Unterlagen für das obligatorische Fach Informatik

Gymnasium – Einführung in die Programmierung mit Python

Jacques Mock Schindler

1. Februar 2026

Inhaltsverzeichnis

Welcome

Here you will find information about computer science lessons.

Schedule

Date	Topic
16/02/2026	Data Representation I: Number Systems (Binary, Decimal, Hexadecimal)
02/03/2026	Data Representation II: Character Encoding (ASCII, Unicode)
09/03/2026	Data Representation III: Images (RGB, Pixels, Resolution)
16/03/2026	Concept of Algorithms: Definition, Properties, Notation
23/03/2026	Algorithm Representation: Pseudocode and Nassi-Shneiderman Diagrams
30/03/2026	Flowcharts and State Diagrams
13/04/2026	TEST 1: Data Representation & Algorithm Representation
04/05/2026	Functions I: Revision and Consolidation (Parameters, Return)
11/05/2026	Functions II: Scope and Namespaces (local vs. global)
18/05/2026	Functions III: Modularisation and Code Organisation
01/06/2026	Functions IV: Advanced Recursive Functions
08/06/2026	Exam Preparation and Subroutine Exercises
15/06/2026	TEST 2: Subroutines (Functions, Scope, Modularisation)
22/06/2026	Mini-Project I: Developing a Function-Based Application
29/06/2026	Mini-Project II: Completion and Presentation
06/07/2026	Reflection, Outlook for 3rd Semester, Feedback Session

The programme reflects the current state of planning. Changes are to be expected throughout the semester.

Hardware and Software Requirements

A laptop is required for the computer science lessons (you will not be able to solve the tasks set in class using an iPad). You will need administrator rights on your laptop to install the required software.

Furthermore, you must ensure that your battery is sufficiently charged at the start of the lesson to last for a double period.

Assessment

Two written tests are planned per semester. In addition, oral participation will be graded. Asking questions is specifically considered as part of oral participation.

The final grade is calculated as a weighted average of the two written tests and the grade for oral participation. The average grade of the two written tests is weighted at 90%, while the grade for oral participation is weighted at 10%.

If anyone would like a personal consultation, you can sign up for a meeting here (Rent a Mock).

Teil I.

Anleitungen

1. Speicherorganisation

Informationen sind in Computern in Dateien gespeichert. Die gespeicherten Informationen können dabei ganz unterschiedlicher Art, wie zum Beispiel Texte, Bilder oder Videos, sein. Als Modell, wie man sich Dateien vorstellen kann, hat sich das Bild von Dossiers in Ordnern etabliert. Die einzelnen Dossiers sind die Dateien und die Ordner sind die Strukturen, in denen die Dateien abgelegt sind. Diese Konstruktion kann über mehrere Ebenen hinaus verschachtelt werde (Ordner in Gestellen, die wiederum in einzelnen Räumen stehen, etc.).

1.1. Dateinamen und Pfade

Damit Dateien identifiziert und gefunden werden können, müssen sie einen Namen haben. Grundsätzlich gibt es keine Einschränkungen, wie Dateien benannt werden. Die meisten Dateinamen bestehen allerdings aus zwei Teilen: dem eigentlichen Dateinamen und der Dateinamenserweiterung.

Der eigentliche Dateiname wird idealerweise so festgelegt, dass er einen Rückschluss auf den Inhalt der Datei zulässt.
Die Dateinamenserweiterung ist ein Zusatz, der Auskunft über den Dateityp gibt. Sie steht hinter einem Punkt hinter dem eigentlichen Dateinamen. Auf Windows Rechnern wird die Dateinamenserweiterung im Dateimanager in der Standardeinstellung nicht angezeigt. Um dies zu ändern, muss in den Einstellungen des Dateimanagers die Option “Dateinamenerweiterungen anzeigen” aktiviert werden (Ansicht > Anzeigen > Dateinamenserweiterung).

Damit Dateien besser ausgetauscht werden können, empfiehlt es sich, für die Namen lediglich Buchstaben, Zahlen und Unterstriche (sog. ASCII-Zeichen) zu verwenden.

Damit man Dateien finden kann, muss man wissen, wo sie abgelegt worden sind. Übertragen auf das Modell von Dossiers in Ordnern bedeutet das, zu wissen, welches Dossier in welchem Ordner in welchem Gestell in welchem Raum abgelegt ist. Wie in einem realen Archiv, geht man dabei vom Raum zum Gestell, zum Ordner und schliesslich zum Dossier. In der Informatik wird dieser Weg als Pfad bezeichnet. Auf einem Windows-Rechner beginnt dieser Pfad mit dem sogenannten Laufwerksbuchstaben, gefolgt von einem Doppelpunkt und einem Backslash (\). Aus historischen Gründen ist der Laufwerksbuchstabe auf Windows-Rechnern Standardmässig der Buchstabe `C`.

Ein Beispiel für einen Pfad könnte so aussehen:

```
1 C:\Users\fritz\Documents\text.docx
```

In diesem Beispiel ist `C:` der Laufwerksbuchstabe, `Users` der Ordner, `fritz` der Unterordner, `Documents` der Unterordner von `fritz` und `text.docx` die Datei, die im Ordner `Documents` abgelegt ist. `Users` ist ein von Windows standardmässig angelegter Ordner, in dem die persönlichen Daten der Benutzer abgelegt werden. Der Ordner `fritz` ist der persönliche Ordner des Benutzers `fritz`. Der Ordner `Documents` wird ebenfalls standardmässig von Windows im Ordner jedes Benutzers angelegt. Dem Benutzer `fritz` steht es frei, diesen Ordner zu verwenden und darin Dateien oder Underordner anzulegen.
Der Dateiname `text.docx` verweist mit seiner Dateinamenserweiterung `.docx` auf eine Datei, die mit dem Programm Microsoft Word erstellt worden ist.

1.2. Dateien in der Cloud

Dateien können nicht nur lokal auf dem Computer gespeichert werden. Damit von überall und jederzeit auf Dateien zugegriffen werden kann, werden Dateien in der *Cloud* gespeichert. Dabei handelt es sich um Server in Rechenzentren, die über das Internet erreichbar sind. Beispiele für solche Cloud-Dienst sind OneDrive von Microsoft oder Google Drive.
Ablageorte auf OneDrive werden in Windows direkt in der Verzeichnisstruktur des Betriebssystems angezeigt, solche von Google erhalten auf Windows einen eigenen Laufwerksbuchstaben (`G:\`).

Damit auf die Dateien in der Cloud zugegriffen werden kann, ist eine Internetverbindung erforderlich.

1.3. Dateistruktur für die Schule

Für den schulischen Bedarf erscheint es sinnvoll eine Dateistruktur nach Fächern anzulegen. Im Ordner `Documents` wird dazu für jedes Fach ein eigener Unterordner angelegt. Innerhalb der jeweiligen Fachordner kann eine weitere Struktur nach Semester oder nach Thema sinnvoll sein. Ein Beispiel für die Ordnerstruktur eines Erstklässlers an der KBW kann so aussehen:

```
Documents\
          ├──Schule\
          |       ├──Deutsch
          |       ├──Franz
          |       ├──Mathe
          |       ├──WR
          |       └──...
          ├──Privat\
          |       ├──Rechnungen
          |       └──...
```

2. Arbeitsumgebung (Arbeiten mit Jupyter Notebooks)

2.1. Ausgangslage

In der Informatik geht es darum, wie Informationsverarbeitung mit Hilfe von Computern automatisiert werden kann.
Die Automatisierung der Informationsverarbeitung erfordert die Verwendung von Programmiersprachen. Im Informatikunterricht wird in erster Linie mit der Programmiersprache Python gearbeitet.
Im folgenden findet sich eine Anleitung für die Installation der für den Unterricht erforderlichen Programme.

2.2. Installation von Python

Dieser Abschnitt führt Sie Schritt für Schritt durch die Installation von Python auf einem Windows-Rechner.

> ⚠ **Microsoft Store Falle**
>
> Achten Sie beim Herunterladen von Python darauf, dass Sie sich auf der offiziellen Seite von Python (https://www.python.org) und **nicht** im Microsoft Store befinden. Wenn Sie versehentlich die Python Version aus dem Microsoft Store installiert haben, kann das bei der Arbeit an den Schulprojekten zu Problemen führen.
> Deinstallieren Sie die Microsoft Version von Python und installieren Sie die Version von der offiziellen Website.

1. Laden Sie die neueste Version von Python von der offiziellen Website herunter: python.org.
2. Führen Sie das heruntergeladene Installationsprogramm durch Doppelklick auf die Datei aus. Stellen Sie sicher, dass Sie die Option “Add Python to PATH” aktivieren, bevor Sie auf “Install Now” klicken.

> **Die 'PATH'-Umgebungsvariable**
>
> Stellen Sie sich die PATH-Variable wie ein Adressbuch für die Kommandozeile (Terminal) vor. Wenn Sie einen Befehl wie `python` eingeben, schaut der Computer in diesem Adressbuch nach, wo das entsprechende Programm zu finden ist.
> Indem Sie das Häkchen bei "Add Python to PATH" setzen, fügen Sie die Adresse des Python-Interpreters zu diesem Adressbuch hinzu. Ohne diesen Eintrag weiss der Computer nicht, wo er suchen soll, und meldet, dass er den Befehl nicht kennt.

3. Überprüfen Sie die Installation, indem Sie die Eingabeaufforderung öffnen (Terminal → Windows-Taste + R, dann `cmd` eingeben) und den Befehl `python --version` eingeben. Dies sollte die installierte Python-Version anzeigen.

2.3. Hello World

Es hat sich eingebürgert, dass das erste Programm, das ausgeführt wird, ein Programm ist, das den Text "Hello World" auf dem Bildschirm ausgibt. Um dieser Tradition zu folgen, führen Sie die folgenden Schritte aus:

1. Öffnen Sie ein Terminal (Windows-Taste + R, dann `cmd` eingeben).

> **Das Terminal**
>
> Unter dem Begriff "Terminal" versteht man ein Programm, das eine textbasierte Benutzeroberfläche bereitstellt, um mit dem Betriebssystem zu interagieren. In einem Terminal können Sie Befehle eingeben und erhalten die Ausgaben direkt im Fenster.

2. Geben Sie den Befehl `python` ein, um die Python-Shell zu starten. Die Python Shell sollte ungefähr so, wie das folgende Bild aussehen.

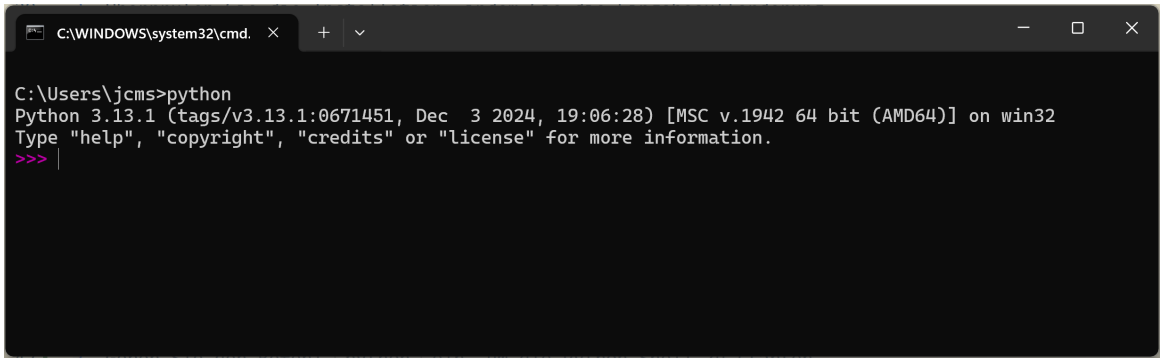


Abbildung 2.1.: Python Shell

3. Geben Sie den folgenden Befehl ein und drücken Sie anschliessend die Eingabetaste:

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

Das Resultat sollte wie das folgende Bild aussehen.

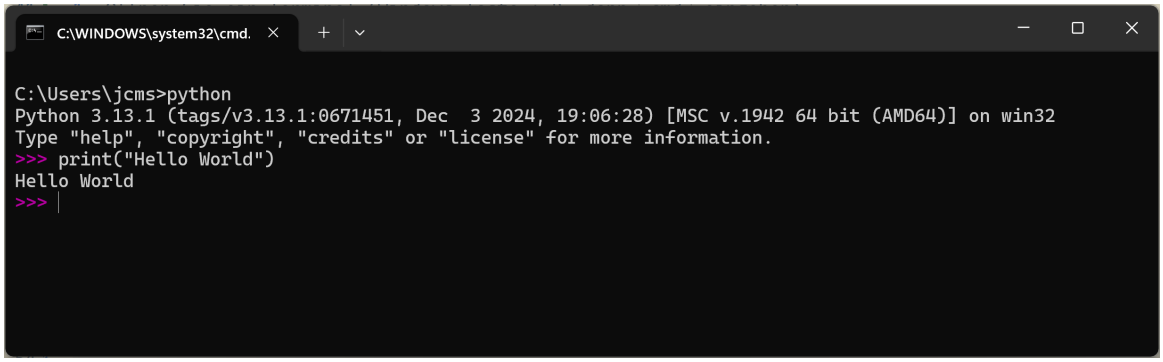


Abbildung 2.2.: Python Shell

Gratuliere - Sie haben Ihr erstes Python-Programm erfolgreich ausgeführt!

2.4. Arbeitsumgebung

Im Unterricht wird nicht direkt in der Python-Shell gearbeitet, sondern mit sogenannten Jupyter Notebooks. Jupyter Notebooks ermöglichen es, in der gleichen Datei sowohl Code (Programm Teile) als auch formatierten Text (in Markdown) zu verarbeiten. Eine Jupyter Notebook Datei hat die Endung `.ipynb`. Vom Jupyter Notebook unterschieden werden muss die Arbeitsoberfläche in welcher die Jupyter Notebooks bearbeitet werden. Diese Oberfläche nennt sich JupyterLab und läuft in einem Webbrowser.

> **Das Jupyter Ökosystem**
>
> Die im Unterricht verwendeten Jupyter Notebooks sind Teil eines ganzen Jupyter Ökosystems. Der Name Jupyter setzt sich aus den drei Programmiersprachen **Ju**lia, **Pyt**hon und **R** zusammen, die in diesem Ökosystem eine zentrale Rolle spielen. Zum Jupyter Ökosystem gehören auch zahlreiche Erweiterungen und Tools, die die Arbeit mit Notebooks und Daten erleichtern.
> Der Unterricht beschränkt sich auf die Verwendung von Jupyter Notebooks mit der Programmiersprache Python sowie den Einsatz von JupyterLab als Arbeitsumgebung.

Damit dies alles funktioniert, braucht es ein paar weitere Vorbereitungsarbeiten.

