Workshop Python

1. **Allgemeines**
2. **Syntax**
3. **Typsystem**

Teil 1

Allgemeines

Python heute:

Seit 2001 ist die **Python Software Foundation (PSF)** Herausgeber, Maintainer sowie Rechteinhaber (Non-Profit Organisation mit Sitz in Delaware, USA; ca. 200.000$ Jahresbudget, zahlreiche Industriesponsoren)

Python wird unter der **PSF-Lizenz** vertrieben (kompatibel zur GNU GPL)

Präsident der PSF: Guido van Rossum

Officers & Board of Directors: Derzeit 18 Personen mit Industrie- und akademischem Hintergrund, die sich um Python besonders verdient gemacht haben. Zum größten Teil ehrenamtliche Positionen.

Die PSF organisiert regelmäßig die **PyCon**, inzwischen auch regionale PyCons (2011: PyCon.DE in Leipzig)

Weltweit mehr als 100 **Python User Groups** (davon 6 in Deutschland)

Wo kommt Python zum Einsatz?

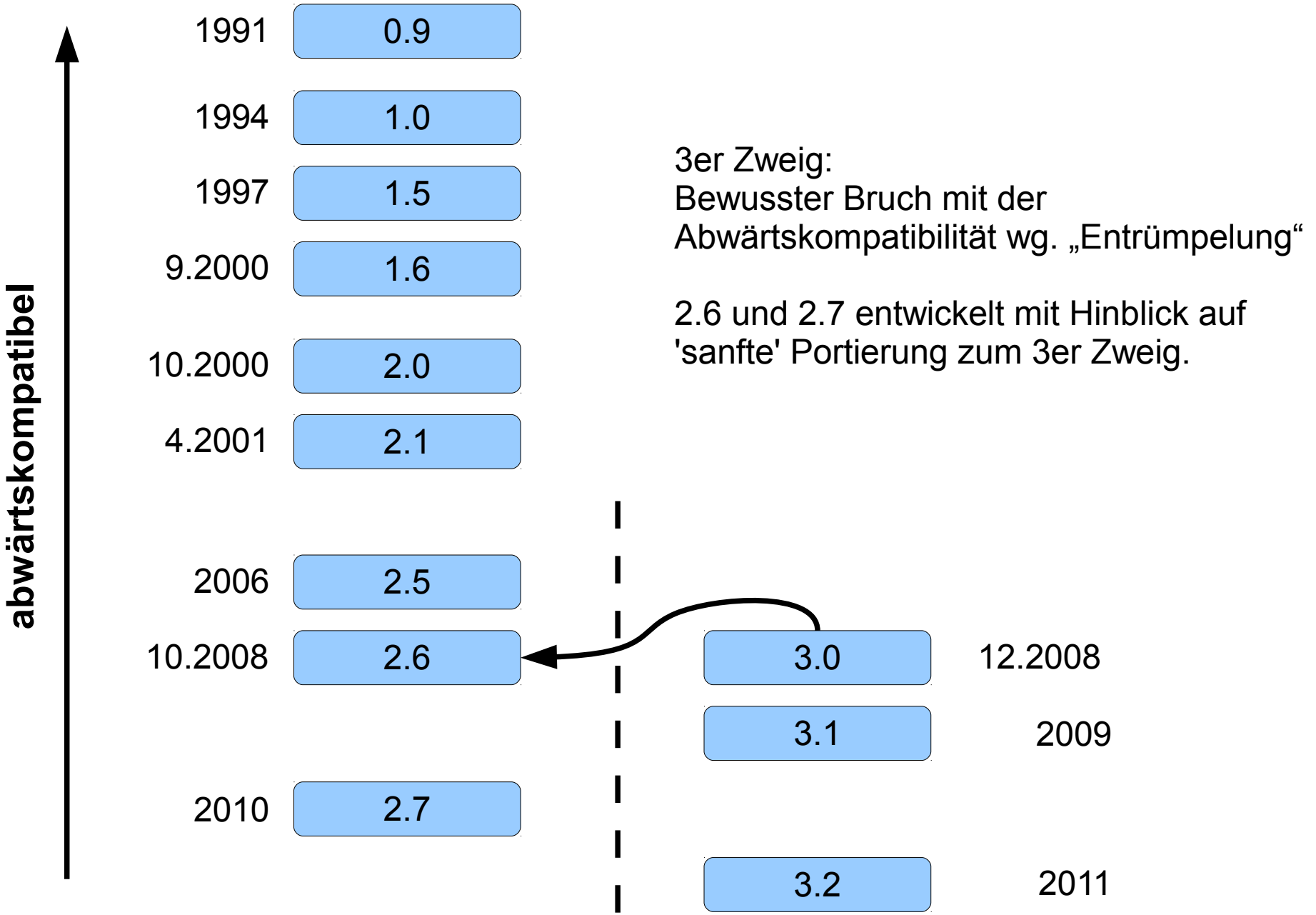
- *Google* (derzeitiger Arbeitgeber von Guido van Rossum): Websuchmaschine
- *YouTube*: Movie sharing service
- *EVE Online:* Science-Fiction Massively Multiplayer Online Game
- *Intel, Cisco, HP, Seagate, IBM*: Hardware Testing
- *Industrial Light&Magic, Pixar*: Movie animation ('Findet Nemo')
- *JPMorgan, UBS*: Finanzalgorithmen
- *NASA, Los Alamos, Fermilab*: Wissenschaftliches Programmieren
- *Maya* sowie *Blender*: Scripting API in Python (3D modelling/animation software)
- *One Laptop Per Child*: User Interface / Activity Model

http://www.python.org/about/success/

Wofür ist Python besonders geeignet?

- Systemprogrammierung / Systemadministration (z.B. Fedora / Red Hat *anaconda*)
- Rapid Prototyping
- Internet Scripting (Apache *mod_python* etc.)
- GUI (viele Linux Tools wie *update-manager, gourmet, calibre* ...)
- Scripting extensions (GIMP, Blender etc.)
- Spiele und 3D
- Gluing Language / Component Integration

http://www.python.org/about/apps/



3er Zweig:
Bewusster Bruch mit der Abwärtskompatibilität wg. „Entrümpelung“

2.6 und 2.7 entwickelt mit Hinblick auf 'sanfte' Portierung zum 3er Zweig.

Python 2.7

- Ende des 2-er Zweigs, keine weitere Entwicklung
- Extended Support
- De-facto Status Quo

Python 3.x

- Offizieller aktueller Zweig
- Einige Libraries fehlen noch
- Umwandlungstool 2to3

Inkompatible Neuigkeiten in Python 3:

- *print* wurde von einem built-in zu einer Funktion *print()*
- Konsequentere Verwendungen von *Views* und *Iteratoren* statt Listen
- Größere Änderungen im String-Handling (Strings sind default Unicode)
- Diverse kleinere Änderungen in der Syntax

Näheres siehe: http://docs.python.org/py3k/whatsnew/3.0.html

Workflow für die Migration von Altprogrammen nach Python 3:

• Python <= 2.5: Manuell in gültigen Python 2.6 / 2.7 Code konvertieren (meistens sowieso lauffähig)

• Mit der Option -3 diejenigen Codefragmente ausgeben lassen, die nicht mittels 2to3 nach Python 3 konvertiert werden können.

```
> python -3 old_script.py
```

• Falls -3 Probleme anzeigt, Code manuell ändern.

• Mit dem in Python 3 mitgelieferten Skript 2to3 den Code nach Python 3 konvertieren lassen

```
> 2to3 -w old_script.py
```

• Neuen Code testen.

```
> python
Python 2.6.6 (r266:84292, Dec 27 2010, 00:02:40)
[GCC 4.4.5] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>> 2 + 5
7
>>> a = [1, 4, 5]
>>> a
[1, 4, 5]
>>> help()
Welcome to Python 2.6!  This is the online help utility.
[... lot more text ...]

help> int
[int help page, exit with 'q']

help> q
>>> help(for)
```

Easter-Egg in der interaktiven Python-Shell:

```
>>> import this
```

```
>>> dir(list)    # Rufe Liste aller Objektmethoden auf
['__add__', '__class__', '__contains__', '__delattr__',
'__delitem__', '__delslice__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getitem__',
'__getslice__', '__gt__', '__hash__', '__iadd__', '__imul__',
'__init__', '__iter__', '__le__', '__len__', '__lt__',
'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__reversed__', '__rmul__', '__setattr__',
'__setitem__', '__setslice__', '__sizeof__', '__str__',
'__subclasshook__', 'append', 'count', 'extend', 'index',
'insert', 'pop', 'remove', 'reverse', 'sort']

>>> help(list)
(Help page on list opens, quit with q)


>>> help(list.reverse)
(Help page on list.reverse method opens:)
Help on method_descriptor:

reverse(...)
    L.reverse() -- reverse *IN PLACE*
```

```
> python3
Python 3.1.3 (r313:86834, Nov 28 2010, 11:28:10)
[GCC 4.4.5] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

Besonderheit in der interaktiven Shell:

Compound Statements werden mit einer Leerzeile abgeschlossen

```
>>> for i in range(5):
...     print i
...
0
1
2
3
4
```

helloworld.py

```
#!/usr/bin/env python
# -*- coding: utf8 -*-

"""
Hello World
"""

# Definition der parameterlosen Funktion "hello"
def hello():
    """
    Funktion hello():
    Ausgabe des Strings 'Hello world'
    """
    print 'Hello world'

hello()
```

```
> python helloworld.py
> chmod +x helloworld.py
> ./helloworld.py
```

Quelle: http://www-user.tu-chemnitz.de/~hot/PYTHON/

helloworld.py

```
#!/usr/bin/env python
# -*- coding: utf8 -*-

"""
Hello World V2: als Modul oder Programm nutzbar
"""

# Definition der parameterlosen Funktion "hello"
def hello():
    """
    Funktion hello():
    Ausgabe des Strings 'Hello world'
    """
    print 'Hello world'

if __name__ == '__main__':
    # die Funktion "hello" ausführen,
    # wenn der vorliegende Code als Programm gestartet
    # und nicht als Modul importiert wurde
    hello()
```

Quelle: http://www-user.tu-chemnitz.de/~hot/PYTHON/

hello.py

```
#!/usr/bin/env python
# -*- coding: utf8 -*-

# helloworld.py als Modul importieren
import helloworld

# die Funktion "hello" des Moduls "helloworld" aufrufen
helloworld.hello()

# die DOC-Strings aus helloworld.py ausgeben
print helloworld.__doc__
print helloworld.hello.__doc__
```

Nach Ausführung von hello.py lassen wir uns das Verzeichnis anzeigen. Was hat sich geändert? Funktioniert folgendes?

