1.8 Funktionen: Programme aus Bausteinen

Leitidee

Du hast schon viele Funktionen benutzt, ohne viel darüber nachzudenken: `print(...)`, `input(...)`, `int(...)`, `round(...)` oder `random.randint(...)`. Du schreibst ihren Namen, gibst ihnen in Klammern Werte mit und bekommst ein Ergebnis. Wie sie innen arbeiten, musst du dafür nicht wissen. In diesem Abschnitt schreibst du **eigene Funktionen**. So schreibst du eine Rechnung, die du oft brauchst, nur einmal auf und benutzt sie dann so oft du willst. Die Idee kennst du aus Mathe: Die Funktion $f(x) = x^2$ macht aus 4 die 16 und aus 7 die 49.

Lernziele

Nach diesem Abschnitt kannst du

- 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.

1.8.1 Eine eigene Funktion definieren

Stell dir vor, dein Programm soll an mehreren Stellen eine Trennlinie ausgeben. Statt die Linie jedes Mal neu 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()
```

AUSGABE

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

- **Zeile 1:** Das Wort `def` (von englisch *define*, „festlegen“) beginnt eine **Funktionsdefinition**. Danach folgt der Name der Funktion, hier `trennlinie`, dann runde Klammern und ein **Doppelpunkt**. Die Klammern sind leer, weil diese Funktion keine Werte braucht.
- **Zeile 2:** Diese Zeile ist um 4 Leerzeichen **eingerückt**. Alle eingerückten Zeilen unter `def` bilden den **Funktionsrumpf**. Das kennst du von `if` und `for`: Was eingerückt ist, gehört dazu.
- **Zeile 3:** Die leere Zeile ist nicht nötig, macht das Programm aber übersichtlicher. Die Funktion endet, wo die Einrückung endet.
- **Zeile 4:** Hier beginnt das eigentliche Programm, das **Hauptprogramm**. Wichtig: Beim Lesen der Zeilen 1 und 2 hat Python noch **nichts ausgegeben**. Es hat sich nur gemerkt, was `trennlinie` bedeuten soll.
- **Zeile 5:** `trennlinie()` ist ein **Funktionsaufruf**. Jetzt führt Python den Funktionsrumpf aus und gibt die Linie aus.

- **Zeilen 6–7:** Noch eine Ausgabe und noch ein Aufruf. Die Linie erscheint ein zweites Mal, obwohl sie nur einmal im Programm steht.

Du kannst dir eine Funktion wie ein **Rezept** vorstellen. Die Definition schreibt das Rezept auf. Davon wird noch niemand satt. Erst beim Aufruf wird nach dem Rezept gekocht – und das so oft, wie man möchte.

Definition: Funktion

Eine **Funktion** ist eine Folge von Anweisungen, die unter einem Namen zusammengefasst ist. Über diesen Namen kann man sie beliebig oft aufrufen, also ausführen lassen.

Merke

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

Was `parameter1` und `parameter2` in den Klammern bedeuten, lernst du gleich im Abschnitt über Parameter. Bei `trennlinie()` bleiben die Klammern leer.

Stolperstein

Vergisst du die Einrückung, meldet Python einen **IndentationError** („Einrückungsfehler“):

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

FEHLERMELDUNG

