

Algorithmen und Programmieren II

Python (Teil 4)



SoSe 2020

Prof. Dr. Margarita Esponda

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!

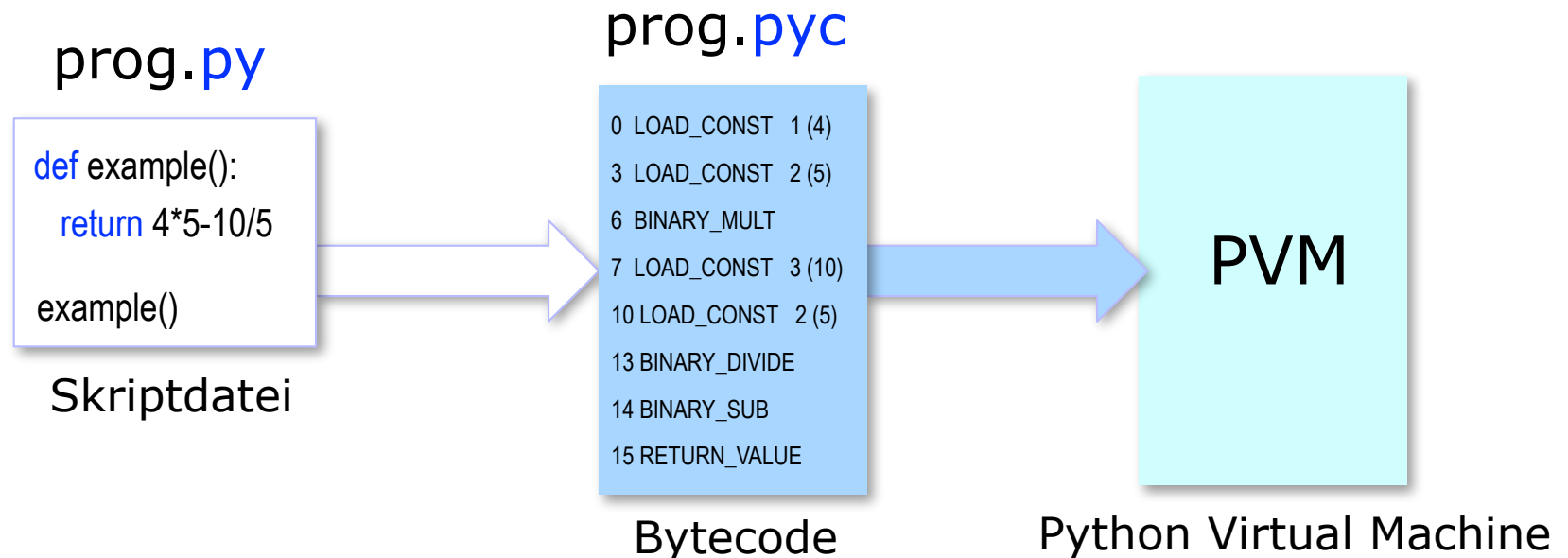
↓
`y = 1`
`x = 2`
`a = 10.5`
`sum = y+a`
`mult = a*x`

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)

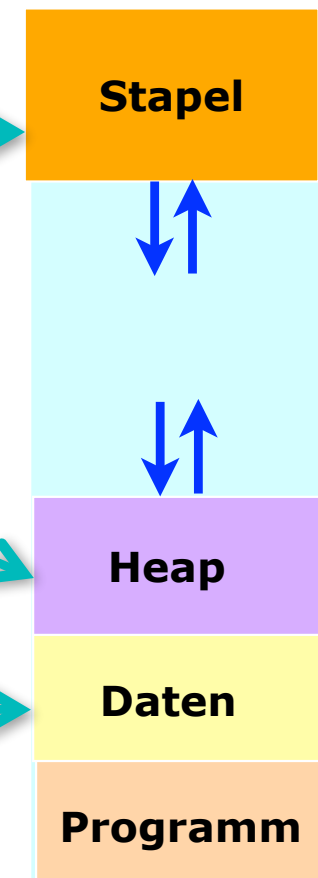
Ausführungsstapel

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.

