Grundlagen

1 Herzlich Willkommen!

Wir freuen uns sehr, dass ihr Lust habt bei unserem Kurs mitzumachen! Dieses Dokument ist in drei Teile aufgeteilt. Der erste Teil beinhaltet die Grundlagen mit Erklärungen, vielen Beispielen und kleinen Aufgaben. Der zweite Teil beinhaltet Aufgaben um diese Grundlagen festigen zu können. Falls ihr Lust auf komplexere Aufgaben habt, findet ihr diese im dritten Teil. Im letzten Teil findet ihr herausfordernde Projekte mit denen ihr richtig viel anfangen könnt. Stellt uns bitte alle möglichen Fragen, dafür sind wir da! Am Ende des Kurses stellen wir euch die Musterlösung für die Aufgaben bereit. Falls ihr also Lust habt weiter daran zu arbeiten, könnt ihr dort nachsehen, sollte es mal haken.

2 Das Terminal

`/Aufgabe_Terminal.py`

Zu allererst sollt ihr das Terminal kennen lernen. Das Terminal ist ein simples Programm zur Darstellung und Bearbeitung von Dateien auf deinem Computer. Im Gegensatz zu vielen anderen Anwendungen, wie zum Beispiel einem Internetbrowser, hat das Terminal keine grafische Oberfläeche. Das Terminal hat viele verschiedene Bezeichnungen, es wird auch oft als cmd, Prompt, Konsole, Shell, Bash oder Kommandozeile bezeichnet. Es ist wichtig zu wissen, dass sich die Befehle, also das was ihr in das Terminal schreibt, je nach Betriebssystem unterscheiden. In diesem Kurs arbeitet ihr mit Rechnern, die Linux als Betriebssysstem verwenden, weshalb wir uns im Folgenden mit den Befehlen für Linux Rechner beschäftigen.

Schritt 1: Terminal öffnen
Im ersten Schritt wollen wir das Terminal öffnen. Klickt dazu auf die Maus in der Ecke links unten, gebt *Terminal* in die Suche ein und wählt das Xfce Terminal aus. Es sollte sich nun ein schwarzes Fenster, nämlich das Terminal öffnen. In der ersten Zeile des Terminals steht euer Benutzer*in-Name, ein @-Zeichen und dann der Name des Rechners auf dem ihr eingeloggt seid.

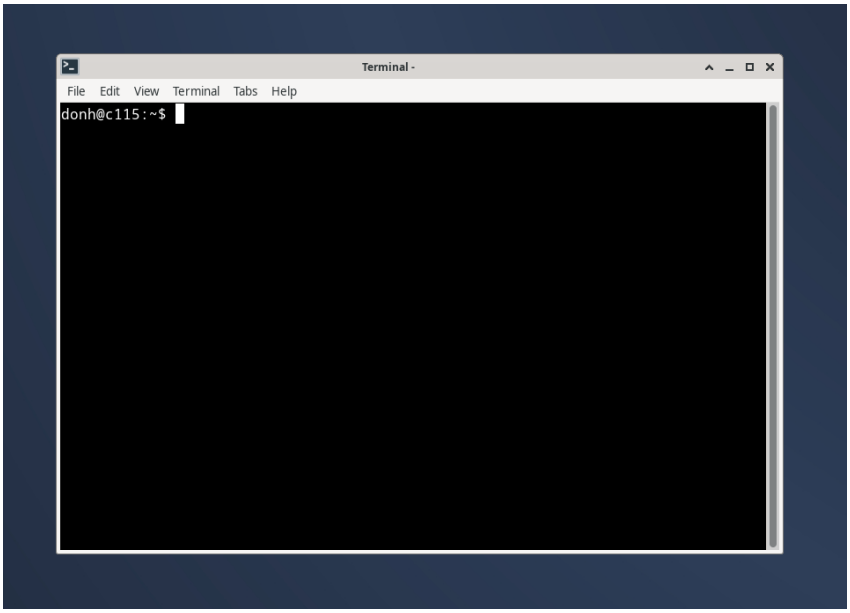


Figure 1: Hier seht ihr das Terminal. Die Benutzerin mit Benutzerin-Namen *donh* arbeitet an einem Rechner dessen Name *c115* ist.

Schritt 2: Programme mit Hilfe des Terminals ausführen
Wir können mit Hilfe des Terminals Programme ausführen. Um Programme ausführen zu können, müssen wir zuest an den Speicherort des Programms gelangen. Diese können an ganz verschiedenen Orten gespeichert sein, also zum Beipsiel auf dem Desktop, in Downloads etc. Folgende Befehle können wir im Terminal

eingeben, um zum Speicherort zu gelangen und die Datei zu öffnen:

- `ls`: Gebt diesen Befehl ein und drückt dann die Enter-Taste, so wird euch ein akutelles Inhaltsverzeichnis ausgeben. Das Inhaltsverzeichnis enthält alle Unterverzeichnisse, zu denen ihr akutell gelangen könnt.
- `cd`: Gebt ihr diesen Befehl ein und drückt dann die Enter-Taste, so könnt ihr zu einem Unterverzeichnis gelangen, welches euch im Inhaltsverzeichnis angezeigt wurde. Eine mögliche Eingabe wäre beispielsweise `cd Desktop`. Wir befinden uns nun im Speicherort Desktop.
- `cd ..`: Mit diesem Befehl könnt ihr zurück in das Überverzeichnis wechseln. Befindet ihr euch aktuell im Speicherort Desktop, so könnt ihr mit der Eingabe `cd ..` in das Überverzeichnis zurück.
- Zum Ausführen eines Python-Programms, begebt euch in das Verzeichnis, wo das Programm liegt, gebt python3, Leerzeichen, und dann den Programmnamen ein und drückt die Enter-Taste. Mit der Eingabe `python3 Beispiel.py` führt ihr das Programm mit dem Namen Beispiel aus. Sind Programme mit der Programmiersrpache Python geschrieben, so sind sie mit der Endung .py abgespeichert.

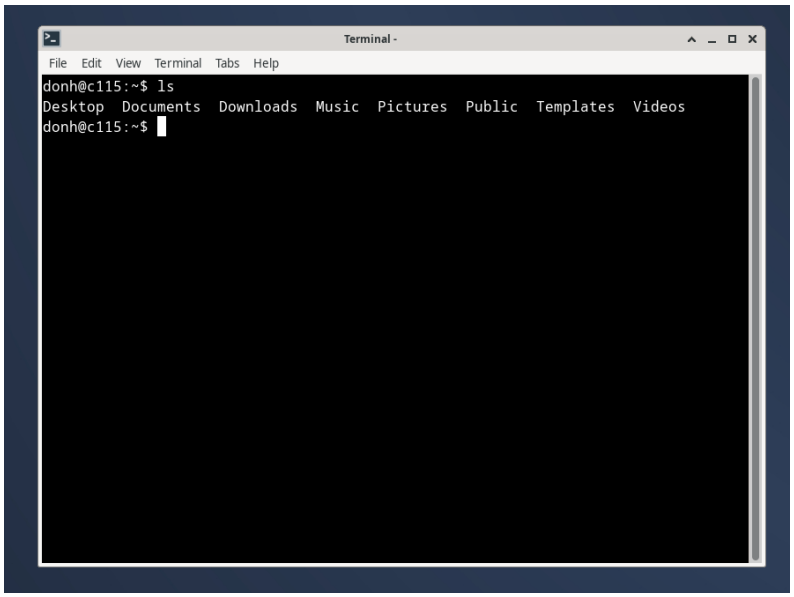


Figure 2: Mit dem Befehl `ls` wird das akutelle Inhaltsverzeichnis angezeigt.