```
  File "programm.py", line 2
    print("--------------------")
    ^^^^^
IndentationError: expected an indented block after function definition on line 1
```

Die letzte Zeile heißt: „Nach der Funktionsdefinition in Zeile 1 wurde ein eingerückter Block erwartet.“ Gefunden hat Python das Problem in Zeile 2 (*line 2*). Rücke diese Zeile mit 4 Leerzeichen (oder der Tabulatortaste) ein. Fehlt der Doppelpunkt am Ende von Zeile 1, kommt ein `SyntaxError: expected ':'`.

Stolperstein

Eine Funktion muss **definiert sein, bevor** sie aufgerufen wird, denn Python liest das Programm von oben nach unten. Steht `trennlinie()` vor dem `def`, meldet Python `NameError: name 'trennlinie' is not defined`. Schreibe deshalb alle Funktionsdefinitionen an den Anfang des Programms. Und vergiss beim Aufruf die Klammern nicht: `trennlinie` ohne `()` ruft die Funktion nicht auf.

1.8.2 Parameter: der Funktion Werte mitgeben

Die Trennlinie sieht immer gleich aus. Oft soll eine Funktion aber mit **verschiedenen Werten** arbeiten. Eine Begrüßung zum Beispiel soll jedes Mal einen anderen Namen enthalten. Dafür schreibst du in die Klammern hinter dem Funktionsnamen einen **Parameter**:

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

AUSGABE

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

- **Zeile 1:** In den Klammern steht jetzt `name`. Das ist ein **Parameter**: eine Variable der Funktion, die ihren Wert erst beim Aufruf bekommt. Er ist wie ein Platzhalter, eine noch leere Schachtel.
- **Zeile 2:** Im Funktionsrumpf benutzt du `name` wie jede andere Variable.
- **Zeile 4:** Beim Aufruf steht in den Klammern der Wert `"Alex"`. Diesen Wert nennt man **Argument**. Python legt ihn in den Parameter, als stünde dort `name = "Alex"`, und führt dann den Rumpf aus.
- **Zeile 5:** Beim zweiten Aufruf bekommt `name` den Wert `"Sam"`.

Eine Funktion darf auch **mehrere Parameter** haben. Sie werden durch Kommas getrennt. Beim Aufruf werden die Argumente **der Reihe nach** auf die Parameter verteilt: das erste Argument in den ersten Parameter, das zweite in den zweiten.

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

AUSGABE

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

In Zeile 6 sind die Argumente vertauscht. Python bemerkt das nicht: `gruss` bekommt `"Sam"`, `name` bekommt `"Moin"`. Achte deshalb beim Aufruf auf die Reihenfolge.

Definition: Parameter und Argument

Ein **Parameter** ist eine Variable in der Funktionsdefinition, die beim Aufruf einen Wert bekommt. Der Wert, der beim Aufruf übergeben wird, heißt **Argument**.

Merke

Parameter sind die „Eingabewerte“ einer Funktion – beim Aufruf werden ihnen konkrete **Argumente** übergeben, und zwar in derselben Reihenfolge.

Stolperstein

Die Anzahl der Argumente muss zur Anzahl der Parameter passen:

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

FEHLERMELDUNG

