Teilbare Zahlen

Hands-On: Schreiben Sie eine Funktion, die eine Zahl auf Teilbarkeit testet!

- Als Argumente werden die Zahl selbst und der Teiler übergeben.
- Ist die Zahl teilbar, wird “yes” ausgegeben, sonst “no”

Teilbare Zahlen

Hands-On: Schreiben Sie eine Funktion, die eine Zahl auf Teilbarkeit testet!

- Als Argumente werden die Zahl selbst und der Teiler übergeben.
- Ist die Zahl teilbar, wird “yes” ausgegeben, sonst “no”

```
>>>  def teilbar(zahl, teiler):
>>>    mod = zahl % teiler
>>>    if mod==0:
>>>      print "yes"
>>>    else:
>>>      print "no"
```

Rekursion

- Bei einer Rekursion wird eine Funktion durch sich selbst definiert.
- Beim Aufruf der Funktion wird die Funktion selbst wieder aufgerufen.
- Abbruchkriterium verhindert endlose Rekursion.

Rekursion

- Bei einer Rekursion wird eine Funktion durch sich selbst definiert.
- Beim Aufruf der Funktion wird die Funktion selbst wieder aufgerufen.
- Abbruchkriterium verhindert endlose Rekursion.

Factorial

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

Docstrings

- Ein Grund für Funktionen: Funktionalität kapseln
- Jemand, der die Funktion nicht geschrieben hat, soll sie verwenden können. ⇒ Dokumentation nötig!
- Ein String (ein- oder mehrzeilig) nach dem Header ist für interaktive Dokumentation vorgesehen.
- Python ignoriert den String (Ausdruck ohne Zuweisung) bei der Funktionsausführung.
- Das Hilfesystem (und der Code-Leser) kann den String verwenden.
- Nicht nur für Funktionen, sondern auch für Klassen, Module, ...

```
>>> def blubb():
>>>     "Unsinnige Funktion, die 'blubb' ausgibt"
>>>     print "blubb"
>>> blubb()
blubb
>>> help(blubb)
```

Variablen in Funktionen

Hands-On: Welcher Wert wird am Ende für x ausgegeben?

```
>>> def f(x):
>>>   x = x + 1
>>> x = 3
>>> f(x)
>>> x
```

Variablen in Funktionen

Hands-On: Welcher Wert wird am Ende für x ausgegeben?