1. Erstellen Sie im Ordner “Informatik” einen Unterordner mit dem Heutigen Datum als Namen. Formatieren Sie das Datum nach dem Schema “YYMMDD”, für den 1. August 2025 wäre das zum Beispiel “250801”.
2. Öffnen Sie den soeben erstellten Ordner.
3. Geben Sie die Tastenfolge `Ctrl + L` ein, um die Adresszeile des Dateimanagers zu aktivieren.
4. Überschreiben Sie den Inhalt der Adresszeile mit dem Text `cmd` und drücken Sie die Eingabetaste. Dadurch wird ein Terminal geöffnet, das direkt im aktuellen Ordner arbeitet.
5. Geben Sie im neu geöffneten Terminal den folgenden Befehl ein und drücken Sie die Eingabetaste:

   ```
   python -m venv .venv
   ```

   Dadurch wird eine sogenannte Python Virtual Environment erstellt (venv wegen **V**irtual **Env**ironment). Dieses Python Virtual Environment dient dazu, die eigenen Programmierprojekte unabhängig voneinander gestalten zu können.
6. Aktivieren Sie das Python Virtual Environment mit dem folgenden Befehl:

   ```
   .venv\Scripts\activate
   ```

   Ihr Terminal sieht nach dem Erstellen und Aktivieren der Python Virtual Environment ungefähr so aus:

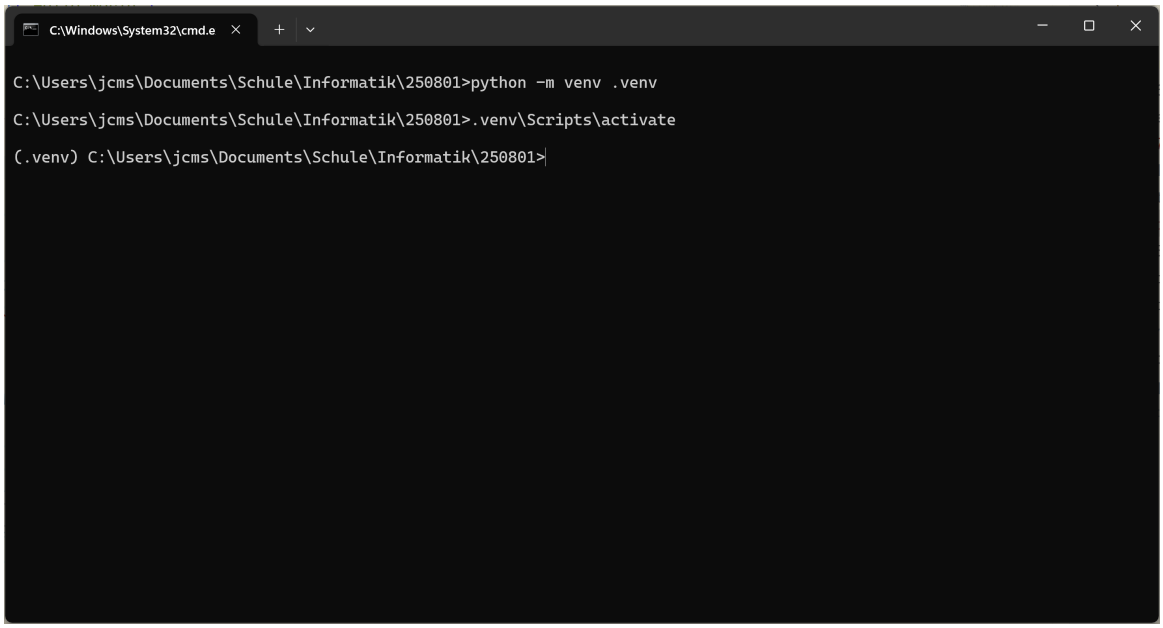


Abbildung 2.3.: Aktivierte Python Virtual Environment

Das Wort in der Klammer am Anfang der Zeile zeigt den Namen der aktiven Python Virtual Environment an. Im vorliegenden Fall ist das .venv.

7. In der nun aktivierten Python Virtual Environment installieren Sie die benötigten Pakete mit dem folgenden Befehl:

```
pip install jupyter
```

Das dauert eine Weile.

Während der Installation werden die benötigten Pakete (Ergänzungen zur bestehenden Python Installation) heruntergeladen und in der Python Virtual Environment gespeichert. Das Terminal sieht dabei ungefähr so aus:

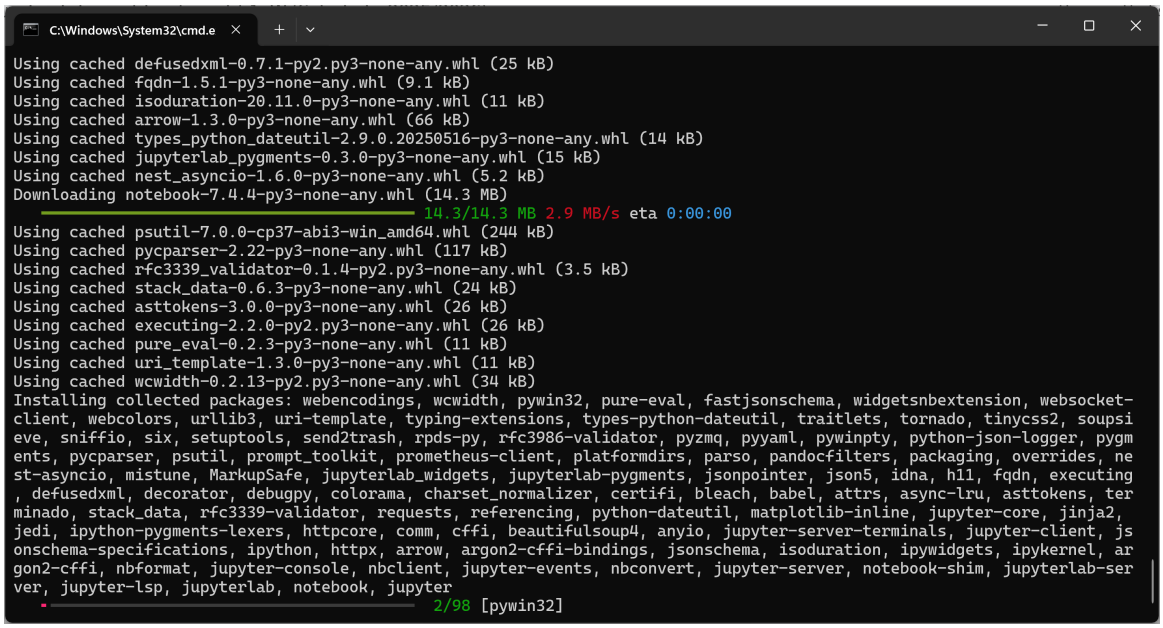


Abbildung 2.4.: Terminal während der Jupyter Installation

Alle in einer Python Virtual Environment installierten Pakete sind innerhalb dieser Umgebung dauerhaft verfügbar und müssen daher für das gleiche Projekt kein zweites Mal installiert werden.

8. Starten Sie den Jupyter Server mit dem folgenden Befehl:

```
jupyter-lab
```

Dies startet den Jupyter Notebook Server und öffnet automatisch ein Browserfenster mit der Jupyter Notebook Oberfläche.

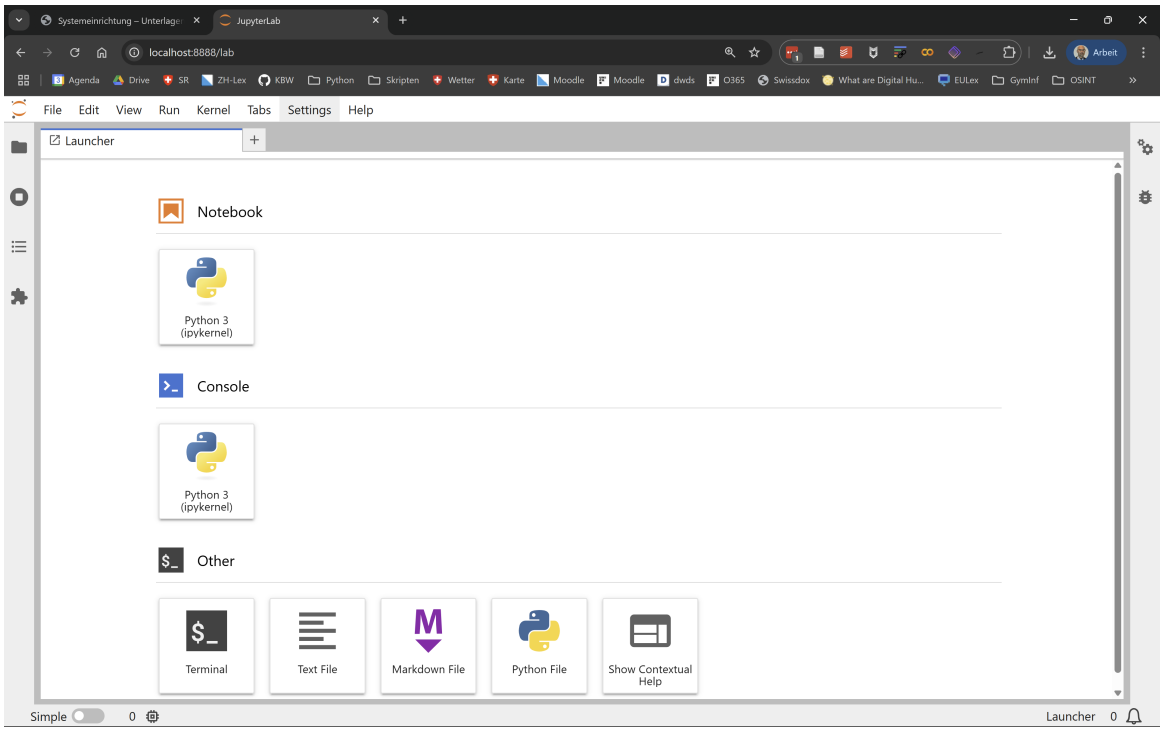


Abbildung 2.5.: Startseite Jupyter Lab

9. Klicken Sie auf den Button “Python 3 (ipykernel)” unter dem Titel Notebook.

   Damit starten Sie ein neues Jupyter Notebook. Der Cursor blinkt in einer leeren Zelle. Bei dieser Zelle handelt es sich um eine sogenannte Code-Zelle. In einer Code-Zelle können Sie Python Code eingeben und ausführen.
   Überprüfen Sie das, indem Sie in der Zelle den Befehl `print("Hello World")` eingeben und anschliessend die Tastenfolge `Shift + Enter` drücken (alternativ können Sie auch auf den Button “Run” in der Werkzeugleiste klicken).

   Das Resultat sollte wie das folgende Bild aussehen.

Abbildung 2.6.: Hello World in einem Jupyter Notebook

In einem Jupyter Notebook können Sie nicht nur Python Code ausführen, sondern auch Text (formatiert in Markdown) darstellen.
Für die Darstellung von Text müssen Sie die Zelle als Text-Zelle markieren. Dazu klicken Sie auf den Button “Code” in der Werkzeugleiste und wählen im Dropdown-Menü die Option “Markdown” aus.

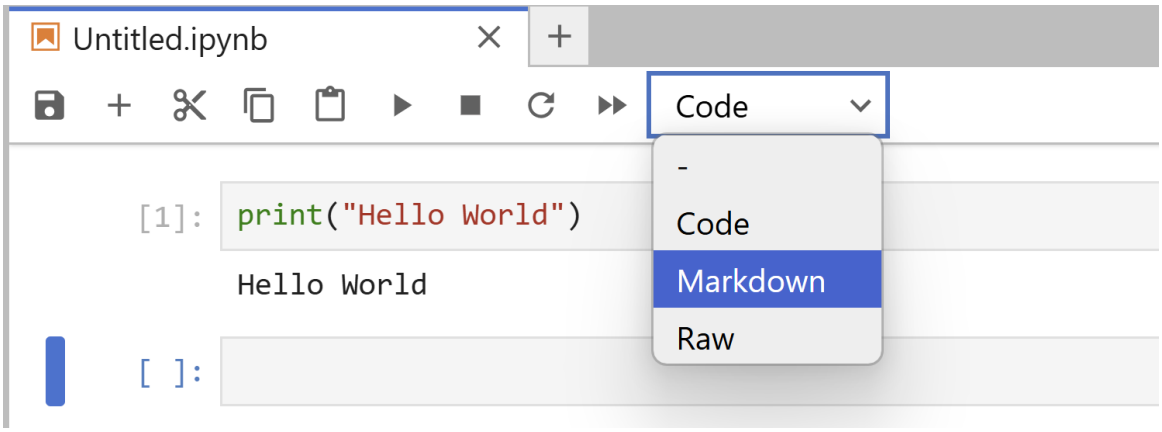


Abbildung 2.7.: Umstellen der Zelle auf Markdown

Probieren Sie das aus. Schreiben Sie einen Titel und einen kurzen Text in die Zelle unterhalb der Code-Zelle mit `pringt("Hello World")`. Damit der Text in der Zelle formatiert angezeigt wird, müssen Sie die Zelle mit der Tastenfolge `Shift + Enter` ausführen (analog zum Ausführen von Code-Zellen).

Eine Zelle ist entweder eine Code-Zelle oder eine Text-Zelle. Für den Wechsel zwischen Code- und Text-Darstellung müssen Sie je eine neue Zelle anlegen. Das geht mit der Tastenfolge `Esc + B` (für "Below") oder `Esc + A` (für "Above") während Sie sich in einer Zelle befinden. Alternativ können Sie auch die Buttons "Insert Cell Below" oder "Insert Cell Above" aus den Werkzeugen der Zelle verwenden.

10. Das Jupyter Notebook ist eine Datei mit der Endung `.ipynb`. Neu erstellte Jupyter Notebooks erhalten den Namen "Untitled.ipynb". Um diesen Namen zu ändern, klicken Sie mit der rechten Maustatste auf den Titel "Untitled" in der oberen linken Ecke des Jupyter Notebooks und wählen Sie die Option "Rename" aus dem Kontextmenü. Anschliessend können Sie den Namen des Jupyter Notebooks eingeben.

2.5. Öffnen bestehender Jupyter Notebooks

Häufiger als das Erstellen eines neuen Jupyter Notebooks ist das Öffnen eines bereits bestehenden Jupyter Notebooks. Hier wird das entsprechende Vorgehen beschrieben.

1. Navigieren Sie in den Ordner in dem sich das Jupyter Notebook befindet.
2. Stellen Sie sicher, dass der Ordner über eine Python Virtual Environment mit installierten Jupyter Paketen verfügt.

   Öffnen Sie dazu im ausgewählten Ordner das Terminal (`ctrl + L` anschliessend `cmd` und Eingabetaste). Dann starten Sie die Python Virtual Environment und geben den Befehl `pip list` ein. Dieser Listet alle in der Python Virtual Environment installierten Pakete auf. Falls die Jupyter Pakete nicht aufgelistet werden, müssen Sie diese wie oben beschrieben installieren.

3. Starten Sie den Jupyter Server mit dem Befehl `jupyter-lab`.
4. Öffnen Sie das Dateiverzeichnis. Dazu müssen Sie auf dem linken Rand das Ordner-Symbol anklicken.
5. Wählen Sie das Jupyter Notebook aus, das Sie öffnen möchten. Mit einem Doppelklick auf das Jupyter Notebook wird dieses geöffnet.

   Wenn Sie das Dateiverzeichnis wieder schliessen möchten, klicken Sie auf das Ordner-Symbol auf der linken Seite erneut.

Diese Schritte funktionieren auch, wenn Sie ein Jupyter Notebook öffnen möchten, das Sie von jemand anderem erhalten haben. Sie müssen dieses dazu lediglich in den Ordner kopieren, in dem sich die Python Virtual Environment mit den installierten Jupyter Paketen befindet.

2.6. Häufige Fehlermeldungen

Problem / Fehlermeldung (Was Sie sehen)	Mögliche Ursache (Warum es passiert)	Lösung (Was Sie tun können)
Der Befehl `python` ist entweder falsch geschrieben oder konnte nicht gefunden werden.	Python wurde bei der Installation nicht zur PATH-Variable hinzugefügt. Der Computer weiss nicht, wo er `python.exe` finden soll.	1. Deinstallieren Sie Python über die Systemsteuerung. 2. Installieren Sie Python erneut. 3. Achten Sie diesmal unbedingt darauf, das Häkchen bei "Add Python to PATH" zu setzen.
Der Befehl `jupyter-lab` ist entweder falsch geschrieben oder konnte nicht gefunden werden.	Sie haben vergessen, die virtuelle Umgebung zu aktivieren. Der Befehl `jupyter-lab` existiert nur innerhalb der aktivierten Umgebung.	1. Überprüfen Sie, ob `(.venv)` am Anfang der Kommandozeile steht. 2. Falls nicht, führen Sie den Aktivierungsbefehl erneut aus: `.venv\Scripts\activate`.

