Python programmieren lernen

Die AG befasst sich mit der Programmierung in Python, auf einem Raspberry Pi, wir verwenden dabei Python 3.

Durch die Verwendung des Raspberry Pi ist es möglich, sich nicht rein auf den Bildschirm und Tastatur als Ausgabe- und Eingabemedium zu beziehen, es ist auch möglich über die Schnittstellen des Pi mit einfachen Elektronik Komponenten zu arbeiten.

Dieses Dokument soll als Hilfe zur Verfügung stehen und über das ganze Schuljahr wachsen, mit den Informationen, die in der Schule bearbeitet werden.

Begonnen wird mit einer sehr kurzen Einführung des Raspberry Pi, um dann schnell zur Programmierung mit Python zu kommen.

Raspberry Pi

Der Raspberry Pi wird von einer englische Non-Profit Organisation entwickelt. Ziel der Organisation ist, dass Schüler einen einfachen und kostengünstigen Einstieg in die Welt der Computer erhalten.

Bezugsquelle

Das hier verwendete Set, kann hier erworben werden. Alternativ kann man sich auch Sets bei Conrad, Reichelt, ELV oder Amazon kaufen (preislich gibt es nur minimale unterschiede).

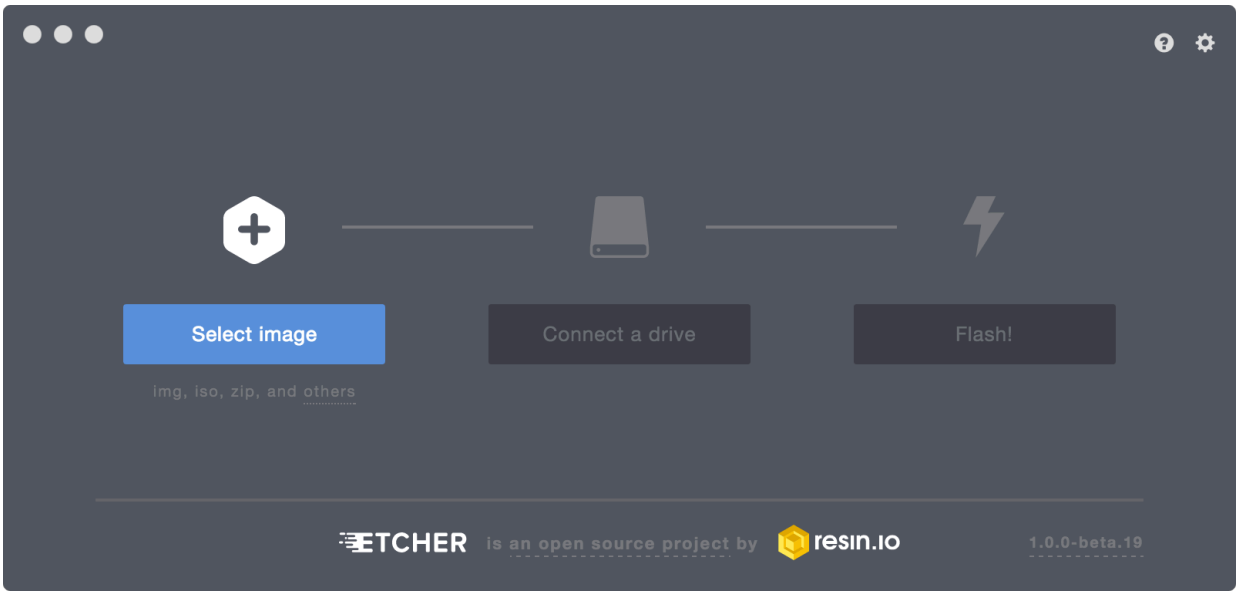
Installation

Nach dem "Zusammenbau" des Raspberry Pi muss man die microSD Karte mit einem Raspberry Pi Image bespielen. Wir verwenden hier Raspbian, welches von der Raspberry Pi Foundation angeboten wird.

Bezogen werden kann es direkt von der Webseite der Foundation, von hier die Version mit Desktop herunterladen.

Um das Image aus dem heruntergeladenen ZIP auf die SD Karte zu bekommen, verwendet man am einfachsten Etcher, welches für alle gängigen Betriebsysteme kostenlos zur Verfügung steht.

Nach dem das ZIP entpackt und Etcher installiert wurde, startet man Etcher und bekommt direkt das Bedienerfreundliche UI von Etcher angezeigt. Einfach **Select image** anklicken und das zuvor ausgepackte Raspbian Image auswählen.



Im nächsten Schritt die korrekte microSD Karte auswählen, falls mehrere externe Speichermedien am Rechner vorhanden sind. Im letzten Schritt mit **Flash!** das Image auf die Karte schreiben. Hier kann es vorkommen, dass man dies noch mal durch Eingabe des Benutzerpassworts (oder das Passwort eines Administrators) bestätigen muss.

