Funktionen

`print(...)` oder `input(...)` oder `len(...)` sind Beispiele für (bereits vorhandene) Funktionen in Python. Man benutzt Funktionen immer auf die gleiche Weise: Man schreibt den Funktionsnamen und in runden Klammern die sog. Argumente, die der Funktion mitgegeben werden.

- `print(...)`: Funktion schreibt den Python-Ausdruck, der als Argument mitgegeben wird, in die Konsole.
- `input(...)`: Funktion schreibt den Python-Ausdruck, der als Argument mitgegeben wird, in die Konsole; wartet auf eine Eingabe des Benutzers und liefert die Eingabe zurück.
- `len(...)`: liefert die Länge des als Arguments übergebenen Strings zurück.

Was bedeutet die Formulierung "liefert … zurück" genau? Liefert eine Funktion ein Ergebnis zurück, so wird im Python-Ausdruck an der Stelle, an der die Funktion aufgerufen wird, das zurückgelieferte Ergebnis eingesetzt. Beispiel:

Die Funktion `len("Hallo")+3` liefert den Wert 5 zurück. Damit wird aus dem Ausdruck `len("Hallo")+3` wird `5+3` und schließlich `8`.

Die Anzahl der übergebenen Argumente (auch Parameter genannt) ist unterschiedlich: es gibt Funktionen ohne Argument, mit einem Argument oder mit mehreren Argumenten. Beim Aufruf einer Funktion ohne Argument müssen trotzdem runde Klammern geschrieben werden, z.B.:

```
eingabe = input()
```

Definition einer Funktion

Man kann in Python eigene Funktionen definieren. Beispiel:

```
def schreibeHallo():
    print("Hallo!")
    print("Heute ist ein schöner Tag, oder nicht?")
```

Bei der Definition wird ein eingerückter Codeblock verwendet, analog zu den Codeblöcken, die in der Ablaufsteuerung bei `if` oder `while` verwendet werden.

Nun kann die so definierte Funktion `schreibeHallo()` wie alle anderen Python-Funktionen benutzt werden:

```
print("irgendein anderer Text")
schreibeHallo()
print("noch mehr Text")
```

```
irgendein anderer Text
Hallo!
Heute ist ein schöner Tag, oder nicht?
noch mehr Text
```

Die Funktion `schreibeHallo()` wurde ohne Argumente definiert. In diesem Fall darf man auch keines verwenden.

```
schreibeHallo("Peter")
```

```
TypeError: schreibeHallo() takes 0 positional arguments but 1 was given
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[3], line 1
----> 1 schreibeHallo("Peter")
TypeError: schreibeHallo() takes 0 positional arguments but 1 was given
```

Funktionsdefinition mit Argumenten

Argumente kann man bei der Definition der Funktion in den runden Klammern angeben. Dabei darf man beliebige Variablennamen verwenden.

```
def schreibeHallo(name):
    print(f"Hallo, {name}!")
    laenge = len(name)
    print(f"Dein Name ist {laenge} Buchstaben lang.")
```

Jetzt hat `schreibeHallo` einen Parameter. Der muss beim Aufruf auch angegeben werden.

```
schreibeHallo("Egon")
schreibeHallo("Walburga")
schreibeHallo()
```

```
Hallo, Egon!
Dein Name ist 4 Buchstaben lang.
Hallo, Walburga!
Dein Name ist 8 Buchstaben lang.

TypeError: schreibeHallo() missing 1 required positional argument: 'name'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[5], line 3
      1 schreibeHallo("Egon")
      2 schreibeHallo("Walburga")
----> 3 schreibeHallo()
TypeError: schreibeHallo() missing 1 required positional argument: 'name'
```

Die Bezeichnung `name` für das Argument ist eine interne Bezeichnung, die nur in der Funktionsdefinition Bedeutung hat. Außerhalb der Definition spielt diese Bezeichnung keine Rolle.

```
meinVorname = "Ignatius"
schreibeHallo(meinVorname)
```

```
Hallo, Ignatius!
Dein Name ist 8 Buchstaben lang.
```

Man kann auch Funktionen mit mehreren Argumenten definieren:

```
def schreibeHallo(name, alter):
    if alter < 18:
        print(f"Hi {name}!")
    elif alter < 65:
        print(f"Hallo, {name}!")
    else:
        print(f"Guten Tag, {name}!")
```

```
schreibeHallo("Max", 14)
schreibeHallo("Anna", 30)
schreibeHallo("Opa", 70)
```

```
Hi Max!
Hallo, Anna!
Guten Tag, Opa!
```

