

Objektorientierte Programmierung (ALP II)

Python (Teil 3)



SoSe 2020

Prof. Dr. Margarita Esponda

Bit-Operatoren

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

Bit-Operatoren

Beispiele:

Ausdruck			Wert
<code>~5</code>	<code>~00000101</code>	<code>11111010</code>	<code>-6</code>
<code>5<<3</code>	<code>00000101<<3</code>	<code>00101000</code>	<code>40</code>
<code>5>>3</code>	<code>00000101>>3</code>	<code>00000000</code>	<code>0</code>
<code>-10>>2</code>	<code>11110110>>2</code>	<code>11111101</code>	<code>-3</code>
<code>3&10</code>	<code>00000011&00001010</code>	<code>00000010</code>	<code>2</code>
<code>3 10</code>	<code>00000011 00001010</code>	<code>00001011</code>	<code>11</code>
<code>3^10</code>	<code>00000011^00001010</code>	<code>00001001</code>	<code>9</code>

Imperatives Programmieren

Letzte Vorlesung



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**-, **continue**-, **return**-Anweisung usw.

while-Anweisung

while *Ausdruck* :
Anweisungen

Letzte Vorlesung



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)
```



?

while-Schleifen

Glücksspieler

```
import random

bargeld = int(input("bargeld = "))

while bargeld > 0:
    if random.randint(0,1):
        bargeld -= 1
    else:
        bargeld += 1

print("You are a great loser!")
```

while-Schleifen

```
import random

bargeld = int(input("bargeld = "))

while bargeld > 0:
    print(bargeld, end=': ')
    counter = bargeld
    while counter > 0:
        print('$', end='')
        counter -= 1
    if random.randint(0,1):
        bargeld -= 1
    else:
        bargeld += 1
    print()

print("You are a great loser!")
```

```
>>>
bargeld = 5
5:$$$$$
4:$$$$$
3:$$$$
4:$$$$$
5:$$$$$
4:$$$$$
5:$$$$$
6:$$$$$$$
7:$$$$$$$
8:$$$$$$$$$
9:$$$$$$$$$$$
8:$$$$$$$$$
7:$$$$$$$
6:$$$$$$$
5:$$$$$
6:$$$$$
7:$$$$$$$
6:$$$$$
5:$$$$$
4:$$$$$
3:$$$$
2:$
3:$$$
2:$
1:$
2:$
1:$
>>>
```


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**-, **continue**-, **return**-Anweisung usw.

heute



break-Anweisung

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

while True:

s = input('text: ')

if s == 'end':

break

print ('the length of the is = ', len(s))

print ('Tchüss.')

Python Programm

```
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)
```

Python Programm

```
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)
```

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: ')
    if s == 'no print':
        continue
    print ('length of the text', len(s))
```



Höhere Datenstrukturen

Python unterstützt vier *sequentielle höhere Datentypen*

Listen (dynamic arrays)

Tuples (immutable lists)

Dictionaries (hash tables)

Höhere Datenstrukturen

Dictionaries *Dictionaries* sind eine Sammlung von Schlüssel- und Wertpaaren.

Ein Dictionary ist also eine Liste aus Schlüsseln (*keys*), denen jeweils ein Wert (*value*) zugewiesen ist.

