# Python für Anfänger\*innen

Dieser Workshop ist eine Einführung in die Programmierung mit Python für Teilnehmende mit wenig oder keiner Erfahrung. Dieser Workshop enthält einen Überblick über Datentypen, den Einsatz von Variablen, Operatoren, logische Operationen, sowie Schleifen und Funktionen.

## Steckbrief

### Zielgruppe

3 bis 8 Teilnehmer\*innen zwischen 12 und 18 Jahren in Präsenz.

### Lizenz

Text: CC BY 4.0 Jugend hackt Berlin / Tobias Kolb

Code: GPL 3.0

### Kategorie

Coding

### Online-Version mit Links zu weiteren zugehörigen Dokumenten

[den Link ergänzen wir, sobald das OER hochgeladen wird]

### Was brauche ich dafür?

**Hardware:** Laptops, sowohl zur Verfügung gestellt als auch durch Teilnehmende mitgebracht

**Software:** Jupyter Notebook, PowerPoint oder andere Presenter Software

**Geräte:** Beamer

**Internet:** Ja

**Sonstiges:** -

**Personenanzahl:** 3 bis 8 Teilnehmer\*innen, 2 bis 4 Mentor\*innen

### Wie lange dauert der Workshop?

Ein Workshop dauert in der kurzen Variante 3 Stunden, in der langen Variante 5 Stunden.

[Kursablauf 4](#_Toc153153057)

[Vorbereitung 4](#_Toc153153058)

[Einleitung 5](#_Toc153153059)

[Abschnitt 1: Willkommen 5](#_Toc153153060)

[Abschnitt 2: Theorie 5](#_Toc153153061)

[Abschnitt 3: Code Along 6](#_Toc153153062)

[Abschnitt 4: Aufgaben 14](#_Toc153153063)

### Einführung

Der Kurs ist inhaltlich auf ca. 5 Stunden in Präsenz ausgelegt. Die Betreuung während des Kurses findet durch mindestens eine\*n Betreuer\*in auf 3 Teilnehmer\*innen statt. Die Betreuer\*innen müssen nicht alle Programmiererfahrung haben, weil es bei der Durchführung des Kurses auch oft darum geht die Teilnehmenden dazu zu motivieren Lösungen selbst zu erarbeiten und bei Problemen mit der Aufgabe sich gegenseitig zuzuhören, damit Teilnehmende über das Problem nachdenken und es dadurch ggf. selbstständig lösen können. Es ist jedoch ratsam, den Kurs nicht mit nur einer Aufsichtsperson mit praktischer Programmiererfahrung durchzuführen. Gerade während des Code Alongs kann es zu Situationen kommen in denen einzelne Teilnehmende inhaltliche Schwierigkeiten haben. In so einem Moment hilft es, wenn ein\*e Betreuer\*in der\*m Teilnehmer\*in diskret dabei helfen kann, das Problem zu debuggen, ohne dass es zu sehr in den zeitlichen Ablauf eingreift und die präsentierende Betreuer\*in das Code Along unterbrechen muss.

## Kursablauf

### Vorbereitung

Für die Vorbereitung zum Workshop ist es zentral die Jupyter Notebook Infrastruktur aufzusetzen.

Ein Ansible Script zum aufsetzen von Virtuellen Maschinen mit Jupyter in der Hetzner Cloud findet ihr [hier](https://github.com/Jugendhackt/ber-lab-jupyter-ansible). Das Script befindet sich nicht in den Kursmaterialien, weil es kontinuierlich weiterentwickelt wird.

Wenn es nicht möglich ist eigene Jupyter Notebooks auf Servern bereitzustellen, ist eine Alternative Google Colab zu verwenden. Dies ist aus Datenschutzgründen nicht ratsam, aber kostenlos und erfordert nur minimalen Aufwand. Google Colab hat bereits die meisten Pakete, die für den Kurs notwendig sind, installiert.