```
> python helloworld.pyc
```

Quelle: http://www-user.tu-chemnitz.de/~hot/PYTHON/

Der Python-Interpreter compiliert vor Ausführung Sourcecode in Python Bytecode.

Die PVM führt dann den Bytecode aus.

Der Source Code ist nicht notwendig zur Ausführung eines Python-Programmes, es genügt den Bytecode weiterzugeben.

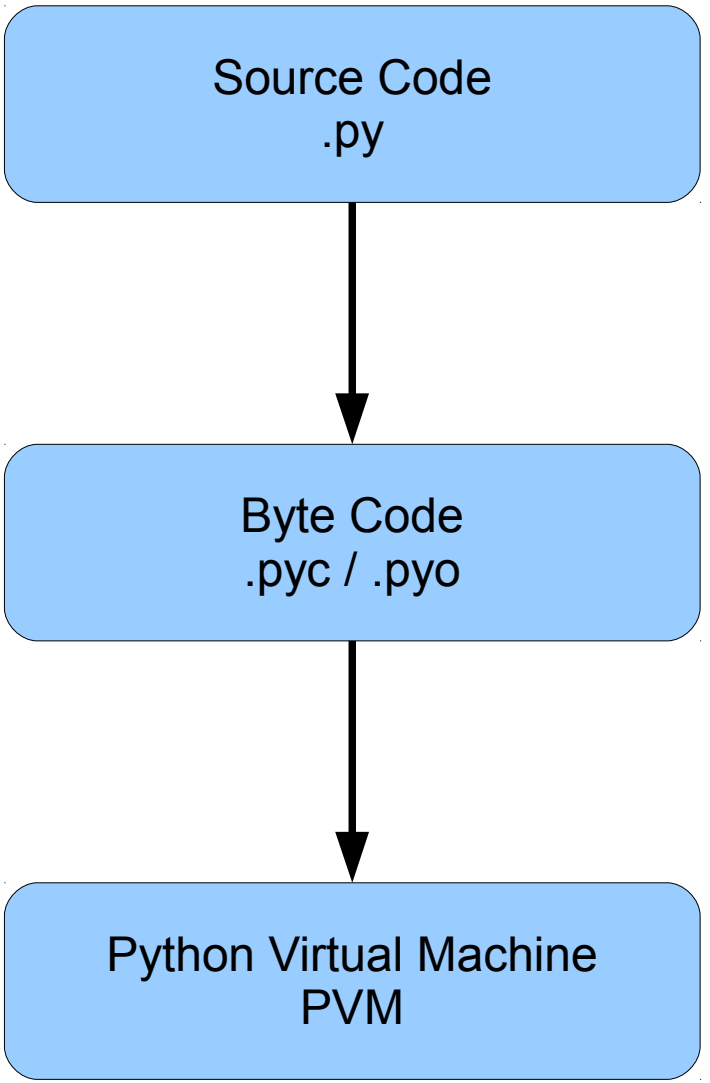
.pyc Files automatisch erzeugt beim Import eines Modules, falls der Source Code geändert wurde.

Compilierung kann mit dem Modul *compileall* erzwungen werden:

```
> python -m compileall .
```

.pyo Files sind optimierter Bytecode:

```
> python -O -m compileall .
```



Module können auch in die Python Shell importiert werden:

```
>>> import helloworld
>>> dir(helloworld)
['__builtins__', '__doc__', '__file__', '__name__',
'__package__', 'hello']
>>> help(helloworld)
Help on module helloworld:

NAME
    helloworld - Hello World V2: als Modul oder Programm
nutzbar

FILE

/home/steffen/Entwicklung/Programmierworkshop/20_Python/hell
oworld.py

FUNCTIONS
    hello()
        Funktion hello():
        Ausgabe des Strings 'Hello world'

>>> reload(helloworld)     # Module neu laden nach Änderung
```

An dieser Stelle können wir gleich mal die Funktion von *2to3* testen:

```
> python3 hello.py
> python3 helloworld.pyc
```

Was passiert?

```
> 2to3 helloworld.py hello.py
> ls
```

Was wird angezeigt?

```
> 2to3 -w helloworld.py hello.py
> ls
```

Was passiert jetzt? Was hat sich geändert?

```
> python3 hello.py
> python3 helloworld.pyc
```

Konventionen in dieser Präsentation:

Kommando im interaktiven Python-Interpreter mit Ausgabe:

```
>>> print 5 + 4
9
```

Codefragment in einem Python-Skript:

```
a = 1
b = 2
print a + b
```

Syntax:

Parameter, Objekt etc. `<object>`
Optionale Komponente: `[Optional]`

`Beispiel:` `<variable> = <expr> [# Comment]`

Achtung, Stolperfalle!

Achtung, Stolperfalle für C-Programmierer!

Unterschied Python 2 / Python 3

Vorbereitung für die Übungen

- Ein Arbeitsverzeichnis `python_workshop_01/` (oder ähnlich)

- 3 x ein Terminal-Window
    - Python 2.x interaktive Shell
    - Python 3.x interaktive Shell
    - Kommandozeile mit Directory im Arbeitsverzeichnis

- Ein Texteditor, wo wir folgendes Grundgerüst anlegen:

uebungen.py

```
#!/usr/bin/env python
# -*- coding: utf8 -*-


if __name__ == '__main__':
    pass
```

Teil 2

Syntax

Programmierparadigmen

Python ist eine sog. *Multiparadigmatische* Sprache (Hybridsprache).

Sie enthält Elemente von

a) Imperativer / Prozeduraler Programmierung

b) Objektorientierter Programmierung

c) Funktionaler Programmierung

d) Aspektorientierter Programmierung

Dieses Kapitel des Workshops behandelt hauptsächlich imperative Sprachelemente.

Objektorientierte sowie Funktionale Sprachelemente werden in späteren Kapiteln behandelt.

Aspektorientierte Sprachelemente gehören zu den fortgeschritteneren Elementen von Python; evtl. in einem weiteren Workshop

Kontrollfluss

Statements (Anweisungen) werden von oben nach unten abgearbeitet.

Control Flow Statements können den Kontrollfluss verändern
(for, while, if, function calls etc.)

```
def do_something():
    print 'Doing something!'

a = 1
b = 2

if a == 1:
    print a + b
else:
    print a - b

do_something()

for a in range(5):
    print a
```

Statements (einzeilig)

Statements enden mit dem Ende der Zeile.

```
a = 1
b = 2
print a + b
```

Mehrere Statements können auch auf einer Zeile sein, falls sie mit einem Strichpunkt (`;`) voneinander getrennt werden.

```
a = 1; b = 2; print a + b
```

C-Programmierer die frisch mit Python anfangen erkennt man am Strichpunkt nach jedem Statement.

Ist zwar nicht falsch, ist in Python aber nur für mehrere Statements auf einer Zeile notwendig.


```
# Typischer 'C-style' Python code
# Wer so in Python programmiert, erntet Spott und Hohn!

def do_something():
    print 'Doing something!';

a = 1;
b = 2;

if (a == 1):
    print a + b;
else:
    print a - b;

do_something();

for a in range(5):
    print a;
```

Statements (mehrzeilig)

Statements können über mehrere Zeilen gehen, falls die Zeile beendet wird mit:

Einem Backslash	`\`
Einer nicht geschlossenen Klammer auf der Zeile	`( [ {`
Einem nicht geschlossenen Triple-Quote String auf der Zeile	`""" '''`

```
if 1900 < year < 2100 and 1 <= month <= 12 \
   and 1 <= day <= 31 and 0 <= hour <= 24 \
   and 0 <= minute < 60 and 0 <= second < 60: # Valid date
      return True
```

```
month_names = ['Januar',    'Februar', 'März',     'April',
               'Mai',       'Juni',    'Juli',     'August',
               'September', 'Oktober', 'November', 'Dezember']
```

```
really_long_string = '''Lorem ipsum dolor sit amet,
                        consectetur adipisicing elit,
                        sed do eiusmod tempor '''
```

Blöcke

Blöcke werden durch Einrückung (*Indentation*) der Statements mit der gleichen Anzahl und Art von Whitespace (Spaces oder Tabulatoren) gekennzeichnet.

Der Header und der nachfolgende Block bilden ein *Compound Statement.* Der Block Header endet dabei mit einem Doppelpunkt (`:`)

Die gebräuchlichsten Compound Statements sind Verzweigungen (if / elif / else), Schleifen (for / while), sowie Funktionen (def) und Klassen (class)

Einfache Statements können auch auf der gleichen Zeile wie der Header auftreten.