Problem / Fehlermeldung (Was Sie sehen)	Mögliche Ursache (Warum es passiert)	Lösung (Was Sie tun können)
`ImportError: DLL load failed...` oder ähnliche Fehler unter Windows	Ein häufiges Problem mit der Installation von `pywin32`, einer wichtigen Windows-Bibliothek, die von Jupyter benötigt wird.	1. Stellen Sie sicher, dass Ihre virtuelle Umgebung aktiv ist. 2. Führen Sie den Befehl `pip install --upgrade pywin32` aus, um die Bibliothek zu reparieren.
Kernel Error oder der Status "Kernel starting, please wait..." ändert sich nicht	Die Verbindung zwischen der Browser-Oberfläche und dem Python-"Gehirn" (dem Kernel) ist gestört. Dies kann viele Ursachen haben.	1. Der einfachste erste Schritt: Klicken Sie im JupyterLab-Menü auf "Kernel" -> "Restart Kernel...". 2. Wenn das nicht hilft, schliessen Sie JupyterLab im Terminal (mit der Tastenkombination Strg + C) und starten Sie es mit `jupyter-lab` neu.
`Permission denied` (Zugriff verweigert) bei der Installation von Paketen	Sie versuchen, Pakete an einem systemweiten Ort zu installieren (z. B. in `C:\Program Files`), für den Sie keine Schreibrechte haben.	Dies ist genau das Problem, das virtuelle Umgebungen lösen! Stellen Sie sicher, dass Ihre `venv` aktiv ist (`(.venv)` muss sichtbar sein). Dadurch wird sichergestellt, dass alle Pakete lokal in Ihren Projektordner installiert werden, wo Sie die vollen Rechte haben.

Teil II.

Einführung in Python

3. Problemlösung in der Informatik

3.1. Ausgangslage

Jede Disziplin hat ihre eigene Art, Probleme zu lösen. Das ist in der Informatik nicht anders. In der Informatik versucht man, grosse Probleme in kleinere Teilprobleme zu zerlegen. Das macht man so lange, bis die Teilprobleme so klein sind, dass sie einfach zu lösen sind.

Dies soll hier anhand von verschidenen Grafiken gezigt werden.

Damit in Python einfach mit Grafiken gearbeitet werden kann, wird das Paket `PyTamaro` verwendet. Dieses Paket wurde von der Università della Svizzera italiana (USI) extra für die Informatik-Ausbildung entwickelt.

Damit das Paket verwendet werden kann, muss es zuerst in der aktuellen Python Virtual Environment installiert werden.
Dazu öffnen Sie ein Terminal im Ordner, in dem sich dieses Jupyter Notebook befindet. Anschliessend starten Sie die Python Virtual Environment mit dem Befehl:

```
.venv\Scripts\activate
```

Danach können Sie `PyTamaro` mit dem Befehl

```
pip install pytamaro
```

installieren.

Um das Paket zu verwenden, muss es *importiert* werden. Die genauen Zusammenhänge müssen im Moment nicht bekannt sein. Wichtig ist lediglich, dass die folgende Zelle ausgeführt wird.

```
from pytamaro.de import (
    rechteck, kreis_sektor,
    blau, rot, weiss, schwarz,
    neben, ueber, ueberlagere,
    drehe, kombiniere,
    zeige_grafik, speichere_grafik,
)
```

3.2. Beispiel: Tricolore

Die Vorgehensweise wird anhand der Französischen Nationalflagge (Tricolore) gezigt.

Um die Zeichnung der Tricolore zu planen, wird die Grafik in ihre Einzelteile zerlegt.
Die Tricolore besteht aus drei gleich grossen Rechtecken in den Farben blau, rot und weiss.
Diese Rechtecke werden nebeneinander angeordnet.

Das bedeutet, dass die Länge und die Breite der Rechtecke definiert werden muss und basierend auf diesen Werten die drei Rechtecke gezeichnet werden. Anschliessend werden die drei Rechtecke nebeneinander angeordnet.

Der Befehl zum Zeichnen eines Rechtecks lautet

```
name = rechteck(länge, breite, farbe)
```

Bevor die Zeichnung tatsächlich erstellt wird, soll hier der Befehl im Detail erklärt werden:

- `name` ist der Name, unter dem das Rechteck gespeichert wird. Dieser Name kann später verwendet werden, um auf das Rechteck zuzugreifen.
- `rechteck` ist der Befehl, der ein Rechteck zeichnet. In der Klammer hinter dem Befehl werden die sogenannten *Argumente* angegeben. Diese steuern, wie das Rechteck aussieht.
- `länge` und `breite` sind die Argumente, die die Grösse des Rechtecks bestimmen. Diese Werte können beliebig gewählt werden.
- `farbe` ist das Argument, das die Farbe des Rechtecks bestimmt. Aufgrund der Eigenheiten von `PyTamaro` können ausschliesslich die Farben verwendet werden, welche importiert worden sind.

3.2.1. Rechtecke zeichnen

Als erstes wird hier gezeigt, wie das blaue Rechteck gezeichnetwird.

Damit das Resultat kontrolliert werden kann, wird die Grafik mit dem Befehl

```
zeige_grafik(name)
```

angezeigt.

```
bleu = rechteck(50, 100, blau)
zeige_grafik(bleu)
```

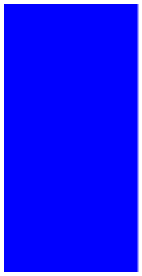
Nachdem das blaue Rechteck gezeichnet wurde, kann das weisse und das rote Rechteck analog gezeichnet und angzeigt werden.

```
# TODO: Rechtecke blanc und rouge zeichnen -> Schreiben Sie hier Ihren
# Code
blanc = rechteck(50, 100, weiss)
rouge = rechteck(50, 100, rot)
```

Als nächstes werden die drei Rechtecke nebeninander angeordnet. Dazu wird der Befehl `neben` verwendet. Dieser Befehl nimmt zwei Argumente entgegen: das erste Rechteck und das zweite Rechteck. Das erste Rechteck wird links vom zweiten Rechteck gezeichnet.

```
resultat = neben(linke grafik, rechte grafik)
```

```
zwei_drittel = neben(bleu, blanc)
zeige_grafik(zwei_drittel)
```



Analog können Sie nun das rote Rechteck rechts der zwei Drittel anordnen. Nennen Sie das Resultat `tricolore` und zeigen Sie es an.

```
# TODO: Trcolore zusammenfügen -> Schreiben Sie hier Ihren
# Code
```

3.3. Beispiel: Österreichische Flagge

Zeichnen Sie die Österreichische Flagge. Das Seitenverhältnis der Flagge ist 2:3.

Um Elemente übereinander anzuordnen, wird der Befehl `ueber` verwendet. Die Syntax dieses Befehls lautet:

```
resultat = ueber(obere grafik, untere grafik)
```

```
# TODO: Östereichische Flagge zeichnen -> Schreiben Sie hier Ihren
# Code
```

3.4. Beispiel: Schweizerfahne

Zeichnen Sie eine korrekt propotionierte Schweizerfahne. Die Dimensionen können Sie der folgenden Grafik entnehmen:

Verwenden Sie dazu die Befehle `rechteck`, `drehe` und `ueberlagere`.

```
# TODO: Schweizerfahne zeichnen -> Schreiben Sie hier Ihren
# Code
```

3.5. Beispiel: Tessiner Wappen

Als Referenz an die USI zeichnen Sie als letztes Beispiel das Tessiner Wappen.

Verwenden Sie dazu neben den bereits bekannten Befehlen zusätzlich die Befehle `ueber` und `kreis_sektor`.

```
# TODO: Tessiner Wappen zeichnen -> Schreiben Sie hier Ihren
# Code
```

3.6. Musterlösungen

3.6.1. Trikolore

```
bleu = rechteck(50, 100, blau)
blanc = rechteck(50, 100, weiss)
rouge = rechteck(50, 100, rot)

deux_tiers = neben(bleu, blanc)
tricolore = neben(deux_tiers, rouge)

zeige_grafik(tricolore)
```

3.6.2. Österreichische Flagge

```
roter_balken = rechteck(90, 20, rot)
weisser_balken = rechteck(90, 20, weiss)

oberer_teil = ueber(roter_balken, weisser_balken)
osterreichische_flagge = ueber(oberer_teil, roter_balken)

zeige_grafik(osterreichische_flagge)
```

3.6.3. Schweizerfahne

```
hintergrund = rechteck(320, 320, rot)
weisser_balken = rechteck(200, 60, weiss)
weisser_balken2 = drehe(90, weisser_balken)

kreuz = ueberlagere(weisser_balken, weisser_balken2)

schweizerfahne = ueberlagere(kreuz, hintergrund)

zeige_grafik(schweizerfahne)
```

3.6.4. Tessiner Wappen

```
roter_teil = rechteck(160, 240, rot)
blauer_teil = rechteck(160, 240, blau)
roter_sektor = kreis_sektor(160, 90, rot)
roter_sektor = drehe(180, roter_sektor)
blauer_sektor = kreis_sektor(160, 90, blau)
blauer_sektor = drehe(270, blauer_sektor)

ti_unten = neben(roter_sektor, blauer_sektor)
ti_oben = neben(roter_teil, blauer_teil)

tessin = ueber(ti_oben, ti_unten)

zeige_grafik(tessin)
```

4. Variabeln in Python

4.1. Vorbemerkungen: Python als Rechner

Python verfügt über eingebaute mathematische Fähigkeiten. Es kann die Grundrechenarten und kennt die Hierarchie der Operationen. Sie können das überprüfen, in dem Sie in der folgenden Zelle die Rechnung

$$2 + 3 \cdot 4$$

ausführen.

```
1  # hier können Sie die Rechnung ausführen
```

Die folgende Tabelle gibt einen Überblick über die direkt in Python verfügbaren mathematischen Funktionen:

Beschreibung	Befehl	Beispiel	Resultat
Addition	`+`	2 + 3	5
Subtraktion	-	3 - 2	1
Multiplikation	*	3 * 2	6
Division	`/`	3 / 2	1.5
Potenzen	**	3 ** 2	9
Wurzeln	`**(1/n)`	16 ** (1/2)	4.0
Ganzzahlige Division	`//`	7 // 2	3
Modulo	`%`	7 % 2	1

4.2. Variablen

In Python sind Variablen symbolische Namen für gespeicherte Daten. Variablen verweisen dabei auf den Speicherbereich im Computer, in welchem die entsprechenden Daten physikalisch abgelegt sind. Aus diesem Grund werden Variablen gelegentlich auch als Zeiger bezeichnet. Was genau für Daten in diesem Speicherbereich abgelegt werden, spielt keine Rolle und kann während der Ausführung eines Programmes auch ändern.

In Python werden Variablen Werte mit dem Gleichheitszeichen zugewiesen. Um der Variable x den Wert 2 zuzuweisen, ist die Eingabe `x = 2` erfoderlich. Die Variable muss links vom Gleichheitszeichen, der zuzuweisende Wert rechts davon stehen.

Überprüfen Sie dies, indem Sie in der folgenden Zelle der Variabel y den Wert 3 und der Variabel z den Wert 4 zuweisen. Anschliessend multiplizieren Sie die beiden Variablen miteinander.

```
# hier die Aufgabe einfüllen
```

Wenn Variablen neue Werte zugewiesen werden, wird die Referenz auf den Speicherbereich mit dem alten Wert gelöscht. Die Daten, welche ohne Verweis durch eine Variable im Speicher liegen, werden vom in Python eingebauten *Garbage Collector* im Hintergrund gelöscht und der so freigewordene Speicherplatz kann wieder verwendet werden.

Sie können überprüfen, dass Variablen neue Werte zugewiesen werden können, indem Sie in der untenstehenden Zelle die Variabeln y und z addieren. Sie erhalten dann das Resultat 7. Das heisst, den Variablen y und z sind immer noch die Werte 3 und 4 zugewiesen.

```
# addieren Sie hier y und z
```

Wenn Sie in der folgenden Zelle der Variabel y den Wert 5 zuweisen und anschliessend y und z addieren erhalten Sie als neues Resultat 9.

```
# weisen Sie hier y den neuen Wert zu
```

Variablen können auch Resultate von Berechnungen zugewiesen werden. Ausserdem können Variablen ganze Wörter als Namen haben. Dies ist gegenüber einzelnen Buchstaben vorzuziehen, weil dann aussagekräftige Namen gewählt werden können. Grundsätzich sind die Namen von Variabeln frei wählbar. Es gibt allerdings eine Reihe von reservierten Begriffen, welche in der Programmiersprache Python eine eigene Bedeutung haben. Unzulässig sind ausserdem Namen, die mit Ziffern beginnen.

Für die Darstellung von Namen für Variablen hat sich in Python eingebürgert, Variablen klein zu schreiben und Wörter durch Underlines zu trennen (*this_is_a_valid_variable*). Diese Darstellung nennt sich *Snake Case*. Zudem werden Variablen meist mit englischen Begriffen bezeichnet.

Weisen Sie in der nächsten Zelle der Variable `result` das Resultat der Rechnung $y + z$ zu und geben Sie das Resultat mit `print(result)` aus. `print()` ist eine Funktion, die Python zur Verfügung stellt. Was Funktionen sind, wird im nächsten Abschnitt erklärt.

```
# weisen Sie hier der Variable result das Resultat zu
```

4.3. Funktionen in Python

Python verfügt über viele bereits vordefinierte Funktionen. Die oben verwendete Funktion `print()` ist ein Beispiel dafür. Um zu demonstrieren, wie Funktionen in Python definiert werden, zeige ich Ihnen als Beispiel eine Funktion, mit der zwei Zahlen addiert werden.

```
# Definition der Funktion
def get_sum(x, y):
    return x + y

# Aufruf der Funktion
result = get_sum(3,4)
# Ausgabe des Resultats des Funktionsaufrufs
print(result)
```

`def` ist das Schlüsselwort für die Definition einer Funktion. `get_sum` ist der von mir gewählte Name dieser Funktion. Für die Wahl des Namens einer Funktion gelten die gleichen Regeln, wie für Variablen. In den Klammern stehen die sogenannten Parameter, welche der Funktion übergeben werden, damit sie etwas damit macht. Mit dem Doppelpunkt wird die *Signatur* der Funktion abgeschlossen. Die *Signatur* zeigt idealerweise, *was* eine Funktion *womit* macht. Sie gibt aber keine Auskunft darüber, wie sie das macht.

Python gruppiert Befehle, die zusammengehören, durch die gleiche Tiefe der Einrückung. Eine Einrückung hat üblicherweise die Tiefe von vier Leerzeichen. Im Beispiel oben gibt es nur eine eingerückte Zeile, weil die Funktion nur aus einem Befehl besteht. Mit `return` gibt die Funktion das Resultat zurück.

Im Beispiel wird das Resultat der Berechnung, welche die Funktion ausführt der Variable `result` zugewiesen. Der Wert der Variable `result` wird mit `print(result)` ausgegeben.

Definieren Sie in der folgenden Zelle eine Funktion, mit der zwei Zahlen multipliziert werden.

```
# hier kommt Ihre Funktion hin
```

4.4. Datentypen

Als nächstes geht es um die Frage, auf welche Inhalte eine Variable zeigen kann.

Im Grundsatz kann eine Variable auf beliebige Inhalte verweisen.

Am einfachsten ist die Verwendung der grundlegenden Datenypen (basic data types), welche Python zur Verfügung stellt. Dies sind (mit ihren englischen Bezeichnungen):

- Integer (Ganzzahl)
- Floating-Point Number (Gleitkommazahl)
- Complex Number (komplexe Zahl)
- String (Zeichenkette)
- Boolean Type (Wahrheitswert)

Darüber hinaus ist es möglich, eigene Datentypen zu programmieren. Hier aber zuerst eine Beschreibung der grundlegenden Datentypen von Python.

4.4.1. Integer

Die Bezeichnung für Integer in Python ist ein kurzes `int`.

Anders als in anderen Progammiersprachen gibt es in Python theoretisch keine Beschränkung, wie gross ein Integer sein kann. Die einzige Grenze ist der Speicherplatz des auf dem der Integer gespeichert werden soll.

Wenn einer Variable ein grosser Integer zugewiesen wird, kann dieser zur besseren Lesbarkeit auch mit einem Underline als Tausendertrennzeichen geschrieben werden (`100_000`).

Um das Auszuprobieren, weisen Sie in der folgenden Zelle der Variable *a* den Wert von einer Million und der Variable *b* den Wert von einer Milliarde zu. Anschliessend addieren Sie *a* und *b* und weisen das Resultat der Variable *big_sum* zu. Zum Schluss geben Sie den Wert von *big_sum* mit der Funktion `print()` aus.

```
# hier können Sie Ihre Berechnung vornehmen
```

Eingegebene Zahlen werden automatisch als Dezimalzahlen interpretiert.

Integers können jedoch auch als Binär-, Oktal- oder Hexadezimalzahlen eingegeben werden. Die Eingabe erfordert dann allerdings ein Präfix, welches das Zahlensysten identifiziert. Die folgende Tabelle stellt die möglichen Präfixe zusammen.

