DS 8 – Funktionen: Programme aus Bausteinen

Informatik 10 – MINT-Zweig | Textuelle Programmierung und Datenstrukturen

Lernziele

Ich kann

- eine Funktion mit `def` definieren und sie aufrufen,
- Funktionen mit Parametern schreiben und ihnen beim Aufruf Argumente übergeben,
- mit `return` einen Rückgabewert zurückgeben und damit weiterrechnen,
- erklären, warum `print` in einer Funktion etwas anderes ist als `return`,
- beschreiben, was bei einem Funktionsaufruf Schritt für Schritt passiert,
- eine Funktion mit mehreren Aufrufen testen.

🗣️ Sprachhilfe

Beispiel: die Funktion `f` mit dem Parameter `x` und dem Rumpf `return x * x + 1`. Der Aufruf `f(4)` liefert 17.

- *„Zur Eingabe 4 bekomme ich die 17."* → „Wenn ich der Funktion die Zahl 4 gebe, liefert sie mir die 17 als Ergebnis." → „Wenn die Funktion `f` mit dem Wert 4 für den Parameter `x` aufgerufen wird, berechnet sie das Ergebnis 17 und gibt es zurück."
- „Der Funktion wird der Wert 4 übergeben. Die Funktion berechnet 4 · 4 + 1 und gibt 17 zurück."

Worum es geht

Du hast schon viele Funktionen benutzt: `print(...)`, `input(...)`, `int(...)`, `round(...)`. Du schreibst ihren Namen, gibst ihnen in Klammern Werte mit und bekommst ein Ergebnis. Wie sie innen arbeiten, musst du nicht wissen. Heute schreibst du **eigene Funktionen**. So schreibst du eine Rechnung, die du oft brauchst, nur einmal auf.

Die Idee kennst du aus Mathe: $f(x) = x^2$ macht aus 4 die 16 und aus 7 die 49.

1 Eine eigene Funktion definieren

Dein Programm soll an mehreren Stellen eine Trennlinie ausgeben. Statt sie jedes Mal abzutippen, gibst du der Anweisung einen Namen.

```
1 def trennlinie():
2     print("--------------------")
3
4 print("Einkaufsliste")
5 trennlinie()
6 print("Milch, Brot, Käse")
7 trennlinie()
```

```
Einkaufsliste
--------------------
Milch, Brot, Käse
--------------------
```

- **Zeile 1:** `def` (englisch *define*, „festlegen") beginnt eine **Funktionsdefinition**: Name, runde Klammern, **Doppelpunkt**. Die Klammern sind leer, weil die Funktion keine Werte braucht.
- **Zeile 2:** Die eingerückten Zeilen unter `def` bilden den **Funktionsrumpf**, wie bei `if` und `for`.

- **Zeile 4:** Hier beginnt das **Hauptprogramm**. Beim Lesen von Zeile 1 und 2 hat Python noch **nichts ausgegeben**, es hat sich nur gemerkt, was `trennlinie` bedeutet.
- **Zeilen 5 und 7:** `trennlinie()` ist ein **Funktionsaufruf**. Erst jetzt wird der Rumpf ausgeführt, und das zweimal, obwohl die Linie nur einmal im Programm steht.

Eine Funktion ist wie ein **Rezept**: Die Definition schreibt das Rezept auf, davon wird noch niemand satt. Erst beim Aufruf wird gekocht, so oft man möchte.

> 📌 **Merksatz:** Eine **Funktion** fasst eine Folge von Anweisungen unter einem Namen zusammen und macht sie wiederverwendbar. **Definition:** `def funktionsname(parameter1, parameter2):` gefolgt vom eingerückten **Funktionsrumpf**.

⚠️ Stolpersteine

Führe beide Zellen aus und lies die letzte Zeile der Fehlermeldung.

```
def trennlinie():
print("--------------------")
```

```
  File "<Zelle 2>", line 2
    print("--------------------")
    ^^^^^
IndentationError: expected an indented block after function definition on line 1
```

```
1 trennlinie2()
2
3 def trennlinie2():
4     print("--------------------")
```

```
Traceback (most recent call last):
  File "<Zelle 3>", line 1, in <module>
NameError: name 'trennlinie2' is not defined
```

1. `IndentationError: expected an indented block after function definition on line 1`: Nach `def` wird ein eingerückter Block erwartet. Fehlt der Doppelpunkt, kommt `SyntaxError: expected ':'`.
2. `NameError: name 'trennlinie2' is not defined`: Python liest von oben nach unten. Eine Funktion muss **definiert sein, bevor** sie aufgerufen wird. Schreibe die Definitionen deshalb an den Anfang. Und vergiss beim Aufruf die Klammern nicht.