2.1 Aufgabe

Führt das Programm mit dem Namen `Aufgabe_Terminal.py` aus. Die gesuchte Datei befindet sich im Ordner `01-Grundlagen`, dieser befindet sich im Ordner `Aufgaben`, welcher sich im Ordner `Starcode` befindet. Der Ordner Starcode ist im Speicherort Desktop.

3 Variablen - Zuweisung und Verknüpfung

3.1 Theorie

Hier sollt ihr euch mit den verschiedenen Variablentypen und deren Verknüpfung vertraut machen. Eine Variable ist im allgemeinen ein Behälter zur Aufbewahrung von bestimmten Werten. Man kann im Verlauf des Programms auf diese Variablen, oder genauer: auf ihren Inhalt zugreifen, oder ihnen einen neuen Wert zuweisen.

Im Folgenden sind teilweise Beispiele angegeben, die ihr im Terminal überprüfen könnt. Gebt dazu im Terminal `python3` ein, und drückt dann die Enter-Taste, um die Python-Umgebung zu aktivieren.

Zuweisung von Variablen
Eine Variable muss immer einen eindeutigen Namen haben sowie eine initiale Zuweisung. Die Zuweisung geschieht mit '=', wobei der Variablenname auf der linken Seite des Zeichens steht, und der Wert, der ihr zugewiesen wird rechts. Eine Zuweisung lässt sich beliebig häufig ändern. Im Beispiel `x = 2` weisen wir der Variable x den Wert 2 zu.

Verschiedene Variablentypen
Es gibt viele verschiedene Variablentypen. Wir stellen hier die gängigsten Variablentypen kurz vor.

- `int`: Mit dem Variablentyp `int` (dies steht für Integer) können wir ganze Zahlen abspeichern. Im Beispiel `ganzeZahl = 2` weisen wir der Variable mit dem Namen *ganzeZahl* den Wert 2 zu. Die Variable *ganzeZahl* hat also den Variablentyp `int`.
- `float`: Mit dem Variablentyp `float` können wir Kommazahlen abspeichern. Im Beispiel `kommaZahl = 9.2` weisen wir der Variable mit dem Namen *kommaZahl* den Wert 9.2 zu. Die Variable *kommaZahl* hat also den Variablentyp `float`.
- `string`: Mit dem Variablentyp `string` können wir sogenannte Zeichenketten abspeichern. Jedes beliebige Zeichen kann als string abgespeichert werden. Strings sind entweder mit doppelten oder einfachen Anführungszeichen gekennzeichnet. Im Beispiel `y = 'huhu'` weisen wir der Variable mit dem Namen y den Wert $'huhu'$ zu. Die Variable y hat den Variablentyp `string`. Auch bei der Variable `zeichenkette = "3"` handelt es sich um eine Variable des Variablentyps `string`.
- `bool`: Mit dem Variablentyp `bool` können wir sogenannte Wahrheitswerte, also `True` und `False` abspeichern. Diese werden auch bool'sche Variablen genannt. Im Beispiel `istDieBananeGerade=False` weisen wir der Variable *istdieBananeGerade* den Wahrheitswert $False$ zu.

Variablentypen herausfinden
Ist euch der Typ einer gegebenen Variable nicht bekannt, so könnt ihr diesen mit `type()` herausfinden. Schreibt ihr nacheinander die folgenden Zeilen in das Terminal, so gibt euch `<class 'int'>` zurück.

```
z = 1
type(z)
```

Variablentypen umwandeln
Ihr könnt eine gegebene Variable in einen anderen Variablentyp umwandeln. Schreibt ihr nacheinander die folgenden Zeilen in das Terminal, so gibt euch Python `<class 'int'>` zurück.

```
a = '4'
a = int(a)
type(int(a))
```

Dies funktioniert auch umgekehrt: Habt ihr eine Variable vom Variablentyp `int` so könnt ihr diese mit `str()` in einen string umwandeln. Das Umwandeln von Variablentypen ist auch für andere Variablentypen möglich.

Operatoren
Ihr könnt Variablen mit sogenannten Operatoren verknüpfen. Ihr erhaltet dann einen Wert, den ihr in einer neuen Variable speichern könnt, falls ihr in wieder verwenden möchtet. Wir unterscheiden im Folgenden zwischen Vergleichsoperatoren und Rechenoperatoren.

Vergleichsoperatoren
Wir stellen wir euch die gängigsten Vergleichsoperatoren vor. Verknüpfen wir Variablen mit Vergleichsoperatoren, so erhalten wir einen Wahrheitswert als Ergebnis.

- `==`: Dieser Operator prüft, ob links und rechts dasselbe steht. Dies ist für alle Datentypen möglich. Schreibt ihr nacheinander die folgenden Zeilen in das Terminal, so wird euch der Wahrheitswert `False` ausgegeben.

```
a = 3
b = 4
a == b
```

- `!=`: Dieser Operator prüft, ob links und rechts *nicht* gleich sind.
- `>=`, `<=`, `>`, `<`: Diese Operatoren funktionieren nur für die Datentypen `int` und `float`.
- `and`: Dieser Operator prüft ob die linke Seite *und* die rechte Seite wahr sind. Schreibt ihr nacheinander die folgenden Zeilen in das Terminal, so wird euch der Wahrheitswert `False` ausgegeben.

```
c = True
d = False
c and d
```

- `or`: Dieser Operator prüft, ob die linke Seite *und/oder* die rechte Seite wahr sind.

Rechenoperatoren
Wir stellen euch hier kurz die bekannten Rechenoperatoren vor.

- `+`: Dies ist der bekannte Additionsoperator. Wir können mit diesem Operator aber auch Strings aneinanderhängen. Schreibt ihr nacheinander die folgenden Zeilen in das Terminal, so wird euch `'huhuna'` ausgegeben.

```
u = 'huhu'
y = 'na'
u + y
```

- `i+=1`: Dies ist äquivalent zu $i = i + 1$.
- `i-=1`: Dies ist äquivalent zu $i = i - 1$. Ist das Subtrahieren, wie ihr es kennt, geht für `int` und `float`.

Weitere Rechenoperatoren findet ihr auf dem cheatsheet oder im weiteren Verlauf dieses Kurses.

3.2 Aufgabe

Im folgenden sollt ihr fünf kleine Aufgaben bearbeiten, in denen ihr Variablen unterschiedlichen Typs definieren und euch im Terminal ausgeben lassen sollt.