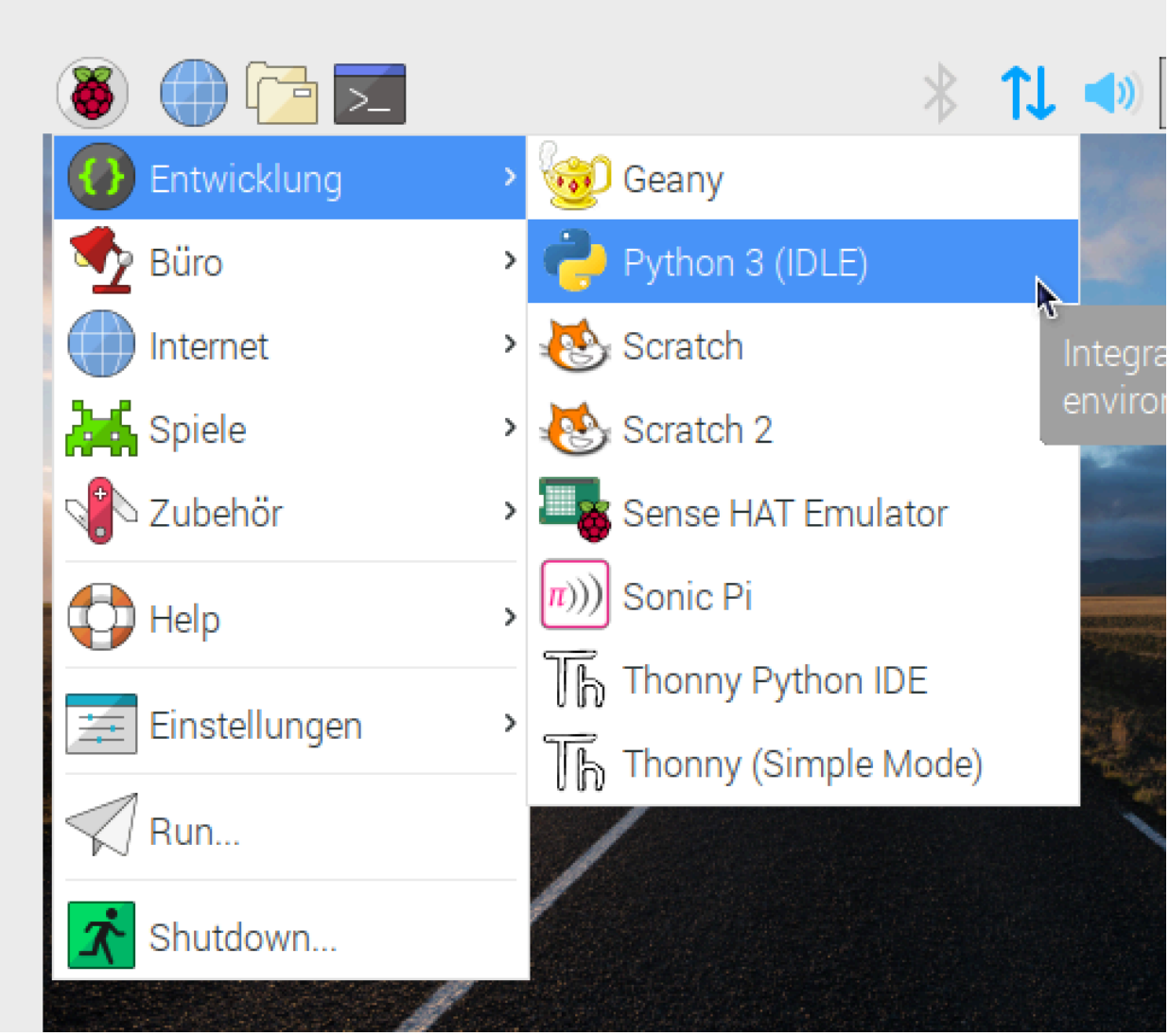
Ist der Schreibvorgang abgeschlossen, kann man die mircoSD Karte entnehmen und in den Raspberry Pi einsetzen. Jetzt alle Kabel am Raspberry Pi anschließen und als letztes mit Strom verbinden. Beim ersten Start von Raspbian muss man noch manche Installationsschritte, welches Sprache verwendet wird, durchgehen (den Anweisung auf dem Bildschirm folgen).

Jetzt ist der Pi einsatzfähig.

Python

Starten von Python

Der einfachste Weg um mit der Entwicklung in Python zu starten, ist das Ausführen der Python IDLE (IDLE = Integrated Development and Learning Environment, könnte Passen, oder der Name kommt wirklich von Eric Idle, einem Mitglied von Monty Python Comedy-Gruppe), dafür öffnet man das Menü ***Himbeere*** *-> Entwicklung -> Python 3 (IDLE)*.



Startet man Python IDLE, öffnet sich direkt ein Fenster, welches auf eine Eingabe wartet. Dies wird durch ">>>" signalisiert, die Python Shell (oder auch REPL = Read Eval Print Loop).

Die Shell reagiert direkt auf Eingabe und führt diese sofort aus.

- **R**ead: Es wird eingelesen, was der Benutzer eingegeben hat.
- **E**val: Die Eingabe wird vom Python Interpreter verarbeitet
- **P**rint: Das Ergebnis der Verarbeitung wird dem Benutzer ausgegeben. Enthält die Eingabe einen Fehler, wird dieser ausgegeben.
- **L**oop: Die Shell wartet auf die nächste Eingabe

Wie in jeder Programmiersprache, ist der Einstieg ein "Hello World", welcher Dank REPL in einer Zeile daherkommt.

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

Info zur Python-Shell

Um den letzten eingegebenen Befehl nochmal zu bearbeiten, kann man diesen mit `Alt+P` zurückholen (P steht hier für *previous*, auf Deutsch *vorheriger*). Um in die andere Richtung zu springen, drückt man die Kombination `Alt+N` (N steht hier für *next*, auf Deutsch *nächster*).