```
a = 1; b = 2
if a == 1:
    print a + b
    do_something()
    print 'A is One!'
else: print a - b
```

Compound Statements müssen auf einer eigenen Zeile beginnen:

```
a = [1, 4, 5]; for p in a: print p  # Invalid Syntax!
```

In Python bestimmt ausschließlich die Indentation, zu welchem Block ein Statement gehört!


```
/* C nested if example */
if (a == 1)
    if (b == 1)
        do_something();
else
    do_something_else();   /* Is called if a = 1 and b != 1 */
```

```
# Python nested if example:
if a == 1:
    if b == 1:
        do_something()
else:
    do_something_else()    # Is called if a != 1
```

Kommentare

Ein Kommentar wird mit einem Hash-Zeichen (`#`) eingeleitet und geht bis zum Ende der Zeile.

```
# Das klassische "Hallo, Welt" Programm

print 'Hello, World!'          # Hallo Welt!

# Das war das "Hallo, Welt" Programm!
```

Kommentare

Es gibt in der offiziellen Python-Spezifikation keine mehrzeiligen Kommentare!

Hack (aber Guido van Rossum - approved): Mehrzeilige Triple-Quoted Strings werden ignoriert, falls sie kein Docstring sind.

```
##
#   So sieht offiziell ein langer, langer, ...
#   ... sehr langer mehrzeiliger Kommentar
#  in Python aus.
##

'''
    Das ist im Prinzip auch ein Kommentar,
    funktioniert auch, ist aber nicht ganz korrekt,
    denn er könnte als Docstring interpretiert werden
'''
```

Whitespace

Whitespace ist zur linken Seite eines Statements als Indentation relevant.
Whitespace zur rechten Seite eines Statements wird ignoriert.
Leerzeilen werden komplett ignoriert.

Ausnahme: In Strings

Documentation Strings

Falls eine Funktion, ein Modul oder eine Klasse mit einem String beginnt, wird dieser im Documentation attribute gespeichert.

Näheres siehe später, „Das Dokumentationssystem von Python“

User-Defined Names

Neu definierte Namen (Variablen, Funktionen, Klassen, Methoden etc.)
unterliegen folgenden Regeln:

Struktur

Erstes Zeichen entweder ein Buchstabe oder ein Underscore (_)
Danach beliebig Buchstaben, Zahlen oder Underscores

Erzeugung

Namen werden bei der ersten Zuweisung neu angelegt. Namen müssen bereits existieren wenn sie referenziert werden.
(Zähler z.B. müssen zu 0 initialisiert werden)

Case sensitivity

Namen sind case-sensitive!

Ungültige Zeichen

Speziell die Zeichen $ und ? werden nicht von Python verwendet
(außer in Strings und Regular Expressions)

Underscores

Namen die mit ein oder mehreren Underscores beginnen haben besondere Bedeutungen für den Python-Interpreter (kommt später -> OOP)

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

* Nur in Python 2.6/2.7 (exec, print)
\+ Nur in Python 3.x (nonlocal, False, True)

Liste der reservierten Namen im Python Hilfesystem:

```
>>> help()
help> IDENTIFIERS
```

Assignments (Zuweisungen)

Assignments weisen einem oder mehreren *Targets* eine *Referenz* zu einem *Objekt* zu.

Das Objekt wird von der *Expression* auf der rechten Seite des Statements geliefert.

Mehreren Targets können gleichzeitig das gleiche Objekt zugewiesen werden.

```
<target> = <expr>
<target1> = <target2> = <expr>
```

```
a = 5                   # a wird erzeugt,
                        # und zeigt jetzt auf eine 5

a = 6.0                 # a zeigt jetzt auf ein Float 6.0

b = a = "Python"        # Sowohl b als auch a zeigen jetzt
                        # auf einen String

c = d = [1, 2, 4]       # c und d zeigen jetzt auf ein Listenobjekt

ergebnis = berechne(a) # ergebnis zeigt auf Rückgabewert der
                       # Funktion berechne(a)
```

Augmented Assignments sind eine verkürzte Form einer häufig anzutreffenden Zuweisung.

Sie sind auch etwas schneller, da x nur einmal ausgewertet werden muss.
(x kann ein komplexes Objekt sein)

```
x = x + y                   # Klassische Form

x += y                      # Augmented Assignment
```

```
x += y; x -= y              # Addition / Subtraktion
x *= y; x /= y              # Multiplikation / Division
x **= y;                    # Potenzierung

x %= y; x //= y;            # Modulo / Floor Division

x &= y; x |= y;             # Bitweise AND / OR
x ^= y                      # Bitweise XOR
x <<= y; x >>= y;           # Bitweise Links / Rechts-Shift
```

In einem **Sequence Assignment** werden mehrere Targets auf der linken Seite auf eine Sequenz von Expressions auf der rechten Seite gematcht:

```
<trgt1>, <trgt2>, ... <trgtN> = <expr1>,<expr2>,...<exprN>
<trgt1>, <trgt2>, ... <trgtN> = <matching_length_iterable>
```

```
x, y = 1, 2                  # x ist 1, y ist 2
sum, diff = x+y, x-y

x, y = y, x                  # Swap x und y auf einer Zeile

a = [1, 2, 4]
x1, x2, x3 = a               # Weist x1-x3 Inhalt Liste a zu

x1,x2,x3,x4,x5 = range(5)    # Numeriert x1-x5 von 0-4

x1,x2 = range(6)             # Wirft einen ValueError
```

Python 3 kennt als Verallgemeinerung die **Extended Assignments:**

Genau ein Target kann mit einem Stern * ausgezeichnet werden, auf das dann alle überzähligen Elemente gematcht werden:

```
<trgt1>, *<trgt2>, ... = <expr1>,<expr2>,...<exprN>
<trgt1>, *<trgt2>, ... = <matching-length-iterable>
```

```
>>> a, *b, c = [1, 2, 3, 4, 5]
>>> a
1
>>> b
[2, 3, 4]
>>> c
5
>>> x1, *x2 = range(6)
>>> x1
0
>>> x2
[1, 2, 3, 4, 5]
```

Expressions (Ausdrücke)

Expressions liefern ein Objekt.

Ein Statement bestehend nur aus einer Expression hat nur dann einen Effekt, falls es ein Funktions- oder Objektmethodenaufruf ist.

```
<object>
<operator expr>
funktion(argument-list)
object.method(argument-list)
```

```
5

5 + a

"Python"

a or b

len(d)

m.append(2)
```

Operator	Beschreibung
X + Y, X - Y, X * Y, X / Y, X ** Y	Grundrechenarten
X % Y, X // Y	Modulo / Floor Division
- X	Negativer Wert
X < Y, X <= Y, X > Y, X >= Y	Vergleich größer / kleiner
X == Y, X != Y	Vergleich auf gleichen / ungleichen Wert
X is Y, X is not Y	Vergleich auf Identität / keine Identität
X in Y, X not in Y	Test ob X in Y enthalten / nicht enthalten ist (Iterables)
X and Y, X or Y	Logisches UND / ODER
not X	Logische Negation
X if <TEST> else Y	Ternärer Entscheidungsoperator
X & Y, X | Y, X ^ Y	Bitweises AND / OR / XOR
X << Y, X >> Y	Bitweiser Shift Links / Rechts
~ X	Bitweise Negation

print gibt Werte aus (auf den Bildschirm, in Files, etc...)

```
Python 2.x Syntax (print als built-in):

print [<value> [, <value> ...] [,] ]

print >> <file> [, <value> [, <value> ... [,] ]
```

```
print                   # Ausgabe einer Leerzeile
print 5                 # Ausgabe einer '5'
print a                 # Ausgabe des Inhalts von a
print a,                # Ausgabe des Inhalts von a, ohne Newline
print a, b,             # Ausgabe von a und b, ohne Newline

import sys
print >> sys.stderr, "Fatal error!" # Ausgabe nach stderr
```

```
Python 3.x Syntax (print als Funktion):

print ([<value> [, <value> ...]
    [, sep=<string>] [, end=<string>] [,file=<file>] )
```

```
print ()                    # Ausgabe einer Leerzeile
print (a, b, end='')        # Ausgabe von a und b, ohne Newline
print (5, 6, sep=":")       # Ausgabe von "5:6"

import sys
print ("Error!", file=sys.stderr)    # Ausgabe nach stderr
```

Weitere Formatierungsmöglichkeiten siehe Kapitel 3 -> String Formatierungen

if / elif / else führt Blöcke in Abhängigkeit von Bedingungen aus:

```
if <expr1>:               # Initial test
   <statements1>          # Conditional block
elif <expr2>:             # Optional ELSE IF
   <statements2>
elif <expr3>:             # Yet another optional ELSE IF
   <statements3>
else:                     # Optional else
   <statements4>
```

```
if a < 0:
   print "A ist negativ"
elif a % 2 == 0:
   print "A ist positiv und gerade"
else:
   print "A ist positiv und ungerade"
```

Stolperfalle 1: Testausdrücke brauchen keine Klammern. Ausdrücke in Klammern sind zwar nicht falsch, aber nicht notwendig.

```
if (a == 4):          # Das ist C-Style!
    do_something()
elif a == 5:          # So ist es richtig
    do_something_else()
```

Stolperfalle 2: Zuweisungen sind keine Ausdrücke in Python.

