Erweiterte Funktionen in Python

Fortgeschrittene Parameter und Funktionsverhalten

In Python können Funktionen nicht nur mit festen Parametern geschrieben werden. Es gibt viele zusätzliche Möglichkeiten, um Funktionen flexibler und anpassbarer zu gestalten.

Heute lernen wir:

- Wie man eine Funktion schreibt, die eine beliebige Anzahl an Werten verarbeiten kann – auch wenn vorher nicht bekannt ist, wie viele genau übergeben werden.
- Wie man Funktionen so schreibt, dass bestimmte Werte beim Aufruf mit Namen angegeben werden müssen – damit der Code klarer und weniger fehleranfällig ist.
- Wie Listen oder Dictionaries direkt beim Funktionsaufruf entpackt werden können – sodass man nicht jeden Wert einzeln angeben muss.
- Wie man beim Schreiben einer Funktion Hinweise zum erwarteten Datentyp geben kann – damit der Code besser verständlich und leichter wartbar wird.

Diese Techniken helfen dabei, professionelleren Python-Code zu schreiben, der gut strukturiert, leicht wartbar und wiederverwendbar ist – besonders in größeren Programmen oder Teams.

*args - Variable Anzahl von Positionsargumenten

Mit *args können Sie einer Funktion eine variable Anzahl von Positionsargumenten übergeben. Die Argumente werden als Tupel in der Funktion verfügbar gemacht.

```
def summieren(*zahlen):
    print(zahlen)
    return sum(zahlen)


print(summieren(1, 2, 3))       # Ausgabe: 6
print(summieren(10, 20, 30, 40)) # Ausgabe: 100
```

- *zahlen ist ein **Tupel**, das alle übergebenen Positionsargumente enthält.
- So kannst du beliebig viele Zahlen verarbeiten.
- Argumentname *args ist üblich – aber auch andere Namen möglich (z.B. *werte, *elemente...)

**kwargs - Variable Anzahl von Schlüsselwortargumenten

Mit **kwargs (z. B. **infos) kann man einer Funktion eine beliebige Anzahl von benannten Argumenten (Schlüsselwortargumenten) übergeben. Diese werden innerhalb der Funktion als Dictionary zur Verfügung gestellt.

```
def nutzerprofil(**infos):
    for key, value in infos.items():
        print(f"{key}: {value}")


nutzerprofil(name="Anna", beruf="Lehrerin", ort="Berlin")
```

Ausgabe:
name: Anna
beruf: Lehrerin
ort: Berlin

- **infos sammelt alle übergebenen benannten Argumente als Dictionary (key: value).
- Die Anzahl und die Schlüssel (keys) der übergebenen Werte müssen vorher nicht festgelegt werden.
- Der Parametername **kwargs ist üblich – aber man kann ihn auch anders nennen (z. B. **daten, **infos, **optionen).

Arbeiten mit *args und **kwargs

Wann verwenden?

- *args wird verwendet, wenn Sie eine Funktion schreiben möchten, die eine unbekannte Anzahl von Positionsargumenten akzeptiert.
- **kwargs wird verwendet, wenn Sie eine Funktion schreiben möchten, die eine unbekannte Anzahl von Schlüsselwortargumenten akzeptiert.

Erweiterte Funktionen in Python

Kombination aus normalen Parametern, *args und **kwargs

So können wir sehr flexible Funktionen erstellen, die:

- ein paar feste Parameter haben,
- beliebig viele zusätzliche Werte (*args),
- sowie optionale benannte Einstellungen (**kwargs) entgegennehmen.

```
def beispiel(a, *args, **kwargs):
    print("a:", a)                    # a: 10
    print("args:", args)              # args: (20, 30)
    print("kwargs:", kwargs)          # kwargs: {'name': 'Max', 'beruf': 'Dozent'}
beispiel(10, 20, 30, name="Max", beruf="Dozent")


    # a ist ein normale Parameter, *args sammelt alle weiteren Positionsargumente in einem Tupel.
    # **kwargs sammelt alle Schlüsselwortargumente (z. B. name=...) in einem Dictionary.
```

Erweiterte Funktionen in Python

Kombination aus normalen Parametern, *args und **kwargs - Reihenfolge in der Funktionsdefinition

```
def funktion(a, *args , **kwargs ):
    ...
```

1. Normale Parameter (z. B. a)
2. *args – für mehrere Positionsargumente
3. **kwargs – für mehrere benannte (Schlüsselwort) Argumente

Wichtig: Diese Reihenfolge muss eingehalten werden.

Andernfalls gibt Python einen SyntaxError.

Erweiterte Funktionen in Python

Kombination aus normalen Parametern, *args und **kwargs - Was muss man beim Aufruf beachten?

```
def funktion(a, *args , **kwargs ):
    ...
```

Immer zuerst normale und positionsbasierte Argumente, dann Schlüsselwortargumenten :

```
beispiel(10, 20, 30, name="Max", beruf="Dozent")
```

Keine Positionsargumente nach Schlüsselwortargumenten :

```
beispiel(10, 20, 30, name="Max", beruf="Dozent", 5)  # → SyntaxError: positional argument follows keyword argument
```

Keine doppelten Argumente :

```
beispiel(10, name="Max", name="Anna")  # → SyntaxError: keyword argument repeated
```

Keyword-only-Parameter (Schlüsselwortpflicht mit *)

Mit * in der Parameterliste kannst du festlegen, dass alle nachfolgenden Parameter beim Aufruf zwingend benannt werden müssen.

```
def greeting(name, *, anrede="Herr/Frau"):
    print(f"Hallo {anrede} {name}!")
greeting("Müller", anrede="Dr.")     # Hallo Dr. Müller!
greeting("Meier")                    # Hallo Herr/Frau Meier!
greeting("Meier", "Professor")       #  TypeError: greeting() takes 1 positional argument but 2 were given
```

Ein * in der Parameterliste sagt: Ab hier bitte nur noch mit Namen (Schlüssel)!