In der Shell kann man einfache Mathematische Berechnungen durchführen, wie mit einem einfachen Taschenrechner.

```
>>> 5 + 5
```

Es stehen alle Grundrechenarten direkt zur Verfügung:

- Addition -> +
- Subtraktion -> -
- Multiplikation -> *
- Division -> /

Es muss aber auf die korrekte und vollständige Schreibweise geachtet werden, sonst werden entsprechende Fehlermeldungen ausgegeben.

```
>>> 5 +
```

Bei dieser Berechnung fehlt der zweite Summand.

Auch Klammern, kann man verwenden, um entsprechende mathematische Regeln abzubilden.

```
>>> (9 / 3) * (2 + 2)
```

Auch bei den Klammern muss man auf die korrekte Schreibweise achten und man darf keine Notationselemente weglassen.

Darüber hinaus gibt es noch drei weitere Operatoren:

- "**" Potenz Berechnung
  `2*2 = 2^2`
- "//" ganz zahlige Division
  `5 // 2 = 2`
- "%" Modulo-Operation
  `5 % 2 = 1`

Zahlen

Bei den Zahlen gibt es verschiedene Typen, wobei uns für den Moment nur zwei interessieren

- Ganze Zahlen (int)
- Gleitpunktzahlen (float)

Beispiel für eine Ganze Zahl ist die *5*, eine Gleitpunktzahl wäre *5.3*. Hier gilt zu beachten, dass die Gleitpunktzahl, wie der Name schon sagt, durch einen Punkt getrennt wird und nicht durch ein Komma!

```
>>> 5.3 + 4.2
```

```
>>> 5,3 + 4,2
```

Was passiert hier im zweiten Beispiel? Es wird eine Liste von Zahlen erstellt, wobei die mittlere Zahl als Mathematischer Ausdruck gewertet wird 3+4=*7*. Damit besteht die Liste aus drei Zahlen.

Mathematische Funktionen

Mit der erstellten Liste kann man direkt weitere Dinge machen, man kann mittels der in Python enthaltenen mathematischen Funktionen, z.B. die größte und die kleinste Zahl der Liste ermitteln. Die Syntax ist immer die gleiche, erst kommt der Funktionsname, z.B. *max*, und dann in Klammer der Wert (auch Parameter genannt), welcher von der Funktion verarbeitet werden soll.

```
>>> min(5, 7, 2)
```

```
>>> max(5, 7, 2)
```

Neben dieser beiden Listen Funktionen, gibt es auch noch weitere einfache mathematische Funktionen, wie die Absolutzahl Berechnung.

```
>>> abs(-5)
```

Im Gegensatz zu *max()* und *min()*, erwartet die Funktion *abs()* nur einen Wert, und es kommt zu einem entsprechenden Fehler, wenn man hier mehr als einen Wert angibt.

Eine weitere Funktion ist das Runden von Zahlen, mit der Funktion *round()*. Hier kann man nur die zu rundende Zahl angeben, oder noch zusätzlich die Anzahl der Nachkommastellen.

```
>>> round(4.536)
```

```
>>> round(4.536, 2)
```

Zu den genannten Funktionen gibt es noch drei weitere Standard Funktionen:

Funktion	Beschreibung	Beispiel Eingabe	Ausgabe
float()	Erzeugt aus dem übergebenen Parameter eine Gleitkommazahl	float(4)	4.0
int()	Erzeugt aus dem übergebenen Parameter eine Ganze Zahl	int(4.0)	4
type()	Ermittelt den Typ des übergebenen Parameter	type(4)	<class 'int'>

Variablen und Zuweisungen

Bisher wurden die Zahlen immer direkt eingegeben. Bei einem normalen Programm, weiss man aber vielleicht noch nicht, welche Zahl überhaupt berechnet werden soll, bzw. man möchte mit dem Ergebnis weitere Berechnungen durchführen. Also muss man sich diese Werte irgendwie merken. Dies geschieht mithilfe von Variablen. Diese sind eine Referenz oder ein Platzhalter für einen Wert. Die Zuweisung die dabei denkbar einfach `name = wert` Damit hätte die Variable den Namen *name* und man würde *wert* darin festhalten. Ein Bespiel mit einer Zahl:

```
>>> x = 8
```

Damit wurde der Wert 47 in dem Platzhalter *x* gespeichert und kann jetzt beliebig wieder abgerufen werden. Dies geschieht durch die einfache Eingabe, bzw. Verwendung des Platzhalters

```
>>> x
```

In einer Mathematischen Formel kann man dieses *x* genauso direkt verwenden

```
>>> x * 2
```

Ebenfalls kann man den Wert von *x* in einer weiteren Variablen *y* speichern. Damit haben dann *x* und *y* den identischen Wert

```
>>> y = x
```

Der Unterschied zur Mathematik wird deutlich, wenn man eine einfache Berechnung betrachtet:

In der Mathematik schreibt man:
Summand_1 + Summand_2 = Summe

In Python (und den meisten anderen Programmiersprachen) schreibt man:
Summe = Summand_1 + Summand_2

Der Variablenname steht immer vor dem **=**.