Präfix	Bedeutung	Basis
`0b` (Null + Kleinbuchstabe `b`)	Binärzahl	2
`0B` (Null + Grossbuchstabe `B`)		2
`0o` (Null + Kleinbuchstabe `o`)	Oktalzahl	8
`0O` (Null + Grossbuchstabe `O`)		8
`0x` (Null + Kleinbuchstabe `x`)	Hexadezimalzahl	16
`0X` (Null + Grossbuchstabe `X`)		16

In der folgende Zelle finden Sie ein entsprechendes Beispiel.

Python kann Integer in verschiedenen Zahlensystemen darstellen. Um das Zahlensystem bei der Zuweisung zu spezifizieren, verwenden Sie die entsprechenden Präfixe. Die folgenden Beispiele zeigen die Zuweisung der Zahlen 10, 8, 255 und 16 in den Zahlensystemen Dezimal, Oktal, Hexadezimal und Binär:

```
dezimal      = 10
oktal        = 0o10
hexadezimal  = 0x10
binaer       = 0b10
```

Die Anzeige der Werte der entsprechenden Variablen erfolgt grundsätzlich im Dezimalsystem.

```
b = 0b101010
o = 0o52
x = 0x2a

print(b, o, x)
```

4.4.2. Gleitkommazahl

Die Bezeichnung für Gleitkommazahlen in Python ist `float`. Python interpretiert Zahlen mit einem Dezimalpunkt als Gleitkommazahlen. Optional können Zahlen mit `e` oder `E` in "wissenschaftlicher" Schreibweise eigegeben werden (1000 = 1e3 bzw. 1e-3 = 0.001).

Weisen Sie in der folgenden Zelle den Variablen `million` und `billionth` die passenden Werte in wissenschaftlicher Schreibweise zu.

```
# hier die Werte den beiden Variablen zuweisen
```

4.4.3. Komplexe Zahlen

Python kann auch mit komplexen Zahlen umgehen. Der Abschnitt zu diesem Thema kann wieder aufgegriffen werden, wenn Sie in Mathe die komplexen Zahlen besprochen haben.

4.4.4. String

Zeichenketten (String) werden von Python als `str` bezeichnet.

Zeichenketten sind beliebige Zeichenfolgen. Damit Python Zeichenketten als solche erkennt, müssen sie durch die Verwendung von einfachen oder doppelten Anführungs- und Schlusszeichen als solche gekennzeichnet werden.

`"Ich bin eine Zeichenkette."` oder `'Ich bin auch eine Zeichenkette.'`

Wenn man innerhalb einer Zeichenkette Anführungszeichen braucht, müssen die eingrenzenden Anführungszeichen von der "anderen Sorte" sein (`"It's cool learning Python!"` oder `'Der Lehrer sagt: "Es ist cool Python zu lernen."'`). Eine andere Möglichkeit reservierte Zeichen zu verwenden ist der Gebrauch eines "escape"-Zeichens. In Python ist das der "backslash" (\). Die beiden Beispielsätze von vorher hätten entsprechend auch folgendermassen geschrieben werden können:

`'It\'s cool learning Python!'` bzw. `"Der Lehrer sagt: \"Es ist cool Python zu lernen.\""`

Die Länge von Zeichenketten wird lediglich durch die Speicherkapazität des verwendeten Systems begrenzt. Zeichenketten können nicht nur sehr lang, sondern auch leer sein (`''`).

Zeichenketten können, wie alle Datentypen, Variablen zugewiesen werden. Dies zeigt das folgende Beispiel.

```
# Zuweisung eines Strings zu einer Variabel
standard_greeting = "Hello World"

# Ausgabe der Variabel
print(standard_greeting)
```

4.4.5. Boolean Type

Wahrheitswerte werden in Python als `bool` bezeichnet. Wahrheitswerte können entweder "wahr" oder "falsch" sein.

```
wahr      = True
falsch    = False
```

Die Ausgabe findet sich in der folgenden Zelle.

```
wahr      = True
falsch    = False
print(wahr, falsch)
```

Welche Warheitswerte sich bei Vergleichen ergeben, kann mit den folgenden Operatoren überprüft werden:

```
==   # Gleichheit
!=   # Ungleichheit
>    # Größer als
<    # Kleiner als
>=   # Größer oder gleich
<=   # Kleiner oder gleich
```

```
# Testen Sie hier den Wahrheitswert von Vergleichen mit Zahlen
```

Wahrheitswerte werden zur Steuerung von Programmflüssen verwendet. Mit einem Wahrheitswert kann zum Beispiel gesteuert werden, wie oft ein Programmteil wiederholt werden soll.

4.5. Funktionen mit Type-Hints

Zum Abschluss komme ich noch einmal auf die Definition von Funktionen zurück. In Python können Variablen - anders als zum Beispiel in Java - beliebige Datentypen zugewiesen

werden. Wenn Variablen im Verlauf eines Programms mehrfach verwendet werden, können ihnen auch unterschiedliche Datentypen zugewiesen werden. Dies ist allerdings schlechter Programmierstil.

Aus diesem Grund ist es sinnvoll, bei der Definition einer Funktion zu deklarieren, welche Datentypen die Parameter haben und welcher Datentyp der Rückgabewert hat. Dies soll mit dem Beispiel der Funktion `get_quotient` verdeutlicht werden.

```
1 def get_quotient(x : int, y : int) -> float:
2     return x / y
```

Hier wird angegeben, dass die Parameter x und y vom Datentyp `int` sein sollen. Der Datentyp des Rückgabewertes wird hinter `->` geschrieben. Im Beispiel ist der Rückgabewert vom Typ `float`. Das ist so, weil die Funktion zum Beispiel $3/4 = 3.5$ rechnet.

Aber Achtung: die Funktion arbeitet auch dann korrekt, wenn ein anderer als der deklarierte Datentyp übergeben wird. Voraussetzung ist lediglich, dass der Datentyp mit den verwendeten Operationen kompatibel ist. Die "Type-Hints" dienen lediglich der besseren Nachvollziehbarkeit, was die Funktion macht.

4.6. Ausgewählte Musterlösungen

4.6.1. Funktion zur Multiplikation zweier Zahlen

```
1 def get_product(a, b):
2     return a * b
3 
4 product = get_product(3, 4)
5 print(product)
```

4.6.2. Gleitkommazahlen in wissenschaftlicher Schreibweise

```
1 million = 1e6
2 billonth = 1e-9
3 print(million, billonth)
```

```
1000000.0 1e-09
```

5. Wiederholungen in Python (For-Loops)

Eine Stärke von Computerprogrammen ist die wiederholte Ausführung von Anweisungen. Viele Programmiersprachen stellen dafür ein Konstrukt mit dem Namen 'For-Schleife' zur Verfügung. Eine 'For-Schleife' funktioniert unabhängig von einer konkreten Programmiersprache folgendermassen:

```
FÜR variable VON startwert BIS endwert [MIT schrittweite]
    Anweisungen
ENDE FÜR
```

Übersetzt nach Python sieht das so aus:

```
1  for i in range(n):
2      do...
```

`startwert BIS endwert [MIT schrittweite]` wird dabei durch `range(n)` ausgedrückt. Dabei ist `n` der Endwert. Gezählt wird bis zum aber ohne den Endwert. Startwert und Schrittweite haben Vorgabewerte. Der Vorgabewert für den Start ist `0`, derjenige der Schrittweite `1`. Weil `range()` diese vorgegebenen Werte hat, müssen diese nicht explizit angegeben werden. Wenn der Startwert abweichend vom Vorgabewert festgelegt werden soll, kann dieser explizit angegeben werden. Der Aufruf von `range()` sieht dann so aus:

```
1  range(startwert, endwert)
```

Falls eine von `1` abweichende Schrittweite festgelegt werden soll lautet der Aufruf

```
1  range(startwert, endwert, schrittweite)
```

In diesem Fall müssen neben dem Endwert sowohl der Startwert und die Schrittweite angegeben werden. Andernfalls kann nicht zwischen den einzelnen Angaben zu Endwert, Startwert und Schrittweite unterschieden werden.

5.1. Anwendungsübung zu Wiederholungen in Python

Im Folgenden bauen Sie eine Blume mit Python for-loops und PyTamaro.

In der folgenden Zelle werden zuerst alle von PyTamaro zur Verfügung gestellten Funktionen importiert.

```
1 from pytamaro.de import *
```

Das Ziel ist es, eine Blume wie die abgebildete zu zeichnen.

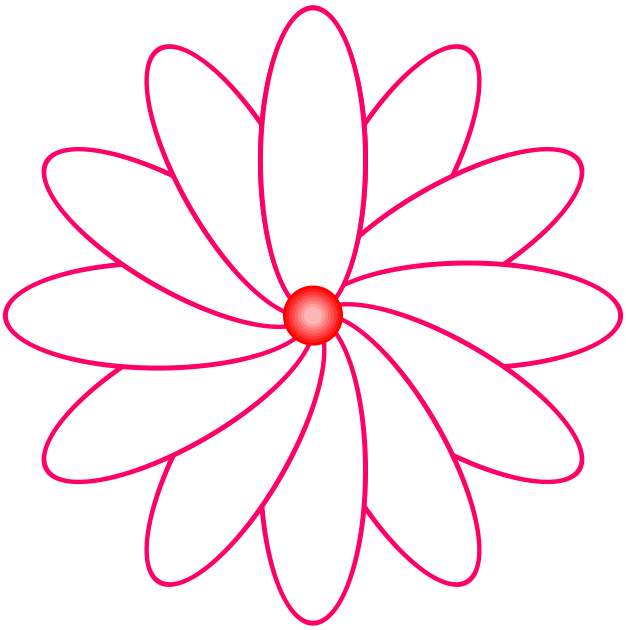


Abbildung 5.1.: Blume

Die wichtigsten Funktionen von PyTamaro für diese Aufgabe sind `fixiere()` und `kombiniere()`.

Die Funktion `fixiere()` legt für eine Grafik einen frei gewählten Ankerpunkt fest. Dazu muss man wissen, dass jede in PyTamaro gezichnete Grafik von einer sogenannten “Bounding Box” umgeben ist. Diese Bounding Box ist ein Rechteck, das die Grafik vollständig umschliesst. Standardmässig liegt der Ankerpunkt einer Grafik in der Mitte der Bounding Box. Mit `fixiere()` kann man den Ankerpunkt jedoch an eine andere Stelle der Bounding Box verschieben. PyTamaro stellt die folgenden Positionen für Ankerpunkte zur Verfügung:

- `mitte`: Mitte der Bounding Box (Standard)
- `mitte_links`: Mitte der linken Seite der Bounding Box
- `mitte_rechts`: Mitte der rechten Seite der Bounding Box
- `oben_links`: Obere linke Ecke der Bounding Box
- `oben_mitte`: Mitte der oberen Seite der Bounding Box
- `oben_rechts`: Obere rechte Ecke der Bounding Box
- `unten_links`: Untere linke Ecke der Bounding Box
- `unten_mitte`: Mitte der unteren Seite der Bounding Box
- `unten_rechts`: Obere rechte Ecke der Bounding Box

Der Ankerpunkt wird mit dem Befehl `fixiere(position, grafik)` gesetzt.

Der Ankerpunkt ist wichtig für die Funktion `drehe()`, die eine Grafik um einen bestimmten Winkel dreht. Die Drehung erfolgt immer um den Ankerpunkt.

Die Funktion `kombiniere(vordere Grafik, hintere Grafik)` fügt zwei Grafiken zusammen. Die erste gegebene Grafik liegt im Vordergrund und die zweite im Hintergrund. Die Grafiken werden so ausgerichtet, dass ihre Ankerpunkte übereinanderliegen.

5.1.1. Schritt 1: Zerlege das Bild in seine Einzelteile

Die Blume besteht im wesentlichen aus zwei Formen:

1. Blütenblätter in Form von Ellipsen sowie
2. einer Scheibe in der Form eines Kreises in der Mitte.

Um die Blume zu zeichnen, müssen diese beiden Formen zuerst einzeln erstellt werden.

```
bluetenblatt = ellipse(50, 150, blau)
# die Farbe blau wurde gewhält, damit das Blatt vor dem Hintergrund
# sichtbar ist

zeige_grafik(bluetenblatt)
```

```
scheibe = ellipse(30, 30, rot)
zeige_grafik(scheibe)
```

5.1.2. Schritt 2: Positioniere die Blütenblätter

Die Blütenblätter sind rund um die Mitte der Blume angeordnet. Da es zwölf Blütenblätter sind, beträgt der Winkel zwischen zwei Blütenblättern $\frac{360°}{12} = 30°$. Die Drehung der Blüten-

blätter erfolgt um den Ankerpunkt `unten_mitte` der Blütenblätter.
Die Blütenblätter müssen also zuerst fixiert und dann gedreht werden.

```
bluetenblatt_fixiert = fixiere(unten_mitte, bluetenblatt)
bluetenblatt_30 = drehe(30, bluetenblatt_fixiert)
zeige_grafik(bluetenblatt_30)
```

5.1.3. Schritt 3: Zeichnen aller erforderlichen Blütenblätter

Die Blume hat zwölf Blütenblätter. Diese können von Hand jedes einzelne erstellt werden. Das dritte Blütenblatt ist um 60° gedreht und wird entsprechend mit

```
bluetenblatt_60 = drehe(60, bluetenblatt_fixiert)
```

erstllt. Das kann man für alle zwölf Blütenblätter machen bis man beim zwölften Blütenblatt angekommen ist, das um 330° gedreht ist.

```
bluetenblatt_330 = drehe(330, bluetenblatt_fixiert)
```

Das ist aber sehr mühsam und fehleranfällig. Viel einfacher ist es, eine Schlaufe (`for`-loop) zu verwenden, die die Blütenblätter basierend auf der Grundform automatisch erstellt.

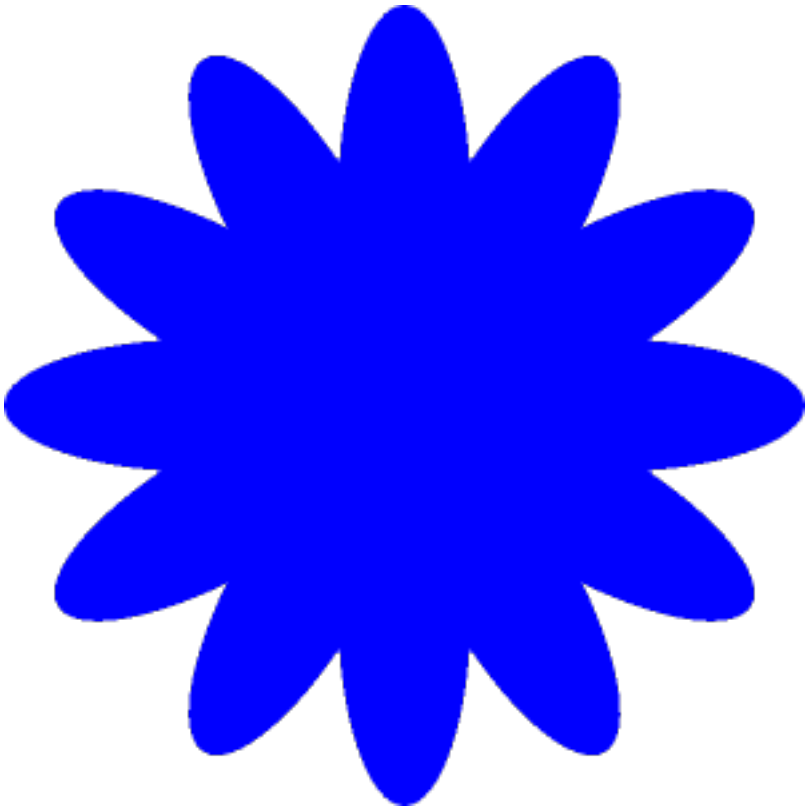
```
for i in range(12):
    winkel = i * 30
    bluetenblatt_gedreht = drehe(winkel, bluetenblatt_fixiert)

zeige_grafik(bluetenblatt_gedreht)
```

Diese Schlaufe durchläuft die Zahlen von 0 bis 11 (also 12 Zahlen) und berechnet für jede Zahl den entsprechenden Drehwinkel. Dann wird das Blütenblatt um diesen Winkel gedreht. Allerdings wird das gedrehte Blütenblatt in jeder Iteration der Schlaufe in der gleichen Variable `bluetenblatt_gedreht` gespeichert. Am Ende der Schlaufe enthält diese Variable nur das letzte gedrehte Blütenblatt (das um 330° gedrehte Blütenblatt).