```
if (a = get_a())            /* Das klappt nur in C */
    do_something_with_a(a);
```

```
a = get_a()
if a:                       # So wird es in Python gemacht
    do_something_with_a(a)
```

Stolperfalle 3: C-Programmierer vergessen regelmäßig den Doppelpunkt

```
if a > 5                    # Syntax Error!
    do_something_with_a(a)
```

while iteriert einen Block, bis die Test-Bedingung nicht mehr erfüllt ist

```
while <test1>:                  # Loop test
   <statements1>                # Loop body
   if <test2>: break            # Exit loop, skip else
   if <test3>: continue         # Go to <test1> top of loop
else:                           # Optional else
   <statements2>                # Run if didnt exit with break
```

```
x = y // 2                      # Ganzzahlige Division
while x > 1:
   if y % x == 0:               # Rest (Modulo)
      print y, 'has factor', x
      break
   x -= 1
else:
   print y, 'is prime'
```

C: Assignment in Schleifentest möglich:

```
while ((x = next()) != NULL)
{
    do_something(x);
}
```


```
while True:
    x = next()
    if not x: break
    do_something(x)
```

Assignments sind *Statements*, keine Expressions!

Obiges C-Konstrukt muss anders aufgebaut werden:

```
x = next()
while x:
    do_something(x)
    x = next()
```

```
x = True
while x:
    x = next()
    if x:
        do_something(x)
```

for iteriert alle Elemente eines *Iterable* im Block durch:

```
for <target> in <iterable>:
    <statements1>
    if <test1>: break       # Exit loop, skip else
    if <test2>: continue    # Go to loop head, next <target>
else:                       # Optional else
    <statements2>           # Run if didnt exit with break
```

```
liste = [4,6,8,9,12,15,17]
for p in liste:
    if p % 2 == 0:
        continue
    if test_auf_primzahl(p):
        break
else:
    print "Liste enthält keine Primzahl"
```

```
range([start ,] stop [, step])
xrange([start ,] stop [, step])
```

Python 2:

range liefert eine Liste:

```
>>> a = range(0, 8, 2)
>>> a
[0, 2, 4, 6 ]
```

xrange liefert ein xrange-Objekt, was den nächsten Wert nur auf Anfrage generiert.

xrange ist für große Bereiche erheblich effizienter bzgl. Speicher und Geschwindigkeit und sollte bei *for*-Schleifen mit großen Bereichen verwendet werden oder falls mit einem *break* zu rechnen ist ('Lazy evaluation')

Python 3:

xrange existiert nicht mehr in Python 3. *range* liefert jetzt ein Iterator-Objekt, was in *for*-Schleifen verwendet werden kann.

Details zu Iteratoren siehe später im Kapitel "Iteratoren und Generatoren".

def definiert eine Funktion:

```
def <function-name>(arg1, arg2, ... argN):
    <function-body>
    [return <return-value>]    # Optional; Funktion ohne return
                               # gibt None zurück
```

Für Funktionsargumente können default-Werte angegeben werden:

```
def <function-name>(arg = <value>, ...):
```

Beispiel:

```
def circle_area(radius = 1.0):
    area = 3.1415 * radius * radius
    return area

print circle_area(5.0)     # Berechnet Kreisfläche r = 5.0
print circle_area()        # Berechnet Kreisfläche r = 1.0
```

Namespace Regeln für Variablen in Funktionen:

- **Referenzierung**: `x`

  Sucht nach dem Namen x in folgenden Bereichen bis x gefunden ist:

  - Lokale Funktion
  - Einschließende Funktionen
  - Globaler Scope (Modul)
  - Built-In Scope

- **Assignment**: `x = <value>`

  Erzeugt eine neue lokale Variable x, falls x noch nicht im lokalen Namespace

- **Global**: `global x; x = <value>`

  Deklariert x als globale Variable innerhalb einer Funktion;
  x wird nicht lokal angelegt.

```
x = 10
def func_1():
    print "x in func_1: ", x
def func_2():
    x = 20
    print "x in func_2: ", x
def func_3():
    global x
    x = 20
    print "x in func_3: ", x

print "Globales x: ", x                  # Ausgabe: 10
func_1()                                 # Ausgabe: 10
print "Globales x nach func_1: ", x      # Ausgabe: 10
func_2()                                 # Ausgabe: 20
print "Globales x nach func_2: ", x      # Ausgabe: 10
func_3()                                 # Ausgabe: 20
print "Globales x nach func_3: ", x      # Ausgabe: 20
```

```
pass                # Do nothing
```

Im Unterschied zu C braucht Python ein NO-OP Statement:


```
void to_be_coded_later() {} /* C: empty function body */
```

```
def to_be_coded_later():
    pass                    # placeholder for empty function body
```

```
if   a < 2: do_something()
elif a < 5: do_something_else()
elif a < 8: pass                        # necessary!
else:       do_something_completely_different()
```

```
import <module> [, <module2> [, ...] ]
```

Importiert das Module *module*, dessen Elemente dann mittels *module.attribute* aufgerufen werden können:

```
import random
import math

a = random.randrange(1,5) # Liefert Zufallszahl von 1 bis 4

b = math.cos(4.5)
```

Näheres zu Modulen siehe Kapitel 4 "Module"
sowie Kapitel 8 "Die Standard Library"

```
from <module> import <name> [, <name2> [, ...] ]
from <module> import *
```

Importiert das Element *name* aus dem Modul *module*, was dann direkt mit *name* aufgerufen werden kann.
`*` importiert sämtliche Elemente direkt:

```
from random import randrange
from math import *

a = randrange(1,5) # Liefert Zufallszahl von 1 bis 4
b = cos(4.5)
```

```
from <module> import <name> as <othername>
```

Achtung: Sollte *name* bereits definiert sein, dann wird er von *from* überschrieben!

Abhilfe: mit *as <othername>* kann ein alternativer Name zugewiesen werden.

Frage: Was ist mit Switch / *Case*?

Antwort: Gibt es in Python nicht!


```
/* C switch construct */
switch (x) {
   case 1:  do_something_a();     break;
   case 2:  do_something_b();     break;
   default: do_something_else(); break;
}
```

```
# Python 'switch'
if   x == 1: do_something_a()
elif x == 2: do_something_b()
else:        do_something_else()
```

Frage: Was ist mit *do / while* ?

Antwort: Gibt es in Python auch nicht!


```
/* C do/while loop */
do {
    do_something();
} while ( next() );
```

```
# Python 'do / while'
x = True
while x:
    do_something()
    x = next()
```

Teil 3

Typsystem

Python ist (ziemlich) **Stark typisiert:**

```
>>> a = 2
>>> b = '3'
>>> a + b
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> str(a) + b
'23'
>>> a + int(b)
5
```

Python ist **Dynamisch typisiert:**

```
a = 2             # a is now an integer.
                  # Note: No variable declaration required.
a = 'abc'         # a is now a string
a = [2, 3, 4]     # a is now a list
```

Python ist **Implizit typisiert:**

```
def calculate(a, b):
   return a*b

print calculate (2,3)          # prints „6“
print calculate ('Apples ',3) # prints „Apples Apples Apples “
```

Typ	Beispiel	Veränderbar	Sequenz	Iterierbar
None	None	Nein	Nein	Nein
Boolean	True	Nein	Nein	Nein
Integer	1768	Nein	Nein	Nein
Float	3.1415928	Nein	Nein	Nein
Complex	3.0+4.5J	Nein	Nein	Nein
String	'Monty Python'	Nein	Ja	Ja
List	[1, 'SPAM', 4.5]	Ja	Ja	Ja
Tupel	(4, 6)	Nein	Ja	Ja
Dictionary	{'Name': 'Eric', 'Job': 'Actor'}	Ja	Nein	Ja
Set / Frozenset	{ 'a', 'abc', 'd'}	Ja / Nein	Nein	Ja

None ist ein eigenständiger Typ in Python. Verwendung ähnlich NULL in C.

```
>>> type(None)
<type 'NoneType'>
```

None hat den Wahrheitswert FALSE:

```
a = None
if a: do_something()    # will not be called
```

Funktionen ohne Rückgabewert liefern *None*:

```
def return_nothing():
    return

print return_nothing() # Prints 'None'
```

Der Typ **Boolean** besteht aus den Werten *True* sowie *False:*

```
>>> type(True)
<type 'bool'>
>>> type(False)
<type 'bool'>
```

0 gilt als False, alle anderen Integer gelten als True:

```
a = True
if a:
   print 'It is true'
else:
   print 'It is false'

a1 = True if 0 else False     # -> False
a2 = True if 1 else False     # -> True
a3 = True if 2 else False     # -> True
```

Integer (Ganzzahl)

Integers bestehen aus Zahl und evtl. Vorzeichen.

Ein großes / kleines `L` kennzeichnet sog. *long integer* mit unbegrenzter Größe (bzw. so viel wie in den Speicher passt)

Ein normales Integer was über die maximale Größe ansteigt wird automatisch in ein long integer konvertiert.

```
# Python 2.6
a = 15                              # Normal integer (idR 32 bit)
b = -256                            # Normal integer, negativ
a = 15L                             # Long integer (unlimited)
c = 7922816251426433759354          # c ist automatisch long int
```

In Python 3 sind alle Integer long:

```
# Python 3.x
a = 15                            # Long integer (unlimited)
```

Integerdivision

Normale Division von Integer ist Integer in Python 2 (Old-Style), Float in Python 3. Verhalten kann mit Flag `python -Q` beeinflusst werden.

Zur Vermeidung von Bugs:

- `//` für Integer-Division verwenden
- sicherstellen, daß der Divisor bei `/` ein Float ist.