```
TypeError: gruesse() missing 1 required positional argument: 'name'
```

Übersetzt: „Der Funktion `gruesse()` fehlt 1 benötigtes Argument: `name`." Gibst du zu viele Argumente an, z. B. `gruesse("Alex", "Sam")`, heißt es: `takes 1 positional argument but 2 were given` – „nimmt 1 Argument, aber 2 wurden übergeben".

Versuch: Probiere es aus

Rufe `gruesse` mit deinem eigenen Namen auf. Was passiert bei `gruesse(15)`? Lies die Fehlermeldung und erkläre sie mit dem, was du über das Verketten mit `+` weißt.

1.8.3 Rückgabewert: die Funktion liefert ein Ergebnis

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

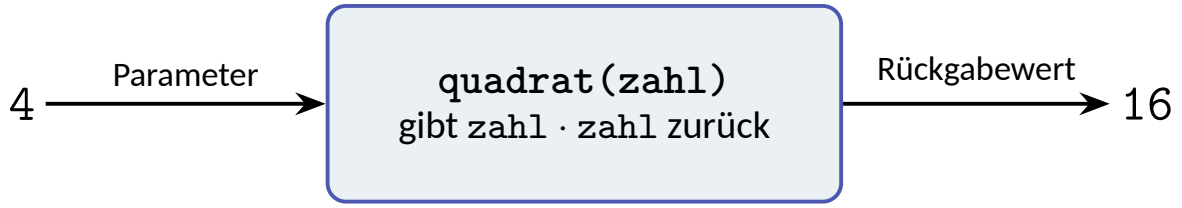
```
1 def quadrat(zahl):
2     ergebnis = zahl * zahl
3     return ergebnis
4
5 print(quadrat(4))
6 print(quadrat(7))
7
8 x = quadrat(10)
9 print("x ist jetzt:", x)
```

AUSGABE

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

- **Zeile 1:** Die Funktion `quadrat` hat einen Parameter `zahl`.
- **Zeile 2:** Im Rumpf wird `zahl * zahl` ausgerechnet und in `ergebnis` gespeichert.
- **Zeile 3:** `return ergebnis` gibt den Wert von `ergebnis` an die Stelle zurück, an der die Funktion aufgerufen wurde. Diesen Wert nennt man **Rückgabewert**.
- **Zeile 5:** Lies die Zeile von innen nach außen. Zuerst wird `quadrat(4)` ausgeführt und liefert `16`. Der Aufruf wird dann durch diesen Wert ersetzt – es steht also gewissermaßen `print(16)` da.
- **Zeile 6:** Genauso: `quadrat(7)` liefert `49`, das ausgegeben wird.
- **Zeile 8:** Diesmal wird der Rückgabewert `100` in der Variablen `x` gespeichert.
- **Zeile 9:** `x` wird ausgegeben.

Eine Funktion mit Parametern und Rückgabewert funktioniert wie eine **Maschine**: Oben wirft man etwas hinein, die Maschine arbeitet, und unten kommt ein Ergebnis heraus.



Die Funktion `quadrat` als Maschine: Das Argument 4 geht als Parameter hinein, der Rückgabewert 16 kommt heraus

Definition: Rückgabewert

Der **Rückgabewert** ist das Ergebnis, das eine Funktion mit `return` an die Stelle zurückgibt, an der sie aufgerufen wurde.

Weil ein Aufruf wie `quadrat(3)` am Ende einfach für einen Wert steht, kannst du ihn überall einsetzen, wo ein Wert stehen darf – in Rechnungen, Bedingungen oder als Argument einer anderen Funktion:

```
1 def quadrat(zahl):
2     ergebnis = zahl * zahl
3     return ergebnis
4
5 summe = quadrat(3) + quadrat(4)
6 print("3*3 + 4*4 =", summe)
7 if quadrat(5) > 20:
8     print("5*5 ist größer als 20.")
```

AUSGABE

```
3*3 + 4*4 = 25
5*5 ist größer als 20.
```

Wichtig: Mit `return` **endet die Funktion sofort**. Steht danach im Rumpf noch etwas, wird es nicht mehr ausgeführt.

Ein Rückgabewert muss keine Zahl sein. Diese Funktion setzt einen Gruß zusammen und gibt ihn als Zeichenkette zurück:

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

AUSGABE

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

Merke

- 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).

Arbeitsauftrag 92: Vorhersage

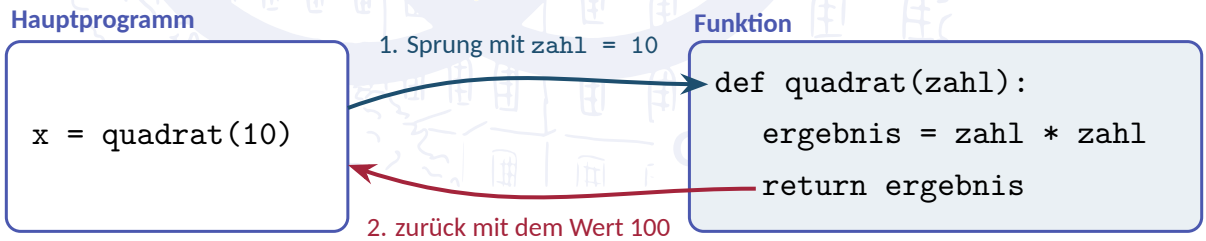
Lies das Programm, **ohne** es auszuführen. Schreibe auf, was es ausgibt. Überprüfe deine Vorhersage danach am Computer.

