

# Python Syntax

## Vorkurs Bauingenieurwesen - Informatik

### Rechnen

Zeichen	Beschreibung	Beispiel	Ergebnis
+	Plus	4 + 5	9
-	Minus	12 - 7	5
*	Mal	3 * 6	18
/	Geteilt (exakt)	5 / 2	2.5
//	Geteilt (abgerundet)	5 // 2	2
%	Rest	5 % 2	1
**	Hoch	10 ** 3	1000

### Variablen

Eine Variable ist ein Name für ein Speicherort.

```
# lege einen neuen Speicherort mit name "friedhelm" an,  
# speichere dort vorerst die Zahl 3.
```

```
friedhelm = 3
```

```
# überschreibe was an Speicherort "friedhelm" steht  
# mit dem Ergebnis der Rechnung (also der Zahl 26)
```

```
friedhelm = 4 * 5 + 6
```

```
# speichere in "friedhelm" das was  
# vorher dort gespeichert war plus 4 (also 30)
```

```
friedhelm = friedhelm + 4
```

### Typen

Typ	Beschreibung	Beispiel
int	ganze Zahl	3
float	Kommazahl	3.0
bool	Wahr oder Falsch	False
str	Zeichenkette	"Hallo Freunde!"

### Entscheiden

Zeichen	Beschreibung	Beispiel	Ergebnis
==	gleicher Wert	"hallo" == "hallo"	True
!=	ungleicher Wert	"hello" != "Hallo"	True
<	kleiner als	4 < 5	True
>	größer als	4 > 5	False
<=	kleiner oder gleich	7 <= 7	True
>=	größer oder gleich	4 >= 5	False
not	Gegenteil	not True	False
and	Und	True and False	False
or	Oder	True or False	True

# Kontrollfluss

## Bedingte Ausführung

Grundsätzlich wird ein Programm immer Zeile für Zeile ausgeführt. Will man eine Zeile nur in bestimmten Fällen ausführen, nutzt man die `if`-Bedingung.

Nur, wenn der Ausdruck nach `if` und vor `:` zum Wert `True` ausgewertet wird, werden die weiter eingerückten Anweisungen darunter ausgeführt.

```
if 3 > 4:
    print("Hilfe, die Mathematik ist kaputt!")
    print("ruft die Feuerwehr!")
print("Diesen Text sehen wir immer.")
```

Ausgabe:

Diesen Text sehen wir immer.

Wenn die Bedingung Wahr ist mache das eine, wenn nicht das andere:

```
if 3 > 4:
    print("Hilfe, die Mathematik ist kaputt!")
    print("ruft die Feuerwehr!")
else:
    print("Puuh, die Mathematik ist noch heil.")
print("Diesen Text sehen wir immer.")
```

Ausgabe:

Puuh, die Mathematik ist noch heil.  
Diesen Text sehen wir immer.

Unterscheide mehrere Einzelfälle:

```
if 3 == 4:
    print("Ganze Zahlen sind kaputt!")
elif 3.0 > 4.0:
    print("Ganze Zahlen sind heil.")
    print("Kommazahlen sind kaputt!")
elif "hallo" == "tschüss":
    print("Ganze Zahlen sind heil.")
    print("Kommazahlen sind heil.")
    print("Zeichenketten sind kaputt!")
else:
    print("Entwarnung")
print("Diesen Text sehen wir immer.")
```

Ausgabe:

Entwarnung  
Diesen Text sehen wir immer.

## Wiederholen von Anweisungen

Will man *fast* die selben Anweisungen mehrfach ausführen, nutzt man Schleifen. Die `while`-Schleife funktioniert dabei zu Beginn wie eine `if`-Bedingung: Nur wenn der Logikausdruck zwischen `while` und `:` zum Wert `True` ausgewertet wird, springt das Programm in den weiter eingerückten Bereich und führt diese Zeilen aus.

Im Gegensatz zur `if`-Bedingung wird dieser Prozess nach Ausführung des eingerückten Bereiches wiederholt, bis der Logikausdruck das erste Mal `False` ist.

(Über die `for`-Schleife reden wir später.)

Version 1:	Version 2:	Ausgabe:
<code>x = 2</code>	<code>for x in range(2, 6):</code>	2
<code>while x &lt; 6:</code>	<code>    print(x)</code>	3
<code>    print(x)</code>	<code>print("fertig!")</code>	4
<code>    x = x + 1</code>		5
<code>print("fertig!")</code>		fertig!