```
# Python 2.x
>>> 4 / 2
2
>>> 5 / 2
2
>>> 5 // 2
2
```

```
# Python 3.0
>>> 4 / 2
2.0
>>> 5 / 2
2.5
>>> 5 // 2       # Floor Division
2
```

Nicht-dezimale Integer werden mit einer führenden Null (0) eingeleitet.

Bei Oktalzahlen folgt	ein kleines / großes **O**
Bei Hexadezimalzahlen folgt	ein kleines / großes **X**
Bei Binärzahlen folgt	ein kleines / großes **B**

Bei Python <= 2.6 kann bei Oktalzahlen das kleine / große **O** entfallen.

Man sollte sich aber angewöhnen, Oktalzahlen immer mit **O** zu schreiben

```
# Python <= 2.6:
a = 025       # Octal         ( = 21 decimal)
```

```
# Python 2.6 / 2.7 and
# Python 3.x:
a = 0O25      # Octal         ( = 21 decimal)
a = 0X15      # Hexadecimal   ( = 21 decimal)
a = 0B010101  # Binary        ( = 21 decimal)
```

Floating point (Fließkommazahlen) werden geschrieben mit

- einem Dezimalpunkt (.)
- und / oder einem Exponenten (e / E) :

```
a = 1.23
b = -23.56
c = 3.14e-10
d = 4E210
e = 4.0E-21
```

Intern sind Floating point numbers als *C Double* implementiert.

Bei gemischten Rechnungen float / integer rechnet Python in float:

```
>>> 3 * 3.5
10.5
>>> 3.0 / 2
1.5
>>> 4 / 2.0
2.0
```

Komplexe Zahlen werden geschrieben als

Realteil+*Imaginärteil*J

wobei der Imaginärteil mit einem kleinen/großen J gekennzeichnet ist.

Der Realteil kann auch entfallen:

```
a = 3+4J
b = 3.0+4.0j
c = 3J
```

Bestandteile komplexer Zahlen sind immer Floats:

```
>>> a = 3J
>>> a / 2
1.5j
>>> a.real
0.0
>>> a.imag
1.5
```

Strings werden mit double (") oder single (') quotes begrenzt, und können Quotes der jeweils anderen Sorte ohne Escaping enthalten:

```
a = "Monty Python's"
b = 'Monty Pythons "Flying Circus"'
```

Mehrzeilige Strings werden mit entweder double oder single triple quotes begrenzt:

```
c = """This is
a multiline string"""
d = '''This is
another multiline string'''
```

Quotes der gleichen Art müssen mit Backslash gekennzeichnet werden, ebenso Escape Sequenzen:

```
e = 'Monty Python\'s Flying Circus\n'
```

Mehrere Strings die durch Whitespace getrennt sind werden zusammengefasst:

```
f = "Monty " "Python's " "Flying " "Circus"
```

Raw Strings werden mit einem **`r`** oder **`R`** eingeleitet, und ignorieren Escape Sequenzen. Nützlich für Regular Expressions sowie für DOS-Pfade:

```
a = r"Newline is indicated by \n"
```

Strings in Python 3 sind default *Unicode*, in Python 2 8-bit

Unicode-Strings werden mit einem **`u`** oder **`U`** eingeleitet,
8-bit Byte strings werden mit einem **`b`** oder **`B`** eingeleitet:

```
b = u"Das ist ein Unicode-String mit äöüß" # in Python 2.6/2.7
```

```
c = b"Das ist ein 8-bit Byte String"       # in Python 3.x
```

Format Expressions konvertieren %-Ziele in formatierte Stringelemente:

```
<string_with_%Targets> % (<s1>, <s2>, ...)
```

```
s1 = "The Knights who say %s !" % "Ni"
   # Ergibt 'The Knights who say Ni !'

s2 = "%i %s of %s" % (500, "grams", "cheese")
   # Ergibt '500 grams of cheese'

s3 = "%f %.2f %6.2f %+08.2f" % (1/3.0, 1/3.0, 1/3.0, 1/3.0)
   # Ergibt '0.333333 0.33   0.33 +0000.33'

s4 = "%(amount)i %(name)s" % {"name" : "Tomatoes",
                              "amount" : 100}
   # Ergibt '100 Tomatoes'
```

```
%[(keyname)][flags][width][.precision]typecode

<keyname>                Name der Referenz in einem Dictionary

<flags>       -          Ausrichtung auf linke Seite
              +          Zeige immer das Vorzeichen
              [space]    Bei positiven Zahlen Leerzeichen einfügen
              0          Ungenutzte Stellen mit 0 auffüllen

<width>                  Gesamte Zahl der dargestellten Stellen
<precision>              Zahl der Stellen nach .

<typecode>    s          String
              i          Integer
              x          Hexadezimal
              f          Floating-Point
              e          Floating-Point mit Exponent
              g          Floating-Point mit/ohne Exponent
```

Zugriff auf Teile eines Strings (und anderer Sequenzen) mittels Index:

```
>>> s = "ABCDEF"
```

Länge eines Strings: `len(s)`

```
>>> len(s)    # 6
```

Indexing: `s[i] # -len(s) <= i < len(s)`

```
>>> s[0]      # 'A'
>>> s[2]      # 'C'
>>> s[-1]     # 'F'
>>> s[-6]     # 'A'
```

Slicing: `s[i:j]`

```
>>> s[1:3]    # 'BC'
```

Slicing mit Stride: `s[i:j:k]`

```
>>> s[1:5:2] # 'BD'
>>> s[::2]   # 'ACE'
>>> s[::-1]  # 'FEDCBA'
```

Listen sind *veränderbare* und *indizierte* Arrays von *Referenzen* auf *beliebige* Objekte.

Listen werden mittels eckiger Klammern `[]` gebildet:

```
>>> a = [1, 2, 3]
>>> a
[1, 2, 3]
```

Auf Listenelemente kann wie bei Strings per Index zugegriffen werden:

```
>>> a[0]   # 1
>>> a[-1]  # 3
>>> a[0:2] # [1, 2]
```

Listenelemente können per Index verändert werden:

```
>>> a[2] = 4
>>> a
[1, 2, 4]
```

Listen können mehrdimensional sein:

```
>>> b = ['abc', 'cde', 'def']
>>> b[1][1]
'd'
```

Listen können beliebige Objekte enthalten:

```
>>> c = [1, 'python', [2, 3] ]
>>> c[2][1]
3
```

Leere Liste:

```
>>> d = [] # Empty list
>>> d
[]
>>> d[0]
IndexError: list index out of range
```

Operation	Beschreibung
`s[i] = x`	Element bei *i* in Liste *s* durch *x* ersetzen
`s[i:j] = t`	Elemente von i bis j in Liste s durch Iterable t ersetzen
`del s[i:j]`	Elemente von i bis j in Liste s entfernen
`s[i:j:k] = t`	Elemente i bis j durch Iterable t ersetzen, Schrittweite k
`del s[i:j:k]`	Elemente i bis j entfernen, Schrittweite k
`s.append(x)`	Fügt Element x hinten an Liste s an
`s.extend(x)`	Fügt Elemente des Iterables x einzeln hinten an Liste s an
`n = s.count(x)`	Ermittelt die Anzahl der Elemente x in Liste s
`i = s.index(x)`	Ermittelt das erste Vorkommen von x in Liste s
`s.insert(i,x)`	Fügt x an Stelle i in Liste s ein
`s.remove(x)`	Löscht das erste Vorkommen von x in Liste s
`s.reverse()`	Dreht die Liste s in-place um
`s.sort()`	Sortiert die List s in-place
`l = len(s)`	Ermittelt die Zahl aller Elemente in der Liste s
`m = min(s)` `m = max(s)`	Ermittelt das kleinste / größte Element in der Liste s

Achtung: Listenmethoden ändern die Liste *In-Place*!

Folgendes wird nicht klappen:

```
L = [ 1, 2, 3, 4]
L = L.reverse()
```

Problem: Die Methode *reverse()* ändert die Liste In-Place, und gibt als Wert *None* zurück. L ist danach *None*

Richtig geht es so:

```
L = [ 1, 2, 3, 4]
L.reverse()
```

Tupel sind ähnlich Listen *indizierte* Arrays von *Referenzen* auf *beliebige* Objekte.
Tupel sind allerdings *unveränderbar*.

```
a = ()                      # Leeres Tupel
a = (1, )                   # Tupel mit 1 Element (Komma notwendig)
a = (1, 2)                  # Tupel mit 2 Elementen
a = ([2, 3], [3, 4])        # Tupel die 2 Listen enthält

a[0]                        # gibt [2, 3]
a[0] = 2                    # wirft einen TypeError
```

Warum Tupel wenn es doch Listen gibt?

a) Tupel sind schneller und speichereffizienter.
b) Softwaredesign: (aber Python lässt einem freie Hand)
- Verwende ein *Tupel*, wenn die Daten *gemeinsam* ausgewertet / verändert werden
- Verwende eine *Liste*, wenn die Daten *einzeln* ausgewertet / verändert werden

Beispiele:

Tupel: Koordinaten (x,y)
Liste: Eckpunkte eines Polygons

```
polygon = [ (1, 1), (1, 3), (3, 3), (3, 1) ]
```

Tupel: Adresse (Straße, Hausnummer, PLZ, Ort)
Liste: Addressliste

```
adressen = [
    ["Linus", ("Millikan Way", "127", "OR 97005", "Beaverton") ],
    ["Richard", (" ...") ] ]
```

Wir öffnen den Python-Interpreter und tippen folgendes:

```
>>> a = 3
>>> b = a
>>> a = 4
>>> a
4
>>> b
?????????
```

Was wird ausgegeben? Was haben wir erwartet?