```
1 def verdopple(zahl):
2     return 2 * zahl
3
4 a = verdopple(3)
5 b = verdopple(a) + 1
6 print(a, b)
7 print(verdopple(verdopple(5)))
```

Meine Vorhersage:

1.8.4 Was beim Aufruf passiert

Ohne Funktionen läuft ein Programm einfach von oben nach unten. Ein Funktionsaufruf unterbricht diese Reihenfolge: Python **springt** in die Funktion, führt ihren Rumpf aus und **springt** mit dem Rückgabewert **zurück** an die Stelle des Aufrufs. Dort geht es weiter.



Beim Aufruf `quadrat(10)` springt Python in die Funktion, der Parameter `zahl` bekommt den Wert 10. Mit `return` springt Python zurück, und der Aufruf wird durch den Rückgabewert 100 ersetzt

Verfolge den Ablauf an diesem Programm:

```
1 def quadrat(zahl):
2     ergebnis = zahl * zahl
3     return ergebnis
4
5 x = quadrat(10)
6 y = quadrat(3) + 1
7 print(x, y)
```

AUSGABE
```
100 10
```

1. **Zeile 1:** Python liest die Definition und merkt sich die Funktion. Die Zeilen 2 und 3 werden jetzt **nicht** ausgeführt.
2. **Zeile 5:** Aufruf `quadrat(10)`. Python springt in die Funktion, der Parameter `zahl` bekommt den Wert `10`. Zeile 2 rechnet `ergebnis = 100`, Zeile 3 gibt `100` zurück. Zurück in Zeile 5 bekommt `x` den Wert `100`.
3. **Zeile 6:** Aufruf `quadrat(3)`. Wieder geht es in die Funktion, diesmal mit `zahl = 3`. Der Rückgabewert ist `9`. Zurück in Zeile 6 wird `9 + 1 = 10` gerechnet und in `y` gespeichert.
4. **Zeile 7:** Ausgabe von `x` und `y`.

Im Ablaufprotokoll des Hauptprogramms sieht es so aus, als würde Zeile 5 in einem einzigen Schritt erledigt:

Schritt	Zeile	Anweisung	x	y
1	1	`def quadrat(zahl):`	–	–
2	5	`x = quadrat(10)`	100	–
3	6	`y = quadrat(3) + 1`	100	10
4	7	`print(x, y)`	100	10

In Wirklichkeit läuft bei jedem Aufruf die Funktion einmal ab. Das Ablaufprotokoll **innerhalb der Funktion** zeigt die beiden Durchläufe:

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

Die Variablen `zahl` und `ergebnis` gibt es nur, solange die Funktion läuft. Man nennt sie **lokale Variablen**. Von außen kann man nicht auf sie zugreifen: Schreibt man im Programm oben am Ende noch `print(ergebnis)`, meldet Python `NameError: name 'ergebnis' is not defined`.

Aus der Maschine kommt nur heraus, was mit `return` zurückgegeben wird. Genau deshalb braucht man `return`: Mit dem, was du bisher kennst, ist es der einzige Weg, auf dem das Ergebnis aus der Funktion zum Hauptprogramm gelangt.

1.8.5 `print` oder `return`?

Der häufigste Fehler mit Funktionen: `print` statt `return` zu schreiben. Auf den ersten Blick sieht beides gleich aus, denn in beiden Fällen erscheint am Ende eine Zahl. Vergleiche die beiden Funktionen:

```
1  def quadrat_mit_print(zahl):
2      print(zahl * zahl)
3
4  def quadrat_mit_return(zahl):
5      return zahl * zahl
6
7  a = quadrat_mit_print(4)
8  b = quadrat_mit_return(4)
9  print("a ist", a)
10 print("b ist", b)
```

AUSGABE

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

- **Zeile 7:** `quadrat_mit_print(4)` gibt sofort `16` auf dem Bildschirm aus – das ist die erste Ausgabezeile. Die Funktion hat aber kein `return`. Deshalb gibt sie dem Hauptprogramm nichts zurück, und `a` bekommt den besonderen Wert **`None`** (englisch für „nichts“).
- **Zeile 8:** `quadrat_mit_return(4)` gibt nichts aus. Sie liefert aber `16` zurück, und dieser Wert landet in `b`.
- **Zeilen 9–10:** Jetzt sieht man den Unterschied: In `a` steht `None`, in `b` steht `16`.

Will man mit dem Ergebnis weiterrechnen, bricht das Programm mit der `print`-Variante sogar ab:

```
1  def quadrat_mit_print(zahl):
2      print(zahl * zahl)
3
4  summe = quadrat_mit_print(3) + quadrat_mit_print(4)
```

AUSGABE

```
9
16
```

FEHLERMELDUNG

```
TypeError: unsupported operand type(s) for +: 'NoneType' and 'NoneType'
```

Die Funktion hat zwar `9` und `16` angezeigt, aber beide Aufrufe liefern `None`. Und `None + None` kann Python nicht rechnen. Die Meldung heißt: „Der Operator `+` ist für die Typen `NoneType` und `NoneType` nicht möglich.“ Wenn in einer Fehlermeldung `NoneType` auftaucht, 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

Merke

- Eine Funktion ohne `return` liefert `None` zurück.
- `print` in einer Funktion zeigt einen Wert nur an. Soll das Programm mit dem Ergebnis weiterarbeiten, muss die Funktion es mit `return` zurückgeben.

Tipp

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

Arbeitsauftrag 93: print oder return?

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

```
1 def minuten_zu_sekunden(minuten):
2     print(minuten * 60)
3
4 dauer = minuten_zu_sekunden(5)
5 print("Das sind", dauer, "Sekunden.")
```

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

Ausgabe und Erklärung:

Gymnasium Ottweiler

1.8.6 Funktionen testen

Eine Funktion kann man beliebig oft mit verschiedenen Argumenten aufrufen. Das ist praktisch, um sie zu **testen**: Du rufst sie mit Werten auf, bei denen du das richtige Ergebnis schon kennst, und vergleichst. So gehst du vor:

1. **Vorher** von Hand ausrechnen, was herauskommen muss (erwartetes Ergebnis).
2. Die Funktion mit diesen Werten aufrufen und das tatsächliche Ergebnis ausgeben.
3. Vergleichen. Weicht ein Ergebnis ab, steckt ein Fehler in der Funktion – oder in deiner Rechnung.

Wähle die Testwerte mit Bedacht: einen ganz normalen Wert, aber auch Sonderfälle wie `0`, negative Zahlen oder Kommazahlen. Genau dort verstecken sich Fehler am häufigsten.

```
1 def quadrat(zahl):
2     return zahl * zahl
3
4 print(quadrat(4), "erwartet: 16")
5 print(quadrat(0), "erwartet: 0")
6 print(quadrat(-3), "erwartet: 9")
7 print(quadrat(1.5), "erwartet: 2.25")
```

AUSGABE

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

Alle vier Tests stimmen. Den Vergleich kann Python auch selbst übernehmen: `print(quadrat(-3) == 9)` gibt `True` aus, wenn der Test stimmt, und sonst `False`.

So kannst du es sagen

So beschreibst du, was eine Funktion tut. Als Beispiel dient 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.“*
- *„Eine Funktion liefert je nach Parameter ein spezifisches Ergebnis, zum Beispiel 17 für die Eingabe 4.“*
- *„Der Funktion wird der Wert 4 übergeben. Die Funktion berechnet $4 \cdot 4 + 1$ und gibt 17 zurück.“*

1.8.7 Übungen

So gehst du beim Schreiben einer Funktion vor: Überlege zuerst, **welche Werte** die Funktion braucht (Parameter) und **was sie zurückgeben** soll (Rückgabewert). Schreibe dann den Rumpf. Teste die Funktion zum Schluss mit mehreren Aufrufen, deren Ergebnis du vorher ausgerechnet hast.

Arbeitsauftrag 94: Celsius in Fahrenheit

In den USA misst man Temperaturen in Grad Fahrenheit (°F). Die Umrechnung lautet: °F = °C · 9/5 + 32.

1. Implementiere eine Funktion `celsius_zu_fahrenheit(grad_celsius)`, die eine Temperatur in Grad Celsius entgegennimmt und den Wert in Fahrenheit **zurückgibt**.

2. Teste sie mit mehreren Werten. Zur Kontrolle: Wasser gefriert bei 0 °C = 32 °F und siedet bei 100 °C = 212 °F.

Arbeitsauftrag 95: Maximum von zwei Zahlen

Implementiere eine Funktion `maximum(a, b)`, die die größere von zwei Zahlen zurückgibt. Verwende dafür **nicht** die eingebaute Funktion `max()`, sondern eine Verzweigung. Teste deine Funktion – auch mit zwei gleichen Zahlen.

Arbeitsauftrag 96 (Zusatz): Gerade Zahlen

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

1.8.8 Weitere Übungen

Hier kannst du das Gelernte weiter üben. Die Punkte im Titel zeigen, wie schwer eine Aufgabe ist: ●○○ leicht, ●●○ mittel, ●●● knifflig. Fang mit den leichten an.

Arbeitsauftrag 97: Fachbegriffe am Beispiel ●○○

Am Kiosk kostet jede Pfandflasche 25 Cent Pfand zusätzlich zum Preis. Lies das Programm:

```
1 def preis_mit_pfand(preis, flaschen):
2     return preis + flaschen * 0.25
3
4 gesamt = preis_mit_pfand(3.5, 6)
5 print(gesamt)
```

Nenne zu diesem Programm:

1. den Namen der Funktion,
2. die Parameter,
3. die Argumente beim Aufruf,
4. die Zeile, in der die Funktion aufgerufen wird,
5. den Rückgabewert für diesen Aufruf.

Meine Antworten:

Arbeitsauftrag 98: Rückgabewerte vorhersagen ●○○

Gegeben ist die Funktion `rechne`:

```
def rechne(x):
    return 3 * x - 1
```

Bestimme **ohne Computer**, welchen Wert die Aufrufe zurückgeben. Prüfe danach am Computer, z. B. mit `print(rechne(2))`.

Aufruf	Rückgabewert
`rechne(2)`	
`rechne(0)`	
`rechne(-1)`	
`rechne(10)`	
`rechne(rechne(1))`	

Arbeitsauftrag 99: Eintritt im Freibad ●○○

Im Freibad kostet der Eintritt 4 Euro. Für jede Stunde, die man bleibt, kommen 2 Euro dazu.

1. Implementiere eine Funktion `eintritt(stunden)`, die den Preis in Euro **zurückgibt**.
2. Teste sie mit den Werten aus der Tabelle. Rechne die erwarteten Werte vorher von Hand aus.

Aufruf	erwartet	tatsächlich
`eintritt(1)`		
`eintritt(3)`		
`eintritt(0)`		

Arbeitsauftrag 100: Drei Fehler ●○○

Die Funktion soll die Fläche eines Dreiecks berechnen (Grundseite mal Höhe durch 2). Das Programm enthält **drei Fehler**. Finde sie und schreibe das Programm richtig auf. Teste es: Für die Grundseite 4 und die Höhe 3 muss `6.0` herauskommen.

```
1 def flaeche_dreieck(g, h)
2 return g * h / 2
3
4 print(flaeche_dreieck(4))
```

Die drei Fehler:

Arbeitsauftrag 101: Ein Test deckt den Fehler auf ●●○

Die Funktion soll den Mittelwert von zwei Zahlen zurückgeben. Sie läuft ohne Fehlermeldung.

```
def mittelwert(a, b):
    return a + b / 2
```

1. Rechne von Hand aus, was für `mittelwert(4, 6)` herauskommen **muss**.
2. Teste die Funktion mit `mittelwert(4, 6)` und mit `mittelwert(0, 0)`. Notiere die Ergebnisse.
3. Erkläre den Fehler und verbessere die Funktion.
4. Erkläre, warum der Test mit `0` und `0` allein den Fehler nicht gefunden hätte.

Meine Antworten:

Arbeitsauftrag 102: Kinokarten ●●○

Ein Kino verlangt je nach Alter einen anderen Preis:

- unter 6 Jahren: 0 Euro (frei),
- von 6 bis unter 18 Jahren: 6 Euro,
- ab 18 Jahren: 9 Euro.

1. Implementiere eine Funktion `kinopreis(alter)`, die den Preis **zurückgibt**. Verwende eine mehrseitige Verzweigung.
2. Teste die Funktion an den Grenzen: mit den Altern 5, 6, 17 und 18.

Arbeitsauftrag 103: Funktion in der Schleife ●●○

Fülle das Ablaufprotokoll des Hauptprogramms aus. Trage nach jedem Schritt die Werte von `i` und `summe` ein (– bedeutet: noch kein Wert). Gib dann an, was das Programm ausgibt.