Wie man die Variablen benennt bleibt bis auf ein paar kleine Regeln, jedem selbst überlassen. Es ist aber sinnvoll, Variablen einen sprechenden Namen zu geben, damit man auch noch Monate später direkt weiss, was diese Variable enthält.

Die Regeln: **Ein Variablenname...**

- besteht aus Buchstaben (a...z,A...Z), Ziffern (0..9) oder Unterstrichen (_)
- beginnt mit einem Buchstaben oder einem Unterstrich
- darf kein reserviertes Wort sein (z.B. *not*, *if*, etc.)
- sollte keinem Funktionsnamen entsprechen (z.B. *abs*, *int*, etc.)

Beispiel für gültige Namen:

- richtig
- _richtig
- Richtig1
- richtigerName
- dieser_Name_ist_auch_erlaubt

Beispiel für ungültige Namen:

- 2_Werte
- dieser-ist-ein-ungültiger

Besonderheiten

`x += 5`
Das **+=** ersetzt hier **= [Variablenname] +**.

Ebenfalls ist es möglich, eine Funktion dadurch einen neuen Namen zu geben:
`runden = round`
Damit für **runden(5.87, 1)** zum selben Ergebnis wie **round(5.87, 1)**.
Ob dies allerdings sinnvoll ist, oder nicht doch eher für Verwirrung sorgt, sollte sich jeder selbst überlegen.

Verwenden von Bibliotheken

In Python sind Bibliotheken (auch Libraries genannt), Programmpakete, die Funktionen enthalten, die Entwickler anderen Entwicklern zur Verfügung stellen, um sie wieder zu verwenden.
Dadurch spart man sich, das Rad jedes mal neu zu erfinden. Zum Beispiel gibt es Mathematische Berechnungen, wie das Wurzelziehen von Zahlen, die man immer wieder mal braucht, und nicht jedes mal neu schreiben will.

Es gibt verschiedene Arten, um bestehende Bibliotheken, in den eigenen Programmcode zu integrieren.
Alle Varianten eint das Schlüsselwort **import**, mit dem angegeben wird, dass etwas importiert/eingefügt werden soll.

1. Variante
   `import math`
   Hier wird die Bibliothek **math**, welche uns weitere mathematische Funktionen zur Verfügung stellt, geladen. Wenn man diese Funktionen verwenden will, muss man dem Funktionsnamen, immer das Wort **math** voranstellen:
   `math.pi`
   Zum Zugriff auf die Zahl **pi** (Pi ist eine Konstante, dies ist einer Variable, mit einem Festzugewiesenen Wert, der nicht geändert werden kann)

2. Variante
   `import math as m`
   Hiermit bekommt die Bibliothek einen Alias oder Alternativnamen, nämlich in diesem Fall, den Buchstaben **m**_. Damit sieht der Zugriff auf **pi** so aus:
   `m.pi`

3. Variante
   `from math import pi`
   Dadurch wird nur **pi** von der **math** Bibliothek eingebunden, und kann direkt verwendet werden ohne vorangestellten Bibliotheksnamen, was dass so aussieht:
   `pi`

4. Variante
   `from math import *`
   So werden alle Funktionen und Konstanten von **math** eingebunden. Dadurch kann es wieder zu Nebenwirkungen kommen, wenn man eine Funktion mit gleichen Namen verwendet, wie in der Library **math**. Die Schreibweise ist identisch zur 3. Variante.

Informationen über eine Bibliothek bekommt man mittels der Funktion **help()**, um sich **math** anzeigen zu lassen, dann schreibt man also: `help(math)`

Wichtig ist hier, dass man vorher die Bibliothek mittels **import** geladen hat. **help()** kann man ebenfalls für einzelne Funktionen benutzen: `help(abs)` oder `help(math.pi)`, um von einer Funktion der **math** Library Informationen zu bekommen.

Programme erstellen

Im nächsten Schritt wollen wir den Programmcode, den wir schreiben, in Dateien abspeichern. Dadurch müssen wir ihn nicht immer und immer wieder schreiben.
In der Python IDLE gibt es hierfür einen einfachen Editor. Diese startet man, via **File -> New File**. Es öffnet sich ein zweites Fenster, in welchem man den Programmcode eingeben kann.
Zum Testen, erzeugen wir nun ein Python Skript, welches uns das bekannte *"Hello World"* ausgibt:

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

Um das Skript in der IDLE direkt laufen lassen zu können, müssen wir es erst speichern. Dafür klicken wir im Editorfenster **File -> Save**. Es öffnet sich ein Speichern Dialog, in dem wir dem Programm einen Namen geben müssen, in dem Fall genügt der Name *test.py*. Wurde das Skript gespeichert, kann man es mittels **Run -> Run Module** ausführen. Das Resultat sieht man in der Python Shell (dem ersten geöffneten Fenster). Vor jeder Ausführung muss man das Skript speichern. Hat man etwas geändert und drückt auf Run Module, kommt eine Entsprechende Rückfrage, ob man die Änderungen vor der Ausführung speichern möchte, was man tun sollte.