```
...
>>> b
3
```

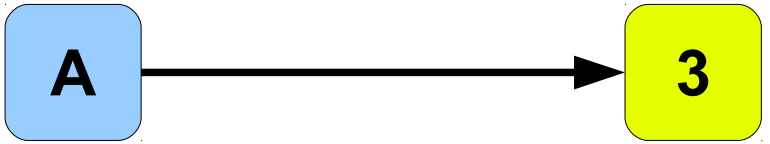
Wir tippen jetzt folgendes:


```
>>> a = [1, 2, 3]
>>> b = a
>>> a[2] = 4
>>> a
[1, 2, 4]
>>> b
?????????
```

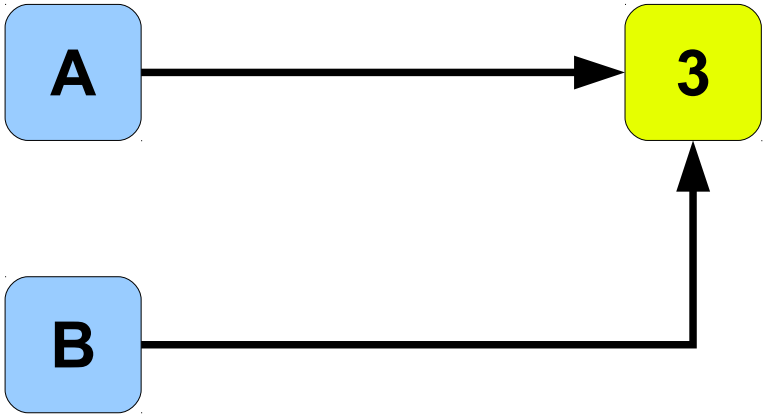
Was wird ausgegeben? Was haben wir erwartet?
Was ist der Unterschied zu vorher?

