Grundlagen: Eigene Funktionen schreiben in Python

Was ist eine Funktion?

- Ein Block von wiederverwendbarem Code, der eine spezifische Aufgabe ausführt.
- Funktionen helfen, den Code übersichtlicher und leichter lesbar zu machen.

Warum Funktionen verwenden?

- Code-Wiederverwendbarkeit
- Vereinfachung komplexer Probleme
- Verbessert die Code-Organisation
- Einfache Fehlersuche und Wartung

Grundlagen: Eigene Funktionen schreiben in Python

Eine Funktion definieren

Syntax der Funktionsdefinition:

```
def funktionsname(parameter):
    # Code-Block (auszuführende Anweisungen)
    return  wert
```

Erklärung:

def: Schlüsselwort zur Definition einer Funktion.

funktionsname: Name, der der Funktion zugewiesen wird.

parameter: Optionale Werte, die der Funktion beim Aufruf übergeben werden, und die sie zur Erledigung ihrer Aufgaben verwendet.

 # Code-Block: Der Teil des Codes, der ausgeführt wird, wenn die Funktion aufgerufen wird.

return: Optional, gibt einen Wert von der Funktion zurück. Beendet die Funktion.

Grundlagen: Eigene Funktionen schreiben in Python

Beispiel 1:

```
def greet(name):
    print("meaw")
    return f"Hallo, {name}!"
    # return beendet die Funktion und gibt einen Wert zurück. (Ohne return gibt die Funktion None zurück.)
```

Funktion aufrufen:

```
greet('Alice')   # meaw


print(greet('Alice') )
        #meaw
        # Hallo, Alice!
```

Grundlagen: Eigene Funktionen schreiben in Python

Das None Schlüsselwort

- Eine spezielle Konstante in Python, die das Fehlen eines Wertes darstellt.
- Wird oft von Funktionen zurückgegeben, die keinen expliziten Rückgabewert haben.

```
def greet(name):
    print("meaw")
    # Hier fehlt eine return-Anweisung

print(greet("Alice"))

#meaw
#None
```

Grundlagen: Eigene Funktionen schreiben in Python

Beispiel 2:

```
def begruessung():
    print("Hallo! Willkommen im Python-Kurs.")


begruessung()  # Funktionsaufruf
                    # Ausgabe : Hallo! Willkommen im Python-Kurs
```

- Die Klammern beim Aufruf sind zwingend erforderlich, auch wenn es keine Parameter gibt.
- Funktionen müssen **vor dem** Aufruf definiert sein.

Parameter vs. Argumente

Parameter:

- Variablen, die in der Klammer bei der Funktionsdefinition aufgelistet sind. Parameter sind Platzhalter für Werte, die beim Aufruf übergeben werden.

Argumente:

- Werte, die der Funktion beim Aufruf übergeben werden.

```
def multiplizieren(x, y):  # x, y sind Parameter
    return x * y

ergebnis = multiplizieren(3, 4)  # 3, 4 sind Argumente
```

Optionale Parameter (Standardwerte)

Parameter können Standardwerte haben, die verwendet werden, wenn kein Argument übergeben wird.

```
def begruessung(name="Gast"):
    print("Hallo,", name)

begruessung()       # Hallo, Gast
begruessung("Anna")  # Hallo, Anna
```

Vereinfacht Funktionsaufrufe.

Bietet Flexibilität.

Positionsargumente

Argumente, die einer Funktion in der Reihenfolge übergeben werden, die den Parametern entspricht.

```
def teilen(a, b):
    return a / b

ergebnis = teilen(10, 2)  # a=10, b=2
```

Die Reihenfolge der Argumente ist entscheidend.

Schlüsselwortargumente (Keyword arguments)

Argumente, die einer Funktion durch explizites Benennen jedes Parameters übergeben werden.

```
def teilen(a, b):
        return a / b

ergebnis = teilen(b=4, a=10)  # a=10, b=4
```

Die Reihenfolge spielt keine Rolle.