5.1.4. Schritt 4: Kombiniere alle Blütenblätter

Um das unerwünschte Verhalten zu vermeiden, dass am Ende der Schlaufe nur das letzte gedrehte Blütenblatt in der Variable `bluetenblatt_gedreht` enthalten ist, müssen die gedrehten Blütenblätter in jeder Iteration der gewünschten Gesamtgrafik hinzugefügt werden. Dazu wird die Funktion `kombiniere()` verwendet.

```
blume = bluetenblatt_fixiert

for i in range(12):
    winkel = i * 30
    bluetenblatt_gedreht = drehe(winkel, bluetenblatt_fixiert)
    blume = kombiniere(blume, bluetenblatt_gedreht)

zeige_grafik(blume)
```



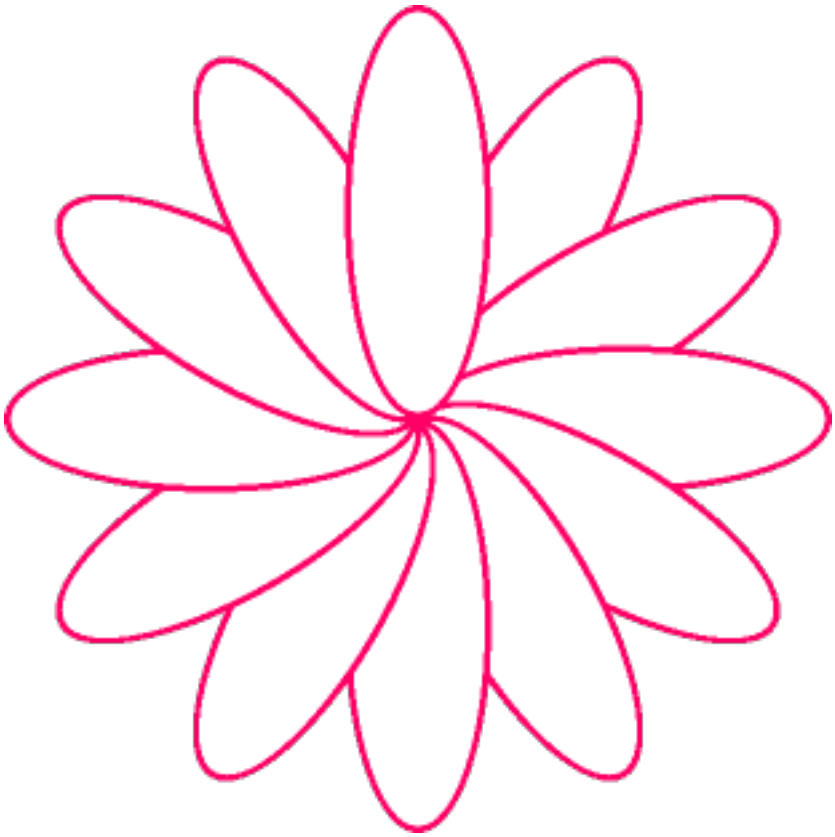
Damit sind die Blütenblätter in der gewünschten Position und Anzahl zusammengefügt. Damit Sie allerdings aussehen, wie die Blume in der Vorlage, müssen die einzelnen Blütenblätter weiss eingefärbt und mit einem rosa Rand versehen werden.

Rosa ist keine vordefinierte Farbe in PyTamaro. Sie können jedoch eine beliebige Farbe mit der Funktion `rgb_farbe(rot, gruen, blau)` erstellen. Dabei sind `rot`, `gruen` und `blau` Zahlenwerte von 0 bis 255, die den Anteil der jeweiligen Farbe an der Gesamtfarbe angeben. Damit nicht alle Kombinationen durchprobiert werden müssen, können Sie mit einem Online-Tool wie RGB Color Picker die gewünschte Farbe auswählen und die entsprechenden Werte für rot, grün und blau ablesen.

```
weisses_blatt = ellipse(50, 150, weiss)
rosa_rand = ellipse(55, 155, rgb_farbe(255, 0, 102))

bluetenblatt_mit_rand = ueberlagere(weisses_blatt, rosa_rand)

zeige_grafik(bluetenblatt_mit_rand)
```

Anschliessend können die Blütenblätter mit Rand in der vorher geschriebenen Schlaufe zur Gesamtgrafik zusammengefügt werden.

```
bluetenblatt_mit_rand_fixiert = fixiere(unten_mitte, bluetenblatt_mit_rand)

blume_weiss = bluetenblatt_mit_rand_fixiert

for i in range(12):
    winkel = i * 30
    bluetenblatt_gedreht = drehe(winkel, bluetenblatt_mit_rand_fixiert)
    blume_weiss = kombiniere(blume_weiss, bluetenblatt_gedreht)

zeige_grafik(blume_weiss)
```



5.1.5. Schritt 5: Scheibe im Zentrum der Blume

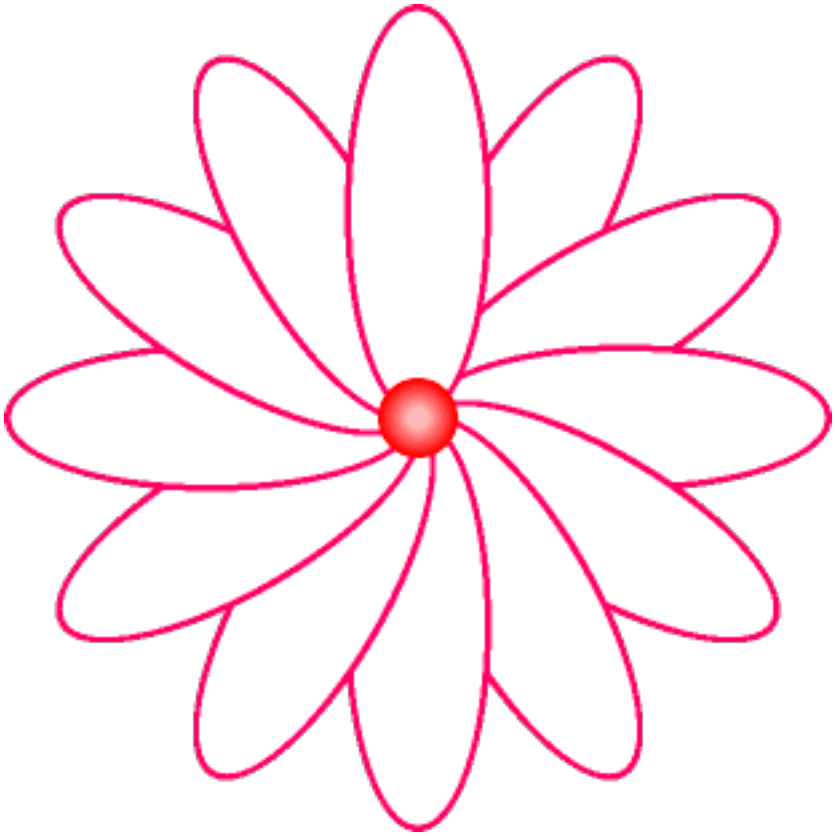
Bei genauer Betrachtung der Blume fällt auf, dass die Scheibe in der Mitte der Blume nicht einfach einfarbig gelb ist, sondern gegen die Mitte hin einen Farbverlauf aufweist. Diesen Farbverlauf wird erzeugt, indem mehrere Kreise mit kleiner werdendem Radius und abnehmender Farbintensität übereinandergelegt werden.

```
scheibe = ellipse(30, 30, rgb_farbe(255, 0, 0))

for i in range(1, 8):
    tmp = ellipse(30 - 3*i, 30 - 3*i, rgb_farbe(255, i*26, i*26))
    scheibe = ueberlagere(tmp, scheibe)

zeige_grafik(scheibe)
```

5.1.6. Schritt 6: Kombination der Elemente zur vollständigen Blume

Zum Schluss wird die Scheibe auf die Blütenblätter gelegt und Blume so vervollständigt.

```
blume_komplett = ueberlagere(scheibe, blume_weiss)
zeige_grafik(blume_komplett)
```



6. Programmverzweigungen (Bedingungen)

Es gibt Situationen, in denen es wünschenswert ist, dass ein Teil eines Programms nur ausgeführt wird, wenn eine bestimmte Voraussetzung erfüllt ist.
Als Beispiel dafür mag die Blume von vergangener Woche dienen. Wenn eine Blume gezeichnet werden soll, welche abwechslungsweise dunkelblaue und hellblaue Blütenblätter hat.

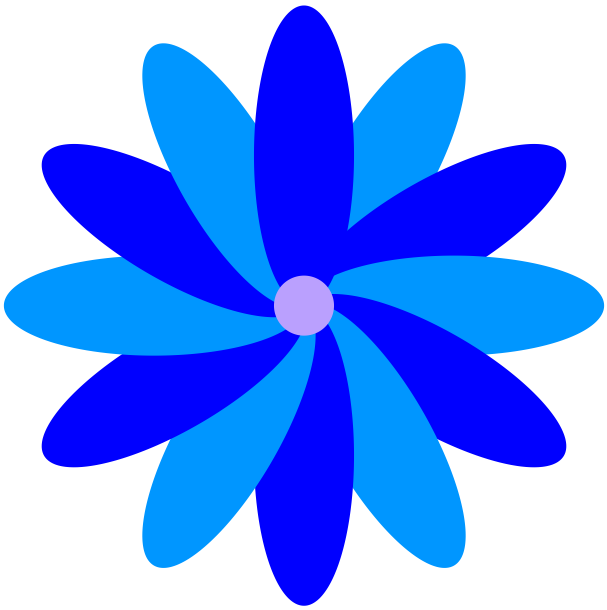


Abbildung 6.1.: Beispielblume

Diese Blume wurde ebenfalls mit einer `for`-Schlaufe gezeichnet. Allerdings wurde abwechslungsweise ein dunkelblaues und ein hellblaues Blütenblatt verwendet.
Abwechslungsweise kann auch als wenn eine gerade Zahl kommt bzw. wenn eine ungerade Zahl kommt verstanden werden. Kommen beim Zählen doch abwechslungsweise gerade und ungerade Zahlen. Übersetzt in die `for i in range(12)`-Schlaufe heisst das, immer wenn der Wert in der Variabel `i` eine gerade Zahl ist dann soll ein dunkelblaues Blüentblatt verwendet werden und immer wenn der Wert der Variabel `i` eine ungerade Zahl ist, soll ein hellblaues Blütenblatt verwendet werden.

In Python verwendet man dazu die folgende Syntax:

```
1 if BEDINGUNG:
2     ANWEISUNG(en)
3 else:
```

```
4       ANWEISUNG(en)
```

Das heisst, immer wenn die `BEDINGUNG` erfüllt ist (wenn sie `wahr`) ist, werden die eingerückten Anweisungen ausgeführt. In allen anderen Fällen werden die nach `else:` eingerückten Anweisungen ausgeführt.

Für die Zeichnung der Blume bedeutet das also, dass immer wenn `i` eine gerade Zahl ist, soll ein dunkelblaues Blütenblatt verwendet werden. Eine gerade Zahl ist dabei jede Zahl, die ohne Rest durch zwei Teilbar ist.
Ob dies der Fall ist, kann mit dem Modulo Operator (`%`) getestet werden (vgl. Abschnitt Vorbemerkungen: Python als Rechner.) Eine Zahl ist ohne Rest durch zwei Teilbar, wenn gilt

$$i \ mod \ 2 = 0$$

Übersetzt nach Pyhton schreibt sich das als

```
1  if i % 2 == 0:
```

Für die Prüfung, ob das Resultat 0 ist, wird dabei `==` verwendet. Dies rührt daher, dass das einfache gleichheitszeichen bereits als Operator für die Zuweisung eines Wertes zu einer Variabel besetzt war.

Der Vollständige Code für die Abwechslungsweise Verwendung eines dunkelblauen bzw. eines hellblauen Blütenblattes sieht folgendermassen aus:

```
1  if i % 2 == 0:
2      blume = kombiniere(blume, bluetenblatt_dunkel)
3  else:
4      blume = kombiniere(blume, bluetenblatt_hell)
```

Falls mehr als zwei Fälle unterschieden werden müssen, lautet die Python Syntax

```
1  if BEDINGUNG:
2      ANWEISUNG(en)
3  elif BEDINGUNG:
4      ANWEISUNG(en)
5  else:
6      ANWEISUNG(en)
```

Dabei können beliebig viele `elif` Bedingungen wiederholt werden.

Als Bedingung kann nicht nur eine Gleichheit überprüft werden. Es ist auch möglich grösser `>`, grösser oder gleich `>=`, kleiner `<` und kleiner oder gleich `<=` zu prüfen.

6.1. Anwendungsübung zu Bedingungen in Python

6.1.1. Aufgabenstellung

Zeichnen Sie die folgende Blume.

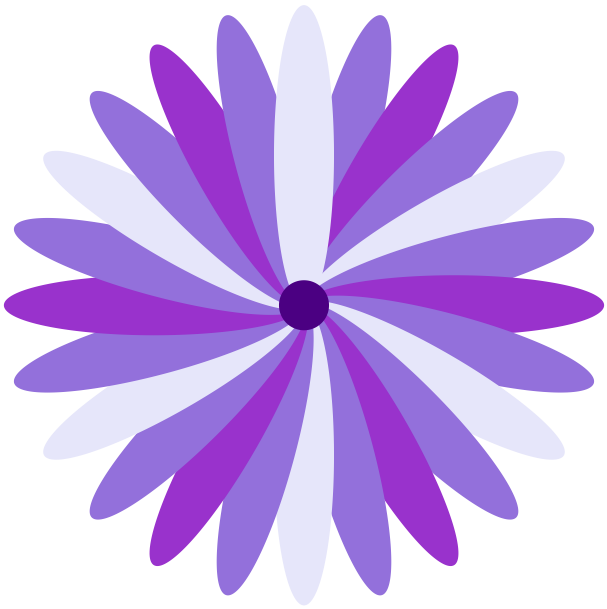


Abbildung 6.2.: Blumenvorlage

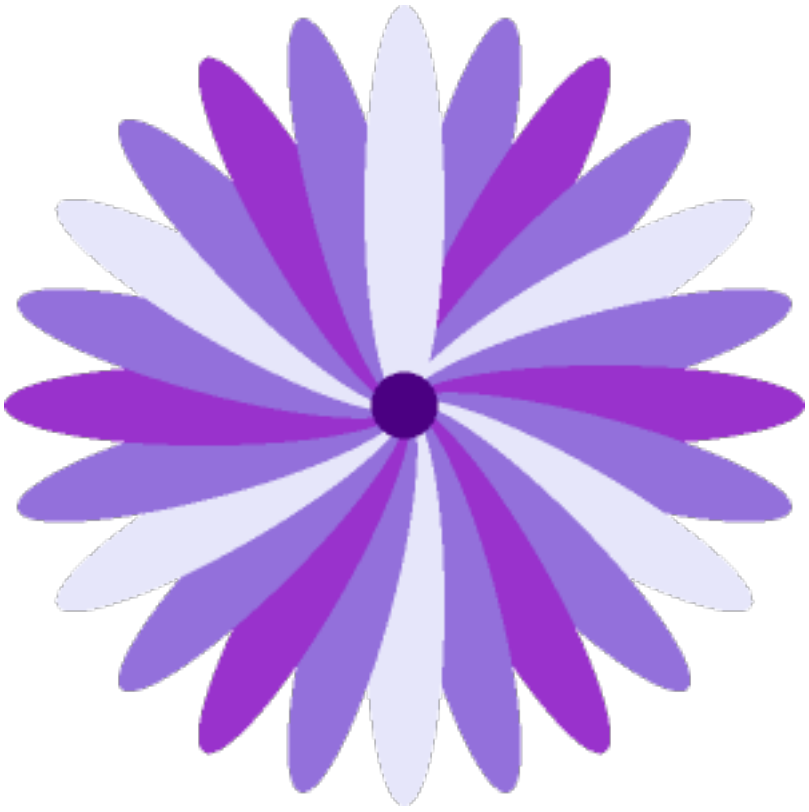
Die erfoderlichen Importe und Farbdefinitionen finden Sie in der folgenden Code Zelle.

```
from pytamaro.de import *

lavender = rgb_farbe(230, 230, 250)
mediumpurple = rgb_farbe(147, 112, 219)
darkorchid = rgb_farbe(153, 50, 204)
indigo = rgb_farbe(75, 0, 130)
```

```
# Hier ist Platz für Ihre Blume
# Hinweis: die Verschiebung einer Farbe um eine Postion kann erreicht
# werden, wenn i % 4 nicht 0 sondern eine Zahl ergibt. Bei % 4 können
# dies die Zahlen 1, 2 oder 3 sein.
```

6.1.2. Musterlösung

```
bluetenblatt_l = ellipse(30, 150, lavender)
bluetenblatt_m = ellipse(30, 150, mediumpurple)
bluetenblatt_d = ellipse(30, 150, darkorchid)
scheibe_i = ellipse(25, 25, indigo)

bluetenblatt_m_fix = fixiere(unten_mitte, bluetenblatt_m)
bluetenblatt_l_fix = fixiere(unten_mitte, bluetenblatt_l)
bluetenblatt_d_fix = fixiere(unten_mitte, bluetenblatt_d)

blume3 = scheibe_i
for i in range(24):

    if i % 4 == 0:
        bl_temp = drehe(i * 15, bluetenblatt_l_fix)
        blume3 = kombiniere(blume3, bl_temp)
    elif i % 4 == 2:
        bd_temp = drehe(i * 15, bluetenblatt_d_fix)
        blume3 = kombiniere(blume3, bd_temp)
    else:
        bm_temp = drehe(i * 15, bluetenblatt_m_fix)
        blume3 = kombiniere(blume3, bm_temp)

zeige_grafik(blume3)
```



7. Programmieren einfacher Funktionen

7.1. Kreisfläche

Die Fläche eines Kreises berechnet sich nach der Formel

$$A = \pi r^2$$

wobei A die Fläche π die Kreiszahl - hier mit zwei Nachkommastellen - 3.14 und r den Radius des Kreises darstellt.