Beispiele: `{}` **# Leeres Wörterbuch (*Dictionary*)**

`{ 1:'Goethe', 2:'Schiller', 3:5.67 }`

`atomic_num = {'None' : 0, 'H' : 1, 'He' : 2}`

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
```


Höhere Datenstrukturen

Dictionaries

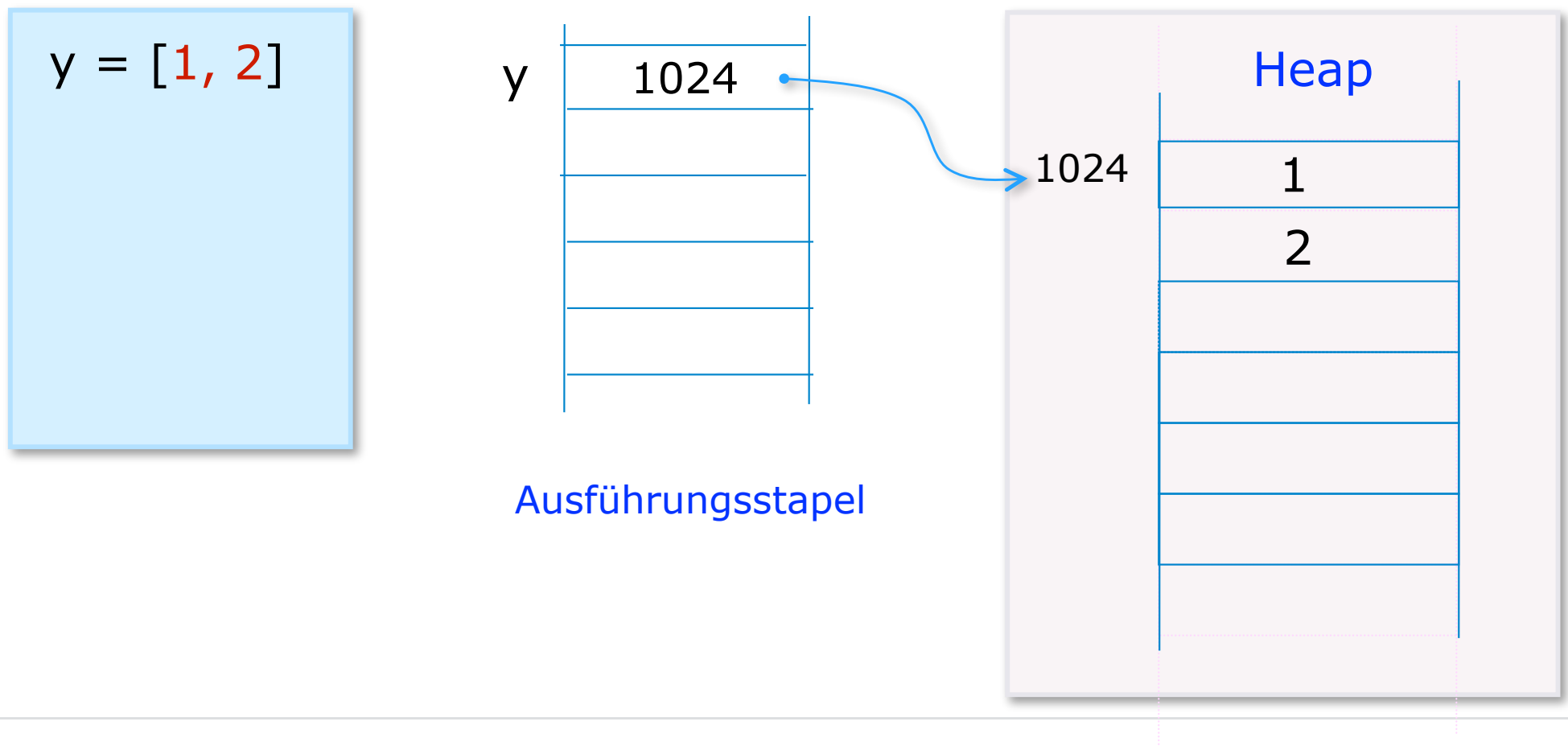
- Beliebige Datentypen können kombiniert werden.
- Sehr effizient implementiert mit Hilfe von Hashtabellen.

Beispiele:

```
>>> dic = {}  
>>> dic[1] = 'Hi'  
>>> dic['a'] = 'Hallo'  
>>> dic[3.1416] = 'pi'  
>>> dic[(1,2,3,4)] = 'Reihe'  
>>> dic  
{'a': 'Hallo', 1: 'Hi', (1, 2, 3, 4): 'Reihe', 3.1416: 'pi'}  
>>>
```

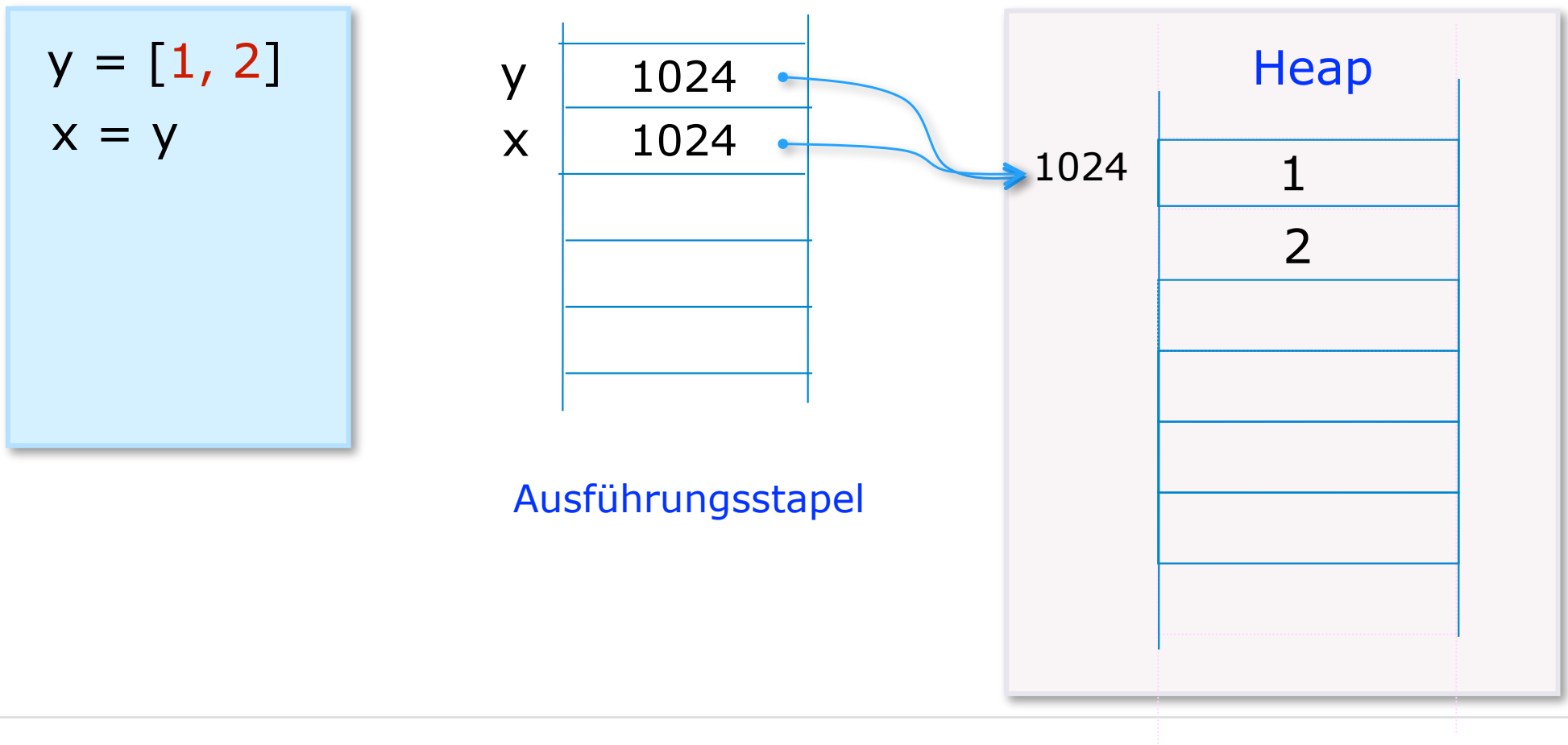
Dynamisches Typsystem von Python

Python arbeitet nur mit Referenzen



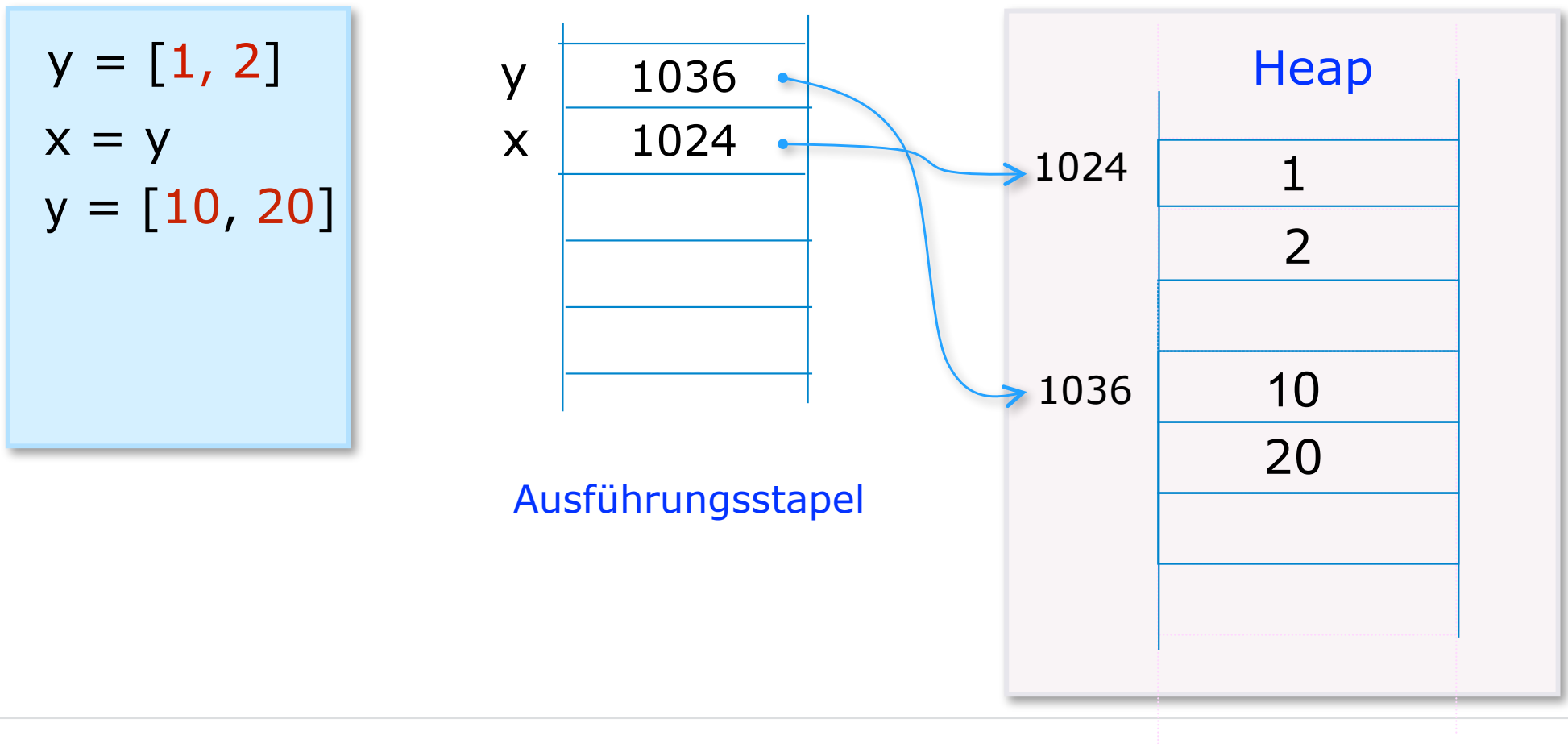