1. Definiert fünf Variablen, je eine int-, float- und bool'sche- sowie zwei String-Variablen und lasst sie euch nacheinander auf dem Terminal ausgeben.
2. Addiert die beiden String-Variablen zusammen und addiert auf eure int-Variable `7` drauf. Lasst euch auch diese Ergebnisse im Terminal ausgeben.
3. Überlegt euch einen Vergleich, der den selben Wahrheitswert wie eure bool'sche Variable hat. Schaut, was euch im Terminal ausgegeben wird, wenn ihr den Wahrheitswert des Vergleichs und den eurer bool'schen Variable mit dem `and`-Operator verknüpft. Falls noch nicht der Fall, verändert eure bool'sche Variable so, dass der Wahrheitswert `False` ausgegeben wird.

4 Print und Input

`Aufgabe_PrintInput.py`

4.1 Theorie

Wir stellen die Funktion `print()` und `input()` vor. Bis jetzt habt ihr im Terminal programmiert. Wollt ihr größere Programme schreiben, so könnt ihr dies in einem Text-Editor tun, in diesem Kurs benutzt ihr den Text-Editor Geany. Damit euch euer Programm dann etwas ausgibt, könnt ihr die `print()` Funktion benutzen.

- `print()`: Mit der print-Funktion könnt ihr bestimmte Nachrichten ausgeben. Dabei wird der Datentyp der ausgegebenen Nachricht immer in den Datentyp String konvertiert. Das Programm mit dem Code

  ```
  t = 'Hallo'
  print(t)
  ```

  gibt `'Hallo'` aus. Ihr könnt auch mehrere Dinge gleichzeitig printen lassen. Das Programm mit dem Code

  ```
  a = 1
  b = 2
  c = 3
  print(a,b,c)
  ```

  gibt `1 2 3` aus. Ihr könnt Variablen in Sätze einbinden und diese printen. Das Programm mit dem Code

  ```
  a = 2
  print('Meine Lieblingszahl ist ' + str(a) + '.')
  ```

  gibt den Satz `Meine Lieblingszahl ist 2.` zurück. Hier ist zu beachten, dass ihr die Variable a, die den Variablentyp `int` hat, zuerst in einen String `str(a)` umwandeln müsst.

- `input()`: Die input-Funktion ermöglicht eine Eingabe der User*in über die Tastatur. Ihr solltet die Eingabe als Variable abspeichern. Die Eingabe wird als String gespeichert, ihr müsst den Variablentyp also gegebenenfalls umwandeln, je nach wie ihr die Eingabe weiter verarbeitet. Das Programm mit dem Code

  ```
  Eingabe = input("Was ist deine Lieblingsfarbe?")
  print('Deine Lieblingsfarbe ist ' + Eingabe + '.')
  ```

  fragt euch nach eurer Lieblingsfarbe. Habt ihr diese eingegeben, wird euch der Satz mit eurer Eingabe ausgegeben.

4.2 Aufgabe

Schreibt nun im Text-Editor Geany ein kurzes Programm, welches euch nach euren Namen, eurer Lieblingsfarbe und euren Hobbies fragt. Das Programm soll diese Infos im Terminal ausgeben, ohne, dass ihr die Infos direkt in den Code schreibt. Nutzt die Funktionen `print()` und `input()`. Um euer Programm auszuführen, müsst ihr zuerst, wie in Aufgabe 1, mit dem Terminal in den richtigen Ordner navigieren.

5 If, Else, Elif-Bedingungen

`Aufgabe_if-else.py, Aufgabe_PW-Check.py`

5.1 Theorie

Für fast alle Programmiersprachen gibt es die Möglichkeiten Bedingungen zu überprüfen und entsprechend im Programmablauf darauf zu reagieren. Wenn (`if`) eine bestimmte Bedingung erfüllt ist, könnt ihr festlegen, was das Programm weiter tun soll. Ist diese Bedingung nicht erfüllt (`else`), könnt ihr dem Programm sagen, dass es etwas anderes tun soll. Das Programm mit dem Code

```
a = 3
b = 7
    if (a == b):
        print('Die Zahlen a und b sind gleich.')
    else:
        print('Die Zahlen a und b sind nicht gleich.')
```

gibt euch `'Die Zahlen a und b sind gleich.'` zurück, wenn die Zahlen gleich sind. Sind die Zahlen nicht gleich, so gibt euch das Programm `'Die Zahlen a und b sind nicht gleich.'` zurück. Ihr könnt belieblig viele `if`-Bedingungen abfragen, aber nur eine else Bedingung. Wenn ihr abfragen wollt, ob jeweila nur eine der Bedingungen wahr ist, so könnt ihr dies mit `elif` tun. Dies ist genauer auf dem cheatsheet beschrieben, dass ihr ebenfalls im Starcode Ordner findet. Im Allgemeinen ist es wichtig, die Anweistungen, die nach `if`, `else` und `elif` kommen, einzurücken. Dies geschieht in dem ihr am Anfang der Zeile die Tab-Taste drückt. Ihr seht in Zeile 3 und 4 des obenstehenden Beispiels, was damit gemeint ist.

5.2 Aufgabe

Schreibt ein Programm, welches die drei untenstehenden Teilaufgaben erfüllt. Benutzt dafür die Datei `Aufgabe_if-else.py`. Beachtet, dass Python jede Eingabe, die mit `input()` getätigt wird, als *String* auffasst. Also vergesst nicht die Eingabe jeweils für die Teilaufgabe umzuwandeln.

1. Die User*in soll zu der Eingabe der Postleitzahl aufgefordert werden. Das Programm soll anhand dieser entscheiden, ob die Postleitzahl in Berlin ist und dementsprechend antworten (Berliner Postleitzahlen liegen im Bereich 10115 bis 14199).

2. Als nächstes sollen drei Zahlen durch die User*in eingegeben werden. Euer Programm soll dann die größte der drei Zahlen ausgeben. Ihr könnt `if`-Anweisungen ineinander schachteln. Werft dazu einen Blick auf das Cheat-Sheet.

3. Zuletzt soll euer Programm die User*in fragen, ob sie Kuchen mag. Antwortet die User*in mit ja, gebt etwas nettes aus. Antwortet sie mit 'nein', gebt `'ok'` aus. Entspricht die Eingabe weder 'ja' oder 'nein', gebt `'ungültige Eingabe'` aus. Wenn ihr noch etwas rumbasteln wollt, überlegt euch wie ihr den `or`-Operator nutzt, damit auch 'JA', 'NEIN', 'Ja' und 'Nein' als gültige Eingaben akzeptiert werden.

5.3 Aufgabe

Schreibt ein kurzes Programm, das ein vorher festgelegtes Passwort mit einem über `input()` vom Terminal eingelesenes Passwort vergleicht und gebt zurück, ob die Passwörter übereinstimmen oder nicht. Benutzt hierfür die Datei `Aufgabe_PW-Check.py`.

