

# Python-Cheatsheet

## Variablen

- Platzhalter (Behälter) für einzelne Werte
- müssen immer einen Wert haben (Zuweisung mit `=` z.B. `a = 1`)
- der Datentyp richtet sich nach dem Wert der Variable
- Verändern von Variablen: `a += 1`: Kurzform von `a = a + 1` (analog: `-=`, `*=`, `/=`)
- Bezeichner (Variablennamen): klein schreiben
- Ein- und Ausgabe über die Konsole mit Hilfe der Funktionen
  - `input()` erfragt eine Konsoleingabe und liest sie (`String`)
  - `print(..., end=" ")`, Default für `end`: Zeilenumbruch

## Datentypen

Datentyp	Bedeutung	Variabeldefinition	<code>type(x)</code>
Integer	Ganzzahl	<code>a = 5</code>	<code>&gt; int</code>
Float	Fliesskommazahl	<code>a = 5.0</code>	<code>&gt; float</code>
String	Zeichenkette	<code>a = "Hallo!"</code>	<code>&gt; str</code>
Boolean	Wahrheitswert	<code>a = True</code>	<code>&gt; bool</code>

## Arithmetische Operatoren

<code>+</code>	Addition	<code>a + b</code>	<code>&gt; 7</code>
<code>-</code>	Subtraktion	<code>b - a</code>	<code>&gt; 3</code>
<code>*</code>	Multiplikation	<code>a * b</code>	<code>&gt; 10</code>
<code>/</code>	Division	<code>a / b</code>	<code>&gt; 0.4</code>
<code>//</code>	Ganzzahlige Division	<code>a // b</code>	<code>&gt; 0</code>
<code>%</code>	Modulo (Rest der Division)	<code>a % b</code>	<code>&gt; 2</code>
<code>**</code>	Potenz	<code>a**b</code>	<code>&gt; 32</code>

## Funktionen

- Erstellen (definieren) **eigener Befehle** durch Zusammenfassen eines Codeblocks in einer Funktion
- Ermöglichen eine bessere Lesbarkeit, Wartbarkeit sowie Strukturierung des Codes
- Funktionen können Parameter entgegennehmen, müssen aber nicht. Die Parameter können auch Standardwerte haben.
- Rückgabewert: die Rückgabe ist optional, in Python wird aber immer etwas zurückgegeben: (Default (Standardwert): `None`) In vielen Sprachen werden Funktionen ohne Rückgabe Prozeduren genannt
- Definition:
 

```
def funktionsname(optionale, parameter):
    # Funktionskörper: Code
    return rückgabewert
```
- Beispiel: Funktion ohne Parameter
 

```
def sag_hallo():
    print("Hallo!")
```
- Beispiel: Funktion mit Parameter
 

```
def sag_hallo(name):
    print("Hallo", name)
```
- Beispiel: Parameter mit Standardwert
 

```
def sag_hallo(name, gruss="Hallo"):
    print(gruss, name)
```
- Aufruf: `sag_hallo("Ben", "Grüezi")` → Grüezi Ben
- Aufruf: `sag_hallo("Ben")` → Hallo Ben
- Beispiel: Funktion mit Rückgabewert
 

```
def addiere(a, b):
    return a + b
```

## Logik

### Logische Ausdrücke

- Gehen einen Boolean zurück (`True`, `False`).
- Lieber zu viele Klammern als zu wenig.

### Logische Operatoren

Logische Operatoren dienen der Negation bzw. der Verknüpfung von Bedingungen.

`A = True, B = False`

and	logisches und (AND)	<code>A and B</code>	<code>&gt; False</code>
or	logisches oder (OR)	<code>A or B</code>	<code>&gt; True</code>
not	Negation (NOT)	<code>not A</code>	<code>&gt; False</code>

### Relationale Operatoren

Relationale Operatoren dienen dem Vergleich von Werten.

`a = 2, b = 5`

<code>&lt;</code>	kleiner als	<code>a &lt; b</code>	<code>&gt; True</code>
<code>&lt;=</code>	kleiner gleich	<code>a &lt;= b</code>	<code>&gt; True</code>
<code>&gt;</code>	größer als	<code>a &gt; b</code>	<code>&gt; False</code>
<code>&gt;=</code>	größer gleich	<code>a &gt;= b</code>	<code>&gt; False</code>
<code>==</code>	gleich	<code>a == b</code>	<code>&gt; False</code>
<code>!=</code>	ungleich	<code>a != b</code>	<code>&gt; True</code>

## Kontrollstrukturen

### Verzweigungen

- einseitig: `if`  
 - zweiseitig: `if - else`  
 - mehrstufig: `if - elif - ... - elif - else`  
 - Nach `if` und `elif` erfolgt eine **Bedingungsprüfung**. Sie gibt einen Boolean zurück und enthält logische oder relationale Operatoren.  
 - Beispiel:
 

```
if heute in ["Mo", "Di", "Mi", "Do"] and not ferien:
    aufstehen(6.00)
elif heute == "Fr" and not ferien:
    aufstehen(9.00)
else:
    ausschlafen()
```



### Schleifen

- Mit Hilfe von Schleifen werden Codeblöcke so oft ausgeführt wie nötig.  
 - **kopfgesteuert: while-Schleife** wird solange ausgeführt wie eine Bedingung erfüllt ist.
 

```
while Bedingung: # Schleifenkopf
    # Schleifenkörper (Codeblock)
```

- Allfällige Zähler müssen in der Schleife explizit angepasst werden.  
 - **zählergesteuert: for-Schleife** wird ausgeführt, während eine Laufvariable (Zähler) einen Bereich durchläuft
 

```
for laufvariable in range(start, stop, step):
    # Schleifenkörper (Codeblock)
```

- Die Laufvariable (Zähler) wird bei jedem Durchgang um die Schrittweite `step` verändert. Diese ist optional, muss also nicht angegeben werden (Standardwert 1).

