

integer, float, boolean, string, bytes

```
int 783 0 -192 0b010 0o642 0xF3
      null binär oktal hexadezimal
float 9.23 0.0 -1.7e-6
bool True False
str "One\nTwo"
      escaped new line
      'I\m'
      escaped '
bytes b"toto\xfe\775"
      hexadezimal oktal
      unveränderbare
```

Grund-Typen

■ geordnete Folgen, schneller indizierter Zugriff, Werte wiederholbar

```
list [1,5,9] ["x",11,8.9] ["mot"]
tuple (1,5,9) 11,"y",7.4 ("mot",)
```

Nicht veränderbare Werte (immutables) ➔ Ausdruck nur mit Kommas ➔ tuple

■ key Container, a priori ohne Ordnung, schneller key Zugriff, jeder key ist einmalig

```
dictionary dict {"key": "value"} dict (a=3, b=4, k="v")
(key/value Verknüpfungen) {1: "one", 3: "three", 2: "two", 3.14: "pi"}
collection set {"key1", "key2"} {1, 9, 3, 0} set {}
      keys=hashbare Werte (Grundtypen, unveränderbare...) frozenset unveränderbares set empty
```

Container Typen

Für Variablen, Funktionen, Module, Klassen... Namen

a...zA...Z gefolgt von a...zA...Z_0...9

- Sonderzeichen sind zu vermeiden
- python-Schlüsselwörter sind verboten
- GROSZ-/kleinschreibung spielt eine Rolle

⊙ a toto x7 y_max BigOne

⊙ 8y and for

Identifizierer

= Variablenzuweisung

➔ Zuweisung ⇔ Binden eines Namens an einen Wert

1) Auswertung des Wertes auf der rechten Seite

2) Zuweisung zum linken Namen

```
x=1.2+8+sin(y)
a=b=c=0 Zuweisung zum selben Wert
y,z,r=9.2,-7.6,0 Mehrfache Zuweisung
a,b=b,a Wertetausch
a,*b=seq } Entpacken einer Folge in
*a,b=seq } Element und Liste
x+=3 Inkrement ⇔ x=x+3
x-=2 Dekrement ⇔ x=x-2
x=None « undefiniert » konstanter Wert
del x Entferne Name x aus dem Speicher ...
```

Konvertierungen

type (Ausdruck)

int("15") → 15

int("3f",16) → 63 ganzzahlige Zahlenbasis kann im zweiten Parameter stehen

int(15.56) → 15 schneide Nachkommastellen ab

float("-11.24e8") → -1124000000.0

round(15.56,1) → 15.6 runde auf eine Dezimale (0 Dezimalen → Ganzzahl)

bool(x) False für kein x, leeren Container x, None oder False x ; True für andere x

str(x) → "..." Zeichenkette von x für die Anzeige (Formatierung im Hintergrund)

chr(64) → '@' ord('@') → 64 code ↔ char

repr(x) → "..." Literal-Darstellung von x

```
bytes([72,9,64]) → b'H\t@'
list("abc") → ['a','b','c']
dict([(3,"three"),(1,"one")]) → {1:'one',3:'three'}
set(["one","two"]) → {'one','two'}
```

Trenner str und Folge von str → zusammengesetzter str

```
':'.join(['toto','12','pswd']) → 'toto:12:pswd'
```

str getrennt durch whitespaces → list von str

```
"words with spaces".split() → ['words','with','spaces']
```

str getrennt durch Separator str → list von str

```
"1,4,8,2".split(",") → ['1','4','8','2']
```

Folge eines Typs → list eines anderen Typs (via comprehension list)

```
[int(x) for x in ('1','29','-3')] → [1,29,-3]
```

Für Listen, Tupel, Zeichenketten, bytes...

negativer Index	-5	-4	-3	-2	-1
positiver Index	0	1	2	3	4

```
lst=[10,20,30,40,50]
```

positiver slice	0	1	2	3	4	5
negativer slice	-5	-4	-3	-2	-1	

Anzahl der Elemente

```
len(lst) → 5
```

Index beginnt bei 0 (hier von 0 bis 4)

