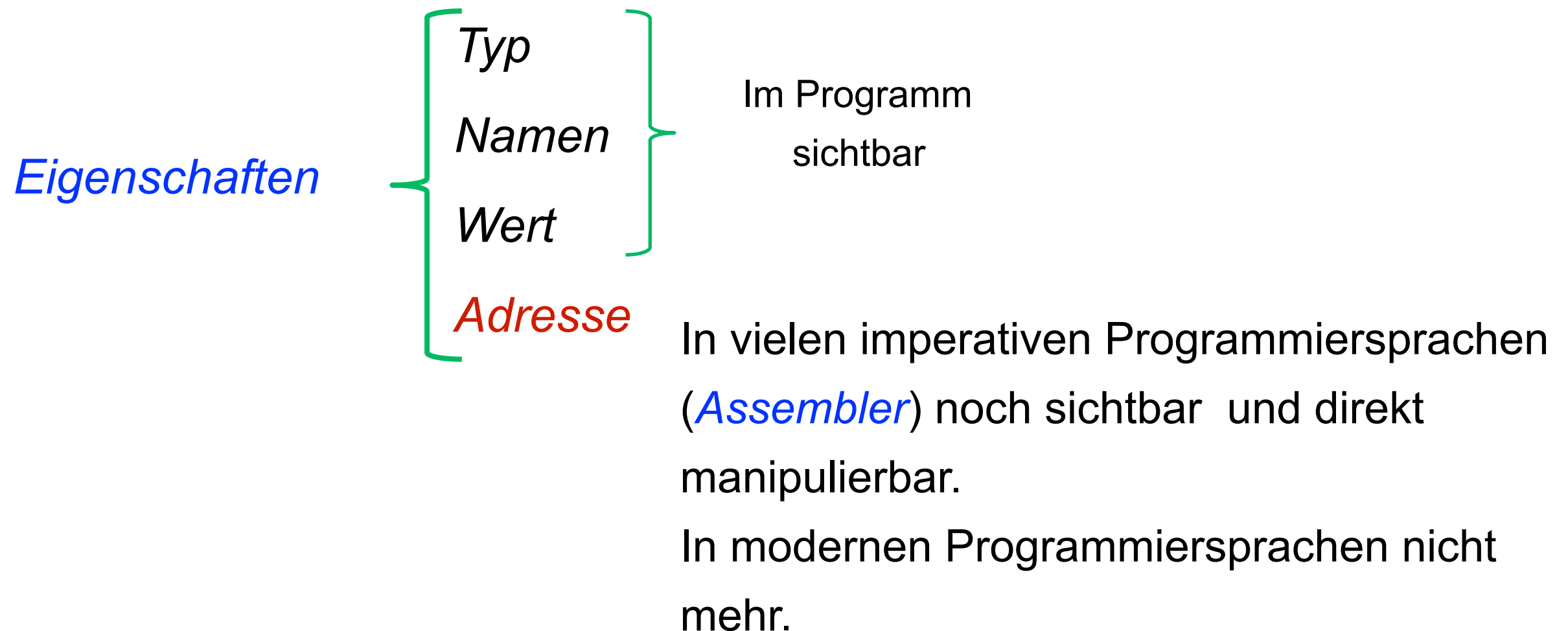
[2, 3, 4, 5, 6, 10]

20047992

Variablen

Imperative Programmiersprachen

Variablen sind Speicherbehälter



Zeigervariablen

Ähnlich wie bei der GOTO-Anweisung sind Zeigervariablen ein **sehr diskutierter Datentyp** in der Welt der imperativen Programmiersprachen.

Zeiger als expliziter Datentyp kommt vor allem in maschinennahen Programmiersprachen wie z. B. **Assembler**, **C** oder **C++** vor. Zeiger dürfen hier auf beliebigen Speicherpositionen stehen und können mit Hilfe von arithmetischen Operationen manipuliert werden.