Neben der Ausführung über die IDLE, kann man Python Programme auch direkt aus der System Shell ausführen
`python3 test.py`

Interaktive Skripte

Nachdem bisher nur feste Werte bei den Programmen verwendet wurden, soll im nächsten Schritt der Benutzer um eine Eingabe gebeten werden und diese wird dann in einem Skript zur Berechnung verwendet. Damit wird das EVA Prinzip der Datenverarbeitung möglich:

- Der Benutzer gibt Daten ein (**E**ingabe)
- Das Programm verwendet die Eingabe des Benutzers, um Berechnungen durchzuführen (**V**erarbeitung)
- Das Ergebnis wird dem Benutzer mitgeteilt (**A**usgabe)

Eingaben werden in Python mit der Funktion `input()` abgefragt. Als Funktionsparameter kann man einen Text angeben, welcher dem Benutzer angezeigt wird. Die Eingabe kann dann direkt in einer Variablen gespeichert werden.

```
>>> a = input("Länge des Rechteck in cm:")
```

Ein komplettes Beispiel-Programm zur Berechnung des Flächeninhalts eines Rechtecks:

```
# Eingabe
laenge = input("Länge des Rechteck in cm:")
breite = input("Breite des Rechteck in cm:")

# Verarbeitung
flaeche = laenge * breite

# Ausgabe
print("Das Rechteck hat eine Fläche von ", flaeche, "cm2")
```

Programmverzweigungen

In den bisher behandelten Programmen, wurden alle Zeile nacheinander abgearbeitet. Möchte man allerdings auf die Eingabe den Benutzer unterschiedlichen reagieren, wäre eine Verzweigung notwendig. Diese Verzweigung bekommt man mit dem Schlüsselwort **if** und einer darauffolgenden Bedingung.
Die Bedingung wird ausgewertet und je nachdem, wird der Programmcode im **if** Verarbeitungsblock berücksichtigt oder nicht.

```
if <Bedingung>:
    <Verarbeitungsblock>
```

In diesem Beispiel sieht man direkt eine Besonderheit von Python. Während in den meisten Programmiersprachen die Programmblöcke mit Klammern oder Schlüsselwörtern abgebildet werden, erfolgt dies in Python nur mittels Einrückung des Programmcodes. Also alles was nach dem **if** identisch eingerückt ist, gehört bis zur nächsten Zeile die nicht entsprechend eingerückt ist, zum if-Verarbeitungsblock. Bedingungen sind Prüfungen, die entweder *wahr* (TRUE) oder *falsch* (FALSE) sind, das Ergebnis ist somit ein boolscher Ausdruck (ein boolscher Wert kann entweder wahr oder falsch sein).

Zur Prüfung, können verschiedene Operatoren verwendet werden:

Operator	Bedeutung	Beispiel	Ergebnis
<	kleiner als	5 < 10	Wahr - True
		5 < 5	Falsch - False
>	größer als	10 > 5	Wahr - True
<=	kleiner gleich	5 <= 6	Wahr - True
		5 <= 5	Wahr - True
>=	größer gleich	5 >= 6	Falsch - False
==	gleich	1 == 1.0	Wahr - True
!=	ungleich	2 != 3	Wahr - True
is	identisch	2 is 2	Wahr - True
is not	nicht identisch	4 is not 8	Wahr - True

Die Vergleichsoperatoren können auch hintereinander mit mehreren Werten verknüpft werden:
`1 < 2 < 3 <= 3`

Beispiel:

```
a = 1
b = 2

if (a > b):
    print("Ja, Bedingung erfüllt")
```

Verknüpfung von Bedingungen

Muss man mehrere Bedingungen abfragen, "ist das Auto rot?" und "hat das Auto vier Türen?", kann man diese, wie im normal gebrauch verbinden. Mögliche Bindewörter sind: * and -> und-Verknüpfung, alle Bedingungen müssen wahr sein, damit die Komplette Bedingung erfüllt ist * or -> oder-Verknüpfung, eine Bedingung muss wahr sein, damit die Komplette Bedingung erfüllt ist * not -> Verneinung, der aktuelle Wert der Bedingung wird umgekehrt

```
a = 1
b = 2

if a < 2 and b == 2:
    print("Ja, Bedingung erfüllt")
```

Wird die Bedingung nicht erfüllt, und man möchte genau dann etwas anderes machen, gibt es die Möglichkeit einen sogenannten **else** Teil hinzufügen.

```
a = 1
b = 2

if a > b:
    print("Ja, Bedingung erfüllt")
else:
    print("Nein, Bedingung nicht erfüllt")
```

Neben dem **else** Teil gibt es noch eine Kombination aus **else** und **if**, den **elif** Teil. Wird die vorangegangene Bedingung nicht erfüllt, wird die nächste Bedingung geprüft. Sollte die auch nicht erfüllt werden, kann man entweder beliebige weitere **elif** folgen lassen, oder mit einem **else** die Verzweigung beenden.
Sinnvoll ist dieses Vorgehen, wenn man weiss, das die weiteren Bedingungen sowieso nicht erfüllt werden können.

```
a = 1
```

```
b = 2

if a > b:
    print("a ist größer als b")
elif a < b:
    print("b ist größer als a")
else:
    print("a ist gleich b")
```

Programm unterbrechen

Manchmal kann es erforderlich sein, dass man den aktuellen Programmablauf pausiert, z.B. wenn man eine Nachricht ausgibt und dem Anwender die Möglichkeit geben will, diese zu lesen, bevor es weitergeht.

Diese Pausen kann man mit der Library **time** einbauen, und dann die Funktion **sleep** aufrufen. Bei dieser Funktion gibt man die Sekunden, die gewartet werden sollen, als Parameter an.

```
import time time.sleep(3)
```

Minecraft via Python

Im nächsten Schritt wollen wir mittels Python auf die Minecraft Welt des Raspberry Pi Einfluss nehmen. Dazu verwenden wir die Library **mcpi** und daraus das Unterpaket **minecraft**.

