Kantonsschule Im Lee

Informatik

Programmieren

Skript

Hauptautor Cyril Wendl

Ko-Autor Thomas Graf

[git] ▪ Branch: (None) @ 8aff860 ▪ Release: (2026-01-14)

Inhaltsverzeichnis

Kapitel 1

Getting Started

1.1 Installation von Python und VS Code

Um mit dem Programmieren loslegen zu können, müssen wir zuerst die Programmiersprache auf unserem Computer installieren, sowie einen guten Code-Editor, mit welchem wir Python-Code schreiben und ausführen können. Ein solches Programm wird typischerweise Integrated Development Environment (IDE) genannt. In diesem Skript verwenden wir die kostenfreie Programmiersprache *Python*, sowie die ebenfalls kostenfreie, weit verbreitete IDE mit dem Namen *Visual Studio Code (VS Code)*. Der nachfolgende Abschnitt leitet Sie durch die Installation sowohl auf Windows wie auf MacOS.

1.1.1 Anleitung für MacOS

Um VS Code unter MacOS zu installieren, benötigen Sie zuerst den Paket-Manager `brew`. Was macht `brew`? Laut der offiziellen Webseite: „Homebrew installiert Zeug, das du brauchst, das Apple aber nicht mitliefert."

Falls Sie `brew` noch nicht auf Ihrem Mac installiert haben, tun Sie dies wie folgt:

1. Öffnen Sie ein neues Terminal-Fenster, indem Sie zunächst die Spotlight-Suche mit ⌘+␣ (⌘ + Leertaste) öffnen. In der Spotlight-Suche müssen Sie nun den Suchbegriff `Terminal` eingeben und schliesslich die Suchanfrage mit Drücken der Taste ⏎ (`ENTER`-Taste) ausführen.
2. Geben Sie folgende Code-Zeile ein und führen Sie diese aus (indem Sie mit der ⏎-Taste bestätigen):

   ```
   /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install
       /HEAD/install.sh)"
   ```

Folgen Sie den Instruktionen, welche `brew` Ihnen im Terminal gibt! Sie erhalten einige Befehle, welche Sie kopieren und in demselben Terminal-Fenster ausführen müssen.

Führen Sie danach folgende Befehle einzeln aus (jeweils durch das Drücken der ⏎-Taste), um VS Code, Python und die Turtle zu installieren.

Installieren des IDE VS Code:

```
brew install --cask visual-studio-code
```

Installieren von Python 3 (neuste Version):

```
brew install python3
```

Installieren der Library `tkinter`, welche für die `turtle`-Grafik benötigt wird:

```
brew install python-tk
```

Nun sollten sowohl VS Code als auch Python installiert sein. Falls Sie VS Code nicht öffnen können und stattdessen eine Sicherheits-Warnmeldung erhalten, folgen Sie dieser Anleitung, um das Programm dennoch zu öffnen.

1.1.2 Anleitung für Windows

Öffnen Sie das Programm `PowerShell` (Sie können mit der ⊞-Taste danach suchen) **als Administrator**. Bei uns sieht die Situation aus wie in Abbildung 1.1.

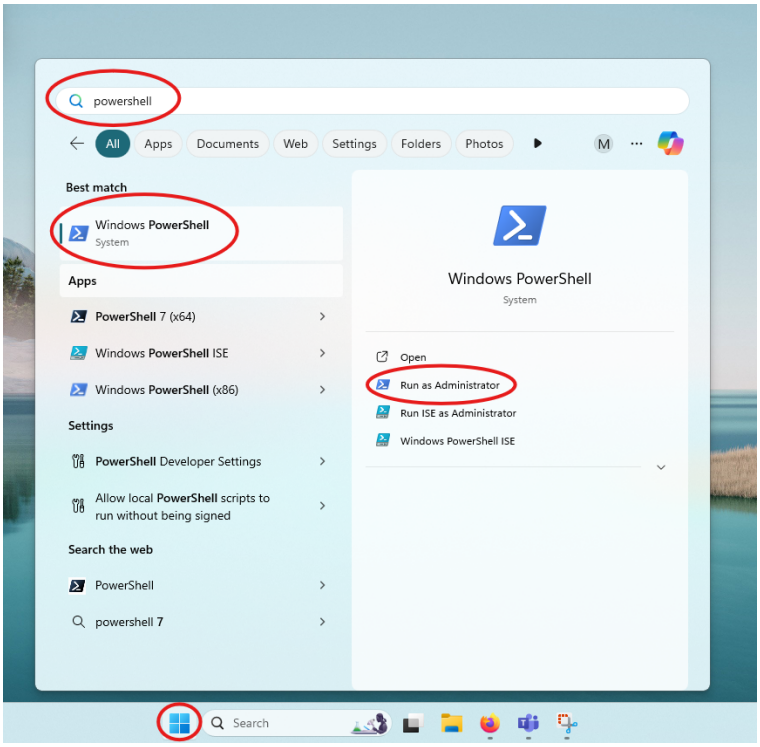


Abbildung 1.1: `PowerShell` unter Windows als **Administrator** öffnen.

Achtung

Wichtiger Hinweis 1.1:
Beim Kopieren und Einfügen der nachfolgenden Befehle in die `PowerShell` werden die Leerzeichen in den Befehlen entfernt. Diese sind aber wichtig! Sie müssen diese selber ergänzen (navigieren Sie mithilfe der Pfeiltasten).

In der `PowerShell` geben Sie folgende Zeile ein, um den Package-Manager `winget` zu installieren (bestätigen mit der ↵-Taste):

```
winget install -e --id Microsoft.PowerShell
```

Führen Sie danach folgende Befehle einzeln aus (jeweils durch das Drücken der ↵-Taste), um VS Code und Python zu installieren:

```
winget install -e --id Microsoft.VisualStudioCode
```

Der folgende Befehl listet die verfügbaren Python-Versionen auf. Führen Sie ihn aus und merken Sie sich die höchste Versionsnummer (so ist zum Beispiel Python 3.12 > Python 3.11):

```
winget search --id Python.Python
```

Installieren Sie nun die neueste (höchste Nummer) Version von Python mit dem nachfolgenden Befehl. Dabei muss allerdings der Platzhalter `.3.__` durch die neueste Versionsnummer von Python ersetzt werden (e.g. anstelle von `.3.__` muss `.3.12` stehen).

```
winget install -e --id Python.Python.3.__ --scope machine
```

Sie können jetzt mit der ⊞-Taste das Programm-Menu öffnen und nach *Visual Studio Code* suchen, welches nun installiert sein sollte.

1.2 VS Code für Python konfigurieren (MacOS und Windows)

Im Folgenden (falls diese auftauchen) müssen Sie Pop-Ups der Form wie sie in Abbildung 1.2 gezeigt sind, stets mit `Yes, I trust the authors` bestätigen.

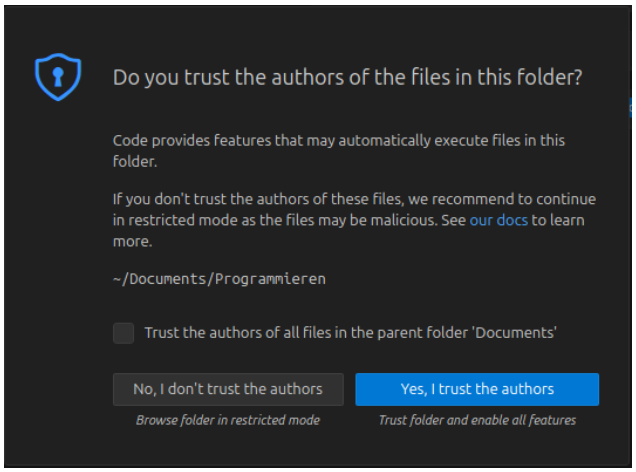


Abbildung 1.2: Meldungen dieser Art können Sie mit „Ja / Yes“ bestätigen.

Öffnen Sie zunächst VS Code und betrachten Sie Abbildung 1.3. Navigieren Sie mit der Maus ganz links zu den *Extentions* (Baustein-Icon) und suchen Sie im Suchfeld nach *python*. Installieren Sie die Erweiterung gemäss Abbildung 1.3[1]

[1]Python language support with extension access points for IntelliSense (Pylance), Debugging (Python Debugger), linting, formatting, refactoring, unit tests, and more.

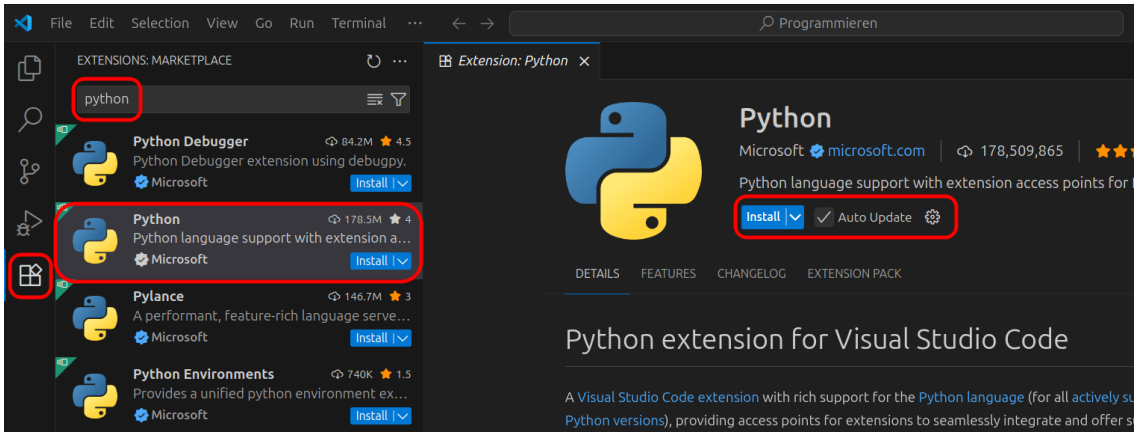


Abbildung 1.3: Installation der Python-Extension in VS Code.

1.3 Ordner / Verzeichnis für meine Programme

Wir empfehlen Ihnen, einen neuen Ordner (ein neues Verzeichnis) zu erstellen. Darin sollten Sie in Zukunft alle Programme, welche Sie im Grundlagenfach Informatik schreiben, abspeichern und verwalten. Wir haben das entsprechende Verzeichnis deshalb einfach „`Grundlagenfach`“ genannt. Am besten erstellen Sie Ihren Ordner in einem **Cloud-Service**, den Sie verwenden (z.B. `OneDrive`). Dadurch werden Ihre Daten zwischen all Ihren Geräten synchronisiert. In diesem Ordner sollten Sie keine persönlichen Daten ablegen.

Nun sind wir bereit, unser erstes Python-Programm zu schreiben und auszuführen.

1. In VS Code, klicken Sie `File` und dann `Open Folder...` und navigieren Sie zu dem Ordner, den Sie erstellt haben.
2. Erstellen Sie in diesem Ordner ein neues File mit dem Namen `hello_world.py`. Die Dateiendung `.py` gibt an, dass es sich dabei um ein *Py*thon-File handelt.

Bei uns sieht die Situation nun aus wie in Abbildung 1.4.

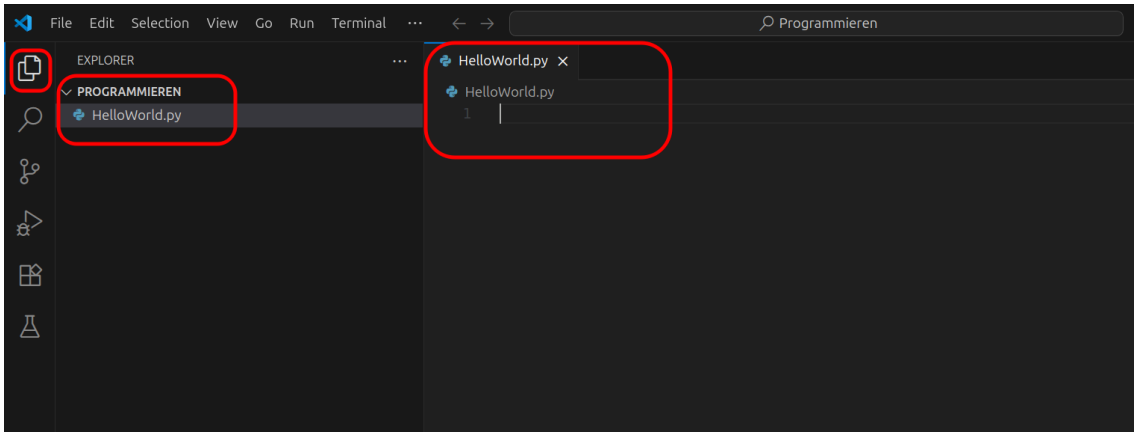


Abbildung 1.4: Python-Programm `hello_world.py` in VS Code erstellen.

1.4 Erstes Python-Programm schreiben und ausführen

Jetzt schreiben wir unser erstes Python-Programm! Dieses besteht aus nur einer einzigen Zeile und lautet:

```
print("Hello, World!")
```

Programm 1.1: `hello_world.py`

Die kleine Ziffer 1 ist die Zeilennummer (diese wird vom Code-Editor automatisch gesetzt). Wir führen nun das Programm aus („lassen das Programm laufen“), indem wir (oben rechts) auf den ▶️-Knopf klicken. Bei uns sieht die Situation aus wie in Abbildung 1.5.

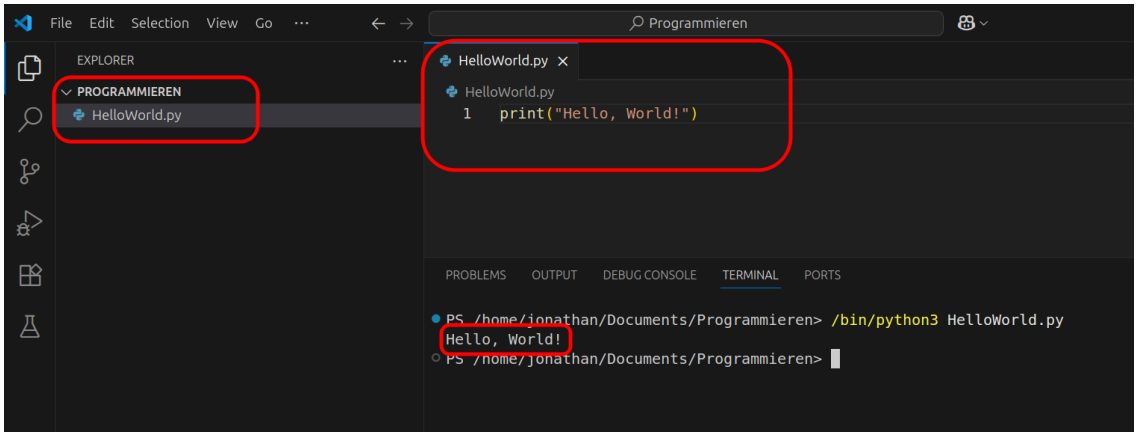


Abbildung 1.5: Python-Programm `hello_world.py` in VS Code ausführen.

Programm 1.1, macht nichts weiter, als den Text

```
Hello, World!
```

auf im Terminal (Englisch: `terminal`) auszugeben[2].

1.5 Installation von NumPy und Matplotlib

Bitte öffnen Sie nochmals ein Terminal-Fenster (MacOS) oder eine `PowerShell` (Windows) und installieren NumPy mit dem nachfolgenden Befehl:

```
für MacOS:
brew install numpy

für Windows:
pip install numpy
```

Installieren Sie bitte auch noch die Matplotlib-Bibliothek mit dem nachfolgenden Befehl:

```
für MacOS:
pip3 install matplotlib

für Windows:
pip install matplotlib
```

[2] https://de.wikipedia.org/wiki/Hallo-Welt-Programm

Kapitel 2

Einführung in Python und erste Schleifen

Um mit einem Computer zu „sprechen“, brauchen wir eine Sprache, die er versteht: eine **Programmiersprache**. Indem wir Programme schreiben, geben wir dem Computer klare Anweisungen, welche Aufgaben er erledigen soll. In diesem Skript nutzen wir die neueste Version der Programmiersprache Python.

Genau wie unsere menschlichen Sprachen haben Programmiersprachen Wörter mit einer festen Bedeutung. Wörter, die dem Computer sagen, was er tun soll, nennen wir Befehlswörter oder einfach **Befehle**.

Ein **Programm** ist im Grunde eine Reihe von Befehlen in einer Programmiersprache, die zusammen eine bestimmte Aufgabe lösen. Stell dir ein Programm als eine genaue Anleitung vor, die ein Computer Schritt für Schritt abarbeiten kann.

Das Hauptziel des Programmierens ist die Automatisierung von Abläufen. Wir übertragen die Ausführung einer Aufgabe komplett an den Computer. Deshalb muss ein Programm absolut eindeutig sein und genau beschreiben, was zu tun ist. Es darf keine Missverständnisse geben.

In diesem Kapitel schreiben Sie Ihre ersten eigenen Programme. Dabei lernen Sie das Konzept der Schleifen kennen. Schleifen sind unglaublich praktisch, denn sie ermöglichen es, wiederkehrende Aufgaben automatisch mehrfach auszuführen. Sie werden sehen, wie nützlich das in vielen Alltagssituationen ist!

2.1 Einige grundlegende Befehle und Operationen

2.1.1 `print`-Funktion und built-in Funktionen

In unserem `Hello, World!`-Programm (Programm 1.1) haben wir bereits die Python-Funktion `print(...)` gesehen. Diese Funktion ermöglicht es uns, Dinge (genauer: Python-Objekte) auszugeben („zu printen“) und dadurch für den User sichtbar zu machen. Die `print`-Funktion ist eine *built-in Funktion*, das heisst, sie ist direkter Bestandteil der Python-Sprache und ist in Python standardmässig verfügbar.

Beispiel 2.1:
Weitere Beispiele von Prints in Python:

```python
print("Hallo, World!")

print(5 / 2, "ist grösser als", 3 / 2)

print(3 * 15)
```

Programm 2.1: `prints.py`

Bemerkung 2.1:
Unter dem Link

https://docs.python.org/3/library/functions.html#print

finden Sie eine Übersicht aller built-in Funktionen in Python. Wir empfehlen Ihnen für diesen Link ein Lesezeichen (Bookmark) in Ihrem Webbrowser zu erstellen. Einige dieser Funktionen werden wir im Folgenden gemeinsam kennenlernen.

2.1.2 Python-Kommentare

Kommentare in Python dienen dazu, Programmcode genauer zu beschreiben. Kommentare werden von Python vollständig ignoriert und dienen lediglich dem besseren Verständnis des (menschlichen) Lesers.

Beispiel 2.2:
Dieses Beispiel zeigt die Funktionsweise von Zeilenkommentaren und Blockkommentaren.

```python
# Dies ist ein Zeilenkommentar.
# Ein Zeilenkommentar beginnt mit einem Rautesymbol / Hashtag.
print("Treffende Kommentare können dem Verständnis dienlich sein.")

"""
Dies ist ein Blockkommentar.
Ein solcher Kommentar darf sich über mehrere Zeilen erstrecken.

Ein Blockkommentar beginnt und endet mit jeweils drei Anführungszeichen.

Kommentare werden von Python ignoriert.
Diese Eigenschaft kann man sich zu
Nutze machen, um ausgewählte Programmteile temporär zu deaktivieren.
"""

# Wäre die folgende Zeile nicht kommentiert, würden wir
# einen Fehler erhalten (Divsion durch Null):
# print(7 / 0)

# Diese Berechnung ist aber ok:
print(7 / 3)

# Kommentare beginnen erst NACH dem Rautesymbol:
```

```
print("Das WIRD geprintet.")  # das wird aber ignoriert
```

Programm 2.2: `kommentare.py`

2.1.3 Einfache Arithmetik

In Python lassen sich einfache Rechenoperationen ähnlich wie bei einem Taschenrechner angeben. Dabei hält sich Python an die gewohnten Konventionen wie zum Beispiel „Punkt vor Strich“. Symbole gängiger arithmetischer Operationen sind in Tabelle 2.1 zusammengefasst.

mathematische Operation	**In Python**
Addition	+
Subtraktion	-
Multiplikation	*
Division	/
Potenzieren	**

Tabelle 2.1: arithmetische Operationen in Python

Übersicht 2.1 (arithmetische Operationen in Python)**:**

Beispiel 2.3:
Wir haben einige typische Rechenoperationen für Sie aufgeführt:

```
# berechnet zwar die Summe 9 + 10,
# gibt aber keinen Output (fehlender print)
9 + 10

# es gilt die Konvention 'Punkt vor Strich':
print(5 + 7 * 3)  # äquivalent zu 5 + (7 * 3) = 26

# Summe
5 + 503  # 508
# Differenz
10 - 24  # -14
# Produkt
8 * 5  # 40
# Division
10 / 4  # 2.5

# Potenz
2**4  # 16

"""
die Quadratwurzel (English: square root, kurz: sqrt)
```

```
ist nicht direkt Teil von Python, sondern muss durch
Importieren der Library 'math' hinzugefügt werden.
"""
import math

math.sqrt(2)  # 1.4142135623730951
```

Programm 2.3: `basisoperationen.py`

2.2 Erste Zeichnungen mit der Python-Turtle

Für den Einstieg in das Programmieren ist es didaktisch sinnvoll, mit der Python-Turtle zu starten. Genau dies werden wir hier tun! Wir werden der Turtle Instruktionen erteilen, damit sie für uns bestimmte Bilder und geometrische Formen zeichnet. Der didaktische Vorteil des Arbeitens mit der Turtle liegt in dem grafischen Output. Dieser lässt Sie (meist) sofort selber erkennen, was Ihr Programm macht und auch potenzielle Fehler lassen sich in der Regel recht einfach auffinden.

Um mit der Turtle arbeiten zu können, müssen Sie das `turtle`-Modul durch den Befehl `import turtle` zu Beginn des Programms (ganz oben) importieren. Ein Modul ist eine „Erweiterung“ von Python, so dass wir zusätzlich zu den *built-in functions* auch die Befehle des Moduls verwenden können.

Mit dem Ausdruck `import turtle as t` sagen wir, dass wir das Modul `turtle` in unser Programm importieren und dieses Modul innerhalb des Programms mit dem Ausdruck `t` benennen werden (wird könnten auch `import turtle as bliblablup` schreiben, dies wäre jedoch wohl weniger praktisch.

Programm 2.4 lässt die Turtle einen Strich mit einer Länge von 140 zeichnen. Programm 2.5 zeichnet ein gleichseitiges Dreieck mit Seitenlänge 100.

```
import turtle as t

# Tempo der Turtle festlegen
t.speed(1)

# gerade Linie / Strich zeichnen
t.forward(140)

# Turtle-Zeichnung stehen lassen
t.done()
```

Programm 2.4: `strich.py`

```
import turtle as t

# Tempo der Turtle festlegen
t.speed(1)

# gleichseitiges Dreieck zeichnen
t.forward(100)
t.left(120)
t.forward(100)
t.left(120)
t.forward(100)
```

```
t.left(120)

# Turtle-Zeichnung stehen lassen
t.done()
```

Programm 2.5: `dreieck.py`

Bemerkung 2.2:
Unter dem Link

https://docs.python.org/3/library/turtle.html#module-turtle

finden Sie eine Übersicht aller Turtle-Funktionen. Wir empfehlen Ihnen für diesen Link ein Lesezeichen (Bookmark) in Ihrem Webbrowser zu erstellen. Einige dieser Funktionen werden wir im Folgenden gemeinsam kennenlernen.

Aufgabe 2.1

Vervollständigen Sie die gegebene Vorlage, um ein Haus ähnlich dem in Abbildung 2.1 mithilfe der Turtle zu zeichnen.

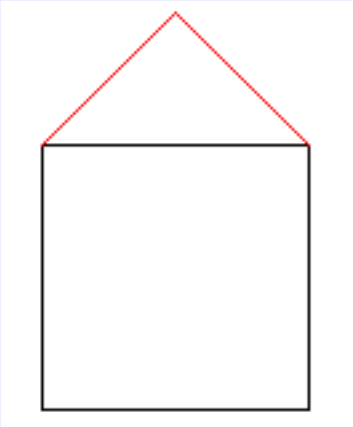


Abbildung 2.1: Traumhaus

- Die Längen dürfen Sie selber bestimmen.
- Das Schrägdach soll in roter Farbe gezeichnet werden und muss ein gleichschenkliges und rechtwinkliges Dreieck sein.
- Das Programm soll die Länge der Kathete dieses rechtwinkligen Dreieckes mit einem Print ausgeben.

```
import math
import turtle as t

# Tempo der Turtle festlegen
t.speed(1)

# Mauern
...
...
...
```

```
# Dach (rechtwinkliges Dreieck)
t.pencolor("red")  # setzte die Stiftfarbe auf rot
...
...
...

# Turtle verstecken (damit sie nicht das Haus verdeckt)
t.hideturtle()

# Turtle-Zeichnung stehen lassen
t.done()
```

Programm 2.6: `haus_vorlage.py`

Aufgabe 2.2

Schreiben Sie ein Programm um einen Dreizack möglichst ähnlich dem in Abbildung 2.2 mithilfe der Turtle zu zeichnen.

Abbildung 2.2: Poseidons Dreizack

Tipp: Durchsuchen Sie die Turtle-Dokumentation (siehe Bemerkung 2.2) um zu lernen, wie die Stiftbreite und Stiftfarbe (hier: `"gold"`) angepasst werden kann.

2.3 Schleifen

Schleifen (Englisch: *loops*) ermöglichen uns, Prozesse zu wiederholen.

Beispiel 2.4:
In Abbildung 2.3 ist eine Treppe gezeigt. Wie können wir diese Treppe mithilfe der Turtle zeichnen? Mit unserem bisherigen Wissen wäre dies sehr mühsam:

```
import turtle as t

t.lt(90)

# diese 4 Zeilen müssen wir 14 weitere Male Kopieren,
# um insgesamt 15 Stufen zu erhalten
t.fd(15)
t.lt(90)
t.fd(25)
t.rt(90)

t.hideturtle()
t.done()
```

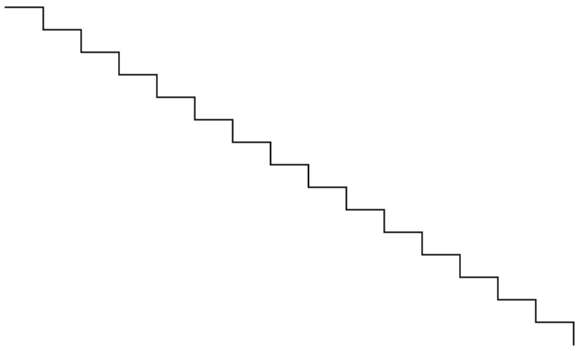


Abbildung 2.3: Stairway to Heaven

Stellen Sie sich vor, Sie müssten eine Treppe mit 10000 Stufen zeichnen 🧐.

Beispiel 2.4 zeigt auf, dass wir eine praktikable Methode zum häufigen Wiederholen von Mustern oder Prozessen benötigen. In Python lassen sich Wiederholungen mit Schleifen realisieren. Die Treppe aus Beispiel 2.4 lässt sich mit einer sogenannten `for`-Schleife ganz einfach wie in Programm 2.8 realisieren.

```
import turtle as t

t.lt(90)
for _ in range(6):
    t.fd(15)
    t.lt(90)
    t.fd(25)
    t.rt(90)

t.hideturtle()
t.done()
```

Programm 2.8: `treppe.py`

Dabei wird folgender Ausdruck:

```
for _ in range(15):
```

der *Schleifenkopf* der Schleife genannt. Die vier Zeilen (6 – 9) werden *Schleifenkörper* der Schleife genannt, wie im unten stehenden Code zu sehen ist.

```
for _ in range(15):
    t.fd(15)
    t.lt(90)
    t.fd(25)
    t.rt(90)
```

Schleifenkopf

Schleife

Körper der Schleife

Der Schleifenkörper ist (in Python) daran erkennbar, dass er (relativ zu dem Schleifenkopf) nach rechts eingerückt (Englisch: *indented*) ist. Diese Einrückung erreichen wir in VS Code durch das einfache Drücken der Tabulator-Taste (⇥). Wir können Code auch wieder um einen Tab nach links rücken, indem wir den entsprechenden Code zuerst selektieren (einfärben) und danach die Shift-Taste sowie die Tabulator-Taste gleichzeitig drücken (⇧+⇥). In anderen Programmen als VS Code kann es sein, dass die Tastaturkombination dafür eine andere ist.

Aufgabe 2.3

In Abbildung 2.4 wird gezeigt, wie ein Kreis schrittweise durch regelmässige Polygone[a] approximiert (angenähert)

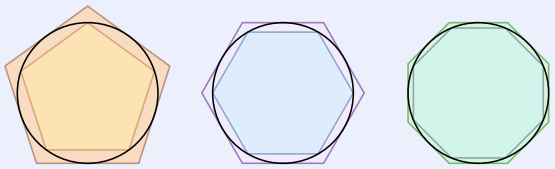


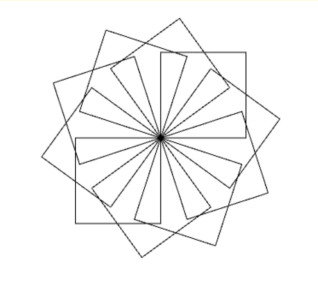
Abbildung 2.4: Schrittweise Annäherung an einen Kreis durch ein- beziehungsweise umbeschriebene regelmässige Polygone (links: 5-Ecke, mittels: 6-Ecke, rechts: 8-Ecke).

- Lassen Sie die Turtle in demselben Bild ein regelmässiges 5-Eck, ein regelmässiges 6-Eck sowie ein regelmässiges 8-Eck zeichnen.
- Die Orientierungen und Grössen der Figuren spielen dabei keine Rolle.
- Die Figuren sollen sich nicht überschneiden.
- Sie wollen die Turtle bewegen, ohne dabei zu zeichen? Schauen Sie sich die Funktionen `t.penup()` und `t.pendown()` in der turtle-Dokumentation an (suchen Sie nach diesen Befehlen mit dem Shortcut ctrl+F (Windows) oder ⌘+F (MacOS)).

[a] https://de.wikipedia.org/wiki/Regelm%C3%A4%C3%9Figes_Polygon

Aufgabe (Challenge) 2.4

Zeichnen Sie eine Blume, wie unten gezeigt.



Aufgabe 2.5

1. Zeichnen Sie die quadratische Schweizer Flagge 🇨🇭 (**Tipp**: zuerst ohne Farben, danach farbig). Um eine Form auszufüllen, schauen Sie sich die Funktionen `t.fillcolor()`, `t.begin_fill()` und `t.end_fill()` in der turtle-Dokumentation an. Suchen Sie nach dem Begriff `„fill"` mit dem Shortcut ctrl+F (Windows) oder ⌘+F (MacOS).
2. Zeichnen Sie die Flagge des Kantons Zürich (quadratisch):

   Farbe „Züri-Blau“: `(0 / 255, 112 / 255, 180 / 255)`
3. Zeichnen Sie eine horizontal dreigeteilte, rechteckige Flagge (wie etwa die Flagge Frankreichs 🇫🇷 oder Italiens 🇮🇹).

Aufgabe (Challenge) 2.6

In den folgenden beiden Aufgaben müssen Sie dasselbe tun wie in Teil 3 von Aufgabe 2.5 aber mit den folgenden Ergänzungen:

1. Die Breite jedes einzelnen der drei Rechtecke soll vom Programm bei jeder Ausführung individuell zufällig gewählt werden (das Programm soll zufällig drei Werte wählen).
2. Nun sollen zusätzlich zu den Breiten auch noch die Farben jeder der drei Rechtecksflächen zufällig gewählt werden.

Übersicht 2.2 (`turtle`-Befehle)**:**

Kurzform	Langform	Beschreibung
`t.fd(x)`	`t.forward(x)`	bewegt die Turtle um `x` Schritte vorwärts
`t.bk(x)`	`t.backward(x)`	bewegt die Turtle um `x` Schritte rückwärts
`t.rt(x)`	`t.right(x)`	dreht die Turtle um `x` Grad nach rechts
`t.lt(x)`	`t.left(x)`	dreht die Turtle um `x` Grad nach links
`t.color(c)`	`t.color(c)`	setzt die Zeichenfarbe auf `c` (z. B. `"red"`)
`t.begin_fill()`	`t.begin_fill()`	Start des Füllbereichs
`t.end_fill()`	`t.end_fill()`	Ende des Füllbereichs (füllt Fläche)
`t.pu()`	`t.penup()`	hebt den Stift (es wird nicht mehr gezeichnet)
`t.pd()`	`t.pendown()`	senkt den Stift (es wird wieder gezeichnet)
`t.ht()`	`t.hideturtle()`	versteckt die Turtle
`t.st()`	`t.showturtle()`	zeigt die Turtle
`t.speed(x)`	`t.speed(x)`	Geschwindigkeit setzen (`0` = schnellste)
`t.goto(x, y)`	`t.goto(x, y)`	bewegt die Turtle zu den Koordinaten `(x, y)`
`t.circle(r)`	`t.circle(r)`	zeichnet einen Kreis mit Radius `r`
`t.tracer(False)`	`t.tracer(False)`	zeigt direkt das Resultat (ohne Animation)[a]
`t.teleport(x, y)`	`t.teleport(x, y)`	teleportiert die Turtle zu `(x, y)`

Tabelle 2.2: Zusammenfassung nützlicher `turtle`-Befehle

[a] Funktioniert nicht in allen Programmier-Umgebungen

Kapitel 3

Variablen, Datentypen & Debugging

3.1 Variablen

Beispiel 3.1:
In Aufgabe 2.5 haben Sie bereits die (quadratische) Flagge des Kantons Zürich mithilfe der Turtle gezeichnet. Ohne Beachtung der Farben könnte der Code dafür wie in Programm 3.1 aussehen.

```
import turtle as t
import math

for _ in range(4):
    t.forward(50)
    t.left(90)

t.forward(50)
t.left(135)
t.forward(50 * math.sqrt(2))

t.hideturtle()
t.done()
```

Programm 3.1: `ohne_variable.py`

Angenommen wir wollten die Grösse der Züri-Flagge ändern, dann müssten wir den Wert `50` an genau **drei Stellen** entsprechend anpassen! Dieses Vorgehen ist zum einen Mühsam und zum Andern höchst fehleranfällig (es kann sehr gut passieren, dass eine Änderung nicht konsequent an allen Stellen durchgeführt wird).

Viel besser ist es, wenn wir der Seitenlänge des Quadrats einen Namen geben. Genau dies haben wir in Programm 3.2 getan: Wir haben der Länge den Namen `laenge` gegeben.

```
import turtle as t
import math

# Wir benutzen die Variable 'laenge', um der Länge einen Namen zu geben:
laenge = 50
```

```
for _ in range(4):
    t.forward(laenge)
    t.left(90)

t.forward(laenge)
t.left(135)
t.forward(laenge * math.sqrt(2))

t.hideturtle()
t.done()
```

Programm 3.2: `mit_variable.py`

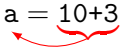
Der Ausdruck `laenge` ist ein Beispiel einer sogenannten **Variable**.

Definition 3.1 (Variable in Python)**:**
Eine *Variable* in Python ist im Wesentlichen ein Name, der auf einen im Speicher (des Computers) abgelegten Wert verweist. Mit der Zuweisung

```
a = 436 # a verweist auf ein Objekt vom Typ "ganze Zahl" mit dem Wert 436
```

wird veranlasst, dass der Variablenname `a` zu einer Referenz (einem Verweis) auf den Speicherort eines Objekts vom Typ „ganze Zahl“ mit dem Wert `436` wird.

Eine Variable wird erstellt, sobald man ihr mit dem Gleichheitszeichen (`=`) ein Objekt zuweist. Das Gleichheitszeichen hat allerdings nicht dieselbe Bedeutung wie in der Mathematik! Vielmehr bedeutet das Gleichheitszeichen in Python: „die linke Seite ist ein Name für das Ding auf der rechten Seite“:



Die Erklärung dieser Zeile ist: „Wir werten den Ausdruck `10 + 3` aus und erhalten die Zahl `13`. Diesen Wert (`13`) speichern wir nun in der Variable `a`“.

Wir können dies einfach überprüfen, indem wir den Wert von `a` ausgeben:

```
a = 10 + 3
print(a) # gibt 13 aus

a = 25 # der Wert der Variable wurde neu definiert
print(a) # gibt jetzt 25 aus
```

Variablen sind kurz gesagt ein Wort, das einen veränderbaren (= variablen) Wert enthält. Der Variablenname darf dabei (fast) frei gewählt werden, sofern folgende Regeln eingehalten werden:

Übersicht 3.1 (Benennung von Variablen)**:**
Bei der Wahl eines Variablennamens gibt es einige Punkte zu beachten:

1. Der Name muss mit einem Buchstaben beginnen (nicht: `1meinname, _abc`).
2. Der Name darf kein von Python reserviertes Wort sein (`for`, `def`, `if`, `print`, etc.).
3. Der Name sollte **sinnvoll** und **beschreibend** (deskriptiv) sein. Wenn Sie beispielsweise die Höhe eines Hauses in einer Variable speichern, ist es sinnvoll, diese `h` oder `hoehe` zu nennen und nicht etwa `x`.