Verbessert die Code-Lesbarkeit.

Kombination aus Standardwerten + Schlüsselwortargumenten

gericht ist ein normaler Parameter → muss immer übergeben werden.

beilage ist ein optional definierter Parameter → Standardwert ist "Reis".

Wenn beim Funktionsaufruf kein Wert für beilage angegeben wird, verwendet Python den Standardwert.

```
def rezept(gericht, beilage="Reis"):
    print(f"Das Gericht ist: {gericht}")
    print(f"Als Beilage gibt es: {beilage}")


rezept("Curry") # Standardwert wird verwendet
rezept("Suppe", "Brot")  # beide Werte werden als Positionsargumente übergeben.
rezept("Pizza", beilage="Salat") #  "Pizza" ist ein Positionsargument, beilage="Salat" ist ein Schlüsselwortargument
```

```
# Ausgabe 1
Das Gericht ist: Curry
Als Beilage gibt es: Reis

# Ausgabe 2
Das Gericht ist: Suppe
Als Beilage gibt es: Brot

# Ausgabe 3
Das Gericht ist: Pizza
Als Beilage gibt es: Salat
```

Du kannst optionale Parameter mit Standardwerten definieren.

In einem Funktionsaufruf kannst du Positionsargumente und Schlüsselwortargumente kombinieren. Das erhöht die Flexibilität und Lesbarkeit deines Codes.

Kombination von Argumenten - Die Richtige Reihenfolge

Kombination von Positions- und Schlüsselwortargumenten:

Zuerst Positionsargumente, gefolgt von Schlüsselwortargumenten.

```
def potenz(basis, exponent):
    return basis ** exponent

ergebnis = potenz(2, exponent=3)
print(ergebnis) #8

# potenz(exponent=3, 2)  # SyntaxError
#  Achtung: Ein Positionsargument nach einem Schlüsselwortargument ist nicht erlaubt – das führt zu einem Syntaxfehler
```

Grundlagen: Eigene Funktionen schreiben in Python

Funktionen sind Objekte! (einfache erklärung ohne OOP Details)

Funktionen wie ganz normale Werte behandeln :

In Python kann man mit Funktionen fast so umgehen wie mit Zahlen, Texten oder Listen. Man kann sie z. B. einer Variable zuweisen oder weitergeben (und später aufrufen) – ohne sie direkt auszuführen.

Beispiel: Funktion einer Variable zuweisen

```
def hallo():
    return "Hallo Welt"


print(hallo())  # ganz normaler Funktionsaufruf → Ausgabe: Hallo Welt
x = hallo       # Wir speichern die Funktion in x
                # Achtung: ohne Klammern! Funktion wird gespeichert, nicht ausgeführt
print(x())      # Jetzt rufen wir die Funktion über x() auf → auch: Hallo Welt
```

Eine Funktion zu speichern ist nützlich, wenn man sie umbenennen, als Argument übergeben (an andere Funktion), später flexibel verwenden, in Listen speichern oder z. B. als Callback bei einem Ereignis nutzen möchte.

Grundlagen: Eigene Funktionen schreiben in Python

Funktion als Argument an eine andere Funktion zu übergeben

```
def ausfuehren(f):
    print(f())
def hallo():
    return "Hallo!"
ausfuehren(hallo)  # Übergabe der Funktion selbst (ohne Klammern!)   # Ausgabe: Hallo!
```

Um mit einer Liste oder Tabelle von Funktionen zu arbeiten

```
def quadrat(x): return x * x        # Diese Funktion ist in einer einzigen Zeile definiert. Das ist eine kurze Schreibweise, die man verwenden kann, wenn die Funktion nur eine Zeile braucht.
def wurzel(x): return x ** 0.5
funktionen = [quadrat, wurzel]
for f in funktionen:
    print(f(9))  # zuerst 81, dann 3.0
```