Ansonsten gibt es weitere wichtige optionale Schritte für die Vorbereitung:

1. Der Kurs muss beworben werden, damit Teilnehmende auf ihn aufmerksam gemacht werden.
2. Das Kursmaterial sollte vor der Durchführung einmal getestet werden.
3. Anpassungen am Kursmaterial sind möglicherweise notwendig, um bspw. Links zu aktualisieren oder Platzhalter zu ersetzen.
4. Ein Willkommensspiel, damit die Teilnehmenden sich kennenlernen können, muss geplant werden.
5. Aufgrund der Dauer des Kurses ist eine Verpflegung mit Obst, Getränken und ähnlichen Snacks ratsam.
6. Sollte der Kurs ohne Beisein der Eltern durchgeführt werden, sollte eine Telefonliste vorbereitet werden.
7. Es ist ratsam für die Buchung des Kurses ein Buchungssystem wie [Pretix](https://pretix.eu/about/en/) zu verwenden.

**Einleitung**

Der Kurs besteht im Wesentlichen aus 4 Abschnitten:

1. Abschnitt: Willkommen (30 min)
2. Abschnitt: Theorie als Präsentation (ca. 15 min + 15 min Pause)
3. Abschnitt: Python Code Along in Jupyter Notebook (ca. 75 min + 15 min Pause)
4. Abschnitt: Praxisphase mit einer Auswahl an Projekten (135 min inkl. Pausen)
5. Abschnitt: Vorstellung der Ergebnisse (15 min)

Die Abschnitte werden im Folgenden genauer erläutert.

### Abschnitt 1: Willkommen (30min)

Der Kurs beginnt mit einer Vorstellungsrunde oder einem Willkommensspiel. Zu Beginn des Kurses können auch Namensschilder, gerne mit selbstgewählten Pronomen, verteilt werden. Die Teilnehmenden können mit Fragen an das Thema herangeführt werden:

* Habt ihr schon einmal programmiert?
* Was wollt ihr heute lernen?
* Wie ist euer Wissensstand in Python?
* Was interessiert euch am meisten?
* Was ist Programmieren?

### Abschnitt 2: Theorie (15min)

Darauf folgt eine Präsentation zu dem Thema, die den Teilnehmenden Theorie vermittelt. Die Präsentation befindet sich in den Kursmaterialien. Es handelt sich hierbei nur um zwei Folien, in denen zunächst die Teilnehmenden gefragt werden was Programmieren eigentlich ist und dann kurz Python als Steckbrief vorgestellt wird. Um den Kursablauf transparent zu machen, kann ein Pad mit den wichtigsten Inhalten, dem Ablauf und wichtigen Links gezeigt werden.

### Abschnitt 3: Code Along (75min)

Zu Beginn des Code Alongs sollten die Jupyter Notebooks auf der eigenen oder Fremdplattform geöffnet werden. Als kostenlose Plattform eignet sich [Google Colab](https://colab.research.google.com/).

Bevorzugt sollten eigene Instanzen von Jupyter Notebooks auf separaten Virtuellen Maschinen aufgesetzt werden. Hinweise zur Vorbereitung findet ihr im entsprechenden Abschnitt.

Jupyter Notebooks sind eine Software, die es ermöglicht Code nicht über eine ganze Textdatei hinweg auszuführen, sondern Codeblöcke in sogenannten Zellen nacheinander auszuführen. Diese Zellen teilen sich allerding einen sog. Namespace. Das heißt, einmal in einer Zelle eingeführte Variablen gelten für das gesamte Jupyter Notebook sobald sie in einer Zelle eingeführt wurden und die Zelle ausgeführt wird.

Jeder der nachfolgenden Codeblöcke ist dazu gedacht in einer Zelle eines Jupyter Notebooks ausgeführt zu werden.

Der Gedanke hinter dem Code Along ist, dass Teilnehmende ihre Selbstwirksamkeit mit Code erfahren.

#### Ablauf des Abschnitts

Ein\*e Betreuer\*in öffnet ein Jupyter Notebook auf dem eigenen Laptop und präsentiert das Notebook über den Beamer den Teilnehmer\*innen. Die einzelnen Codeblöcke werden dann händisch von der\*dem Betreuer\*in abgetippt und ausgeführt. Das geschieht in einer Geschwindigkeit, in der die Teilnehmenden mithalten können. Die\*der präsentierende Betreuer\*in erklärt zu jedem Block inhaltlich was passiert. Fragen während der Präsentation sind erwünscht.