Man muss sich mittels `Minecraft.create()` mit dem laufenden Minecraft verbinden und kann dann direkt interagieren.

Das einfachste Beispiel ist hier einen Text in den Minecraft Chat zu schreiben

```
from mcpi import minecraft

mc = minecraft.Minecraft.create()

mc.postToChat("Hallo Minecraft")
```

Mit dem Befehl `mc.player.getPos()`, bekommt man die aktuelle Position des Spielers ausgegeben. Diese ist in x, y, z aufgeteilt. Mit dem Befehl `setPos()` und den Koordinaten (x,y,z) als Parameter, kann man den Spieler dann an einen beliebigen Ort stellen.

Mittels dem Befehl `setBlock(x, y, z [,id])` kann an der angegebenen Koordinate, einen Block platzieren. Mit dem optionalen Parameter **id** wird die Art des Blocks definiert, Beispiel für Block Typen:

ID	Typ
0	Luft
1	Stein
2	Gras
3	Erde/Dreck

Es ist auch Möglich Blöcke als Konstanten (vordefinierte Variablen), zu laden.

```
from mcpi import block

mc.setBlock(x, y, z, block.DIRT.id)
```

Mehrere Blocks kann man mit dem Befehl `setBlocks(x1, y1, z1, x2, y2, z2, id)` setzen, wobei x1,y1,z1 die Startposition setzt und x2,y2,z2 die Endposition.

Schleifen / Wiederholungen

Soll ein Befehl mehrmals ausgeführt werden, so möchte man diesen nicht mehrmals untereinander schreiben, sondern mittels einer Schleife immer und immer wieder ausführen. Es gibt verschiedene Arten von Schleifen, die im folgenden einzel betrachtet werden.

Die while Schleife

Die **while**-Schleife beginnt nach dem Schlüsselwort mit einer Bedingung, identisch zu den Programmverzweigungen **if**. Danach kommt der Verarbeitungsblock, welcher so oft ausgeführt wird, bis die Bedingung nicht mehr erfüllt wird.
Das heisst, nach jedem Durchlauf des Verarbeitungsblocks wird geprüft, ob die Bedingung noch erfüllt wird. Ist dies der Fall, wird der Verarbeitungsblock erneut abgearbeitet. Sobald die Bedingung nicht mehr erfüllt wird, wird der Block übersprungen und das Programm läuft weiter

```
while <Bedingung>:
    <Verarbeitungsblock>
```

Möchte man eine Endlos-Schleife programmieren, kann man statt einer Bedingung auch einfach **True** angeben. Hier sollte man allerdings im Verarbeitungsblock ein time.sleep einbauen. In der Konsole muss man Programm mittels *STRG+C* beenden, um aus der Schleife wieder herauszukommen.

```
while True:
    print("Hallo Welt!")
    time.sleep(1)
```

Die for Schleife

Statt der Bedingung, wie bei der while-Schleife, hat die **for**-Schleife einen Deklarationsteil. In diesem wird eine Variable definiert, welche im Verarbeitungsblock verwendet und ausgewertet werden kann und eine Sequenz, die angibt wie oft die Schleife durchlaufen werden soll.

```
for <Variable> in <Sequenz>:
    <Verarbeitungsblock>
```

Für die Sequenz wird meist ein Bereich (Range) verwendet. Neben einem Bereich, kann man auch eine Liste verwenden die durchlaufen werden soll, so wird die Schleife für jeden Eintrag in der Liste genau einmal durchlaufen.

```
for zahl in range(1, 10):
    print("Durchlauf " + zahl)

for frucht in ["Apfel", "Birne", "Orange"]:
    print("Frucht " + frucht + " gefunden")
```

Eigene Funktionen

Will man einen bestimmten Programmteil an verschiedenen Stellen wiederverwenden, ist es einfacher für diesen eine Funktion zu definieren, anstatt diesen zu kopieren und damit zu vervielfältigen. Wird nämlich der Programmteil im Nachhinein verändert, müsste jede Kopie angepasst werden. Durch die Verwendung von Funktionen, muss man die Anpassung nur einmal durchführen.

```
def <Funktionsname>(<Parameter Liste>):
  <Verarbeitungsblock>
```

Der Funktionsname kann nach den gleichen Regeln wie bei Variablennamen festgelegt werden, sollte aber sprechend für die Funktion gewählt werden. In der nachfolgenden Klammer, kann man eine Liste möglicher Parameter angeben, die durch Komma getrennt werden. So ist es möglich den Ablauf der Funktion durch unterschiedliche Werte zu beeinflussen. Im Verarbeitungsblock folgt der eigentliche Programmcode, der ausgeführt werden soll. Beispiel:

```
def aktuelleZeitAusgeben():
  print(time.strftime("%H:%M:%S"))
```