Aufgabe: Implementieren Sie eine Funktion `kreisflaeche()` welche ein Argument `r` für Radius entgegennimmt und die Fläche eines Kreises zurückgibt.

```
1  def kreisflaeche():
2      # TODO: implementieren Sie die Funktion
```

7.2. Quotient

Ein Quotient kann unterschiedlich dargestellt werden. Eine Möglichkeit ist die Darstellung als Dezimalzahl.

Aufgabe: Implementieren Sie eine Funktion `quotient()` welche zwei Dezimalzahlen als Argumente entgegennimmt und den Quotienten als Dezimalzahl zurückgibt. Sehen Sie für den Sonderfall der Division durch Null eine Lösung vor.

```
1  def quotient():
2      # TODO: implementieren Sie die Funktion
```

7.3. Fakultät

Die Fakultät ist jene Funktion, welche das Produkt der natürlichen Zahlen bis n berechnet und wird als $n!$ geschrieben. Es gilt also

$$n! = 1 \cdot 2 \cdot 3 \cdot ... \cdot n$$

Wobei zu beachten ist, dass $0! = 1$ gilt.

Aufgabe: Implementieren Sie eine Funktion `fakultaet` welche als Argument $n \in \mathbb{N}$ entgegennimmt.

```
def fakultaet():
    # TODO: implementieren Sie die Funktion
```

7.4. Quadratische Gleichungen (Mitternachtsformel)

Quadratische Gleichungen der Form $ax^2 + bx + c = 0$ können mit der Formel

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

gelöst werden.
Für die Implementierung ergibt sich daraus allerdings das Problem, dass die Formel zwei Resultate ergibt und eine Python Funktion nur ein Objekt zurückgeben kann.

Als Workaround können die beiden Resultate in ein Tupel gepackt werden. Ein Tupel ist eine unveränderliche Zusammenstellung mehrer Werte. In Python werden sie in Klammern dargestellt. Die Zuweisung eines Tupels zu einer Variabel sieht dementsprechend folgendermassen aus:

```
t = (wert_1, wert_2, ..., wert_n)
```

Die detaillierten eigenschaften von Tupeln können hier ausser Acht gelassen werden. Es reicht x_1 und x_2 einem Tupel `resultat` zuzuweisen und `resultat` zurückzugeben.

Aufgabe: Implementieren Sie eine Funktion `mitternachtsformel`, welche die Argumente a, b und c entgegennimt und das Tupel x_1 und x_2 zurückgibt.

```
def mitternachtsformel():
    # TODO: implementieren Sie die Funktion
```

7.5. Musterlösungen

7.5.1. Kreisfläche

```
def kreisflaeche(r: float) -> float:
    """
    Berechnet die Fläche eines Kreises anhand des gegebenen Radius.

    Parameter
```

```
    ----------
    r : float
        Der Radius des Kreises.

    Rückgabewert
    ------------
    float
        Die Fläche des Kreises.
    """
    PI = 3.14  # Näherungswert für die Kreiszahl Pi
    a = PI * r ** 2  # Formel: Fläche = Pi * r^2
    return a
```

7.5.2. Quotient

```
def quotient(z: float, n: float) -> float:
    """
    Berechnet den Quotienten zweier Zahlen.

    Gibt eine Fehlermeldung zurück, falls der Nenner null ist.

    Parameter
    ----------
    z : float
        Der Zähler der Division.
    n : float
        Der Nenner der Division.

    Rückgabewert
    ------------
    float oder str
        Der berechnete Quotient oder eine Fehlermeldung als Zeichenkette,
        falls eine Division durch Null versucht wurde.
    """

    # Fehlermeldung für Division durch Null
    fehlermeldung = "Eine Division durch Null ist unzulässig."
    if n == 0:
        return fehlermeldung  # Fehlerfall: Division durch Null
    else:
        q = z / n  # Berechnung des Quotienten
        return q
```

7.5.3. Fakultät

```python
def fakultaet(n: int) -> int:
    """
    Berechnet die Fakultät einer nicht-negativen ganzen Zahl n.

    Die Fakultät n! ist das Produkt aller positiven ganzen Zahlen
    von 1 bis n. Für n < 0 ist die Fakultät nicht definiert.

    Parameter:
        n (int): Die Zahl, deren Fakultät berechnet werden soll.

    Rückgabewert:
        int: Die Fakultät von n, oder eine Fehlermeldung als String bei n < 0.
    """
    if n < 0:
        # Fakultät ist für negative Zahlen nicht definiert
        return f'Die Fakultät für {n} ist nicht definiert.'
    elif n == 0:
        # Per Definition ist 0! = 1
        return 1
    else:
        prod = 1  # Initialisiere das Produkt mit 1
        for i in range(1, n + 1):
            prod *= i  # Multipliziere prod mit dem aktuellen Wert von i
        return prod  # Gib das berechnete Produkt zurück
```

7.5.4. Mitternachtsformel

```python
def mitternachtsformel(a: float, b: float, c: float) -> tuple:
    """
    Löst eine quadratische Gleichung der Form ax² + bx + c = 0
    mit der Mitternachtsformel.

    Parameter
    ----------
    a : float
        Der Koeffizient vor x² (muss ungleich 0 sein).
    b : float
        Der Koeffizient vor x.
    c : float
        Der konstante Term.

```

```
    Rückgabewert
    ------------
    tuple oder str
        Ein Tupel mit den beiden Lösungen (x1, x2) oder eine Fehlermeldung
        als Zeichenkette, falls die Gleichung keine reellen Lösungen besitzt.
    """
    # Berechnung der Diskriminante: d = b² - 4ac
    d = b ** 2 - 4 * a * c
    if d < 0:
        # Keine reellen Lösungen vorhanden
        return "Diese Gleichung hat keine Lösung."
    else:
        # Berechnung der beiden Lösungen mit der Mitternachtsformel:
        # x1 = (-b + d ** (1/2)) / (2a)
        # x2 = (-b -  d ** (1/2)) / (2a)
        x1 = (-b + d ** (1/2)) / (2 * a)
        x2 = (-b - d ** (1/2)) / (2 * a)
        return (x1, x2)
```

8. Prüfungsvorbereitung

8.1. Belegung von Variabeln

Das folgende Code Snippet demonstriert die Belegung von Variabeln. Insbesondere soll es auch zeigen, was geschieht, wenn die Variabeln in Operationen einbezogen werden.

Überlegen Sie sich bevor Sie das Snippet ausführen zuerst Anweisung für Anweisung, welche Werte in den Variabeln a, b und c nach Ausführung der jeweiligen Anweisung gespeichert sind. Notieren Sie sich die Werte auf ein Blatt Papier. Anschliessend führen Sie die Zelle zur Kontrolle Ihrer Überlegungen aus.

```
# Anfangswerte zuweisen
a = 5
b = 10
c = a
print(f"Nach Initialisierung:, a = {a}, b = {b}, c = {c}")

# Wert von a ändern
a = a + 3
print(f"Nach a = a + 3: a = {a},  b = {b},  c = {c}")

# Wert von b ändern
b = a * c
print(f"Nach b = a * c: a = {a},  b = {b},  c = {c}")

# Wert von c ändern
c = b - a
print(f"Nach c = b - a: a = {a},  b = {b},  c = {c}")
```

8.2. Funktionen

Die Definiton einer neuen Funktion erweitert den Umfang der in Python zur Verfügung stehenden Befehle. Entsprechend ist es möglich, einmal definierte Funktionen in neuen Funktionen zu verwenden.

Unten finden Sie ein einfaches Beispiel für diese Vorgehensweise.

```python
def erfolgsberechnung(ertrag: float, aufwand: float) -> float:
    """Berechnet den wirtschaftlichen Erfolg als Differenz von Ertrag
    und Aufwand.

    Args:
        ertrag: Der gesamte erzielte Ertrag.
        aufwand: Der gesamte angefallene Aufwand.

    Returns:
        Der berechnete Erfolg (Ertrag minus Aufwand).
    """
    erfolg = ertrag - aufwand
    return erfolg


def renditeberechnung(ertrag: float, aufwand: float,
                      anfangskapital: float,
                      schlusskapital: float) -> float:
    """Berechnet die Rendite in Prozent basierend auf Erfolg
    und durchschnittlichem Kapital.

    Die Funktion nutzt die 'erfolgsberechnung', um den Erfolg
    zu ermitteln, und setzt diesen ins Verhältnis zum durchschnittlich
    gebundenen Kapital.

    Args:
        ertrag: Der gesamte erzielte Ertrag.
        aufwand: Der gesamte angefallene Aufwand.
        anfangskapital: Das Kapital zu Beginn der Periode.
        schlusskapital: Das Kapital am Ende der Periode.

    Returns:
        Die berechnete Rendite, ausgedrückt in Prozent (%).
    """
    erfolg = erfolgsberechnung(ertrag, aufwand)
    durchschnittskapital = (anfangskapital + schlusskapital) / 2
    rendite = erfolg / durchschnittskapital * 100
    return rendite
```

8.2.1. Berechnung des Volumens eines Zylinders

Die folgende Grafik zeigt einen Zylinder.

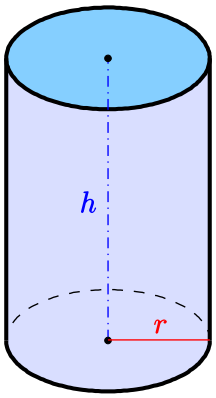


Abbildung 8.1.: Zylinder

Die Formel für die Berechnung des Volumens lautet

$$v = \pi \cdot r^2 \cdot h$$

Zerlegen Sie dazu die Berechnung in zwei Teile. Definieren Sie zuerst eine Funkiton für die Berechnung der Bodenfläche und verwenden Sie diese anschliessend in einer Funktion zur Berechnung des Volumens.

Für die Kreiszahl π können Sie das Modul `math` importieren. Dies stellt Ihnen mit dem Befehl `math.pi` π in einer grossen genauigkeit zur Verfügung.

```
1  # TODO: implementieren Sie hier Ihre Funktionen
```

8.2.2. Berechnung der kinetischen Energie eines Körpers

Die kinetische Energie eines Körpers berechnet sich nach der Formel

$$E_{kin} = \frac{1}{2} \cdot m \cdot v^2$$

wobei m für die Masse des Körpers und v für dessen Geschwindigkeit steht. Implementieren Sie eine Funktion für die Berechnung der kinetischen Energie eines Körpers. Beachten Sie dabei, dass experimentell lediglich Daten über die Masse des Körpers, die zurückgelegte Strecke des Körpers sowie die dafür erforderliche Zeit zugänglich sind.

```
1  # TODO: implementieren Sie hier Ihre Funktionen
```

8.3. Musterlösungen

8.3.1. Berechnung des Volumnens eines Zylinders

```python
import math

def kreisflaeche(r : float) -> float:
    area = math.pi * r ** 2
    return area

def zylindervolumen(r : float, h : float) -> float:
    bodenflaeche = kreisflaeche(r)
    volumen = bodenflaeche * h
    return volumen
```

8.3.2. Berechnung der kinetischen Energie eines Körpers

```python
def geschwindigkeit(s: float, t: float) -> float:
    """Berechnet die Geschwindigkeit v aus zurückgelegter Strecke und
    Zeit.

    Args:
        s: Die zurückgelegte Strecke (in Metern).
        t: Die dafür benötigte Zeit (in Sekunden).

    Returns:
        Die berechnete Geschwindigkeit v (in m/s).
    """
    v = s / t
    return v


def kinetische_energie(m: float, s: float, t: float) -> float:
    """Berechnet die kinetische Energie eines Körpers basierend auf
    experimentellen Daten.

    Die Funktion berechnet zuerst die Geschwindigkeit v aus s und t
    mithilfe der Funktion 'geschwindigkeit' und wendet dann die Formel
    E_kin = 0.5 * m * v^2 an.

    Args:
        m: Die Masse des Körpers (in Kilogramm).
        s: Die zurückgelegte Strecke (in Metern).
```

```
27          t: Die dafür benötigte Zeit (in Sekunden).
28
29      Returns:
30          Die berechnete kinetische Energie E_kin (in Joule).
31      """
32      v = geschwindigkeit(s, t)
33      e_kin = (m * v ** 2) / 2
34      return e_kin
```

9. Datenstrukturen: Listen

9.1. Daten in Listen speichern

Bisher hatten Sie Datentypen wie `int`, `float` `str` und `bool` angeschaut. Wir möchten nun mehrere Werte in einer Liste speichern.

Ich möchte zum Beispiel alle Wochentagekürzel in einer Liste speichern. Um eine Liste im Code zu speichern, muss man die Liste mit `[` beginnen und mit `]` enden. Die Elemente werden mit einem Komma `,` getrennt.

```
1 wochentage = ["Mo", "Di", "Mi", "Do", "Fr", "Sa", "So"]
2 print(wochentage)
```

9.1.1. Aufgabe: Monatsliste erstellen

Machen Sie eine Liste mit den 12 Monaten und printen Sie die Monate.

```
1 """Aufgabe: Monatsliste erstellen"""
2
3 monate = # TODO
4 print(monate)
```

9.2. Index

Die Werte werden durchnummeriert. Die dazugehörige Nummer eines Werts nennen wir den *Index*.

Das erste Element hat den Index 0!

Um ein Element aus der Liste auszugeben können wir nach dem Variablennamen die eckigen Klammen und den Index angeben.

`wochentage[0]` ist `"Mo"`. `wochentage[1]` ist `"Di"`.

9.2.1. Aufgabe: Index lesen

Lesen Sie den Codeblock und schrieben Sie hier auf, was Sie als Output erwarten, *bevor* Sie den Codeblock ausführen!