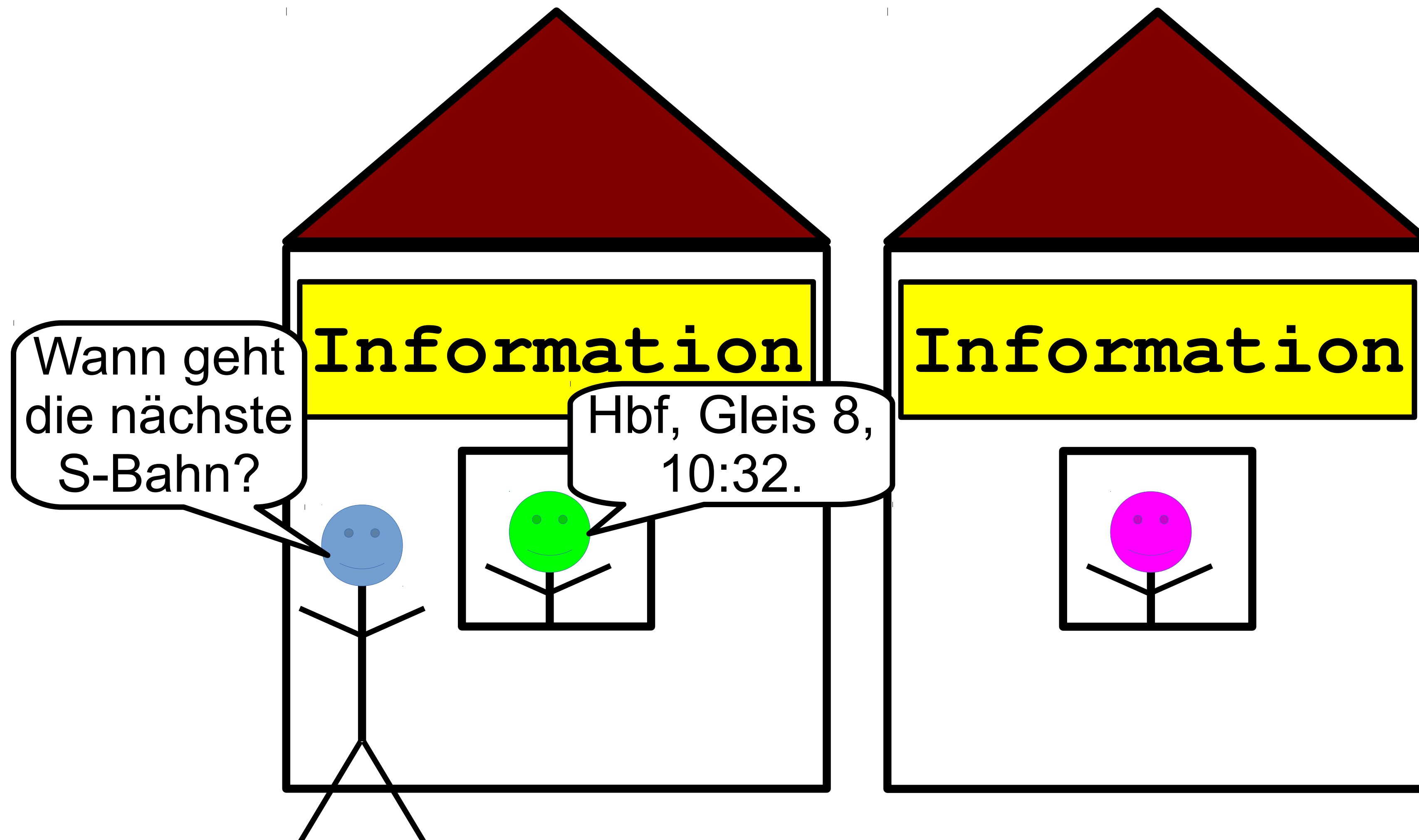
```
>>> def f(x):
>>>   x = x + 1
>>> x = 3
>>> f(x)
>>> x
```

x=3

Variablen in Funktionen: Irgendwo in München

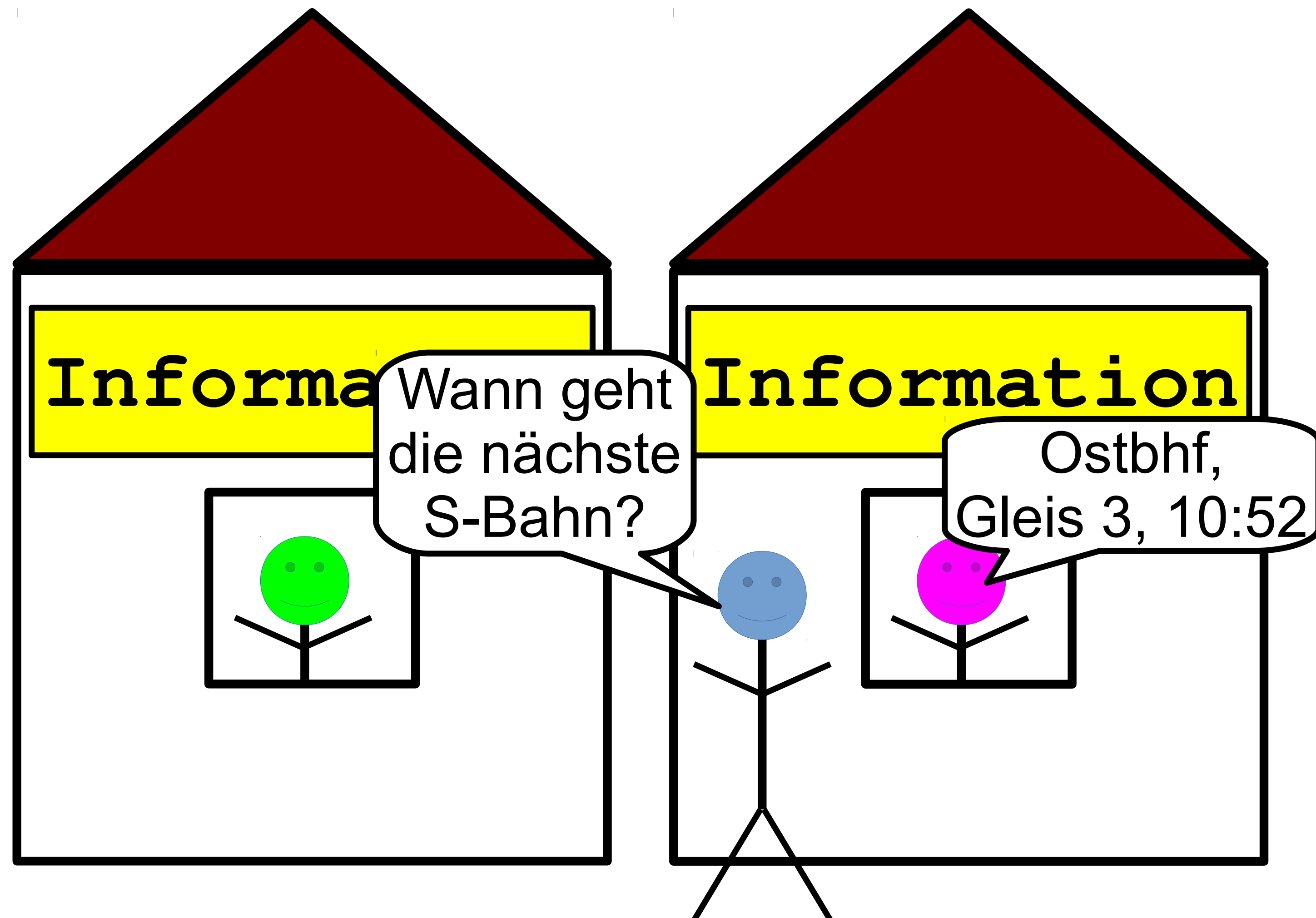
- Informationsschalter: gibt Auskunft über **einen bestimmten** Bahnhof