#### 1. Hallo Welt

|  |
| --- |
| print('Hallo Welt!')  print('Hallo Welt')  print() #Leerzeile  print('Mein Name ist Beispielname') |

In diesem Abschnitt werden mehrere **print**-Anweisungen ausgeführt. “Print“ ist eine Methode. Methoden sind abstrakte Codeblöcke, die Parameter entgegennehmen. Im Fall von Print ist das Text, ein sog. **String**. Strings sind Verkettungen von **Charactern**, also Schriftzeichen. Print nimmt Strings entgegen und verarbeitet sie so, dass sie in der Kommandozeile über **stdout** ausgegeben werden. Oder im Fall von Jupyter Notebooks unter der Zelle ausgegeben werden in dem die Print Anweisungen ausgeführt werden. Hier werden ausgeben: "Hallo Welt!" mit unterschiedlichen Anführungszeichen, und eine Leerzeile.

#### 2. Variablen

|  |
| --- |
| mein\_name = "Beispielname"  mein\_alter = 18  mein\_pi = 3.14 |

In diesem Abschnitt werden drei Variablen definiert:

* **mein\_name** wird als String mit dem Wert "Beispielname" gesetzt.
* **mein\_alter** wird als Ganzzahl mit dem Wert 18 initialisiert.
* **mein\_pi** wird als Fließkommazahl mit dem Wert 3.14 festgelegt.

Kombiniert:

|  |
| --- |
| print("Hallo, ich heiße ", mein\_name)  print(mein\_alter)  print(2 \* mein\_alter) |

* Der Name aus der Variable **mein\_name** in einem Satz ausgeben ("Hallo, ich heiße Beispielname").
* Der Wert von **mein\_alter** (18) direkt ausgeben.
* Das Doppelte von **mein\_alter** (also 36) berechnen und ausgeben.

#### 3. Kommentare

|  |
| --- |
| # Ich bin ein Kommentar  # Was im Kommentar steht wird nicht ausgeführt:  # mein\_name = "Mein Name"  print(mein\_name) # Ich bin ein inline Kommentar |

In diesem Abschnitt wird der Wert der Variable **mein\_name** ausgegeben, der zuvor als "Beispielname" festgelegt wurde. Die Zeilen, die mit **#** beginnen, sind Kommentare und haben keinen Einfluss auf die Ausführung des Codes. Kommentare dienen der Nachvollziehbarkeit für Personen, die den Code lesen. So wird in Kommentaren oft die Intention hinter einem Codeabschnitt beschrieben, oder Angaben zur Autor\*innenschaft hinterlegt. Es kann aber auch nützlich sein, Kommentare dafür zu nutzen, vorher geschriebene Codezeilen inaktiv zu machen, ohne sie zu löschen. Im Beispiel bleibt **mein\_name** unverändert und "Beispielname" wird ausgegeben. Der Text nach dem print-Befehl ist ebenfalls ein Kommentar und beeinflusst die Ausführung nicht.

#### 4. Datentypen

|  |
| --- |
| print(type(mein\_name))  print(type(mein\_alter))  print(type(mein\_pi))  mein\_bool = 1 > 2  print(type(mein\_bool)) |

In diesem Abschnitt werden die Datentypen der zuvor definierten Variablen **mein\_name**, **mein\_alter** und **mein\_pi** ausgegeben. Anschließend wird eine neue Variable **mein\_bool** definiert, die das Ergebnis des Vergleichs **1 > 2** speichert, und der Typ von **mein\_bool** wird ebenfalls ausgegeben.