4. Variablennamen dürfen nur alphanumerische Symbole (`a` – `z`, `A` – `Z`, `0` – `9`) und Unterstriche enthalten. Insbesondere also weder Leerzeichen noch Umlaute.
5. Da Variablennamen keine Leerzeichen enthalten dürfen, sollten zusammengesetzte Namen mithilfe von Unterstrichen (`_`) gebildet werden:

```
laenge_rechteck = 100
number_of_occupied_squares = 5
```

Achtung

Wichtiger Hinweis 3.1:
In der Mathematik bedeutet das Gleichheitszeichen „der rechte Teil der Gleichung ist gleich dem linken“. Dies ist in Python nicht so! Hier bedeutet das Gleichheitszeichen: „werte den rechten Teil aus und **speichere** das Resultat im linken Teil“. Es gibt also *keine* Speicherung von Werten in Python ohne Verwendung des Gleichheitszeichens.

Aufgabe 3.1

Berechnen Sie die folgenden Ausdrücke in Python und speichern Sie das Resultat jeweils in einer Variable:

Berechnung	Name der Variable
$17 - 3 \cdot 8$	`res1`
$(5 - 2) \cdot 4 + 1$	`res2`
3^6	`res3`
$18^{(2+3)}$	`res4`

Aufgabe 3.2

Gegeben sind zwei Variablen `x` und `y`. Sie möchten die Werte dieser zwei Variablen austauschen, so dass danach `x` den ursprünglichen Wert von `y` enthält und umgekehrt. Sie experimentieren zunächst mit folgendem Vorgehen:

```
1 # anfängliche Werte für x und y:
2 x = 5
3 y = 11
4
5 # Versuch die Werte von x und y zu tauschen:
6 x = y
7 y = x
```

Führen Sie das Programm aus und geben Sie die Werte von `x` und `y` auf der Konsole aus. Funktioniert das Programm wie erwartet? Weshalb (nicht)?

Aufgabe 3.3

Wie könnte der Code aus Aufgabe 3.2 angepasst werden, sodass der Tausch korrekt vollzogen wird?

Tipp: Verwenden Sie neben `x` und `y` noch eine dritte Variable.

Aufgabe (Challenge) 3.4

In den Variablen `x` und `y` sei jeweils eine ganze Zahl gespeichert. Schreiben Sie ein Programm, welches die Werte von `x` und `y` tauscht, ohne eine weitere „Hilfsvariable“ wie in Aufgabe 3.3 zu verwenden.

Tipp: Definieren Sie zuerst `x = x + y`

3.2 Teilen mit Rest

In diesem Abschnitt werden wir einige Überlegungen anstellen, welche Sie vielleicht an Ihre Zeit in der Primarschule zurückerinnern werden.

Beispiel 3.2:

- Angenommen seit einem gewissen Zeitpunkt seien 23 Tage vergangen. Der Zeitpunkt liegt also 3 ganze Wochen zurück und in der vierten Woche sind bereits 2 Tage vergangen. Wir können die Zahl 23 darstellen als $23 = 7 \cdot 3 + 2$. Dabei ist 3 (der Quotient) das Resultat der **ganzzahligen Division** von 23 (Dividend) geteilt durch 7 (Divisor). Dabei bleibt ein **Rest** von 2.
- Bei der ganzzahligen Division 23 geteilt durch 7 bestimmen wir also, wie häufig 7 vollständig (ganz) in 23 „passt“.
- Nehmen wir weiter an, der Tag 0 sei ein Montag. Dann war auch schon Tag -7 ein Montag und ebenso Tag $+7$. Offensichtlich entsprechen zwei Tage genau dann demselben Wochentag, wenn ihre Differenz (die Differenz ihrer Nummern) durch 7 teilbar ist[a]. So sind beispielsweise die Tage 23 und 37 beide Mittwoche.

[a] Mit anderen Worten: Ihre Differenz ist ein ganzzahliges Vielfaches von 7.

Definition 3.2 (ganzzahlige Divsion und Modulo-Operation (informell))**:**
Es seien a und b natürliche Zahlen und $b \neq 0$.

- Wir bezeichnen in Python mit `a // b` das Resultat der ganzzahligen Division von `a` geteilt durch `b` (wie oft hat `b` ganz in `a` Platz).
  Beispiel: `23 // 7` ist 3.
- Mit `a % b` (sprich: `a` **modulo** `b`) bezeichnen wir den Rest, welcher bei der ganzzahligen Division von `a` geteilt durch `b` bleibt. Wir nennen `%` die **Modulo-Operation**.
  Beispiel: `23 % 7` ist 2.

Für mehr Details zur Division mit Rest siehe Abschnitt C.1.

Beispiel 3.3:
Beispiele für die ganzzahlige Division in Python:

```
20 // 3 # ist 6
80 // 4 # ist 20
37 // 5 # ist 7
```

Beispiel 3.4:
Beispiele für Modulo-Berechnungen (Rest der ganzzahligen Division) in Python:

```
8 % 3 # ist 2
8 % 4 # ist 0
9 % 5 # ist 4
5 % 9 # ist 5
```

Aufgabe 3.5

Füllen Sie die zweite Spalte **von Hand aus**, berechnen Sie danach (zur Kontrolle) die Werte in Python.

Code	Resultat
`1 % 3`	?
`2 % 3`	?
`3 % 3`	?
`4 % 3`	?
`5 % 3`	?
`6 % 3`	?

Aufgabe 3.6

Es nehmen $p > 0$ Personen an einem Fest teil. Für das Fest wurden liebevoll m viele Muffins gebacken. Jede Person soll genau gleich viele Muffins erhalten. Berechnen Sie, wie viele Muffins nach dieser gerechten Verteilung auf p Personen übrig bleiben werden. Verwenden Sie dazu den Modulo-Operator.

Beispiel 3.5:
Offensichtlich ist eine natürliche Zahl n genau dann gerade, wenn die Gleichheit $n \,\%\, 2 = 0$ gilt, also n beim Teilen durch 2 den Rest 0 hat.

3.3 Zusammengesetzte Zuweisungsoperatoren

Wir haben bislang schon mehrfach Werte von Variablen verändert. In diesem Abschnitt führen wir einige gebräuchliche Python-Operatoren ein. Betrachten Sie das folgende Programm:

```
alter = 11
print(alter) # alter hat den Wert 10
alter = alter + 1 # alter hat neu den Wert 11
print(alter) # printet 11
alter + 5 # berechnet die Summe alter + 5, Inhalt von alter bleibt unverändert
print(alter) # printet 11
z = alter + 5 # Inhalt von alter bleibt unverändert
print(alter) # printet 11
alter += 7 # alter hat neu den Wert 18
print(alter) # printet 18
```

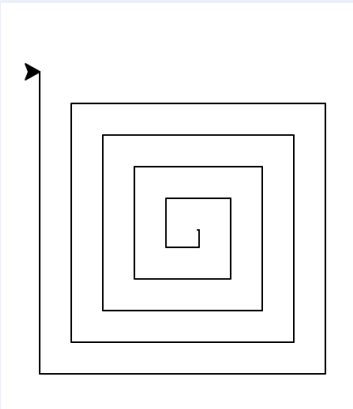
Es existiert ein fundamentaler Unterschied zwischen den beiden Ausdrücken `x + 5` (Wert von `x` bleibt unverändert) und `x += 5` beziehungsweise `x = x + 5` (Wert von `x` wird überschrieben). Anstelle von `x = x + y` werden wir häufig den zusammengesetzten Operator `+=` in der Form `x += y` verwenden. Analoge zusammengesetzte Operatoren für weitere mathematische Operatoren sind in Tabelle 3.1 aufgeführt.

zusammengesetzte Zuweisungsoperation	Bedeutung
`x += y`	speichere `x + y` in `x`
`x -= y`	speichere `x - y` in `x`
`x *= y`	speichere `x * y` in `x`
`x /= y`	speichere `x / y` in `x`
`x //= y`	speichere `x // y` in `x`
`x \%= y`	speichere `x \% y` in `x`

Tabelle 3.1: häufig verwendete zusammengesetzte Operatoren

Aufgabe 3.7

Zeichnen Sie eine Spirale der Form:



Die kürzeste Seitenlänge der Spirale ist 10 und nach jeder 90-Grad-Rotation soll die Seitenlänge um 10 länger werden.

Aufgabe 3.8

Die Kubikzahl einer Zahl x ist gleich der Zahl hoch 3, also x^3. Verwenden Sie eine Schleife um die 10 Kubikzahlen $1^3, 2^3, \ldots, 10^3$ auszugeben. Beginnen Sie mit einer Variablen `x = 1` und erhöhen Sie den Wert von `x` in jedem Durchgang der Schleife um 1.

3.4 Arbeiten mit Text (Strings)

In Python kann man nicht nur arithmetische Operationen mit Zahlen durchführen, sondern auch mit Text arbeiten. In der Informatik verstehen wir unter einem *Text* jede endliche Folge von Symbolen der Tastatur. Eine solche Folge von Symbolen bezeichnet man auch als *Zeichenkette* (Englisch: *string*). Beispiele von Strings haben wir bereits in Beispiel 2.1 gesehen. Wir können Strings, ebenso wie Zahlen, in Variablen speichern.

Beispiel 3.6:
Folgendes Beispiel speichert den Text `"Hello, World!"` in der Variable `a`, der Wert der Variable wird danach auf der Konsole ausgegeben:

```
a = "Hello, World!"
print(a)  # printet den String "Hallo, World!" in der Konsole

# anstelle von doppelten Anführungszeichen "..."
# dürfen auch einfache Anführungszeichen '...' verwendet werden:
b = 'Kantonsschule'

# aber auch sowas ist ein String:
c = "23adsf34#     2 @!!30y"
```

3.4.1 Verkettung und Vervielfachung von Strings

Zeichenketten können verkettet, also aneinandergehängt werden:

```
wort = "Kan" + "tons" + "schule"
print(wort) # Kantonsschule
```

Die „Addition“ von Strings wird *Konkatenation* (Englisch: *concatenation*) genannt.

Strings können mithilfe des `*`-Operators vervielfacht werden:

```
wort = "abc"
print(4 * wort) # abcabcabcabc
```

Falls Sie also Wiederholungen einer Zeichenkette wünschen, können Sie diese mit einer ganzen Zahl multiplizieren. Diese Zahl bestimmt die Anzahl der Wiederholungen der Zeichenkette. So entspricht der Ausdruck `"a" * 3` beispielsweise dem String `"aaa"`.

Aufgabe 3.9

Führen Sie das nachfolgende Programm aus und erklären Sie, was das Programm tut:

```
print(" " * 3 + "X" * 1)
print(" " * 2 + "X" * 3)
print(" " * 1 + "X" * 5)
print(" " * 0 + "X" * 7)
print(" " * 3 + "X" * 1)
print(" " * 3 + "X" * 1)
```

Aufgabe 3.10

Erstellen Sie nun ein Programm, das (analog zu Aufgabe 3.9) lediglich mit `prints` und Stringoperationen ein „Herz“ ❤️ zeichnet.

Aufgabe (Challenge) 3.11

Schreiben Sie ein Programm, welches die Form einer Banane auf der Konsole ausgibt, indem Sie die Stringoperationen `*` und `+` verwenden.

3.5 Datentypen

Bisher haben wir zwei Arten, oder *Typen* von Variablen gesehen: Zahlen und Text (Strings).

```
x = 2 + 3 # x ist eine Variable vom Typ "ganze Zahl"
y = "Hallo Welt!" # y ist eine Variable vom Typ "Text"
z = 'Donald Knuth' # ebenfalls eine Variable vom Typ "Text"
```

Variablen vom Typ „Text“ werden gut daran erkannt, dass der Wert der Variable (also der Text) in Anführungszeichen steht.

Bei Zahlen gilt es in Python zwischen der Darstellung einer ganzen Zahl (Englisch: *integer number* → *int*) und der Darstellung einer Dezimalzahl (Englisch: *floating point number* → *float*) zu unterscheiden, siehe Tabelle 3.2. Bitte beachten Sie, dass Dezimalzahlen immer mit Dezimalpunkt und nicht etwa mit einem Komma geschrieben werden, also `2.75` und nicht etwa `2,75`. Kommas werden in Python als Trennzeichen in Auflistungen verwendet.

Beispiel	Datentyp (englische, Abk.)	Datentyp (Deutsch)
`"Hallo Welt", 'abc', "42"`	string (str)	Zeichenkette
`15, 45, -5, 0`	integer (int)	ganze Zahl
`12.23, -5.33, 23.0`	float (float)	Kommazahl

Tabelle 3.2: Auswahl elementarer Datentypen in Python und Beispiele

Folgendes Beispiel illustriert, wie wichtig es ist, sich des Datentyps einer Variable bewusst zu sein. Das Plus-Symbol kann für zwei unterschiedliche Dinge verwendet werden:

1. Zahlen addieren: `print(2 + 3) # gibt 5 aus`
2. Strings verketten: `print("Hallo" + " " + "Welt") # gibt "Hallo Welt" aus`

Wichtig dabei ist, dass auf beiden Seiten des Plus-Zeichens (+) Werte desselben Datentyps stehen, da Python ansonsten nicht weiss, ob es addieren oder verketten soll. Daher ist es in gewissen Fällen nötig, eine Zahl in einen Text umzuwandeln, beispielsweise dann, wenn eine Zahl und ein Text im gleichen Print ausgegeben werden sollen. Hierzu verwenden wir den Befehl `str()`. Beispielsweise gibt uns `str(15)` die Zeichenkette `"15"`.

Beispiel 3.7:
Führen Sie folgenden Code in VS Code aus und beobachten Sie das Resultat in der Konsole:

```
n = 20
m = 30
prod = n * m
print(str(n) + " mal " + str(m) + " ist " + str(prod))
```

Eine etwas elegantere Art, Variablen in einem print mit Text zu verbinden ist, die Python-Schreibweise `print(f"...")` zu verwenden.

Beispiel 3.8:
Führen Sie folgenden Code in VS Code aus und beobachten Sie das Resultat in der Konsole:

```
n = 20
m = 30
quot = n / m
print(f"{n} durch {m} ist {quot}")
```

Diese Schreibweise erlaubt es uns, Variablen innerhalb der geschweiften Klammern zu verwenden, ohne sie vorher in Strings umwandeln zu müssen. Wir beobachten allerdings, dass der Quotient nicht sehr schön formatiert wird. Wir können die Anzahl der Dezimalstellen mit der Formatierung `:.2f` anpassen. Dabei bezeichnet `:.2f` eine Dezimalzahl mit 2 Nachkommastellen. Führen Sie folgenden Code aus:

```
n = 20
m = 30
quot = n / m
print(f"{n} durch {m} ist {quot:.2f}")
```

Aufgabe 3.12

Schreiben Sie ein Python-Programm, welches in einer Schleife die Quotienten von 2/3, 3/3, 4/3, etc. bis 11/3 berechnet und auf der Konsole ausgibt, z.B.:`"4 durch 3 ist 1.333"`. Verwenden Sie dazu die `f"`-Schreibweise und formatieren Sie die Quotienten so, dass immer 3 Nachkommastellen angezeigt werden.

Aufgabe 3.13

Eine Dame verrät uns die dreistellige Vorwahl 079 ihrer 10-stelligen Mobiltelefonnummer. Zudem verrät sie uns auch, dass die letzten drei Ziffern eine gerade Zahl zwischen 0 und 200 darstellen (also eine der Zahlen $000, 002, 004, \ldots, 200$).

Schreiben Sie ein Python-Programm, welches alle möglichen Nummern auflistet. Die erste Nummer sollte 0790000000 sein, die letzte 0790000200. Das Programm soll die Telefonnummern als Zeichenkette (eine Nummer pro Zeile) ausgeben.

Tipps:

- Mit `str(num)` wird aus der Zahl `num` eine Zeichenkette.
- Erinnern Sie sich, was die Operation `+` in dem Ausdruck `"In" + "form" + "atik"` macht?

Achtung

Wichtiger Hinweis 3.2 (Quotienten ganzer Zahlen sind in Python floats)**:**
Im Allgemeinen ist der Quotient n/m von zwei ganzen Zahlen n, m keine ganze Zahl. In Python wird deshalb das Resultat einer Division immer als Dezimalzahl (float) angeschaut, auch wenn n durch m ohne Rest teilbar ist[a]:

```
bruch = 6 / 2 # bruch ist ein float und kein int
print(bruch) # gibt 3.0 und nicht etwa 3 aus
bruch = int(bruch) # explizite Typenumwandlung von float zu int
print(bruch) # gibt 3 aus

x = 5.999999
x = int(x) # Weglassen von Dezimalstellen
print(x) # gibt 5 aus
```

Die folgende Schleife erzeugt einen Fehler (`TypeError`):

```
# das ist nicht ok:
for _ in range(6 / 2):
    print("hallo")

TypeError: 'float' object cannot be interpreted as an integer

# das ist ok:
for _ in range(int(6 / 2)):
    print("hallo")
```

da eine Schleife nur „ganzzahlig-viele“ Durchgänge durchführen kann, doch `6 / 3` ist `3.0`, also vom Typ float und nicht int.

[a] Das Konzept von *Teilbarkeit* ergibt in den reellen oder rationalen Zahlen keinen Sinn.

Aufgabe (Challenge) 3.14

Schreiben Sie ein Programm, welches 123456789 Sekunden in Jahre (à 365 Tage), Tage, Stunden, Minuten und Sekunden umrechnet. Das Programm soll den folgenden Text ausgeben, wobei alle Zahlen (ausser 123456789) im Programm berechnet werden: „123456789 Sekunden sind 3 Jahre, 333 Tage, 21 Stunden, 33 Minuten, 9 Sekunden!“

Aufgabe 3.15

Mit welchen Datentypen würden Sie folgende Informationen über sich selbst in Python speichern?

- Ihr Vorname
- Ihr Alter in Jahren
- Ihre Grösse in Metern

3.6 Textinput

Mithilfe der `input`-Funktion kann der User während der Programmausführung zu einer Eingabe aufgefordert werden.

Beispiel 3.9:
Mit der Funktion `input("Hilfstext...")` können wir eine Variable *während* der Programmausführung erstellen:

```
"""
Folgender Befehl druckt den Hilfstext
'Was ist ihr Name?' auf der Konsole.

Die Antwort kann auf der Konsole eingetippt
werden und wird danach in der Variable name gespeichert.

Mit der letzten Zeile wird der eingegeben Name
danach 5-mal gedruckt.
"""
name = input("Was ist ihr Name?")
print(name * 5)
```

Führen Sie diesen Code in VS Code aus und beobachten Sie das Resultat.

Aufgabe 3.16

Schreiben Sie ein Programm, das Sie nach Ihrem Namen fragt, indem der Hilfstext `"Wie heissen Sie?"` auf der Konsole ausgegeben wird. Verwenden Sie dazu die Funktion `name = input("Hilfstext hier...")`. Danach soll Sie das Programm neunmal begrüssen, indem es auf 3 Zeilen je dreimal den Text `Hallo [Ihr Name]` druckt.

Bemerkung 3.1 (Input erwartet immer einen String)**:**
Bitte beachten Sie, dass die `Input`-Funktion in Python den Tastaturinput immer als String auffasst. Falls Ihr Input als Zahl angesehen werden soll, dann müssen Sie den Input `x` mit `int(x)` beziehungsweise `float(x)` konvertieren:

```
alter = input("Wie alt bist du?") # alter ist vom Typ 'String'
print("In einem Jahr wirst du", int(alter) + 1, "Jahre alt sein.")
# andere, mögliche Schreibweise mit f"...:
# print(f"In einem Jahr wirst du {int(alter) + 1} Jahre alt sein.")

# dies würde einen Fehler geben, da wir hier die Summe eines Strings (alter)
# und einer ganzen Zahl zu berechnen versuchen:
print("In einem Jahr wirst du", alter + 1, "Jahre alt sein.")
```

3.7 Debugging

Häufig kommt es beim Schreiben von Code zu „unerklärlichen“ Fehlern: Der Code macht nicht, was man will, stürzt ab oder liefert nicht das gewünschte Resultat. Wir sprechen dabei von *bugs* (englisches Wort für *Käfer*), womit allgemein *Fehler* beziehungsweise unerwünschtes Verhalten oder unerwünschte Resultate gemeint sind. Daher ist es wichtig, zu lernen, wie man einem Problem auf die Schliche kommt. Darum handelt es sich beim sogenannten *debugging* um das Beheben unterschiedlicher Fehlerarten, welche in den folgenden Abschnitten kurz erklärt werden.

3.7.1 Syntaxfehler

Syntax-Fehler (Englisch: *syntax error*) sind in Programmiersprache vergleichbar mit Rechtschreibfehlern in Aufsätzen: Es handelt sich um Schreibweisen, die nicht erlaubt sind. Beispielsweise würde folgender Code eine Fehlermeldung geben:

```
print "Hello World" # Fehler!
```

Der Fehler tritt aufgrund der Tatsache auf, dass nach dem Befehl `print` Klammern stehen müssen. Der korrekte Code würde wie folgt geschrieben:

```
print("Hello World")
```

Syntax-Fehler, also Fehler in der grammatikalischen Struktur einer Programmiersprache, führen dazu, dass der Code nicht richtig ausgeführt werden kann.

Weitere, ähnliche Fehler können etwa auftreten, weil die Klammern nicht geschlossen wurden oder weil die Anführungszeichen bei Texten vergessen wurden:

```
print("Hello World" # Fehler (Klammer nicht geschlossen)!
print(Hello World) # Fehler (keine Anführungszeichen)!
```

3.7.2 Laufzeitfehler

Laufzeitfehler (Englisch: *runtime error*) werden erst bei der Ausführung des Programms erkannt. Dabei stimmt die grammatikalische Struktur (Syntax) des Codes zwar, der Code ergibt aber keinen Sinn: Beispielsweise wird versucht, auf Variablen zuzugreifen, welche es gar nicht gibt. Fehler können beispielsweise aufgrund von Gross- und Kleinschreibung auftreten:

```
X = 10 + 5
print(x) # Fehler: kleines "x" gibt es nicht!
```

Ein weiterer Laufzeitfehler könnte wie folgt aussehen

```
print(Hello, World)
```

In diesem Fall wollte der Autor des Codes den Text `"Hello, World"` auf die Konsole drucken, hat jedoch die Anführungszeichen vergessen. Dies führt dazu, dass der Code die Variablen `Hello` und `World` drucken will, die es jedoch nicht gibt.

Aufgabe 3.17

Führen Sie alle obigen Beispiele in VS Code aus und beobachten Sie die Fehlermeldungen. Verstehen Sie, was mit den Fehlermeldungen gemeint ist?

3.7.3 Semantische Fehler

Semantische Fehler sind Logik-Fehler, also Fehler aufgrund der Tatsache, dass die Programmiererin Denkfehler beim Schreiben gemacht hat. In diesem Fall gibt der Code zwar keine Fehlermeldung aus, die Ausgabe des Codes entspricht jedoch nicht dem erwarteten Resultat. Beispielsweise könnte folgender Code zu einem unerwarteten Resultat führen:

```
mittelwert = 3 + 13 / 2
print(mittelwert) # 9.5
```

Das Resultat sollte 8 sein, denn der Mittelwert von 3 und 13 ist $(3 + 13)/2 = 8$ und nicht etwa $3 + 13/2 = 9.5$. Der Fehler hat damit zu tun, dass der Programmierer vergessen hat, Klammern zu setzen. Korrekt wäre folgendes Beispiel:

```
mittelwert = (3 + 13) / 2
print(mittelwert) # 8
```

3.7.4 Debugging-Strategien

Einige Strategien, um fehlerhaften Code zu beheben, sind:

1. Lesen Sie Fehlermeldungen aufmerksam und versuchen sie zu verstehen, woher diese stammen könnten. Auch die Zeilenangaben sind dabei sehr hilfreich.
2. Brechen Sie ihren Code in einzelne Teile auf. Geben Sie Zwischenresultate auf der Konsole aus, indem Sie `print`-Befehle verwenden — dies kostet Sie nichts.
3. Rechnen Sie einige Beispiele von Hand aus, sofern der Code etwas Komplexeres berechnet.
4. Testen Sie Ihren Code mit (extremen oder besonderen) Testfällen, also anderen Werten für Ihre Variablen.

Aufgabe 3.18

Um welche Art von Fehler handelt es sich im Code aus Aufgabe 3.2?

3.8 Weitere Aufgaben

Aufgabe (Challenge) 3.19

Berechnen Sie die ersten 8 Zahlen der Fibonacci-Zahlenserie, welche wie folgt aussieht:

$$0, 1, 1, 2, 3, 5, 8, 13, \ldots$$

Dabei ist jede Zahl die Summe ihrer beiden Vorgänger in der Folge. Die ersten zwei Glieder 0 und 1 der Folge sind vorgegeben.

Verwenden Sie dazu eine `for`-Schleife sowie drei Variablen.

Zeichnen Sie danach mit der Turtle eine Spirale, bestehend aus 8 Viertelkreisen, welche als Radius[a] jeweils das **Zehnfache** der zuletzt berechneten Fibonacci-Zahl $1, 2, 3, 5, 8, 13, \ldots$ haben.

Die Spirale sollte folgendermassen aussehen:

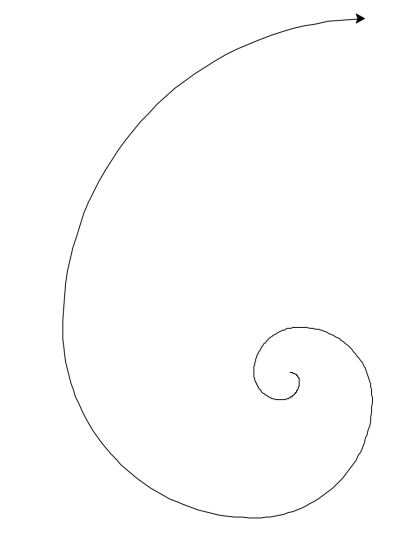


Abbildung 3.1: Fibonacci-Spirale

[a] Natürlich zeichnen wir hier regelmässige Polygone und keine Kreise.

Kapitel 4

Funktionen

4.1 Eigene Funktionen in Python definieren

Sie kenne bereits die Funktionen `print` sowie `t.forward` und haben diese auch schon mehrmals selbst verwendet. In diesem Kapitel werden Sie lernen, wie Sie eigene Funktionen in Python definieren und schliesslich sinnvoll verwenden können. Funktionen helfen enorm dabei, Programme leserlicher, kürzer, effizienter und wartbarer zu machen. Funktionen sind deshalb aus modernen Programmen kaum wegzudenken sind.

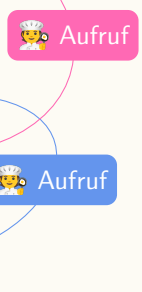
Beispiel 4.1:
Wir betrachten nochmals den Code aus Challenge 2.4:

```
import turtle as t

for _ in range(10):
    # zeichne ein Viereck
    for _ in range(4):
        t.fd(100)
        t.rt(360 / 4)
    # leichte Rechtsdrehung
    t.rt(360 / 10)

# Turtle-Zeichnung stehen lassen
t.done()
```

Programm 4.1: `blume.py`

Die Blume entsteht durch das wiederholte Zeichnen eines Quadrats, wobei sich die Turtle nach jedem gezeichneten Quadrat um 36 Grad rotiert. Wir könnten den Code leserlicher machen, indem wir zunächst eine Funktion `quadrat()` definieren, welche lediglich ein Quadrat zeichnet. Danach definieren wir eine weitere Funktion `quadrat_blume`, welche die Funktion `quadrat` insgesamt 10 Mal aufrufen wird:

```
1  import turtle as t
2
3  def quadrat():
4      for _ in range(4):
5          t.fd(100)
6          t.rt(360/4)
7
8  def quadrat_blume():
9      for _ in range(10):
10         quadrat()
11         t.rt(360/10)
12
13 quadrat_blume()
14
15 t.done()
```



Die Erstellung der Definitionen einer Funktion ist vergleichbar mit dem Verfassen eines Kochrezepts (oder eines Bauplans). Die Funktion beschreibt dabei einen Programmablauf, also ein „Rezept“. Die alleinige Existenz eines Kochrezepts führt natürlich nicht zu einem fertigen Gericht. Dieses entsteht erst bei der Ausführung des Rezepts. Genauso hat in Python die Definition noch keinen Effekt. Der Effekt (die Wirkung der Funktion) erfolgt erst bei ihrem Aufruf (in dem obigen Beispiel erfolgen solche Funktionsaufrufe in den Zeile 10 und 13).

Übersicht 4.1:
Innerhalb des eingerückten Bereichs unter `def` wird eine Wirkung definiert. Diese Wirkung wird aber erst ausgelöst, wenn die Funktion aufgerufen (und somit der Code ausgeführt) wird!
Das Schreiben von Definitionen hat mehrere Vorteile:

- Der Code wird **modular**: Man kann nun einfachere Funktionen schreiben, wie beispielsweise `quadrat()`, und komplexere Funktionen, die Unter-Funktionen aufrufen, wie beispielsweise `quadrat_blume()`. Auf diese Weise wird ein komplexer Code in einzelne, einfachere Teile (Module) unterteilt.
- Der Code wird **übersichtlicher**: Jede Teil-Aktivität ist eine Funktion
- Falls das komplexere Programm nicht das gewünschte Resultat liefert, können wir es *debuggen*, indem wir die einfacheren Funktionen zuerst aufrufen und sicherstellen, dass diese richtig funktionieren.

Beispiel 4.2:
Die folgende Gegenüberstellung illustriert, wie Funktionen dabei helfen, Programme übersichtlicher zu machen und Struktur in den Code zu bringen:

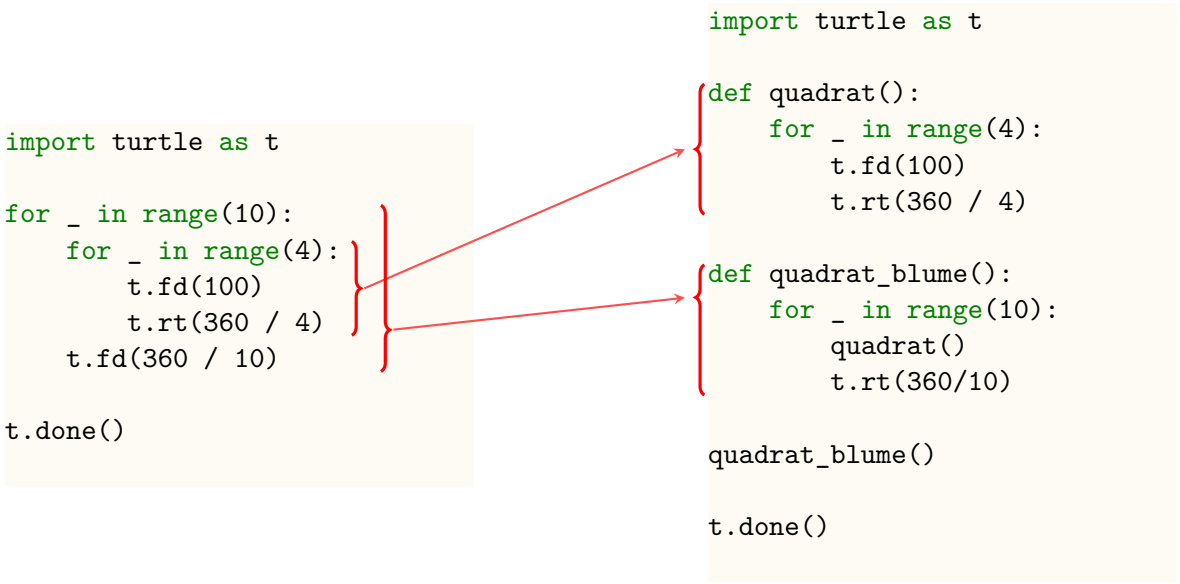


Abbildung 4.1: Vergleich von Schleifen mit Funktionsdefinitionen

Tipps für das Schreiben von Funktionen:

- Der Name einer Funktion sollte selbsterklärend und beschreibend sein. Im Idealfall lässt der Name der Funktion schon viel über ihren Effekt erahnen. Beispielsweise ist der Funktionsname `quadrat` viel beschreibender als ein nichtssagender Name wie zum Beispiel `f`.
- Etwas komplexere Tätigkeiten sollten jeweils als eine Funktion definiert werden und damit einen Namen erhalten. So kann beispielsweise das Zeichnen eines Quadrats in einer Funktion `quadrat` definiert werden, während das Zeichnen einer Blume in der Funktion `quadrat_blume` definiert wird. Diese Funktion kann dann wiederum die Funktion `quadrat` aufrufen.

Aufgabe 4.1

Führen Sie den folgenden Code zuerst aus und betrachten Sie das Resultat. Schreiben Sie den Code um, indem Sie zwei Funktionen schreiben:

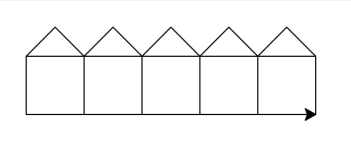
1. `viertelkreis`
2. `abgerundetes_quadrat`

```
import turtle as t

for _ in range(4):
    for _ in range(9):
        t.fd(4)
        t.lt(10)
    t.fd(100)
```

Aufgabe 4.2

Packen Sie Ihre Lösung aus Aufgabe 2.1 in eine Funktion mit dem Namen `haus`. Testen Sie zunächst, ob die Funktion wirklich das gewünschte Resultat liefert (ein einzelnes Haus), indem Sie die Funktion aufrufen. Schreiben Sie dann eine zweite Funktion `haeuserreihe`, welche eine Häuserreihe aus 5 Häusern zeichnet, indem `haus` fünfmal aufgerufen wird.

Ihr Code sollte folgendes Bild ausgeben:



4.2 Parameter

Stellen Sie sich vor, sie wollen drei unterschiedlich grosse Quadrate zeichnen. Sie könnten folgendermassen vorgehen:

```
import turtle as t


def quadrat50():
    for _ in range(4):
        t.fd(50)
        t.rt(90)


def quadrat100():
    for _ in range(4):
        t.fd(100)
        t.rt(90)


def quadrat150():
    for _ in range(4):
        t.fd(150)
        t.rt(90)


quadrat50()
quadrat100()
quadrat150()
```

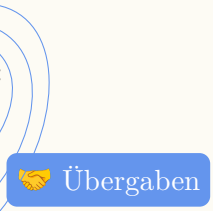
Programm 4.2: `squares_noparams.py`

Das ging gerade noch! Was jedoch, wenn Sie 20 unterschiedlich grosse Quadrate zeichnen wollen? Brauchen Sie dann 20 Definitionen? Zum Glück nicht, wie Beispiel Beispiel 4.3 zeigt.

Beispiel 4.3:
Folgendes Beispiel verwendet einen *Parameter* `laenge`, um die Länge des Quadrats jedes Mal

anzupassen:

```
import turtle as t

def quadrat(laenge):
    for _ in range(4):
        t.fd(laenge)
        t.rt(90)

quadrat(50)
quadrat(100)
quadrat(150)
```

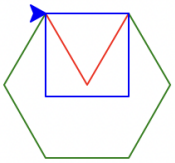


Funktionen können mehrere (oder auch gar keine) Parameter haben. Ersteres ist besonders nützlich, wenn Sie eine Funktion schreiben wollen, die mehrere Werte benötigt, um ihre Aufgabe zu erfüllen. Zum Beispiel könnte eine Funktion, die ein Vieleck zeichnet, sowohl die Anzahl der Ecken als auch die Farbe des Vielecks benötigen.

Beispiel 4.4:
In folgendem Beispiel kann nicht nur die Anzahl Ecken, sondern auch die Farbe eines Vielecks übergeben werden.

```
import turtle as t


def farb_vieleck(ecken, farbe):
    t.color(farbe)
    for _ in range(ecken):
        t.fd(50)
        t.rt(360 / ecken)


farb_vieleck(3, "red")
farb_vieleck(6, "green")
farb_vieleck(4, "blue")

t.done()
```

Programm 4.3: `farb_vieleck.py`



Übersicht 4.2 (Nomenklatur bei Python-Funktionen)**:**
Wir haben in diesem Kapitel mehrere wichtige Begriffe im Zusammenhang mit Funktionen in Python kennengelernt. Diese Begriffe wollen wir hier anhand eines Beispiels übersichtlich darstellen:

```
def produkt(x, y):
    print(x * y)


produkt(5, 3)
```

Name der Funktion : `produkt`
Das ist der Name (Bezeichner) der Funktion, mit dem sie definiert und aufgerufen wird.

Parameter der Funktion : `x`, `y`
Das sind die Platzhalter in der Funktionsdefinition, die beim Aufruf der Funktion mit Werten (Argumenten) belegt werden.

Argumente der Funktion : `5`, `3`
Das sind die tatsächlichen Werte, die beim Aufruf an die Funktion übergeben werden.

Funktionsaufruf : `produkt(5, 3)`
Das ist der Ausdruck, mit dem die Funktion ausgeführt wird.

Funktionssignatur : `produkt(x, y)`
Das ist die Kombination aus dem Funktionsnamen und der Parameterliste, also wie die Funktion definiert ist und wie sie verwendet werden soll.

Körper der Funktion : `print(x * y)`
Der Anweisungsblock innerhalb der Funktion. Er definiert, was die Funktion macht.

Die allgemeine Form einer Funktion in Python sieht also so aus:

```
# Definition einer Funktion
def funktionsname(parameter1, parameter2, ...):
    # Körper der Funktion
    ...


# Funktionsaufruf
funktionsname(argument1, argument2, ...)
```

Aufgabe 4.3

Ändern Sie den Code aus Beispiel 4.4 so ab, dass zusätzlich zur Farbe und der Anzahl Ecken auch noch die **Stiftdicke** für jedes Vieleck verändert werden kann. Verwenden Sie dazu einen weiteren Parameter sowie die Funktion `t.width`.

Aufgabe 4.4

Erstellen Sie die Funktionen mit den folgenden Signaturen:

- `def summiere(x1, x2)`: Berechnet die Summe der zwei Parameter `x1` und `x2` und gibt das Resultat mittels `print` auf der Konsole aus.
- `def summe_quadrate(x, y)`: Berechnet die Summe x^2+y^2 und gibt diese mittels `print` auf der Konsole aus.