Dynamisches Typsystem von Python

Python arbeitet nur mit Referenzen



Dynamisches Typsystem von Python

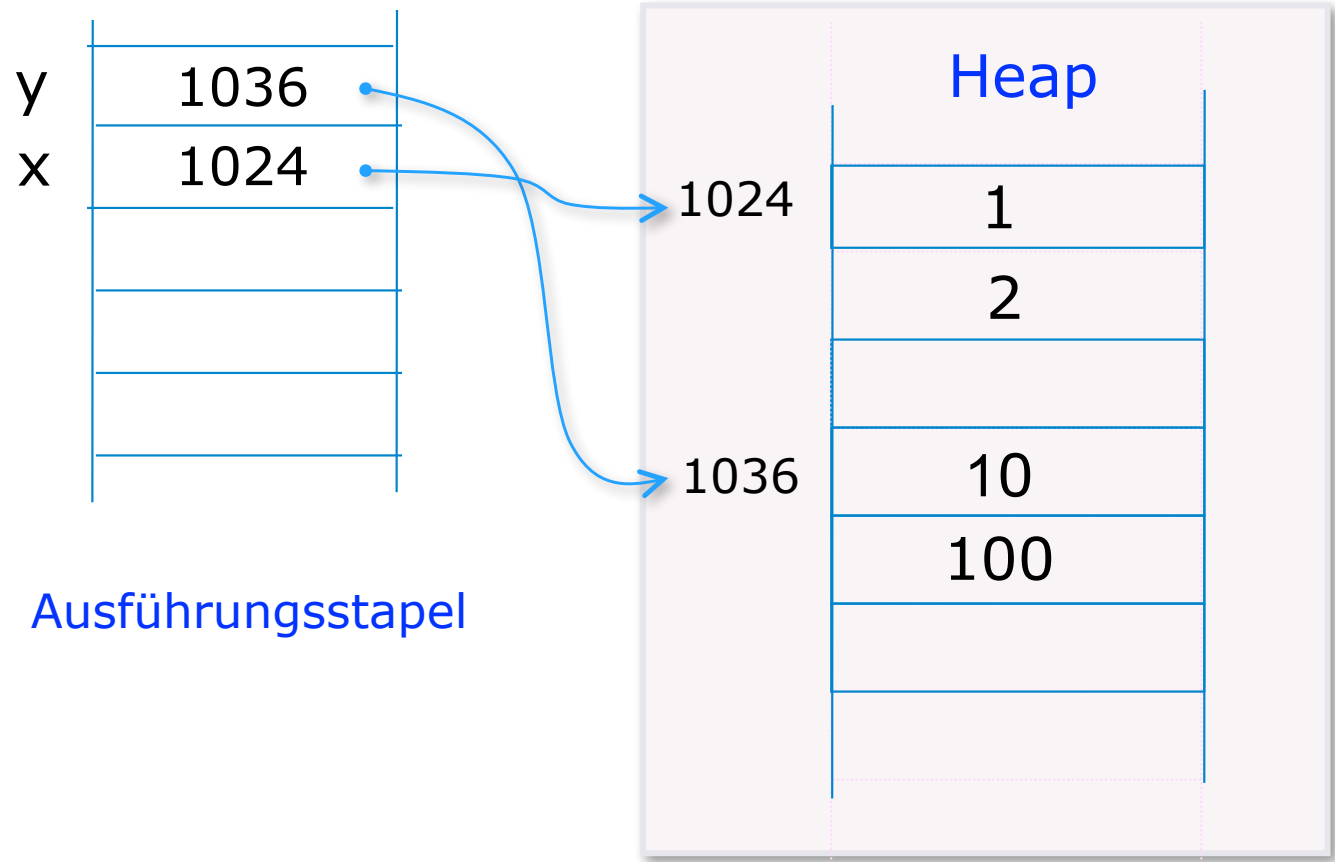
Python arbeitet nur mit Referenzen



Dynamisches Typsystem von Python

Python arbeitet nur mit Referenzen

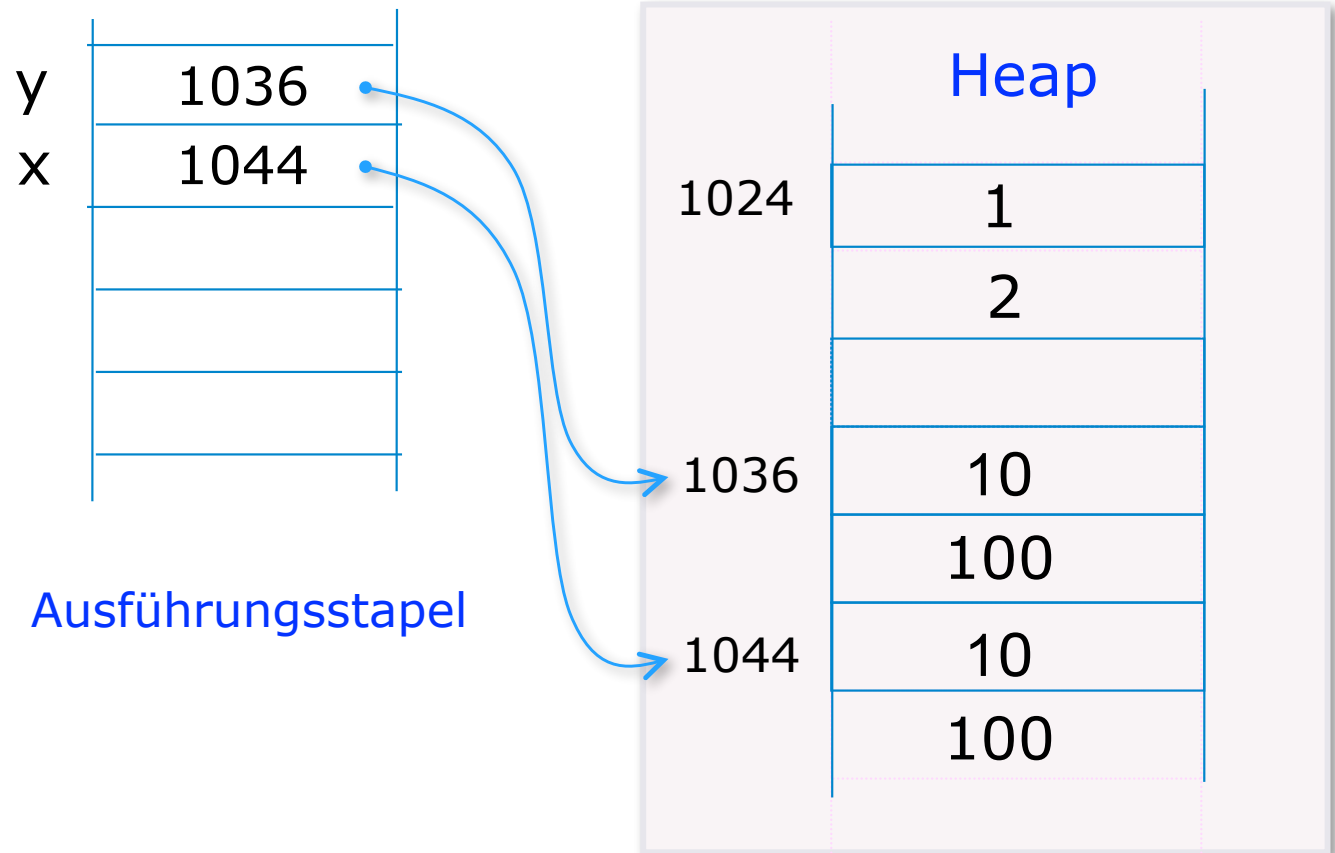
```
y = [1, 2]
x = y
y = [10, 20]