In **Java** und **Python** sind Referenz-Variablen intern vorhanden, aber für den Programmierer nicht explizit sichtbar.

In **C++** und **C#** gibt es die Möglichkeit explizit mit Zeigern zu arbeiten oder nur implizit, wie in Java, mit Referenzen arbeiten.

Zeiger- und Referenz-Datentypen

Zeigervariablen und Referenzvariablen in imperativen Programmiersprachen stellen **Variablen, die Speicheradressen beinhalten**, dar.

Zeiger- und Referenz-Variablen haben die Macht der **Indirekten Adressierung**.

Mit Hilfe von Zeiger- und Referenz-Datentypen können **dynamische Datenstrukturen** erzeugt werden.

Dynamische Datenstrukturen, die erst zur Laufzeit entstehen, befinden sich in dem **heap**-Bereich eines Prozesses (Programm in Ausführung).

Parameter-Übergabe mit nicht veränderbaren Datentypen

```
def changeDouble ( d = 1.3 ):  
    d = 2  
    print( d )  
    print( id (d) )
```

```
a = 2.5  
changeDouble ( a )  
print ( a )  
print ( id ( a ) )
```

Ausgabe?

```
2  
3492880  
2.5  
8466964
```

Parameter-Übergabe mit unveränderbaren Datentypen

```
def changeDouble( d=1.3 ):
```

```
    # d = 2
```

```
    print( d )
```

```
    print( id(d) )
```

```
a = 2.5
```

```
changeDouble( a )
```

```
print( a )
```

```
print( id(a) )
```

Ausgabe?

2.5

8466964

2.5

8466964

Rekursive Funktionen

Funktionen können rekursiv definiert werden.

Beispiel:

```
def fact (n) :  
    if n==0:  
        return 1  
    else:  
        return n*fact(n-1)
```

Anwendung:

```
>>> print( fact(5) )  
>>> 120
```

Funktionsdokumentation

Funktionen können einen Dokumentationstext beinhalten, der als Blockkommentar in der **ersten Zeile der Funktion** geschrieben werden muss.

```
def fact (n) :  
    """ Berechnet die Fakultätsfunktion der Zahl n """  
    if n==0:  
        return 1  
    else:  
        return n*fact(n-1)
```

```
>>> help (fact)  
Help on function fact in module __main__:  
fact(n)  
    Berechnet die Fakultätsfunktion der Zahl n
```

Funktionen als Objekte

- In Python sind Funktionen Objekte (**first-class objects**)
- haben ein Datentyp

```
>>> type ( factorial )
```

```
>>> <type 'function'>
```

- ermöglicht Meta-Programmierung

higher-order functions

aus FP:

Eine Funktion wird als **Funktion höherer Ordnung** bezeichnet, wenn **Funktionen als Argumente** verwendet werden oder wenn eine **Funktion als Ergebnis** zurück gegeben wird.

Funktionen als Objekte

```
def myMap(ls, f):  
    """assumes ls is a list and f is a function"""  
    for i in range(len(ls)):  
        ls[i] = f(ls[i])
```

```
list1 = [2, 3, 4, 5, 0, 1]  
myMap(list1, factorial)  
print ( list1 )
```

Ausgabe:

```
>>>
```

```
[2, 6, 24, 120, 1, 1]
```

Funktionen höherer Ordnung

```
def myMap (f, xs):  
    """ assumes f is a function and xs is a list """  
    result = []  
    for x in xs:  
        result.append(f(x))  
    return result
```

```
nums = [2,3,4,5,0,1]  
result_list = myMap (factorial, nums)  
print(nums)  
print(result_list)
```

Ausgabe:

```
>>>
```

```
[2, 3, 4, 5, 0, 1]
```

```
[2, 6, 24, 120, 1, 1]
```

Funktionen als Objekte

```
import sys
def print_char_picture(decide_char_func):
    size = 40
    for i in range( 0, size):
        for j in range( 0, size):
            sys.stdout.write( decide_char_func( j, i, size) )
        print()

def diagonal( x, y, size):
    if x==y:
        return '@'
    else:
        return '.'

def grid( x, y, size):
    if (x%4==0) or (y%4==0):
        return '.'
    else:
        return ' '

print_char_picture(diagonal)
print_char_picture(grid)
```

del-Anweisung

Die Bindung eines Variablennamens zu einem Objekt wird aufgehoben.