4.2.1 Lebensdauer (*scope*) einer Variable

Im Code aus Beispiel 4.3 wird bei *jedem Aufruf* der Definition `quadrat` eine neue Variable `laenge` erstellt, welche nur *innerhalb* der Funktion existiert und welche jedes Mal einen anderen Wert hat. Den Wert erhält die Variable zum Zeitpunkt des *Aufrufs* der Funktion, also auf den Zeilen 8, 9 und 10!

Die Variable `laenge` existiert also nur innerhalb der Definition `quadrat`. Wenn wir nach Zeile 6 im Hauptprogramm (nicht eingerückt) den Befehl `print(laenge)` eingeben würden, würden wir eine Fehlermeldung erhalten, da es die Variable nicht mehr gibt. Parameter sind also immer ***lokale* Variablen**, ebenso wie Variablen, welche wir innerhalb von Funktion erstellen. Diese Variablen „sterben“ nach Ausführung der Definition, daher spricht man auch von der Lebensdauer einer Variable, bzw. von deren Reichweite (Englisch: *scope*)

Variablen, welche im Hauptprogramm (nicht eingerückt) erstellt werden, sind sogenannte ***globale* Variablen**.

Aufgabe 4.5

Beschreiben Sie, welche der Variablen im unten stehenden Code lokal oder global sind, und welche Variablen auch Parameter sind. Sie sollten insgesamt 5 Variablen finden!

```
import turtle as t
import math


def haus(laenge):
    anzahl_mauern = 4
    # Mauern
    for _ in range(anzahl_mauern):
        t.fd(laenge)
        t.lt(90)

    # Bewegung zu Dach hin
    t.lt(90)
    t.fd(laenge)

    # Dach
    t.rt(45)
    t.fd(laenge / math.sqrt(2))
    t.rt(90)
    t.fd(laenge / math.sqrt(2))
    t.rt(45)
```

```
    t.fd(laenge)
    t.lt(90)


def haeuserreihe(laenge_pro_haus):
    anzahl_haeuser = 5
    for _ in range(anzahl_haeuser):
        haus(laenge_pro_haus)


seitenlaenge = 50
haeuserreihe(seitenlaenge)

# Zeichnung stehen lassen
t.done()
```

Programm 4.4: `houses.py`

Aufgabe 4.6

Was könnte der Vorteil von *lokalen Variablen* sein?

4.3 Werte zurückgeben mit `return`

4.3.1 Einzelne Funktionen

Wie Sie bereits wissen, können Definitionen mit Kochrezepten verglichen werden: Sie beschreiben einen Ablauf, ohne diesen auszuführen. Damit der Code einer Funktion ausgeführt wird, müssen Sie die Definition **aufrufen**: In Beispiel 4.4 wurde dies beispielsweise mit `farb_vieleck(3, "red")` gemacht. Erst aufgrund dieser Zeile wurde etwas gezeichnet!

Dies kann nützlich sein, wenn sie eine Aufgabe mehrmals und in unterschiedlichen Varianten ausführen wollen, wie beispielsweise in Beispiel 4.4.

Eine wichtige Limitation von Definitionen war bisher, dass Sie zwar Dinge berechnen und auf der Konsole drucken konnten, allerdings verschwinden alle Parameter und lokalen Variablen nach der Ausführung einer Funktion. Dies bedeutet, dass alle Variablen, die wir je in einer Definition erstellt haben, nur in dieser Definition „lebten“. Wir konnten jedoch nicht mehr auf Parameter oder lokale Variablen zugreifen, nachdem das Programm beendet wurde.

Beispiel 4.5:
Betrachten Sie folgenden Code. Weshalb führt er zu einer Fehlermeldung?

```
1 def summiere(x1, x2):
2     summe = x1 + x2
3
4 summiere(3, 5)
5 print(summe)
```

Die Konsole gibt folgende Meldung aus: `[Zeile: 5] NameError: name 'summe' is not`

`defined`. Dies bedeutet, dass die Variable `summe` auf Zeile 5 nicht existiert. Die Fehlermeldung erklärt sich dadurch, dass alle Variablen, die innerhalb einer Definition erstellt werden, nur innerhalb dieser Definition „*leben*“, also existieren. Dass Variablen mittels dem Befehl `return` auch an das Hauptprogramm „zurückgegeben“ werden können, lernen wir in diesem Kapitel.

Funktionen können jedoch auch Werte an das Hauptprogramm zurückgeben: In diesem Kapitel lernen wir, wie Funktionen, ähnlich wie „Sous-Chefs“ (Hilfs-Köche) in einer komplexen Küche Zutaten (Parameter) entgegennehmen und Resultate an andere „Sous-Chefs“ (Funktionen) via das Hauptprogramm weitergeben werden können.

Eine Funktion kann man sich vorstellen wie ein Kochrezept, das einige Zutaten entgegennimmt und ein Resultat (Gericht) zurückgibt. Die Zutaten, oder „Inputs“, werden dabei häufig als „**Parameter**“ bezeichnet, währenddem das fertige Gericht, oder „Resultat“ häufig als „**`return`-Wert**“ bezeichnet wird. Diese Idee ist in Abbildung 4.2) veranschaulicht.

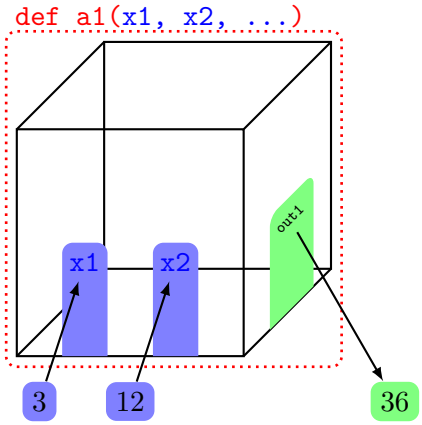


Abbildung 4.2: Illustration einer Funktion mit Inputs (Parametern) und Outputs (`return`-Wert)

Beispiel 4.6:
Betrachten Sie nochmals den folgenden, leicht modifizierten Code aus Beispiel 4.5. Wenn wir den Wert mit `return` an das Hauptprogramm zurückgeben und in einer Variable (im Hauptprogramm, auf Zeile 5) abspeichern, funktioniert der Code nun: Wir können auch ausserhalb der Funktion `summiere()` auf einen Wert zugreifen, welcher innerhalb der Definition berechnet wurde, und diesen potentiell weiterverwenden.

```
1 def summiere(x1, x2):
2     summe = x1 + x2
3     return summe
4
5
6
7 res=summiere(3, 5)
8 print(res)
```

Wert an das Hauptprogramm zurückgeben...

... und Wert in Variable speichern, z.B. `res`

Achtung

Wichtiger Hinweis 4.1:
Man muss sich diesen Code wie folgt vorstellen: Auf Zeile 7 wird die Funktion `summiere` aufgerufen, die Werte 3 und 5 werden für die Parameter `x1` und `x2` übergeben. Zeile 2 berechnet nun die Summe dieser beiden Werte. Auf Zeile 3 wird nicht die Variable `summe` an das Hauptprogramm zurückgegeben, sondern deren Wert (in diesem Fall: 8). Dies ist sehr wichtig: Nur weil wir `return summe` schreiben, heisst dass nicht, dass wir ausserhalb der Funktion `summiere` auf eine Variable `summe` zugreifen können. Vielmehr kann man sich vorstellen, dass der blau markierte Bereich auf Zeile 7 nun „ersetzt“ wird durch den Wert 8, welcher vom Unterprogramm an das Hauptprogramm zurückgegeben wird. Zeile 7 liest sich also nach der Ausführung des Unterprogramms so als ob man `res = 8` geschrieben hätte. Man könnte selbstverständlich auch einen anderen Variablenname statt `res` auswählen, z.B. `x`, `y`, `resultat` (kurz `res`) oder einfach `summe`. In letzterem Falle wäre `summe` auf Zeile 7 eine globale Variable und somit gänzlich unterschiedlich von der lokalen Variable `summe` auf Zeile 2.

Aufgabe 4.7

Gegeben seien die Höhe `h` und die Grundseite `b` eines Dreiecks. Schreiben Sie eine Funktion `flaeche_dreieck(h, b)`, welche den Flächeninhalt dieses Dreiecks berechnet und mit `return` zurückgibt. Testen Sie die Funktion mit den Werten `h=5` und `b=10`, speichern Sie das Resultat in einer Variable `result` und geben Sie deren Wert per `print` *nach dem Funktionsaufruf* (im Hauptprogramm) aus.

Überprüfen Sie die Korrektheit Ihrer Funktion, indem Sie sie auf Moodle hochladen.

Aufgabe 4.8

Geben sei eine ganze Zahl n. Schreiben Sie eine Funktion `square(n)`, welche das Quadrat von n berechnet und mit `return` zurückgibt. Testen Sie die Funktion mit dem Wert 5, speichern Sie das Resultat in einer Variable `result` und geben Sie deren Wert per `print` *nach dem Funktionsaufruf* (im Hauptprogramm) aus.

Überprüfen Sie die Korrektheit Ihrer Funktion, indem Sie sie auf Moodle hochladen.

Aufgabe 4.9

In welchen Fällen ist es sinnvoller, einen Wert einfach per `print` auf der Konsole auszugeben, und wann ist ein `return`-Befehl sinnvoller? Ist das Zurückgeben des Werts in Aufgabe 4.8 und Aufgabe 4.7 sinnvoll, oder würde hier auch ein `print` reichen?

4.3.2 Mehrere Funktionen

Wie Sie in Aufgabe 4.9 gesehen haben, ist ein `return` in vielen Fällen nicht notwendig, ein einfaches `print` reicht in vielen Fällen aus. Wozu kann ein `return` eigentlich nützlich sein? Wenn Sie Code schreiben, wird dieser oftmals schnell komplizierter. Somit kann es hilfreich sein, die einzelnen Schritte in Unter-Programme aufzuteilen. Dies erleichtert zudem die **Modularisierung** und **Wiederverwendbarkeit** von Code: Beispielsweise könnte eine Funktion, die das Maximum einer Liste

berechnet, an verschiedenen Orten in einem längeren Programm mehrmals zum Einsatz kommen

Zum Vergleich: stellen Sie sich vor, Sie arbeiten in einer edlen Michelin-Sterne-Küche an einem raffinierten Gericht, in das viele Arbeitsschritte involviert sind. Häufig müssen in solchen Restaurants bis zu 200 Schritte pro Gericht ausgeführt werden. Daher könnte es sinnvoll sein, einzelne Aufgaben, wie beispielsweise die Herstellung der Saucen, an Ihre Sous-Chefs zu delegieren, und eine Person zu bestimmen, die das ganze Gericht zusammenfügt. Häufig werden Funktionen genau mit solchen komplexen Aufgaben im Hinterkopf geschrieben. Diese Idee ist in Abbildung 4.3 illustriert.

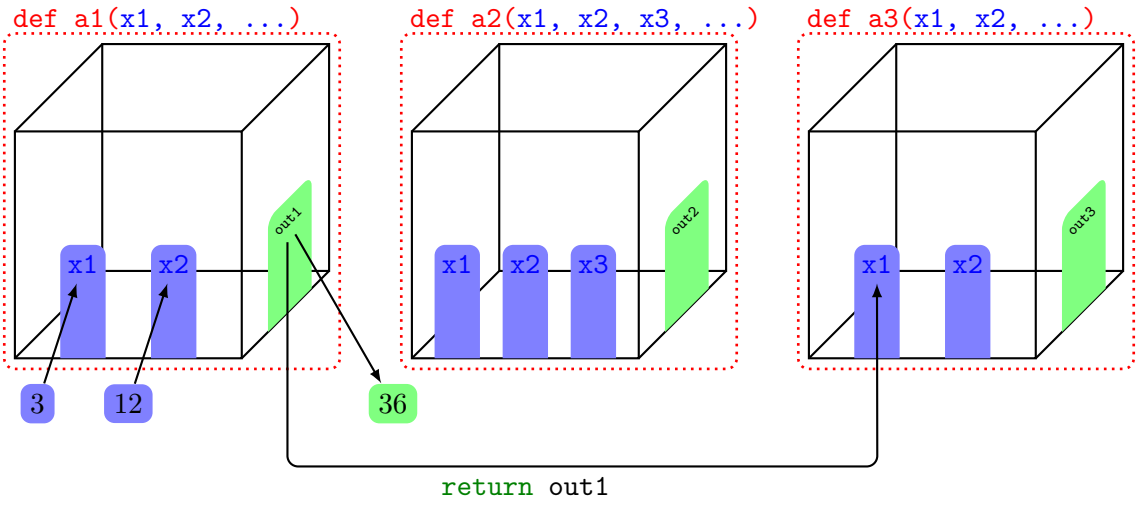


Abbildung 4.3: Illustration einer Code-Struktur, bei welcher mehrere Funktionen zusammenarbeiten

In Abbildung 4.3 haben wir eine Funktion `a1`, die zwei Parameter entgegennimmt (`x1` und `x2`). Mit diesen Parametern macht sie etwas (z.B. die Summe berechnen, ein Quadrat zeichnen oder irgend eine sonstige Aufgabe) und gibt danach einen neuen, berechneten Wert `a1` zurück. Dieser kann im Hauptprogramm abgespeichert und an andere Funktionen weitergegeben werden, beispielsweise an die Funktion `a3`.

Beispiel 4.7:
Stellen Sie sich vor, Sie arbeiten für einen Supermarkt und sollten den Preis Ihrer Produkte berechnen. Sie kaufen Ihre Artikel bei einem Grossverteiler ein, daher erhalten Sie auf alle Produkte 10 % Rabatt. Der Grossverteiler befindet sich im Ausland, Sie müssen also noch 7 % Mehrwertsteuer hinzufügen. Der Grossverteiler gibt Ihnen den Original-Preis Ihrer Waren *vor* Rabatt und *vor* Mehrwertsteuer.

Folgender Code berechnet den Endpreis, für den Sie ihre Waren verkaufen werden, indem folgende zwei Funktionen aufgerufen werden:

1. `berechne_rabatt` berechnet den Preis nach Abzug eines Rabatts für ein Produkt.
2. `berechnet_gesamtpreis` ruft `berechne_rabatt` auf und fügt zum rabattierten Preis die Mehrwertsteuer hinzu.

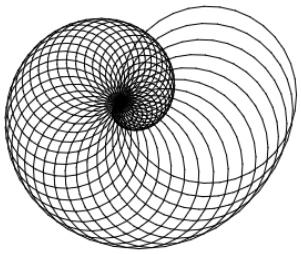
```python
def berechne_rabatt(preis, rabatt_prozent):
    rabatt = preis * (rabatt_prozent / 100)
    return preis - rabatt # Gibt den Preis nach Abzug des Rabatts zurück

def berechne_gesamtpreis(preis, rabatt_prozent, mwst_prozent):
    rabattpreis = berechne_rabatt(preis, rabatt_prozent)
    mwst = rabattpreis * (mwst_prozent / 100)
    return rabattpreis + mwst  # Gibt den Gesamtpreis inklusive Mwst. zurück

# Anwendung
preis = 100  # Basispreis in CHF
rabatt_prozent = 10  # Rabatt in %
mwst_prozent = 7.7  # Mehrwertsteuer in %

endpreis = berechne_gesamtpreis(preis, rabatt_prozent, mwst_prozent)
print("Der Endpreis nach Rabatt und Mehrwertsteuer ist:", endpreis)
```

Wert zurückgeben (und speichern)

Wert zurückgeben (und speichern)

Mit folgendem Code können wir die Turtle eine Spirale aus mehreren Vielecken zeichnen lassen:

```python
import turtle as t

# don't show turtle moving
t.tracer(False)


def vieleck(umfang, ecken):
    for _ in range(ecken):
        t.fd(umfang / ecken)
        t.rt(360 / ecken)


def vieleck_muster(umfang, ecken):
    for _ in range(60):
        vieleck(umfang, ecken)
        t.rt(6)
        umfang -= 10


vieleck_muster(600, 36)
t.done()
```



Programm 4.5: `spirale_kreis.py`

Beispiel 4.8:
Leider ist unsere Turtle etwas müde und sollte nicht zu viel laufen. Daher möchten wir

fortlaufend die Gesamtdistanz berechnen, um am Ende herauszufinden, ob unsere Turtle zu viel Strecke zurückgelegt hat. Dies können wir tun, indem wir die Funktion einen Wert zurückgeben lassen:

```
import turtle as t

def vieleck(umfang, ecken):
    for _ in range(ecken):
        t.fd(umfang/ecken)
        t.rt(360/ecken)

def vieleck_muster(umfang, ecken):
    gesamtdistanz=0
    for _ in range(60):
        vieleck(umfang, ecken)
        gesamtdistanz+=umfang
        umfang-=10
        t.rt(10)
    return gesamtdistanz

gesamtdistanz = vieleck_muster(600,35)
print(gesamtdistanz)
```

Aufgabe 4.10

Schreiben Sie eine Python-Funktion `def notenskala(maxPunkte, erreichtePunkte)`, welche die erreichte Note in Abhängigkeit der maximal möglichen Punktzahl und der erreichten Punktzahl berechnet und per `return` zurückgibt. Verwenden Sie dafür die folgende Formel:

$$\texttt{note} = \frac{\texttt{erreichtePunkte}}{\texttt{maxPunkte}} \cdot 5 + 1$$

Aufgabe 4.11 Formel 1

Ein Formel-1-Auto fährt eine bestimmte Strecke in einer bestimmten Zeit. Schreiben Sie eine Funktion `durchschnittsgeschwindigkeit(strecke_km, zeit_min)`, die Strecke in Kilometern und Zeit in Minuten als Eingabe erhält und die durchschnittliche Geschwindigkeit in km/h per `return` zurückgibt.

Formel: Durchschnittsgeschwindigkeit = Strecke (km) / Zeit (h)

Aufgabe 4.12 Boxenstopp ⛽

Ein Formel-1-Rennteam möchte die Gesamtzeit eines Rennens inklusive Boxenstopps berechnen. Gegeben sind die Gesamtstrecke in Km, die Durchschnittsgeschindigkeit in Km/h, die Anzahl Stopps sowie die Dauer pro Stopp in Sekunden.

Schreiben Sie zwei Funktionen:

`def fahrzeit_ohne_stopps(strecke, kmh)`: Berechnet die reine Fahrzeit in Sekunden und gibt diese per `return` zurück.

`def gesamtzeit(strecke, kmh, stopps, stoppdauer)`: Berechnet zuerst die Fahrzeit ohne Stopps (mit der ersten Funktion) und berechnet dann die Gesamtzeit mit Stopps, indem zur Fahrzeit ohne Stopps die Anzahl Boxenstopps mal die Dauer eines Boxenstopps (in Sekunden) hinzugefügt wird. Das Resultat soll ebenfalls per `return` zurückgegeben werden.

Formel 1: Fahrzeit (Sekunden) = Strecke (Km) / Geschindigkeit (Km/h) * 3600

Formel 2: Gesamtzeit (Sekunden) = Fahrzeit + (Anzahl Stopps × Stoppdauer)

Tipp 1: Schreiben Sie den Code zuerst in VS Code und testen sie ihn erst nachher auf Moodle.

Tipp 2: Schreiben Sie zuerst die erste Funktion und testen Sie diese mit folgenden Testwerten: Geschwindigkeit = 210 km/h, Strecke = 305 km. Das Resultat sollte sein: 5228.57 Sekunden. Schreiben Sie erst danach die zweite Definition, und rufen Sie darin die erste Definition auf.

4.4 Weitere Aufgaben

Aufgabe 4.13

Sie wollen ein Haus in der Schweiz kaufen. Der Kaufpreis beträgt k Franken. Sie kaufen das Haus mit E Franken Ihres gesparten Geldes (Eigenkapital) und der Restbetrag wird durch die Aufnahme einer Hypothek von $h := k - E$ Franken gedeckt.

Die Tragbarkeitsrechnung in der Schweiz prüft, ob die Kosten einer Immobilie langfristig finaziell tragbar (bezahlbar) sind. Banken und andere Kreditinstitute verwenden diese (sehr grobe) Berechnung, um sicherzustellen, dass Sie Ihre Hypothek auch dann noch bezahlen könnten, falls die Zinsen steigen würden. Als Faustregel gilt, dass die jährlichen (kalkulatorischen[a]) Wohnkosten nicht mehr als 1/3 Ihres Bruttojahreseinkommens (b) ausmachen dürfen.

Die kalkulatorischen jährlichen Wohnkosten setzten sich als die Summe der folgenden drei Positionen zusammen:

- Kalkulatorischer Zins: 5 Prozent des Hypothekenbetrags h, also 5 Prozent von $k - E$
- Nebenkosten: 1 Prozent des Kaufpreises k
- Amortisation: Falls die aufgenommene Hypothek grösser als 2/3 des Kaufpreises k ist, muss der darüberliegende Betrag $h - (2/3)k$ nach 15 Jahren zurückbezahlt sein. Die Bank rechnet also hierführ mit jährlichen Kosten a für die Amortisation von

  $$a := \frac{h - (2/3)k}{15} = \frac{(k - E) - (2/3)k}{15}$$

  und mit $a := 0$, falls die Hypothek 2/3 des Kaufpreises nicht übersteigt.

Zusammengefasst muss also die Ungleichung

$$\frac{\text{Brutto-Jahreseinkommen}}{3} \geq \text{Kalkulatorischer Zins} + \text{Nebenkosten} + \text{Amortisation}$$

oder anderst geschrieben

$$b/3 \geq 0.05(k - E) + 0.01k + \max\{0, (k/3 - E)/15\}$$

erfüllt sein. Wir wollen unsere Berechnung möglichst allgemein halten und ersetzen die kalkulatorische Hypothekenrate von 0.05 durch die positive Variable α und die kalkulatorische Nebenkostenrate 0.01 durch die positive Variable β:

$$b/3 \geq \alpha(k - E) + \beta k + \max\{0, (k/3 - E)/15\}\,. \tag{4.1}$$

Nun gibt es für den Kaufpreis k noch eine weitere Zusatzbedingung: **Der Kaufpreis darf nicht höher sein als das Fünffache des Eigenkapitals**.

1. Lösen Sie die Ungleichung nach k auf. Natürlich wird k abhängig sein von E, b, α und β.
2. Schreiben Sie eine Funktion `max_kaufpreis(E, b, alpha, beta)`, welche für gegebene Werte von E, b, α und β den maximal tragbaren Kaufpreis berechnet und mit `return` zurück gibt.
3. Geben Sie eine Formel für das maximal tragbare k in Excel an, falls E in `A1`, b in `A2`, α in `A3` und β in `A4` gespeichert sind.

[a] Der Begriff *Kalkulatorisch* bedeutet in diesem Zusammenhang, dass dieser Betrag nach bestimmten Vorgaben berechnet wird und überhaupt nicht den tatsächlichen Kosten entsprechen muss.

Kapitel 5

Verzweigungen und bedingte Schleifen

5.1 Verzweigungen mit `if`, `elif` und `else`

In vielen Fällen möchten wir Code nur Ausführen, *falls* eine gewisse Bedingung wahr ist: beispielsweise möchte ein Arzt nur dann eine Warnung erhalten, wenn der Blutdruck eines Patienten zu hoch ist, oder ein selbstfahrendes Auto sollte nur dann piepsen, wenn der Fahrer nicht aufmerksam auf die Strasse schaut. Um solche *konditionale* (bedingte) Logik zu programmieren, können wir die Begriffe `if` („falls“), `elif` („sonst falls“) und `else` („in allen anderen Fällen“) verwenden.

5.1.1 Verzweigungen mit `if`

Beispiel 5.1:
Folgendes Beispiel führt nur zu einem Output, falls eine Temperatur von 30 Grad oder mehr eingegeben wird.

```
def beschreibe_wetter(temperatur):
    if temperatur >= 30:
        print("Es ist heiss")


beschreibe_wetter(33)
```

Programm 5.1: `ex_if.py`

Der Code kann folgendermassen als Fluss-Diagramm aufgezeichnet werden (siehe Abbildung 5.1). Auf dem Flussdiagramm werden die einzelnen Schritte des Codes als Boxen dargestellt. Die Boxen sind durch Pfeile verbunden, die den Fluss des Codes darstellen. Die Entscheidungspunkte sind durch Rauten dargestellt, und die Pfeile zeigen, welche Aktion ausgeführt wird, je nachdem, ob die Bedingung wahr oder falsch ist.

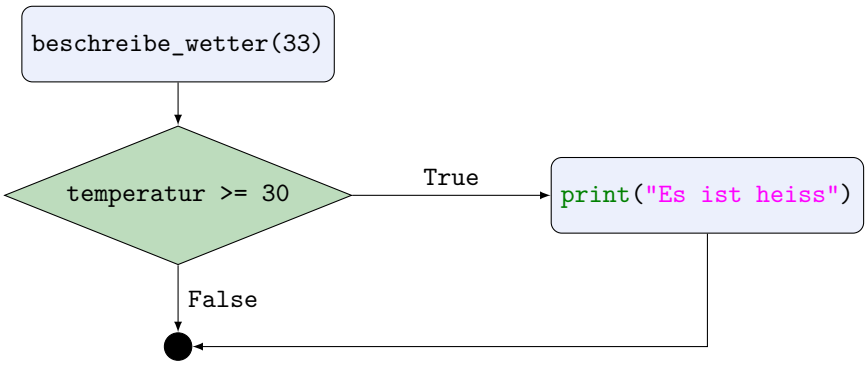


Abbildung 5.1: Flussdiagrammm für den Code aus Beispiel 5.1

Dabei bezeichnen blaue, rechteckig-abgerundete Boxen eine Zeile Code, grüne, rauteförmige Formen einen Test der schwarze Punkt das Programmende. Wenn immer eine Verzweigung vorliegt, geht der Pfeil für `True` nach links, der Pfeil für `False` geht nach rechts

Nach jedem `if` steht ein logischer Ausdruck oder ein logischer Wert.

Definition 5.1:
Logische Ausdrücke, auch ***Boolsche Ausdrücke*** sind Ausdrücke, die genau zwei verschiedene Werte annehmen können: entweder *wahr* (`True`) oder *falsch* (`False`). Beispiele sind:

```
3 == 5 # gibt den Wert False zurück
(9 + 3) < (2 * 8) # gibt den Wert True zurück
```

Wie Sie im obigen Beispiel sehen, wird von einem Wahrheitstest ein **Wert** zurückgegeben: entweder der Wert `True` (Wahr) oder `False` (Falsch).

Mögliche logische Relationen sind in Tabelle 5.1 abgebildet.

Python	Mathematische Bedeutung
`==`	gleich ($=$)
`!=`	ungleich ($\neq$)
`<`	kleiner ($<$)
`<=`	kleiner oder gleich ($\leq$)
`>`	grösser ($>$)
`>=`	grösser oder gleich ($\geq$)

Tabelle 5.1: Logische Relationen und Schreibweise in Python

Logische (oder auch bool'sche) Audrücke wie z.B. `3 < 45` führen zu einem ***Bool'schen Wert***, also einem Wahrheitswert, welcher entweder den Wert `True` (wahr) oder `False` (falsch) hat.

Die typische Verwendungsart logischer Ausdrücke ist in folgendem Code abgebildet:

```
if BOOLSCHER_WERT:
    # dieser Code wird nur dann ausgeführt, wenn BOOLSCHER_WERT wahr (True)
    ist.
```

Aufgabe 5.1

Entwickeln Sie eine Funktion `def quadrat(laenge)` zum Zeichnen von Quadraten. Der Befehl soll aber nur etwas zeichnen, wenn die Seitenlänge mindestens 40 beträgt. Zeichnen Sie sich zuerst ein Flussdiagramm des Codes auf Papier auf.

Aufgabe 5.2

Entwickeln Sie ein Programm, das dem Benutzer im Rahmen einer interaktiven Fragerunde drei Fragen stellt (mit dem Befehl `input("...")`).

Das Programm soll die Anzahl der richtigen Antworten zählen und diese Anzahl ausgeben.

Zur Erinnerung: `input("Frage")` gibt den Text `"Frage"` auf dem Bildschirm aus und wartet auf eine Eingabe des Benutzers. Um die Antwort zu speichern, muss der Befehl in eine Variable gespeicher werden. Zum Beispiel:

```
name = input("Wie heisst du?")
print("Hallo " + name)
```

Die Eingabe eines `input("...")`-Befehls wird immer als Text zurückgegeben, selbst wenn eine Zahl eingetippt wird. Um die Eingabe als Zahl zu interpretieren, muss der Text in eine Zahl umgewandelt werden. Dies geschieht mit dem Befehl `int(input("Frage"))`. Beispiel:

```
age = int(input("Wie alt sind Sie?"))
print("Sie sind " + str(age) + " Jahre alt.")
```

Zeichnen Sie Ihre Lösung als Flussdiagramm des Codes auf Papier auf.

5.1.2 Verzweigungen mit `if` und `else`

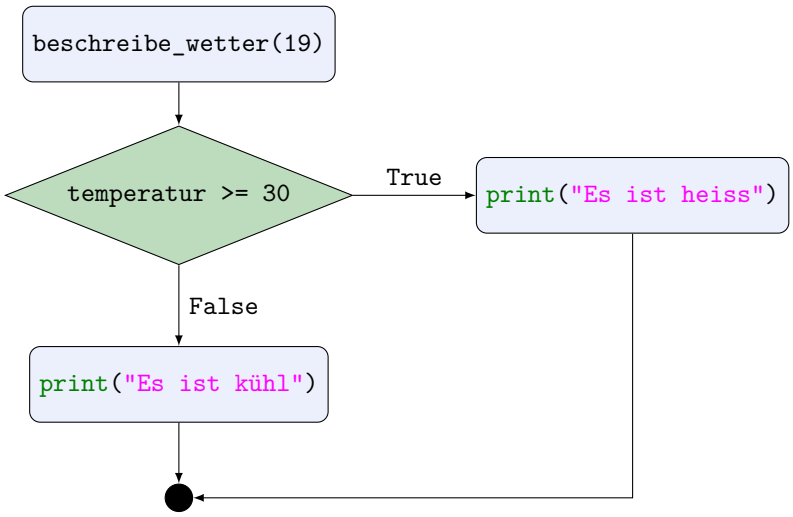
Beispiel 5.2:
In folgendem Code wird mit dem `else`-Ausdruck ein Fall definiert, welcher ausgeführt wird, sofern die `if`-Kondition nicht zutrifft.

```
def beschreibe_wetter(temperatur):
    if temperatur >= 30:
        print("Es ist heiss")
    else:
        print("Es ist kühl")


beschreibe_wetter(19)
```

Programm 5.4: `ex_if_else.py`

Beachten Sie, dass nach dem `else` *kein* logischer Ausdruck steht, da der Code unterhalb des `else` nur dann stattfindet, wenn alle zuvor genannten Tests falsch waren.

Der Code kann ebenfalls als Fluss-Diagramm aufgezeichnet werden:



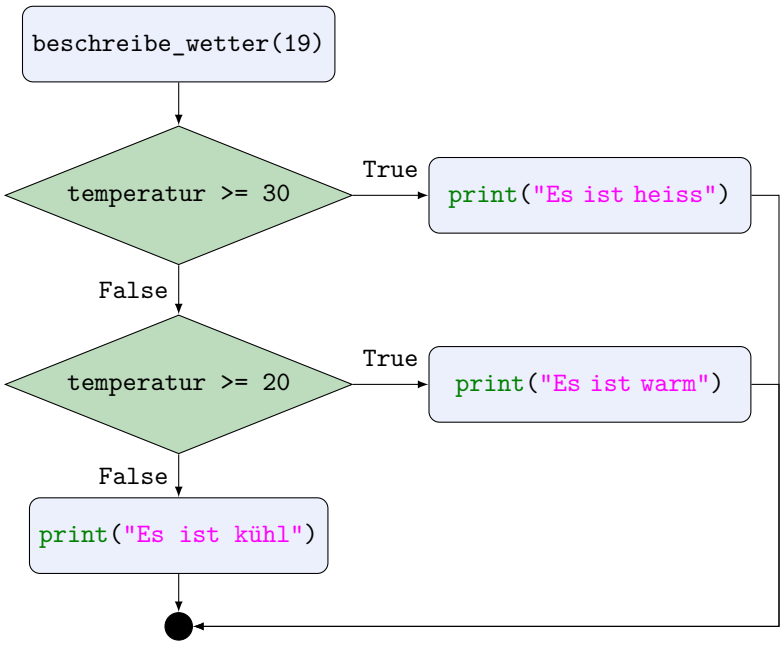
5.1.3 Verzweigungen mit `if`, `elif` und `else`

Beispiel 5.3:
Folgender Code führt eine von drei Möglichkeiten aus:

```
def beschreibe_wetter(temperatur):
    if temperatur >= 30:
        print("Es ist heiss")
    elif temperatur >= 20:
        print("Es ist warm")
    else:
        print("Es ist kühl")


beschreibe_wetter(19)
```

Programm 5.5: `ex_if_elif_else.py`

Der Code kann folgendermassen als Fluss-Diagramm aufgezeichnet werden:



Aufgabe 5.3

Ein Bäcker möchte Kekse backen und gleichmässig in Keksdosen verpacken, so dass jede Dose voll ist. Jede Dose fasst 12 Kekse. Schreibe ein Programm, das berechnet:

- Wie viele Dosen benötigt werden für n Kekse.
- Wie viele Kekse übrig bleiben.

Falls mehr als 500 oder weniger als 1 für `n` eingegeben werden, soll ausgegeben werden:

```
"Ungültige Anzahl Kekse!"
```

Verwenden Sie die Ganzzahldivision (`//`) und Modulo (`%`)! Zur Erinnerung: `//` gibt den ganzzahligen Teil der Division zurück, `%` gibt den Rest der Division zurück.

Aufgabe 5.4

Verwenden Sie einen `input`-Befehl, um den Benutzer nach einer Zahl n zu fragen.

1. Falls die Zahl n=1 eingegeben wird, soll ein blaues Viereck gezeichnet werden.
2. Falls eine Zahl n von 2 bis und mit 6 eingegeben wird, soll ein grünes Sechseck gezeichnet werden.
3. Falls eine Zahl n von 7 oder grösser eingegeben wird, soll ein schwarzes n-Eck gezeichnet werden.

Aufgabe 5.5

Bei einer Flugreise darf der aufgegebene Koffer üblicherweise nicht schwerer sein als 20 kg. Ansonsten bezahlt man einen Aufschlag von CHF 5.- pro kg Übergewicht.

Schreiben Sie ein Unterprogramm koffer(gewicht), der einen Parameter Gewicht entgegennimmt und einen Text auf der Konsole druckt, je nach Fall:

- Wenn der Koffer über 100 kg wiegt, wird er nicht transportiert (`"Der Koffer ist zu schwer"`).
- Wenn der Koffer über 20 kg und bis und mit 100 kg wiegt, muss der Aufpreis berechnet und ausgegeben werden (z.B. `"Ihr Koffer hat X kg Übergewicht. Das kostet (5*X).-"`).
- Wenn der Koffer über 0 kg und bis und mit 20 kg wiegt, muss kein Aufpreis bezahlt werden und es wird gedruckt `"Der Koffer ist gratis"`.
- In allen anderen Fällen soll ausgegeben werden `Das eingegebene Gewicht ist nicht zulässig` (0, negative Zahlen etc.).

Aufgabe 5.6

Entwickeln Sie eine Funktion `def vielecke_sicher(anzahl, seite)` zum Zeichnen von regelmässigen Vielecken mit wählbarer Anzahl Ecken und wählbarer Seitenlänge. Wenn `anzahl` (die Anzahl der Ecken) kleiner als 1 ist, soll das Programm nichts tun. Wenn `anzahl == 1` ist, soll das Programm `"Es gibt kein 1-Eck"` ausgeben. Wenn `anzahl == 2` ist, soll das Programm `"Es gibt kein 2-Eck"` ausgeben. Wenn `anzahl >= 3` ist, soll das Programm das `anzahl`-Eck mit Seitenlänge `seite` zeichnen. Zeichnen Sie auch das dazugehörige Flussdiagramm.

Aufgabe 5.7

Was gibt dieser Code aus? Ist die Ausgabe sinnvoll? Weshalb (nicht)?

```
def beschreibe_wetter(temperatur):
    if temperatur >= 30:
        print("heiss")
    if temperatur >= 20:
        print("warm")
    else:
        print("kühl")


beschreibe_wetter(33)
```

Programm 5.10: `ex_if_if_else.py`

Aufgabe 5.8

Entwickeln Sie eine Funktion `def quadgleich(a, b, c)`, welche quadratische Gleichungen der Form $ax^2 + bx + c = 0$ löst.

Zur Erinnerung: die Formel zur Berechnung der Lösungen ist:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{5.1}$$

Der Befehl soll zuerst $d = b^2 - 4ac$ ausrechnen und abhängig vom Wert von d keine, eine oder zwei Lösungen ausgeben.

- Falls $d < 0$, gibt es keine reelle Lösung
- falls $d = 0$, gibt es genau eine reelle Lösung
- falls $d > 0$, gibt es genau zwei Lösungen

Aufgabe 5.9

Gegeben seien zwei Zahlen `x1` und `x2`. Schreiben Sie eine Funktion `maxzahl(x1, x2)`, welche die grössere der beiden Zahlen mit `return` ausgibt. Wenn beide Zahlen gleich gross sind, soll `x2` ausgegeben werden.