2 Parameter: der Funktion Werte mitgeben

Eine Begrüßung soll jedes Mal einen anderen Namen enthalten. Dafür schreibst du in die Klammern einen **Parameter**.

```
1 def gruesse(name):
2     print("Hallo, " + name + "!")
3
4 gruesse("Alex")
5 gruesse("Sam")
```

```
Hallo, Alex!
Hallo, Sam!
```

- **Zeile 1:** `name` ist ein **Parameter**: eine Variable der Funktion, die ihren Wert erst beim Aufruf bekommt, wie eine noch leere Schachtel.
- **Zeile 4:** `"Alex"` ist das **Argument**. Python legt es in den Parameter, als stünde dort `name = "Alex"`, und führt dann den Rumpf aus.

Mehrere Parameter

Mehrere Parameter werden durch Kommas getrennt. Die Argumente werden **der Reihe nach** verteilt. 🔮 **Vorhersage:** Was gibt die letzte Zeile aus?

```
1 def gruesse(gruss, name):
2     print(gruss + ", " + name + "!")
3
4 gruesse("Hallo", "Alex")
5 gruesse("Servus", "Sam")
6 gruesse("Sam", "Moin")
```

```
Hallo, Alex!
Servus, Sam!
Sam, Moin!
```

In Zeile 6 sind die Argumente vertauscht. Python bemerkt das nicht: `gruss` bekommt `"Sam"`, `name` bekommt `"Moin"`.

> 📌 **Merksatz: Parameter** sind die „Eingabewerte“ einer Funktion. Beim Aufruf werden ihnen konkrete **Argumente** übergeben, und zwar in derselben Reihenfolge.

⚠️ Stolperstein: falsche Anzahl

Führe die Zelle aus. Was passiert bei `gruesse("Alex", "Sam")`?

```
1 def gruesse(name):
2     print("Hallo, " + name + "!")
3
4 gruesse()
```

```
Traceback (most recent call last):
  File "<Zelle 6>", line 4, in <module>
TypeError: gruesse() missing 1 required positional argument: 'name'
```

`TypeError: gruesse() missing 1 required positional argument: 'name'` heißt: „Der Funktion fehlt 1 benötigtes Argument.“ Bei zu vielen Argumenten: `takes 1 positional argument but 2 were given`.

🧪 **Probiere es aus:** Rufe `gruesse` mit deinem eigenen Namen auf. Was passiert bei `gruesse(15)`? Erkläre die Fehlermeldung mit dem, was du über das Verketten mit `+` weißt.

3 Rückgabewert: die Funktion liefert ein Ergebnis

`gruesse` gibt etwas auf dem Bildschirm aus, danach ist das Ergebnis für das Programm verloren. Bei `round` ist das anders: `zahl = round(3.14159, 2)` **gibt** `3.14` **zurück**, und es landet in `zahl`. Das geht auch mit eigenen Funktionen, mit `return`.

```python
def quadrat(zahl):
    ergebnis = zahl * zahl
    return ergebnis

print(quadrat(4))
print(quadrat(7))

x = quadrat(10)
print("x ist jetzt:", x)
```

```
16
49
x ist jetzt: 100
```

- **Zeile 3:** `return ergebnis` gibt den Wert an die Stelle zurück, an der die Funktion aufgerufen wurde: der **Rückgabewert**.
- **Zeile 5:** Von innen nach außen lesen: `quadrat(4)` liefert `16`, der Aufruf wird durch diesen Wert ersetzt. Es steht gewissermaßen `print(16)` da.
- **Zeile 8:** Diesmal wird der Rückgabewert `100` in `x` gespeichert.

Eine Funktion mit Parametern und Rückgabewert ist wie eine **Maschine**: Oben wirft man das Argument 4 hinein, die Maschine arbeitet, unten kommt 16 heraus.