Variablen in Funktionen: Irgendwo in München



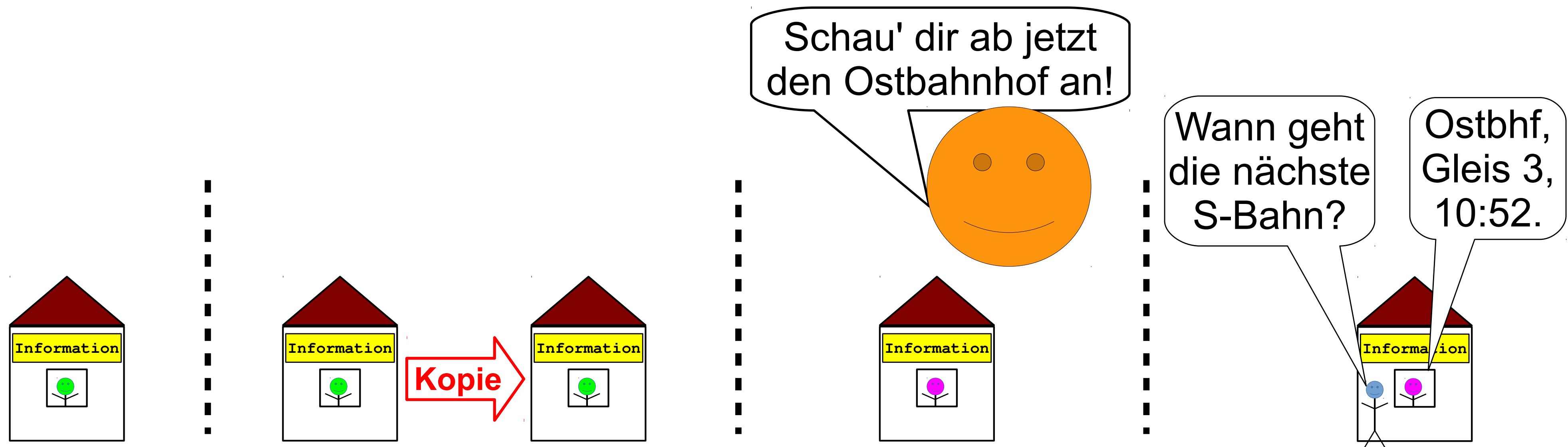
- Informationsschalter: gibt Auskunft über **einen bestimmten** Bahnhof