6 Schleifen

`Aufgabe_Schleifen.py`

6.1 Theorie

In diesem Kapitel wird euch das Konzept der Schleifen vorgestellt. Diese werden benötigt, um einen Codeblock wiederholt auszuführen. Die meisten Schleifen enthalten einen Zähler oder ganz allgemein Variablen, die im Verlauf der Berechnungen innerhalb der Schleife ihre Werte ändern. Außerhalb, d.h. noch vor dem Beginn der Schleife, werden diese Variablen initialisiert. Vor jedem Schleifendurchlauf wird geprüft, ob ein Ausdruck, in dem diese Variable oder Variablen vorkommen, wahr ist. Dieser Ausdruck bestimmt das Beendigungs-Kriterium der Schleife. Solange die Berechnung dieses Ausdrucks `'True'` liefert, wird der Code innerhalb der Schleife ausgeführt. Nachdem alle Anweisungen innerhalb der Schleife durchgeführt worden sind, springt die Programmsteuerung automatisch zum Anfang der Schleife, also zur Prüfung des Beendigungs-Kriteriums zurück und prüft wieder, ob diese weiterhin erfüllt ist. Wenn ja, geht es wie oben beschrieben weiter, ansonsten wird der Code innerhalb der Schleife nicht mehr ausgeführt und es wird mit dem Rest des Skriptes fortgefahren. In Python gibt es zwei Schleifentypen: die `for`-Schleife und die `while`-Schleife.

- `for`-Schleife: Der Aufbau einer `for`-Schleife ist immer relativ ähnlich und beginnt mit dem Wort `for`. Anschließend folgt der Variablenname, dessen Wert sich mit jedem Durchlauf verändert. Wir nennen diese Variable auch *counter*. Damit ihr festlegen könnt, wie oft die Schleife ausgeführt wird, benutzt ihr die `range()` Funktion. Sie definiert den Wertebereich für den counter. Für diese Funktion gibt es insgesamt drei Angaben: `range(start, stop, step)`. So definiert ihr, bei welcher Zahl der counter starten soll, dieser ist per Default 0. Außerdem könnt ihr angeben, bei welchem Wert die Schleife stoppen soll und wie groß die Schrittweite ist, diese ist per Default 1. Wurde die Schleife vollständig ausgeführt, wird der restliche Programmcode (falls vorhanden) abgearbeitet. Der Code der innerhalb der `for`-Schleife steht, muss eingerückt sein. Das Programm mit dem Code

  ```
  for i in range(0, 5, 1):
      print('huhu')
  print('fertig')
  ```

  sollte euch fünf mal den String 'huhu' und anschließend den String 'fertig' ausgeben. Der counter, der in unserem Fall i heißt, startet also bei $i = 0$, nach jedem Durchlauf wird auf den Wert von i 1 addiert, da unsere Schrittweite 1 ist. Ist $i = 5$, wird die Schleife nicht mehr ausgeführt.

- `while`-Schleife: Für eine `while`-Schleife benötigt ihr eine Bedingung, die vor jedem Durchlauf der Schleife überprüft wird. Solange diese Bedingung erfüllt ist, wird die `while`-Schleife ausgeführt. Ist die Bedingung nicht mehr erfüllt, wird die Schleife nicht mehr ausgeführt und das Programm läuft nach der Schleife weiter bis zum Ende des Programmcodes (wenn nach der Schleife noch Code kommt). Das Programm mit dem Code

  ```
  counter = 1
  while (couter < 7):
      print('Der counter ist noch kleiner als sieben.')
      counter = counter + 1
  print('Die while-Schleife wurde sieben mal ausgef\"uhrt.)
  ```

  führt die Schleife sechs mal aus. Ist `counter = 7`, so wird die Schleife nicht mehr ausgeführt und der restliche Programmcode wird abgearbeitet.

6.2 Aufgabe

Schreibt ein Programm, welches folgendes die folgenden drei Teilaufgaben erfüllt:

1. Gebt alle Zahlen von Null bis 23 nacheinander aus. Nutzt hierfür eine *for*-Schleife.

2. Gebt jetzt die Zahlen 23 bis Null aus, nutzt hierfür aber diesmal eine *While*-Schleife.

3. Jetzt soll eine Pyramide aus Sternchen ausgegeben werden, die Pyramide soll sechs Stufen haben. Ein Beispiel für eine Pyramide mit drei Stufen seht ihr unten. Nutzt aus, dass ihr bei einer *for*-Schleife die Schrittweite selbst bestimmen könnt.

```
*
***
*****
```

4. Schreibt eine Schleife, die solange, bis eine bestimmte Eingabe getätigt wird, immerwieder zur Eingabe aufruft. Ungefähr so:

```
Gibt mir die geheime Eingabe: "abc"
Falsch! Gib mir die geheime Eingabe: "def"
Falsch! Gib mir die geheime Eingabe: "irgendwas"
Gut! Danke!
```

   Welche Eingabe das ist, dürft ihr entscheiden.

Hinweis:
Bei `print()` könnt ihr angeben, wie oft ein Zeichen ausgegeben werden soll:

```
print("a") ergibt: a
print(7*"a") ergibt: aaaaaaa
```

7 Listen

`Aufgabe_Listen.py`

7.1 Theorie

Listen sind in Python eine Möglichkeit, mehrere Variablen an einem Ort zu speichern. Eine Liste beginnt und endet immer mit einer eckigen Klammer. Die Variablen bzw. Objekte innerhalb der Liste werden jeweils durch ein Komma getrennt. Das Programm mit dem Code

```
a = 1
b = 2
c = 3
liste =[a, b, c]
print(liste)
```

gibt euch die Liste `[1,2,3]` aus. Es gibt auch leere Liste, diese initialisiert ihr mit `liste = []`. Es gibt viele Python-Funktionen zum Verändern von Listen. Ihr könnt beispielsweise der Liste Elemente hinzufügen, löschen, verändern etc. Ihr könnt mit Schleifen über die Elemente einer Liste iterieren. Das Programm mit dem Code

```
liste=[1,5, 'huhu', '8']
for i in range(0, 4, 1):
    print(liste[i])
```

gibt euch jeweils das $i-te$ Listenelement aus.
Das **erste Element** in einer Liste hat also immer den **Index null**.

7.2 Aufgabe

In dieser Aufgabe sollt ihr euch mit Listen und einfachen Listen-Funktionen vertraut machen. Erstellt dafür eine Liste namens `test_liste` welche die folgenden Elemente beinhaltet: 'hallo', True, 3.1415, 2345, 'ciao' und -4. Führt nun die Operationen im Codeskelett aus, und findet so heraus, was die Funktionen tun. Sobald ihr es herausgefunden habt, gebt dies auf dem Terminal mit `print()` aus.