* **type(mein\_name)** gibt den Typ von **mein\_name** aus, der ein String ist. Strings haben wir bereits eingeführt.
* **type(mein\_alter)** gibt den Typ von **mein\_alter** aus, der eine Ganzzahl (Integer) ist. Ganzzahlige Werte sind alle Werte, die keine Nachkommastelle haben. Integer sind in ihrem Zahlenraum durch die Speichergröße begrenzt. In Python sind Standard-Integer 32 Bit lang und bilden den Zahlenraum –231 bis +231 ab.
* **type(mein\_pi)** gibt den Typ von **mein\_pi** aus, der eine Fließkommazahl (Float) ist. Fließkommazahlen sind in Python 64 Bit lang und bilden den Zahlenraum 1,7×10–308 bis 1,7×10308 ab.
* **type(mein\_bool)** gibt den Typ von **mein\_bool** aus, der ein Boolean ist, da **1 > 2** eine boolesche Bedingung ist (deren Ergebnis **False** ist). Boolsche Werte in der Programmierung repräsentieren binäre Werte, typischerweise **wahr** und **falsch**. Sie werden verwendet, um den Fluss der Logik im Code zu steuern, wie zum Beispiel in bedingten Anweisungen und Schleifen. In Python sind die booleschen Werte **True** und **False**, und sie sind eine Unterart von Ganzzahlen, wobei **True** 1 und **False** 0 entspricht. Boolsche Werte helfen bei der Entscheidungsfindung, der Auswertung von Bedingungen und der Bestimmung des Ergebnisses logischer Operationen in Softwareanwendungen.

Mischen von Datentypen:

|  |
| --- |
| ergebnis0 = mein\_alter + mein\_alter  print(ergebnis0) |

In diesem Abschnitt wird **ergebnis0** zunächst als die Summe von **mein\_alter** (18) und **mein\_alter** berechnet und ausgegeben, also 36.

­­

|  |
| --- |
| # fails with unsupported operand type(s) for +: 'int' and 'str'  ergebnis1 = mein\_alter + mein\_name |

Der Versuch, **mein\_alter** und **mein\_name** zu addieren, scheitert, da man eine Zahl nicht mit einem String addieren kann.

|  |
| --- |
| ergebnis2 = mein\_alter + mein\_pi  print(ergebnis2)  print(type(ergebnis2)) |

Dann wird **ergebnis2** als die Summe von **mein\_alter** (18) und **mein\_pi** (3.14) berechnet und ausgegeben, also 21.14. Der Typ von **ergebnis2** wird als Fließkommazahl (**float**) ausgegeben.

#### 5. Operatoren

|  |
| --- |
| print( 5 + 999 )  print( 100 - 50 )  print( 10 / 2 )  print( 3 \* 2 )  print( 2 \* 3 - 4 ) |

In diesem Abschnitt werden einfache mathematische Operationen ausgeführt und die Ergebnisse ausgegeben. Mathematische Operatoren sind Symbole oder Zeichen, die in mathematischen Ausdrücken verwendet werden, um Operationen zwischen Zahlen oder Variablen durchzuführen. Hier sind die grundlegenden mathematischen Operatoren:

* Addition (+), Beispiel: (5 + 999) ergibt 1004.
* Subtraktion (-), Beispiel: (100 - 50) ergibt 50.
* Division (/), Beispiel: (10 / 2) ergibt 5.
* Multiplikation (\*), Beispiel: (3 \* 2) ergibt 6
* Mischung verschiedener Operatoren folgt der Prioritätsregel. Beispiel: (2 \* 3 - 4) ergibt 2.

#### 6. Listen

|  |
| --- |
| mein\_pi = 3.14  print(mein\_pi) #out: 3.14  eine\_liste = [1,2,3,4]  print(eine\_liste) #out: [1, 2, 3, 4]  noch\_eine\_liste = ["Hallo", "Test", "String"]  print(noch\_eine\_liste) #out: ['Hallo', 'Test', 'String']  gemischte\_datentypen = ["Hallo", 1, mein\_pi, True] #out: [  print(gemischte\_datentypen) #out: ['Hallo', 1, 3.14, True]  print(len(gemischte\_datentypen)) #out: 4  print(type(gemischte\_datentypen)) #out: <class 'list'>  print(gemischte\_datentypen[0]) #out: Hallo  print(gemischte\_datentypen[3]) #out: True |