Rückgabewerte

Selbst definierte Funktionen können ebenso wie z.B. `len(...)` Rückgabewerte haben. Dazu verwendet man in der Funktionsdefinition die Anweisung `return`. Dahinter wird der Rückgabewert angegeben.

```
def rechteckFlaeche(breite, hoehe):
    flaeche = breite * hoehe
    return flaeche
```

Die Verwendung funktioniert wie bei anderen Funktionen mit Rückgabewert auch:

```
b = 5
h = 10
A = rechteckFlaeche(b, h)
print(f"Die Fläche des Rechtecks mit Breite {b} und Höhe {h} ist {A}.")
```

```
Die Fläche des Rechtecks mit Breite 5 und Höhe 10 ist 50.
```

Man kann in der Definition mehrere `return`-Anweisungen verwenden. Die Ausführung der Funktion endet mit der ersten `return`-Anweisung, die ausgeführt wird.

```
def rechteckFlaeche(breite, hoehe):
    if breite < 0 or hoehe < 0:
        return "Ungültige Eingabe: Breite und Höhe müssen positiv sein."
    flaeche = breite * hoehe
    return flaeche
```

```
print(rechteckFlaeche(5, 10))
print(rechteckFlaeche(-5, 10))
```

```
50
Ungültige Eingabe: Breite und Höhe müssen positiv sein.
```

keyword-Argumente

Im Normalfall werden die Argumente einer Funktion in der Reihenfolge übergeben, wie sie in der Funktionsdefinition angegeben. In der Funktionsdefinition im Beispiel oben

```
def schreibeHallo(name, alter):
```

ist der erste Parameter `name`, der zweite `alter`. Die Reihenfolge, in der die Argumente beim Aufruf angegeben werden, muss dieser Reihenfolge entsprechen.

Davon unterscheiden sich die sogenannten *keyword-Argumente*. Dabei bekommen die einzelnen Argumente Namen, die beim Aufruf der Funktion mit angegeben werden mussen. Dadurch kann die Reihenfolge der Argumente beim Aufruf geändert werden.

Die obige Beispielfunktion `schreibeHallo` kann auch so aufgerufen werden:

```
schreibeHallo(alter=14, name="Max")
```

```
Hi Max!
```

Hier werden die Argumente `name` und `alter` explizit benannt. Dadurch ist die Reihenfolge der Argumente beim Aufruf egal.

Man darf keyword-Argumente und normale Argumente auch mischen. Dabei müssen die normalen Argumente zuerst angegeben werden, danach die keyword-Argumente.

```
schreibeHallo("Max", alter=14)
```

```
Hi Max!
```

Verwendet man nach einem keyword-Argument ein nicht-keyword-Argument, so führt das zu einem Fehler:

```
schreibeHallo(name="Max", alter)
```

```
SyntaxError: positional argument follows keyword argument (2499671907.py, line 1)
  Cell In[15], line 1
    schreibeHallo(name="Max", alter)
                                    ^
SyntaxError: positional argument follows keyword argument
```

optionale Argumente und default-Werte

Es ist möglich, in der Funktionsdefinition für Argumente default-Werte anzugeben. Diese Werte werden verwendet, wenn beim Aufruf der Funktion kein entsprechendes Argument angegeben wird.

```
def schreibeHallo(name, alter=30):
    if alter < 18:
        print(f"Hi {name}!")
    elif alter < 65:
        print(f"Hallo, {name}!")
    else:
        print(f"Guten Tag, {name}!")
```

```
schreibeHallo("Max", 14)   # alter wird mit 14 belegt
schreibeHallo("Anna")      # alter wird mit default-Wert 30 belegt
```

```
Hi Max!
Hallo, Anna!
```

Man kann auch mehr als ein Argument mit einem default-Wert versehen.

```
def schreibeHallo(name="John", alter=30):
    ...
```

```
schreibeHallo("Max")
schreibeHallo()
```

```
Hallo, Max!
Hallo, John!
```

Gibt es Argumente ohne default-Wert, müssen diese bei der Definition vor den Argumenten mit default-Wert angegeben werden. Folgendes führt zu einem Fehler:

```
def schreibeHallo(name="John", alter):
    ...
```

```
SyntaxError: non-default argument follows default argument (1235581580.py, line 1)
  Cell In[20], line 1
    def schreibeHallo(name="John", alter):
                                   ^
SyntaxError: non-default argument follows default argument
```

lokaler und globaler Scope