Ich erwarte, dass das Programm folgendes ausgibt:

```
TODO: Diese Zeile ersetzen.
```

```
lieblingstag = wochentage[3]
print(f"Mein Lieblingstag ist {lieblingstag}.")
```

9.3. Überschreiben von Elementen

Wir betrachten in diesem Kapitel eine Liste mit Zahlen (`int`). Wir speichern für jeden Monat die Länge des Monats. Im Falle eines Schaltjahrs möchten wir den Februar überschreiben mit 29. Studieren Sie das Beispiel:

```
monatslaenge = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
schaltjahr = True
if schaltjahr:
    monatslaenge[1] = 29 # Der Februar hat den Index 2.
print(monatslaenge)
```

9.4. Schleifen

Listen eignen sich hervorragend um eine for-Schleife zu bauen. Betrachten Sie das Beispiel, welches den Index als `i` und den Wochentagsname als `wochentag` ausgibt. Die Liste `wochentage` möchten wir auf keinen Fall überschreiben. Also verwenden wir für die *Laufvaribale* andere Variablennamen. In diesem Fall nehmen wir `i` für den Index und `wochentag` (Singular) für den Name des Tages.

```
for i, wochentag in enumerate(wochentage):
    print(f"{i}: {wochentag}")
```

Falls Sie den Index nicht benötigen, können Sie die Schleife mit `for wochentag in wochentage:` machen.

9.4.1. Aufgabe: Monatsschleife komplettieren

Erstellen Sie eine Schleife, welche folgendes Ausgibt:

```
Der Januar hat 31 Tage.
Der Februar hat 29 Tage.
Der März hat 31 Tage.
Der April hat 30 Tage.
Der Mai hat 31 Tage.
Der Juni hat 30 Tage.
Der Juli hat 31 Tage.
Der August hat 31 Tage.
Der September hat 30 Tage.
Der Oktober hat 31 Tage.
Der November hat 30 Tage.
Der Dezember hat 31 Tage.
```

```
"""Aufgabe: Monatsschleife komplettieren"""

for ... m_name in ...:
    laenge = ... i ...
    print(f"Der {m_name} hat {laenge} Tage.")
```

9.5. Länge

Mit der Funktion `len(obj)` kann man die Länge der Liste `obj` abfragen.

```
"""Beispiel für len(obj)"""
print(f"Die Woche hat {len(wochentage)} Tage.")
```

9.6. Ergänzen und entfernen

Mit der Methode `append` kann ein Element am Ende der Liste hinzufügen und mit `pop` wieder entfernen.

```
tasks = ["einkaufen", "spazieren gehen", "staubsaugen"]
tasks.append("essen")
print(f"Neu ist auch 'essen' auf der Taskliste: {tasks}")

tasks.pop(1)
print(f"Welcher Task ist nun erledigt? {tasks}")
```

```
7
8  tasks.pop()
9  print("Wird kein Index angegeben, so wird das letzte Element"
10       f"in der Liste entfernt: {tasks}")
```

9.7. Aufgabe: Fibonacci

Die Fibonacci Reihe lautet 1, 1, 2, 3, 5, 8, 13, 8+13, …

Die Summe der letzten zwei Zahlen ergeben die nächste Zahl.

Ergänzen Sie den Code, damit die Folge länger wird. Editieren Sie Stellen mit drei Punkten.

```
1  """Aufgabe: Fibonacci"""
2
3  fibonacci: list = [1, 1, 2, 3, 5, 8, 13] # Diese Zeile nicht verändern.
4  for _ in range(4):
5      n = ... # Die Länge von der Folge
6      fibonacci ... (fibonacci ... + fibonacci ...)
7      print(fibonacci)
```

9.8. Aufgabe: zeichne Chomp

Wie am Anfang der Lektion mündlich besprochen kodieren wir eine Situation von Chomp.

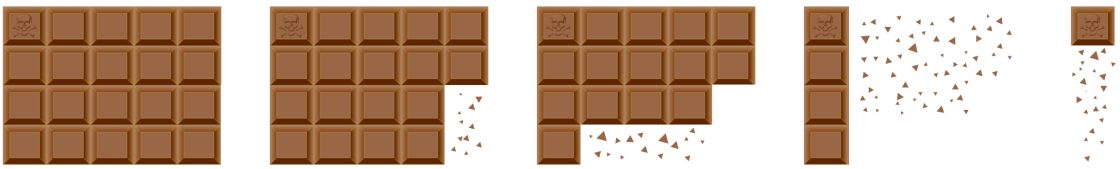


Abbildung 9.1.: Chomp_gameplay

Die dritte Situation beschreiben wir als Liste `[4, 3, 3, 3, 2]`.

Schreiben Sie eine Funktion `zeichne_chomp(chomp: list) -> None` welche die Situation zeichnet.

```
1  """Aufgabe: zeichne Chomp"""
2  from pytamaro.de import *
3
4  def zeichne_chomp(chomp: list) -> None:
```

```
    zelle = kombiniere(
        rechteck(68, 38, rgb_farbe(153, 103, 69)),
        rechteck(70, 40, rgb_farbe(51, 34, 23)),
    )
    leere_zelle = rechteck(70, 40, weiss)
    tafel = leere_grafik()
    maximum = # TODO # Groesse der erste Spalte
    for ... in chomp:
        spalte = leere_grafik()
        ...
        ...
        tafel = neben(tafel, spalte)
    zeige_grafik(tafel)
```

```
# Test für die Aufgabe
chomp_test: list = [4, 3, 3, 3, 2]
zeichne_chomp(chomp_test)
```

9.9. Aufgabe: spiele Chomp

Schreiben Sie eine Funktion `chomp_spielzug(chomp: list, spielzug:list) -> list` welche aus der ersten Situation die neue Situation berechnet. Der Spielzug ist eine Liste von der Form `[x, y]`.

Test: `chomp_spielzug([4, 3, 3, 3, 2], spielzug=[3, 2])` ergibt `[4, 3, 1, 1, 1]`

Bonusidee: Überprüfen Sie, ob der Spielzug überhaupt erlaubt ist.

```
"""Aufgabe: programmiere den Spielzug in Chomp"""

# TODO: Programmieren Sie es selber!


# Hier können Sie den Code testen:
assert chomp_spielzug([4, 3, 3, 3, 2], spielzug=[3, 2]) == [4, 3, 1, 1, 1]
```

9.10. Übersicht

Sie wissen, wie Sie

- eine Liste erstellen.
- Elemente abfragen.

- Elemente ändern.
- mit einer Schleife durch die Liste gehen.
- die Länge einer Liste bestimmen.
- Elemente hinzufügen oder entfernen.

9.11. Zusatzstoff

Weitere Infos und Features von Listen auf https://docs.python.org/3/library/stdtypes.html#sequence-types-list-tuple-range

Stichwörter: Slice, Spezialfall str, insert, in, join

10. Datenstrukturen: Dictionary

Dictionaries (Wörterbücher oder Maps) können mehrere Werte in einem Objekt speichern, wie es Listen auch können. Die Werte haben aber keinen Index sondern einen Schlüssel.

Ein *Dictionary* in Python ist vergleichbar mit einem echten Wörterbuch. Stellen Sie sich vor, Sie suchen die Telefonnummer (Wert) einer Person (Schlüssel). In einem Python-Dictionary könnten Sie den Namen der Person als Schlüssel und ihre Telefonnummer als Wert speichern.

```
"""Beispiel von einem dict"""
telefonbuch = {
    "Feuerwehr": 118,
    "Polizei": 117,
    "Ambulanz": 144,
    "Rega": 1414,
    "Beratung + Hilfe 147": 147,
    "allgemeiner Notruf": 112,
}
```

10.1. Operationen für Dictionaries

- Erstellen eines Dictionaries: `dict_name: dict = {}`

  Erstellt ein neues, leeres Dictionary mit dem Variablennamen *dict_name*.

- Elemente aktualisieren: `dict_name['Schlüssel'] = 'Neuer Wert'`

  Aktualisiert den Wert eines bestehenden Schlüssels oder fügt einen neuen Schlüssel mit dem angegebenen Wert hinzu, wenn er nicht existiert.

- Elemente aufrufen: `var: str = dict_name['Schlüssel']`

  `var` ist ein str mit dem Wert `'Neuer Wert'`. Man kann den Wert auch direkt zum printen oder anderes brauchen.

- Elemente löschen: `dict_name.pop('Schlüssel')`

  Entfernt das Element mit dem angegebenen Schlüssel aus dem Dictionary.

- Durch das Dictionary durchgehen: `for key, value in dict_name.items()`

  Dieser Prozess heisst *Iterieren*. Wir kennen ihn schon von der Liste oder Range. Iterieren heisst die Liste von Anfang bis Ende durchgehen. Jedes Mal wird in die Variablen `key`, `vlaue` den Schlüssel, respektive den Wert des Eintrags zugewiesen. Sie können den Variablen auch andere Namen geben.

10.1.1. Aufgabe: Vervollständigen Sie das dict

```
"""Aufgabe: dict vervollständigen"""

voci = {
    "und": "and",
    "Auto": ... ,
    "Katze": ... ,
    "Tag": ... ,
    "Ende": ... ,
    "Familie": ... ,
    "Zuhause": ... ,
    "Name": ... ,
    "Menschen": ... ,
    "lesen": ... ,
    "Schule": ... ,
    "sprechen": ... ,
}

# Test
print(voci["Schule"])
```

10.1.2. Aufgabe: Schreiben Sie ein dict

Schreiben Sie ein `dict` für Ihre Klasse, sodass der Nachname dem Schlüssel/key und der Vorname dem Wert/value entspricht.

```
"""Aufgabe: schreiben Sie ein dict"""
klasse = ...
```

10.1.3. Aufgabe: Schleife

Iterieren Sie durch die ganze Liste, sodass alle aus der Klasse mit `"Hallo [Vorname] [Nachname]"` begrüsst werden.

```python
"""Aufgabe: Schleife"""
for nachname, vorname in ... :
    print(f"Hallo {vorname} {nachname}")
```

10.1.4. Aufgabe: Selektion

Geben Sie nur die Nachnamen aus, welche ein `e` im Nachnamen haben.

Tipp: Verwenden Sie `if 'e' in nachname:`

```python
"""Aufgabe: Selektion"""

...
```

10.1.5. Aufgabe: Telefonliste

Erstellen Sie eine Liste `threedigtslist`, die nur die Namen der Notfallorganisationen aus dem dict `telefonbuch` welche genau 3 Ziffern beinhalten.

```python
"""Aufgabe: Telefonliste"""

...


# Test
assert type(threedigtslist) == list
assert len(threedigtslist) == 5
for soultion in ['Feuerwehr', 'Polizei', 'Ambulanz', 'Beratung + Hilfe 147', 'allgemeiner
    assert soultion in threedigtslist
```

11. Beurteilung von Algorithmen

Die Effizienz eines Algorithmus kann grundsätzlich nach seinem Rechenaufwand oder Speicherbedarf beurteilt werden. Man spricht in diesem Zusammenhang auch von Zeit- bzw. Platzkomplexität.

Wir haben uns mit dem Zählen von Vergleichen nur mit der Rechenaufwand, also mit der Zeitkomplexität beschäftigt. Die Berechnungen werden sehr stark vereinfacht.

11.1. Bubblesort

Wenn wir mit dem Bubblesort n Objekte sortieren, haben wir $n-1$ Vergleiche im ersten Durchgang. Vereinfacht sagen wir einfach n Vergleiche. Im schlimsten Fall, müssen ist das kleinste Element am falschen Ende. Bei jedem Durchgang verschieben wir das Element nur um einen Platz. Dass heisst wir müssen etwa n Durchgänge machen. Die Laufzeit vom Bubblesort beträgt somit $n \cdot n = n^2$.

11.2. Mergesort

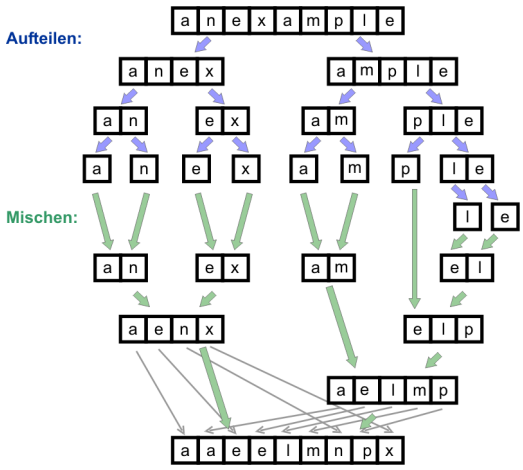


Abbildung 11.1.: Mergesort_example

Bevor wir uns die Laufzeit von Mergesort anschauen, müssen wir quantifizieren, wie oft wir die Listen aufteilen müssen. Haben wir 2 Elemente, separieren wir 1 Mal. Haben wir 4 Elemente separieren wir 2 mal.

len	Teilungen
2	1
4	2
8	3
16	4
32	5
64	6
1024	10

Diese Funktion, welche der Länge zur Anzahl Teilungen zuordnet, nennt man *Logarithmus zur Basis 2*. Man notiert sie wie folgt: $\log_2(n)$

Auf jeder der $\log_2(n)$ Ebene machen wir etwa n Vergleiche. Die Laufzeit vom Mergesort beträgt somit $n \cdot \log_2(n)$.

12. Divide and Conquer

Das Prinzip *divide and conquer* (teile und herrsche) wurde am Beispiel von Mergesort gezeigt, aber nicht explizit erwähnt. Dieses Prinzip basiert auf der Idee, das kleine Probleme einfacher zu lösen sind als grosse. Deshalb versucht man, das zu lösende Problem in Teilprobleme aufzuteilen und diese, kleineren Probleme, zu lösen um anschliessend die Teillösungen zu einer Lösung für das Ursprungsproblem zusammenzusetzen.

Diese Idee wird oft durch *Rekursion* umgesetzt. Der Begriff Rekursion kann am besten am Beispiel einer Kindergeschichte erklärt werden:

> Es isch e mal en Maa gsi, de hät en hole Zah gha. I dem Zah häts es Truckli gha und i däm Truckli häts es Briefli gha. I dem Briefli isch gstande, es isch e mal en Maa gsi, de hät en hole Zah gha. I dem Zah häts es Truckli gha und id däm Truckli häts es Briefli gha…

Rekursive Funktionen sind entsprechend Funktionen, die sich selber aufrufen. Mergesort rief sich selber auf, um die Liste zu sortieren.

12.1. Fibonacci-Folge

Die Fibonacci-Folge - benannt nach dem Italienischen Mathematiker Leonardo Fibonacci - ist eine Zahlenfolge, in welcher die nächste Zahl die Summe der beiden vorangegangenen Zahlen bildet. In Zahlen sieht das folgendermassen aus:

$$1, 1, 2, 3, 5, 8, 13, ...$$

Die Fibonacci-Folge wird rekursiv deifiniert:

$$f_n = f_{n-1} + f_{n-2}, \text{ sofern } n \geq 3$$

Sie haben die Fibonacci-Folge bei den Listen bereits programmiert. Obwohl die Programmiung als Liste *viel* effizienter ist, machen wir hier eine Übung, Fibonacci als rekursive Funktion zu programmieren:

```
"""Beispiel: fibonacci rekursiv definiert mit Printstatement"""
def fibonacci(n: int) -> int:
    print(f"Es wird fibonacci({n}) ausgeführt.")
    if n < 2:
```

```
5           return 1
6       return fibonacci(n-1) + fibonacci(n-2)
```

12.1.1. Aufgabe: Fibonacci-Folge

1. Lesen Sie die Funktion `fibonacci`. Denken Sie im Kopf durch, was passiert, wenn Sie `fibonacci(0)` oder `fibonacci(1)` ausführen. Schreiben Sie Ihre Vermutung auf und führen Sie erst dann die Funktion aus. Vergleichen Sie Ihre Vermutung mit dem Resultat.
2. Denken Sie im Kopf durch, was passiert, wenn Sie `fibonacci(2)` ausführen. Schreiben Sie Ihre Vermutung auf und führen Sie erst dann die Funktion aus. Vergleichen Sie Ihre Vermutung mit dem Resultat.
3. Bonus: Denken Sie im Kopf durch, was passiert, wenn Sie `fibonacci(3)` ausführen. Schreiben Sie Ihre Vermutung auf und führen Sie erst dann die Funktion aus. Vergleichen Sie Ihre Vermutung mit dem Resultat.

12.2. Endlose Rekursion

Wenn wir bei der Funktion `fibonacci` die Zeilen

```
1       if n < 2:
2           return 1
```

weglassen würden, so würde die Funktion endlos sich selber aufrufen, wie im Beispiel der Kindergeschichte.