```
1 def dreifach(zahl):
2     return 3 * zahl
3
4 summe = 0
5 for i in range(1, 4):
6     summe = summe + dreifach(i)
7 print("Summe:", summe)
```

Schritt	Zeile	Anweisung	i	summe
1	1	`def dreifach(zahl):`	–	–
2	4	`summe = 0`	–	
3	5	`for i in range(1, 4):`		
4	6	`summe = summe + dreifach(i)`		
5	5	`for i in range(1, 4):`		
6	6	`summe = summe + dreifach(i)`		
7	5	`for i in range(1, 4):`		
8	6	`summe = summe + dreifach(i)`		
9	5	`for i in range(1, 4):`		
10	7	`print("Summe:", summe)`		

Ausgabe:

Arbeitsauftrag 104: Quersumme als Funktion ●●●

Im Auftrag *Quersumme* (Abschnitt über die while-Schleife) hast du die Quersumme einer Zahl berechnet (Quersumme von 384 ist $3+8+4=15$). Zur Erinnerung: `zahl % 10` liefert die letzte Ziffer, `zahl // 10` entfernt sie.

1. Implementiere eine Funktion `quersumme(zahl)`, die die Quersumme **zurückgibt**. Teste sie mit 384 (erwartet: 15), 7 (erwartet: 7) und 1000 (erwartet: 1).
2. Benutze die Funktion in einer `for`-Schleife: Gib alle Zahlen von 1 bis 100 aus, deren Quersumme genau 10 ist.

Tipp

Schreibe die Schleife aus dem Abschnitt über die while-Schleife in den Funktionsrumpf und ersetze das `print` am Ende durch `return`. Im zweiten Teil steht der Aufruf `quersumme(i)` in der Bedingung einer `if`-Anweisung, so wie `ist_gerade(i)` im Auftrag *Gerade Zahlen*.

1.8.9 Das Wichtigste in Kürze

- Eine **Funktion** fasst Anweisungen unter einem Namen zusammen. Ihre **Definition** beginnt mit `def`, der Rumpf ist eingerückt. Die Definition legt nur fest, was die Funktion tut. Ausgeführt wird sie erst beim **Aufruf**.
- **Parameter** sind die Eingabewerte der Funktion. Beim Aufruf bekommen sie der Reihe nach die **Argumente**.
- `return wert` gibt einen **Rückgabewert** zurück und beendet die Funktion. Der Aufruf steht danach für diesen Wert.
- Ohne `return` liefert eine Funktion `None`. `print` in der Funktion zeigt nur an – weiterrechnen kann man damit nicht.
- Beim Aufruf springt Python in die Funktion und mit dem Rückgabewert zurück. Variablen in der Funktion sind **lokal**.
- Funktionen **testet** man mit mehreren Aufrufen, deren Ergebnis man vorher kennt – auch mit Sonderfällen.

Selbstcheck

- ☐ Ich kann eine Funktion mit `def` definieren und aufrufen.
- ☐ Ich kann den Unterschied zwischen Funktionsdefinition und Funktionsaufruf erklären.
- ☐ Ich kann erklären, was ein Parameter, ein Argument und ein Rückgabewert ist.
- ☐ Ich kann eine Funktion schreiben, die mit `return` ein Ergebnis zurückgibt.
- ☐ Ich kann erklären, warum eine Funktion mit `print` statt `return` den Wert `None` liefert.
- ☐ Ich kann den Ablauf eines Funktionsaufrufs Schritt für Schritt beschreiben.
- ☐ Ich kann eine Funktion mit sinnvoll gewählten Testwerten testen.

Gymnasium Ottweiler