In diesem Abschnitt werden Listen und deren Eigenschaften behandelt:

* Es werden drei Listen **eine\_liste**, **noch\_eine\_liste** und **gemischte\_datentypen** erstellt und ausgegeben.
* Die Liste **gemischte\_datentypen** enthält verschiedene Datentypen, einschließlich eines Strings, einer Zahl, des Wertes von **mein\_pi** und eines Booleschen Wertes.
* Die Funktion **len()** gibt die Länge von **gemischte\_datentypen** als Integer (Ganzzahl) aus.
* Der Datentyp bzw. die Klasse der Liste **gemischte\_datentypen** wird als **list** ausgegeben.
* Die Elemente an den Positionen 0 und 3 von **gemischte\_datentypen** werden ausgegeben, was "Hallo" und **True** entspricht.

**7. Boolesche Operatoren**

|  |
| --- |
| alter\_nachbar = 18  print(mein\_alter > alter\_nachbar)  print(mein\_alter < alter\_nachbar)  print(mein\_alter == alter\_nachbar) |

In diesem Abschnitt werden Vergleichsoperationen zwischen verschiedenen Variablen durchgeführt. Die ersten beiden Zeilen vergleichen **mein\_alter** mit **alter\_nachbar** hinsichtlich Kleiner-als und Gleichheit.

|  |
| --- |
| # fails with: > not supported between instances of 'str' and 'int'  print(mein\_alter > mein\_name) |

In diesem Abschnitt wird ein Vergleich zwischen einem Integer (**mein\_alter**) und einem String (**mein\_name**) durchgeführt, was zu einem Fehler oder einem undefinierten Verhalten führt. Diese Operation gilt der Demonstration von undefiniertem Verhalten. Es ist schlicht nicht möglich eine Ganzzahl bzw. einen Integer (**mein\_alter**) als größer (**>**) Zeichenkette (**mein\_name**) festzustellen.

**8. If / Else (Wenn-Dann)**

|  |
| --- |
| if bedingung1:  # code block A  elif bedingung2:  # code block B  else:  # code block C |

Eine **if-else** (wenn-dann) Anweisung ist ein grundlegendes Konzept in der Programmierung, das es ermöglicht, Entscheidungen basierend auf bestimmten Bedingungen zu treffen und unterschiedliche Codeblöcke auszuführen. Hier eine ausführliche Erklärung dieses Konzepts:

Eine **if-else** Anweisung überprüft eine Bedingung und führt entsprechend dem Ergebnis dieser Überprüfung verschiedene Codeblöcke aus. Die Struktur lässt sich wie folgt zusammenfassen:

1. **If-Anweisung**: Überprüft eine Bedingung. Wenn die Bedingung wahr ist, wird der zugehörige Codeblock ausgeführt.
2. **Else-Anweisung**: Wird ausgeführt, wenn die Bedingung der if-Anweisung falsch ist.
3. **Elif-Anweisung** (optional): Steht für "else if" und wird verwendet, um zusätzliche Bedingungen zu prüfen, falls die vorherigen Bedingungen falsch sind.

|  |
| --- |
| if mein\_alter < alter\_nachbar:  print("Anna ist älter als ich")  else:  print("Ich bin älter als Anna")  wochentag = "Samstag"  if wochentag == "Freitag":  print("Juhu, endlich Freitag :)")  else:  print("Leider kein Freitag") |

In diesem Abschnitt finden zwei separate **if-else**-Anweisungen (wenn-dann) statt:

1. Vergleich von**mein\_****alter** und**alter\_nachbar**: Hier wird überprüft, ob **mein\_alter** kleiner als **alter\_nachbar** ist. Wenn das zutrifft, wird "Anna ist älter als ich" ausgegeben. Andernfalls wird "Ich bin älter als Anna" ausgegeben. Dies setzt voraus, dass **mein\_alter** und **alter\_nachbar** numerische Werte sind und dass Anna das **alter\_nachbar** repräsentiert.
2. Überprüfung des Wochentags: Es wird geprüft, ob der Inhalt der Variable **wochentag** (hier "Samstag") gleich einem Wert "Freitag" ist. Wenn **wochentag** tatsächlich "Freitag" ist, wird "Juhu, endlich Freitag :)" ausgegeben. Da **wochentag** "Samstag" ist, wird hier "Leider kein Freitag" ausgegeben.