Weil ein Aufruf wie `quadrat(3)` am Ende für einen Wert steht, darf er überall stehen, wo ein Wert stehen darf: in Rechnungen, Bedingungen oder als Argument. Mit `return` **endet die Funktion sofort**. Ein Rückgabewert muss auch keine Zahl sein.

```python
def quadrat(zahl):
    return zahl * zahl

summe = quadrat(3) + quadrat(4)
print("3*3 + 4*4 =", summe)
if quadrat(5) > 20:
    print("5*5 ist größer als 20.")

def begruessung(name, gruss):
    return gruss + ", " + name + "!"

print(begruessung("Alex", "Hallo"))
```

```
3*3 + 4*4 = 25
5*5 ist größer als 20.
Hallo, Alex!
```

> 📌 **Merksatz:** Mit `return wert` gibt eine Funktion einen **Rückgabewert** an die aufrufende Stelle zurück und beendet die Funktion. Der **Funktionsaufruf** (z. B. `ergebnis = quadrat(4)`) ist zu unterscheiden von der **Funktionsdefinition** (`def quadrat(zahl):` mit Rumpf).

🔮 Übung 1 – Vorhersage

Lies das Programm, **ohne** es auszuführen. Schreibe auf, was es ausgibt. Überprüfe danach.

```python
def verdopple(zahl):
    return 2 * zahl

a = verdopple(3)
b = verdopple(a) + 1
print(a, b)
print(verdopple(verdopple(5)))
```

```
6 13
20
```

Meine Vorhersage:

…

4 Was beim Aufruf passiert

Ein Funktionsaufruf unterbricht die Reihenfolge von oben nach unten: Python **springt** in die Funktion, führt den Rumpf aus und **springt** mit dem Rückgabewert **zurück**.

```python
def quadrat(zahl):
    ergebnis = zahl * zahl
    return ergebnis

x = quadrat(10)
y = quadrat(3) + 1
print(x, y)
```

```
100 10
```

1. **Zeile 1:** Python merkt sich die Funktion. Zeilen 2 und 3 werden jetzt **nicht** ausgeführt.
2. **Zeile 5:** Sprung in die Funktion mit `zahl = 10`. Zeile 2 rechnet `ergebnis = 100`, Zeile 3 gibt `100` zurück. `x` bekommt `100`.
3. **Zeile 6:** Sprung mit `zahl = 3`, Rückgabewert `9`. Zurück in Zeile 6: `9 + 1 = 10` wird in `y` gespeichert.
4. **Zeile 7:** Ausgabe von `x` und `y`.

Ablaufprotokoll **innerhalb der Funktion**:

Schritt	Zeile	Anweisung	`zahl`	`ergebnis`
1	2	`ergebnis = zahl * zahl`	`10`	`100`
2	3	`return ergebnis`	`10`	`100`
3	2	`ergebnis = zahl * zahl`	`3`	`9`
4	3	`return ergebnis`	`3`	`9`

Variablen in der Funktion bleiben in der Funktion

`zahl` und `ergebnis` gibt es nur, solange die Funktion läuft. Man nennt sie **lokale Variablen**. Aus der Maschine kommt nur heraus, was mit `return` zurückgegeben wird. Führe die Zelle aus.

```python
def quadrat(zahl):
    ergebnis = zahl * zahl
    return ergebnis

x = quadrat(10)
print(ergebnis)
```

```
Traceback (most recent call last):
  File "<Zelle 11>", line 6, in <module>
NameError: name 'ergebnis' is not defined
```

5 `print` oder `return`?

Der häufigste Fehler mit Funktionen. 🔮 **Vorhersage:** Was steht in `a`, was in `b`?

```python
def quadrat_mit_print(zahl):
    print(zahl * zahl)

def quadrat_mit_return(zahl):
    return zahl * zahl

a = quadrat_mit_print(4)
b = quadrat_mit_return(4)
print("a ist", a)
print("b ist", b)
```

```
16
a ist None
b ist 16
```

- **Zeile 7:** `quadrat_mit_print(4)` zeigt sofort `16` an, hat aber kein `return`. Darum bekommt `a` den besonderen Wert **`None`** (englisch für „nichts“).
- **Zeile 8:** `quadrat_mit_return(4)` zeigt nichts an, liefert aber `16` zurück.

Will man weiterrechnen, bricht die `print`-Variante sogar ab:

```python
def quadrat_mit_print(zahl):
    print(zahl * zahl)

summe = quadrat_mit_print(3) + quadrat_mit_print(4)
```

```
9
16
```

```
Traceback (most recent call last):
  File "<Zelle 13>", line 4, in <module>
TypeError: unsupported operand type(s) for +: 'NoneType' and 'NoneType'
```

`TypeError: unsupported operand type(s) for +: 'NoneType' and 'NoneType'`: `None + None` kann Python nicht rechnen. Taucht `NoneType` in einer Fehlermeldung auf, fehlt sehr oft ein `return`.