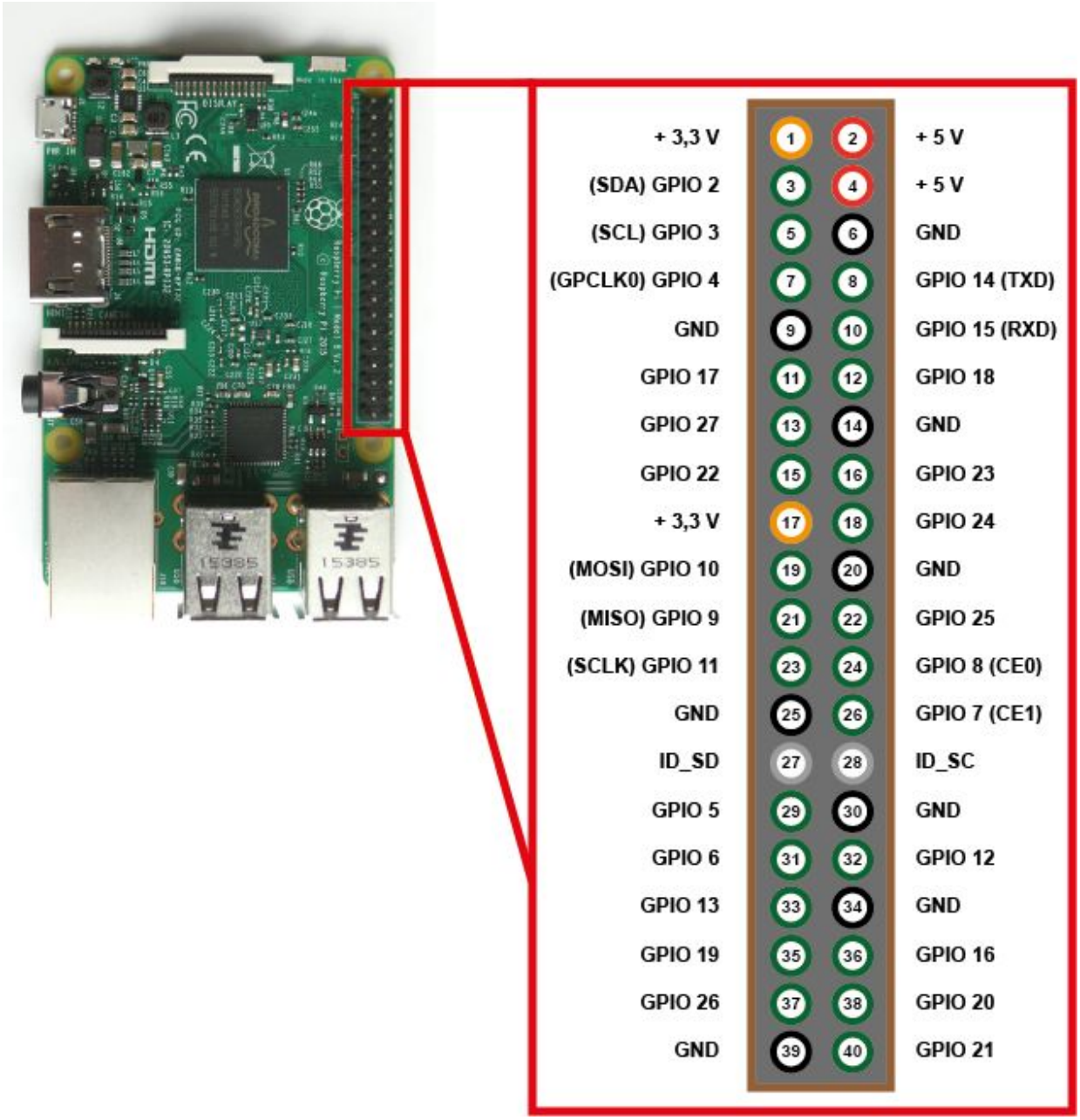
Die Funktion *aktuelleZeitAusgeben*, gibt die aktuelle Uhrzeit in einer bestimmten Formatierung aus. Es wird nur eine Zeile Code durch eine andere ersetzt, allerdings hat man den Vorteil, dass der Name der Funktion sprechender ist. Zum anderen hat es den Vorteil, wenn das Format der Zeitangabe geändert werden soll, muss man dies nur an einer Stelle im Code machen.

Der Aufruf der Funktion ist identisch zu anderen Funktionen:

```
aktuelleZeitAusgeben()
```

GPIO

Neben der einfachen Programmierung bietet der Raspberry Pi auch eine einfache Möglichkeit elektronische Komponenten einzubinden. Dies erfolgt mittels der **GPIO-Pinleiste** (general purpose input/output).



Pi Pinout

Verwendung des GPIO

Um mittels Python auf die verschiedenen Pins des GPIO zugreifen zu können, benötigt man die Bibliothek **RPi.GPIO**. Mit dieser kann man die einzelnen Pin ansprechen und auslesen.

```
import RPi.GPIO as GPIO

GPIO.setmode(GPIO.BCM)
GPIO.setup(23, GPIO.OUT)

GPIO.output(23, GPIO.HIGH)
```

Dieses kurze Programm definiert, mit welcher Benennung man die Pins ansprechen will (`GPIO.setmode(GPIO.BCM)`). Danach wird festgelegt, dass Pin 23 als Ausgang verwendet wird (`GPIO.setup(23, GPIO.OUT)`) und als letzter Schritt, wird dieser Ausgang "angeschaltet" (`GPIO.output(23, GPIO.HIGH)`), also mit 5 Volt versorgt. Hängt an dem Pin z.B. eine LED, so würde diese nun leuchten.