Aufgabe (Challenge) 5.10 Dynamisches Schachbrett

Wir möchten ein quadratisches Schachbrettmuster mit $n \times n$ Feldern erzeugen, wobei n eine gerade positive natürliche Zahl ist. Jedes der n^2 Felder soll dabei eine Grösse von genau $s \times s$ Zeichen haben ($s \geq 1$). Die Zahl 1 repräsentiere die schwarzen Felder, die Zahl 0 die weissen Felder.

Zusätzlich möchten wir wählen können, ob das linke obere Feld schwarz oder weiss sein soll (`upper_left = 'black'` oder `upper_left = 'white'`). Folgende Beispiele zeigen die entsprechenden Schachbrettmuster für verschiedene Wahlen der drei Parameter n, s und `upper_left`:

```
# Schachbrettmuster für n = 2, s = 3, upper_left = 'white'
000111
000111
000111
111000
111000
111000

# Schachbrettmuster für n = 4, s = 1, upper_left = 'black'
1010
0101
1010
0101

# Schachbrettmuster für n = 6, s = 2, upper_left = 'black'
110011001100
110011001100
```

```
001100110011
001100110011
110011001100
110011001100
001100110011
001100110011
110011001100
110011001100
001100110011
001100110011
```

Allgemein soll das Muster immer genau ns Zeichen breit und ebenso hoch sein. Schreiben Sie ein Python-Programm, welches die drei oben beschriebenen Parameter akzeptiert und das entsprechende Schachbrettmuster ausgibt.

5.1.4 Logische Ausdrücke miteinander verbinden: `and` und `or`

Häufig fällen wir im echten Leben Entscheidungen, welche nicht nur von einer Bedingung abhängen, sondern gleich von mehreren, so zum Beispiel:

- Ich gehe per Fahrrad zur Schule, *falls* das Wetter schön ist *und* ich mich körperlich fit fühle.
- Ich esse etwas, *falls* ich Hunger habe *oder* ich Lust darauf habe (auch wenn ich keinen Hunger habe).

Beim ersten Beispiel handelt es sich um eine Verbindung per „*und*“: beide Konditionen müssen wahr sein, damit etwas geschieht. Dies kann in Python mit `and` (englisch für „und“) umgesetzt werden.

Beispiel 5.4:
Folgendes Beispiel illustriert, wie mehrere Bedingungen miteinander verknüpft werden können:

```
temperature = 25 # aktuelle Aussen-Temperatur
fitness = 80 # körperliche Fitness (zwischen 0-100, subjektiv empfunden)
if (temperature > 20) and (fitness > 80):
    print("Ich gehe per Fahrrad zur Schule!")
```

Beim zweiten Beispiel handelt es sich um eine Verbindung mehrerer Bedingungen per „*oder*“: es reicht, dass eine von beiden Bedingungen wahr ist, damit etwas geschieht. Dies kann in Python mittels dem Wort `or` (englisch für „oder“) umgesetzt werden.

Beispiel 5.5:
Folgendes Beispiel illustriert, wie mehrere Bedingungen miteinander verknüpft werden können:

```
hunger = 25 # aktueller Hunger-Wert (zwischen 0-100, subjektiv empfunden)
lecker = 80 # wie lecker ist das Lebensmittel (Skala von 0 bis 100)?
if (hunger > 80) or (lecker > 80):
    print("Ich will das essen!")
```

Aufgabe 5.11

Schreiben Sie eine Funktion `geschwindigkeit_angemessen(geschwindigkeit)`, welche als Parameter `geschwindigkeit` eine Zahl entgegennimmt (z.B. `50` oder `120`). Falls die Geschwindigkeit zwischen 30 und 100 km/h liegt (einschliesslich dieser Werte), soll auf der Konsole ausgegeben werden: „Die Geschwindigkeit ist angemessen“. Ansonsten soll ausgegeben werden „Die Geschwindigkeit ist nicht angemessen“. Verwenden Sie dazu den Ausdruck `and`.

Aufgabe 5.12

Schreiben Sie eine Funktion `temperatur_ist_unangenehm(temperatur)`, welche einen Parameter `temperatur` als Zahl entgegennimmt. Falls die Temperatur kleiner als 10 Grad oder grösser als 30 Grad ist, soll auf der Konsole ausgegeben werden: „Unangenehme Temperatur“. Ansonsten soll ausgegeben werden: „Angenehme Temperatur“. Verwenden Sie dazu den Ausdruck `or`.

Aufgabe 5.13

Entwickeln Sie eine Funktion `def vieleck_kreis(anzahl_ecken, umfang)`, die ein Vieleck zeichnet. Damit das Vieleck aussieht wie ein Kreis, soll es nur gezeichnet werden, wenn die Anzahl der Ecken grösser als 35 ist und wenn der Umfang mindestens 100 ist.

Aufgabe 5.14

Schreiben Sie eine Funktion `positiv_und_gerade(zahl)`, welche einen Parameter `zahl` entgegennimmt, und testet, ob die Zahl positiv *und* gerade ist, und falls dies zutrifft, den Text ausgibt `"Die Zahl ist positiv und gerade"`. Ansonsten soll nichts ausgegeben werden.

Zur Erinnerung: Eine Zahl ist gerade, wenn sie ohne Rest durch 2 teilbar ist. Der Rest einer Ganzzahldivision kann in Python mit dem Modulo-Operator `%` berechnet werden. Beispiel:

```
zahl % 2 == 0
```

Dieser Ausdruck gibt `True` zurück, wenn die Zahl gerade ist, da sie dann vollständig (ohne Rest) durch 2 teilbar ist. Andernfalls gibt der Ausdruck den Wert `False` zurück.

Aufgabe 5.15

Entwickeln Sie ein Programm, das alle natürlichen Zahlen zwischen 0 und 100 auf den Bildschirm schreibt, die durch 7, aber nicht durch 3 teilbar sind.

Aufgabe (Challenge) 5.16

Einen Code mit mehreren Bedingungen kann man statt mit `or` häufig auch mit `if`, `elif` und `else` umsetzen. Überlegen Sie sich, wie Sie den Code aus Aufgabe 5.12 mit `if`, `elif` und `else` statt mit `or` schreiben könnten.

In welchen Fällen ist es sinnvoller, `if`, `elif` und `else` zu verwenden? In welchen Fällen ist es sinnvoller `or` zu verwenden?

Aufgabe (Challenge) 5.17

Einen Code mit mehreren Bedingungen kann man statt mit `and` häufig auch mit verschachtelten `if`-Bedingungen umsetzen. Überlegen Sie sich, wie Sie den Code aus Aufgabe 5.11 mit verschachtelten `if`-Bedingungen statt mit `and` schreiben könnten.

In welchen Fällen ist es sinnvoller, `and` zu verwenden? In welchen Fällen sind verschachtelte `if`-Bedingungen besser geeignet?

Aufgabe (Challenge) 5.18

Diskutieren Sie den Gebrauch der Begriffe „und“ und „oder“ im Alltag und in der Informatik. Wo sehen Sie Unterschiede in der Verwendung dieser Begriffe?

Aufgabe (Challenge) 5.19

Das harmonische Mittel zweier Zahlen a und b ist eine wichtige Grösse in der Informatik, da es in vielen Algorithmen verwendet wird, beispielsweise in der Berechnung von Durchschnittswerten.

Beispiel: Wenn Sie 100 Kilometer mit 50 km/h und 100 Kilometer mit 100 km/h fahren, beträgt die Durchschnittsgeschwindigkeit nicht 75 km/h, sondern 66.67 km/h. Das harmonische Mittel kann in diesem Fall verwendet werden, um die Durchschnittsgeschwindigkeit zu berechnen:

Das harmonische Mittel zweier Zahlen a und b ist $\frac{2}{\frac{1}{a}+\frac{1}{b}}$. Es lässt sich aber nur berechnen, wenn weder a noch b null sind. Entwickeln Sie eine Funktion `def harmonisches_mittel(a, b)`, die für die Parameter a und b das harmonische Mittel ausrechnet, wenn sowohl a als auch b nicht null sind. Ansonsten gibt der Befehl den Text `"Das kann man nicht berechnen."` aus.

Aufgabe (Challenge) 5.20

Schreiben Sie eine Python-Funktion `def ist_schaltjahr(jahr)`, welche prüft, ob ein gegebenes Jahr ein Schaltjahr ist oder nicht.

Ein Jahr ist ein Schaltjahr, genau dann wenn gilt:

1. (das Jahr ist durch 400 teilbar) oder
2. (das Jahr ist durch 4 teilbar aber nicht durch 100)

Für die Prüfung auf Teilbarkeit sollen Sie den Modulo-Operator (`%`) verwenden.

5.1.5 Logische Ausdrücke negieren: `not`

Negieren bedeutet in der Informatik *nicht*, das Gegenteil einer (urpsrünglichen) Aussage zu machen, sondern *alle Aussagen zu machen*, welche durch die ursprüngliche Aussage nicht gemacht wurden, also alles „andere“ als die ursprüngliche Aussage zu sagen.

Beispiel 5.6:
Folgende Tabelle illustriert, was mit der Negation einer Aussage gemeint ist.

Aussage	Negation
„Das Mädchen heisst Elin“	„Das Mädchen heisst nicht Elin“
„Niemand in dieser Klasse ist volljährig“	„Mindestens eine Person in dieser Klasse ist volljährig“
$x > 2$	$x \leq 2$

Aufgabe 5.21

(Von Hand) Notieren Sie zu folgenden Aussagen die Negation, *ohne* die Wörter „nicht“ oder „kein“ zu verwenden:

- $x \geq 3$
- y ist eine negative Zahl
- in der Variable `test` ist der Wert `"Franz"` gespeichert.
- Im Auto sitzen mindestens drei Menschen
- Der Koffer ist leer
- Das Programm ist falsch geschrieben
- Die Anzahl der Jugendlichen in der Klasse ist genau 19
- Morgen wird es in Zürich mindestens 22 Grad Celsius warm

Die Negation einer Aussage kann in Python mit dem Ausdruck `not` (englisch für „nicht“) gemacht werden. Die Negation einer Aussage (mit `not`) ist insbesondere praktisch, um die Negation einer Aussage zu machen, ohne die Aussage komplett umschreiben zu müssen.

Beispiel 5.7:
Wir können den Code aus Beispiel 5.4 einfach mit dem Ausdruck `not` umschreiben, um auszugeben, unterwelchen Bedingungen wir *nicht* per Fahrrad zu Schule gehen wollen:

```
if not((temperature > 20) and (fitness > 80)):
    print("Ich gehe nicht per Fahrrad zur Schule!")
```

Man könnte dieselbe Aussage auch folgendermassen formulieren: „Falls ich mich nicht fit fühle *oder* das Wetter schlecht ist, gehe ich nicht per Fahrrad zur Schule“. Dies sähe in Python folgendermassen aus:

```
if (temperature <= 20) or (fitness <= 80):
    print("Ich gehe nicht per Fahrrad zur Schule!")
```

Weshalb haben wir im ersten Code ein `and` und im zweiten Code ein `or`? Dank dem `not` müssen wir nichts vom ursprünglichen Code in Beispiel 5.4 umformulieren, da wir mit dem `not` einfach alljene Fälle negieren, welche innerhalb der Klammer stehen. Somit kommen beide Codes zum selben Resultat.

Aufgabe 5.22

Schreiben Sie Ihren Code aus Aufgabe 5.14 so um, dass getestet wird, ob eine Zahl weder gerade noch positiv ist. Verwenden Sie dazu unter anderem den Ausdruck `not`. Testen Sie ihre Funktion für die Werte -3, +3, -4 und +4.

Schreiben Sie danach dieselbe Funktion nochmals, ohne den Ausdruck `not` zu verwenden.

Aufgabe 5.23

Entwickeln Sie ein Programm, das alle Zahlen von 1 bis 24 mit `print` ausgibt, die nicht Teiler von 24 sind. Verwenden Sie den Ausdruck `not`.

5.2 Fussgesteuerte Schleifen mit `break`

Bisher haben wir eine Art von Schleife gesehen: `for _ in range(...)`. Dabei gibt die Zahl innerhalb des Befehls `range(...)` an, wie viele Mal der Schleifenkörper wiederholt wird. Manchmal wissen wir jedoch nicht im voraus, wie viele Male eine Schleife wiederholt werden soll, wir kennen jedoch eine Bedingung, bei der die Schleife abgebrochen werden soll. Dies könnte beispielsweise der Fall sein, wenn wir eine Spirale zeichnen wolle, die immer grösser wird, bis die Seitenlänge eine gewisse maximale Länge `max_seite` erreicht hat.

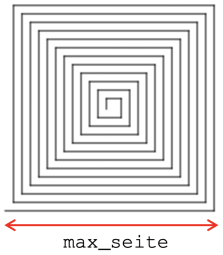


Abbildung 5.3: Bild einer Spirale, deren grösste Seitenlänge `max_seite` lang ist

Natürlich könnte man auch hier berechnen, wie viele Male die `for`-Schleife ausgeführt werden muss. Die Formel, um dies zu berechnen, wäre:

$$\left\lfloor \frac{\texttt{maxseite - seite}}{\texttt{add}} \right\rfloor + 1$$

Es geht allerdings auch einfacher, indem wir eine „unendliche“ Schleife starten, die wir abbrechen, sobald eine gewisse Kondition wahr ist.

Grundsätzlich können wir auch direkt die Werte `True` (Wahr) oder `False` (Falsch) in Logischen Ausdrücken verwenden:

Beispiel 5.8:
Der `print`-Befehl in folgendem Beispiel wird immer ausgeführt:

```
if True:
    print("Hello World")
```

Den Wert `True` könnten wir beispielsweise verwenden, um eine Endlosschleife mit `while True:` zu konstruieren.

```
import turtle as t

# Tempo der Turtle festlegen
t.speed(100)

# gleichseitiges Dreieck zeichnen
seite = 5
max_seite = 50
increment = 5
while True:                    <- Unendliche Schleife
    if seite > max_seite:
        break                  <- Schleifenabbruch
    t.fd(seite)
    t.rt(90)
    seite += increment

# Turtle-Zeichnung stehen lassen
t.done()
```

Aufgabe 5.24

Schreiben Sie eine Funktion `erraten_zahl()`, die ein einfaches Zahlenratespiel implementiert (siehe Code-Vorlage untenan). Die Funktion soll:

1. Eine zufällige Zahl zwischen 1 und 100 generieren (dies wird gemacht mit dem Befehl `random.randint(1, 100)`, ist im Code bereits gemacht).
2. Den Benutzer in einer Schleife auffordern, die Zahl zu erraten (`int(input("Rate die Zahl: "))`).
3. Falls die Eingabe kleiner als die gesuchte Zahl ist, soll ausgegeben werden: „Die Zahl ist grösser.“.
4. Falls die Eingabe grösser als die gesuchte Zahl ist, soll ausgegeben werden: „Die Zahl ist kleiner.“.
5. Falls die Eingabe korrekt ist, soll die Nachricht „Richtig! Du hast die Zahl erraten.“ ausgegeben werden und die Schleife mit `break` beendet werden.

Testen Sie die Funktion, indem Sie sie ausführen und versuchen, die Zahl zu erraten. Vervollständigen Sie folgende Code-Vorlage:

```
import random

def erraten_zahl():
    # Zufallszahl generieren
    ziel_zahl = random.randint(1, 100)

    while True:
        x = int(input("Errate die Zahl!"))
        # IHR CODE HIER
```

Aufgabe 5.25

Schreiben Sie eine Funktion `zeichne_spirale(seitenlaenge, winkel, increment)`, die eine Spirale zeichnet. Die Funktion soll folgende Parameter haben:

- `seitenlaenge`: Die Startlänge der ersten Seite.
- `winkel`: Der Winkel, um den sich die Turtle nach jeder gezeichneten Seite dreht.
- `increment`: Der Wert, um den die Seitenlänge nach jeder gezeichneten Seite erhöht wird.

Die Spirale soll so lange gezeichnet werden, bis die Seitenlänge 200 erreicht oder überschritten hat.

Aufgabe 5.26

Schreiben Sie eine Funktion `ist_quadrat(x)`, die überprüft, ob eine gegebene natürliche Zahl x eine Quadratzahl ist (also ob es eine ganze Zahl a gibt, so dass $x = a \cdot a$).

Das Programm soll mit $a = 1$ starten und überprüfen, ob $a \cdot a = x$. Falls $a \cdot a = x$, soll a ausgegeben werden und die Schleife abgebrochen werden. Ansonsten fährt man mit $a = a + 1$ weiter.

Sobald $a \cdot a > x$ soll die Schleife abgebrochen und ausgegeben werden: `x ist kein Quadrat.`

Verwenden Sie dazu eine `while True`-Schleife.

Aufgabe (Challenge) 5.27

Was gibt folgender Code aus und wann endet er?

```
i = 2
while True:
    if i>5:
        break
```

5.3 Kopfgesteuerte Schleifen mit `while`

Beispiel 5.9:
Folgendes Beispiel zeigt, wie ein `while True:` mit einem `break`-Befehl zu einem einfachen `while` mit Ausführungs-Bedingung vereinfacht werden kann. Beide Codes machen dasselbe. Die Schleife wird solange ausgeführt, wie die Ausführungs-Bedingung wahr ist. Die Abbruchskondition wird vor jeder neuen Schleifenausführung überprüft und die Schleife wird nur dann ausgeführt, wenn die Ausführungs-Bedingung noch wahr ist. Beim linken Code verwenden wir nicht eine Ausführungs-Bedingung sondern eine Abbruch-Bedingung.

```python
import turtle as t

def spirale(seite, add, max_seite):
while True:
    if seite > max_seite:
        break
    t.fd(seite)
    t.rt(90)
    seite+=add

spirale(10, 10, 100)
```

```python
import turtle as t

def spirale(seite, add, max_seite):
    while seite < max_seite:
        t.fd(seite)
        t.rt(90)
        seite+=add

spirale(10, 10, 100)
```

Achtung

Wichtiger Hinweis 5.1 (Endlos-Schleifen)**:**
Beachten Sie folgende Punkte:

- Passen Sie auf, dass Sie keine unendlichen Schleifen produzieren! In diesem Fall kann das Programm, auf dem Python läuft, hängen bleiben. Vergessen Sie daher nie die Abbruchkondition klar zu formulieren!
- Ein `while True:` (unendliche Schleife) ohne `break` kann zum Absturz des Programms führen 🥹
- Je nachdem kann das auch in einem `while` mit einer Kondition passieren, sofern die Bedingung nach dem `while` so geschrieben ist, dass sie immer wahr (`True`) ist (siehe Unterabschnitt 3.7.3 zu semantischen Fehlern).
- **Speichern Sie regelmässig Ihre Aufgaben!**
- Falls das Programm VS Code hängenbleibt: Beenden mit Ctrl + Alt + ⌫ (Windows), bzw. Activity Monitor unter MacOS

Aufgabe 5.28

Schreiben Sie eine Funktion `verdreifache_bis_ueber1Mio(zahl)`, welche eine Zahl `zahl` so lange immer wieder verdreifacht, bis `zahl` erstmals grösser als 1'000'000 ist. Dabei sollen alle Zwischenresultate in der Konsole ausgegeben werden. Am Schluss soll ausserdem die Anzahl Verdreifachungen ausgedruckt werden.

Aufgabe 5.29

Entwickeln Sie eine Funktion, dem eine Zahl $x > 1$ als Parameter übergeben wird. Aus x wird nun eine Folge von Zahlen generiert und ausgegeben. Dabei wird folgende Regel angewendet: Wenn x durch zwei teilbar ist, ist die nächste Zahl $x/2$. Wenn x nicht durch zwei teilbar ist, ist die nächste Zahl $3 \cdot x + 1$. Dieser Prozess wird wiederholt, solange die neu berechnete Zahl grösser als 1 ist.

Aufgabe 5.30

Entwickeln Sie ein Programm, das eine siebeneckige Spirale von aussen nach innen zeichnet. Die Startlänge der Seite und die Verkleinerung der Seite in jedem Schritt sollen Parameter sein. Verwenden Sie eine `while`-Schleife und lassen Sie die Spirale so lange zeichnen, wie die Seitenlänge grösser als `10` ist.

Aufgabe 5.31

Gegeben ist eine natürliche Zahl `x > 1`. Ein echter Teiler einer Zahl ist eine Zahl, die grösser als 1 und kleiner als `x` ist und `x` ohne Rest teilt. 10 hat beispielsweise die Teiler 1, 2, 5 und 10, wovon nur 2 und 5 *echte* Teiler sind.

Schreiben Sie ein Programm, das für eine vom Benutzer eingegebene Zahl `x` mit einer `while` -Schleife den kleinsten und den grössten echten Teiler von `x` bestimmt.

Gibt es keine echten Teiler (d.h. `x` ist eine Primzahl), soll das Programm ausgeben `"x ist eine Primzahl."`. Andernfalls sollen der kleinste und der grösste echte Teiler ausgegeben werden.

Aufgabe 5.32

Wenn man die folgende Zahlenfolge von Brüchen immer weiter addiert, wird die Summe immer grösser:

$$\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \dots$$

Wie viele Brüche müssen addiert werden, damit die Summe mindestens gleich einer gegebenen Zahl x wird?

Schreiben Sie ein Python-Programm, das die Brüche addiert, solange deren Summe kleiner als x ist. Am Ende soll das Programm ausgeben, welcher Nenner beim letzten hinzugefügten Bruch verwendet wurde und wie gross die Summe insgesamt ist.

Wenn über 100 Brüche addiert wurden, soll die `while`-Schleife abgebrochen werden (mit `break`).

Aufgabe (Challenge) 5.33

Wie viele Schleifen durchläuft das folgende Programm?

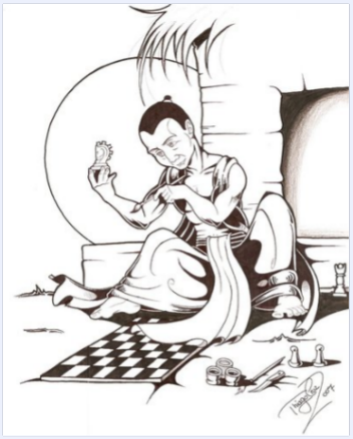
```
a = 1
summe = 0

while summe < 2:
    summe += 1 / a
    a *= 2
    print("a =", a)
    print("Summe =", summe)
```

Programm 5.31: `schleifen_wiederh.py`

Aufgabe 5.34

Schreiben Sie ein Programm, das den Benutzer wiederholt auffordert, Wörter einzugeben. Sobald das Wort „Voldemort“ eingegeben wird, soll der Prozess beendet werden. Der Computer soll anschliessend eine Aneinanderreihung aller vor „Voldemort“ eingegebenen Wörter ausgeben.

Aufgabe 5.35 Weizenkornlegende



Sissa ibn Dahir lebte angeblich im dritten oder vierten Jahrhundert in Indien und gilt Legenden zufolge als der Erfinder des Schachspiels.

Der indische Herrscher Shihram tyrannisierte seine Untertanen und stürzte sein Land in Not und Elend. Um die Aufmerksamkeit des Königs auf seine Fehler zu lenken, ohne seinen Zorn zu entfachen, schuf der weise Brahmane Sissa ein Spiel, in welchem der König als wichtigste Figur ohne Hilfe anderer Figuren und Bauern nichts ausrichten kann. Der Unterricht im Schachspiel machte auf den Herrscher Shihram einen starken Eindruck. Er wurde milder und liess das Schachspiel verbreiten, damit alle davon Kenntnis nehmen.

Um sich für die anschauliche Lehre von Lebensweisheit und zugleich Unterhaltung zu bedanken, gewährte er dem Brahmanen einen freien Wunsch. Dieser wünschte sich Weizenkörner: Auf das erste Feld eines Schachbretts wollte er ein Korn, auf das zweite Feld das Doppelte, also zwei, auf das dritte wiederum die doppelte Menge, also vier und so weiter.

Sie sollen die Menge Weizenkörner, welche Sissa vom Herrscher gefordert hat, berechnen. Gehen Sie folgendermassen vor:

Auf dem ersten Schachfeld liegt 1 Reiskorn, auf dem zweiten Feld liegen 2 Reiskörner, auf dem dritten Feld liegen 4 Körner usw.

Allgemein liegen auf dem $n+1$-ten Schachfeld genau doppelt soviele Körner wie auf dem n-ten Schachfeld. Unser Schachbrett habe n-Felder, wobei $n \leq 67$. Wie viele Reiskörner würden in dieser Situation insgesamt auf dem Schachfeld liegen (angenommen so viel Reis hätte Platz)?

Schreiben Sie eine Funktion `reis(n)`, welche die gesuchte Anzahl Reiskörner in Abhängigkeit der Anzahl n der Felder berechnet und per `return` zurückgibt.

- `reis(3)` sollte 7 ausgeben.

- `reis(8)` sollte 255 ausgeben.
- `reis(64)` sollte 18446744073709551615 ausgeben.

Kapitel 6

Datenstrukturen

6.1 Listen

6.1.1 Einführung in Listen

Bisher haben wir uns mit Variablen beschäftigt, die nur einen Wert speichern können::

- `x = 3` (eine Zahl / „integer“)
- `name = "Hallo"` (einen Text / „string“)
- `ist_wahr = True` (einen Wahrheitswert / „boolean“)

In der Informatik ist es jedoch oft nötig, mit vielen Werten gleichzeitig arbeiten zu können, gerade im Kontext von *Big Data*. Eine Möglichkeit, in Python mit vielen Werten zu arbeiten, sind **Listen**.

Definition 6.1 (Liste)**:**
Eine **Liste** ist eine Sammlung von Werten, die in einer Variable gespeichert werden können. Eine Liste kann beliebig viele Werte enthalten und diese Werte können von einem beliebigen Typ wie zum Beispiel „integer“, „string“ oder „boolean“ sein.

Beispiel 6.1 (Listen erstellen)**:**
Folgendes Beispiel zeigt, wie eine Liste in Python definiert wird. Wir können dabei, wie auch bei anderen Variablentypen, beliebige Namen verwenden. Die Inhalte der Liste werden in eckigen Klammern `[]` geschrieben und die einzelnen Werte werden durch Kommas `,` getrennt.

```
# Liste, die nur Zahlen enthält
liste_1 = [1, 2, 3, 4, 5]
# Liste, die nur Strings enthält
liste_2 = ["Hallo", "Welt", "Python"]
# Liste, die nur Wahrheitswerte (Typ bool) enthält
liste_3 = [True, False, True]
# Liste, die Werte von verschiedenen Typen enthält
liste_4 = [1, "Hallo", True, 3.14]
```

Beispiel 6.2 (Zugriff auf Listen)**:**
Wenn wir auf die einzelnen Werte in der Liste zugreifen möchten, können wir dies mit dem

Index tun. Der Index ist eine Zahl, die angibt, an welcher Stelle sich der Wert in der Liste befindet. Der Index beginnt bei 0, das heisst, der erste Wert in der Liste hat den Index 0, der zweite Wert hat den Index 1 und so weiter.

```
# Liste a erstellen
a = [1, 2, 3, 4, 5]
# Zugriff auf das erste Element der Liste
print(a[0]) # gibt 1 aus
# Zugriff auf das zweite Element der Liste
print(a[1]) # gibt 2 aus
# Zugriff auf das letzte Element der Liste
print(a[-1])
```

Beispiel 6.3 (Listen-Werte verändern)**:**
Um den Wert an einer bestimmten Stelle in der Liste zu ändern, können wir ebenfalls den Index verwenden. Wir können den Wert an dieser Stelle einfach durch einen neuen Wert ersetzen.

```
# Liste a erstellen
a = [1, 2, 3, 4, 5]
# Ändern des Wertes an der Stelle 0 in der Liste a
a[0] = 10
print(a) # gibt [10, 2, 3, 4, 5] aus
```

Beispiel 6.4 (Länge einer Liste)**:**
Der Befehl `len(liste)` gibt die Länge der Liste zurück, also die Anzahl der Werte, die in der Liste gespeichert sind. Dies ist nützlich, wenn wir wissen möchten, wie viele Werte in der Liste enthalten sind.

Listen erfüllen vielfältige Aufgaben in der Informatik. Sie können verwendet werden, um Daten zu speichern, zu sortieren, zu filtern und zu analysieren. In Python gibt es viele eingebaute Funktionen und Methoden, die speziell für Listen entwickelt wurden, um diese Aufgaben zu erleichtern.

Beispiel 6.5 (Zahlen-Liste summieren)**:**
Folgendes Beispiel zeigt auf, wie eine Liste in einer Funktion verwendet werden kann, um eine Summe zu berechnen. Die Funktion `berechne_summe` nimmt eine Liste von Zahlen als Eingabe und gibt die Summe dieser Zahlen zurück.

```
def summiere(daten):
    i = 0  # Hilfs-Index, um auf Elemente von daten zuzugreifen
    summe = 0  # Variable, um alle Elemente von "daten" zu summieren

    # Jedes Element von Daten zu Summe hinzufügen
    for _ in range(len(daten)):
        summe += daten[i]  # Wert von daten[i] zu Summe hinzufügen
        i += 1  # Hilfs-Index um 1 vergrössern (Werte?)

    print(summe)
```

```
summiere([4, 2, -6, 17, 5, 12])  # Ausgabe: 34
```

Programm 6.1: `summe.py`

Beispiel 6.6 (Schleifen ohne Index)**:**
Etwas effizienter kann auf jedes Element der Liste mit dem Befehl `for zahl in liste` zugegriffen werden. Dabei wird die Variable `zahl` nacheinander auf jedes Element der Liste gesetzt.

```
def summiere(daten):
    summe = 0
    for zahl in daten:
        summe += zahl
    print(summe)


summiere([4, 2, -6, 17, 5, 12])  # Ausgabe: 34
```

Programm 6.2: `summe_for_zahl_in.py`

Aufgabe 6.1

Erstellen Sie eine Funktion `vergroessere_um_fuenf(liste)`, die mithilfe einer Schleife jeden Wert in der Liste `daten = [20, -7, 8, 2, 1, 6]` um 5 erhöht. Die Anzahl der Wiederholungen der Schleife soll dabei mit `len()` bestimmt werden. Kontrollieren Sie Ihr Programm mit `print(daten)`.

Aufgabe 6.2

Entwickeln Sie eine Funktion `berechne_durchschnitt(liste)`, die für die Liste (z.B. `[5, 0, -2, 3, 51, 8, 13, -100, -10, -1]`) den Durchschnittswert der Beträge aller Elemente berechnet und mit `print()` ausgibt.

Der Durchschnitt einer Liste von Zahlen ist die Summe aller Zahlen geteilt durch die Anzahl der Zahlen.

Aufgabe 6.3

Erstellen Sie ein Programm, das alle geraden Zahlen in der Liste `daten = [5, 7, 8, 6, 3]` verdoppelt. Kontrollieren Sie Ihr Programm mit `print(daten)`.

Aufgabe 6.4

Das Skalarprodukt wird in vielen Lebensbereichen verwendet, z.B. in der Mathematik, Physik und Informatik. Im Alltag begegnen wir dem Skalarprodukt häufig in der Finanzwelt, z.B. bei der Berechnung des Gesamtpreises von Produkten:

Produkt	**Menge** `m`	**Preis** `p`
🍌	800 g	2.- / kg
🍎	1200 g	2.50 / kg
🍑	2300 g	5.- / kg

Schreiben Sie eine Funktion, die das Skalarprodukt zweier Listen berechnet. Das Skalarprodukt ist die Summe der Produkte der jeweils entsprechenden Elemente beider Listen. Beispiel: Für die Listen `m = [0.8, 1.2, 2.3]` (in kg) und `p = [2.0, 2.5, 5.0]` (Preis pro kg) berechnet das Skalarprodukt den Gesamtpreis.

Wie berechnen wir den **Gesamtpreis?** Dies kann mit dem **Skalarprodukt** gemacht werden: `m[1] * p[0] + m[1] * p[1] + ... + m[-1] * p[-1]`. Zur Erinnerung: `m[-1]` gibt uns das letzte (hinterste) Element der Liste `m`.[a]

[a] Alternativ können wir anstelle von `m[-1]` auch `m[len(m) - 1]`

Beispiel 6.7 (Kleinste Zahl in einer Liste)**:**
Mit folgendem Code können wir die kleinste Zahl in einer Liste finden. Wir verwenden eine Schleife, um alle Zahlen in der Liste zu durchlaufen und die kleinste Zahl zu finden. Der Code gibt am Schluss die kleinste Zahl in der Liste aus.

```
def finde_kleinste_zahl(liste):
    kleinste_zahl = liste[0]
    index = 0
    for _ in range(len(liste)):
        if liste[index] < kleinste_zahl:
            kleinste_zahl = liste[index]
        index += 1

    print(kleinste_zahl)  # Kleinste Zahl ausgeben


# Beispielaufruf der Funktion
finde_kleinste_zahl([3, 1, 4, 1, 5, 9, 2, 6, 5, 3, -5])  # gibt -5 aus
```

Programm 6.7: `findmin.py`

Aufgabe 6.5

Verändern Sie den Code aus Beispiel 6.7 so, dass die Funktion `finde_kleinste_zahl` nicht nur die kleinste Zahl in einer Liste von Zahlen ausgibt, sondern auch deren Position (Index) in der Liste.

Aufgabe 6.6

Schreiben Sie eine Funktion, welche gleichzeitig den grössten und den kleinsten Wert sowie deren Indizes (Positionen) in einer Liste von Zahlen zurückgibt.

Aufgabe 6.7

Schreiben Sie eine Funktion, die zählt, wie oft die Zahl 10 in einer Liste von Zahlen vorkommt. Testen Sie Ihre Funktion mit der Liste `[1, 2, 3, 10, 4, 10, 5]`. Die Funktion soll die Anzahl der Vorkommen von 10 ausgeben (2 in diesem Fall).

Aufgabe (Challenge) 6.8

Schreiben Sie eine Funktion, die überprüft, ob eine Liste von Zahlen sortiert ist oder nicht. Falls die Liste sortiert ist, soll die Definition den Wert `True` zurückgeben, ansonsten den Wert `False`.

Tipps:

1. Gehen Sie jedes Element der Liste mit einer Schleife durch, und überprüfen Sie, dass das Element grösser oder gleich dem vorigen Element ist.
2. `return` bricht die Funktion ab und gibt einen Wert an das Hauptprogramm zurück. Sie können also, sobald eine Zahl in falscher Reihenfolge gefunden worden ist, direkt `False` zurückgeben.
3. Falls Sie nie ein falsches Element gefunden haben, geben Sie am Ende der Definition `True` zurück.

Aufgabe 6.9

Eine Supermarkt-Kette bittet Sie, die Rabatte auf ausgewählte Produkte zu berechnen. Sie gibt Ihnen hierzu zwei Listen: die erste Liste enthält die Preise der Produkte *ohne Rabatt* und die zweite Liste enthält die Rabatte in Prozent.

```
# Preise in Franken
liste_preise = [39.95, 65.95, 66.95, 76.95, 9.95, 10.95, 13.95]
# Rabatte in Prozent
liste_rabatte = [30, 40, 30, 35, 20, 15, 35]
```

Das Beispiel bedeutet, dass das erste Produkt nicht 39.95, sondern 30 % weniger als 39.95 kosten sollte, also 70 % von 39.95 = 27.965.

Der rabattierte Preis eines einzelnen Produkts wird folgendermassen berechnet:

```
preis_rabattiert = preis_normal * (1 - rabatt_in_prozent / 100)
```

Schreiben Sie eine Definition, die die rabattierten Preise für alle Produkte berechnet und als Liste auf der Konsole ausgibt. Sie müssen die Preise nicht runden.

Aufgabe 6.10

Sie hatten zum Mittagessen ein Sandwich sowie einen Energy-Drink. Sie möchten gerne wissen, wie viele Kalorien das gesamte Mittagessen hatte, die Nährwerte sind jedoch nur pro 100 Gramm oder 100 Milliliter (wobei 100 Gramm = 100 Milliliter sind) angegeben. Berechnen Sie die Gesamt-Kalorien, indem Sie folgende zwei Listen erstellen:

- `l_kcal`: Liste der Kalorien pro 100 Gramm (bzw. 100 Milliliter) für das Sandwich und den Energy-Drink.
- `l_gram`: Liste der Mengen in Gramm, bzw. Milliliter für das Sandwich und den Energy-Drink.

Die Nährwert-Informationen entnehmen Sie dem Bild Abbildung 6.1

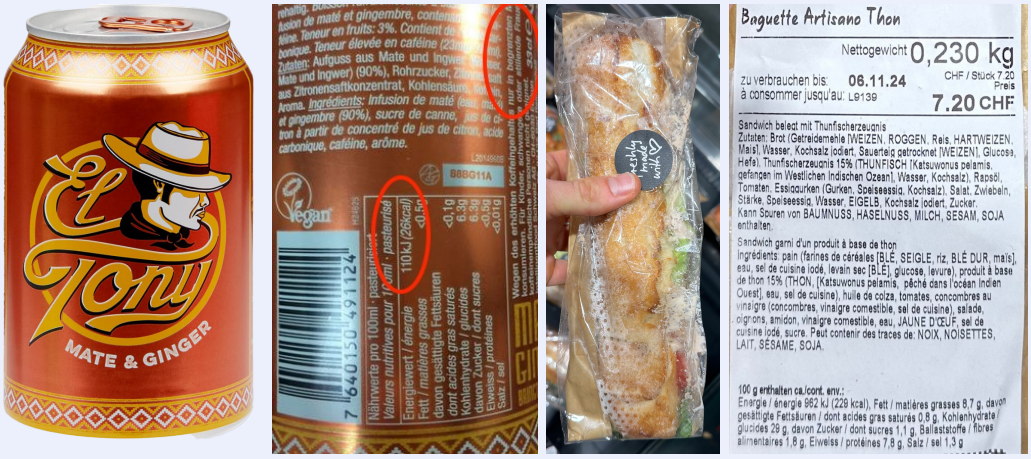


Abbildung 6.1: Mittagessen und dazugehörige Kalorien-Informationen

Aufgabe (Challenge) 6.11

Schreiben Sie nun eine Definition, mit der Sie beliebig viele Nährwerte und Mengenangaben mittels Input eingeben. Die Definition soll folgendes machen:

1. Mittels `input("...")` nach einer Kalorienangabe für ein Lebensmittel fragen (pro 100 Gramm).
2. Mittels `input("...")` nach der konsumierten Menge für dasselbe Lebensmittel fragen.
3. Mittels `input("...")` fragen, ob noch weitere Lebensmittel hinzukommen.