Warum * verwenden?

- Verhindert Missverständnisse bei der Übergabe von Werten.
- Erhöht die Lesbarkeit und Klarheit bei vielen Parametern.
- Verhindert Fehler durch falsche Positionszuordnung.

Erweiterte Funktionen in Python

Positions-only-Parameter – Nur per Position erlaubt (/)

- Mit / kannst du festlegen, dass bestimmte Parameter nur durch ihre Position übergeben werden dürfen – nicht mit Namen!
- / sagt Python: Alles vor dem / darf nicht mit Namen aufgerufen werden.
- Wird oft genutzt, wenn es semantisch keinen Sinn ergibt, Argumente zu benennen (z. B. bei mathematischen Operationen).
- Performance-kritische Funktionen: z. B. aus C-Extensions→ Nur Positionsargumente = Python kann schneller verarbeiten (Parser-Optimierung)
- Hinweis: Selten gebraucht, aber wichtig zu kennen

```
def addiere(a, b, /):
    return a + b
addiere(3, 4)          # Kein Error- So erwartet
addiere(a=3, b=4)   #  addiere() got some positional-only arguments passed as keyword arguments: 'a, b'
```

Typ-Annotationen – Was erwartet die Funktion eigentlich?

Mit Typ-Annotationen kannst du Python (und dir selbst!) mitteilen, welche Datentypen eine Funktion erwartet und was sie zurückgibt. **Sie sind Hinweise, keine Befehle!** Hilfreich für Lesbarkeit, Fehlersuche & automatische Tools wie mypy, IDEs etc.

```
def quadrat(x: int) -> int:
    return x * 2


print(quadrat(5)) # 10
print(quadrat("Hallo") ) #HalloHallo
                    #  Kein Fehler – obwohl "Hallo" kein int ist!  Python prüft die Annotation nicht und führt den Code trotzdem aus.!
```

Warum trotzdem nutzen?

- Bessere Dokumentation (auch für andere) ,
- Tools wie mypy erkennen Fehler automatisch,
- IDEs wie PyCharm, VS Code geben Hinweise & Autovervollständigung,
- Besonders nützlich in größeren Projekten oder im Team

Argumente entpacken beim Funktionsaufruf (* und **)

Liste entpacken mit *

```
def infos(name, stadt):
    print(f"{name} lebt in {stadt}")


daten_liste = ["Anna", "Berlin"]
infos(*daten_liste)    # entspricht: infos("Anna", "Berlin")
                       # Der Stern * vor daten_liste sorgt dafür, dass Python die Liste entpackt
                       # *["Anna", "Berlin"]  →  "Anna", "Berlin". Die Funktion erhält also zwei Argumente: name="Anna", stadt="Berlin"
                       # Hinweis: Nach dem Entpacken sind "Anna" und "Berlin" normale Strings (Typ str)
                       # Ausgabe Anna lebt in Berlin
```

Warum ist das nützlich?

Stell dir vor, du bekommst Daten z. B. aus einer Datei, einer Benutzereingabe oder einem Webformular – dann hast du oft eine Liste oder ein Tupel. Mit * kannst du diese Daten direkt an die Funktion übergeben, ohne jedes Element einzeln auszulesen oder zu schreiben.

Erweiterte Funktionen in Python

Argumente entpacken beim Funktionsaufruf (* und **)

Dictionary entpacken mit **

Mit ** kannst du ein Dictionary entpacken und seine Schlüssel-Wert-Paare direkt als benannte Argumente an eine Funktion übergeben.

```
def infos(name, stadt):
    print(f"{name} lebt in {stadt}")


daten_dict = {"name": "Tom", "stadt": "Hamburg"}
infos(**daten_dict)        # **daten_dict sorgt dafür, dass Python: "name" → name= "Tom" ,    "stadt" → stadt="Hamburg" an die Funktion übergibt.
                           # Der Aufruf entspricht intern:  infos(name="Tom", stadt="Hamburg")
```

Typischer Einsatz:

Du hast Daten aus einer JSON-Datei, Benutzerformular, API

Du möchtest flexibel und dynamisch Funktionen aufrufen, ohne Werte manuell zuzuweisen

Erweiterte Funktionen in Python

Zusammenfassung

Thema	Was es tut
*args	Sammelt **beliebig viele Positionsargumente** in einem Tupel
**kwargs	Sammelt **beliebig viele Schlüsselwortargumente** in einem Dictionary
* in Funktionsparametern	Erzwingt, dass folgende Parameter **nur mit Namen** übergeben werden
/ in Funktionsparametern	Erzwingt, dass vorherige Parameter **nur per Position** übergeben werden
Typ-Annotationen	Geben erwartete Datentypen an. Typ-Annotationen sind Hinweise, keine Vorschriften – Python prüft sie nicht.
* beim Funktionsaufruf	Entpackt Iterable (z. B. Liste, Tupel) → übergibt Werte als Positionsargumente
** beim Funktionsaufruf	Entpackt Dictionary → übergibt Werte als benannte Argumente

Erweiterte Funktionen in Python

Mehr erkunden:

https://www.geeksforgeeks.org/python/args-kwargs-python/

https://www.scaler.com/topics/python/args-and-kwargs-in-python/

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