**9. For-Schleife**

|  |
| --- |
| früchte = ["Apfel", "Kirsche", "Birne", "Apfel", "Pfirsich", "Kiwi"]  for frucht in früchte:  print(frucht)    # mit range():  for i in range(0,6):  print(i , früchte[i])  # Mit if-verknüpft:  for frucht in früchte:  if frucht == "Pfirsich":  print("Der Pfirsich wurde gefunden")  print(frucht) |

In diesem Abschnitt werden For-Schleifen eingeführt:

1. Eine **for**-Schleife, die über die Liste **früchte** iteriert und jede Frucht in der Liste ausgibt.
2. Zwei weitere **for**-Schleifen: eine mit **range()**, die Indizes und Früchte ausgibt, und eine, die überprüft, ob eine Frucht ein Pfirsich ist, und in diesem Fall eine spezielle Nachricht ausgibt, sowie alle Früchte aus der Liste.

**10. While-Schleife**

|  |
| --- |
| i = 1  while i < 6:  print(i)  i += 1 |

In diesem Code-Abschnitt wird eine **while**-Schleife verwendet, die bei **i = 1** startet und läuft, bis **i** den Wert 6 erreicht. Innerhalb der Schleife wird der Wert von **i** in jeder Iteration gedruckt und dann um 1 erhöht. Die Ausgabe wird die Zahlen 1, 2, 3, 4 und 5 nacheinander sein.

Der Unterschied zwischen einer **for**-Schleife und einer **while**-Schleife besteht hauptsächlich in ihrer Verwendung und Struktur. Eine for-Schleife wird verwendet, wenn die Anzahl der Iterationen im Voraus bekannt ist und durchläuft eine Sequenz oder eine Reihe von Werten. Im Gegensatz dazu wird eine **while**-Schleife verwendet, wenn die Anzahl der Iterationen nicht im Voraus bekannt ist und die Schleife so lange laufen soll, wie eine bestimmte Bedingung wahr ist.

Frage an die Teilnehmenden: wie sieht die For-schleife mit äquivalentem Output aus?

|  |
| --- |
| for i in range(1,6):  print(i) |

**Lösung**: Es wird eine **for**-Schleife mit der Funktion **range(1,6)** verwendet, die von 1 bis 5 läuft (6 ist ausschließend). In jeder Iteration wird der aktuelle Wert von **i** ausgegeben. Die Ausgabe wird die Zahlen 1, 2, 3, 4 und 5 nacheinander sein.

#### 11. String Operatoren

|  |
| --- |
| mein\_string = "Hallo, mein Name ist Johan"  print(len(mein\_string))  print(mein\_string.upper())  print(mein\_string.lower()) |

In diesem Abschnitt wird mit dem String **mein\_string** gearbeitet:

* Die Länge des Strings wird ausgegeben (**len(mein\_string)** ergibt die Anzahl der Zeichen).
* Der String wird in Großbuchstaben umgewandelt und ausgegeben (**mein\_string.upper()**).
* Der String wird in Kleinbuchstaben umgewandelt und ausgegeben (**mein\_string.lower()**).

#### 12. Funktionen

|  |
| --- |
| def myFunc(var1, var2):  result = var1 + var2  return result    print(myFunc(1,2)) |

In diesem Abschnitt wird eine Funktion **myFunc** definiert, die zwei Parameter (**var1** und **var2**) übernimmt, ihre Summe berechnet und zurückgibt (**return**).

Der **return**-Befehl in einer Funktion beendet die Ausführung der Funktion und gibt einen Wert zurück an den Aufrufer. Dieser Wert kann ein einfaches Datenobjekt wie eine Zahl oder ein String, eine komplexere Datenstruktur wie eine Liste oder ein Dictionary oder sogar None sein, wenn kein Wert angegeben wird. Der **return**-Befehl wird verwendet, um das Ergebnis einer Berechnung oder einer Operation aus der Funktion heraus zu übermitteln, sodass der aufrufende Code diesen Wert weiterverwenden kann.