Schritt 1-2 sollen so lange wiederholt werden, bis in Schritt 3 `False` eingegeben wird.

6.1.2 Algorithmen

6.1.2.1 Sortier-Algorithmen

Eine der häufigsten Anwendungen in der Informatik ist das Sortieren von Daten. Es gibt viele verschiedene Algorithmen, um Daten zu sortieren, und jeder Algorithmus hat seine eigenen Vor- und Nachteile, insbesondere hinsichtlich der Geschwindigkeit und der benötigten Rechenleistung. Im Folgenden wird der Bubble-Sort-Algorithmus vorgestellt, der eine einfache Methode ist, um Daten zu sortieren. Der Bubble-Sort-Algorithmus funktioniert, indem er die Liste von Werten durchläuft und benachbarte Werte vergleicht. Wenn ein Wert grösser ist als der nächste Wert, werden die

beiden Werte vertauscht. Dieser Vorgang wird so lange wiederholt, bis die gesamte Liste sortiert ist. Die ersten zwölf Schritte des Bubble-Sort-Algorithmus sind in Abbildung 6.2 dargestellt. Der Algorithmus wird so lange wiederholt, bis die gesamte Liste sortiert ist.

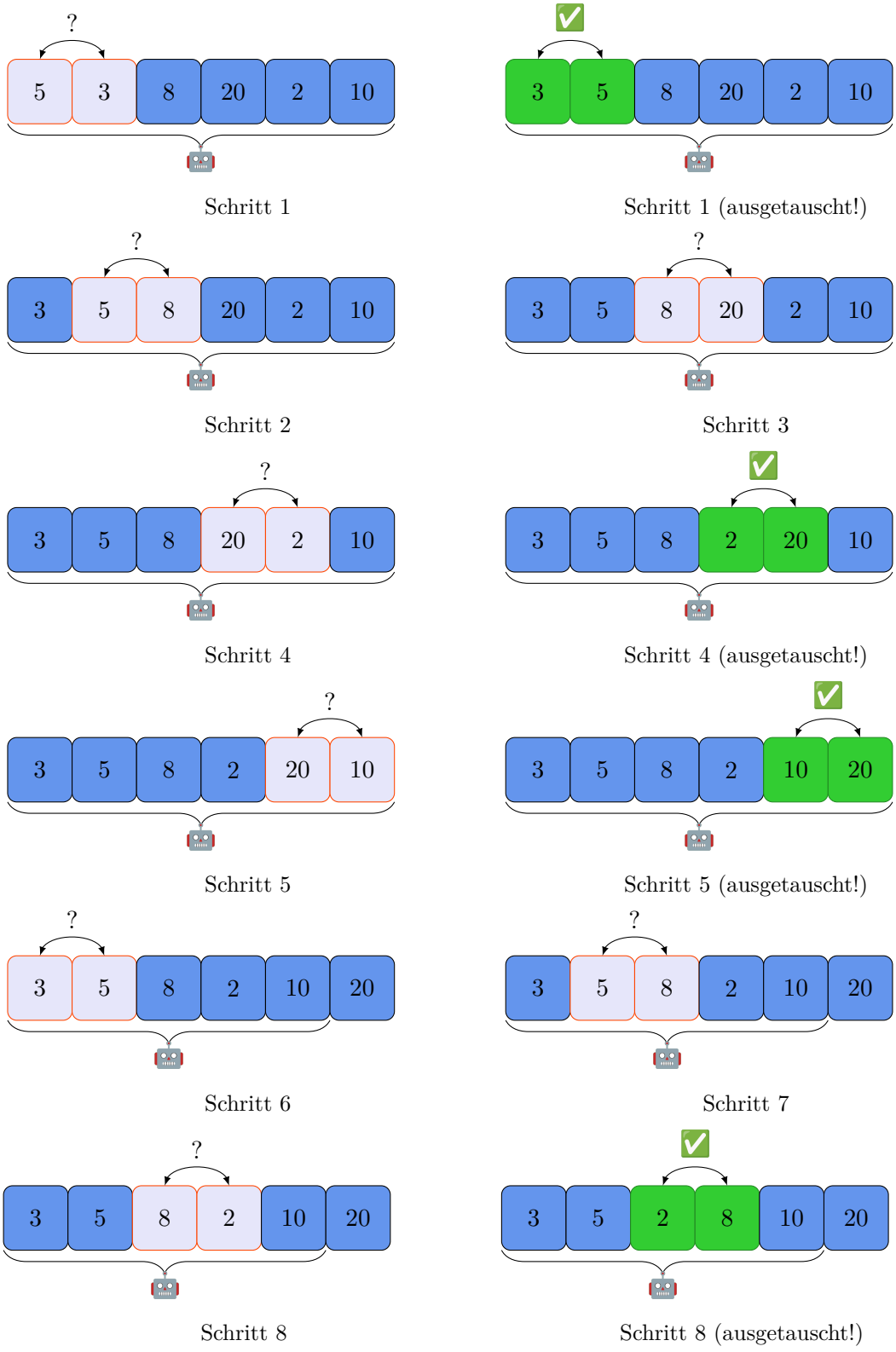


Abbildung 6.2: Bubble-Sort-Algorithmus (erste 8 Schritte)

Folgender Code zeigt, wie der Bubble-Sort-Algorithmus in Python implementiert werden kann. Der Algorithmus wird in einer Funktion `bubble_sort` definiert, die eine Liste von Zahlen als Eingabe erhält und die sortierte Liste zurückgibt. Die Funktion verwendet eine Schleife, um die Liste zu durchlaufen und benachbarte Werte zu vergleichen. Wenn ein Wert grösser ist als der nächste Wert, werden die beiden Werte vertauscht. Dieser Vorgang wird so lange wiederholt, bis die gesamte Liste sortiert ist.

```
def bubble_sort(liste):
    # Initialisiere den äusseren Schleifenzähler
    i = 0
    # Äussere Schleife: Wiederhole den Sortiervorgang n-mal
    for _ in range(len(liste) - 1):
        # Initialisiere den inneren Schleifenzähler
        j = 0
        # Innere Schleife: Vergleiche benachbarte Elemente
        for _ in range(len(liste) - 1 - i):
            # Wenn das aktuelle Element grösser als das nächste ist, tausche sie
            if liste[j] > liste[j + 1]:
                temp = liste[j]  # Temporäre Variable zum Speichern des Werts
                liste[j] = liste[j + 1]  # Tausche die Werte
                liste[j + 1] = temp  # Setze den gespeicherten Wert an die neue
   Position
            j += 1  # Erhöhe den inneren Schleifenzähler
        i += 1  # Erhöhe den äusseren Schleifenzähler

    # Gib die sortierte Liste aus
    print(liste)


# Beispielverwendung
numbers = [64, 34, 25, 12, 22, 11, 90]
bubble_sort(numbers)
```

Programm 6.15: `bubble_sort_w_comments.py`

Aufgabe 6.12

Schauen Sie sich den Python-Code für Bubble an sowie die folgenden drei Listen:

- `x = [3, 4, 1, -3, 6]`
- `x = [3, 4, 5, 6, 7]`
- `x = [7, 6, 5, 4, 3]`

Notieren Sie von Hand, wie die Listen nach jedem Durchgang der äusseren Schleife aussehen (für eine Liste von Länge 5 sollten Sie beispielsweise 4 Zwischenresultate notieren). Kontrollieren Sie Ihr Resultat, indem Sie das Programm mit diesen Listen ausführen.

Der Algorithmus hat eine Zeitkomplexität von $O(n^2)$, was bedeutet, dass die Laufzeit des Algorithmus quadratisch mit der Anzahl der Werte in der Liste wächst. Dies macht den Bubble-Sort-Algorithmus für grosse Datenmengen ineffizient. Aufgrund der inneren Schleife (`for _ in range(n - 1 - i)`) kann die Laufzeit noch geringfügig verbessert werden: In jedem Durchlauf der äusseren Schleife ist das jeweils grösste Element bereits an die richtige Position gewandert, sodass die innere Schleife jedes Mal um ein Element kürzer wird.

- **nicht optimierte Version:** Die Anzahl der Vergleiche beträgt

$$(n-1)^2$$

  Das bedeutet: Für eine Liste mit 6 Elementen sind das $5^2 = 25$ Vergleiche.
- **optimierte Version:** Die Anzahl der Vergleiche entspricht der Summe der Zahlen von 1 bis $n-1$:

$$(n-1) + (n-2) + \ldots + 2 + 1 = \sum_{m=1}^{n-1} m.$$

  Für eine Liste mit 6 Elementen ergibt das $\sum_{m=1}^{5} m = 1+2+3+4+5 = 15$ Vergleiche.
- Im Mathematikunterricht lernen Sie, dass die Gleichheit

$$(n-1) + (n-2) + \ldots + 2 + 1 = \sum_{m=1}^{n-1} m = \frac{1}{2}n^2 + \frac{1}{2}n$$

  gilt. Damit bleibt die Anzahl der Vergleiche von Bubble-Sort auch mit dieser Optimierung im Wesentlichen quadratisch.

Im Allgemeinen wächst die Anzahl der Vergleiche beim Bubble-Sort-Algorithmus quadratisch mit der Länge der Liste, also $O(n^2)$. Die Optimierung reduziert die Anzahl der Vergleiche, ändert aber nichts an der grundsätzlichen Komplexität.

6.1.2.2 Such-Algorithmen

In der Informatik stellt sich häufig die Frage, wie man möglichst schnell herausfinden kann, ob eine bestimmte Zahl in einer Liste enthalten ist.

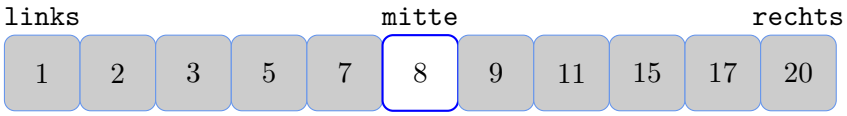
Ist die Liste unsortiert, wie zum Beispiel bei `x = [5, 3, 8, 20, 2, 10]`, bleibt uns nichts anderes übrig, als jedes Element der Liste einzeln zu überprüfen. Dies entspricht einer linearen Suche, bei der im ungünstigsten Fall alle Elemente betrachtet werden müssen.

Ist die Liste jedoch bereits sortiert, wie zum Beispiel bei `x = [2, 3, 5, 8, 10, 20]`, können wir effizientere Suchverfahren anwenden. Dies lässt sich mit der Suche nach einem Gegenstand in einem Koffer vergleichen. Wenn der Koffer unordentlich gepackt ist, müssen wir jeden einzelnen Gegenstand herausnehmen, um den gesuchten zu finden. Wenn der Koffer jedoch ordentlich gepackt ist, können wir die Gegenstände viel schneller finden (siehe Abbildung 6.3).

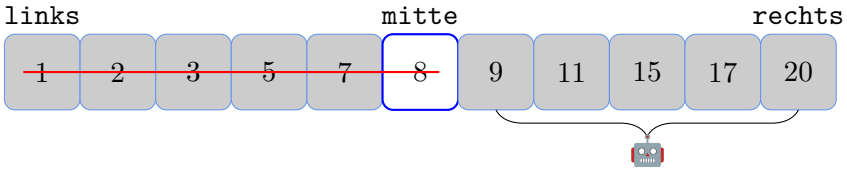
Abbildung 6.3: Unordentlich gepackter Koffer vs. ordentlich gepackter Koffer

Ein besonders schneller Algorithmus ist die sogenannte **binäre Suche**, bei der die Liste immer wieder halbiert wird, um das gesuchte Element zu finden.

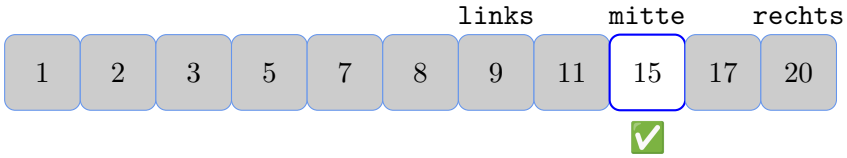
Das Suchen bezeichnet allgemein den Vorgang, ein bestimmtes Element in einer Datenmenge möglichst effizient zu finden. Im Folgenden lernen wir die binäre Suche als effizienten Such-Algorithmus für sortierte Listen kennen.



(a) Schritt 1



(b) Schritt 1 (nächster Vergleich)



(c) Schritt 2

Der Algorithmus zur Umsetzung der binären Suche ist in Python wie folgt implementiert:

```python
def binaere_suche(liste, ziel):
    # Definiere die Start- und Endpunkte des Suchbereichs
    links = 0
    rechts = len(liste) - 1

    # Solange der Suchbereich gültig ist, also links <= rechts
    while links <= rechts:
        # Berechne das mittlere Element
        mitte = (links + rechts) // 2

        # Wenn das mittlere Element das gesuchte Ziel ist, gib den Index zurück
        if liste[mitte] == ziel:
            print("Ziel Gefunden an Position", mitte)
            break

        # Wenn das Ziel grösser ist als das mittlere Element,
        # dann ist das Ziel im rechten Teil der Liste
        elif liste[mitte] < ziel:
            links = mitte + 1

        # Wenn das Ziel kleiner ist als das mittlere Element,
        # dann ist das Ziel im linken Teil der Liste
        else:
            rechts = mitte - 1


# Beispiel-Liste (muss sortiert sein)
meine_liste = [2, 3, 4, 10, 40]
ziel = 10

# Binäre Suche aufrufen
```

```
32 binaere_suche(meine_liste, ziel)
```

Programm 6.17: `binary_search.py`

Bei solchen, etwas komplexeren Codes, kann es nützlich sein, sich die Entwicklung der Variablenwerte im Verlauf der Ausführung des Codes von Hand zu notieren (siehe Tabelle 6.1).

Code \ Variable	`links`	`mitte`	`rechts`
1. `while`			
2. `while`			
3. `while`			
...			

Tabelle 6.1: Beispielhafte Zeit-Tabelle für die binäre Suche (zum Ausfüllen), jeweils nach Zeile 9

Beispiel 6.8:
Wir können für folgende Liste `[4, 8, 9, 11, 15, 23, 42]` die binäre Suche verwenden. Der Code sucht nach der Zahl `9` und gibt den Index der Zahl in der Liste zurück. Wir evaluieren jeweils die Werte der Variablen `links`, `mitte` und `rechts` nach Zeile 9, um den Ablauf des Codes zu verstehen. Die Tabelle Tabelle 6.2 zeigt die Werte der Variablen nach jedem Schritt der Schleife.

Code \ Variable	`links`	`mitte`	`rechts`
1. `while`	0	3	6
2. `while`	0	1	2
3. `while`	2	2	2
Gefunden!			

Tabelle 6.2: Beispielhafte Zeit-Tabelle für die binäre Suche, jeweils nach Zeile 9 evaluiert

Aufgabe 6.13

Verwenden Sie die folgende Liste: `[-20, -17, -13, -13, 2, 5, 7, 7, 9, 10]`. Vollziehen Sie den Ablauf des Programms für folgende Werte

1. -20
2. 6

Zeichnen Sie eine Zeit-Tabelle wie in Tabelle 6.1 und überprüfen Sie Ihre Resultate, indem Sie zwischen den Zeilen Zeilen 9 und 10 `print`-Befehle verwenden, um die Werte von `links`, `mitte` und `rechts` auszugeben.

Aufgabe 6.14

Ändern Sie den Code für die binäre Suche so ab, dass ein `while True:` gemeinsam mit einer boolsche Variable `gefunden` sowie dem Befehl `break` verwendet wird.

6.1.3 Listen verändern

Häufig müssen Listen verändert werden, beispielsweise um Elemente hinzuzufügen oder zu entfernen, oder um diese neu zu ordnen. Python bietet hierfür verschiedene Methoden an. Methoden sind Funktionen, die auf bestimmte Variablen angewandt werden. In 7.1 werden wir lernen, wie Methoden definiert werden können. Hier lernen wir einige vordefinierte Methoden für Listen kennen.

Definition 6.2 (Listen verändern)**:**
Listen könnnen in Python dynamisch verändert werden, indem Elemente hinzugefügt oder entfernt werden. Einige der wichtigsten Methoden sind:

- `liste.append(wert)` fügt **am Ende** der Liste einen neuen Wert hinzu.
- `liste.insert(index, wert)` fügt an der angegebenen Position (`index`) einen neuen Wert ein. Alle nachfolgenden Elemente werden nach rechts verschoben.
- `liste.pop(index)` entfernt das Element an der angegebenen Position (`index`) und gibt es zurück. Wird kein Index angegeben, wird das letzte Element entfernt.

Weitere Methoden zum Verändern von Listen sind in der offiziellen Python-Dokumentation aufgelistet.

Beispiel 6.9 (Elemente hinzufügen und entfernen)**:**
Folgender Code zeigt auf, wie Listen in Python verändert werden können. Wir erstellen eine Liste von Zahlen und fügen dann neue Zahlen hinzu, entfernen das letzte Element und das erste Element. Die Ergebnisse werden anschliessend ausgegeben.

```
zahlen = [1, 2, 3]
zahlen.append(4)  # [1, 2, 3, 4]
zahlen.insert(1, 10)  # [1, 10, 2, 3, 4]
letztes = zahlen.pop()  # entfernt 4, jetzt [1, 10, 2, 3]
erstes = zahlen.pop(0)  # entfernt 1, jetzt [10, 2, 3]
print(zahlen)  # [10, 2, 3]
print(letztes)  # 4
print(erstes)  # 1
```

Programm 6.19: `dynamic_lists.py`

Aufgabe 6.15

Fügen Sie der Liste `fruechte = ["Apfel", "Banane"]` zuerst `"Orange"` am Ende hinzu, dann `"Kiwi"` an der zweiten Stelle. Entfernen Sie danach das erste Element der Liste und geben Sie die Liste aus.

Aufgabe 6.16

Erstellen Sie eine leere Liste `zahlen`. Fügen Sie mit einer Schleife die Zahlen 1 bis 5 mit `append` hinzu. Entfernen Sie dann das Element an der dritten Stelle mit `pop` und geben Sie die Liste aus.

Aufgabe 6.17

Gegeben ist die Liste `farben = ["rot", "blau", "grün"]`. Fügen Sie `"gelb"` an der zweiten Stelle ein und entfernen Sie das letzte Element mit `.pop()`. Geben Sie die veränderte Liste aus.

Aufgabe 6.18

Gegeben seien zwei gleich lange Listen `A` und `B`.

Schreiben Sie eine Python-Funktion `verschmelzen(A, B)`, welche die beiden Listen zu einer Liste `C` zusammenfügt. Das Zusammenfügen soll „reissverschlussartig“ geschehen: Elemente aus `A` und `B` sollen sich in `C` abwechseln, beginnend mit einem Element aus `A`. Betrachten Sie dazu die Beispiele. Schliesslich soll die Liste `C` mit print ausgegeben werden.

```
# Beispiel:
verschmelzen([4, 2], [5, 9]) # Ausgabe: [4, 5, 2, 9]
verschmelzen([1, 1, 1], [2, 2, 1]) # Ausgabe: [1, 2, 1, 2, 1, 1]
```

Aufgabe 6.19

Gegeben sei eine Liste `A` von ganzen Zahlen.

Schreiben Sie eine Python-Funktion `entferne_duplikate(A)`, welche systematisch eine Liste `B` aufbaut, welche genau die Elemente von `A` enthält aber ohne Duplikate (mehr als einmal vorkommende Elemente). Die Liste `B` soll schliesslich durch einen `print`-Befehl ausgegeben werden.

Tipps:

- Verwenden Sie zwei ineinander geschachtelte Schleifen, um zu überprüfen, ob ein Element bereits in der Liste `B` enthalten ist.
- Verwenden Sie eine bool'sche Variable, um zu verfolgen, ob ein Element bereits in der Liste `B` vorhanden ist:

```
ist_vorhanden = False
for element_B in B:
    if element_B == A[i]:
        ist_vorhanden = True
if not ist_vorhanden:
    B.append(A[i])
```

Aufgabe (Challenge) 6.20

Probieren Sie weitere Methoden zum Verändern von Listen aus, indem Sie folgende Begriffe verwenden: `.extend(...)`, `.remove(...)`, `.sort()`, `.reverse()`. Eine Auflistung aller möglichen Methoden für ein Objekt der Klasse `list` finden Sie in der offiziellen Python-Dokumentation.

6.2 Wörterbücher (dictionaries)

In der Informatik werden Daten oft nicht nur als Listen, sondern auch als sogenannte **Dictionaries** (Wörterbücher) gespeichert. Ein Dictionary ist eine Sammlung von Schlüssel-Wert-Paaren. Anders als bei Listen werden die Werte nicht durch einen Index, sondern durch einen eindeutigen **Schlüssel** (key) lokalisiert. Dies ist vielfach praktischer als die Speicherung in Listen, da man direkt mit einem Begriff auf die Werte zugreifen kann, ohne die Position des Werts in der Liste kennen zu müssen.

Definition 6.3 (Dictionary)**:**
Ein **Dictionary** ist eine Datenstruktur, die jedem Schlüssel (*key*) einen Wert (*value*) zuordnet. Die Schlüssel müssen eindeutig sein und können z.B. Zahlen oder Zeichenketten sein.

Beispiel 6.10 (Dictionary für Kontaktdaten)**:**
Ein typisches Beispiel für ein Dictionary ist ein Adressbuch, in dem zu jedem Namen die Telefonnummer gespeichert ist:

```
telefonbuch = {
    "Anna": "079 123 45 67",
    "Ben": "078 987 65 43",
    "Clara": "077 555 44 33"
}
print(telefonbuch["Anna"])  # gibt "079 123 45 67" aus
```

Beispiel 6.11 (Dictionary für Produktpreise)**:**
Auch in einem Online-Shop werden Produkte oft mit ihren Preisen als Dictionary gespeichert:

```
preise = {
    "Apfel": 0.80,
    "Banane": 0.50,
    "Brot": 2.50
}
print(preise["Brot"])  # gibt 2.5 aus
```

Beispiel 6.12 (Werte hinzufügen und ändern)**:**
Sie können einem Dictionary neue Schlüssel-Wert-Paare hinzufügen oder bestehende Werte ändern:

```
preise["Milch"] = 1.60          # neues Produkt hinzufügen
preise["Apfel"] = 0.90          # Preis ändern
print(preise)
```

Beispiel 6.13 (Alle Schlüssel und Werte durchgehen)**:**
Mit einer Schleife können Sie alle Einträge eines Dictionaries durchgehen:

```
for produkt in preise:
    print(produkt, "kostet", preise[produkt], "Franken")
```

Beispiel 6.14 (Überprüfen, ob ein Schlüssel existiert)**:**
Um zu überprüfen, ob ein Schlüssel in einem Dictionary existiert, können Sie den `in`-Operator verwenden:

```
if "Brot" in preise:
    print("Brot ist im Dictionary vorhanden.")
else:
    print("Brot ist nicht im Dictionary vorhanden.")
```

Aufgabe 6.21

Erstellen Sie ein Dictionary `noten`, das die Noten von drei Schülern speichert: `"Lea"` (Note 5.5), `"Tim"` (Note 4.0), `"Sara"` (Note 6.0). Geben Sie die Note von `"Sara"` aus.

Aufgabe 6.22

Fügen Sie dem Dictionary `noten` aus der vorherigen Aufgabe einen neuen Schüler „Alex“ mit der Note 5.0 hinzu. Ändern Sie Tims Note auf 4.5 und geben Sie das gesamte Dictionary aus.

Aufgabe 6.23

Sie verwalten die Lagerbestände eines kleinen Geschäfts. Erstellen Sie ein Dictionary `lager` mit den Produkten `"Cola"` (10 Stück), `"Fanta"` (5 Stück) und `"Wasser"` (20 Stück). Schreiben Sie ein Programm, das die Anzahl der Flaschen `"Fanta"` um 2 reduziert (z.B. durch Verkauf) und das neue Dictionary ausgibt.

Aufgabe 6.24

Erstellen Sie ein Dictionary, das für verschiedene Länder die jeweilige Hauptstadt speichert:

- Für die Schweiz: `"Hier gibt es keine Hauptstadt, nur eine Bundesstadt."`
- Für Deutschland: `"Berlin"`
- Für Frankreich: `"Paris"`

Lassen Sie den Benutzer mit `input()` nach einem Land fragen und geben Sie die entsprechende Hauptstadt aus dem Dictionary mit `print` aus.

Falls das eingegebene Land nicht im Dictionary existiert, soll ausgegeben werden:

`"Land nicht gefunden."`

siehe Beispiel 6.14 für Hinweise dazu, wie dies umgesetzt werden kann.

Aufgabe 6.25

Carlas Englisch ist nicht so gut. Helfen Sie Carla, ein paar Sätze zu übersetzen. Schreiben Sie in einem Dictionary namens `deutsch_zu_englisch` die Übersetzung der folgenden Wörter in Englisch: „Die“, „Der“, „Das“, „Stuhl“, „Sofa“, „Lampe“, „ist“, „rot“, „grün“, „gelb“, „blau“, „schwarz“, „weiss“.

Implementieren Sie eine Funktion `uebersetzen(satz)`, die eine Liste erstellt, welche die englische Übersetzung jedes Wortes in der Liste `satz` enthält. Die Liste soll Wort für Wort erstellt und am Schluss mit `print` ausgegeben werden.

Vorlage:

```
deutsch_zu_englisch = {
    ...
    ...
    ...
}


def uebersetzen(satz):
    uebersetzter_satz = []
    ...
    ...
    print(uebersetzter_satz)

uebersetzen(["Die", "Lampe", "ist", "rot"])
```

Tipp: Arbeiten Sie sich schrittweise heran!

1. Schreiben Sie nun einen Code, um auf jedes Element (jedes Wort) der Liste `satz` zuzugreifen, speichern Sie das Wort in einer Variable `wort`.
2. Greifen Sie nun auf das entsprechende englische Wort im Dictionary `deutsch_zu_englisch` zu.
3. Fügen Sie das englische Wort der neuen Liste hinzu.

Die Werte eines Dictionaries können selber ebenfalls Listen, Dictionaries oder andere Python-Objekte sein, was besonders nützlich ist, wenn mehrere Werte zu einem Schlüssel gespeichert werden sollen.

Aufgabe 6.26

Erstellen Sie ein Dictionary `likes`, das für verschiedene Nutzer die Anzahl der Likes auf einem Social-Media-Post als Liste speichert. Beispiel:

```
likes = {
    "Anna": [5, 8, 12],
    "Ben": [3, 7, 9],
    "Clara": [10, 15, 20]
}
```

Schreiben Sie einen Code, die für einen eingegebenen Nutzernamen die durchschnittliche Anzahl der Likes berechnet und ausgibt.

Aufgabe 6.27

Erstellen Sie ein Dictionary `wettervorhersage`, das für verschiedene Tage die Wetterdaten als weiteres Dictionary speichert. Beispiel:

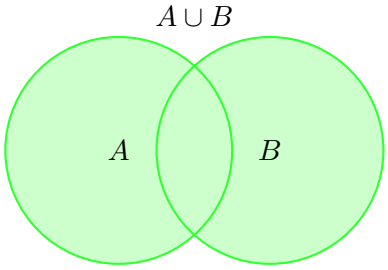
```
wettervorhersage = {
    "Montag": {"Temperatur": 18, "Regen": False},
    "Dienstag": {"Temperatur": 21, "Regen": True},
    "Mittwoch": {"Temperatur": 17, "Regen": False}
}
```

Schreiben Sie einen Code, das für einen eingegebenen Tag die Temperatur und ob es regnet ausgibt. Falls es Regnet oder unter 15 Grad ist, soll zusätzlich die Meldung „Ich gehe mit dem Bus“ ausgegeben werden, ansonsten „Ich gehe per Fahrrad“.

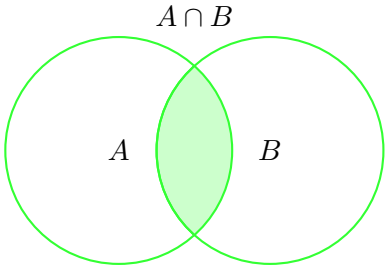
6.3 Mengen (sets)

In Python stehen die Mengenoperationen, welche Ihnen aus dem Mathematikunterricht wohlvertraut sind, zur Verfügung. Wir werden die drei mengentheoretischen Operationen *Vereinigung*, *Schnittmenge* und *Differenz* einführen und mittels ***Venndiagrammen***[1] illustrieren.

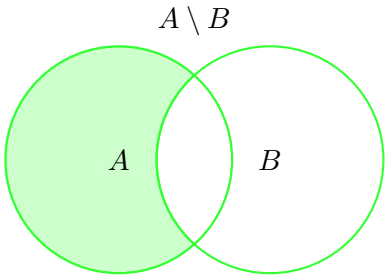
- **A** $\cup$ **B**, in Python: `A | B`
  $A \cup B := \{\ x \in M\ ;\ (x \in A) \vee (x \in B)\ \}$
  Die ***Vereinigung*** von A und B. Die Vereinigung von A und B enthält genau alle Elemente, die in A **oder** B liegen.



- **A** $\cap$ **B**, in Python: `A & B`
  $A \cap B := \{\ x \in M\ ;\ (x \in A) \wedge (x \in B)\ \}$
  Die ***Schnittmenge*** von A und B. Die Schnittmenge von A und B enthält genau alle Elemente, die in A **und** in B liegen.



- **A** $\setminus$ **B**, in Python: `A - B`
  $A \setminus B := \{\ x \in M\ ;\ (x \in A) \wedge (x \notin B)\ \}$
  Die ***Differenz*** von A und B. Die Differenz von A und B enthält genau alle Elemente, die in A aber **nicht** in B liegen.



Falls M eine endliche Menge ist (nicht unendlich viele Elemente enthält), dann bezeichnet $|M|$ die Anzahl der Elemente in M. In Python finden wir die Anzahl der Element in der Menge M mit dem Befehl `len(M)`.

Beispiel 6.15:
Sets können in Python mit geschweiften Klammern `{` und `}` oder mit dem Befehl `set()` erstellt werden. Folgender Code zeigt die drei Mengenoperationen an einem Beispiel mit Früchten.

[1] Benannt nach dem englischen Mathematiker *John Venn Junior*.

```python
# Mengen mit Früchten erstellen
set_1 = {"Apfel", "Banane", "Orange", "Traube"}

# Sets können auch aus Listen erstellt werden
liste_2 = ["Banane", "Orange", "Kiwi", "Mango"]
liste_3 = ["Traube", "Kiwi", "Melone"]
# Listen zu Sets konvertieren
set_2 = set(liste_2)
set_3 = set(liste_3)

# Schnittmenge: Finden Sie die gemeinsamen Früchte
gemeinsame_fruechte = set_1 & set_2
print("Gemeinsame Früchte zwischen set_1 und set_2:", gemeinsame_fruechte)
print("Anzahl der gemeinsamen Früchte:", len(gemeinsame_fruechte))

# Vereinigungsmenge: Kombinieren Sie alle einzigartigen Früchte
alle_fruechte = set_1 | set_2
print("Alle einzigartigen Früchte aus set_1 und set_2:", alle_fruechte)
print("Anzahl aller einzigartigen Früchte:", len(alle_fruechte))

# Differenzmenge: Finden Sie die Früchte in set_1, die nicht in set_3 sind
nur_in_set_1 = set_1 - set_3
print("Früchte in set_1, aber nicht in set_3:", nur_in_set_1)
print("Anzahl der nur in set_1 vorhandenen Früchte:", len(nur_in_set_1))
```

Programm 6.28: `example_sets_fruit.py`

Aufgabe 6.28

An einer Schule können Schüler mehrere Kurse wählen. Manche Kurse überschneiden sich, andere sind Pflicht.

```python
# Gegeben:
schueler_a = ["Mathe", "Englisch", "Informatik", "Biologie"]
schueler_b = ["Mathe", "Englisch", "Informatik", "Kunst", "Musik"]
schueler_c = ["Informatik", "Englisch", "Sport", "Geschichte"]

pflichtkurse = ["Mathe", "Englisch"]
wahlkurse    = ["Informatik", "Biologie", "Kunst", "Musik", "Sport", "
    Geschichte"]
```

1. Finden Sie alle **Kurse**, die mindestens einer der Schüler belegt.
2. Finden Sie alle **Kurse**, die von allen drei Schülern gemeinsam belegt werden.
3. Bestimmen Sie alle **Kurse**, die exklusiv nur Schüler A hat.
4. Ermitteln Sie die **Pflichtkurse**, die zwar existieren, aber von mindestens einem Schüler **nicht gewählt** wurden.
5. Stellen Sie eine Liste aller **Wahlkurse** zusammen, die **alle drei Schüler gemeinsam** gewählt haben.

Tipp: Mit dem Befehl `set(liste)` können Sie eine Liste in eine Menge umwandeln. Mit dem

Befehl `list(menge)` können Sie eine Menge wieder in eine Liste umwandeln.

```
# Beispiel-Ausgabe:
Alle belegten Kurse: {'Biologie', 'Kunst', 'Musik', 'Sport', 'Mathe', '
    Englisch', 'Geschichte', 'Informatik'}
Gemeinsame Kurse aller: {'Englisch', 'Informatik'}
Exklusiv nur Schüler A: {'Biologie'}
Pflichtkurse, die fehlen: {'Mathe'}
Gemeinsame Wahlkurse: {'Informatik'}
```

6.4 Tupel

Tupel sind eine weitere grundlegende Datenstruktur in Python, die Ähnlichkeiten mit Listen aufweisen, sich aber in einem entscheidenden Punkt unterscheiden: ihrer Unveränderlichkeit.

Definition 6.4:
Ein Tupel in Python ist eine geordnete, unveränderliche Sammlung von Elementen. Tupel sind ähnlich wie Listen, können aber nach ihrer Erstellung nicht mehr verändert werden. Man erstellt sie, indem man Elemente in runde Klammern setzt, getrennt durch Kommas.

Tupel in Python haben die folgenden Eigenschaften:

- **Unveränderlichkeit (Immutability):** Versucht man, ein Element zu ändern, erhält man eine Fehlermeldung. Zum Beispiel würde `koordinaten[0] = 5` einen Fehler verursachen.
- **Geordnetheit:** Die Reihenfolge der Elemente bleibt erhalten.
- **Heterogenität:** Ein Tupel kann verschiedene Datentypen enthalten, wie ganze Zahlen, Zeichenketten oder sogar andere Tupel. Zum Beispiel: `person = ('Anna', 30, True)`.
- **Anwendungsbereiche:** Tupel werden oft verwendet, wenn man sicherstellen will, dass Daten nicht versehentlich geändert werden, wie z. B. bei Datenbankkoordinaten, Rückgabewerten von Funktionen oder als Schlüssel in einem Wörterbuch.

Beispiel 6.16:

```
# ein Tupel (für 2D-Koordinaten) erstellen
koordinaten = (10, 20)

# das Tupel mit print ausgeben
print(koordinaten)

# das erste Element des Tupels ausgeben (Index 0)
print(koordinaten[0])

# das zweite Element des Tupels ausgeben (Index 1)
print(koordinaten[1])

# Error! Tupel sind unveränderlich!
koordinaten[0] = 5
```

6.5 Weitere Aufgaben

Aufgabe 6.29 Notenspiegel für mehrere Personen

Sie möchten ein Programm schreiben, das für mehrere Personen in Ihrer Klasse den Notenschnitt berechnet und prüft, ob jede Person ihr Ziel erreicht hat.

- **Teilaufgabe 1:** Schreiben Sie die Funktion `schnitt(punkte_dict)`.
  - Der Parameter `punkte_dict` ist ein Dictionary, in dem jeder Schlüssel ein Name ist und jeder Wert eine Liste von Noten für die jeweilige Person (oder Punkten).
  - Die Funktion berechnet für jede Person den Durchschnitt und gibt ein Dictionary zurück, in dem die Namen den berechneten Schnitten zugeordnet sind.
  - Beispiel: `schnitt({"Anna":[5.0,5.5,4.5,6.0],"Ben":[4.0,3.5,4.5]})`
    Rückgabe: `{"Anna":5.25,"Ben":4.0}`
- **Teilaufgabe 2:** Schreiben Sie die Funktion `ziel_erreicht(schnitte, zielnote)`.
  - Der Parameter `schnitte` ist ein Dictionary mit Namen und Schnittwerten (Rückgabe von `schnitt`).
  - Der Parameter `zielnote` ist eine Zahl, z. B. 4.0, die angibt, ab welchem Schnitt eine Person ihr Ziel erreicht hat.
  - Die Funktion gibt ein Dictionary zurück, das für jede Person angibt, ob das Ziel erreicht wurde (`True` oder `False`).
  - Beispiel: `ziel_erreicht({"Anna":5.25,"Ben":4.0}, 4.5)`
    Rückgabe: `{"Anna":True,"Ben":False}`

Aufgabe 6.30 Mensa-Auswahl und Budgetprüfung

Sie möchten ein Programm schreiben, das Ihnen hilft, passende Gerichte aus der Mensa auszuwählen und zu prüfen, ob diese Auswahl in Ihr Budget passt.