8 Funktionen

`Aufgabe_Funktionen.py`

8.1 Theorie

Eine Funktion ist ein Programmcode, der gezielt aufgerufen werden muss bzw. kann. Das ermöglicht euch, diese Funktion auch bei Bedarf öfter aufzurufen und somit übersichtlichen Code zu schreiben, der weniger Fehlerquellen enthält. Für eine Funktion vergebt ihr einen Namen, den ihr an jeder beliebigen Stelle in eurem Python-Programm aufrufen können. Zur Defintion einer Funktion dient das Schlüsselwort `def`, das heiß Python erwartet dieses Wort, wenn ihr eine Funktion definieren wollt. Nach diesem Schlüsselwort kommt der Name der Funktion. Der Code innerhalb der Funktion muss eingerückt sein.

Funktionen ohne Argumente
Die Argumente, die eine Funktion erwarten, schreibt ihr in die runden Klammern nach dem Namen der Funktion. Erwartet die Funktion keine Argumente, so sind die runden Klammern leer. Der nachfolgende Programmcode zeigt eine Funktion mit dem Namen *ausgabe*, die keine Argumente erwartet.

```
def ausgabe():
    print('huhu')
ausgabe()
```

In den ersten beiden Zeilen wird die Funktion definiert, in der dritten Zeile wird die Funktion aufgerufen. Es wird euch der String 'huhu' zurück gegeben.

Funktionen mit Argument(en)
Im folgenden Code wird eine Funktion definiert, die ein Argument mit dem Namen 'x' erwartet.

```
def farbe(x):
    print(x)
x = 'rosa'
farbe(x)
x = 'blau'
farbe(x)
```

In den ersten beiden Zeilen wird die Funktion mit dem Namen 'farbe' definiert. In der dritten Zeile wird die Variable x deklariert, in der vierten Zeile wird die Funktion mit dem Argument x aufgerufen. Wir ändern dann den Wert der Variable und rufen die Funktion erneut auf. Führt ihr dieses Programm aus, so erhaltet ihr zuerst die Ausgabe `rosa` und dann die Ausgabe `blau`.

`return`-Befehl
Der `return`-Befehl wird benutzt, um die Funktion an dieser Stelle wieder zu verlassen. Bei Bedarf können wir die Funktion an dieser Stelle auch einen Wert zurückgeben lassen. Bei dem `return`-Befehl ist zu beachten, dass dieser nichts im Terminal ausgibt. Wollt ihr das, was die Funktion zurück gibt (returnt), im Terminal angezeigt haben, so müsst ihr dies mit printen lassen.

Das Programm mit dem Code:

```
def maximum(x, y):
   if x > y:
      return x
   else:
      return y
x = 2
y = 1
maximum(x,y)
```

verlässt die Funktion `maximum` in der dritten Zeile und returnt den Wert 2. Im Terminal wird euch nichts angezeigt. Führt ihr das folgende Programm aus:

```
def minimum(x, y):
    if x < y:
      return x
    else:
      return y
x = 2
y = 1
print(minimum(x,y))
```

so wird euch im Terminal die Zahl 1 ausgegeben.

8.2 Aufgabe

1. Es sollen fünf Additionen und ihr Ergebnis im Terminal ausgegeben werden: `23456+4637, 987675+11132, 20679+6774222243, 123345+834` und `13459325+98765432`. Anstatt 10 Variablen zu deklarieren und mühsam jedes mal `print("Das Ergebnis von ", zahl1, " + ", zahl2, " ist:  ", zahl1+zahl2)` schreiben zu müssen, schreibt eine Funktion `ausgabe(x,y)`, die genau das tut und zwei Zahlen als Eingabe erwartet.
2. Schreibt die Funktion `quadratzahl(x)`, welche prüfen soll, ob die Eingabe einer User*in eine Quadratzahl ist oder nicht. Je nach dem soll eine geeignete Ausgabe auf dem Terminal passieren. Die Ausgabe soll nicht innerhalb der Funktion passieren. Benutzt hier den `return` Befehl.
3. Überlegt euch Funktionen, sodass ein Muster auf der Konsole 'gemalt' wird. Es kann eins der Beispielmuster unten sein, oder ihr denkt euch selber eins aus. Diese Funktionen sollten aber können Argumente erwarten.

```
Muster 1: 00000
          00000
          00000

Muster 2: ~~~~~~~~~~
          **********
          ~~~~~~~~~~
          **********
```

9 Geschafft!

Herzlichen Glückwunsch! Ihr habt euch die Grundlagen für Python angeeignet und könnt nun, wenn ihr wollt, anfangen, Aufgaben in beliebiger Reihenfolge – je nach euren Interessen und eurem Wissen – zu bearbeiten.

Wir wünschen euch viel Erfolg und vorallem viel Spaß!

Aufgaben, um die Grundlagen zu festigen

10 Münzwurf

`Aufgabe_Muenzwurf.py`

In dieser Aufgabe sollt ihr ein Rate-Spiel programmieren, welches auf einem simulierten Münzwurf basiert. Die Arbeitsschritte sind für euch im Folgenden aufgeführt.

1. Importiert das 'random'-Modul, welches Funktionen zum generieren von Zufallszahlen bereitstellt. Zum importieren des Moduls müsst ihr am Anfang des Programms

   ```
   import random
   ```

   schreiben.

2. Fordert die User*in auf, ihre Wahl ("Kopf" oder "Zahl") einzugeben

3. Generiere eine Zufallszahl (1 oder 2) für die Münzwurf-Simulation. Diese Zufallszahlen könnt ihr mit folgendermaßen generieren:

   ```
   num = random.randint(1, 2)
   ```

4. Legt basierend auf der generierten Zufallszahl das Ergebnis "Kopf" oder "Zahl" fest.

5. Überprüft, ob die Wahl der User*in mit dem Ergebnis übereinstimmt und lasst dementsprechend eine Gewonnen- oder Verloren-Nachricht auf dem Terminal ausgeben.

11 Zahl erraten

`Aufgabe_Zahlenraten.py`

Schreibt ein Programm, das einer User*in ermöglicht, eine zufällig generierte Zahl in einem bestimmten Bereich zu erraten. Zur Umsetzung findet ihr im Folgenden eine kurze Anleitung.

1. *Bereich eingeben:* Das Programm soll die User*in nach der unteren und oberen Grenze für die Zufallszahl fragen. So könnt ihr den Bereich, in dem die Zufallszahl liegt, definieren.

   ```
   Untere Grenze eingeben:-
   Obere Grenze eingeben:-
   ```

2. *Spielregeln:* Das Programm soll eine Zahl im eben definierten Bereich generieren. Die User*in soll diese Zahl nun erraten. Sie hat dafür sieben Versuche.

   ```
   Du hast nur sieben Versuche, die Zahl zu erraten!
   ```

3. *Vermutung abgeben:* Die User*in soll nun eine Vermutung abgeben. Da sie maximal sieben Versuche hat, kann sie maximal sieben Vermutungen hintereinander abgeben.