Das Objekt bleibt noch im Speicher bis keine Variable mehr auf dieses Objekt zeigt und wird dann vom *Garbage Collector* beseitigt.

Beispiel:

```
a = 100
```

```
del a
```

```
print (a)
```

➔ Laufzeitfehler!

is-Anweisung

Der Ausdruck

`a is b`

liefert genau dann den Wahrheitswert **True**,
wenn **a** und **b** identisch sind.

Beispiel:

```
>>> a = [100, 200, 300]
>>> b = a
>>> a is b
True
```

```
>>> a = [100, 200, 300]
>>> b = [100, 200, 300]
>>> a is b
False
>>> a == b
True
```


pass-Anweisung

Die **pass**-Anweisung bewirkt nichts. Sie wird als **Platzhalter** bei Verzweigungen aus rein syntaktischen Gründen benötigt, um Einrückungsfehler während der Entwicklungsphase zu vermeiden.

Beispiel:

```
x = int(input("x="))
if x>0:
    pass
else:
    print("x is negativ")
```

from-Anweisung

Beispiel:

```
from math import sin, cos
print(sin(0))
```

exec-Anweisung

Die `exec`-Anweisung wird verwendet, um Python-Anweisungen auszuführen, die in einem String oder in einer Datei gespeichert sind.

Python-Skripte können zur Laufzeit erzeugt werden und mittels der `exec`-Anweisung ausgeführt werden.

Beispiel:

```
>>> exec( 'print("Hello")' )
Hello
>>> exec( 'print(2*3**2)' )
18
```

break-Anweisung

Die **break**-Anweisung wird verwendet, um die Ausführung einer Schleife vorzeitig zu beenden.

```
while True:
```

```
    s = input( 'Text eingeben: ' )
```

```
    if s == 'end':
```

```
        break
```

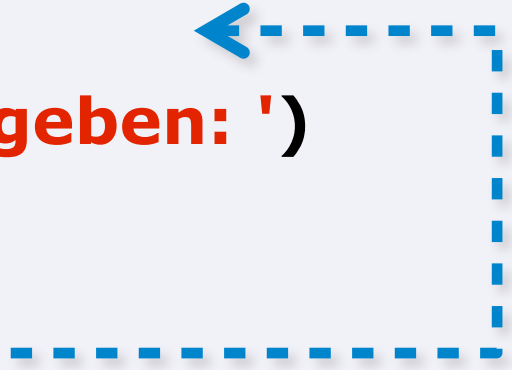
```
    print ( 'Die Laenge des Texts ist', len(s) )
```

```
print ( 'Tchüss.' )
```

continue-Anweisung

Die **continue**-Anweisung wird verwendet, um die restlichen Anweisungen der aktuellen Schleife zu überspringen und direkt mit dem nächsten Schleifen-Durchlauf fortzufahren.

```
while True:
    s = input('Text eingeben: ')
    if s == 'kein print':
        continue
    print ('Die Laenge des Texts ist', len(s))
```



test-Funktionen

```
import random

def sum(nums):
    summe = 0
    for i in range(len(nums)):
        summe = summe + int(nums[i])
    return summe

def test_sum():
    print(sum([]))
    print(sum([9, 0, 3, 2, 5, 4, 9, 0, 9, 110]))

def test_sum1():
    array = []
    for i in range(10):
        array.append( random.randint(1,100) )
    print("sum( ", array, ")= ", sum(array))

def test_sum2():
    print("type the numbers to be added separated with spaces: ")
    array = list(map(int, input().split()))
    print("sum( ", array, ")= ", sum(array))
```

```
>>>
0
151
sum( [17, 15, 6, 79, 85, 52, 75, 76, 35, 17] )= 457
type the numbers to be added separated with spaces:
17 15 6 79 85 52 75 76 35 17
sum( [17, 15, 6, 79, 85, 52, 75, 76, 35, 17] )= 457
```