- **Teilaufgabe 1:** Schreiben Sie die Funktion `filter_menue(menue, max_preis, nur_vegi)`.
  - Der Parameter `menue` ist eine Liste von Dictionaries, die jeweils ein Gericht mit Name, Preis und einem Wahrheitswert `vegi` enthalten.
  - Der Parameter `max_preis` gibt den maximal erlaubten Preis pro Gericht an.
  - Der Parameter `nur_vegi` gibt an, ob nur vegetarische Gerichte erlaubt sind (mögliche Werte: `True` oder `False`).
  - Die Funktion gibt eine Liste mit den Gerichten (als Dictionaries) zurück, welche die Bedingungen erfüllen.
  - Beispiel: `filter_menue([{"name":"Pasta","preis":9.5,"vegi":True},{"name":"Schnitzel","preis":12.0,"vegi":False}], 10.0, True)`
    Rückgabe: `[{"name":"Pasta","preis":9.5,"vegi":True}]`
- **Teilaufgabe 2:** Schreiben Sie die Funktion `budget_check(gerichte, budget, tage)`.
  - Der Parameter `gerichte` ist eine Liste der gewählten Gerichte (z.B. Rückgabe von `filter_menue`).
  - Der Parameter `budget` ist das verfügbare Budget für die Woche.
  - Der Parameter `tage` ist die Anzahl der Tage, an denen in der Mensa gegessen wird.
  - Die Funktion berechnet die Gesamtkosten und prüft, ob das Budget ausreicht. Sie gibt ein Dictionary zurück mit den Schlüsseln `"summe"` und `"ok"`.
  - Beispiel: `budget_check([{"preis":9.5},{"preis":9.5},{"preis":9.5}], 30.0,`

```
5)
```
Rückgabe: `{"summe":28.5,"ok":True}`

Kapitel 7

Objektorientierte Programmierung

7.1 Klassen

Bisher haben wir mit einfachen Variablentypen (*integer*, *string*, *boolean*) und grundlegenden Datenstrukturen wie *Listen* und *Dictionaries* gearbeitet. Dabei haben wir bereits gesehen, dass viele dieser Objekte über sogenannte **Methoden** verfügen. So konnten wir beispielsweise mit der Methode `.append()` ein neues Element an eine Liste anhängen. Diese Methoden sind typisch für den jeweiligen Datentyp und ermöglichen es uns, bequem mit den enthaltenen Daten zu arbeiten.

In Python sind tatsächlich **alle Variablen Objekte**. Jedes Objekt besitzt bestimmte **Eigenschaften** (Attribute) und kann über **Methoden** manipuliert oder abgefragt werden.

Python erlaubt es uns jedoch nicht nur, bestehende Objekttypen zu verwenden, sondern auch **eigene Datentypen zu definieren**. Dies geschieht mithilfe von **Klassen**, einem zentralen Konzept des Programmierparadigmas der Objektorientierte Programmierung (OOP). Klassen dienen als Bauvorlage für komplexe Objekte, die sowohl **Daten** (Attribute) als auch **Verhalten** (Methoden) enthalten können. Auf diese Weise können wir reale oder abstrakte Dinge im Programm strukturiert und nachvollziehbar modellieren.

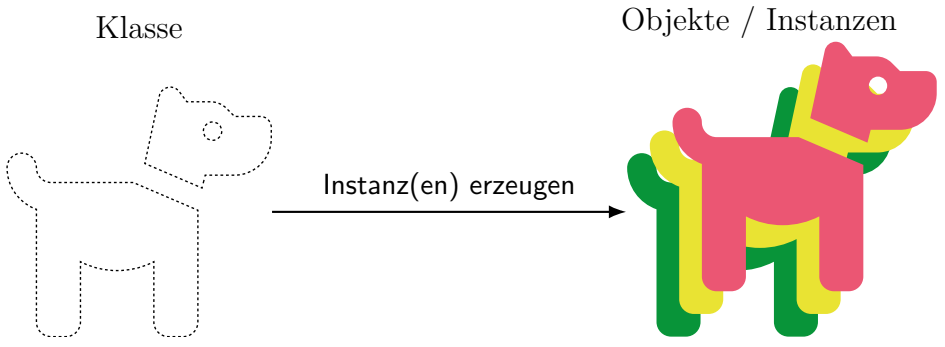
Definition 7.1:
Eine **Klasse** ist eine Vorlage oder Schablone für Objekte. Sie fasst **Daten** (Attribute) und **Funktionen** (Methoden) zusammen und beschreibt damit die **Eigenschaften** und das **Verhalten** einer ganzen Gruppe von Objekten. Ein **Objekt** ist eine konkrete Instanz (= ein tatsächlich existierendes Exemplar) einer Klasse.

Die folgenden beiden Analogien verdeutlichen die Beziehung zwischen einer Klasse und ihren Objekten:

- Eine Klasse ist wie ein **Kochrezept**. Das Rezept beschreibt, welche *Zutaten* (Attribute) benötigt werden und welche *Schritte* (Methoden) ausgeführt werden, um ein Gericht zuzubereiten. Wenn Sie das Rezept befolgen, entsteht ein *konkretes Gericht* — das ist das Objekt. Aus demselben Rezept (Klasse) können Sie beliebig viele Gerichte (Objekte) kochen, die gleich aufgebaut, aber individuell gewürzt sind (unterschiedliche Attributwerte).
- Eine Klasse kann man sich auch wie eine **Charaktervorlage** in einem Videospiel vorstellen. Die Vorlage *Krieger* oder *Magier* definiert allgemeine *Eigenschaften* (z.B. Lebenspunkte, Stärke, Intelligenz) und *Fähigkeiten* (z.B. angreifen, heilen, zaubern). Wenn Sie im Spiel einen neuen Charakter erstellen, z.B. *Irelia die Kriegerin*, dann ist das ein *Objekt*, welches auf der

Vorlage *Krieger* (der Klasse) basiert. Alle Krieger haben ähnliche Fähigkeiten, aber individuelle Werte.

Objekte werden in Games ständig verwendet, beispielsweise um Spieler, Gegner, Tiere, Wolken oder andere Elemente zu „spawnen“, also um diese mit zufälligen Variationen zu erzeugen. Die Grundidee von Klassen ist in Abbildung 7.1 dargestellt.




```
class Hund:
```

```
hund_1 = Hund("Thommy", "Grün", ...)
hund_2 = Hund("Bello", "Gelb", ...)
hund_3 = Hund("Rocky", "Rot", ...)
# ...
```

Attribute	**Methoden**	**Attribute**	**Methoden**
`name`	`gassi_gehen()`	`name: Tommy`	`gassi_gehen()`
`farbe`	`belle()`	`farbe: Gruen`	`belle()`
`augenfarbe`	`gehe_vorwaerts()`	`augenfarbe: Braun`	`gehe_vorwaert()`
`gewicht`	`ist_kastriert()`	`gewicht: 16`	`ist_kastriert()`
`laenge`	`kom_her()`	`laenge: 89`	`komm_her()`
...	...	...	...

Abbildung 7.1: Illustration von Klassen und Instanzen in Python: Klassen (links) besitzen Eigenschaften und Methoden, welche für jede Instanz dieser Klasse (rechts) definiert und aufgerufen werden können.

Im Grunde genommen kennen wir Klassen in Python bereits: So erstellen wir beispielsweise jedes Mal eine Instanz der Klasse `list`, wenn wir eine Variable des Typs `list` (eine Liste) erstellen (siehe Abbildung 7.2).

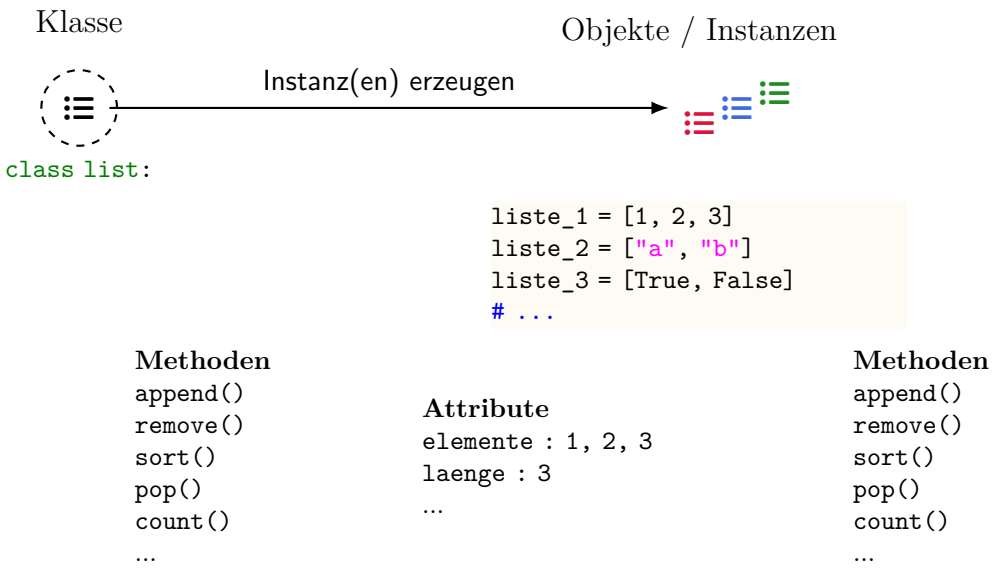


Abbildung 7.2: Illustration von Klassen und Instanzen für Listen in Python: Die Klasse `list` definiert die Struktur und Methoden, während konkrete Listen-Objekte individuelle Inhalte besitzen.

Beispiel 7.1 (Klassen definieren)**:**
Eine Klasse wird mit dem Schlüsselwort `class` definiert. Im folgenden Beispiel erstellen wir eine Klasse `Person` mit den Attributen `name` und `alter` sowie zwei Methoden `vorstellen` und `geburtstag_feiern`.

```
class Person:
    def __init__(self, name, alter):
        self.name = name
        self.alter = alter

    def vorstellen(self):
        print(f"Hallo, ich heisse {self.name} und  bin {self.alter} Jahre
   alt.")

    def geburtstag_feiern(self):
        self.alter += 1
        print(f"{self.name} ist jetzt {self.alter} Jahre alt.")


# Objekte der Klasse 'Person' erstellen
anna = Person("Anna", 16)
jan = Person("Jan", 17)

# Methoden aufrufen
anna.vorstellen()  # Ausgabe: Hallo, ich heisse Anna und bin 16 Jahre alt.
jan.vorstellen()  # Ausgabe: Hallo, ich heisse Jan und bin 17 Jahre alt.

# Geburtstag feiern
anna.geburtstag_feiern()  # Ausgabe: Anna ist jetzt 17 Jahre alt.
```

Programm 7.1: `person.py`

Beispiel 7.1 führt bereits einige zentrale Konzepte einer Klasse ein:

- **Der Konstruktor `__init__`**: Diese spezielle Methode wird **automatisch aufgerufen**, sobald ein neues Objekt erzeugt wird (z.B. bei `anna = Person("Anna", 16)`). Ihre Hauptaufgabe besteht darin, das Objekt mit Anfangswerten zu initialisieren. Die Werte in den Klammern (`"Anna"` und `16`) werden dabei als Parameter an die `__init__`-Methode übergeben.
- **Die Instanzvariable `self`**: Das Schlüsselwort `self` repräsentiert das konkrete Objekt (die Instanz), mit dem gerade gearbeitet wird. Wird beispielsweise `anna.geburtstag_feiern()` aufgerufen, verweist `self` innerhalb der Methode auf das Objekt `anna`. Dadurch wird sichergestellt, dass `self.alter += 1` das Alter von `anna` und nicht das eines anderen Objekts verändert. `self` muss immer der erste Parameter jeder Methode innerhalb einer Klasse sein.
- **Attribute definieren**: Innerhalb der Klasse werden Attribute eines Objekts mit `self.attributname = wert` erstellt. Im obigen Beispiel weist `self.name = name` dem Attribut `name` des Objekts den übergebenen Wert zu.
- **Auf Attribute zugreifen**: Um den Wert eines Attributs innerhalb einer Methode zu lesen oder zu ändern, wird ebenfalls `self.attributname` verwendet.
- **Methoden aufrufen**: Nachdem ein Objekt erstellt wurde (z.B. `anna = Person("Anna", 16)`), können seine Methoden mit der Punkt-Notation aufgerufen werden. Der Aufruf `anna.geburtstag_feiern()` führt die Methode für das Objekt `anna` aus und erhöht dessen Alter.

Beispiel 7.2 (Bankkonto-Klasse)**:**
Ein praktisches Beispiel ist eine Klasse für Bankkonten:

```
class Bankkonto:
    def __init__(self, kontonummer, inhaber, kontostand):
        self.kontonummer = kontonummer
        self.inhaber = inhaber
        self.kontostand = kontostand

    def einzahlen(self, betrag):
        if betrag > 0:
            self.kontostand += betrag
            print(f"CHF {betrag} eingezahlt. Neuer Kontostand: CHF {self.kontostand}")
        else:
            print("Fehler: Betrag muss positiv sein")

    def abheben(self, betrag):
        if betrag > 0:
            if betrag <= self.kontostand:
                self.kontostand -= betrag
                print(
                    f"CHF {betrag} abgehoben. Neuer Kontostand: CHF {self.kontostand}"
                )
            else:
                print("Fehler: Nicht genügend Guthaben")
        else:
            print("Fehler: Betrag muss positiv sein")

    def kontoinfo(self):
        print(f"Konto {self.kontonummer} ({self.inhaber}): CHF {self.kontostand}")


# Objekt erstellen
mein_konto = Bankkonto("CH123456", "Lea Müller", 1000)

# Methoden aufrufen
mein_konto.kontoinfo()
mein_konto.einzahlen(500)
mein_konto.abheben(200)
mein_konto.kontoinfo()
```

Programm 7.2: `bankkonto.py`

Aufgabe 7.1

Erstellen Sie eine Klasse `Buch` mit den Attributen `titel`, `autor` und `seitenzahl`. Implementieren Sie eine Methode `info()`, die eine Zusammenfassung der Buchinformationen ausgibt. Erstellen Sie zwei Buchobjekte und rufen Sie die `info()`-Methode auf.

Aufgabe 7.2

Erweitern Sie die `Bankkonto`-Klasse um eine Methode `ueberweisen(self, zielkonto, betrag)`. Diese Methode soll eine Geldüberweisung von einem Konto auf ein anderes ermöglichen.

Anforderungen an die Methode:

1. **Guthaben prüfen:** Stellen Sie sicher, dass das Guthaben auf dem Quellkonto (`self`) für den `betrag` ausreicht.
2. **Überweisung durchführen:** Falls das Guthaben ausreicht, ziehen Sie den Betrag vom aktuellen Konto ab und fügen Sie ihn dem `zielkonto` hinzu.
3. **Feedback geben:** Geben Sie eine Erfolgs- oder Fehlermeldung auf der Konsole aus, um den Benutzer über den Status der Überweisung zu informieren.

Tipp: Sie können die bereits existierenden Methoden `abheben()` und `einzahlen()` wiederverwenden, um Ihren Code sauber und kurz zu halten!

7.2 Vordefinierte Klassen in Python

Wie bereits erwähnt ist jegliche Variable in Python ein Objekt, das zu einer bestimmten Klasse gehört. Python bietet eine Vielzahl von vordefinierten Klassen, welche wir in den vorausgehenden Kapiteln bereits angeschaut haben, beispielsweise Datentypen wie `int`, `float`, `str` (String), `list` (Liste) oder `dict` (Dictionary), sowie weitere Klassen. Diese Klassen haben ihre eigenen Methoden, die spezifische Operationen auf den Objekten dieser Klassen ermöglichen.

Bemerkung 7.1 (Vordefinierte Klassen in Python)**:**
Eine Auflistung aller vordefinierten Klassen in Python finden Sie in der offiziellen Dokumentation unter https://docs.python.org/3/library/stdtypes.html. Dort sind alle Datentypen und deren Methoden beschrieben, die in Python verfügbar sind. Alternativ kann folgender Code ausgeführt werden, um die Namen aller vordefinierten Klassen in Python zu erhalten:

```
import builtins
import inspect

builtin_classes = [name for name, obj in vars(builtins).items() if inspect.
    isclass(obj)]
print(builtin_classes)
```

Programm 7.5: `builtin_classes.py`

Weshalb kann man in Python eine Variable für gewisse vordefinierte Klassen erstellen, ohne explizit den Namen der Klasse anzugeben? Hier handelt es sich um etwas „Python-Magie“, bei der Python den Typ der Variable automatisch erkennt und die entsprechende Klasse verwendet. Wenn

wir beispielsweise eine Variable `x = 5` erstellen, wird `x` automatisch als Objekt der Klasse `int` erstellt. Python kümmert sich im Hintergrund um die Zuweisung der richtigen Klasse, sodass wir uns nicht explizit darum kümmern müssen. Wir könnten jedoch die Klasse explizit angeben, indem wir beispielsweise `x = int(5)` schreiben, was dasselbe Ergebnis liefert.

Wie wir gesehen haben, beinhalten gewisse Klassen bereits Methoden, wie beispielsweise `.append()` für Listen oder `.keys()` für Dictionaries. Diese Methoden sind spezifisch für die jeweilige Klasse und ermöglichen es uns, auf einfache Weise mit den Objekten dieser Klassen zu interagieren. Falls wir wissen möchten, welche Methoden eine bestimmte Klasse hat, können wir die Funktion `dir()` verwenden, um eine Liste aller verfügbaren Methoden und Attribute zu erhalten. Zum Beispiel:

Beispiel 7.3 (Verfügbare Methoden einer Klasse)**:**
Folgendes Beispiel zeigt, wie wir die verfügbaren Methoden für eine Liste und einen String abfragen können. Dies kann sowohl für eine konkrete Instanz wie auch für den Klassennamen selbst erfolgen:

```
# Zeige verfügbare Methoden für eingebaute Typen mit dir()

# Für eine konkrete Instanz
x = [1, 2, 3]
print("Methoden für die Instanz x (Liste):")
print(dir(x))

# Für den Klassennamen selbst
print("Methoden für die Klasse 'list':")
print(dir(list))  # gibt dasselbe Ergebnis wie dir(x) aus

# Beispiel für einen anderen Typ (str)
s = "hallo"
print("Methoden für die Instanz s (String):")
print(dir(s))

print("Methoden für die Klasse 'str':")
print(dir(str))  # gibt dasselbe Ergebnis wie dir(s) aus
```

Programm 7.6: `verfuegbare_methoden.py`

7.3 Klassenmethoden und Attribute

Neben den Instanzattributen (die zu jedem Objekt gehören) können Klassen auch Klassenattribute haben, die für alle Instanzen gleich sind.

Beispiel 7.4 (Klassenattribute)**:**
Folgendes Beispiel zeigt, wie Klassenattribute definiert und verwendet werden können, beispielsweise um SchülerInnen mit Attributen wie Name und Klasse zu erstellen:

```
class Schueler:
    schule = "Kantonsschule im Lee"  # Klassenattribut

    def __init__(self, name, klasse):
```

```
        self.name = name  # Instanzattribut
        self.klasse = klasse  # Instanzattribut

    def info(self):
        return f"{self.name}, Klasse {self.klasse}, {self.schule}"


# Objekte erstellen
s_1 = Schueler("Lisa", "3a")
s_2 = Schueler("Tim", "4b")

print(s_1.info())  # Lisa, Klasse 3a, Kantonsschule im Lee
print(s_2.info())  # Tim, Klasse 4b, Kantonsschule im Lee

# Klassenattribut über die Klasse ändern
Schueler.schule = "KLW"

# Änderung wirkt sich auf alle Instanzen aus
print(s_1.info())  # Lisa, Klasse 3a, KLW
print(s_2.info())  # Tim, Klasse 4b, KLW
```

Programm 7.7: `klassenattribute.py`

Beispiel 7.5 (Instanzzähler)**:**
Ein häufiger Anwendungsfall für Klassenattribute ist das Zählen von Instanzen:

```
class Produkt:
    anzahl_produkte = 0  # Klassenattribut für alle Produkte

    def __init__(self, name, preis):
        self.name = name
        self.preis = preis
        Produkt.anzahl_produkte += 1  # Erhöhe bei jeder neuen Instanz

    @classmethod
    def get_anzahl_produkte(cls):
        print("Anzahl Produkte:", cls.anzahl_produkte)


# Objekte erstellen
p_1 = Produkt("Laptop", 1200)
p_2 = Produkt("Smartphone", 800)
p_3 = Produkt("Maus", 30)

Produkt.get_anzahl_produkte()  # Anzahl Produkte: 3
```

Programm 7.8: `instanzzaehler.py`

Aufgabe 7.3

Erstellen Sie eine Klasse `Auto` mit den Instanzattributen `marke`, `modell` und `baujahr`. Fügen Sie ein Klassenattribut `anzahl_autos` hinzu, das die Gesamtzahl der erstellten Auto-Objekte zählt. Implementieren Sie eine Klassenmethode `get_statistik()`, die die Anzahl der Autos zurückgibt.

7.4 Vererbung und Polymorphismus

7.4.1 Vererbung

Ein mächtiges Konzept der Objektorientierung ist die Vererbung. Sie ermöglicht es, eine neue Klasse basierend auf einer vorhandenen Klasse zu erstellen und deren Eigenschaften und Methoden zu übernehmen.

Definition 7.2:
Bei der **Vererbung** erbt eine Kindklasse (Subklasse) Attribute und Methoden von einer Elternklasse (Superklasse). Die Kindklasse kann zusätzliche Attribute und Methoden haben oder vorhandene überschreiben. Die Vererbung geschieht mithilfe des Schlüsselworts `class Kindklasse(Elternklasse):` sowie der Verwendung von `super()`, um auf die Elternklasse zuzugreifen. Dabei wird zuerst eine Instanz der Elternklasse erstellt, bevor die Kindklasse ihre eigenen Attribute und Methoden hinzufügt oder überschreibt.

Beispiel 7.6 (Vererbung)**:**

```
    # Elternklasse
class Fahrzeug:
    def __init__(self, marke, modell, baujahr):
        self.marke = marke
        self.modell = modell
        self.baujahr = baujahr
        self.km_stand = 0

    def fahren(self, strecke):
        self.km_stand += strecke
        print(f"Fahre {strecke} km. Neuer Kilometerstand: {self.km_stand} km")

    def info(self):
        return f"{self.marke} {self.modell} ({self.baujahr}), {self.km_stand} km"


# Kindklasse
class ElektroAuto(Fahrzeug):
    def __init__(self, marke, modell, baujahr, batterie_kapazitaet):
        super().__init__(marke, modell, baujahr)  # Elternklassen-Konstruktor aufrufen
        self.batterie_kapazitaet = batterie_kapazitaet
        self.ladezustand = 100  # Prozent

    def laden(self):
        self.ladezustand = 100
        print(f"{self.marke} {self.modell} wurde vollständig geladen.")

    def fahren(self, strecke):
        verbrauch = strecke * 0.2  # 20% Verbrauch pro 100 km
        if self.ladezustand - verbrauch >= 0:
            self.km_stand += strecke
            self.ladezustand -= verbrauch
            print(
                f"Fahre {strecke} km elektrisch. Ladezustand: {self.ladezustand:.1f}%"
```

```
            )
        else:
            print("Nicht genug Batterieladung für diese Strecke!")

    def info(self):
        basis_info = super().info()  # Methode der Elternklasse aufrufen
        return f"{basis_info}, Batterie: {self.batterie_kapazitaet} kWh, Ladung: {self.ladezustand:.1f
    }%"


# Objekte erstellen
normales_auto = Fahrzeug("VW", "Golf", 2020)
elektro_auto = ElektroAuto("Tesla", "Model 3", 2021, 75)

# Methoden testen
normales_auto.fahren(100)
print(normales_auto.info())

elektro_auto.fahren(200)
print(elektro_auto.info())
elektro_auto.laden()
print(elektro_auto.info())
```

Programm 7.10: `elektroauto.py`

In diesem Beispiel:

- `ElektroAuto` erbt von `Fahrzeug` und erhält alle seine Attribute und Methoden.
- Mit `super().__init__(...)` rufen wir den Konstruktor der Elternklasse auf.
- Die Methode `fahren()` wird in der Kindklasse überschrieben, um die spezifische Funktionalität eines Elektroautos zu implementieren.
- Mit `super().info()` greifen wir auf die Methode der Elternklasse zu.

Aufgabe 7.4

Erstellen Sie eine Elternklasse `Person` mit den Attributen `name` und `alter` sowie einer Methode `geburtstag_feiern()`. Erstellen Sie dann eine Kindklasse `Schueler` mit einem zusätzlichen Attribut `schulklasse` und einer überschriebenen Methode `vorstellen()`, die auch die besuchte Schulklasse ausgibt.Implementieren Sie eine zusätzliche Methode, welches es erlaubt, dem Schüler eine neue Schulklasse zuzuweisen.

Aufgabe 7.5

Erstellen Sie eine Basisklasse `Bankkonto` wie im vorherigen Beispiel. Dann erstellen Sie eine Unterklasse `Sparkonto`, die eine zusätzliche Methode `zinsen_gutschreiben(zinssatz)` hat, die Zinsen basierend auf dem aktuellen Kontostand gutschreibt.

7.4.2 Polymorphismus

Unterschiedliche Klassen können Methoden mit denselben Namen haben, aber jede Klasse hat ihre eigene Implementierung. Dies wird als **Polymorphismus** bezeichnet. Das Wort Polymorphismus stammt aus dem Griechischen und bedeutet soviel wie "viele Formen", von griechisch *poly* = viel und *morphos* = Form. Polymorphismus ermöglicht, dass verschiedene Klassen auf die gleiche Weise angesprochen werden können, obwohl sich die Methoden unterscheiden. Dies ist insbesondere bei Kindesklassen von Bedeutung, die die Methoden der Elternklasse überschreiben oder auf unterschiedliche Weise implementieren können.

Beispiel 7.7 (Polymorphismus)**:**
Folgendes Beispiel zeigt, wie verschiedene Klassen die gleiche Methode `geraeusch()` implementieren können:

```
class Tier:
    def geraeusch(self):
        return "Ein Tier macht ein Geräusch"


class Hund(Tier):
    def geraeusch(self):
        return "Wuff"


class Katze(Tier):
    def geraeusch(self):
        return "Miau"


tiere = [Tier(), Hund(), Katze()]
for tier in tiere:
    print(tier.geraeusch())

# Ausgabe:
# Ein Tier macht ein Geräusch
# Wuff
# Miau
```

Programm 7.13: `polymorphism.py`

7.5 Praktisches Beispiel: Bibliothekssystem

Als umfassenderes Beispiel implementieren wir ein einfaches Bibliothekssystem mit mehreren Klassen, die verschiedene Aspekte einer Bibliothek modellieren.

Beispiel 7.8 (Bibliothekssystem)**:**
Folgendes Beispiel zeigt, wie wir Klassen für Bücher, Autoren usw. für eine eine Bibliothek erstellen können:

```
class Buch:
    def __init__(self, titel, autor, isbn):
        self.titel = titel
        self.autor = autor
        self.isbn = isbn
        self.ausgeliehen = False

    def info(self):
        status = "ausgeliehen" if self.ausgeliehen else "verfügbar"
        return f'"{self.titel}" von {self.autor} (ISBN: {self.isbn}) - {status}'


class Bibliotheksmitglied:
    def __init__(self, name, mitgliedsnummer):
        self.name = name
        self.mitgliedsnummer = mitgliedsnummer
```

```
        self.ausgeliehene_buecher = []

    def buch_ausleihen(self, buch):
        if not buch.ausgeliehen:
            self.ausgeliehene_buecher.append(buch)
            buch.ausgeliehen = True
            print(f"{self.name} hat '{buch.titel}' ausgeliehen.")
            return True
        else:
            print(f"Fehler: '{buch.titel}' ist bereits ausgeliehen.")
            return False

    def buch_zurueckgeben(self, buch):
        if buch in self.ausgeliehene_buecher:
            self.ausgeliehene_buecher.remove(buch)
            buch.ausgeliehen = False
            print(f"{self.name} hat '{buch.titel}' zurückgegeben.")
            return True
        else:
            print(f"Fehler: {self.name} hat '{buch.titel}' nicht ausgeliehen.")
            return False

    def info(self):
        anzahl = len(self.ausgeliehene_buecher)
        info = f"{self.name} (Nr. {self.mitgliedsnummer}) - {anzahl} Bücher ausgeliehen"
        if anzahl > 0:
            info += ":\n"
            for buch in self.ausgeliehene_buecher:
                info += f"- {buch.titel}\n"
        return info


class Bibliothek:
    def __init__(self, name):
        self.name = name
        self.buecher = []
        self.mitglieder = []

    def buch_hinzufuegen(self, buch):
        self.buecher.append(buch)
        print(f"Buch '{buch.titel}' wurde zur Bibliothek hinzugefügt.")

    def mitglied_registrieren(self, mitglied):
        self.mitglieder.append(mitglied)
        print(f"{mitglied.name} wurde als Mitglied registriert.")

    def buch_suchen(self, suchbegriff):
        ergebnisse = []
        for buch in self.buecher:
            if (
                suchbegriff.lower() in buch.titel.lower()
                or suchbegriff.lower() in buch.autor.lower()
                or suchbegriff in buch.isbn
            ):
                ergebnisse.append(buch)
        return ergebnisse

    def verfuegbare_buecher(self):
        return [buch for buch in self.buecher if not buch.ausgeliehen]

    def statistik(self):
        verfuegbar = len(self.verfuegbare_buecher())
        ausgeliehen = len(self.buecher) - verfuegbar
        return (
            f"Bibliothek {self.name}:\n"
            f"- Gesamtzahl Bücher: {len(self.buecher)}\n"
            f"- Verfügbare Bücher: {verfuegbar}\n"
            f"- Ausgeliehene Bücher: {ausgeliehen}\n"
            f"- Anzahl Mitglieder: {len(self.mitglieder)}"
        )
```

```python
# Beispielverwendung
bibliothek = Bibliothek("Stadtbibliothek Winterthur")

# Bücher erstellen und hinzufügen
buch_1 = Buch("Harry Potter und der Stein der Weisen", "J.K. Rowling", "9783551557414")
buch_2 = Buch("Der Herr der Ringe", "J.R.R. Tolkien", "9783608939842")
buch_3 = Buch("Die unendliche Geschichte", "Michael Ende", "9783522202664")

bibliothek.buch_hinzufuegen(buch_1)
bibliothek.buch_hinzufuegen(buch_2)
bibliothek.buch_hinzufuegen(buch_3)

# Mitglieder erstellen und registrieren
mitglied_1 = Bibliotheksmitglied("Lisa Müller", "M001")
mitglied_2 = Bibliotheksmitglied("Tom Schneider", "M002")

bibliothek.mitglied_registrieren(mitglied_1)
bibliothek.mitglied_registrieren(mitglied_2)

# Bücher ausleihen
mitglied_1.buch_ausleihen(buch_1)
mitglied_2.buch_ausleihen(buch_3)

# Suche durchführen
print("\nSuchergebnisse für 'Harry':")
ergebnisse = bibliothek.buch_suchen("Harry")
for buch in ergebnisse:
    print(buch.info())

# Statistik anzeigen
print("\n" + bibliothek.statistik())

# Infos zu Mitgliedern anzeigen
print("\nMitgliederinformationen:")
print(mitglied_1.info())
print(mitglied_2.info())

# Buch zurückgeben
mitglied_1.buch_zurueckgeben(buch_1)

# Aktualisierte Statistik
print("\n" + bibliothek.statistik())
```

Programm 7.14: `library_system.py`

Aufgabe 7.6

Erweitern Sie das Bibliothekssystem um eine neue Klasse `Zeitschrift`, die von `Buch` erbt, aber zusätzlich eine `ausgabe` (z.B. „Mai 2024“) hat. Überschreiben Sie die `info()`-Methode entsprechend, und fügen Sie mindestens eine Zeitschrift zur Bibliothek hinzu.

Aufgabe (Challenge) 7.7

Entwickeln Sie ein vollständiges Lagerverwaltungssystem mit folgenden Klassen:

- `Produkt` (Basisklasse mit Name, Artikelnummer, Preis)
- `Elektronikprodukt` (erbt von Produkt, zusätzlich mit Garantiedauer)
- `Lebensmittel` (erbt von Produkt, zusätzlich mit Haltbarkeitsdatum)
- `Lager` (verwaltet Produkte, mit Methoden zum Hinzufügen, Entfernen, Suchen und Bestandsanzeige)
- `Bestellung` (enthält Produkte und Mengen, berechnet Gesamtpreis)

Implementieren Sie auch eine Methode, die abgelaufene Lebensmittel identifiziert und aus dem Lager entfernt.

7.6 Zusammenfassung

Die objektorientierte Programmierung mit Klassen ist ein mächtiges Werkzeug zur Strukturierung von Code und Modellierung realer Objekte. Wichtige Konzepte sind:

- Klassen definieren Vorlagen für Objekte mit Attributen und Methoden
- Objekte sind konkrete Instanzen von Klassen
- Methoden ermöglichen es, auf Attribute zuzugreifen und diese zu verändern.
- Vererbung ermöglicht die Wiederverwendung und Erweiterung von Code in Form von Kindklassen, die ihre Attribute von Elternklassen erben und gegebenenfalls überschreiben.
- Polymorphismus ermöglicht es, Methoden mit denselben Namen in verschiedenen Klassen zu verwenden, wobei jede Klasse ihre eigene, unabhängige Implementierung hat.

Durch die Verwendung von Klassen können komplexe Systeme modelliert und implementiert werden, wodurch der Code besser strukturiert, lesbarer und wartbarer wird. Insbesondere in Games sind Klassen ein zentrales Konzept, um verschiedene Spielobjekte (wie Spieler, Gegner, Items) zu modellieren und deren Verhalten zu steuern. In der Praxis werden Klassen häufig in Kombination mit anderen Konzepten wie Vererbung und Polymorphismus verwendet, um flexible und erweiterbare Softwarearchitekturen zu schaffen.

Kapitel 8

Praktische Anwendungen

8.1 Kalorienverbrauch

Folgende Beispiele sollen die meisten der bisher gelernten Programmier-Konzepte konkret veranschaulichen. Stellen Sie sich vor, Sie sind SportlerIn und möchten Ihren Kalorienbedarf berechnen, um sich für auf einen kommenden Wettkampf optimal zu ernähren. Hierzu wollen Sie zunächst Ihren theoretischen täglichen Kalorienbedarf berechnen. Der theoretische tägliche Kalorienbedarf berechnet sich wie folgt:

1. Ihr **Grundumsatz** (englisch Base Metabolic Rate (BMR)): Dies ist ihr Grundbedarf, also die Anzahl Kilokalorien, welche Sie theoretisch benötigen, falls Sie sich gar nicht bewegen.
2. Ihr **Leistungsumsatz**, d.h., zusätzliche Energie, die Sie bei Aktivitäten wie Spazieren, Radfahren, Joggen usw. verbrennen.

Beide Komponenten sind schwierig abzuschätzen. Sie möchten jedoch eine erste Einschätzung Ihrer BMR haben, indem sie einige bekannte Formeln auf sich selber anwenden und diese vergleichen.

Aufgabe 8.1

Schreiben Sie eine Python-Funktion, um den Grundumsatz anhand der **Harris-Benedict-Formel** zu berechnen und geben Sie das Resultat per `return` zurück.

1. Für Männer:

$$\begin{aligned}\text{BMR} = & 88.362 + \\ & (13.397 \times \text{Gewicht in kg}) + \\ & (4.799 \times \text{Körpergrösse in cm}) - \\ & (5.677 \times \text{Alter in Jahren})\end{aligned}$$

2. Für Frauen:

$$\begin{aligned}\text{BMR} = & 447.593 + \\ & (9.247 \times \text{Gewicht in kg}) + \\ & (3.098 \times \text{Körpergrösse in cm}) - \\ & (4.330 \times \text{Alter in Jahren})\end{aligned}$$

Wir haben nun eine erste, grobe Einschätzung des BMR. Allerdings gibt es auch noch weitere

mögliche Formeln, mithilfe denen der BMR berechnet werden kann. Diese möchten wir nun ebenfalls berechnen, um die unterschiedlichen Schätzungen des BMR anhand der verschiedenen Methoden zu vergleichen.

Aufgabe 8.2

Schreiben Sie zwei weitere Funktionen, um den BMR anhand der **Mifflin-St-Jeor-Gleichung** sowie der **Katch-McArdle-Formel** zurückzugeben und berechnen Sie den Durchschnitt aller drei Formeln in einer weiteren Funktion.

1. **Mifflin-St-Jeor-Gleichung**:
   (a) Männer:

$$\begin{aligned}\text{BMR} = {} & 10 \times \text{Gewicht (kg)} \\ & + 6.25 \times \text{Grösse (cm)} \\ & - 5 \times \text{Alter (Jahre)} \\ & + 5\end{aligned}$$

   (b) Frauen:

$$\begin{aligned}\text{BMR} = {} & 10 \times \text{Gewicht (kg)} \\ & + 6.25 \times \text{Grösse (cm)} \\ & - 5 \times \text{Alter (Jahre)} \\ & - 161\end{aligned}$$

2. **Katch-McArdle-Formel** (gleich für Männer sowie Frauen):

$$\begin{aligned}\text{BMR} = {} & 370 \\ & + (21.6 \times \text{FFM in kg})\end{aligned}$$

   Wobei die fettfreie Masse (FFM) zuerst berechnet werden muss:

$$\begin{aligned}\text{FFM} = {} & \text{Körpergewicht (kg)} \\ & \times (1 - \text{Körperfettanteil})\end{aligned}$$

Nun haben wir eine etwas verlässlichere Einschätzung unseres Grundumsatzes. Zum Grundumsatz können wir nun auch noch den Leistungsumsatz hinzufügen, um den gesamten theoretischen täglichen Energieumsatz in Kilokalorien zu berechnen.