   ```
   Zahl raten:-
   ```

4. *Feedback:* Bei jeder Vermutung soll das Programm Feedback geben. Wurde die richtige Zahl erraten, so soll das Programm `Glückwunsch! Du hast die richtige Zahl erraten! Du hast es nach x Versuchen geschafft` zurück geben. Das x ist hier ein Platzhalter für die tatsächliche Anzahl der Versuche. Ist die geratene Zahl zu niedrig, soll das Programm `Die geratene Zahl ist zu niedrig!` ausgeben. Ist die geratene Zahl zu hoch, so soll das Programm `Die geratene Zahl ist zu hoch!` ausgeben.

5. *Spielende:* Das Spiel endet nach sieben Versuchen oder bei der richtigen Vermutung. Wurde die Zahl nicht erraten, so gebt die richtige Zahl sowie `Du hast die Zahl leider nicht erraten, viel Glück beim nächsten Mal!`

Beispiel:

```
Untere Grenze eingeben: 1
Obere Grenze eingeben: 100
Du hast nur sieben Versuche, die Zahl zu erraten!

Zahl raten: 50
Die geratene Zahl ist zu niedirg!

Zahl raten: 75
Die geratene Zahl ist zu hoch!

Zahl raten: 65
Glückwunsch, du hast die richtige Zahl erraten! Du hast es nach 3 Versuchen geschafft!
```

12 Wahrsager*in

`Aufgabe_Wahrsagerin.py`

Schreibt ein Programm, das eine Wahrsager*in simuliert. Zur Umsetzung findet ihr im Folgenden eine kleine Anleitung.

1. Das Programm soll die User*in auffordern, eine Ja- oder Nein-Frage zu stellen.
2. Die Wahrsager*in soll eine zufällige Antwort aus einer vordefinierten Liste von Antworten ausgeben. Verwendet hierzu das `random`-Modul, um die zufällige Auswahl der Antworten zu ermöglichen.
3. Gebt die Frage der User*in und die Antwort der Wahrsager*in aus.

13 Produkt zweier natürlicher Zahlen ohne Multiplikations-Operator

`Aufgabe_Produkt_ohne_Malnehmen.py`

Hier sollt ihr ein kleines Programm schreiben, bei welchem die User*in zwei natürliche Zahlen eingeben kann, und das Produkt dieser Zahlen ausgegeben wird. Der einzige Haken hierbei ist, dass der Multiplikations-Operator (*) nicht benutzt werden darf.

1. Schreibt zuerst eine Variante, in der ihr eine *for*-Schleife nutzt. Tipp: Der `range()` Operator ist hier sehr hilfreich.
2. Setzt das ganze jetzt mit einer *while*-Schleife um. Tipp: Achtet darauf, dass ihr in keiner Endlosschleife landet.

14 Wieviele Zahlen sind durch 9 teilbar?

`Aufgabe_9teilbarkeit.py`

Hier erhaltet ihr eine Liste mit 200 Zahlen. Ihr sollt ein Programm schreiben, das herausfindet, wie viele Zahlen dieser Liste durch 9 teilbar sind. Zur Umsetzung findet ihr im Folgenden drei Hinweise.

Hinweise:

1. Eine Zahl ist genau dann durch 9 teilbar, wenn ihre Quersumme durch 3 teilbar ist.

2. Benutzt den **Modulo Operator**. Dieser ist wie folgt definiert: Seien a und b zwei natürliche Zahlen, dann ist $a \mod b$ der Rest, der übrigbleibt, wenn wir a durch b ganzzahlig teilen.
   Beispielsweise ist:
   - $5 \mod 3 = 2$, da $\frac{5}{3} = 1$ mit Rest 2 und
   - $60 \mod 2 = 0$, da $\frac{60}{2} = 30$ mit Rest 0.
3. Der Modulo Operator entspricht in Python dem % Zeichen.

15 Der etwas andere Taschenrechner...

`Aufgabe_Rechner.py`

In dieser Aufgabe sollt ihr einen Taschenrechner programmieren, der auf Grundlage der beiden vorherigen Aufgaben folgendes können soll:

1. Das Programm soll die User*in nach einer mathematischen Operation fragen.
2. Das Programm soll die Operatoren Addition, Subtraktion, Multiplikation (ohne Malnehmen), Division, Potenzrechnung (siehe Hinweise) sowie Teilbarkeit ausführen können. In Bezug auf die Teilbarkeit soll das Programm zumindest zurück geben, ob eine Zahl gerade oder ungerade ist, und, ob eine Zahl durch 9 teilbar ist. Hier dürft ihr aber natürlich zusätzlich andere Teilbarkeitsprüfungen einfügen (*Zum Beispiel, ob eine Zahl durch 4,5 oder 6 teilbar ist, oder ob eine Zahl eine Primzahl ist*).
3. Das Programm soll die User*in zur Eingabe der Zahl(en) auffordern und auf ungültige Eingaben reagieren.
4. Das Programm soll entsprechend der Eingabe die Lösung ausgeben.
5. Nach jeder Berechnung soll die User*in gefragt werden, ob sie eine weitere Berechnung durchführen möchte.

Hinweise:
Potenzrechnung kann unterschiedlich implementiert werden. Entweder verwendet ihr die direkte Python Notation: x^a`=x**a`
Oder ihr macht es ähnlich wie bei der Multiplikation ohne den entsprechenden Operator, mit einer Schleife: $x^a = \underbrace{x \cdot x \cdot x \cdot ... \cdot x \cdot x}_{\text{a Mal}}$.
Desweiteren könnte es hilfreich sein, alle Operationen, die ihr anbieten wollt, in jeweils eigene Funktionen auszulagern:

```
def addition(x, y):
    return x + y
...
def ist_primzahl(x):
    for i in range(0,x):
        if (...):
            ...
            return False
        else:
            return True
#und so weiter...
```

Ein Programmaufruf könnte dann so aussehen:

```
Wähle eine der folgenden Optionen:
 1: Addition
 2: Subtraktion
 ...
 4: Prüfen, ob Zahl durch 9 teilbar ist
 5: Prüfen, ob Zahl eine Primzahl ist
 "4"
Gib eine Zahl ein: "344448"
Diese Zahl ist durch 9 teilbar. Soll eine weitere Operation
durchgeführt werden (ja/nein)? "weiß nicht"
Dies ist eine ungültige Eingabe. Das Programm wird neu gestartet.
Wähle eine der folgenden Optionen:
 ...
```

Die Terme in Anführungszeichen sind die Eingaben der User*in. Damit die User*in immer wieder gefragt wird, ob sie noch eine Operation machen möchte, sofern sie nicht "nein" eingibt, bietet sich eine endlose `while`-Schleife an.

Aufgaben mit etwas komplexeren Herangehensweisen

16 Prüfen, ob eine Zahl eine Narzisszahl ist
`Aufgabe_Narzisszahlen.py`

Eine natürliche Zahl z mit l Ziffern ($z = z_l z_{l-1} ... z_2 z_1$) heißt **Narzisszahl**, wenn folgendes für diese Zahl gilt:

$$z = \sum_{i=1}^{l} (z_i)^l$$

Ein Beispiel für eine 3-stellige Narzisszahl ist 371, denn:

$$371 = \sum_{i=1}^{3} (z_i)^3 = 1^3 + 7^3 + 3^3 = 1 + 343 + 27$$

Eure Aufgabe ist es nun, ein Programm zu schreiben, welches alle 4-stelligen Narzisszahlen ausgibt. Schreibt hierzu eine eigene Funktion `narzisscheck(zahl)`, die für eine einzelne Zahl prüft, ob diese eine Narzisszahl ist und dementsprechend 1 zurückgibt, wenn die Zahl eine Narzisszahl ist und 0, wenn die Zahl keine Narzisszahl ist.