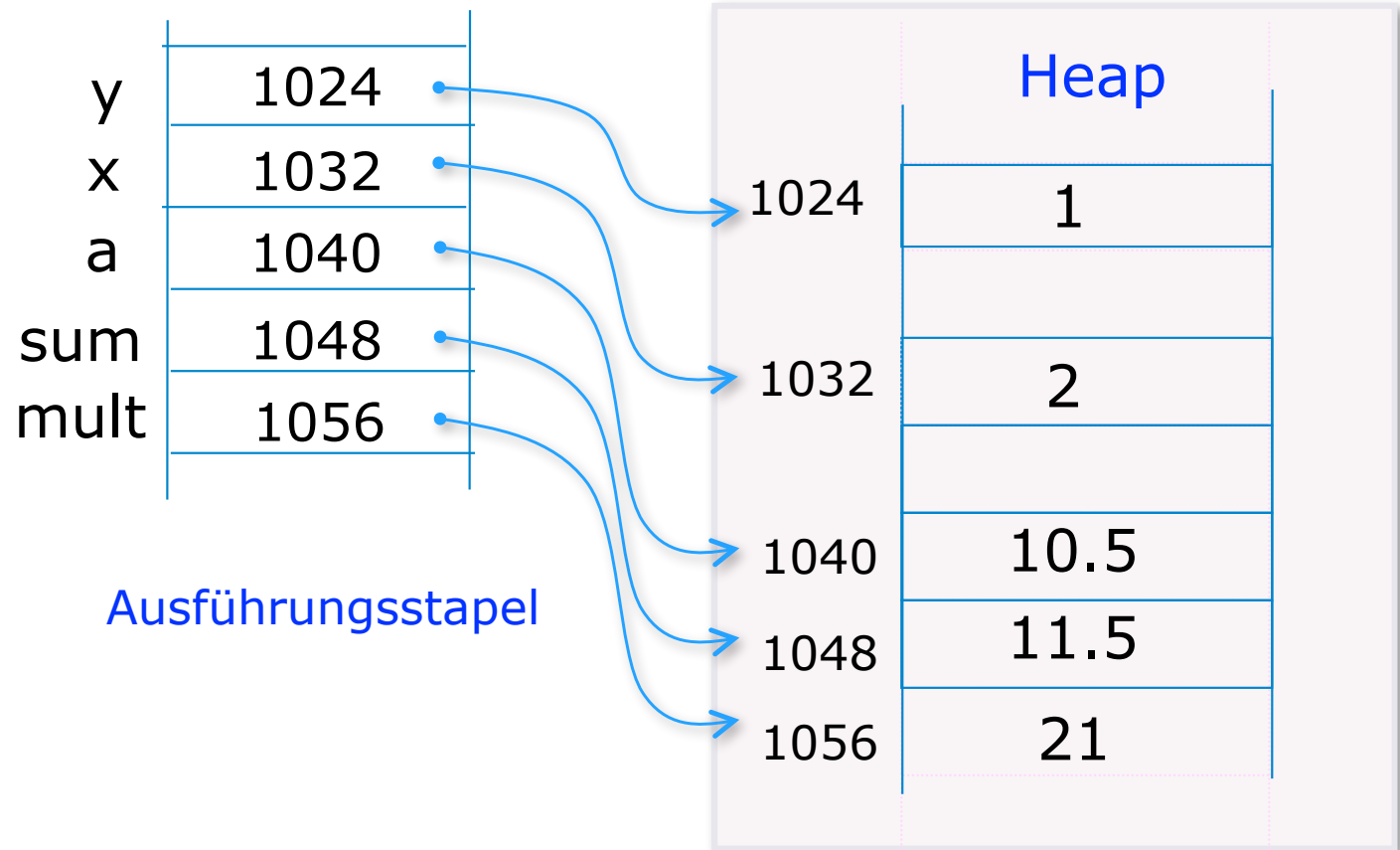
Virtuelle Maschine Prozessabbild



Dynamisches Typsystem von Python

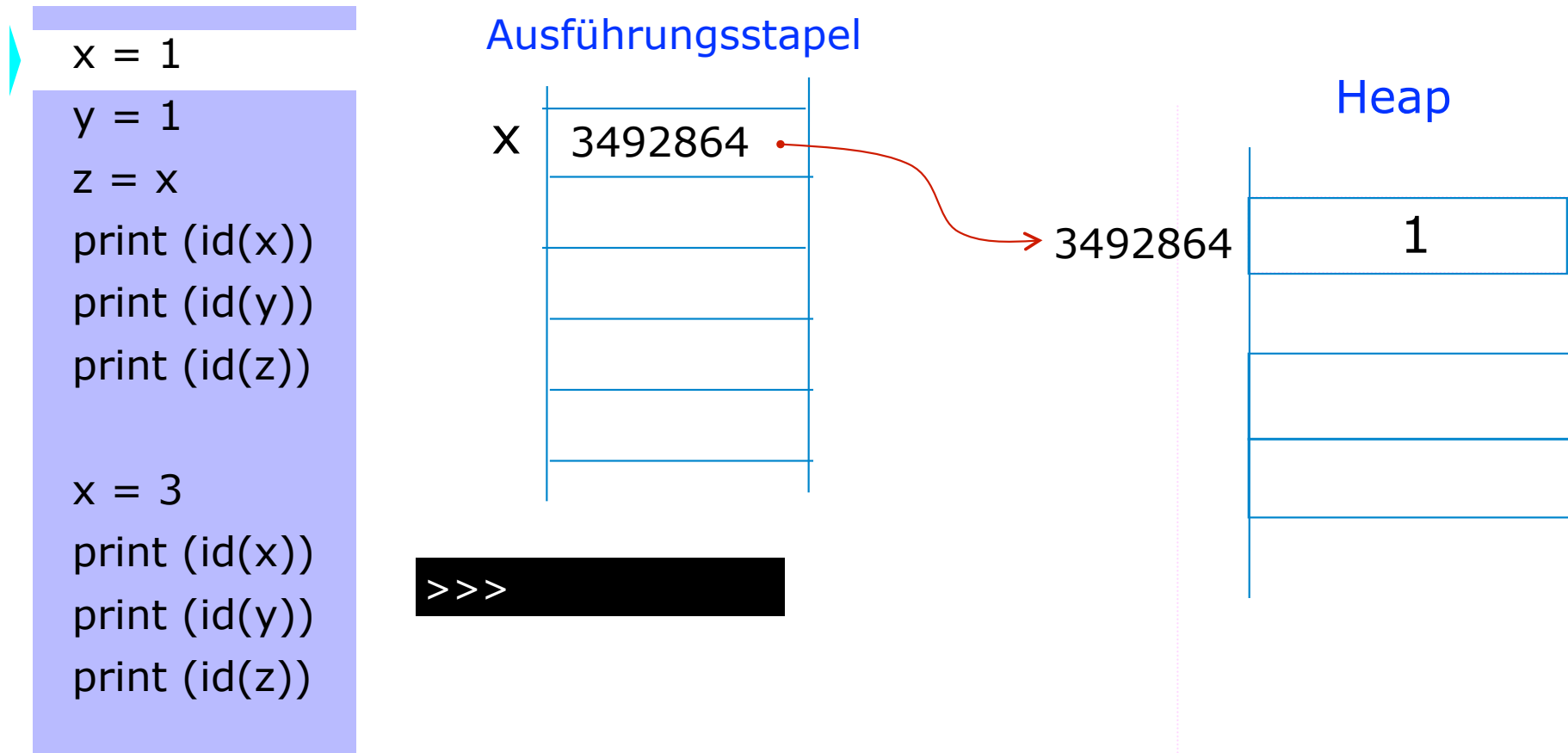
Python arbeitet nur mit Referenzen

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



Dynamisches Typsystem von Python

Python arbeitet nur mit Referenzen



Dynamisches Typsystem von Python

Python arbeitet nur mit Referenzen

```
x = 1
```

```
y = 1
```

```
z = x
```

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

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

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