Variablen in Funktionen: Irgendwo in München



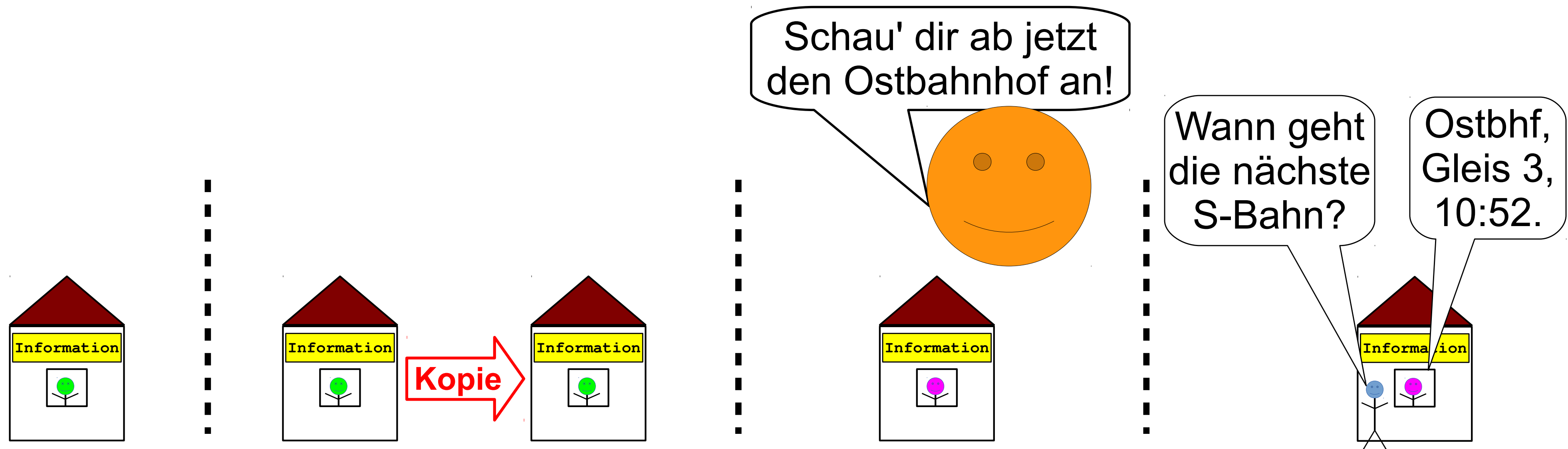
- Informationsschalter: gibt Auskunft über **einen bestimmten** Bahnhof

Variablen in Funktionen



- Bahnhöfe: Objekte
- Informationsschalter: **Referenz** auf ein Objekt Bahnhof
  $\rightarrow$ kann Information vom Bahnhof “auslesen”
  $\rightarrow$ kann auf beliebigen Bahnhof zeigen