Hinweise: Überlegt euch, wie ihr die einzelnen Ziffern der Zahl erhaltet. In Python kann eine einzelne Zahl auch als `string` interpretiert werden.

17 Lösen biquadratischer Funktionen
`/Aufgabe_biquadratischeGleichung`

Eine biquadratische Gleichung ist eine Gleichung der Form:

$$\begin{aligned} & ax^4 + bx^2 = c \\ \Leftrightarrow\ & ax^4 + bx^2 - c = 0 \end{aligned}$$

Ziel dieser Aufgabe ist es, die (reellen) Lösungen der biquadratischen Funktion zu berechnen und auszugeben. Die User*in soll hierbei die Möglichkeit haben, die Funktion `loesungen(a,b,c)` mit beliebigen Parametern $a, b, c \in \mathbb{R}$ im Terminal aufzurufen. Beachtet, dass eine Gleichung vierten Grades auch nicht-reelle Lösungen haben kann.

Hinweise:

1. Die Wurzelfunktion $\sqrt{x}$ kann umgeschrieben werden als $x^{\frac{1}{2}}$. In Python gibt es allerdings auch das Mathe-Modul, welches mit `import math` ganz oben im Code eingebunden werden kann. Dieses Modul beinhaltet auch die Quadratwurzelfunktion (*englisch: square root*), welche dann mit `math.sqrt(x)` aufgerufen werden kann.

2. Damit eine User*in im Terminal eine Funktion mit unterschiedlichen Parametern aufrufen kann, muss das Programm mit: `python3 -i programmname.py` aufgerufen werden. Um dies zu testen, findet ihr in diesem Ordner ein Programm `beispiel.py` welches eine Funktion `hallo(name)` enthält und einen String *name* als Eingabe erwartet und dann auf der Konsole eine Begrüßung mit dem Namen ausgibt. Das sollte dann etwa so aussehen:

   (a) /verzeichnis/zum/ordner/AufgabeX/ > python3 -i beispiel.py
   (b) >>> hallo("ABC")
   (c) >>> Hola/Hello/Salut/Ciao ABC! Viel Erfolg bei der Aufgabe:)
   (d) >>> hallo("LalalA")
   (e) >>> Hola/Hello/Salut/Ciao LalalA! Viel Erfolg bei der Aufgabe:)

   Mit **Strg+d** könnt ihr den interaktiven Python Modus verlassen und das Programm erneut starten, wenn ihr Veränderungen vorgenommen habt.

18 Caeser-Verschlüsselung

`Aufgabe_Caeser.py`

Beim programmieren werden einzelne Buchstaben mittels der ASCII Tabelle vom Computer verstanden. Einem Buchstaben wird hierbei eine eindeutige Zahl zugeordnet. Die Kleinbuchstaben a-z (ohne Umlaute) erhalten hierbei die Zahlen 97-122. Mit den pythoninternen Funktionen `ord(buchstabe)` und `chr(zahl)` erhält man die zu einem Buchstaben zugehörige Zahl bzw den zu einer Zahl zugehörigen Buchstaben.
Beispiel: ord('b')=98, chr(121)=y.
Die Caeser-Verschlüsselung verschlüsselt einen Text, indem sie das Alphabet verschiebt. Ist der Schlüssel beispielsweise 2, so wird a zu c, b zu d, c zu e und so weiter. Eure Aufgabe ist es, den Text "wannistwiedersommer" mit dem Schlüssel 11 zu verschlüsseln.

Hinweise:

1. Schreibt eine extra Funktion `verschiebung(zeichen, schluessel)` die nur einen Buchstaben und den Verschiebungs-Schlüssel erhält und auf Basis dessen den Buchstaben verschlüsselt und den verschlüsselten Buchstaben zurückgibt.

2. Auch Zahlen, die nicht zwischen 97 und 122 liegen, sind mit Zeichen belegt. Der verschlüsselte Text soll aber nur die Zeichen a-z enthalten, achtet also beim schreiben eurer Verschiebungs-Funktion darauf, dass dies konsistent ist.

19 Kaprekar Konstante finden

`Aufgabe_Kaprekarkonstante.py`

Die Kaprekar-Konstante für n-stellige Zahlen findet man, in dem mit einer beliebigen n-stelligen Zahl startet, bei der *nicht* alle n Ziffern gleich sein dürfen. Nun bildet man die größt- und kleinstmögliche Zahl, die sich aus den Ziffern bilden lässt und subtrahiert diese voneinander. Hat ein Ergebnis zwischendurch nicht mehr n Ziffern, so wird es von vorne mit Nullen aufgefüllt. Dieser Vorgang wird so lange wiederholt, bis sich das Ergebnis der Subtraktion nicht mehr ändert. Dieses Ergebnis ist dann die *Kaprekar-Konstante*.
Ein beispielhafter Vorgang für 3-Stellige Zahlen:

- erste Zahl: 989
- größtmögliche Zahl (gmZ): 998, kleinstmögliche Zahl (kmZ): 899
- Subtrahieren: 998-899= 99
- neue Zahl: 99 *nicht mehr 3-stellig → mit 0 auffüllen*
- neue Zahl: 099
- gmZ: 990, kmZ: 099
- neue Zahl → 990-099=891
- gmZ: 981, kmZ: 189
- neue Zahl → 981-189=792
- gmZ: 972, kmZ: 279
- neue Zahl → 972-279=693
- gmZ: 963, kmZ: 369
- neue Zahl → 963-369=549
- gmZ: 954, kmZ: 459

- neue Zahl → 954-459=495
- gmz: 954, kmZ: 459
- neue Zahl → 954-459=495
- ⇒ Kaprekar-Konstante ist **495**

Eure Aufgabe ist es nun ein Programm zu schreiben, das die Kaprekar-Konstante für 4-stellige Zahlen zu findet. Schreibt dazu die Ziffern der Zahlen in eine Liste. Schreibt zuerst eine extra Funktion, die euch diese Liste sortiert. Nutzt hierfür keine importierten Funktionen. Schreibt anschließend die Funktion, die die Konstante berechnet. Brecht die weitere Berechnung ab, wenn der Input ungültig ist (zum Beispiel bei vier gleichen Ziffern oder bei einer nicht vier-stelligen Zahl) und vergesst nicht, dass eventuell Nullen hinzugefügt werden müssen.