```
...
>>> b
[1, 2, 4]
```

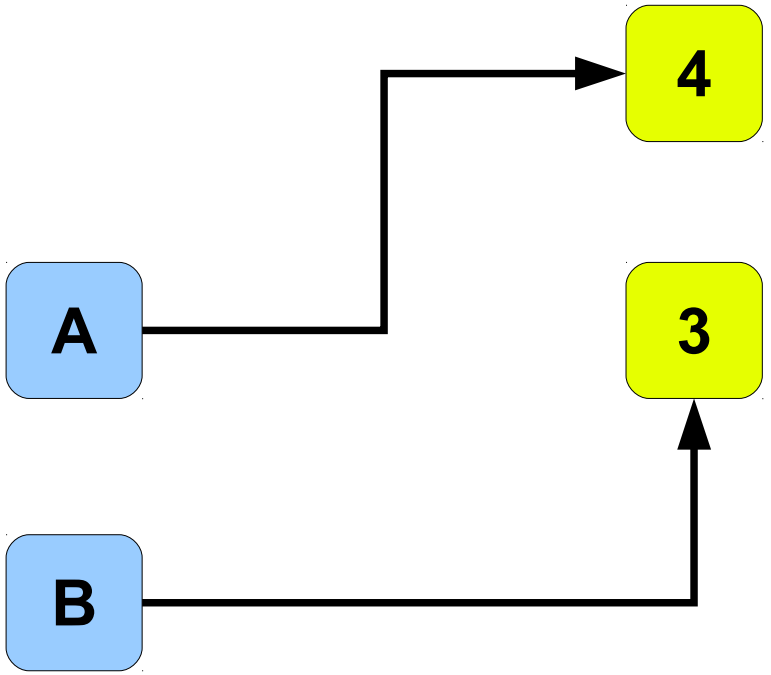
```
>>> a = 3
```



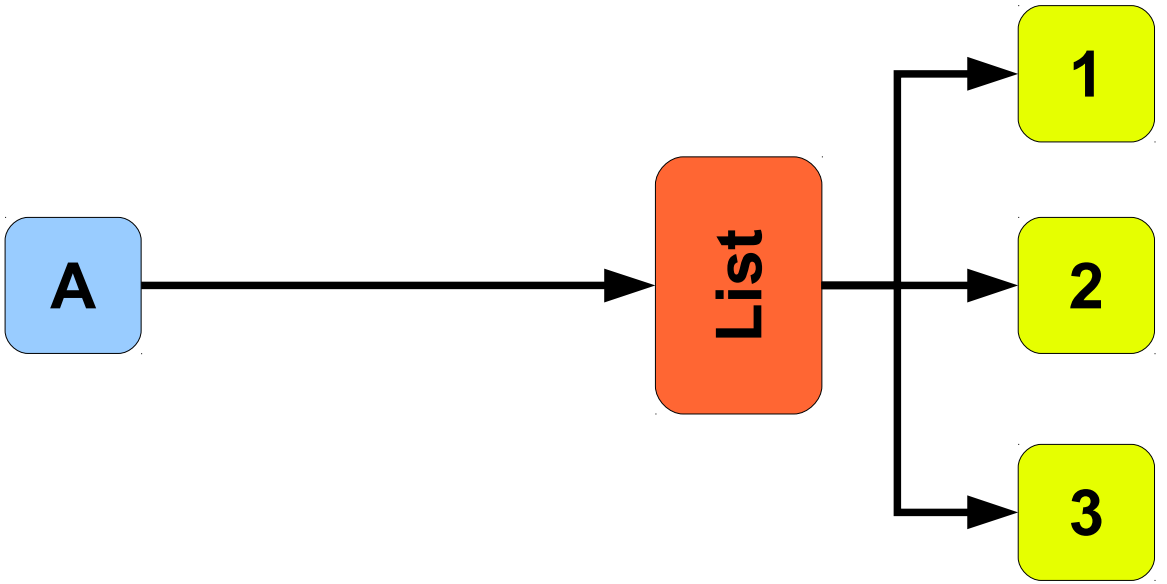
```
>>> a = 3
>>> b = a
```



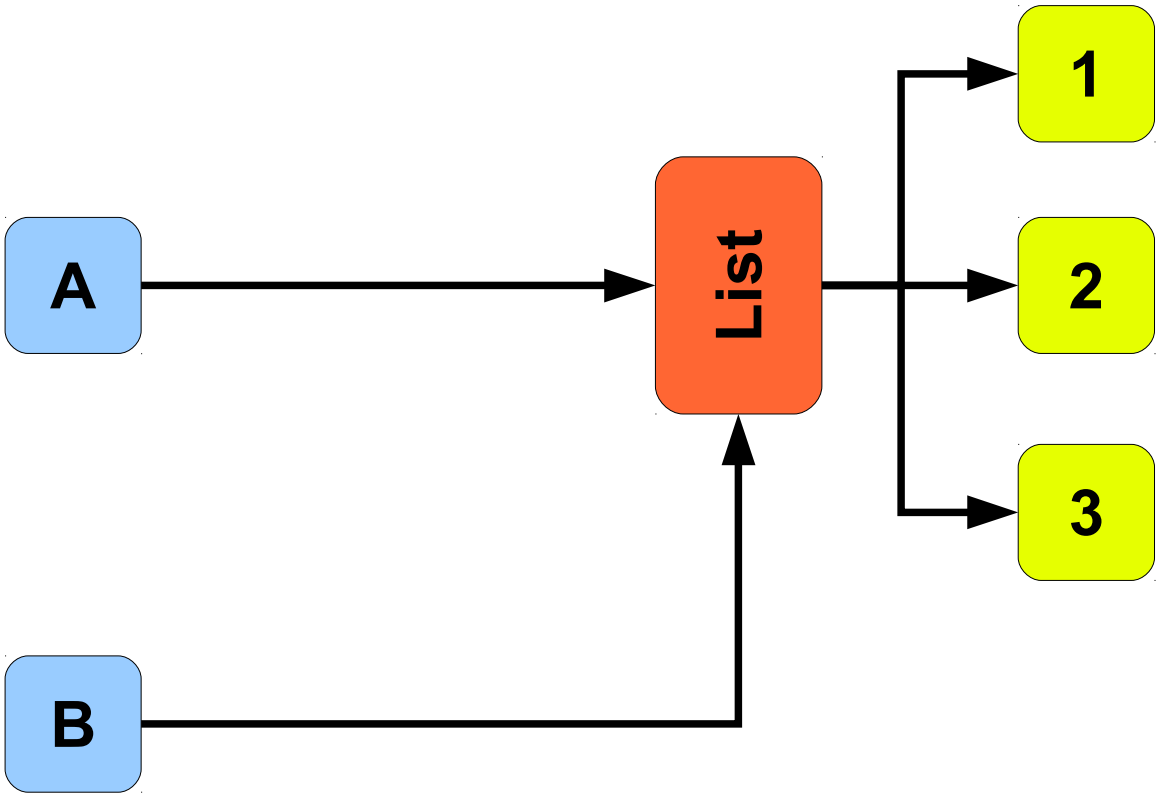
```
>>> a = 3
>>> b = a
>>> a = 4
```



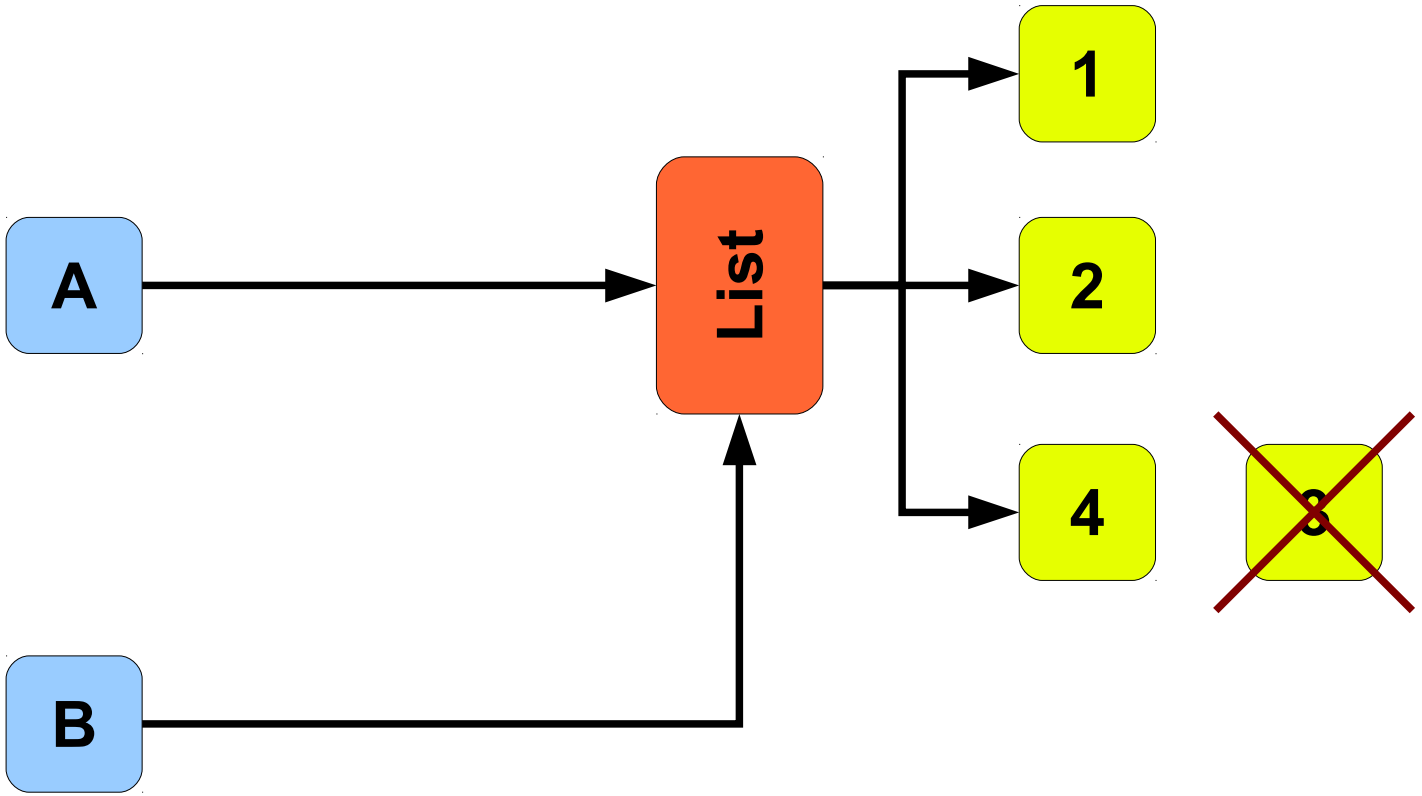
```
>>> a = [1, 2, 3]
```



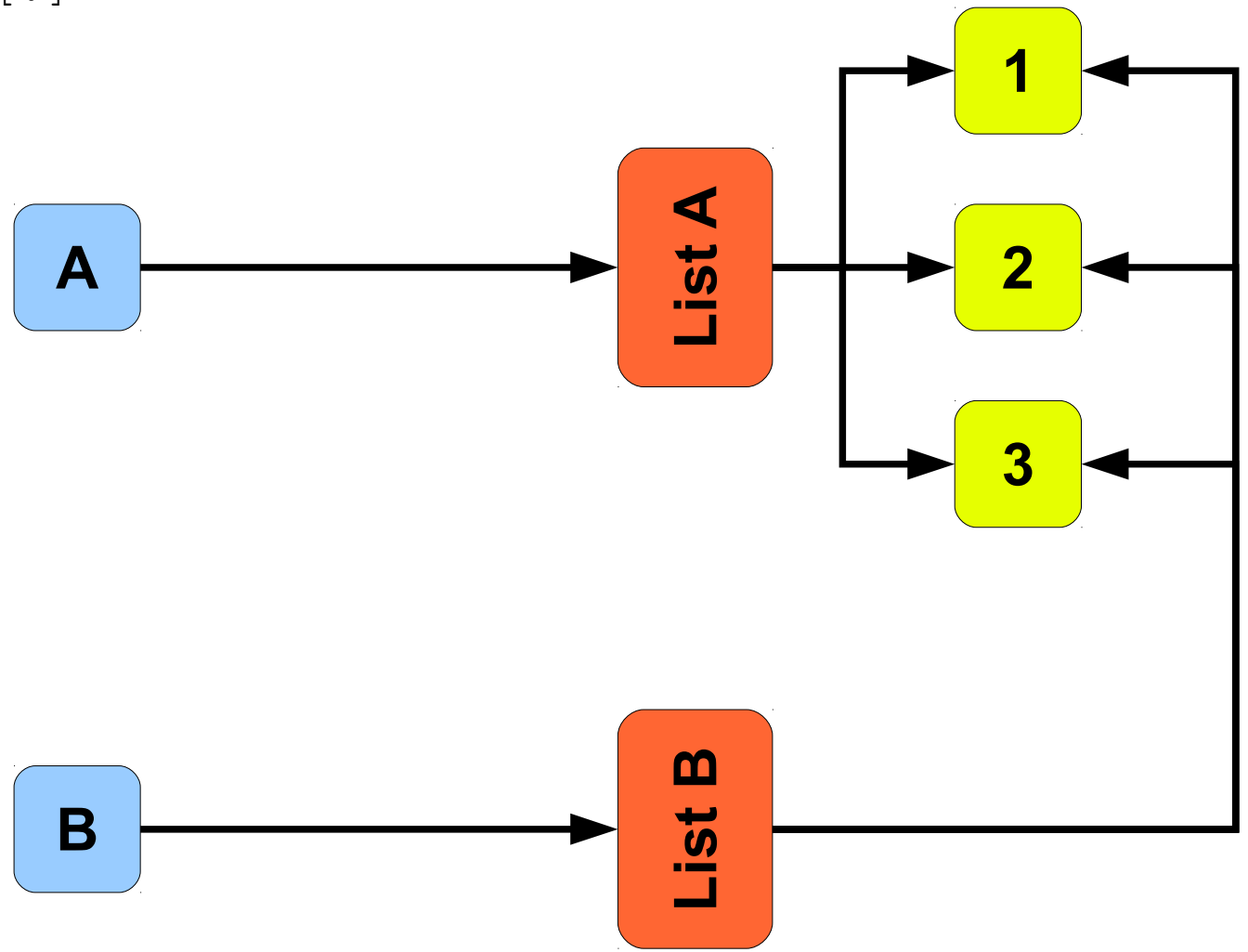
```
>>> a = [1, 2, 3]
>>> b = a
```



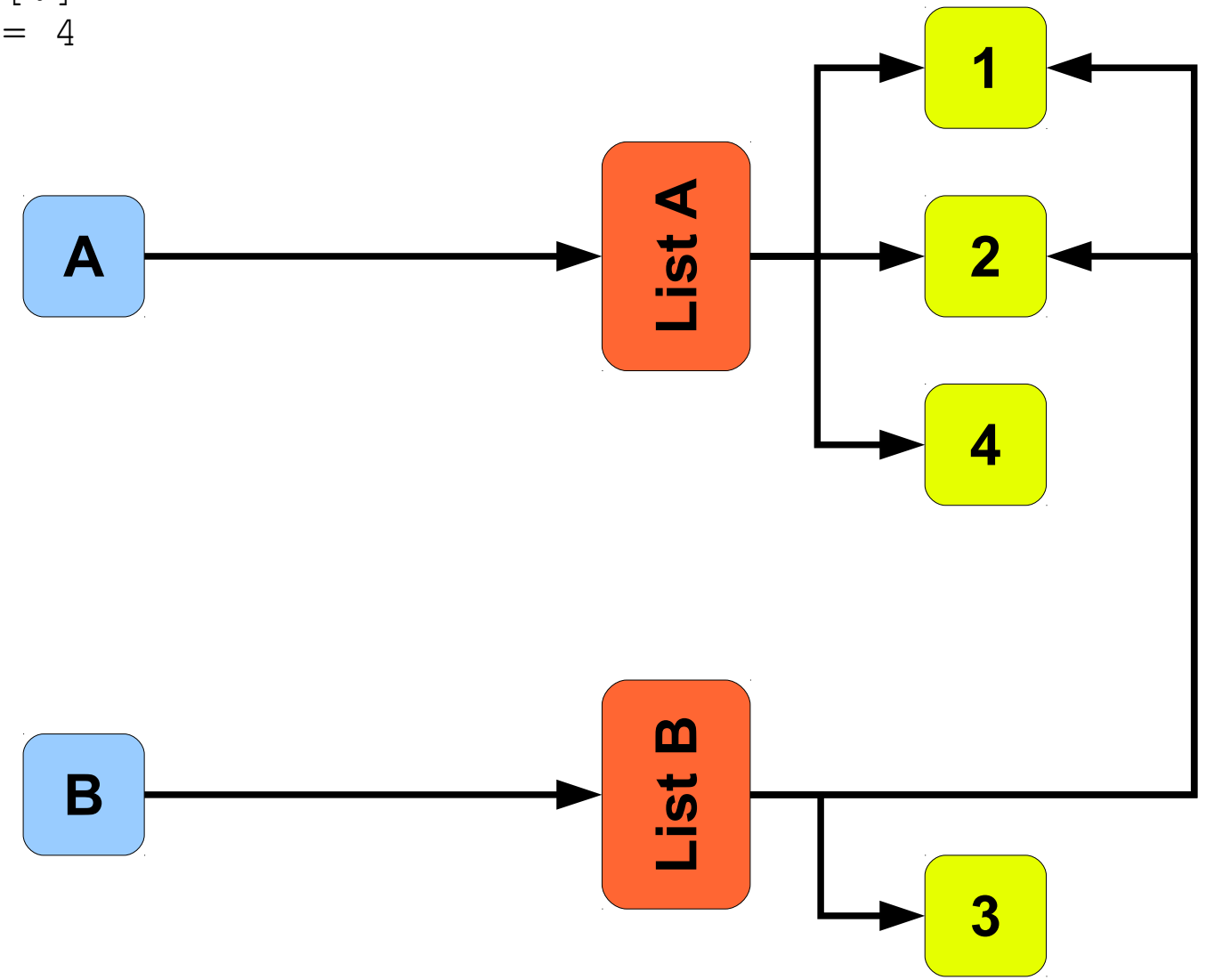
```
>>> a = [1, 2, 3]
>>> b = a
>>> a[2] = 4
```