# Funktionen

Die meisten Funktionen beim Programmieren sind von Außen identisch zu mathematischen Funktionen. Es wird ein (oder mehrere) Argument(e) von der Definitionsmenge auf die Bildmenge abgebildet. Definitionsmenge und Bildmenge sind in einem Programm bestimmte Typen. Als Beispiel wird die selbe Funktion  $f$  in der Mathematik geschrieben als

$$f: \mathbb{Z} \rightarrow \mathbb{Z}, \quad x \mapsto 2 \cdot x + 5$$

und in Python als

```
def f(x: int) -> int:
    return 2 * x + 5
```

Die Menge der ganzen Zahlen  $\mathbb{Z}$  entspricht in Python dem Typ `int`. Eine äquivalente Definition von  $f$  in Python wäre

```
def f(x: int) -> int:
    a = 2 * x # der name a existiert nur, solange f berechnet wird
    b = a + 5 # der name b existiert nur, solange f berechnet wird
    return b # der wert von b wird zurückgegeben
```

Die Variablen `a` und `b` existieren nur, bis der Computer an der Zeile `return b` ankommt. Erreicht der Computer eine Zeile die mit `return` beginnt, gilt die Funktion als fertig berechnet und der Ausdruck rechts davon (in diesem Fall `b`) wird ausgewertet und als Funktionswert zurückgegeben. Das Schlüsselwort `def` zeigt dem Computer an, dass an dieser Stelle eine neue Funktion beginnt.

So wie Python den Ausdruck  $3 * 2 + 4$  zu dem Wert 10 auswertet, kann auch die Funktion `f` angewendet werden. Der Ausdruck `f(7)` - 1 etwa wird zu 18 berechnet.

In einer Funktion selbst kann eine beliebige Abfolge von Anweisungen ausgeführt werden:

```
def fakultaet(n: int) -> int:
    ergebnis = 1 # diese Variable existiert nur, solange fakultaet berechnet wird.
    while n > 0:
        ergebnis *= n
        n -= 1
    return ergebnis # beende Funktion und gebe ergebnis zurück
n = 7 # diese Zeile wird nie ausgeführt
```

Die Funktion `fakultaet` berechnet die Fakultät einer Zahl. In Matheschreibweise ist die Fakultät (geschrieben mit einem Ausrufezeichen) definiert als

$$n! = n \cdot (n - 1) \cdot \dots \cdot 3 \cdot 2 \cdot 1$$

# Listen

Möchte man eine ganze Gruppe von Werten zusammen speichern, etwa alle Namen von Leuten in einer Klasse oder alle Primzahlen kleiner als 100 nutzt man eine Liste. Eine Liste kann beliebig viele Einträge speichern. Jeder Eintrag hat dabei eine feste Position, der erste Eintrag hat Position 0.

Der folgende Code produziert die Ausgabe `b` in der Konsole.

```
x = ["a", "b", "c", "d", "e", "f", "g"]
print(x[1])
```

Die eckigen Klammern haben dabei zwei verschiedene Rollen. Zuerst zeigen sie an, wo eine neue Liste anfängt und aufhört. Diese neue Liste kriegt den Name `x`. Wenn man hinter eine bestehende Liste (hier `x`) eckige Klammern schreibt, greift man auf Einträge der Liste zu. Im Beispiel wird der Eintrag an Index 1 gelesen, also `"b"`. Weitere Varianten sind

- `x[1:3]` erzeugt eine neue Liste die die Werte von `x` an `x'` Indices 1 und 2 hat, also `["b", "c"]`.

- `x[:3]` erzeugt eine neue Liste, die gleich beginnt wie `x`, aber schon nach den ersten 3 Einträgen aufhört, also `["a", "b", "c"]`.
- `x[1:]` erzeuge eine neue Liste, die gleich endet wie `x`, aber den ersten Eintrag von `x` nicht enthält, also `["b", "c", "d", "e", "f", "g"]`.
- `x[::2]` erzeuge eine Liste, die jeden zweiten Eintrag von `x` enthält, beginnend am Anfang, also `["a", "c", "e", "g"]`.
- `x[-1]` gibt den letzten Eintrag von `x` zurück, also `"g"`.
- `x[-2]` gibt den vorletzten Eintrag von `x` zurück, also `"f"`.

Möchte man eine neue Liste erzeugen, gibt es verschiedene Möglichkeiten

- explizit hinschreiben: `[1, 2, 3, 4]`.
- kleinere Listen aneinanderhängen: `[0, 2] + [1, 1, 1]` ergibt `[0, 2, 1, 1, 1]`.
- Eine kleinere Liste wiederholen: `5 * [0, 1]` ergibt `[0, 1, 0, 1, 0, 1, 0, 1, 0, 1]`.
- Etwas anderes in eine Liste umwandeln: `list("abcde")` ergibt `["a", "b", "c", "d", "e"]`.

Weitere wichtige Operationen mit einer Liste `x` sind

- Ein neuen Eintrag hinten dran hängen: `x.append(6)`
- Den letzten Eintrag entfernen: `x.pop()`
- Den Eintrag an Index 3 überschreiben: `x[3] = "Lelele"`
- Alle Einträge löschen: `x.clear()`