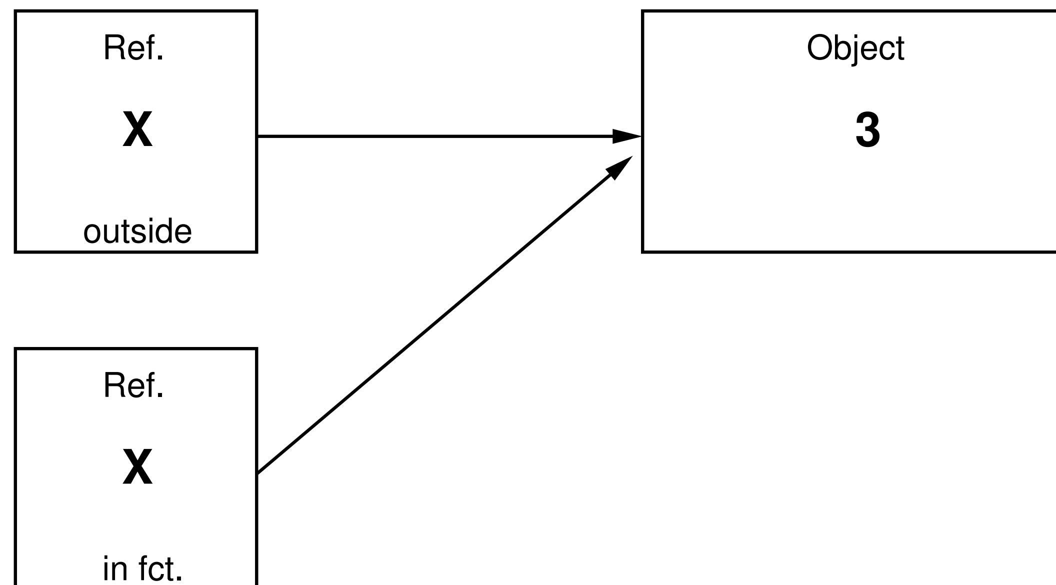
Variablen in Funktionen



- Variablen sind **Referenzen auf Objekte**
- Funktionsaufruf mit Variable
  $\rightarrow$ übergibt **Kopie** der Referenz
  $\rightarrow$ Funktion kann sowohl auf referenziertem Objekt arbeiten als auch
  $\rightarrow$ Referenz auf neues Objekt setzen
- Variable außerhalb Funktion bleibt Referenz auf urspr. Objekt

Call by Object Reference

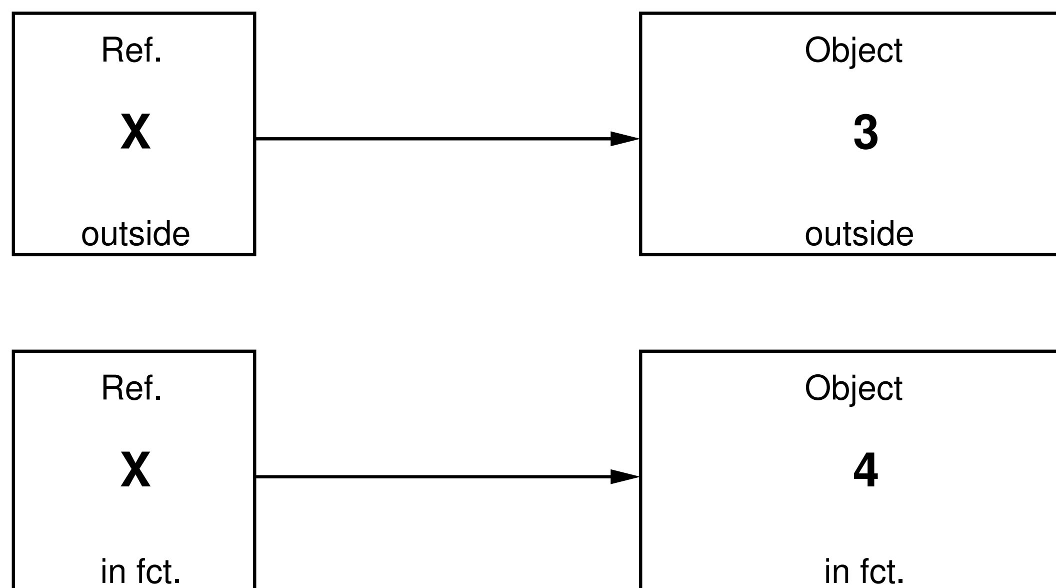
- Bei Funktionsaufruf: Referenz auf Parameterobjekt („Adresse“ , id) wird übergeben
- Referenz (id) wird kopiert (Vorteil: wenig Daten)



Call by Object Reference (2)

- ACHTUNG: Zuweisung ändert Referenzobjekt in der Funktion!

```
>>> def add_one(x):
>>>     x = x + 1
>>> x = 3
>>> add_one(x)
>>> x                #still x==3
```



Call by Object Reference (3)

- Nach der Zuweisung zeigt die lokale Variable auf ein neues Objekt.
- Mit Methoden kann dieser Effekt verhindert und das bisherige Objekt verändert werden.

```
>>> def append1(l):
>>>   l = l + [4]
>>>
>>> def append2(l):
>>>   l.append(4)
>>>
>>> l = [1,2,3]
>>> append1(l); l
[1,2,3]
>>> append2(l); l
[1,2,3,4]
```

Anonyme Funktionen

- Normale Funktionen müssen separat definiert werden.
- Sie bekommen einen Namen, mit dem auf sie zugegriffen wird.

```
>>> def inc(i):
>>>   return i+1
>>> inc(3)
4
```

- Namenlose/Anonyme Funktionen erhalten keinen Namen.
- Sie können an beliebigen Stellen in den Code eingebaut werden.