```
>>> a = [1, 2, 3]
>>> b = a[:]
```



```
>>> a = [1, 2, 3]
>>> b = a[:]
>>> a[2] = 4
```



Die jeweilige Referenz (Speicheradresse) kann mit der Funktion id() bestimmt werden.
(Hier illustrativ; reale Speicheradressen sind i.d.R. größer)

```
>>> a = 3
>>> b = a
>>> id(a), id(b)
(100, 100)
>>> a = 4
>>> id(a), id(b)
(101, 100)
```

```
>>> a = [1, 2, 3]
>>> b = a
>>> id(a), id(b)
(1000, 1000)
>>> id(a[2]), id(b[2])
>>> (103, 103)
>>> a[2] = 4
>>> id(a[2]), id(b[2])
>>> (105, 105)
```

```
>>> a = [1, 2, 3]
>>> b = a[:]
>>> id(a), id(b)
(1000, 2000)
>>> id(a[2]), id(b[2])
>>> (103, 103)
>>> a[2] = 4
>>> id(a[2]), id(b[2])
>>> (105, 103)
```

Zur Bestimmung von *Gleichheit* und *Identität* können die Operatoren **==** sowie **is** verwendet werden:

```
>>> a = 3
>>> b = a
>>> (b == a, b is a)
(True, True)
>>> a = 4
>>> (b == a, b is a)
(False, False)
```

```
>>> a = [1, 2, 3]
>>> b = a
>>> (b == a, b is a)
(True, True)
>>> a[2] = 4
>>> (b == a, b is a)
(True, True)
```

```
>>> a = [1, 2, 3]
>>> b = a[:]
>>> (b == a, b is a)
(True, False)
>>> a[2] = 4
>>> (b == a, b is a)
(False, False)
```

List Comprehensions

Sehr häufig folgende Struktur o.ä. zum Initialisieren einer Liste:

```
# Weise L die ersten 5 Quadratzahlen zu

L = []
for x in range(5):
    L.append(x*x)
```

Verkürzte Form: *List Comprehensions* (aus der funktionalen Programmierung)

```
# Weise L die ersten 5 Quadratzahlen zu

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

Eine List Comprehension kann einen Test enthalten:

```
L = []
for x in range(10):
    if x % 2 == 0:
        L.append(x*x)
```

```
L = [x*x for x in range(10) if x % 2 == 0]
```

Iterationen über mehrere Variablen sind möglich:

```
L = []
for x in range(5):
    for y in range(5):
        L.append(x*y)
```

```
L = [x*y for x in range(5) for y in range(5)]
```

List comprehensions, allgemeine Form:

Prozedurale Form:

```
L = []
for x1 in i1:
    if <condition1>:
        for x2 in i2:
            if <condition2>:
                ...
                for xN in iN:
                    if <conditionN>:
                        L.append(<expression>)
```

Äquivalente List comprehension:

```
[<expression> for <expr1> in <iterable1> [if <condition1>]
              for <expr2> in <iterable2> [if <condition2>]
          ... for <exprN> in <iterableN> [if <conditionN>] ]
```

Comprehensions mit runden Klammern erzeugen kein Tupel, sondern einen sog. *Generator*.

Näheres siehe später, "Iteratoren und Generatoren"

```
>>> a = ( x*x for x in range(5) )
>>> a
<generator object <genexpr> at 0xb7385c34>
>>> next(a)
0
>>> next(a)
1
>>> next(a)
4
[...]
```

Dictionaries (assoziative Arrays)

Dictionaries weisen einem eindeutigen *key* einen *value* zu.

Sowohl *key* als auch *value* können dabei beliebige Objekte sein, solange *key* eindeutig hashbar ist. (Tupel sind hashbar, Listen aber **nicht**!)

```
<dict> = { key : value [, key2 : value2 , ... keyN : valueN] }
```

```
>>> a = { "Name" : "Guido van Rossum", "Beruf" : "BDFL" }
>>> a['Beruf']
'BDFL'
>>> a['Name']
'Guido van Rossum'
>>> a['Wohnort']
KeyError: 'Wohnort'

>>> a = { (1, 2) : "Tupel1" }
>>> a = { [1, 2] : "Liste1" }
TypeError: unhashable type: 'list'
```

Dictionary Methoden

```
d.keys()          # liefert Keys
d.values()        # liefert Values
d.items()         # liefert (Keys, Values)

d.has_key(key)    # Test ob key in d vorhanden (nur Python 2)
key in d          # True, falls key in d (Python 2 / 3)
d[key] = value    # Erzeugt oder Ändert den Eintrag key
del d[key]        # Löscht das Paar (key,value) aus d
d = {}            # Erzeugt ein neues, leeres Dictionary
```

Python 3.x kennt auch Dictionary Comprehensions:

```
>>> d = { x : x*x for x in range(5) }
>>> d
{ 0 : 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

2 / 3

Unterschied Python 2 / 3:

- Dictionary Methoden in Python 2 ergeben *Listen*
- Dictionary Methoden in Python 3 ergeben *View Objekte*

Näheres siehe später, "Iteratoren und Generatoren"

Python 2:

```
>>> d = { 'A' : 12, 'B' : 34, 'C' : 54 }
>>> e = d.values()
>>> e
[12, 34, 54]
```

Python 3:

```
>>> d = { 'A' : 12, 'B' : 34, 'C' : 54 }
>>> e = d.values()
>>> e
dict_values([12, 34, 54])
>>> list(e)
[12, 34, 54]
```

Sets (Mengen)

Sets sind ungeordnete, iterierbare Mengen eindeutiger Refenzen auf Objekte

```
<aset> = set(<iterable>)   # Generiere ein Set (Python 2.6)
<aset> = {<iterable>}      # Generiere ein Set (Python 2.7/3.x)

<value> in <aset>    # Test ob value in aset vorhanden

<aset>.add(x)        # Füge X hinzu
<aset>.remove(x)     # Lösche X, Error falls nicht in <aset>
<aset>.discard(x)    # Lösche X, Ignore falls nicht in <aset>
<aset>.pop()         # Lösche beliebiges Element
<aset>.clear()       # Lösche alle Elemente

<aset1> - <aset2>    # Differenzmenge
<aset1> | <aset2>    # Vereinigungsmenge
<aset1> & <aset2>    # Schnittmenge
<aset1> ^ <aset2>    # Symmetrische Differenz

<aset1> <= <aset2>   # Test auf Untermenge
<aset1> < <aset2>    # Test auf echte Untermenge
```

Type conversions

```
int(<string> [,base])   # String -> Int
float(<string | int>)   # String/Int -> Float
str(<object>)           # Object -> String (print-friendly)
repr(<object>)          # Object -> String (eval-friendly)
eval(<string>)          # Wertet String als Expression aus

ord(<char>)             # Single Char -> Char Code
chr(<int>)              # Char Code -> Single Char

hex(<int>)              # Int -> Hexadecimal als String
bin(<int>)              # Int -> Binary als String
oct(<int>)              # Int -> Octal als String

list(<iterable>)        # Iterable -> List
tuple(<iterable>)       # Iterable -> Tupel
dict(<iterable>)        # Iterable of length 2 elements -> Dict
set(<iterable>)         # Iterable -> Set
```

Vorsicht mit `eval()`!

Aufgepasst bei Verwendung von `eval()` für Auswertung nicht vertrauenswürdiger Eingabe (Internet-Scripting etc.)!

```
eval(<string>)          # Wertet String als Expression aus

eval("2+5")             # Liefert 7

# Aber folgendes funktioniert auch:
eval("__import__('os').system(rm -rf ...)")
```