Aufgabe 8.3

Berechnen Sie Ihren Leistungsumsatz (durch Leistung benötigte Energie) anhand der Tabelle 8.1.

Gewicht (kg)	60	65	70	75	80	85	90	95	100	105
Spazieren	4.5	5.1	5.4	5.7	6.0	6.4	6.7	7.3	8.0	9.0
Schnelles Gehen	5.5	6.3	7.0	7.5	8.0	8.6	9.5	10.0	10.6	11.3
Fahrradfahren	6.5	7.5	8.0	9.0	9.5	10.3	11.0	11.7	12.5	13.6
Schwimmen	8.0	9.2	10.0	10.8	11.5	12.5	13.7	14.4	15.4	16.5
Rudern	11.0	13.0	14.3	15.0	16.5	17.5	19.0	20.5	21.8	23.5
Jogging	13.0	14.5	16.0	17.5	19.0	20.0	22.0	23.5	24.5	26.5
Laufen	16.0	18.5	20.0	22.0	23.5	25.0	27.5	29.0	31.0	33.5
Sprint	19.0	21.5	24.0	26.0	28.0	30.0	32.5	34.7	35.6	39.0

Tabelle 8.1: Kalorienverbrauch für unterschiedliche Aktivitäten, pro Minute, in Abhängigkeit des Körpergewichts

Wenn z.B. eine 80 kg schwere Person 1 Stunde lang rudert und 20 Minuten spaziert wären die Listen wie folgt:

```
l_zeit = [60, 20] # Zeit in Minuten
l_kcal = [16.5, 6] # Kalorien pro Aktivität
```

Erstellen Sie zwei Listen:

1. Eine Liste für die Zeit, während der Sie täglich einer Aktivität nachgehen
2. Eine Liste mit dem Kalorienverbrauch pro Minute für jede dieser Aktivitäten

Schreiben Sie eine Python-Funktion `berechne_leistungsumsatz`, um Ihren Leistungsumsatz zu berechnen. Das erwartete Resultat für das Beispiel hier wäre: `1110` (= Summe von `[990, 120]`). Das Resultat soll an das Hauptprogramm zurückgegeben und in einer Variable `kalorienverbrauch` gespeichert werden.

Aufgabe 8.4

Schreiben Sie nun eine weitere Funktion, die Ihren gesamten Energieumsatz berechnet, als Summe der folgender Teile:

- **BMR**, den Sie in Aufgabe 8.2 berechnet haben (Durchschnitt der drei Formeln)
- **Leistungsumsatz**, den Sie in Aufgabe 8.3 berechnet haben

Diese Funktion ist extrem kurz und umfässt nur eine einzige (neue) Zeile.

Achtung

Die Zahlen zu Energiebedarf und Kalorien, die Sie im Rahmen dieser Übungen berechnet haben, sollten Sie mit Vorsicht geniessen, aus folgenden Gründen:

- Ihr realer Energiebedarf kann signifikant von Ihrem berechneten Energiebedarf abweichen. Viele weitere Faktoren, die nicht in den Formeln enthalten sind, können diesen beeinflussen, beispielsweise Ihre Köpertemperatur, Ihre Fitness, Ihre Muskelmasse, sowie Ihre Non-Exercise Activity Thermogenesis (NEAT), wobei Letzteres Energie bezeichnet, die Sie durch Bewegungen, die nicht Sport sind, verbrauchen (beispielsweise durch „Zappeln“ aber auch kurze Spaziergänge und alltägliche Aktivitäten).
- Sie sollten sich bezüglich konsumierter Kalorien vor Augen halten, dass „zu viele“ konsumierte Kalorien nicht automatisch mit einer Gewichtszunahme verbunden sind. In einem Kilo Körperfett stecken ca. 8000 Kilokalorien, wobei der Köper über verschiedene Mechanismen verfügt, um ein stabiles Körpergewicht zu halten, etwa die Erhöhung oder Senkung der Körpertemperatur im Falle eines kurzfristigen Kalorien-Überschusses, respektive -Defizits. Allerdings kann es interessant sein, zu wissen, dass einige, nicht sehr sättigende Lebensmittel besonders viele Kalorien enthalten —- wie beispielsweise hochverarbeitete Snacks oder Süssgetränke.

8.2 Bilder Bearbeiten (Anwendung von Listen und Schleifen)

In diesem Abschnitt werden wir uns mit der Bearbeitung von Bildern in Python beschäftigen. Sie lernen, wie Sie auf einzelne Bildpunkte (Pixel) zugreifen und deren Farbwerte gezielt verändern können. Schritt für Schritt werden Sie verschiedene Effekte wie Invertierungen, Filter oder das Erzeugen von Mustern für Schwarz-Weiss-, Graustufen- und Farbbilder programmieren. Diese Übungen bieten eine praktische Anwendung für viele der bisher gelernten Konzepte wie Schleifen und Listen in einem visuell ansprechenden Kontext.

8.2.1 Vorbereitung

1. Bitte Installieren Sie das Python-Paket `Pillow` wie folgt:

   ```
   für MacOS (Bash):
   python3 -m pip install --upgrade pip
   python3 -m pip install pillow

   für Windows (PowerShell):
   py -m pip install --upgrade pip
   py -m pip install pillow
   ```

2. Erstellen Sie auf Ihrem eigenen Rechner einen neuen Ordner 📁 mit dem Namen `Bilder_Bearbeiten`.
3. Laden Sie die Datei `dateien.zip` von Moodle herunter, und zwar so, dass die Datei `dateien.zip` in dem neu angelegten Ordner `Bilder_Bearbeiten` abgespeichert ist.
4. Die Datei `dateien.zip` ist komprimiert und muss zuerst entpackt werden. Entpacken geht ganz einfach so:
   Windows: Rechtsklick auf `dateien.zip` und dann `Alle extrahieren`.
   macOS: Doppelklick auf `datein.zip`
5. Öffnen Sie das Python-File `bilder_template.py` in `VS Code`.
6. In diesem Python-File `bilder_template.py` ist eine Vorlage (Template) zu allen nachfolgenden Aufgaben bereits gegeben. Bitte lösen Sie die Aufgaben, indem Sie diese Vorlage Schritt für Schritt durch Ihren Python-Code ergänzen.

8.2.2 Aufgaben zur Bearbeitung von Bildern

8.2.2.1 Schwarz-Weiss-Bilder

Aufgabe 8.5 Anzahl der schwarzen Pixel in einem gegeben Bild zählen

Schreiben Sie eine Python-Funktion `count_black_pixels(filename)`, welche das ein Schwarz-Weiss-Bild mit dem Dateinamen `filename` als Argument erhält und die Anzahl der schwarzen Pixel in diesem Bild zählt und diese Anzahl mit `return` zurück gibt. Betrachten Sie dazu unbedingt die Vorlage.

Aufgabe 8.6 Schwarz-Weiss-Bild invertieren

Schreiben Sie eine Python-Funktion `invert_black_white_image(filename)`, welche ein Schwarz-Weiss-Bild mit dem Dateinamen `filename` als Argument erhält und dieses Bild invertiert: Alle weissen Pixel (Pixel mit Wert 1) sollen schwarz werden und alle schwarzen Pixel (Pixel mit Wert 0) sollen weiss werden.

Stellen Sie schliesslich das originale Bild und das invertierte Bild nebeneinander dar.

Aufgabe 8.7 Bild von zufällig gewählten (schwarz-weiss) Pixeln

Schreiben Sie eine Python-Funktion `random_black_white_image(width, height)`, welche ein Schwarz-Weiss-Bild mit den gegebenen Dimensionen (Breite / Höhe) erzeugt. Jeder Pixel soll dabei zufällig entweder schwarz (0) oder weiss (1) gewählt werden. Der Aufruf

```
random_black_white_image(400, 600)
```

sollte dann beispielsweise ein Schwarz-Weiss-Bild der Breite 400 Pixel und Höhe 600 Pixel generieren, wobei jeder Pixelwert (0 oder 1) zufällig gewählt wurde.

8.2.2.2 Graustufenbilder

Aufgabe 8.8 Graustufen invertieren

Schreiben Sie eine Python-Funktion `invert_grayscale_image(filename)`, welche ein Graustufenbild (256 Stufen) mit dem Dateinamen `filename` als Argument erhält und dieses Bild invertiert: Aus dem hellen (weissen) Pixel mit Wert 255 soll der dunkle (schwarze) Pixel mit Wert 0 werden, der dunkle Pixel mit Werte 1 soll zum hellen Pixel mit Wert 254 werden und so weiter.

Stellen Sie das originale Bild und das invertierte Bild nebeneinander dar.

Aufgabe 8.9 Nur 8 verschiedene Graustufen

Schreiben Sie eine Python-Funktion `only_8_shades_of_gray(filename)`, welche ein Graustufenbild (256 Stufen) mit dem Dateinamen `filename` als Argument erhält und mit nur 8 (anstelle von 256) verschiedenen Graustufen darstellt: Die Pixel mit Werten in $\{0, 1, 2, \ldots, 31\}$ sollen alle durch den Graustufenwert 0 (schwarz) dargestellt werden, die Pixel mit Werten in $\{32, 33, 34, \ldots, 63\}$ durch den Wert 32 und so weiter. Es ergeben sich also 8 Gruppen (Mengen), welche je 32 verschiedene Graustufenwerte durch denselben Graustufenwert darstellen.

Stellen Sie das originale Bild und das transformierte Bild nebeneinander dar.

Aufgabe 8.10 Vertikale Streifen

Schreiben Sie ein Python-Funktion `vertical_stripes(filename)`, welche ein Graustufenbild mit dem Dateinamen `filename` als Argument erhält. Die Funktion soll „vertikale Streifen“ in gleichmässigen Abständen durch das Bild legen.

8.2.2.3 RGB-Bilder

Aufgabe 8.11 Grünanteil erhöhen

Schreiben Sie ein Python-Funktion `increase_green(filename)`, welche ein RGB-Bild mit dem Dateinamen `filename` als Argument erhält und den Grünanteil dieses RGB-Bilds um 20 Prozent verstärkt / erhöht. Es ist kein Problem, wenn ein Wert bei der Erhöhung den Wert 255 übersteigt: Die Image-Library interpretiert jeden Wert ≥ 255 als 255.

Stellen Sie das originale Bild und das transformierte Bild nebeneinander dar.

(Bei Bildern mit Grünanteil von > 212 wird die Erhöhung natürlich de facto weniger als 20 Prozent betragen.)

Aufgabe 8.12 Sepia-Filter

In dieser Aufgabe wollen wir einen Sepia-Filter (Sepia-Effekt) erstellen. Wie dieser Filter definiert ist, können Sie unter https://de.wikipedia.org/wiki/Sepia_(Fotografie) nachlesen. Der Sepia-Filter findet häufige Anwendung in den sozialen Medien (e.g. Instagram).

Schreiben Sie ein Python-Funktion `sepia_filter(filename)`, welche ein RGB-Bild mit dem Dateinamen `filename` als Argument erhält und den Sepia-Effekt auf das Bild anwendet.

Stellen Sie das originale Bild und das transformierte Bild nebeneinander dar.

8.2.2.4 Weitere Aufgaben

Aufgabe 8.13 Primzahlen als Pixel

Schreiben Sie eine Python-Funktion `is_prime(number)`, welche für eine gegebene natürliche Zahl `number` entscheidet, ob `number` eine Primzahl ist (`return True`) oder nicht (`return False`).

```
Testfälle:

is_prime(0)  # return False
is_prime(23) # return True
is_prime(97) # return True
is_prime(91) # return False
```

Erstellen Sie nun eine leere Liste. Diese Liste soll am Ende ein Schwarz-Weiss-Bild von 200×200 Pixeln kodieren und somit eine Länge von 40000 haben. Der k-te Eintrag der Liste für

$$k \in \{0, 1, 2, \ldots, 39999\}$$

soll 0 (schwarz) sein, falls k eine Primzahl ist und 1 (weiss) sonst.

Aufgabe 8.14 Eigene Aufgaben

Erstellen Sie eigene interessante Aufgaben. Falls Sie eine besonders kreative Aufgabe entwickelt haben, senden Sie mir diese bitte per Mail.

Kapitel 9

Game

9.1 Einführung in Pygame

In diesem Kapitel entwickeln wir Schritt für Schritt ein eigenes 2D-Game mit `pygame-ce`. `pygame-ce` ist eine moderne, Community-gepflegte Variante von Pygame und wird genau gleich importiert mit `import pygame as pg`. Sie eignet sich hervorragend, um in Python Grafiken, Animationen, Sound und Interaktionen umzusetzen.

Essentielle Bausteine eines Games sind:

- Fenster (Auflösung, Titel) und Zeichenfläche („screen“)
- Game-Schleife mit **Ereignissen** (Tastatur, Maus) und **Zeitsteuerung** (Frames Per Second (FPS) / „refresh rate“)
- Zeichnen: Hintergrund, Farben, geometrische Figuren, Bilder („Sprites“)
- **Zustände** und **Objekte** (z.B. Spieler als `Rect`), **Bewegungen** und **Kollisionen**
- **Medien**: Bilder, Icon, Schrift, Sound / Musik

Definition 9.1 (Game-Schleife)**:**
Ein Game läuft in einer Endlosschleife, bis das Programm beendet wird. In jeder Runde werden der Eingabestatus gelesen („Events“), der interne Zustand aktualisiert („Update“) und die Szene gezeichnet („Render“). Eine **Clock** (= Uhr) begrenzt die Bildwiederholrate (FPS), sodass das Game stabil und gleichmässig läuft.

Um ***pygame-ce*** in VS Code zu installieren, gehen Sie wie folgt vor:

1. Öffnen Sie ein Terminal-Fenster in VS Code („Terminal“ → „New Terminal“).
2. Installieren Sie pygame-ce mit pip:
   `pip install pygame-ce`
3. Überprüfen Sie die Installation, indem Sie eine neue Python-Datei `game_test.py` mit folgendem Code ausführen:

   ```
   import pygame as pg
   print(pg.ver)
   ```

   Dieser Code sollte, sofern `pygame-ce` korrekt installiert ist, die Versionsnummer von `pygame-ce` ausgeben. Damit ist `pygame-ce` einsatzbereit und Sie können mit den unten stehenden Übungen starten.

Achtung

Wichtiger Hinweis 9.1:
Falls die Installation nicht funktioniert, erstellen Sie zuerst ein virtuelles Umfeld (venv) in VS Code und installieren Sie `pygame-ce` darin:

- Öffnen Sie ein Terminal-Fenster in VS Code („Terminal“ → „New Terminal“).
- Erstellen Sie ein virtuelles Environment:
  `python3 -m venv venv`
  Aktivieren Sie das venv:
  Windows: `venv\Scripts\activate`
  macOS: `source venv/bin/activate`
- Führen Sie nun nochmals die obigen Schritte zur Installation von `pygame-ce` durch.

Beispiel 9.1 (Minimale Game-Schleife)**:**
Folgender Code zeigt den minimalen Aufbau eines Pygame-Programms mit Fenster, Game-Schleife und Ereignisverarbeitung. Kopieren Sie den Code in eine Datei `game_intro.py` und führen Sie ihn in VS Code aus.

```
import pygame as pg

pg.init()  # Pygame initialisieren (starten)
HEIGHT = 600  # Höhe des Fensters
WIDTH = 800  # Breite des Fensters
WINDOW = (WIDTH, HEIGHT)  # Fenstergrösse (als Tuple gespeichert)
screen = pg.display.set_mode(WINDOW)  # Fenster erstellen
pg.display.set_caption("Mein erstes Game")  # Fenstertitel setzen
clock = pg.time.Clock()  # Clock für Zeitsteuerung erstellen

running = True  # Hauptschleife
while running:
    # --- Events ---
    for event in pg.event.get():
        if event.type == pg.QUIT:
            running = False

    # --- Update --- (Spielzustand aktualisieren)

    # --- render (zeichnen) ---
    screen.fill((30, 30, 40))
    pg.display.flip()

    # --- Zeitsteuerung ---
    clock.tick(60)  # 60 FPS

pg.quit()
```

Ein schwarzes Fenster sollte erscheinen, das Sie mit dem Schliessen-Knopf beenden können.

Aufgabe 9.1 Fenster-Titel setzen

Setzen Sie einen passenden Fenstertitel, indem Sie folgende Zeile abändern:

```
pg.display.set_caption("IHR TITEL HIER...")
```

Aufgabe (Challenge) 9.2

Laden Sie ein Fenster-Icon (`.png` oder `.jpg`): Laden Sie ein beliebiges Bild aus dem Internet herunter, speichern Sie es unter den Downloads und ziehen Sie es in Ihren Projektordner in VS Code (dort wo auch Ihre Python-Datei ist). Fügen Sie nun diese beiden Zeilen nach `pg.display.set_mode(...)` ein:

```
icon = pg.image.load("path/to/icon.png")
pg.display.set_icon(icon)
```

Wenn Ihr Ordner beispielsweise `Informatik/` heisst und das Bild unter

`Informatik/Game/icon.png`

gespeichert ist, dann verwenden Sie

`pg.image.load("Game/icon.png")`.

Bei Windows ändert sich nur das Fenster-Icon, bei macOS wird das Icon nur im Dock angezeigt.

Aufgabe 9.3 Hintergrund zeichnen

Füllen Sie den Hintergrund pro Frame mit einer Farbe mittels `screen.fill((R,G,B))`. Experimentieren Sie mit zufällig generierten Farbtönen:

```
import random
# ...
# In der Game-Schleife, im Render-Abschnitt:
r = random.randint(0, 255)
g = random.randint(0, 255)
b = random.randint(0, 255)
screen.fill((r, g, b))
```

Die Farbe muss in der Hauptschleife, aber noch vor `pg.display.flip()`, gesetzt werden.

- Was ändert sich, wenn Sie die drei Zeilen für die Farbwerte `r, g, b` vor die Hauptschleife setzen?
- Schreiben Sie den Code so um, dass die Farbe nur alle 10 Frames geändert wird.

Aufgabe 9.4 Geometrische Figuren

Zeichnen Sie geometrische Formen: Rechteck, Kreis, Linie. Nutzen Sie dafür `pg.draw.rect`, `pg.draw.circle` und `pg.draw.line`. Achten Sie darauf, **nach** dem Zeichnen `pg.display.flip()` aufzurufen. Versuchen Sie, mittels folgender Befehle einen Apfel zu zeichnen: Verwenden Sie folgende Befehle:

```
pg.draw.rect(screen, rect_color, pg.Rect(rect_x, rect_y, rect_width,
    rect_height))
pg.draw.circle(screen, circle_color, (circle_x, circle_y), circle_radius)
pg.draw.line(screen, line_color, (start_x, start_y), (end_x, end_y),
    line_width)
```

Dabei bezeichnen die Parameter Folgendes:

- `screen`: die Zeichenfläche
- `rect_color, circle_color, line_color`: Farbe als RGB-Tupel, z. B. `(255, 0, 0)` für Rot
- `pg.Rect(...)`: Rechteck-Objekt mit Position (x,y) und Grösse (width,height)
- `(circle_x, circle_y)`: Mittelpunkt des Kreises
- `circle_radius`: Radius des Kreises
- `(start_x, start_y), (end_x, end_y)`: Start- und Endpunkt der Linie
- `line_width`: Dicke der Linie in Pixeln (optional)

Aufgabe 9.5 Sonnenuntergang erstellen

Erzeugen Sie eine einfache Szene mit Himmel, Sonne und Meer. Tipp: Zeichnen Sie einen Farbverlauf am Himmel, indem Sie in einer Schleife schmale horizontale Rechtecke in leicht unterschiedlichen Farbtönen zeichnen. Die Sonne ist ein Kreis; das Meer ein Rechteck in der unteren Hälfte. Das Meer sollte ebenfalls einen Farbverlauf haben (von hellblau zu dunkelblau). Verwenden Sie folgenden Code als Vorlage:

```
# Himmel (einfacher Gradient)
for y in range(0, HEIGHT//2):
    red = 100 + int(155 * (y / (HEIGHT//2)))  # 100 ... 255
    pg.draw.rect(screen, (red, 120, 180), (0, y, WIDTH, 1))

# Sonne
# IHR CODE HIER...

# Meer
# IHR CODE HIER...
```

Optional: Lassen Sie die Sonne langsam sinken (Animation über mehrere Frames).

Aufgabe 9.6 Bewegungen (Tastatur)

Erstellen Sie ein Spieler-`Rect` (z. B. `50 x 50`) und bewegen Sie es mit den Pfeiltasten. Nutzen Sie `pg.key.get_pressed()` und begrenzen Sie die Bewegung auf das Fenster („Screen Bounds“). Verwenden Sie folgende Code-Vorlage:

```
# Vor der Game-Schleife:
player = pg.Rect(100, 100, 50, 50) # Spieler-Rechteck erstellen
speed = 5 # Bewegungsgeschwindigkeit

# In der Game-Schleife:
keys = pg.key.get_pressed() # alle gedrückten Tasten abfragen
if keys[pg.K_LEFT]:
    player.x -= speed
if keys[pg.K_RIGHT]:
    player.x += speed
if keys[pg.K_UP]:
    player.y -= speed
if keys[pg.K_DOWN]:
    player.y += speed
player.clamp_ip(screen.get_rect())  # Rechteck innerhalb des Fensters halten

pg.draw.rect(screen, (0,0,0), player) # Spieler (Rechteckt) zeichnen
```

Aufgabe 9.7 Kollisionen

Legen Sie ein `Rect` als „Ziel/Item“ an (z. B. kleiner Kreis oder Block). Prüfen Sie eine Kollision mit `player.colliderect(item)`. Bei Kollision: Position des Items neu zufällig setzen und optional Punkte zählen.

```
import random
# ...
# Vor der Game-Schleife:
item = pg.Rect(400, 300, 30, 30) # dieses Item soll gesammelt werden

# ...
# In der Game-Schleife, nach der Spieler-Bewegung:
if player.colliderect(item):
    item.topleft = (random.randint(0, WIDTH-30), random.randint(0, HEIGHT
    -30))

pg.draw.rect(screen, (255, 0, 0), item) # Item zeichnen
```

Aufgabe 9.8 Highscore anzeigen

Erstellen Sie eine Variable `score`, die bei jeder Kollision um 1 erhöht wird. Zeichnen Sie den aktuellen Punktestand mit `font.render(...)` oben links im Fenster. Verwenden Sie folgenden Code, um Text zu zeichnen:

```
# Vor der Game-Schleife:
font = pg.font.Font(None, 36)  # Schriftart und -grösse
# ...
# In der Game-Schleife, im Render-Abschnitt:
score_text = font.render(f"Score: {score}", True, (0, 0, 0))
screen.blit(score_text, (10, 10))  # Text oben links zeichnen
```

Aufgabe 9.9 Ereignisse (Keyboard / Maus)

Reagieren Sie auf `KEYDOWN`- und `MOUSEBUTTONDOWN`-Events. Beispiel: Bei Mausklick wird an der Klickposition ein kleiner Kreis gezeichnet (oder eine Partikelspur gestartet). Bei `ESC` beendet sich das Game. Da die Kreise jedes Frame neu gezeichnet werden müssen, speichern Sie die Positionen in einer Liste.

Um Ereignisse wie etwa einen Maus-Klick zu verarbeiten, verwenden Sie folgenden Code:

```
# Vor der Game-Schleife:
mouse_positions = []  # Liste für Maus-Klick-Positionen

# In der Game-Schleife, im Event-Abschnitt:
for event in pg.event.get():
    if event.type == pg.QUIT:
        running = False
    elif event.type == pg.KEYDOWN and event.key == pg.K_ESCAPE:
        running = False
    elif event.type == pg.MOUSEBUTTONDOWN:
        x, y = event.pos
        mouse_positions.append((x, y))  # Position speichern

# In der Game-Schleife, im Render-Abschnitt:
for pos in mouse_positions:
    pg.draw.circle(screen, (255, 100, 100), pos, 10)
```

Aufgabe 9.10 Medien: Bild und Sound

Laden Sie ein Bild (`.png /.jpg`) und zeichnen Sie es mit `screen.blit(...)`. Skalieren/rotieren Sie es optional. Laden Sie einen Sound (`.wav/ .ogg / .mp3`) mit `pg.mixer.Sound` und spielen Sie ihn bei einer Kollision ab.

Für Bilder:

```
# Vor der Game-Schleife:
img = pg.image.load("assets/player.jpeg") # Bild laden
img = pg.transform.scale(img, (64, 64)) # optional: skalieren
```

```
# In der Game-Schleife, im Render-Abschnitt:
screen.blit(img, player) # Bild an Spieler-Position zeichnen
```

Für Sounds:

```
# Vor der Game-Schleife:
pg.mixer.init() # Mixer initialisieren (ist nötig für Sound)
motor = pg.mixer.Sound("assets/motor.wav") # Sound laden

# In der Game-Schleife, bei Kollision:
if player.colliderect(item):
    motor.play() # Sound abspielen
```

Hinweis: Legen Sie die Medien in einem Ordner (z. B. `assets/`) ab und achten Sie auf relative Pfade. Die Dateien `player.jpeg` und `motor.mp3` finden Sie auf Moodle.

Optional: Skalieren Sie das Bild auf eine bestimmte Höhe, wobei das Seitenverhältnis beibehalten wird, indem Sie das Seitenverhältnis mit den Befehlen `img.get_width()` und `img.get_height()` berechnen.

Tipp: Viele kostenlose Soundeffekte finden Sie unter https://pixabay.com/sound-effects.

Aufgabe (Challenge) 9.11 Bonus: Kleines Sammelspiel

Verbinden Sie die vorherigen Bausteine: Bewegen Sie den Spieler, sammeln Sie Items (Punkte zählen), spielen Sie dabei einen Sound ab und zeichnen Sie im Hintergrund Ihren Sonnenuntergang. Begrenzen Sie die Spielzeit auf 60 Sekunden und zeigen Sie die verbleibende Zeit im Fenstertitel an.

Aufgabe 9.12 Pong-Spiel

Erstellen Sie das klassische Pong-Spiel mit zwei Paddles und einem Ball.

Das folgende Video zeigt das fertige Pong-Spiel: https://youtu.be/vCoBgJlUg_c

Die Paddles werden mit den Tasten `W/S` (links) und `UP/DOWN` (rechts) gesteuert. Der Ball bewegt sich automatisch und prallt von den Paddles und den oberen/unteren Bildschirmrändern ab. Zählen Sie die Punkte, wenn ein Spieler den Ball am Gegner vorbei spielt. Gehen Sie Schritt für Schritt vor, indem Sie folgende Elemente umsetzen:

1. Rechtecke für Paddles und Ball zeichnen
2. Mittellinie zeichnen
3. Tastatur-Eingaben für Paddle-Bewegung (`W/S` und `UP/DOWN`)
4. Ball von der Mitte aus in eine zufällige Richtung starten lassen
5. Kollisionserkennung für Ball und Paddles, Richtung von Ball umkehren bei Kollision oder sofern der Ball die oberen/unteren Ränder berührt
6. Punkte zählen und anzeigen
7. Soundeffekte bei Kollisionen und Punkten

Achtung

Wichtiger Hinweis 9.2 (Hinweise zu Pong)**:**
Die folgende Sammlung von Hinweisen zeigt kleine Bausteine, die im Pong-Beispiel verwendet werden und Ihnen beim Verständnis oder bei eigenen Varianten helfen.

- **Zufällige Start-Richtung für den Ball:**

```
import random
...
ball_speed = 5
# Geschwindigkeit als separate Variablen
ball_speed_x = random.choice((-1, 1)) * ball_speed
ball_speed_y = random.choice((-1, 1)) * ball_speed

# In der Game-Schleife:
ball.x += ball_speed_x
ball.y += ball_speed_y
```

- **Rect-Ränder nutzen und Kollisionen „sauber“ auflösen:**

```
if ball.colliderect(left_paddle):
    ball_speed_x *= -1     # x-Richtung umkehren (Abprall)
```

- **Abprall oben/unten (Wände):**

```
if ball.top <= 0 or ball.bottom >= HEIGHT:
    ball_speed_y *= -1
```

- **Trefferwinkel variieren:** Je weiter oben/unten am Paddle getroffen wird, desto mehr vertikale Geschwindigkeit.

```
offset = (ball.centery - left_paddle.centery) / (paddle_h / 2)  #
    -1..+1
ball_speed_y = ball_speed * offset
```

- **Ball nach Punkt neu zentrieren (als Funktion):**

```
def reset_ball(direction):  # direction: -1 nach links, +1 nach rechts
    ball.center = (WIDTH // 2, HEIGHT // 2)
    global ball_speed_x, ball_speed_y # Zugriff auf globale Variablen
    ball_speed_x = direction * ball_speed
    ball_speed_y = random.choice((-1, 1)) * ball_speed
```

- **Nützliche Rect-Eigenschaften:** `left`, `right`, `top`, `bottom`, `center`, `centery` helfen bei Ausrichtung und Kollisionen.

Aufgabe (Challenge) 9.13 Pong-Bonus

Fügen Sie dem Pong-Spiel folgende Features hinzu:

- Soundeffekte bei Paddle-Kollision und Punktgewinn
- Startbildschirm mit Anweisungen
- Game-Over-Bildschirm nach einer bestimmten Punktzahl (z. B. 10 Punkte)
- Paddle-Geschwindigkeit erhöhen, wenn der Ball getroffen wird
- Hintergrundmusik während des Spiels

Aufgabe 9.14 Videoaufnahme des Spiels

Nehmen Sie ein kurzes Video (ca. 1-2 Minuten) auf, das das Gameplay Ihres Spiels zeigt. Laden Sie das Video auf eine Plattform (z. B. YouTube, Vimeo) hoch und fügen Sie den Link auf Moodle ein. Für die Bildschirmaufnahme können Sie folgende Tools verwenden:

- **Windows:** Verwenden Sie die Bildschirmaufnahme-Funktion von PowerPoint (unter „Einfügen“ → „Bildschirmaufnahme“) oder die Xbox Game Bar (⊞+G), beenden Sie die Aufnahme mit ⌘+⊞+R
- **macOS:** Verwenden Sie die integrierte Bildschirmaufnahme (⇧+⌘+5), beenden Sie die Aufnahme mit ⌘+ctrl+Esc

9.2 Game-Auftrag

9.2.1 Thema

Im Folgenden wählen Sie in Zweier- bis Dreier-Gruppen ein Game-Thema aus und entwickeln ein kleines 2D-Game mit `pygame-ce`. Mögliche Themen sind:

- **Sammelspiel:** Bewegen Sie eine Spielfigur (z. B. Auto, Raumschiff) und sammeln Sie Items ein (z. B. Treibstoff, Sterne). Jedes eingesammelte Item gibt Punkte.
- **Endlos-Runner:** Steuern Sie eine Spielfigur (z. B. Läufer, Auto) und weichen Sie Hindernissen aus (z. B. Mauern, andere Autos). Jedes überstandene Hindernis gibt Punkte.
- **Fangspiel:** Steuern Sie eine Spielfigur (z. B. Katze, Netz) und fangen Sie herumspringende Objekte (z. B. Mäuse, Schmetterlinge). Jedes gefangene Objekt gibt Punkte.
- **Shooter (z.B. Space Invader):** Steuern Sie ein Raumschiff und schiessen Sie auf heranna-hende Gegner. Jeder abgeschossene Gegner gibt Punkte.

Weitere Themen können in Absprache mit der Lehrperson gewählt werden. Wichtig ist, dass das Spiel die geforderten Python-Konzepte sinnvoll einsetzt (siehe Anforderungen) und die Komplexität angemessen ist (nicht zu einfach, aber auch nicht zu komplex).

9.2.2 Anforderungen

Ihr Projekt sollte folgende, essentielle Python-Konzepte sinnvoll einsetzen

- ☐ Verwendung von `pygame-ce` für Fenster, Game-Schleife, Ereignisse, Zeichnen, Kollisionen und Medien (Bilder, Sound), Steuerung mit Tastatur und/oder Maus
- ☐ Verwendung physikalischer Formeln für Bewegungen (z. B. Geschwindigkeit, Beschleunigung, Schwerkraft)
- ☐ Verwendung von **Variablen** zur Speicherung von Spielzuständen (z. B. Spielerposition, Punktestand, verbleibende Zeit)
- ☐ Verwendung von **Funktionen** zur Strukturierung des Codes (z. B. `draw_player()`, `update_game()`, `handle_input()`), mit sowie ohne `return`-Wert
- ☐ Verwendung von **logischen Ausdrücken** und **Schleifen** zur Steuerung des Spielablaufs (z. B. Kollisionserkennung, Punkte zählen, Spielzeit)
- ☐ Verwendung von **Datenstrukturen** (Listen, Dictionaries) zur Verwaltung von mehreren Spielobjekten (z. B. Liste von Items, Liste von Hindernissen)
- ☐ Verwendung von **Klassen** zur Modellierung von Spielobjekten (z. B. `class Player`, `class Item`, `class Obstacle`)
- ☐ Verwendung mehrerer Python-Unterdateien zur besseren Strukturierung des Codes (z. B. `game.py`, `player.py`, `item.py`)

Ihr Game muss **Kommentare** enthalten, die den Code erklären und die Struktur des Programms verdeutlichen. Verwenden Sie sowohl einzeilige Kommentare (mit `#`) als auch mehrzeilige Kommentare (mit `"..."`) für Funktionen und Klassen. **Schauen Sie sich die detaillierten Bewertungskriterien auf Moodle an, um sicherzustellen, dass Ihr Code den Anforderungen entspricht.**

9.2.3 Bonus

- ☐ Python-spezifisch:
  - ☐ Verwendung von `try/except` zur Behandlung von Fehlern (z. B. Laden von Medien, Division durch Null)
  - ☐ Verwendung von `with`-Anweisung zum sicheren Öffnen und Schliessen von Dateien (z. B. Highscore speichern)
  - ☐ Wenn Sie ein Punktesystem mit Highscore (gespeichert in einer Datei) implementieren.
- ☐ Game-spezifisch:
  - ☐ Wenn Sie viele Power-Ups (z. B. temporäre Unverwundbarkeit, Doppelte Punkte) einbauen.
  - ☐ Wenn Sie viele eigens erstellte Hintergrundmusik und/oder Soundeffekte (z. B. bei Kollision, Item-Sammlung) einbauen.
  - ☐ Wenn Sie das Spiel mit einem Gamepad/Controller steuern können.
  - ☐ Wenn Sie einen Online-Multiplayer-Modus (z. B. über LAN) implementieren.
- ☐ Weitere Ideen für Bonus:
  - ☐ Wenn Sie das Spiel mit einem Level-Editor (zum Erstellen eigener Level) erweitern können.
  - ☐ Wenn Sie das Spiel mit einem Story-Modus (mit Handlung und Charakteren) erweitern können.
  - ☐ Wenn Sie das Spiel mit einem besonders raffinierten Achievements-System (Erfolge) erweitern können.
  - ☐ Wenn Sie das Spiel mit einem Debug-Modus (zum Testen und Entwickeln) erweitern können.

9.3 Bewertung

9.3.1 Projektbewertung

Die Bewertung Ihres Projekts erfolgt anhand der auf Moodle aufgelisteten Kriterien. Die maximale Punktzahl beträgt 100 Punkte.

9.3.2 Gruppen-Besprechung des Spiels

Im Anschluss an Ihre Abgabe findet eine Besprechung Ihres Spiels in Gruppen mit der Lehrperson statt (ca. 15 Minuten pro Gruppe). Dabei werden Ihnen Fragen zu Ihrem Spiel gestellt, um Ihr Verständnis und Ihre Reflexion zu überprüfen. Achten Sie darauf, in Ihrem Schlussbereicht zu erwähnen, wer für welche Teile zuständig war – das Verständnis von jedem Gruppenmitglied wird speziell überprüft. Die Besprechung resultiert in einer **individuellen** Verständnis-Prozentzahl von 0%–100% Punkten, welche mit Ihrer Endnote multiplizipert wird.

Ihre Endnote wird wie folgt berechnet:

$$\text{Endnote} = 5 \cdot \left(\text{Projekt-Punkte} \cdot \left(\frac{\text{Verständnis-Prozentzahl}}{100}\right)\right) + 1$$

Achtung

Wichtiger Hinweis 9.3 (KI-Tools)**:**
Es wird von Ihnen erwartet, dass Sie sich aktiv an der Entwicklung des Spiels beteiligen und die geforderten Python-Konzepte verstehen und anwenden können. ChatGPT oder andere KI-Tools dürfen Sie verwenden, solange Sie Ihren gesamten Code verstehen und erklären können. Wie Sie im obigen Benotungsschema erkennen können, kann Ihre Note durch mangelndes Verständnis stark beeinträchtigt werden (bis zur Note 1). Falls Sie ChatGPT oder andere KI-Tools verwenden, müssen Sie dies im Code klar dokumentieren (z.B. mit Kommentaren oder in einer Begleit-Dokumentation). Jede Zeile Code muss mindestens einer Person zugeordnet sein!

Anhang A

Lernziele

Die folgenden Lernziele geben Ihnen einen Überblick über die zentralen Konzepte und Fähigkeiten, die in den Kapiteln behandelt werden. Sie dienen Ihrer Orientierung und unterstützen Sie dabei, Ihren Lernfortschritt zu verfolgen. Bitte beachten Sie, dass diese Auflistung nicht abschliessend ist und das Lösen der Übungsaufgaben sowie das Verständnis der behandelten Konzepte im Skript entscheidend für Ihren Lernerfolg sind.