Anonyme Funktionen

- Normale Funktionen müssen separat definiert werden.
- Sie bekommen einen Namen, mit dem auf sie zugegriffen wird.

```
>>> def inc(i):
>>>   return i+1
>>> inc(3)
4
```

- Namenlose/Anonyme Funktionen erhalten keinen Namen.
- Sie können an beliebigen Stellen in den Code eingebaut werden.

```
>>> inc = lambda(i): i+1
>>> inc(3)
4
```

map

- `map` wendet eine Funktion auf alle Elemente einer Liste an
- Die Funktion kann natürlich auch anonym sein.

```
>>> L = range(10)
>>> map(lambda x: x*x+1, L)
[1, 2, 5, 10, 17, 26, 37, 50, 65, 82]
```

- In diesem Fall wäre das sogar noch kürzer mit list comprehensions (nächste Folie) gegangen.

map

- `map` wendet eine Funktion auf alle Elemente einer Liste an
- Die Funktion kann natürlich auch anonym sein.

```
>>> L = range(10)
>>> map(lambda x: x*x+1, L)
[1, 2, 5, 10, 17, 26, 37, 50, 65, 82]
```

- In diesem Fall wäre das sogar noch kürzer mit list comprehensions (nächste Folie) gegangen.

filter

- `filter` wendet ebenfalls eine Funktion auf alle Elemente einer Liste an.
- Diejenigen Elemente, für die die Funktion `True` zurückgibt, werden zurückgegeben.

```
>>> filter(lambda x: x%2==0, L)
[0, 2, 4, 6, 8]
```

List Comprehension

- Einfache Listen lassen sich z.B. mit `range` erzeugen.
- Für komplexere Listen gibt es list comprehension.

```
>>> b = [i**2 for i in range(0,10) if i%2==0]
>>> b
[0, 4, 16, 36, 64]
```

List Comprehension

- Einfache Listen lassen sich z.B. mit `range` erzeugen.
- Für komplexere Listen gibt es list comprehension.

```
>>> b = [i**2 for i in range(0,10) if i%2==0]
>>> b
[0, 4, 16, 36, 64]
```

- list comprehensions können geschachtelte Schleifen nachbilden:

```
>>> c = [(i,j) for i in range(1,5) if i%2==0
               for j in range(1,5) if j%2!=0]
>>> c
[(2, 1), (2, 3), (4, 1), (4, 3)]
```

List Comprehension (2)

- Generelle Syntax:

```
[expression for item1 in iterable1 if condition1
            for item2 in iterable2 if condition2
            ...
            for itemN in iterableN if conditionN]
```

- Falls keine Bedingung angegeben: automatisch `True`

List Comprehension (2)

- Generelle Syntax:

```
[expression for item1 in iterable1 if condition1
            for item2 in iterable2 if condition2
            ...
            for itemN in iterableN if conditionN]
```

- Falls keine Bedingung angegeben: automatisch `True`
- Syntax entspricht:

```
s = []
for item1 in iterable1:
  if condition1:
    for item2 in iterable2:
      if condition2:
        ....
        for itemN in iterableN:
          if conditionN: s.append(expression)
```

List Comprehension: Hands-On

Hands-On:

- Gegeben: Matrix $A \in \mathbb{R}^{N \times N}$.
- Format: Die Matrix soll als doppelte Liste gegeben sein: “Liste von Listen” (je eine Liste pro Zeile).
- Ziel:
  Verwenden Sie List Comprehension um eine Liste aller Matrixeinträge A_{ij} mit Indizes $i + j = N - 1$ zurückzugegeben (Indizes fangen bei (0,0) an).

Testmatrix:

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

List Comprehension: Hands-On

Hands-On:

- Gegeben: Matrix $A \in \mathbb{R}^{N \times N}$.
- Format: Die Matrix soll als doppelte Liste gegeben sein: “Liste von Listen” (je eine Liste pro Zeile).
- Ziel:
  Verwenden Sie List Comprehension um eine Liste aller Matrixeinträge A_{ij} mit Indizes $i + j = N - 1$ zurückzugegeben (Indizes fangen bei (0,0) an).