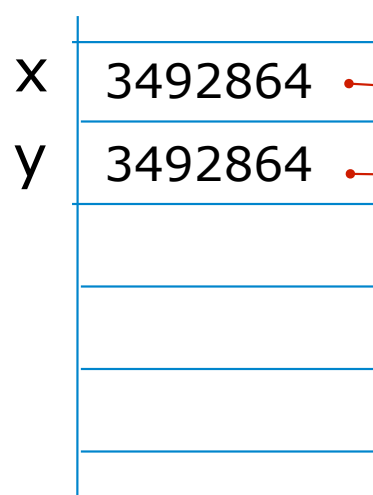
```
x = 3
```

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

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

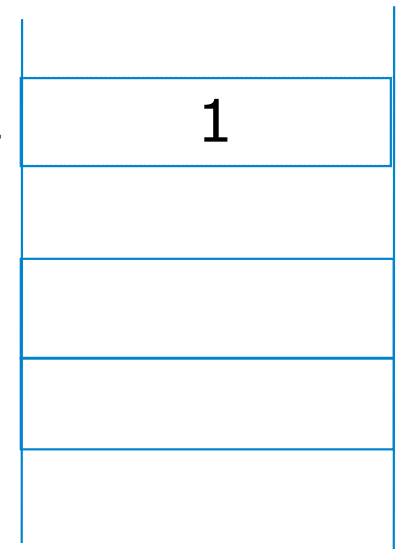
```
print (id(z))
```

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))
```

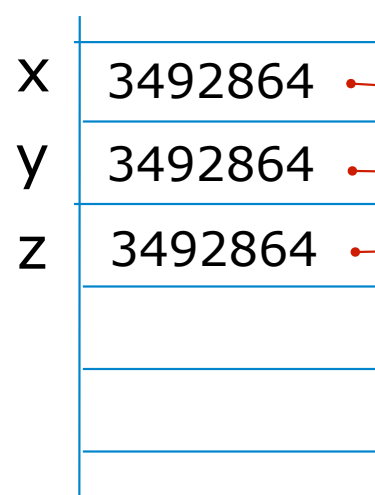
```
x = 3
```

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

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

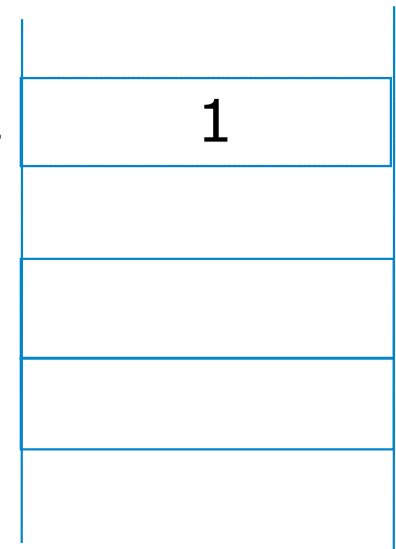
```
print (id(z))
```

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