Meistens wird dieses Problem mit Basisfälle gelöst.

12.3. Beispiel: Anstossen

Gestern assen wir mit 6 Personen zu Abend. Vorgestern mit 140 Personen. Wie oft schlagen die Gläser zusammen wenn alle miteinander anstossen?

Die Aufgabe können wir einfach reduzieren. Wenn ich die 6. Person am Tisch bin, stosse ich mit 5 Personen an. Die 5 Personen müssen untereinander noch anstossen. Das heisst $a_n = a_{n-1} + (n-1)$.

Für die Rekursive Funktion, brauchen wir somit nur noch die Basisfälle. In unserem Beispiel ist es: Wenn nur 0, 1 oder 2 Personen da sind, wie oft wird angestossen?

Personen	Anstossen
0	0
1	0

Personen	Anstossen
2	1

```python
"""Aufgabe: finden Sie 2 Fehler"""
def anstossen(n: int) -> int:
    ## Basisfälle
    if n <= 1:
        return 0
    if n == 2:
        return 2

    return anstossen(n-1) + (n + 1)
```

```python
"""Test"""
for i in range(10):
    print(f"{i} Personen stossen {anstossen(i)} Mal an.")

i = 140
print(f"{i} Personen stossen {anstossen(i)} Mal an.")
```

12.3.1. Bonus: Die Gausssche Summenformel

Der Deutsche Mathematiker Carl Friedich Gauss soll nach der Legende seinen Mathematiklehrer bei einer Strafaufgabe überlistet haben. Gemäss dieser Legende soll der Lehrer Gauss (und seinen Klassenkameraden) den Auftrag gegen haben, die Zahlen von 1 bis 100 zusammenzuzählen.

Gauss soll nach kurzer Zeit mit der Lösung zum Lehrer gegangen sein. Gauss' Lösung basierte auf folgender Formel:

$$1 + 2 + 3 + ... + (n-1) + n = \sum_{k=1}^{n} k = \frac{n(n+1)}{2}$$

Diese Formel kann direkt als Funktion implementiert werden. Das entsprechende Beispiel finden sie in der folgenden Zelle.

```python
def gauss_direct(n : int) -> int:
    return int((n*(n+1))/2)

print(f'Die Summe der Zahlen von 1 '
      f'bis 100 ist {gauss_direct(100)}.')
```

Alternativ kann die Formel aber auch rekursiv implementiert werden. Entsprechend stellt sich die Frage, was wäre ein kleineres, aber grundsätzlich gleichartiges Problem?
Im vorliegenden Fall wäre das einfachste gleichartige Problem, die Summe aus einer Liste von Zahlen mit der Länge 1 und dem Wert 1 zu bilden. Dann ist die Summe auch 1. Als Formel könnte man das so schreiben:

$$\sum_{k=1}^{n} k = 1, \text{ sofern } n = 1$$

Daraus ergibt sich die Frage, wie man von $n = 100$ zu $n = 1$ kommt.

Für alle Fälle, in denen $n > 1$ ist, gilt

$$\sum_{k=1}^{n} k = n + \sum_{k=1}^{n-1} k$$

In dieser Formel kann nun immer wieder $n - 1$ eingesetzt werden. Das ist die Rekursion.

Zusammengefasst kann dies folgendermassen dargestellt werden:

$$\sum_{k=1}^{n} = \begin{cases} 1, & n = 1 \quad \text{Basisfall} \\ n + \sum_{k=1}^{n-1} k, & \forall n > 1 \quad \text{Rekursionsfall} \end{cases}$$

Der Basisfall ist einerseits der einfachste Fall und andererseits auch der letzte Fall, der bearbeitet werden muss.
Beides ist wichtig. Dass der Basisfall der einfachste Fall ist, hilft das Problem zu lösen. Dass der Basisfall der Lezte Fall ist, der bearbeitet werden muss, ermöglicht es, die Rekursion zu beenden.

Im Rekursionsfall steht "$\forall n > 1$". Das $\forall$ Zeichen ist der sogenannte Allquantor und bedeutet hier "**für alle** n grösser als 1.

Das eine Problemlösung rekursiv implementiert wird, bedeutet, dass die Funktion sich selber aufruft.
In der folgenden Zelle wird die gausssche Summenformel gemäss obiger Darstellung rekursiv implementiert.

```
"""Bonusaufgabe: Gauss Rekursiv"""
def gauss_recursive(n : int) -> int:
    # TODO: rekursive Implementation des kleinen Gauss
    pass

print(f'Die Summe der Zahlen von 1 '
      f'bis 100 ist {gauss_recursive(100)}.')
```

12.4. Fakultät

Mit Fakultät ($n!$) wird in der Mathematik jene Funktion bezeichnet mit der alle natürlichen Zahlen $\leq n$ miteinander multipliziert werden.

$$n! = 1 \cdot 2 \cdot ... \cdot (n-1) \cdot n = \prod_{k=1}^{n} k$$

Als Besonderheit muss mann wissen, dass $0! = 1$ gilt.

In Python kann dies mit einer `for` Schleife berechnet werden. In der folgenden Zelle wird eine entsprechende Funktion definiert.

```python
def factorial_loop(n : int) -> int:
    """result wird gleich 1 gesetzt,
    weil 1 das neutrale Element der
    Multiplikation ist """
    result = 1
    for i in range(1, n+1):
        result *= i

    return result

print(f'Das Resultat von 5! ist '
      f'{factorial_loop(5)}.')
```

Die Funktion kann allerdings auch rekursiv definiert werden. Die Definition sieht dann folgendermassen aus:

$$n! = \begin{cases} 1, & n = 0 \vee n = 1 \quad \text{Basisfall} \\ n \times (n-1)!, & \forall n > 1 \quad \text{Rekursionsfall} \end{cases}$$

Entsprechend kann eine Funktion zur Berechnung von $n!$ auch rekursiv implementiert werden.

```python
"""Aufgabe: Fakultät"""
def factorial_recursive(n : int) -> int:
    # TODO: rekursive Implementation der Fakultätsfunktion
    pass

print(f'Das Resultat von 5! ist '
      f'{factorial_recursive(5)}.')
```

12.5. Wörter umdrehen

Gesucht ist eine rekursive Funktion, welche ein Wort rückwärts schreibt (bsp. Kantonsschule nach eluhcssnotnaK).

```
"""Aufgabe: umdrehen"""
def reverse(s : str) -> str:
    if len(s) == 0 or len(s) == 1:
        return ...
    else:
        head = s[0]
        tail = s[1:]
        return ...
```

```
print(reverse('Kantonsschule'))
```

12.6. Bonus: Anzahl Möglichkeiten zu Zahlen

Ich Verkaufe eine Heft für 6 CHF. Wie viele Möglichkeiten gibt es 5 CHF in Münzen (wir betrachten nur die Münzen 1, 2, 5 CHF).

- $5 + 1$
- $2 + 2 + 2$
- $2 + 2 + 1 + 1$
- $2 + 1 + 1 + 1 + 1$

Es gibt 4 Möglichkeiten.

Um die Frage zu beantworten, kann man die Frage auch aufteilen. Die grösste mögliche Münze ist 5. Also gibt es die Möglichkeiten mit einem 5 Fränkler zu bezahlen (1 Möglichkeit) oder ohne keinem 5 Fränkler zu bezahlen (3 Möglichkeiten). Diese Anzahl Möglichkeiten zusammen addiert ergibt die Möglichkeiten. Die Funktion benötigt 2 Argumente: Den zu bezahlenden Betrag und die grösste erlaubte Münze.

```
"""
Bonusaufgabe
Studieren Sie die Funktion und füllen Sie die Lücken mit den Auslassungspunkten.
"""
MUENZEN = [1000, 200, 100, 50, 20, 10, 5, 2, 1]
def anz_zahlmöglichkeiten(
    value: int,
    largest_index: int=0,
    ) -> int:
    """Rekursive Funktion"""
```

```
    if value < 0:
        return 0
    if value == 0:
        return 1
    if largest_index + 1 == len(MUENZEN):
        # Pay with smales coins
        return 1
    # Pay with largest Coin
    ret = ...

    # Pay without larges coin
    ret += ...
    return ret

print(anz_zahlmöglichkeiten(5))
```

```
print(anz_zahlmöglichkeiten(500))  # 6295434
# print(anz_zahlmöglichkeiten(700))  # 40208370
# print(anz_zahlmöglichkeiten(1000)) # 321335887
```

```
"""
Bonusaufgabe
Printen Sie die Möglichkeiten!

[5]
[2, 2, 1]
[2, 1, 1, 1]
[1, 1, 1, 1, 1]
"""
MUENZEN = [1000, 200, 100, 50, 20, 10, 5, 2, 1]
def print_zahlmöglichkeiten(...)
    ...

print_zahlmöglichkeiten(5)
```

13. Prüfungsvorbereitung 2

13.1. Lernziele

Ich erwarte, dass Sie in der Lage sind

- eine Jupyter Arbeitsumgebung zu starten (auswendig);
- Python Listen zu erstellen und zu verändern;
- Werte aus Python Listen direkt oder mit Hilfe von `for` Schleifen zu verarbeiten;
- Python Dictionaries zu erstellen und zu verändern;
- Wert aus Python Dictionaries direkt oder mit Hilfe von `for` Schleifen zu verarbeiten sowie
- rekursive Funktionen zu implementieren.

Die Inhalte der letzten Prüfung werden als Grundlagen vorausgesetzt. Die Prüfung findet mit Papier und Bleistift statt. Als Hilfsmittel dürfen Sie eine handschriftliche Zusammenfassung im Umfang einer Seite A4 mitbringen. Die Zusammenfassung darf alles ausser die Schritte zum Start einer Jupyter Arbeitsumgebung enthalten.

13.2. Listen

13.2.1. Listen erstellen

Das Fähnlein Fieselschweif besteht aus Donald Ducks Neffen Tick, Trick und Track. Erstellen Sie eine Liste `fieselschweif` mit den drei Mitgliedern.

```
1  # TODO: Liste erstellen
```

13.2.2. Listen bearbeiten

Ergänzen Sie die vorher erstellte Liste 'fieselschweif' um den Namen *Dagobert*.

```
1  # TODO: fieselschweif ergänzen
```

Dagobert gefällt es nicht im Fähnlein Fieselschweif. Entfernen Sie Ihn wieder und lassen Sie Ihren Code den Text *Dagobert hat das Fähnlein Fieselschweif wieder verlassen.* ausgeben. Dagobert soll dabei als Variabel `quitter` in den Code eingefügt werden.

```
# TODO: implementieren Sie die obige Vorgabe
```

13.2.3. Über Listen iterieren

Gegeben sei eine Liste `numbers`. Die Liste beinhaltet die natürlichen Zahlen von 1 bis 10 (`numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`). Erstellen Sie mit Hilfe einer `for` schleife basierend auf der Liste `numbers` eine neue Liste `triplet` welche aus den Elementen der Dreierreihe des kleinen Einmaleins besteht.

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# TODO: implementieren Sie die obige Vorgabe
```

13.2.4. Python Dictionary erstellen

Erstellen Sie ein Python Dictionary `mk`, welches die Geschäftsadresse von Jacques Mock Schindler enthält. Die ersten zwei Einträge erhalten Sie als Vorlage.

```
mk = {
    'name': 'Mock Schindler',
    'first_name': 'Jacques',
    # TODO: kompletieren Sie das Dictionary
}
```

13.2.5. Über ein Python Dictionary iterieren

Geben Sie den Inhalt des Dictionary `mk` mit Hilfe einer `for` Schleife aus. Dabei soll die Ausgabe jedes Eintrags auf einer neuen Zeile erfolgen.

```
# TODO: implementieren Sie obige Vorgabe
```

13.2.6. Rekursion

Ein Palindrom ist ein Wort, das in beide Richtungen gelesen werden kann. Ein Beispiel für ein Palindrom ist *Sugus*. Schreiben Sie eine rekursive Funktion `palindrom_tester`, welche testet, ob eine gegebene Zeichenfolge ein Palindrom ist.

Hinweis: Die einzelnen Zeichen eines Strings können wie die Elemente einer Liste mit Hilfe eines Index abgerufen werden. Das erste Zeichen eines Strings hat dabei den Index 0.

Damit Sie einfacher prüfen können, ob Ihre Funktion korrekt arbeitet, übergeben Sie ihr ausschliesslich Strings in Kleinbuchstaben.

```
# TODO: implementieren Sie die obige Vorgabe
```

13.3. Musterlösungen

13.3.1. Listen erstellen

Das Fähnlein Fieselschweif besteht aus Donald Ducks Neffen Tick, Trick und Track. Erstellen Sie eine Liste `fieselschweif` mit den drei Mitgliedern.

```
fieselschweif = ['Tick', 'Trick', 'Track']
```

13.3.2. Listen bearbeiten

Ergänzen Sie die vorher erstellte Liste 'fieselschweif' um den Namen *Dagobert*.

```
fieselschweif.append('Dagobert')
```

Dagobert gefällt es nicht im Fähnlein Fieselschweif. Entfernen Sie Ihn wieder und lassen Sie Ihren Code den Text *Dagobert hat das Fähnlein Fieselschweif wieder verlassen.* ausgeben. Dagobert soll dabei als Variabel `quitter` in den Code eingefügt werden.

```
quitter = fieselschweif.pop()
print(f'{quitter} hat das Fähnlein Fieslschweif wieder verlassen.')
```

13.3.3. Über Listen iterieren

Gegeben sei eine Liste `numbers`. Die Liste beinhaltet die natürlichen Zahlen von 1 bis 10 (`numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`). Erstellen Sie mit Hilfe einer `for` schleife basierend auf der Liste `numbers` eine neue Liste `triplet` welche aus den Elementen der Dreierreihe des kleinen Einmaleins besteht.

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

triplet = []

for number in numbers:
    triplet.append(number * 3)
```

13.3.4. Python Dictionary erstellen

Erstellen Sie ein Python Dictionary `mk`, welches die Geschäftsadresse von Jacques Mock Schindler enthält. Die ersten zwei Einträge erhalten Sie als Vorlage.

```
mk = {
    'name': 'Mock Schindler',
    'first_name': 'Jacques',
    'street': 'Rosenstrasse 1',
    'postcode': 8400,
    'town': 'Winterthur',
}
```

13.3.5. Über ein Python Dictionary iterieren

Geben Sie den Inhalt des Dictionary `mk` mit Hilfe einer `for` Schleife aus. Dabei soll die Ausgabe jedes Eintrags auf einer neuen Zeile erfolgen.

```
for key, value in mk.items():
    print(f'{key}: {value}')
```

13.3.6. Rekursion

Ein Palindrom ist ein Wort, das in beide Richtungen gelesen werden kann. Ein Beispiel für ein Palindrom ist *Sugus*. Schreiben Sie eine rekursive Funktion `palindrom_tester`, welche testet, ob eine gegebene Zeichenfolge ein Palindrom ist.

Hinweis: Die einzelnen Zeichen eines Strings können wie die Elemente einer Liste mit Hilfe eines Index abgerufen werden. Das erste Zeichen eines Strings hat dabei den Index 0.

Damit Sie einfacher prüfen können, ob Ihre Funktion korrekt arbeitet, übergeben Sie ihr ausschliesslich Strings in Kleinbuchstaben.

```
def palindrom_tester(text):
    """
    Prüft rekursiv, ob ein String ein Palindrom ist
    (vorwärts und rückwärts gelesen gleich).
    Beispiele: "Lagerregal", "Anna", "Reittier"
    """
    # Vorverarbeitung: Kleinbuchstaben für einfacheren Vergleich
    # (für den Fall, dass Sie sich nicht an den Tipp gehalten haben)
    text = text.lower()

    # Rekursionsanker 1: Leerer String oder Einzelbuchstabe
```

```
    # ist immer ein Palindrom.
    if len(text) <= 1:
        return True

    # Rekursionsanker 2: Wenn erster und letzter Buchstabe
    # nicht übereinstimmen,
    # kann es kein Palindrom sein. Abbruch mit False.
    if text[0] != text[-1]:
        return False

    # Rekursionsschritt:
    # Wenn die Ränder passen, prüfe den Teilstring ohne den
    # ersten und letzten Buchstaben.
    return palindrom_tester(text[1:-1])
```