## Schleifen – Fortsetzung

- Fussgesteuerte Schleifen gibt es in Python **nicht!** (sie würden solange ausgeführt wie eine Bedingung *nicht* erfüllt ist)

- **Abbruch: Blöcke** können (generell) jederzeit mit `break` verlassen werden. Das Programm geht in diesem Fall nach dem Block weiter.

```
for i in range(10):
    if i > 5:
        break
    print(i) # → Zahlen von 0 bis und mit 5
```

- Beispiel (Summe aller geraden Zahlen von 10 bis 20):

```
mit kopfgesteuerter Schleife
summe = 0
summand = 10
while summand <= 20:
    summe += summand # summe = summe + summand
    summand += 2 # summand = summand + 2
print(summe) # → 90
```

- mit zählergesteuert er Schleife

```
summe = 0
for i in range(10, 21, 2):
    summe = summe + i
print(summe) # → 90
```

- **Verschachtelte Schleifen:**

- innere Schleife wird für jeden Durchgang der äusseren Schleife komplett ausgeführt.
- Nicht dieselbe Laufvariable verwenden
- mehrfache Verschachtelungen möglich

### Bereiche

- sind in Python grundsätzlich **oben offen**:
   
`range(10, 20):`

alle Werte von 10 **bis und ohne** 20: `[10, 20)` bzw. `[10, 20[`

- fangen standardmäßig bei 0 an:
   
`range(20):`

alle Werte von 0 **bis und ohne** 20: `[0, 20)` bzw. `[0, 20[`

- können eine Schrittweite haben, (Standardwert ist 1):
   
`range(10, 20, 3):`

jeder dritte Wert von 10 **bis und ohne** 20

## Arrays (Listen)

- Sequentielle Datenstruktur zum Speichern mehrerer Elemente unter demselben Bezeichner (Namen).

- Speziell an Python: jedes Element kann einen anderen Datentyp haben.

- Beispiel:
   
`datum = [1, "Januar", 1970]`

- Zugriff auf Elemente erfolgt über Indizes

Wert: `datum[0] → 1`

Wertzuweisung: `datum[2] = 2021`  
`→ datum = [1, "Januar", 2021]`

### Listen erstellen

- Leere Liste: `leere_liste = []`

- Liste mit Inhalt:
   
`zweierpotenzen=[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]`

- Liste mit Einheitswerten (z.B. zehn Nullen):
   
`nullen=[0 for x in range(10)]`

- Anhand einer Funktion (Listenabstraktion)
   
`zweierpotenzen=[2**x for x in range(10)]`

- Syntax:
   
`SyntaxError`

- NameError:
   
`NameError`

- IndentationError:
   
`IndentationError`

- TypeError:
   
`TypeError`

- IndexError:
   
`IndexError`

- ZeroDivisionError:
   
`ZeroDivisionError`

## Listen – Fortsetzung

### Zugriff

- auf Elemente der Liste `liste`: über Indizes; **Indexierung**

- erstes Element: `liste[0]`

- letztes Element: `liste[len(liste)-1]` oder von hinten: `liste[-1]`

- auf **Teilbereiche (Slicing)**: mit dem Teilbereichsoperator `[start:stop:step]` (analog `range` bei for-Schleife)

Beispiele:

```
liste = [x for x in range(0, 100)]
```

- `liste[20:40:5] → [20, 25, 30, 35]`

- `liste[:] → Die ganze Liste`

- `liste[::2] → Jedes 2. Element aus der ganzen Liste`

- `liste[len(liste)//2:] → Die 2. Hälfte der Liste`

### Iteration über Listen (Listen durchlaufen)

und alle Elemente der Liste `liste` ausgeben:

- Mit for-Schleife: `i` nimmt alle Indizes der Liste an
   
`for i in range(0, len(liste)):`
`print(liste[i])`

- Mit for-Schleife: `element` nimmt alle Elemente der Liste an
   
`for element in liste:`
`print(element)`

- Funktion `enumerate()`: Zähler mit beliebigem Startwert:
   
`enumerate(liste, startwert)`
`for i, element in enumerate(liste, 5):`
`print(i, ":", element)`

- Länge der Liste (Anzahl Elemente): `len(liste)`
  
`→ letztes Element der Liste liste am Index len(liste)-1`

## Strings

- Ebenfalls indiziert von 0 bis Länge-1

- Können wie Listen bearbeitet werden

- Zusätzliche Funktionen wie:
   
`upper()`, `isupper()`, `lower()`, `islower()`

## Tupel

- Wie Listen: Sequentielle Datenstruktur zum Speichern mehrerer Elemente unter demselben Bezeichner (Namen), indiziert, aber:
   
`nicht veränderbar`: Elemente können gelesen werden, aber *nicht* gelöscht, eingefügt oder verändert.

- gekennzeichnet durch runde Klammern:
   
`mein_tupel = (1, 2, 3)`

## Modularität

- Standardbibliotheken und eigene Pythonscripts können direkt importiert werden:
   
`z.B. import random oder import my_script`

Nach dem Import kann auf den Inhalt des Modules zugegriffen werden: `random.shuffle(liste)`

- Weitere Module müssen erst installiert werden
   
 (in Jupyter direkt in der Codezelle möglich).
   
 Kommando: `pip install Modulname`.

## Fehlermeldungen

SyntaxError	Syntaxfehler
NameError	Element nicht deklariert/falsch geschrieben
IndentationError	Fehlerhafte Einrückung
TypeError	Fehlerhafter Datentyp
IndexError	Zugriff auf einen nicht existierenden Index
ZeroDivisionError	Division durch Null