yield-Anweisung

Die **yield**-Anweisung innerhalb einer Funktion **f** verursacht einen Rücksprung in die aufrufende Funktion und der Wert hinter der **yield**-Anweisung wird als Ergebnis zurückgegeben.

Im Unterschied zur **return**-Anweisung werden die aktuelle Position innerhalb der Funktion **f** und ihre lokalen Variablen zwischengespeichert.

Beim nächsten Aufruf der Funktion **f** springt Python hinter dem zuletzt ausgeführten **yield** weiter und kann wieder auf die alten lokalen Variablen von **f** zugreifen.

Wenn das Ende der Funktion **f** erreicht wird, wird diese endgültig beendet.

yield-Anweisung

```
def myRange(n):  
    i = 0  
    while (i < n):  
        yield i  
        i = i + 1  
  
for i in myRange(5):  
    print(i)
```

```
>>>  
0  
1  
2  
3  
4  
>>>
```

```
def genConstants():  
    yield 3  
    yield 5  
    yield 11  
  
def testMyRange():  
    for x in genConstants():  
        print(x)  
  
testMyRange()
```

```
>>>  
3  
5  
11  
>>>
```

yield-Anweisung

```
def fibonacci():  
    """Unendlicher Fibonacci-Zahlen-Generator"""  
    a, b = 0, 1  
    while True:  
        yield a  
        a, b = b, a + b  
  
def getFibonacci(n):  
    counter = 0  
    for x in fibonacci():  
        counter += 1  
        if (counter > n):  
            break  
    return x  
  
print(getFibonacci(77))
```

```
>>>  
5527939700884757
```


Listen-Generatoren

Python:

```
[ x*x for x in range (5) ]
```

Eine Liste mit den Quadratzahlen von 0 bis 4 wird generiert.

```
>>> [ x%3 for x in [1, 5, -3, -6, 4, 7, 6, 0] if (x>0) ]
```

```
[1, 2, 1, 1, 0]
```

Neue Anweisungen in Python

Wort

kurze Erläuterung

from	Teil einer import-Anweisung
global	Verlegung einer Variablen in den globalen Namensraum
is	test auf Identität
pass	Platzhalter, führt nichts aus
exec	Ausführung von Programmcode
yield	...

Höhere Datenstrukturen

Dictionaries

Beispiele:

```
>>> synonyms = {}
>>> synonyms['pretty'] = 'beautiful'
>>> synonyms['shy'] = 'timid'
>>> synonyms['easy'] = 'facile'
>>> synonyms
{'shy': 'timid', 'easy': 'facile', 'pretty': 'beautiful'}
>>> synonyms['easy']
'facile'
>>> 'pretty' in synonyms
>>> True
```

Verwendung von "*Dictionaries*"

In Python gibt es keine **switch-case**-Anweisung wie in **C** und in **Java**, aber diese Anweisungen können mit Hilfe von *Dictionaries* simuliert werden.

```
def zero(): print( "You typed zero. " )
def sqr(): print( "n is a perfect square " )
def even(): print( "n is an even number " )
def prime(): print( "n is a prime number " )

options = {0 : zero,
           1 : sqr,
           4 : sqr,
           9 : sqr,
           2 : even,
           3 : prime,
           5 : prime,
           7 : prime,
           }
options[4]()
```

Reservierte Wörter in Python

help> keywords

and	else	import	raise
assert	except	in	return
break	exec	is	try
class	finally	lambda	while
continue	for	not	yield
def	from	or	False
del	global	pass	True
elif	if	print	...