Lernziele Kapitel 2: Einführung in Python und erste Schleifen

- ☐ Ich kann mit der Turtle einfache Formen zeichnen (Dreiecke, Rechtecke, Blitz, etc.)
- ☐ Ich kann mit einem breiten Stift ausgefüllte Rechtecke zeichnen (Türe beim Haus)
- ☐ Ich setze die `for _ in range(zahl)`-Schleife ein, um sich wiederholende Tätigkeiten auszuführen.
- ☐ Ich kann mit der Turtle regelmässige Vielecke (Vierecke, Fünfecke, Sechsecke etc.) zeichnen.
- ☐ Ich kann „Kreise“ als Vielecke mit grosser Eckenzahl zeichnen.
- ☐ Ich kann verschachtelte Schleifen verwenden (`for _ in range(zahl)`-Schleifen innerhalb von `for _ in range(zahl)`-Schleifen).
- ☐ Ich kann Text in der Konsole mit `print("...")` ausgeben.
- ☐ Ich setze die Operationen `+` und `*`, um komplizierte Texte (Zeichenketten) zu erzeugen.
- ☐ Ich kenne die Operationen auf Zahlen, die auf Seite 19 oben aufgelistet sind, und kann diese auch einsetzen.
- ☐ Ich kann innerhalb eines Programms eine Streckenlänge mit Pythagoras berechnen.
- ☐ Ich kann die Farbe und Breite des Turtle-Stifts verändern (`t.color("...")`, `t.width(...)`)
- ☐ Ich kann die Turtle bewegen, ohne zu zeichnen, indem ich den „Stift“ ab- sowie aufsetze (`t.pu()`, `t.pd()`)

Lernziele Kapitel 3: Variablen, Datentypen & Debugging

- ☐ Ich kann neue Variablen erstellen, um die Resultate einfacher Berechnungen zu speichern
- ☐ Ich kann mit dem Gleich-Operator (`=`) eine neue Variable erstellen und kann diese in einem Code sinnvoll verwenden.
- ☐ Ich kann Speicherinhalte ändern, indem ich die Befehle `+=`, `-=`, `*=` und `/=` nutze.
- ☐ Ich kann mithilfe des `input`-Befehls neue Variablen während der Ausführung des Codes erstellen.
- ☐ Ich kenne den Unterschied zwischen der Division (`/`) und der Ganzzahl-Division (`//`) und kann beide Operatoren in geeigneten Situationen anwenden.

- ☐ Ich kann den Modulo-Operator `%` verwenden, um den Rest einer Ganzzahl-Division zu berechnen.
- ☐ Ich kann Variablen nutzen, um eine Spirale oder ähnliche, sich stetig vergrössernde oder verkleinernde Formen zu zeichnen.
- ☐ Ich kann Zahlenfolgen (z.B. alle Quadratzahlen bis zu einem bestimmten Wert) mit Python erstellen.

Lernziele Kapitel 4: Funktionen

- ☐ Ich kann eigene Befehle in Python definieren und einsetzen.
- ☐ Ich verstehe den Unterschied zwischen Haupt- und Unterprogramm.
- ☐ Ich kann Befehle mit Parameter definieren und einsetzen, wie z.B. `quadrat(seite)`.
- ☐ Ich kann die Lebensdauer von Variablen (inkl. Parametern) beschreiben (lokal oder global).
- ☐ Ich kann beschreiben, wie sich ein Parameter von andern Variablen unterscheidet.
- ☐ Ich kann Befehle mit mehreren Parameter definieren und einsetzen, wie z.B. `vieleck(ecken , laenge, farbe)`.
- ☐ Ich kann einen Befehl innerhalb eines anderen Befehls aufrufen und dabei Variablenwerte übergeben.
- ☐ Ich kann Speicherinhalte ändern, indem ich den Inhalt durch den Wert eines Ausdrucks überschreibe, z.B. `seite = seite * 2 + 1`
- ☐ Ich kann einfache Befehle kombinieren, um komplexe Probleme zu lösen (Beispiel Häuserreihe).
- ☐ Ich kann Funktionen erstellen, die einen oder mehrere Parameter sowie einen `return`-Wert haben.
- ☐ Ich verstehe den Unterschied zwischen einem `print`-Befehl und einem `return`-Befehl
- ☐ Ich kann das Resultat einer Funktion mit `return` in einer Variable im Hauptprogramm speichern und diese weiterverwenden
- ☐ Ich kann Funktionen mit einem `return`-Ausdruck erstellen, den Rückgabewert der Funktion im Hauptprogramm als Variable speichern und den Wert der Variable mit `print` anzeigen lassen.
- ☐ Ich kann eine (Unter-)Funktion innerhalb einer anderen (Haupt-)Funktion aufrufen, und den Rückgabewert der Unterfunktion in der Hauptfunktion weiterverwenden.
- ☐ Ich nutze den modularen Programmentwurf, um komplexe Probleme in Teilprobleme aufzuteilen.
- ☐ Ich kann den `return`-Wert einer Funktion direkt und ohne Zwischenspeicherung in einem Ausdruck verwenden, z.B. `if rechteck_flaeche(19, 12) < 100: ....`

Lernziele Kapitel 5: Verzweigungen und Logische Ausdrücke

- ☐ Ich kann die `if`-Anweisung einsetzen und verwende dabei die Vergleichsoperatoren `==`, `!=`, `>`, `<`, `<=` und `>=`.
- ☐ Ich kann die `if-else`-Struktur für Verzweigungen mit zwei Fällen verwenden.
- ☐ Ich kann die `if-elif-else`-Anweisung für Verzweigungen mit beliebig vielen Fällen verwenden.
- ☐ Ich kenne den Unterschied im Programmablauf zwischen der `if-elif-else`- und der `if-if -else`-Struktur.
- ☐ Ich kann die logischen Operatoren `and`, `or` sowie `not` einsetzen, um boolsche Ausdrucke zu kombinieren.
- ☐ Ich kann die `break`-Anweisung einsetzen, um Schleifen unter bestimmten Bedingungen abzubrechen.
- ☐ Ich kann `while`-Schleifen einsetzen, um eine Schleife solange auszuführen wie eine Bedingung wahr ist.

- ☐ Ich kann erklären, weshalb ein Code der nach einem ausgeführten `for _ in range(zahl)`-Ausdruck steht, nicht mehr ausgeführt wird.
- ☐ Ich kann für ein einfaches, gegebenes Code-Beispiel bestimmen, wie häufig eine bestimmte `while`-Schleife ausgeführt wird.
- ☐ ich kann die geeignetste Schleifenart auswählen, um ein bestimmtes Problem zu lösen (`for _ in range(zahl)` mit `break` oder `while`).
- ☐ Ich kann unterschiedliche Schleifen-Typen mit einem Flussdiagramm modellieren.
- ☐ Ich kann den `return`-Wert einer Funktion direkt und ohne Zwischenspeicherung in einem Ausdruck verwenden, z.B. `if rechteck_flaeche(19, 12) < 100: ...`.

Lernziele Kapitel 6: Datenstrukturen (Listen, Dictionaries)

- ☐ **Listen: Grundlagen**
  - ☐ Ich kann Listen in Python erstellen.
  - ☐ Ich kann mit dem Index einzelne Elemente der Liste abrufen und verändern.
  - ☐ Ich kann die Anzahl Elemente einer Liste in Python mit `len(liste)` berechnen lassen.
  - ☐ Ich kann die Elemente einer Liste in einer Schleife (`for _ in range(zahl)`, `while` oder `for...in`) durchlaufen.
  - ☐ Ich kann ein Programm schreiben, das in einer Liste die Elemente mit bestimmten Eigenschaften findet (z.B. das Maximum oder alle ungeraden Zahlen).
  - ☐ Ich kann die Elemente einer Liste auf einen Wert reduzieren (z.B. die Summe der Listenelemente berechnen).
  - ☐ Ich kann Listen als Parameter an Funktionen übergeben und innerhalb der Funktion verarbeiten.
- ☐ **Algorithmen**
  - ☐ Ich kann *bubble sort* in Python programmieren und verstehe den Algorithmus im Detail.
  - ☐ Ich kann die binäre Suche in Python programmieren und verstehe den Algorithmus im Detail.
- ☐ **Dynamische Listen**
  - ☐ Ich kann Listen mit dem Befehl `.append(...)` erweitern.
  - ☐ Ich kann Listen mit dem Befehl `.pop()` kürzen.
  - ☐ Ich kann Listen mit dem Befehl `.insert(position, wert)` an einer bestimmten Position erweitern.
- ☐ **Dictionaries**
  - ☐ Ich kann Dictionaries (Wörterbücher) in Python erstellen und verwenden.
  - ☐ Ich kann die Struktur von Schlüssel-Wert-Paaren in Dictionaries erklären und verstehe den Unterschied zu listenbasierten Datenstrukturen.
  - ☐ Ich kann mit Schlüsseln auf die Werte in einem Dictionary zugreifen.
  - ☐ Ich kann neue Schlüssel-Wert-Paare zu einem Dictionary hinzufügen.
  - ☐ Ich kann bestehende Werte in einem Dictionary ändern.
  - ☐ Ich kann über alle Schlüssel eines Dictionaries mit einer `for`-Schleife iterieren.
  - ☐ Ich kann prüfen, ob ein bestimmter Schlüssel in einem Dictionary vorhanden ist (mit `if key in dictionary:`).
  - ☐ Ich kann Dictionaries als Parameter an Funktionen übergeben und innerhalb der Funktion verarbeiten.
  - ☐ Ich kann komplexere Datenstrukturen erstellen, indem ich Listen und Dictionaries kombiniere (z.B. Liste von Dictionaries, Dictionary von Listen).
  - ☐ Ich kann alltägliche Anwendungsfälle für Dictionaries identifizieren (z.B. Telefonbuch, Lagerbestand, Preisliste).
  - ☐ Ich kann die Länge eines Dictionaries, also die Anzahl der Schlüssel-Wert-Paare, mit dem

Befehl `len(dictionary)` bestimmen.
  - ☐ Ich kann einen Schlüssel-Wert-Paar aus einem Dictionary entfernen.
  - ☐ Ich verstehe, dass Dictionaries ungeordnet sind und die Reihenfolge der Elemente keine Rolle spielt.
- ☐ **Mengen (Sets)**
  - ☐ Ich kann Mengen in Python erstellen und verwenden.
  - ☐ Ich kann die grundlegenden Operationen auf Mengen durchführen, wie z.B. Vereinigung, Schnittmenge und Differenz.
  - ☐ Ich kann die grundlegenden Operationen kombinieren, um komplexere Mengenoperationen durchzuführen.
  - ☐ Ich kann typische Anwendungsfälle für Mengen nennen, wie z.B. das Entfernen von Duplikaten aus einer Liste oder das Überprüfen von Mitgliedschaften.
- ☐ **Tupel**
  - ☐ Ich kann Tupel in Python erstellen und auf ihre Elemente zugreifen.
  - ☐ Ich kann den Hauptunterschied zwischen Tupeln und Listen erklären, nämlich die Unveränderlichkeit (Immutability) von Tupeln.
  - ☐ Ich kann die wesentlichen Eigenschaften von Tupeln (geordnet, heterogen) beschreiben.
  - ☐ Ich kann typische Anwendungsfälle für Tupel nennen, wie z.B. die Rückgabe mehrerer Werte aus einer Funktion oder die Verwendung als Schlüssel in einem Dictionary.

Lernziele Kapitel 7: Klassen und Objektorientierte Programmierung

- ☐ Ich kann erklären, was eine Klasse ist und wie sie sich von einem Objekt unterscheidet.
- ☐ Ich kann eine einfache Klasse mit Attributen und Methoden definieren.
- ☐ Ich verstehe den Zweck und die Funktionsweise des Konstruktors (`__init__`).
- ☐ Ich kann erklären, was der Parameter `self` in Methoden bedeutet und wie er verwendet wird.
- ☐ Ich kann Objekte (Instanzen) einer Klasse erstellen und verwenden.
- ☐ Ich kann auf Attribute und Methoden eines Objekts zugreifen.
- ☐ Ich verstehe den Unterschied zwischen Klassenattributen und Instanzattributen.
- ☐ Ich kann das Konzept der Vererbung erklären und einfache Vererbungshierarchien erstellen.
- ☐ Ich kann Methoden in einer Kindklasse überschreiben und dabei die Methoden der Elternklasse mit `super()` aufrufen.
- ☐ Ich kann reale Probleme mit Hilfe objektorientierter Programmierung modellieren.
- ☐ Ich kann die Vorteile der objektorientierten Programmierung für komplexe Anwendungen erklären.

Anhang B

Nützliche Shortcuts

Mit Shortcuts können Sie Ihre Produktivität in vielen Bereichen boosten, daher empfiehlt es sich, diese zu lernen, ebenso wie das Zehn-Finger-System. Falls Sie letzteres noch nicht beherrschen, sollten Sie dieses zuerst auf tipp10.com trainieren.

Folgende Shortcut-Liste erhebt keinen Anspruch auf Vollständigkeit. Falls Sie weitere hilfreiche Shortcuts kennen, können Sie diese gerne an Cyril Wendl senden. Sie dürfen die Übersicht der Shortcuts an jeder Prüfung verwenden.

Aktion	Windows	MacOS
Cursor bewegen	→, ←	→, ←
Cursor bewegen (Wörter)	ctrl + → ctrl + ←	⌥ + → ⌥ + ←
Wörter markieren	ctrl + ⇧ + → ctrl + ⇧ + ←	⌥ + ⇧ + → ⌥ + ⇧ + ←
Zum Anfang / Ende der Zeile gehen	Home End	⌘ + ← ⌘ + →
Ganze Zeile markieren	⇧ + Home ⇧ + End	⌘ + ⇧ + ← ⌘ + ⇧ + →
Gesamten Text markieren	ctrl + A	⌘ + A
Emoji einfügen	⊞ + .	fn + E
Datei/Text kopieren	ctrl + C	⌘ + C
Datei/Text ausschneiden (= kopieren + löschen)	ctrl + X	⌘ + X ⌘ + C [a]
Datei/Text einfügen	ctrl + V	⌘ + V
Text suchen	ctrl + F	⌘ + F
Datei speichern	ctrl + S	⌘ + S
Rückgängig („undo“)	ctrl + Z	⌘ + Z
Vorwärts („redo“)	ctrl + ⇧ + Z	⌘ + ⇧ + Z
Fenster wechseln Fenster zurückwechseln	ctrl + ⇥ ctrl + ⇧ + ⇥	⌘ + ⇥ ⌘ + ⇧ + ⇥ [b]
Programm schliessen	⌥ + F4	⌘ + Q
Computer sperren Fenster links anordnen Fenster rechts anordnen	⊞ + L ⊞ + ← ⊞ + →	⌘ + ctrl + Q Nicht möglich [c]

[a] Um Dateien auf MacOS zu verschieben (nicht kopieren): ⌘ + C, danach ⌘ + ⌥ + V

[b] Um auf MacOS zwischen mehreren Fenstern derselben Anwendung wechseln: ⌘ + <

[c] Keine native Unterstützung durch MacOS, allerdings möglich mit Drittanbieter-Apps wie z.B. dieser Link (Windows), bzw. ⌘ + C (MacOS)

Tabelle B.1: Allgemeine Shortcuts

Aktion	Windows	MacOS
Code Einrücken [a]	⇥	⇥
Code Ausrücken	⇧ + ⇥	⇧ + ⇥

[a] Klappt auch, wenn die einzurückenden / auszurückenden Zeilen nur teilweise markiert sind

Tabelle B.2: Shortcuts für Code

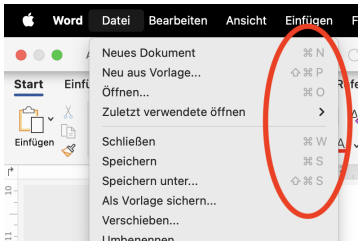
Aktion	Windows	MacOS
Zu Tab rechts Zu Tab links	ctrl + ⇥ ctrl + ⇧ + ⇥	⌘ + ⌥ + → ⌘ + ⌥ + ←
Zu Tab 1, 2, ... gehen	ctrl + zahl	⌘ + zahl
Tab schliessen	ctrl + W	⌘ + W
Neuer Tab	ctrl + T	⌘ + T
Geschlossenen Tab wieder öffnen	ctrl + ⇧ + T	⌘ + ⇧ + T
Seite neu laden	ctrl + R	⌘ + R

Tabelle B.3: Browser-Shortcuts

Zeichen	Windows	MacOS
[	Alt Gr + ü	⌥ + 5
]	Alt Gr + !	⌥ + 6
{	Alt Gr + ä	⌥ + 8
}	Alt Gr + $	⌥ + 9
\	Alt Gr + \	⌥ + ⇧ + 7
|	Alt Gr + 7	⌥ + 7
&	⇧ + 6	⇧ + 6
%	⇧ + 5	⇧ + 5

Tabelle B.4: Spezial-Zeichen

Viele weitere Shortcuts können durch Ausprobieren erraten werden: Häufig steht der Anfangs-Buchstabe des englischen Wortes für die Aktion. So kann man beispielsweise aus den meisten Programmen drucken (en. *print*), indem man die Abkürzung ctrl + P (Windows) bzw. ⌘ + P (MacOS) verwendet. Bei MacOS können zudem viele Abkürzungen über das Programm-Menu (Menuleiste oben am Bildschirm) eingesehen werden:



Anhang C

Details

C.1 Division mit Rest

Definition C.1 (Kongruenz ganzer Zahlen)**:**
Sei $m > 0$ eine fest gewählte natürliche Zahl. Seien a und b ganze Zahlen. Dann heissen a und b **kongruent modulo** m, geschrieben

$$a \equiv b \pmod{m},$$

falls m die Differenz $(a - b)$ teilt.

Definition C.2 (Modulo-Operation)**:**
Sei $m \geq 2$ eine fest gewählte ganze Zahl und a eine ganze Zahl. Dann ist a kongruent modulo m zu genau einer Zahl $b \in \{0, 1, \ldots, m-1\}$. Diese Zahl b bezeichnen wir mit dem Ausdruck $a \ \% \ m$.

Beispiel C.1:
Auf dem Planeten *Vulcan* dauert ein Tag nur 5 Stunden. Deshalb verwenden die *Vulcanians* Uhren der unten abgebildeten Form. Diese Uhren verfügt lediglich über einen Stundenzeiger.

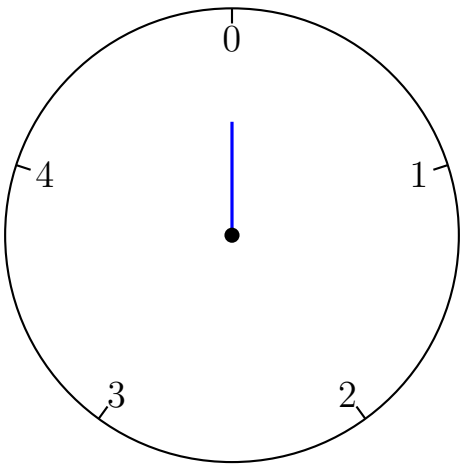


Abbildung C.1: Uhr der *Vulcanians*

vergangene Zeit T in Stunden:	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6
T % 5:	4	0	1	2	3	4	0	1	2	3	4	0	1

Tabelle C.1: Es sei die T die ganzzahlige Anzahl der vergangenen Stunden. Dann ist T % 5 die Uhrzeit, welche auf der Uhr abgelesen werden kann.

Beispiel C.2 (Modulo-Operation)**:**
Wir geben hier einige Beispiele an:

$$\begin{aligned} 10 \ \% \ 7 &= 3 \\ -3 \ \% \ 5 &= 2 \\ -17 \ \% \ 3 &= 1 \\ -7 \ \% \ 3 &= 2 \end{aligned}$$

Falls Sie noch mehr über modulare Arithmetik (und ihre weitreichende Bedeutung) erfahren möchten, empfehlen wir Ihnen wärmstens, das Buch [1] zu studieren.

C.2 Umrechnung von Basis a zu Basis b in Python

Es sei

$$x = x_n x_{n-1} \dots x_2 x_1 x_0 = x_n a^n + x_{n-1} a^{n-1} + \dots + x_2 a^2 + x_1 a^1 + x_0 a^0$$

die Darstellung der Zahl x in Basis $a \geq 2$. Wir wollen die Darstellung

$$x = x'_m x'_{m-1} \dots x'_2 x'_1 x'_0 = x'_m b^m + x'_{m-1} b^{m-1} + \dots + x'_2 b^2 + x'_1 b^1 + x'_0 b^0$$

von x in Basis $b \geq 2$ finden. Die Ziffern $x'_m, \dots, x'_1, x'_0$ sind gesucht.

Wir berechnen

$$\begin{aligned} x \ \% \ b &= (x'_m b^m + x'_{m-1} b^{m-1} + \dots + x'_2 b^2 + x'_1 b^1 + x'_0 b^0) \ \% \ b = \\ &= (x'_m b^m \ \% \ b + x'_{m-1} b^{m-1} \ \% \ b + \dots + x'_2 b^2 \ \% \ b + x'_1 b^1 \ \% \ b + x'_0 b^0 \ \% \ b) \ \% \ b = \\ &= (0 + 0 + \dots + 0 + x'_0 \ \% \ b) \ \% \ b = (x'_0) \ \% \ b = x'_0, \end{aligned}$$

wobei wir die Formel $(x + y) \ \% \ a = (x \ \% \ a + y \ \% \ a) \ \% \ a$ ohne Beweis verwendet haben. Somit ist $x \ \% \ b = x'_0$ die Ziffer (das Gewicht) des kleinsten Stellenwerts der Zahl x in Basis b.

Nun berechnen wir

$$\begin{aligned} x^{(1)} := x//b &= (x'_m b^m + x'_{m-1} b^{m-1} + \dots + x'_2 b^2 + x'_1 b^1 + x'_0 b^0)//b = \\ &= x'_m b^m//b + x'_{m-1} b^{m-1}//b + \dots + x'_2 b^2//b + x'_1 b^1//b + x'_0 b^0//b = \\ &= x'_m b^{m-1} + x'_{m-1} b^{m-2} + \dots + x'_2 b^1 + x'_1 b^0. \end{aligned}$$

Danach erhalten wir die zweithinterste Ziffer x'_1 durch $x^{(1)} \ \% \ b$ und so weiter. In Programm C.1 ist eine Python-Funktion gegeben, welche die Umrechnung von Basis a in Basis b berechnet.

```python
def base_a_to_base_b(number_a, a, b):
    # der String number_a ist eine Zahlendarstellung in Basis a
    if number_a == "0":
        return "0"

    number_10 = 0
    k = 0
    n = len(number_a)
    while k < len(number_a):
        number_10 += int(number_a[k]) * a ** (n - 1 - k)
        k += 1

    number_b = ""  # leerer String
    while number_10 > 0:
        number_b += str(number_10 % b)
        number_10 //= b

    return number_b[::-1]


# Beispiel
print(base_a_to_base_b("2310213647", 8, 9))
```

Programm C.1: `base_a_to_base_b.py`

GitHub-Tutorial

In diesem Tutorial lernen Sie Schritt für Schritt, wie Sie:

- einen **GitHub-Account** erstellen,
- ein neues **Repository** anlegen,
- Ihren Rechner via **SSH** mit GitHub verbinden,
- und ein bestehendes Python-Projekt mit **Git** (`add`, `commit`, `push`) hochladen.

Git bietet folgende Haupt-Vorteile gegenüber dem Arbeiten auf dem lokalen Computer:

- **Versionskontrolle**: Änderungen werden protokolliert, und frühere Versionen können leicht wiederhergestellt werden.
- **Zusammenarbeit**: Mehrere Personen können gleichzeitig an einem Projekt arbeiten, ohne sich gegenseitig zu stören.
- **Backup**: Der Code ist sicher in der Cloud gespeichert und kann von überall abgerufen werden.
- **Automatisierte Pipelines**: Möglichkeit, bei jeder neuen Version einen automatischen Test- und Deployment-Prozess zu starten, beispielsweise, um eine Dash-App auf einem Server zu aktualisieren.

Git und VS Code vorbereiten

Installieren Sie Git mit folgendem Befehl:

```
# macOS
brew install git

# Linux (Ubuntu/Debian)
sudo apt install git
```

Für Windows: Laden Sie Git von https://git-scm.com/download/win herunter und installieren Sie es.

GitHub-Account erstellen

Öffnen Sie https://github.com und klicken Sie auf **Sign up**. Folgen Sie den Schritten, um einen Account anzulegen.

SSH-Schlüssel erstellen und hinterlegen

Erzeugen Sie ein Schlüsselpaar, laden Sie es in den SSH-Agent und fügen Sie den Public Key bei GitHub ein.

Führen Sie folgenden Code auf Ihrem Computer aus (Terminal auf macOS oder Linux, bzw. Git Bash auf Windows):

```
ssh-keygen -t ed25519 -C "ihre_email@schule.ch"
eval "$(ssh-agent -s)"
ssh-add ~/.ssh/id_ed25519
pbcopy < ~/.ssh/id_ed25519.pub  # Public Key in Zwischenablage kopieren
```

Auf GitHub: **Settings** → **SSH and GPG keys** → **New SSH key**.

Testen Sie die Verbindung:

```
ssh -T git@github.com
```

Git konfigurieren

Damit bei jedem commit auf Git die richtigen Angaben verwendet werden, konfigurieren Sie Git mit Ihrem Namen und Ihrer E-Mail-Adresse. Zusätzlich stellen Sie den Standard-Branch auf `main` ein:

```
git config --global user.name "Vorname Nachname"
git config --global user.email "vorname.nachname@stud.edu.zh.ch"
git config --global init.defaultBranch main
```

Neues Repository auf GitHub

Erstellen Sie über das „+“ oben rechts ein **New repository**, geben Sie einen Namen (z. B. `mein-python-projekt`) und wählen Sie „Public“.

Projekt vorbereiten

Wechseln Sie in den Ordner Ihres Projekts:

```
cd /pfad/zu/projekt
```

Erstellen Sie eine `.gitignore`-Datei für Python:

```
__pycache__/
*.pyc
.venv/
.DS_Store
.vscode/
```

Initialisieren, Committen, Pushen

```
git init
git add .
git commit -m "Erster commit!"

git remote add origin git@github.com:<user>/<repo>.git
git push -u origin main
```

Arbeiten mit VS Code

Öffnen Sie den Projektordner in VS Code. Verwenden Sie die **Source Control**-Ansicht (Git-Symbol links), um Änderungen zu **stagen**, zu **committen** und zu **pushen**.

Nützliche Befehle

```
# Änderungen abrufen
git pull

# Remote-URL prüfen
git remote -v
```

C.3 Python Cheatsheet

Einführung

Python-Grundgerüst

Ein einfaches Python-Programm:

```
# Dies ist ein Kommentar
print("Hallo Welt!")  # Ausgabe von Text
```

Grundrechenarten

```
# Addition
3 + 5       # ergibt 8

# Subtraktion
10 - 4      # ergibt 6

# Multiplikation
3 * 7       # ergibt 21

# Division
10 / 3      # ergibt 3.3333...

# Ganzzahldivision
10 // 3     # ergibt 3

# Modulo (Rest)
10 % 3      # ergibt 1

# Potenz
2 ** 3      # ergibt 8
```

String-Operationen

```
# Strings verknüpfen
"Hallo" + " " + "Welt"  # "Hallo Welt"

# String wiederholen
"Ha" * 3                # "HaHaHa"

# Länge eines Strings
len("Python")           # 6

# Zeichen an Position (Index)
"Python"[0]             # "P"
"Python"[1]             # "y"
"Python"[-1]            # "n"

# Teilstring
"Python"[0:2]           # "Py"
"Python"[2:]            # "thon"
```

Einfache Schleifen mit `for`

```
# N-mal wiederholen
for _ in range(5):
    print("Hallo")  # Gibt 5x "Hallo" aus

# Durch Zahlenbereich iterieren
for i in range(1, 6):  # 1, 2, 3, 4, 5
    print(i)
```

Variablen

Variablen deklarieren

```
# Variable erstellen und Wert zuweisen
name = "Max"
alter = 25
pi = 3.14159
ist_student = True

# Mehrere Zuweisungen
a, b, c = 1, 2, 3
```

Eingabe und Ausgabe

```
# Eingabe vom Benutzer
name = input("Gib deinen Namen ein: ")

# Umwandlung in Zahl
alter = int(input("Gib dein Alter ein: "))
gewicht = float(input("Gewicht in kg: "))

# Ausgabe mit print()
print("Hallo", name)
print("Du bist", alter, "Jahre alt")

# Formatierte Ausgabe
print(f"Hallo {name}, du bist {alter} Jahre alt")
```

Wert verändern

```
# Neuen Wert zuweisen
zahl = 10
zahl = 20  # zahl ist jetzt 20

# Wert erhöhen/verringern
zahl = zahl + 5  # zahl ist jetzt 25
zahl += 5        # zahl ist jetzt 30
zahl -= 10       # zahl ist jetzt 20
zahl *= 2        # zahl ist jetzt 40
zahl //= 4       # zahl ist jetzt 10
```

Funktionen

Funktionen definieren

```
# Einfache Funktion ohne Parameter
def begrüssung():
    print("Hallo!")

# Funktion mit Parametern
def begrüsse_person(name):
    print(f"Hallo {name}!")

# Funktion mit Rückgabewert
def quadrat(zahl):
    return zahl * zahl

# Funktion mit mehreren Parametern
def rechteck_fläche(länge, breite)
    :
    return länge * breite

# Funktion mit Standardwert
def potenz(basis, exponent=2):
    return basis ** exponent
```

Funktionen aufrufen

```
# Funktion aufrufen
begrüssung()                #
    Hallo!

# Mit Parameter
begrüsse_person("Lea")      #
    Hallo Lea!

# Rückgabewert verwenden
ergebnis = quadrat(5)       #
    ergebnis = 25
print(ergebnis)

# Mehrere Parameter
fläche = rechteck_fläche(4, 5)  #
    20

# Mit Standardwert
potenz(3)       # 9 (3²)
potenz(2, 3)    # 8 (2³)
```

Verzweigungen und Bedingungen

Vergleichsoperatoren

```
# Gleichheit
a == b  # Ist a gleich b?

# Ungleichheit
a != b  # Ist a ungleich b?

# Grösser/Kleiner
a > b   # Ist a grösser als b?
a < b   # Ist a kleiner als b?
a >= b  # Ist a grösser oder
    gleich b?
a <= b  # Ist a kleiner oder
    gleich b?
```

if, elif, else

```
# Einfaches if
if alter >= 18:
    print("Volljährig")

# if-else
if punkte >= 50:
    print("Bestanden")
else:
    print("Nicht bestanden")

# if-elif-else
if note == 6:
    print("Sehr gut")
elif note >= 5:
    print("Gut")
elif note >= 4:
    print("Genügend")
else:
    print("Ungenügend")
```

Logische Operatoren

```
# UND: Beide Bedingungen müssen
    wahr sein
if alter >= 18 and
    hat_führerschein:
    print("Darf Auto fahren")

# ODER: Mindestens eine der
    Bedingungen muss wahr sein
if hat_mitgliedskarte or ist_gast:
    print("Zutritt erlaubt")

# NICHT: Negiert die Bedingung
if not ist_gesperrt:
    print("Zugriff möglich")
```

while-Schleife

```
# Zähler mit while
zähler = 0
while zähler < 5:
    print(zähler)
    zähler += 1

# Abbruch mit break
while True:
    eingabe = input("Weiter? (j/n)
    : ")
    if eingabe == "n":
        break
```

Listen

Listen erstellen

```
# Leere Liste
meine_liste = []

# Liste mit Werten
zahlen = [1, 2, 3, 4, 5]
namen = ["Anna", "Ben", "Carla"]
gemischt = [1, "Hallo", True,
    3.14]
```

Auf Listen zugreifen

```
# Element an Position
zahlen = [10, 20, 30, 40, 50]
zahlen[0]    # 10 (erstes Element)
zahlen[2]    # 30 (drittes Element
    )
zahlen[-1]   # 50 (letztes Element
    )

# Teilbereich
zahlen[1:3]  # [20, 30]
zahlen[:2]   # [10, 20]
zahlen[3:]   # [40, 50]

# Element ändern
zahlen[0] = 15  # [15, 20, 30, 40,
     50]
```

Listen verarbeiten

```
# Länge einer Liste
len(zahlen)  # 5

# Durch Liste iterieren
for zahl in zahlen:
    print(zahl)

# Mit Index iterieren
for i in range(len(zahlen)):
    print(f"Position {i}: {zahlen[
    i]}")

# Listen-Methoden
zahlen.append(60)    #
    [15,20,30,40,50,60]

# Wert 25 an Position 1 einfügen
zahlen.insert(1, 25) #
    [15,25,20,30,40,50,60]
letzter = zahlen.pop() # entfernt
    60
zahlen.remove(30)    # entfernt
    ersten Wert 30
```

Algorithmen mit Listen

```
# Maximum finden
def finde_maximum(liste):
    maximum = liste[0]
    for zahl in liste:
        if zahl > maximum:
            maximum = zahl
    return maximum

# Summe berechnen
def summe_berechnen(liste):
    summe = 0
    for zahl in liste:
        summe += zahl
    return summe
```

Dictionaries

Dictionary erstellen

```
# Leeres Dictionary
mein_dict = {}

# Dictionary mit Werten
person = {
    "name": "Anna",
    "alter": 25,
    "stadt": "Zürich"
}

# Verschachtelte Dictionaries
schüler = {
    "max": {
        "alter": 16,
        "note": 5.5
    },
    "lisa": {
        "alter": 17,
        "note": 6.0
    }
}
```

Auf Dictionary zugreifen

```
# Wert abfragen
person["name"]      # "Anna"
person.get("name")  # "Anna"

# Wert ändern
person["alter"] = 26

# neuen Schlüssel hinzufügen
person["beruf"] = "Informatikerin"

# Schlüssel entfernen
del person["stadt"]

# prüfen, ob Schlüssel existiert
if "name" in person:
    print(person["name"])
```

Dictionary durchlaufen

```
# alle Schlüssel durchlaufen
for schluessel in person:
    print(schluessel + ": " + str(
    person[schluessel]))

# Schlüssel und Werte
for schluessel, wert in person.
    items():
    print(f"{schluessel}: {wert}")

# nur Werte
for wert in person.values():
    print(wert)

# nur Schlüssel
for schluessel in person.keys():
    print(schluessel)
```

Mengen (Sets)

Mengen (Sets)

```
liste_a = ["Anna", "Ben", "Clara",
    "Anna"]
liste_b = ["Ben", "David", "Eva"]

# Listen zu Mengen umwandeln
set_a = set(liste_a)
set_b = set(liste_b)

# Mengenoperationen
# Elemente, die in mindestens in
    einer der beiden Mengen sind:
vereinigung = set_a | set_b

# Elemente, welche sowohl in set_a
    als auch in set_b sind:
schnittmenge = set_a & set_b

# Elemente, welche zwar in set_a
    aber nicht auch in set_b sind:
differenz = set_a - set_b

print("Vereinigung:", vereinigung)
print("Schnittmenge:",
    schnittmenge)
print("Differenz:", differenz)
print("Anzahl der Elemente in
    set_a:", len(set_a))
```

Objektorientierte Programmierung

Klassen definieren

```
class Person:
    # Konstruktor
    def __init__(self, name, alter
    ):
        self.name = name
        self.alter = alter

    # Methode
    def vorstellen(self):
        print(f"Ich bin {self.name
    }, {self.alter} Jahre alt.")

    # Methode mit Rückgabewert
    def ist_volljährig(self):
        return self.alter >= 18
```

Objekte erstellen und verwenden

```
# Objekt erstellen
bob = Person("Bob", 17)
anna = Person("Anna", 25)

# Methoden aufrufen
bob.vorstellen()  # Ich bin Bob,
    17 Jahre alt.
anna.vorstellen() # Ich bin Anna,
    25 Jahre alt.

# Attribute verwenden
print(bob.name)   # Bob
bob.alter = 18    # Alter ändern

# Methode mit Rückgabewert
if anna.ist_volljährig():
    print("Anna ist volljährig")
```

Klassenattribute

```
class Schüler:
    # Klassenattribut (für alle
    Instanzen gleich)
    schule = "Kantonsschule im Lee
    "
    anzahl = 0

    def __init__(self, name,
    klasse):
        self.name = name
        self.klasse = klasse
        Schüler.anzahl += 1

    # Klassenmethode
    @classmethod
    def get_anzahl(cls):
        return cls.anzahl
```

Vererbung

```
class Fahrzeug:
    def __init__(self, marke,
    modell):
        self.marke = marke
        self.modell = modell

    def info(self):
        return f"{self.marke} {
    self.modell}"

class Auto(Fahrzeug):
    def __init__(self, marke,
    modell, türen):
        super().__init__(marke,
    modell)
        self.türen = türen

    def info(self):
        basis_info = super().info
    ()
        return f"{basis_info} mit
    {self.türen} Türen"
```

Tabellenverzeichnis

Abbildungsverzeichnis

Literatur

[1] Joseph J. Rotman Albert Cuoco. *Learning Modern Algebra: From Early Attempts to Prove Fermat's Last Theorem*. English. 08. January 2015. Cambridge University Press, 2015. ISBN: 978-1939512017.

Glossar

BMR Base Metabolic Rate. 100–102

FPS Frames Per Second. 107

IDE Integrated Development Environment. 2

NEAT Non-Exercise Activity Thermogenesis. 103

OOP Objektorientierte Programmierung. 87