Zugriff auf Sub-Folgen via lst[start slice:end slice:Schrittweite]

```
lst[:-1] → [10,20,30,40] lst[::-1] → [50,40,30,20,10] lst[1:3] → [20,30] lst[:3] → [10,20,30]
lst[1:-1] → [20,30,40] lst[::-2] → [50,30,10] lst[-3:-1] → [30,40] lst[3:] → [40,50]
lst[::2] → [10,30,50] lst[:] → [10,20,30,40,50] flache Kopie der Folge
```

Ohne angegebenen slice → von Anfang / bis zum Ende.

Bei veränderbaren Folgen (list): Entfernen mit del lst[3:5] und Verändern durch Zuweisung lst[1:4]=[15,25]

Indizierung von Folgen und Containern

Individueller Zugriff auf Elemente via lst[index]

```
lst[0] → 10 ⇒ erstes Element lst[1] → 20
lst[-1] → 50 ⇒ letztes Element lst[-2] → 40
```

Bei veränderbaren Folgen (list):

Entfernen mit del lst[3] und

Verändern durch Zuweisung lst[4]=25

Boolesche Logik

Komparatoren: < > <= >= == != (boolesche Resultate) ≤ ≥ = ≠

a and b Logisches Und Beide gleichzeitig

a or b Logisches Oder Eines oder beide

⊗ Falle : and und or geben einen Wert von a oder von b zurück (shortcut evaluation). ➔ stelle sicher, dass a und b boolesch sind.

not a Logisches Nicht

True } Wahr- und Falsch-Konstanten
False }

Ausdrucksblöcke

Eltern-Ausdruck :

Befehlsblock 1...

⋮

Eltern-Ausdruck :

Befehlsblock 2...

⋮

Nächster Befehl nach Block 1

⊗ stelle den Editor so ein, dass er 4 Leerzeichen anstelle eines Tabs macht.

Import von Modulen/Namen

module true ⇒ file true.py

```
from monmod import nom1,nom2 as fct
      → direkter Zugriff auf Namen, umbenennen mit as
import monmod → Zugriff über monmod.nom1 ...
      ⊗ Module und Pakete werden im python-Pfad gesucht (vgl. sys.path)
```

Befehlsblock wird nur ausgeführt wenn eine Bedingung wahr ist.

Bedingte Ausführung

if logische Bedingung:

→ Befehlsblock

Kann in Verbindung mit mehreren elif, elif... und nur einem finalen else benutzt werden. Nur der Block mit der ersten wahren Bedingung wird ausgeführt.

⊗ mit einer Variablen x:

```
if bool(x)==True: ⇔ if x:
if bool(x)==False: ⇔ if not x:
```

if age<=18: state="Kid"
elif age>65: state="Retired"
else: state="Active"

Fließkommazahlen... ungefähre Werte

Operatoren: + - * / // % **

Priorität (...)

ganzzahlige ÷ ÷ Rest

@ → Matrizen-x python3.5+numpy

```
(1+5.3)*2+12.6
abs(-3.2)+3.2
round(3.57,1) → 3.6
pow(4,3) → 64.0
```

⊗ gewöhnliche Prioritäten

Mathe

Winkel in Bogenmaß!

```
from math import sin,pi...
sin(pi/4) → 0.707...
cos(2*pi/3) → -0.4999...
sqrt(81) → 9.0
log(e**2) → 2.0
ceil(12.5) → 13
floor(12.5) → 12
```

Module math, statistics, random, decimal, fractions, numpy etc. (vgl. doc)

Ausnahmen bei Fehlern

Einen Fehler signalisieren:

```
raise AusnKlasse(...)
```

Fehlerbehandlung :

```
try:
    → Normaler Ausführungsblock
except AusnKlasse as e:
    → Fehlerbehandlungsblock
```

⊗ finally startet den Block für die Endverarbeitung in allen Fällen.