	`print(...)` in der Funktion	`return ...`
Was passiert?	Der Wert wird auf dem Bildschirm angezeigt.	Der Wert wird an die Stelle des Aufrufs zurückgegeben.
Wer bekommt den Wert?	die Person vor dem Bildschirm	das Programm
Was liefert der Aufruf?	`None`	den Wert hinter `return`
Weiterrechnen möglich?	nein	ja

📌 **Merksatz:** Eine Funktion ohne `return` liefert `None` zurück. `print` zeigt einen Wert nur an. Soll das Programm mit dem Ergebnis weiterarbeiten, muss die Funktion es mit `return` zurückgeben.

💡 **Faustregel:** Eine Funktion, die etwas **berechnet**, gibt ihr Ergebnis mit `return` zurück. Das **Ausgeben** mit `print` erledigt das Hauptprogramm.

🔍 Übung 2 – print oder return?

Das Programm soll Minuten in Sekunden umrechnen. Es läuft ohne Fehlermeldung, aber die Ausgabe ist seltsam.

1. Gib an, was das Programm ausgibt.
2. Erkläre, warum die Ausgabe nicht stimmt.
3. Verbessere die Funktion.

```
def minuten_zu_sekunden(minuten):
    print(minuten * 60)

dauer = minuten_zu_sekunden(5)
print("Das sind", dauer, "Sekunden.")
```

```
300
Das sind None Sekunden.
```

Ausgabe und Erklärung:

...

6 Funktionen testen

Eine Funktion kann man beliebig oft mit verschiedenen Argumenten aufrufen. Das ist praktisch zum **Testen**:

1. **Vorher** von Hand ausrechnen, was herauskommen muss.
2. Die Funktion mit diesen Werten aufrufen.
3. Vergleichen. Weicht ein Ergebnis ab, steckt ein Fehler in der Funktion oder in deiner Rechnung.

Teste auch Sonderfälle wie `0`, negative Zahlen oder Kommazahlen. Dort verstecken sich Fehler am häufigsten.

```python
def quadrat(zahl):
    return zahl * zahl

print(quadrat(4), "erwartet: 16")
print(quadrat(0), "erwartet: 0")
print(quadrat(-3), "erwartet: 9")
print(quadrat(1.5), "erwartet: 2.25")
print(quadrat(-3) == 9)
```

```
16 erwartet: 16
0 erwartet: 0
9 erwartet: 9
2.25 erwartet: 2.25
True
```

📝 Übungen: Schritt für Schritt selbstständig

Die Übungen werden schrittweise schwieriger. Das **Musterbeispiel** ist ganz vorgemacht. Bei Übung 3 füllst du nur Lücken `___` aus, bei Übung 4 steht nur noch das Gerüst, die Zusatzaufgabe schreibst du allein. Gehe jedes Mal in denselben vier Schritten vor:

Schritt	Frage
1	**Welche Werte** braucht die Funktion? → Parameter
2	**Was soll herauskommen?** → Rückgabewert
3	**Wie rechnet man es aus?** → Rumpf
4	**Stimmt es?** → Testen mit vorher ausgerechneten Werten

🧩 Musterbeispiel (vorgemacht) – Fläche eines Dreiecks

Aufgabe: Schreibe eine Funktion `dreieck_flaeche(grundseite, hoehe)`, die die Fläche eines Dreiecks zurückgibt. Formel: A = g · h / 2.

1. **Welche Werte?** Die Grundseite und die Höhe. Also zwei Parameter: `grundseite` und `hoehe`.
2. **Was soll herauskommen?** Die Fläche, also eine Zahl. Sie wird mit `return` zurückgegeben, nicht mit `print` ausgegeben.
3. **Wie rechnet man?** Die Formel wird fast wörtlich übernommen: `grundseite * hoehe / 2`.
4. **Stimmt es?** Vorher von Hand: g = 4, h = 3 ergibt 4 · 3 / 2 = 6. g = 10, h = 5 ergibt 25. Sonderfall: h = 0 ergibt 0.

