

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 (Variablenamen): 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(element, index)

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 (wie Listen), 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

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: Kursform von a = a + 1 (analog: -=, *=, /=)
- Variablennamen klein schreiben
- Ein- und Ausgabe über die Konsole mit Hilfe der Funktionen
 - input() Liest die Konsoleneingabe, gibt einen String zurück
 - print(), print(..., end=" ")

Datentypen

Datentyp	Repräsentiert...	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

Logische Ausdrücke

Geben einen Boolean zurück.

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
- Beispiel:

```
if heute in range(0,5) and !ferien:
    aufstehen(6.00)
elif heute == 5 and !ferien:
    aufstehen(9.00)
else:
    ausschlafen()
```

Schleifen

Arrays als Datentyp sind nicht in Python integriert. Um Arrays zu verwenden wird eine **Bibliothek** benötigt, z.B. NumPy (Numerical Python), die mathematische Berechnungen mit mehrdimensionalen Arrays (Matrizen) erlaubt.

import numpy as np

Dimension:
Tupel (Höhe, Breite)

Datentyp

Arrays (in Numpy)

- Zufällige Welt (Matrix mit zufälligen Werten) im Bereich [Minimum, Maximum] erstellen [0,2], für zufällige Werte aus {0,1}:
np.random.randint(0,2,(20,40))

Minimum Maximum (exklusiv) Dimension: Tupel (Höhe, Breite)

- **Dimension:**
shape liefert den Tupel (Höhe, Breite)
dimension = new_world.shape → (4, 8)
height = new_world.shape[0]
width = new_world.shape[1]
- **Schleife über alle Zellen:**
np.ndindex liefert alle Koordinatenpaare (x,y) einer Matrix der Dimension (Höhe, Breite)
for index in np.ndindex(dimension):
 print(index) → (0,0), (0,1), ..., (3,7)

VISUALISIERUNG

Eine graphische Darstellung (Visualisierung) ist für eine Simulation nicht zwingend nötig, aber eine gute Visualisierung erleichtert das Verständnis.

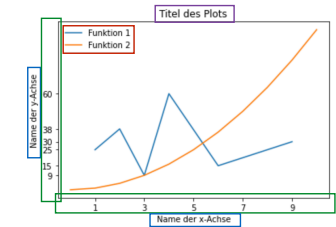
Die Bibliothek **pyplot** aus **matplotlib** erlaubt die graphische Darstellung (Plots).

import matplotlib.pyplot as plt

Beispiel Funktionsplots

- Werte für x und y definieren:
x=[1,2,3,4,6,9]
y=[25,38,9,60,15,30]
- Plot definieren
plt.plot(x,y,label='Funktion 1')
- Werte für einen zweiten Plot definieren:
x1=np.arange(0,11,1)
y1=x1**2 (x²)
- Zweiten Plot definieren
plt.plot(x1,y1,label='Funktion 2')
- Achsennamen
plt.xlabel('Name der x-Achse')
plt.ylabel('Name der y-Achse')
- Achsenbeschriftung:
plt.xticks(np.arange(1,11,step=2))
plt.yticks(y)
- Titel des Plots
plt.title('Titel des Plots')
- Legende
plt.legend()
- Plot anzeigen
plt.show()

Werte verteilen:
np.arange(von, bis (exklusiv), Schrittweite)

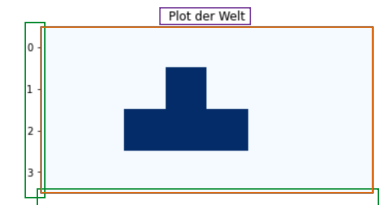


Welt darstellen: Matrix plotten

- **Matrix plotten** (graphisch darstellen):
plt.imshow(new_world, cmap=plt.cm.Blues)

Farbschema; Argument der Funktion imshow spezifizieren mit cmap=...

- **Beschriftung der x- und y-Achsen**
plt.xticks([1,2,5,7]) ← festgelegte Werte
Für Zeilen- und Spalten-Indizes:
plt.xticks(range(0, new_world.shape[1]))
plt.yticks(range(0, new_world.shape[0]))
falls **keine** Beschriftung gewünscht:
leere Arrays übergeben:
plt.xticks([]), plt.yticks([])
- **Titel**
plt.title('Plot der Welt')
- Plot anzeigen
plt.show()



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 (Variablenamen): 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 (Listenabstraktion):

```
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:

```
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**(element, index)

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 (wie Listen), 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