y[1] = 100
```



Dynamisches Typsystem von Python

Python arbeitet nur mit Referenzen

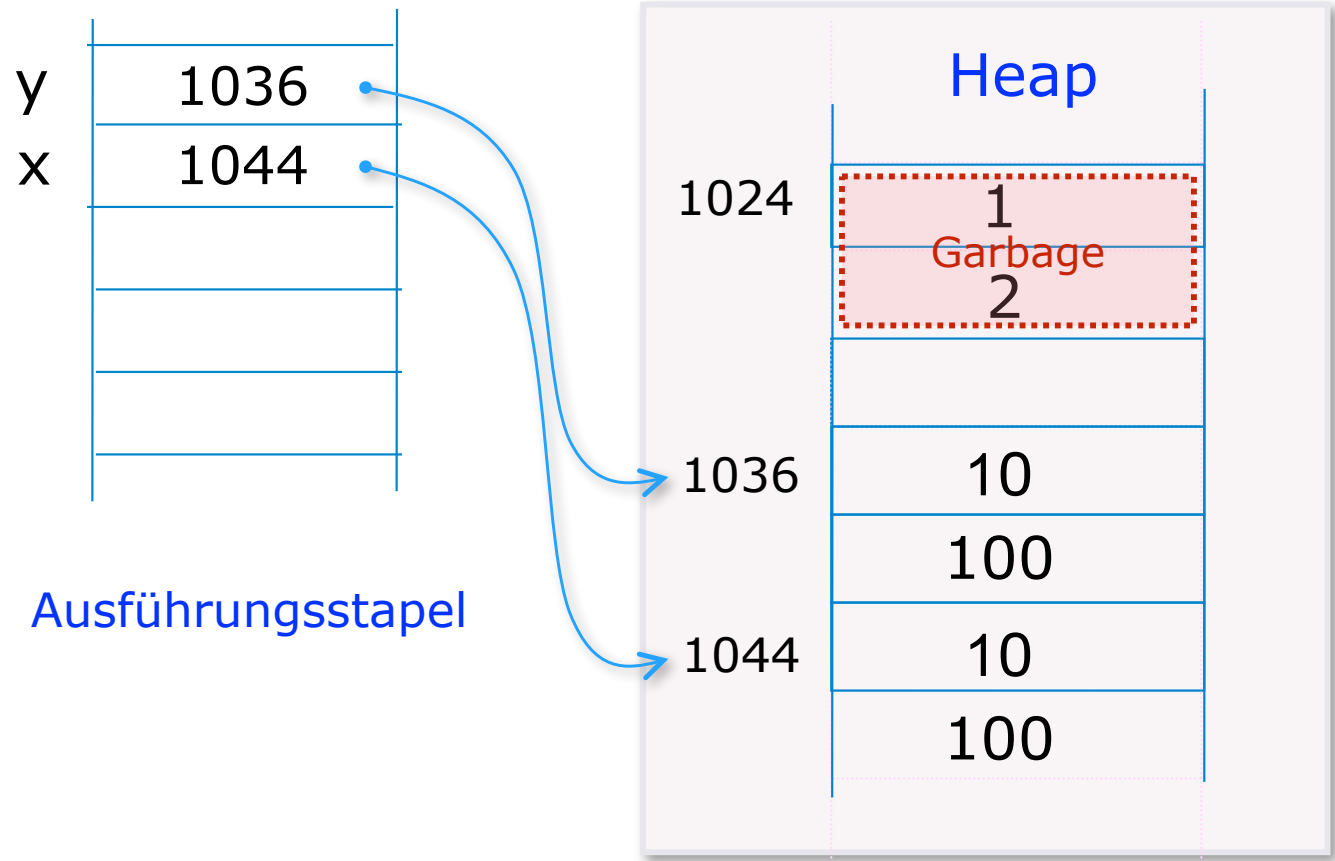
```
y = [1, 2]
x = y
y = [10, 20]
y[1] = 100
x = y[:]
```



Dynamisches Typsystem von Python

Python arbeitet nur mit Referenzen

```
y = [1, 2]
x = y
y = [10, 20]
y[1] = 100
x = y[:]
```



Grundlegende Elemente von imperativen Programmen

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

Funktionsbeispiel:

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

Funktionsaufruf:

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

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.

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

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) -----> 3 6 1 0
print(g, v, x, y) -----> 3 20 30 40

func(x, y) -----> 3 6 40 30
print(g, v, x, y) -----> 3 20 30 40
```

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

guter Programmierstil?
sinnvolle Anwendung?

Ausgabe: >>>

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.