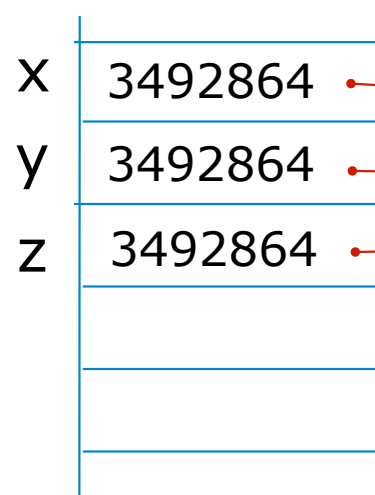
Dynamisches Typsystem von Python

Python arbeitet nur mit Referenzen

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

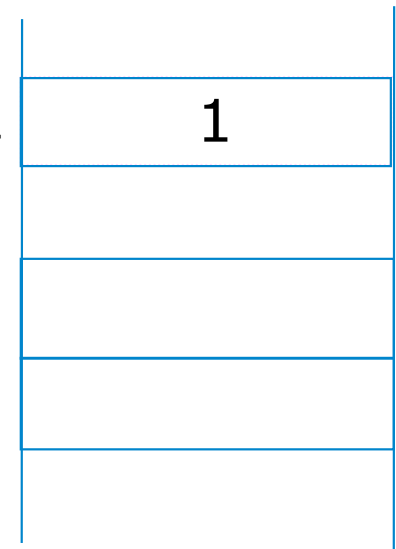
```
x = 3
print (id(x))
print (id(y))
print (id(z))
```

Ausführungsstapel



3492864

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

X	3492864
Y	3492864
Z	3492864

3492864

Heap

1

```
>>>
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

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)            → [2, 3, 4, 5, 6]
print(id(r))        → 20047992