> - `GPIO.setmode(GPIO.BCM)`
> Durch diese Einstellung wird die Nummerierung anhand der Beschriftung verwendet, also GPIO23 ist dann im Code Pin 23.
> - `GPIO.setmode(GPIO.BOARD)`
> Hier werden die Pins nach Nummerierung durchgezählt (1-40). Pin GPIO23, wäre also in diesem Fall Pin 16.

Neben der Definition `GPIO.OUT` für einen Ausgang, gibt es auch noch die Möglichkeit einen Pin des GPIO als Eingang zu definieren. Dafür konfiguriert man mittels der bekannten setup-Funktion den Pin als Eingang `GPIO.setup(17, GPIO.IN)` . Wenn ein Signal am Pin anliegt, bekommt man einen Wert 1 und wenn kein Signal anliegt den Wert 0. Dies kann z.B. verwendet werden, um einen Taster-Zustand einzulesen (0 = nicht gedrückt / 1 = gedrückt).

Den aktuellen Wert ermittelt man über die Funktion `GPIO.input(17)`. Damit bekommt man den Wert zu dem Zeitpunkt, an dem diese Funktion ausgeführt wird. Möchte man dauernd den Zustand ermitteln, um z.B. bei einem Tastendruck zu reagieren, dann müsste man dafür eine Endlosschleife verwenden:

```
import RPi.GPIO as GPIO

GPIO.setmode(GPIO.BCM)
GPIO.setup(17, GPIO.IN)

while True:
    if GPIO.input(17) > 0:
        print("Taster gedrückt")
```

Die Endlosschleife verursacht aber eine gewisse Last auf dem Pi. Daher ist es vielleicht vorzuziehen, einen Callback Interrupt auf dem Eingang zu registrieren. Damit wird eine definierte Funktion (callback) aufgerufen, wenn eine Zustandsänderung am Eingang ermittelt wird (dabei kann man die Registrierung für ansteigende, abfallende oder beide Flanken festlegen).

```
GPIO.add_event_detect(<Pin>, <Flanke>, callback = <Funktion>)
```

Flanke kann RISING (steigend), FALLING (fallend) oder BOTH (steigen und fallend) sein. Für *Funktion* wird der Name zu rufenden Funktion verwendet, dabei ohne Klammern. Die Funktion muss einen Parameter haben, darüber wird der Funktion mitgeteilt, welcher Pin den Interrupt ausgelöst hat.

```
import RPi.GPIO as GPIO
import time

GPIO.setmode(GPIO.BCM)
GPIO.setup(17, GPIO.IN)
GPIO.setup(18, GPIO.IN)
GPIO.setup(19, GPIO.IN)

def Interrupt(channel):
    print("Interrupt erkannt")

GPIO.add_event_detect(17, GPIO.RISING, callback = Interrupt)
GPIO.add_event_detect(18, GPIO.FALLING, callback = Interrupt)
GPIO.add_event_detect(19, GPIO.BOTH, callback = Interrupt)

try:
    while True:
       time.sleep(1)
except KeyboardInterrupt:
    GPIO.cleanup()
```

Mit `GPIO.cleanup()` werden nach dem Programmabbruch mit CRTL+C alle Ein- und Ausgänge zurückgesetzt und die Interrupts gelöscht.

Nebenbei wurde in dem Beispiel noch die Fehlerbehandlung mit `try / except` eingeführt. Im Verarbeitungsblock `try`, wird ein Code Teil ausgeführt, der einen Fehler verursachen kann. Kommt es zu einem Fehler, wird das Programm nicht fehlerhaft beendet, sondern es wird der `except` Verarbeitungsblock ausgeführt. Dies kann nützlich sein, wenn man z.B eine Netzwerkverbindung offen hat, und diese sicher geschlossen werden soll, egal ob es Fehler gab oder nicht.

Klassen

Möchte man Funktionen logisch miteinander gruppieren und z.B. anderen zur Verfügung stellen, empfiehlt es sich, Klasse zu verwenden. Eine Klasse definiert dabei Eigenschaften und Funktionen, die an etwas ausgeführt werden können. Ein Beispiel für eine Klasse wäre ein Auto. Ein Auto hat Eigentschaften wie die Motorleistung oder die Anzahl Sitzplätze. Zusätzlich kann ein Auto Funktionen haben, wie gas geben oder bremsen.

Mit dem Schlüsselwort `class` wird eine Klasse definiert, danach kann man Klassenvariablen und Funktionen definieren.

Dateiname auto.py:

```
class Auto:
    __init__(sitze, ps):
        self.sitze = sitze
        self.ps = ps

    def gas_geben():
        print("Auto gibt gas, mit der Leistung von %s PS" %
self.ps)

    def bremsen():
        print("Auto bremst")
```

Verwendet werden kann das ganze dann so:

```
from auto import Auto

meinAuto = Auto(4, 120)
meinAuto.gas_geben()
meinAuto.bremsen()
```

In dem Beispiel wurde eine Standard-Funktion `__init__()` verwendet, diese Funktion wird automatisch beim Erzeugen der Klasse (des Klassen-Objekts) ausgeführt. In dieser Methode können initiale Werte Übergeben (wie im Beispiel die Werte für Sitze und PS) und die Funktion kann bestimmte Initialisierungsaufgaben ausführen (z.B. den GPIO-Mode setzen), die auf jeden Fall beim Erzeugen ausgeführt werden müssen.