Funktion Je nach Bedingung auswählen:

```
# Zwei Funktionen werden definiert
def admin_start():
    print("Admin-Modus wird gestartet.")
def nutzer_start():
    print("Nutzer-Modus wird gestartet.")
# Benutzername (könnte z. B. per Eingabe kommen)
benutzer = "Admin"
# Je nach Benutzername wird eine Funktion ausgewählt
if benutzer == "Admin":
    aktion = admin_start    # Wir speichern den Namen der Funktion (ohne sie auszuführen!)
else:
    aktion = nutzer_start
# Jetzt wird die gewählte Funktion aufgerufen
aktion()  # Ausgabe: Admin-Modus wird gestartet.
```

Grundlagen: Eigene Funktionen schreiben in Python

Funktionen als Callback / Event-Handler:

In einigen Programmen – z. B. bei grafischen Oberflächen (wie Tkinter oder Webanwendungen) – brauchst du Funktionen, die nicht sofort, sondern später automatisch ausgeführt werden, wenn etwas Bestimmtes passiert.

Ein Callback ist eine Funktion, die nicht sofort aufgerufen wird, sondern die du an ein anderes System übergibst, damit sie später ausgeführt wird – zum Beispiel nach einem Mausklick oder wenn der Benutzer etwas eingibt. Bei Callbacks speicherst du den Namen einer Funktion – das System ruft sie später automatisch auf.

```
def begruessung():
    print("Willkommen!")


button.on_click(begruessung)  # Wir übergeben den Namen der Funktion an das System (on_click)
                              # Python ruft begruessung() erst dann auf, wenn der Button geklickt wird
                              # Das Programm entscheidet selbst, wann die Funktion ausgeführt wird.
                              # Du gibst dem System nur die Anweisung, was zu tun ist, wenn das Ereignis eintritt.
```

Grundlagen: Eigene Funktionen schreiben in Python

Vergleich: Eingebaute vs. eigene Funktion

Python bringt viele nützliche Funktionen mit – aber manchmal ist es besser (oder nötig), eigene Funktionen zu schreiben, um genau das zu tun, was du brauchst.

```
print(len("Python"))  # eingebaute Funktion
                      #  Funktion ist schon in Python eingebaut – du musst nichts extra schreiben oder importieren


def eigene_len(text): # eigene Funktion – von dir selbst geschrieben
    zaehler = 0
    for buchstabe in text:
        zaehler += 1
    return zaehler


print(eigene_len("Python"))  # eigene Funktion aufruf
```

Übrigens: Du kannst jede beliebige eingebaute (in-built) Funktion (wie sum(), max(), round() usw.) selbst nachbauen – einfach mal ausprobieren!

Zusammenfassung & Best Practices

Wichtige Erkenntnisse:

- Das Verständnis von Funktionen macht den Code modular und wiederverwendbar.
- Verschiedene Möglichkeiten, Argumente zu übergeben (Position, Schlüsselwort, gemischt), bieten Flexibilität.
- None steht in Python allgemein für „kein Wert vorhanden“ – z. B. bei leeren Variablen oder fehlenden Ergebnissen.
  --> In Funktionen bedeutet None, dass kein Rückgabewert mit return angegeben wurde – Python gibt dann automatisch None zurück.
- Standardparameter können die Verwendung von Funktionen vereinfachen.

Verwenden Sie klare Namen:

- Funktions- und Parameternamen sollten aussagekräftig sein.

Benennungskonventionen (Naming Convention):

- Verwenden Sie die snake_case-Schreibweise für Funktions- und Parameternamen (z. B. berechne_summe), um Konsistenz und Lesbarkeit im Code zu gewährleisten.

Halten Sie Funktionen klein:

- Konzentrieren Sie sich auf eine einzelne Aufgabe oder Verantwortung.

Grundlagen: Eigene Funktionen schreiben in Python

Mehr erkunden:

https://www.w3schools.com/python/python_functions.asp

https://www.geeksforgeeks.org/python-functions/

https://docs.python.org/3/tutorial/controlflow.html#defining-functions