Die Funktion wird dann mit den Werten 1 und 2 aufgerufen, und das Ergebnis (3) wird ausgegeben.

### Abschnitt 4: Aufgaben (150min)

In diesem Abschnitt befinden sich Übungsaufgaben, die die in Abschnitt 3 vorgestellten Konzepte anwenden. Insgesamt befinden sich hier 3 Aufgaben, die von Aufgabe 1 zu Aufgabe 3 jeweils mehr Zeit in Anspruch nehmen. Sollte der Kurs für einen kürzeren Zeitraum ausgelegt sein, beispielsweise 3 Stunden, bietet es sich an nur eine oder zwei der Aufgaben zu machen.

#### Projekt 1: Summe berechnen

Schreibe ein Programm, dass zwei Integer miteinander addiert und dafür Funktion verwendet.

Beispiellösung:

|  |
| --- |
| def addiere\_zwei\_zahlen(zahl1, zahl2):  return zahl1 + zahl2  # Beispiel für die Verwendung der Funktion  ergebnis = addiere\_zwei\_zahlen(5, 3)  print(ergebnis) |

In diesem Code wird eine Funktion namens **addiere\_zwei\_zahlen** definiert, die zwei Ganzzahlen (**zahl1** und **zahl2**) als Parameter nimmt und ihre Summe zurückgibt. Dann wird die Funktion mit den Werten 5 und 3 aufgerufen, und das Ergebnis der Addition, also 8, wird ausgegeben.

**Projekt 2: Tannenbaum**

Schreibe ein Programm, das das folgende ausgibt:

|  |
| --- |
| \*  \*\*\*  \*\*\*\*\*  \*\*\*\*\*\*\*  \*\*\*\*\*\*\*\*\*  \*\*\* |

Beispiellösung:

|  |
| --- |
| def generate\_christmas\_tree(levels):  max\_width = max(levels)  tree = ""  for level in levels:  stars = '\*' \* level  padding = (max\_width - level) // 2  tree += ' ' \* padding + stars + '\n'  return tree  levels = [1, 3, 5, 7, 9, 3]  christmas\_tree = generate\_christmas\_tree(levels)  print(christmas\_tree) |

Der Python-Code generiert einen Weihnachtsbaum basierend auf einer Liste, die die Anzahl der Sterne (\*) für jede Ebene des Baumes angibt.

Funktion **generate\_christmas\_tree**(**levels**): Diese Funktion nimmt ein Array **levels** entgegen, das die Anzahl der Sterne für jede Ebene des Weihnachtsbaumes repräsentiert.

1. **Bestimmung der maximalen Breite**: Zuerst ermittelt der Code die maximale Breite des Baumes mit **max(levels)**. Diese Breite wird verwendet, um die Sternzeilen zentriert auszurichten.
2. **Aufbau des Baumes**: Der Baum wird in einer Schleife aufgebaut, die über das levels-Array iteriert. Für jede Ebene werden folgende Schritte durchgeführt:
   * **Sternzeile erstellen**: Eine Zeile von Sternen (**\***) wird basierend auf der Anzahl im levels-Array erstellt.
   * **Berechnung des Leerzeichens**: Für jede Ebene wird der notwendige Leerraum (Leerzeichen) berechnet, um die Sterne zentriert auszurichten. Dies geschieht durch **(max\_width - level) // 2**, was den Leerraum auf jeder Seite der Sternzeile bestimmt.
   * **Hinzufügen von Leerzeichen und Sternen**: Der Leerraum und die Sternzeile werden zusammengefügt und an die tree-Stringvariable angehängt.
3. **Rückgabe des Baumes**: Nachdem alle Ebenen verarbeitet wurden, gibt die Funktion den kompletten Weihnachtsbaum als String zurück.
4. **Ausgabe des Weihnachtsbaumes**: Schließlich wird das levels-Array **[1, 3, 5, 7, 9, 3]** an die Funktion übergeben, und der resultierende Weihnachtsbaum wird ausgegeben.