Hinweise:
Eine Liste L in Python lässt sich auf verschiedene Arten umsortieren:

- `L-umgekehrt=L.reverse()`
- `L-umgekehrt=L[::-1]`

Da die Iterations-Länge nicht bekannt ist, hilft eine *for*-Schleife eher nicht. Eine Fast-Endlos-Schleife mit einer Hilfsvariablen hingegen ist hilfreich.

Herausfordernde Projekte

20 TicTacToe und dumme KI

`Aufgabe_Tictactoe.py`

Ziel ist es, ein TicTacToe-Spiel zu schreiben, bei welchem ihr sowohl gegen eine echte zweite Spieler*in antreten könnt, als auch gegen den Computer (dieser muss nicht strategisch klug vorgehen können, sondern soll einfach ein zufälliges freies Feld auswählen).
Die Herangehensweise kann folgendermaßen aussehen:

1. Überlegt euch, wie das Spielfeld auf dem Terminal aussehen kann, und wie ihr es so konstruiert, dass später auf die Felder zugegriffen werden kann. Schreibt eine Funktion `spielfeld()`
2. Schreibt eine Funktion `spielerin_wahl`, in welcher behandelt wird, was passiert, wenn die Spieler*in:
   - Ein gültiges Feld für ihren Zug auswählt,
   - ein ungültiges Feld auswählen will,
   - das Spiel beenden will,

   Beachtet hierbei, dass der Python jede Eingabe als `string` interpretiert, auch wenn ihr eine eine Zahl eingebt.
3. Schreibt eine Funktion `wechsel()`, die die aktuelle Spieler*in switcht.
4. Es wird ein Check benötigt, ob eine Spieler*in gewonnen hat oder ob das Spiel unentschieden ausgegangen ist.
5. Das Hauptprogramm soll alle vorherigen Funktionen vereinen. Dieses soll auch die Unterscheidung des Spielmodus (gegen Person oder Computer) vornehmen und dementsprechend agieren.

Hinweise:
Innerhalb von Funktionen könnt ihr nur auf Variablen zugreifen, die ihr innerhalb dieser Funktion definiert habt. Beispielsweise wird folgender Code

```
a = 5
def addition_mit_a(b):
    print(a + b)
```

einen Fehler produzieren, da die Funktion die Variable a nicht kennt, obwohl a vorher im Code definiert wurde. Um das Problem zu lösen, gibt es in Python folgendes:

```
a=5
def addition_mit_a(b):
    global
    print(a+b)
```

Durch das Wort `global` können Variablen, die nicht innerhalb der Funktion definiert wurden auch innerhalb von Funktionen genutzt werden, ohne dass sie extra übergeben werden müssen.

21 Entkomme dem Labyrinth

`/Aufgabe_Labyrinth`

In dieser Aufgabe sollt ihr eine Spielfigur (<<) durch das Labyrinth steuern, um den Ausgang zu erreichen. Schreibt das Programm in der Funktion `escape()` in der Datei `maze.py`. Um euer Programm zu testen, führt `maze.py` über die Konsole aus. Beachtet, dass das Labyrinth bei jeder Ausführung neu generiert wird und daher immer anders aussieht.

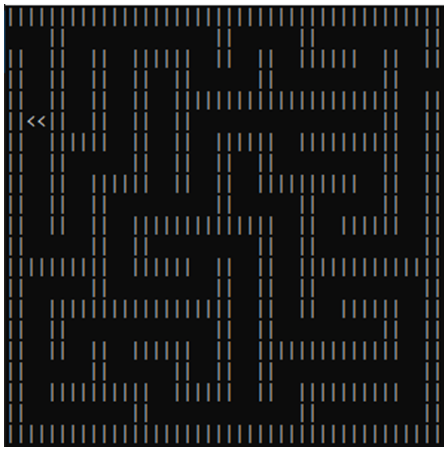


Figure 3: Hier seht ihr ein generiertes Labyrinth.

Hinweis: Benutzt folgende Befehle zum Steuern der Spielfigur:

- `step()` - Bewegt die Spielfigur ein Feld in Blickrichtung.
- `turnRight()` - Dreht die Spielfigur um 90° nach rechts.
- `turnLeft()` - Dreht die Spielfigur um 90° nach links.
- `blocked()` - Überprüft, ob die Spielfigur vor einer Wand steht.
- `wall()` - gibt einen Boolean (True/False) zurück.

Die Blickrichtung der Spielfigur wird folgendermaßen visualisiert: Blickrichtung nach oben: ^^, rechts: >>, unten: vv, links: <<.
Die Größe des Labyrinths lässt sich über die Werte h und w unten im Code zum Testen deines Codes verändern, um den Ausgang schneller zu erreichen. :)

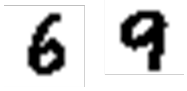
22 6 oder 9?

`/Aufgabe_6-oder-9`

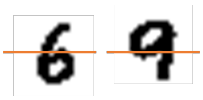
In dieser Aufgabe sollt ihr ein Programm schreiben, welches handgeschriebene 6en und 9en unterscheiden kann. Schreibt euren Code in die Funktion `decide()` unter # TODO. Die Funktion erhält als Eingabeargument eine verschachtelte Liste mit dem Namen `image_array`. Ihr könnt euch `image_array` wie eine Matrix vorstellen, welche die Farbe der einzelnen Pixel eines Bildes speichert. Den Farbwert des Pixels in Zeile z und Spalte s kannst du mit `image_array[z][s]` abrufen.

- Falls `image_array[z][s]` $== 1$ gilt, ist das Pixel in der Zeile z, Spalte s schwarz.
- Falls `image_array[z][s]` $== 0$ gilt, ist das Pixel in der Zeile z Spalte s weiß.

Die Funktion soll für beliebige Eingaben entscheiden, ob auf dem Bild eine 6 oder 9 ist und dementsprechend entweder 6 oder 9 zurückgeben. Bevor ihr eine Lösung in Python implementiert, solltet ihr euch eine Strategie überlegen, mit welcher 6en und 9en effizient unterschieden werden können.



Mit `py six_or_nine test_6` könnt ihr den Code für zufällig ausgewählte Bilder, auf denen einen 6 ist, und mit `py six_or_nine test_9` für Bilder, auf denen eine 9 ist, testen. Manche Bilder sind schwieriger zu erkennen als andere, also testet am besten immer direkt mehrmals hintereinander. Der Code wird am Ende nicht alle Bilder richtig klassifizieren können. Mit `py six_or_nine test_all` könnt ihr schauen, wie viele Bilder euer Code prozentual richtig erkennt.
Tipps für eine Lösungsstrategie:



Betrachtet jeweils die obere und untere Bildhälfte des Eingabebildes. Was fällt euch beim Zählen der schwarzen Pixel pro Bildhälfte für 6en und 9en auf?

Dass der Code bisher noch viele Bilder falsch klassifiziert, liegt daran, dass die Zahlen nicht immer vertikal genau in der Bildmitte liegen: Euer Code sollte also erst ab der Zeile beginnen, schwarze Pixel zu zählen,

in der die Zahl beginnt. Ebenso sollte der Code nur den Bereich bis zur letzten Zeile, in der die Zahl ist, betrachten.