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 Konsoleneingabe und liest sie (*String*)
  - `print()`, `print(…, end=" ")`, Default für `end`: Zeilenumbruch

Datentypen

Datentyp	Bedeutung	Variabeldefinition	`type(x)`
Integer	Ganzzahl	`a = 5`	`-> int`
Float	Fliesskommazahl	`a = 5.0`	`-> float`
String	Zeichenkette	`a = "Hallo!"`	`-> str`
Boolean	Wahrheitswert	`a = True`	`-> bool`

Arithmetische Operatoren

`a = 2, b = 5`

+	Addition	`a + b`	`-> 7`
-	Subtraktion	`b - a`	`-> 3`
*	Multiplikation	`a * b`	`-> 10`
/	Division	`a / b`	`-> 0.4`
//	Ganzzahlige Division	`a // b`	`-> 0`
%	Modulo (Rest der Division)	`a % b`	`-> 3`
**	Potenz	`a**b`	`-> 32`

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 rueckgabewert
```

- 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

- Geben 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)	`A and B`	`-> False`
or	logisches oder (OR)	`A or B`	`-> True`
not	Negation (NOT)	`not A`	`-> False`

Relationale Operatoren

Relationale Operatoren dienen dem Vergleich von Werten.
`a = 2, b = 5`

<	kleiner als	`a < b`	`-> True`
<=	kleiner gleich	`a <= b`	`-> True`
>	grösser als	`a > b`	`-> False`
>=	grösser gleich	`a >= b`	`-> False`
==	gleich	`a == b`	`-> False`
!=	ungleich	`a != b`	`-> True`

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 range(0,5) and !ferien:
    aufstehen(6.00)
elif heute == 5 and !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 muss 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 zählergesteuerter 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)
- fangen standardmässig bei 0 an:
  `range(20)`:
  alle Werte **von 0** bis und ohne 20: [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)]`

Funktionen

- Länge der Liste (Anzahl Elemente): **`len(liste)`**
- Element `element` hinten anhängen: **`append(element)`**
- Element am Index `index` löschen
  (gibt den Wert des Elements zurück): **`pop(index)`**
- Element `element` am Index `index` einfügen:
  `insert(index, element)`

Listen – Fortsetzung

Zugriff

- **auf Elemente** der Liste `liste`: über Indizes; **Indexierung**
  - erstes Element: `liste[0]`
  - letztes Element: `liste[len(liste)]` oder von hinten: `liste[-1]`
- **auf Teilbereiche:** mit dem Teilbereichsoperator **Slicing** **`[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)
```

- Mit der Funktion `enumerate()`: 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