Das Ergebnis ist ein Weihnachtsbaum, der mit Sternen entsprechend den Angaben im levels-Array aufgebaut ist und in jeder Ebene zentriert ausgerichtet ist.

**Projekt 3: Morsecode**

Schreibe ein Programm, das Text in Morsecode übersetzt.

Code zu übersetzen:

|  |
| --- |
| morse\_code = '.... . .-.. .-.. --- .-- --- .-. .-.. -..' |

Beispiellösung:

|  |
| --- |
| def morse\_to\_text(morse\_code):  morse\_dict = {  '.-': 'A', '-...': 'B', '-.-.': 'C', '-..': 'D', '.': 'E',  '..-.': 'F', '--.': 'G', '....': 'H', '..': 'I', '.---': 'J',  '-.-': 'K', '.-..': 'L', '--': 'M', '-.': 'N', '---': 'O',  '.--.': 'P', '--.-': 'Q', '.-.': 'R', '...': 'S', '-': 'T',  '..-': 'U', '...-': 'V', '.--': 'W', '-..-': 'X', '-.--': 'Y',  '--..': 'Z', '-----': '0', '.----': '1', '..---': '2', '...--': '3',  '....-': '4', '.....': '5', '-....': '6', '--...': '7', '---..': '8',  '----.': '9'  }  # Aufteilung des Morse-Codes in Wörter und Zeichen  words = morse\_code.split(' ') # Drei Leerzeichen trennen Wörter  decoded\_text = []  for word in words:  characters = word.split(' ') # Ein Leerzeichen trennt Buchstaben  decoded\_word = ''.join(morse\_dict[char] for char in characters if char in morse\_dict)  decoded\_text.append(decoded\_word)  return ' '.join(decoded\_text)  # Beispiel für die Verwendung der Funktion  morse\_code = '.... . .-.. .-.. --- .-- --- .-. .-.. -..'  text = morse\_to\_text(morse\_code)  print(text) |

Beschreibung der Lösung:

1. Morsecode-Lexikon (**morse\_dict**): Dieses Wörterbuch (**dict**) ordnet jedem Morsecode-Zeichen (als Schlüssel) einen entsprechenden Buchstaben oder eine Zahl (als Wert) zu. In Python ist ein dict (Wörterbuch) eine Sammlung, die ungeordnet, veränderbar und indiziert ist. In morse\_dict repräsentieren die Schlüssel Morsecode-Zeichen (z.B. '.-' für 'A') und die Werte sind die entsprechenden alphanumerischen Zeichen.
2. Die Funktion **morse\_to\_text**: Diese Funktion nimmt einen Morsecode-String als Eingabe und verwendet das **morse\_dict**, um ihn in alphanumerischen Text umzuwandeln.
3. Aufteilung des Morsecode-Strings:
   * Der Morsecode-String wird zuerst in Wörter aufgeteilt. In Morsecode werden Wörter durch drei Leerzeichen (' ') getrennt.
   * Jedes Wort wird dann weiter in einzelne Morsecode-Zeichen aufgeteilt, die durch ein Leerzeichen (' ') voneinander getrennt sind.
4. Dekodierung: Jedes Morsecode-Zeichen eines Wortes wird anhand des **morse\_dict** in den entsprechenden alphanumerischen Charakter umgewandelt. Jedes Wort wird dann zu einem vollständigen alphanumerischen Wort zusammengesetzt.
5. Ergebnis: Die dekodierten Wörter werden zu einem vollständigen Satz zusammengefügt und zurückgegeben.  
     
   Das Programm endet mit einem Beispiel, in dem ein Morsecode-String (**'.... . .-.. .-.. --- .-- --- .-. .-.. -..'**) in Text (**'HELLO WORLD'**) umgewandelt und ausgegeben wird.   
     
   Die Verwendung eines Wörterbuchs ist hier besonders effektiv, da es eine schnelle und direkte Zuordnung von Morsezeichen zu alphanumerischen Zeichen ermöglicht.