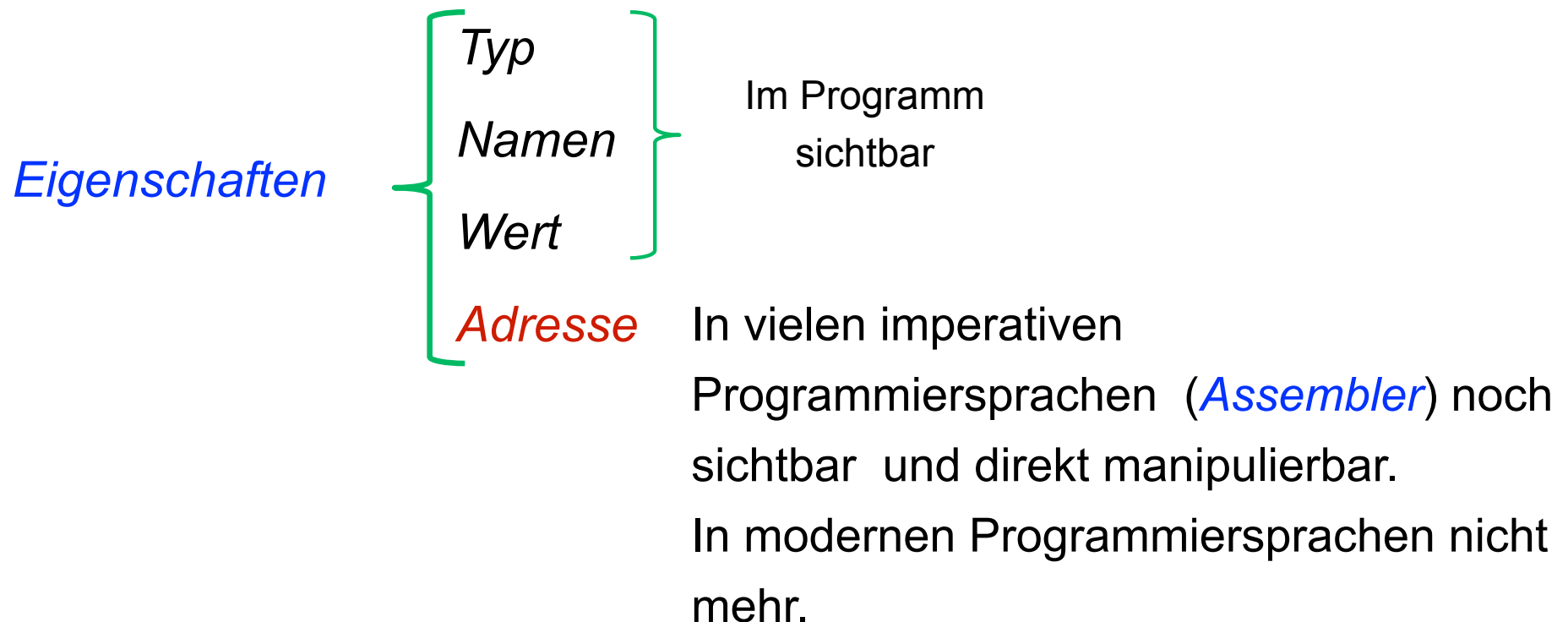
r.append(10)
print(r)            → [2, 3, 4, 5, 6, 10]
print(id(r))        → 20047992

```

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 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 Funktionen


call-by-reference

- Variablen werden bei der Parameterübergabe an Funktionen nicht kopiert.
- Ein Zeiger (Reference) auf die Variable wird übergeben
- Alle Änderungen der Variablen innerhalb der Funktion bleiben nach der Ausführung der Funktion erhalten.


Parameter-Übergabe in Funktionen


call-by-value

cup = 


fillCup()


nach der Ausführung der fillCup()-Funktion


cup = 


fillCup()


call-by-reference

cup = 

fillCup()

cup = 

fillCup()



Parameter-Übergabe in Python-Funktionen

"call-by-object"

"Call by Object Reference" or "Call by Sharing"

Beim Aufruf eine 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.

Parameter-Übergabe mit nicht veränderbaren Datentypen

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

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

Ausgabe?

```
2.5  
4470739696  
2.5  
4470739696
```

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  
4563061440  
2.5  
4568671376
```

Parameter-Übergabe mit veränderbaren Datentypen

```
def changeList ( list=[1, 2, 3, 4] ):  
    list[0] = 100  
    print( list )  
    print( id(list) )
```

```
a = [5, 7, 8]  
changeList(a)  
print(a)  
print( id(a) )
```

Ausgabe?

```
[100, 7, 8]  
6488192  
  
[100, 7, 8]  
6488192
```

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

```
myMap (f, [])    = []
myMap (f, (x:xs)) = (f x) : myMap (f, xs)
```

```
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))
```



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 += 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(10))
```

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]
```

Online **Python** Tutor - Visualize program execution

<http://www.pythontutor.com/visualize.html>

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	Ausführung von Programmcode

Reservierte Wörter in Python

help> keywords

and	else	import	pass
assert	except	in	print
break	exec	is	raise
class	finally	lambda	return
continue	for	not	try
def	from	None	while
del	global	nonlocal	with
elif	if	or	yield
			finally

Effiziente Lösung von Problemen