Testmatrix:

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

```
N = 3
A = [[1,2,3],[4,5,6],[7,8,9]]
c = [A[i][j] for i in range(0,N)
             for j in range(0,N) if (i+j==N-1)]
```

Beispiel: Numerische Integration (Quadratur)

Berechnung des Integrals

$$F_1(f, a, b) := \int_a^b f(x)\, dx$$

Beispiel: Numerische Integration (Quadratur)

Berechnung des Integrals

$$F_1(f, a, b) := \int_a^b f(x)\, dx$$

Verschiedene numerische Verfahren; hier jetzt:

Trapezregel

- Bestimmung des Funktionswertes an beiden Rändern
- Multiplikation des Mittelwerts mit der Intervallbreite
- $F_1 \approx T := (b - a) \cdot \frac{f(a)+f(b)}{2}$
- Polynom 1. Ordnung (Fläche entspricht Trapez)

Beispiel: Numerische Integration (Quadratur)

Berechnung des Integrals

$$F_1(f, a, b) := \int_a^b f(x)\, dx$$

Verschiedene numerische Verfahren; hier jetzt:

Trapezregel

- Bestimmung des Funktionswertes an beiden Rändern
- Multiplikation des Mittelwerts mit der Intervallbreite
- $F_1 \approx T := (b-a) \cdot \frac{f(a)+f(b)}{2}$
- Polynom 1. Ordnung (Fläche entspricht Trapez)

Quadraturfehler

$$|T - F_1| \leq \frac{1}{12} \cdot \sup_{x \in [a,b]} |f''(x)| \cdot (b-a)^3$$

Summenregeln

- Bei großen Intervallen wird auch der Fehler sehr groß.
- $\Rightarrow$ Unterteilung in n kleine Intervalle
- Jedes Teilintervall hat Länge $h = (b - a)/n$

Summenregeln

- Bei großen Intervallen wird auch der Fehler sehr groß.
- $\Rightarrow$ Unterteilung in n kleine Intervalle
- Jedes Teilintervall hat Länge $h = (b - a)/n$

Trapezsumme

$$TS := h \cdot \left[\frac{f(a)}{2} + \sum_{i=1}^{n-1} f(a + ih) + \frac{f(b)}{2} \right]$$

Implementierung der Trapezsumme

```
def trapezsumme(f, a, b, n):
  h = (b-a)/n
  sum = (f(a) + f(b))/2
  for i in range(1,n):
    sum = sum + f(a + h*i)
  return sum*h
```

Implementierung der Trapezsumme

```
def trapezsumme(f, a, b, n):
  h = (b-a)/n
  sum = (f(a) + f(b))/2
  for i in range(1,n):
    sum = sum + f(a + h*i)
  return sum*h
```

Funktion zum Test

- Funktion, die analytisch leicht zu integrieren ist
- $\Rightarrow$ Fehler lässt sich leicht berechnen

```
import math
def f(x):
  return math.sin(math.pi*x)
def f_int(x):
  return -math.cos(math.pi*x)/math.pi
```

Berechnung mit unterschiedlicher Genauigkeit: Fehler empirisch

- $n = 2^1 \ldots 2^6$

```
a_bsp = 0.
b_bsp = 1.
exakt = f_int(b_bsp) - f_int(a_bsp)
for i in range(1,7):
    n = 2**i
    t = trapezsumme(f, a_bsp, b_bsp, n)
    print '%4d %-12.6g %-12.6g' % (n, t, exakt-t)
```

n	Näherung	Fehler
2	0.5	0.13662
4	0.603553	0.0330664
8	0.628417	0.00820234
16	0.634573	0.00204662
32	0.636108	0.000511409
64	0.636492	0.000127837

Fehler der Trapezsumme: Fehler analytisch

- In jedem Teilintervall ist der Fehler

$$\leq \frac{1}{12} \cdot \sup_{x \in [a,b]} |f''(x)| \cdot h^3$$

- Bei $n = (b - a)/h$ Intervallen ist damit der Gesamtfehler

$$|TS - F_1| \leq \frac{1}{12} \cdot \sup_{x \in [a,b]} |f''(x)| \cdot (b - a) \cdot h^2$$

- Verdopplung des Rechenaufwands (Halbierung von h) reduziert den maximalen Fehler um den Faktor 4