Lies den Code Zeile für Zeile und führe ihn aus.

```python
# Schritt 1 und 2: Name, Parameter und Rückgabewert festlegen
def dreieck_flaeche(grundseite, hoehe):
    # Schritt 3: rechnen und das Ergebnis zurückgeben
    flaeche = grundseite * hoehe / 2
    return flaeche

# Schritt 4: testen mit vorher ausgerechneten Werten
print(dreieck_flaeche(4, 3), "erwartet: 6")
print(dreieck_flaeche(10, 5), "erwartet: 25")
print(dreieck_flaeche(7, 0), "erwartet: 0")
```

```
6.0 erwartet: 6
25.0 erwartet: 25
0.0 erwartet: 0
```

Weil `/` immer eine Kommazahl liefert, erscheint `6.0` statt `6`. Das ist richtig.

📝 Übung 3 – Celsius in Fahrenheit (Lücken füllen)

In den USA misst man Temperaturen in Grad Fahrenheit: °F = °C · 9/5 + 32. Die Funktion `celsius_zu_fahrenheit(grad_celsius)` soll den Wert in Fahrenheit **zurückgeben**.

Gehe wie im Musterbeispiel vor:

1. **Welche Werte?** Eine Temperatur in Grad Celsius, der Parameter heißt `grad_celsius`.
2. **Was soll herauskommen?** Die Temperatur in Fahrenheit.
3. **Wie rechnet man?** Übersetze die Formel in Python. Das Malzeichen ist `*`.
4. **Stimmt es?** Die Testaufrufe stehen schon da: Wasser gefriert bei 0 °C = 32 °F und siedet bei 100 °C = 212 °F.

Ersetze jedes `___` durch den richtigen Ausdruck. Solange noch ein `___` dasteht, meldet Python `NameError: name '___' is not defined`.

Lückencode

```
def celsius_zu_fahrenheit(grad_celsius):
    fahrenheit = ___
    return ___

print(celsius_zu_fahrenheit(0), "erwartet: 32")
print(celsius_zu_fahrenheit(100), "erwartet: 212")
print(celsius_zu_fahrenheit(20), "erwartet: 68")
```

📝 Übung 4 – Maximum von zwei Zahlen (mit Gerüst)

Schreibe eine Funktion `maximum(a, b)`, die die größere von zwei Zahlen zurückgibt. Verwende **nicht** `max()`, sondern eine Verzweigung.

1. **Welche Werte?** Zwei Zahlen, `a` und `b`.
2. **Was soll herauskommen?** Die größere der beiden Zahlen.
3. **Wie rechnet man?** Vergleiche mit `if a > b:` und gib im passenden Zweig `a` oder `b` zurück. Eine Funktion darf mehrere `return` haben.
4. **Stimmt es?** Rechne vorher aus, was bei den Testaufrufen herauskommen muss, und trage es hinter `erwartet:` ein. Teste auch mit zwei gleichen Zahlen.

Das Gerüst gibt die Kopfzeile und die Kommentare vor. Ersetze die Kommentare durch Programmzeilen.

```
def maximum(a, b):
    # wenn a größer als b ist:
    #     a zurückgeben
    # sonst:
    #     b zurückgeben
    pass

print(maximum(3, 9), "erwartet: ")
print(maximum(17, 5), "erwartet: ")
print(maximum(4, 4), "erwartet: ")
```

```
None erwartet:
None erwartet:
None erwartet:
```

🚀 Zusatzaufgabe – Gerade Zahlen (selbstständig)

1. Schreibe eine Funktion `ist_gerade(zahl)`, die `True` zurückgibt, wenn die Zahl gerade ist, sonst `False`. Der Rückgabewert ist also vom Typ `bool`.
2. Nutze die Funktion in einer `for`-Schleife, um alle geraden Zahlen von 1 bis 20 auszugeben.

Gehe wieder in den vier Schritten vor. Hinweis: Eine Zahl ist gerade, wenn beim Teilen durch 2 der Rest `0` bleibt (`%`).

📌 Das Wichtigste in Kürze

- Eine **Funktion** fasst Anweisungen unter einem Namen zusammen: `def name(parameter):` mit eingerücktem Rumpf. Ausgeführt wird sie erst beim **Aufruf**.
- **Parameter** bekommen beim Aufruf die **Argumente**, der Reihe nach.
- `return wert` gibt einen **Rückgabewert** zurück und beendet die Funktion. Ohne `return` liefert sie `None`.
- `print` zeigt nur an, `return` gibt dem Programm den Wert. Wer berechnet, gibt zurück.
- Variablen in der Funktion sind **lokal**.
- Testen: vorher das erwartete Ergebnis ausrechnen, auch Sonderfälle wie `0` und negative Zahlen prüfen.

Zum Weiterüben: `DS08_Weitere_Uebungen.ipynb`.