

Hands-On Computer Science

Nicolas Meseth

17. November 2025

Inhaltsverzeichnis

Vorwort	6
Für wen ist dieses Buch gedacht?	6
Was macht dieses Buch anders?	6
So lernen wir: Hands-on von Anfang an	7
Experimente im Überblick	7
Die Hardware: unser Experimentier-Set	7
So nutzt ihr dieses Buch am besten	8
Fehler als Lernmotor: Frust gehört dazu	9
Empfehlenswerte Bücher	9
Auf geht's	10
Vorbereitung	11
Benötigte Software	11
Erweiterung für Visual Studio Code	12
Beispielcode herunterladen	12
Code über Git herunterladen (empfohlen)	12
Code als ZIP-Datei herunterladen	12
Virtuelle Python-Umgebung einrichten (optional)	12
Abhängigkeiten installieren	13
Umgebung testen	14
Los geht's!	14
1 Farben	15
Zusammenfassung	15
1.1 Experimentaufbau	15
1.1.1 Hardware	15
1.1.2 Erste Schritte mit der LED	16
1.2 Erstes Programm: LED ansteuern	16
1.2.1 Programme	21
1.2.2 Boilerplate Code	21
1.2.3 Bibliotheken	21
1.2.4 Klassen und Objekte	22
1.2.5 Schlüsselwörter	23
1.2.6 Objekte erzeugen	23
1.2.7 Methoden	23

1.2.8	Ein Objekt für die LED	24
1.2.9	Zusammenfassung unseres ersten Programms	24
1.2.10	Und jetzt?	24
1.3	Licht und Farben	25
1.3.1	Blick auf die Physik	25
1.3.2	Additive Farbmischung	27
1.3.3	Subtraktive Farbmischung	27
1.4	Pulsierende LED	29
1.4.1	Abzählbare Wiederholungen	31
1.4.2	Bedingte Wiederholungen	33
1.5	Farbkreise	35
1.6	Regenbogen-LED	37
1.6.1	Runde für Runde	39
1.6.2	Geschwindigkeit steuern	40
2	Zahlen	44
	Zusammenfassung	44
2.1	Experimentaufbau	44
2.1.1	Hardware	44
2.1.2	Erste Schritte mit dem Drehknopf	45
2.2	Zähler auslesen	45
2.3	Kontrollstrukturen	50
2.4	LED-Dimmer 1.0	50
2.5	Zahlensysteme	52
2.5.1	Unser Dezimalsystem	53
2.5.2	Das Oktalsystem	54
2.5.3	Das Binärsystem	55
2.5.4	Andere Systeme	58
2.6	Bits & Bytes	59
2.6.1	Zwei Zustände	59
2.6.2	Acht Bits macht ein Byte	60
2.6.3	Kilo, Mega, Giga	62
2.7	LED-Dimmer 2.0	62
2.7.1	<code>min()</code> und <code>max()</code>	63
2.7.2	Helligkeit entkoppeln	64
2.7.3	Konstanten	65
2.8	Druckknopf auslesen	67
2.9	LED-Dimmer 3.0	68
2.9.1	Farbe per Variable steuern	68
2.9.2	Farbe per Knopfdruck ändern	69
2.10	Funktionen	71

3 Texte	76
Zusammenfassung	76
3.1 Experimentaufbau	76
3.1.1 Hardware	76
3.1.2 Erste Schritte mit dem Abstandssensor	77
3.2 Lichtschranke	77
3.3 Hinderniserkennung	83
3.4 Universelles Eingabegerät	84
3.5 Texte kodieren	86
3.5.1 Wie viele Bits benötigen wir?	87
3.5.2 Wörterbücher	93
3.6 ASCII-Code	94
3.6.1 Unicode	97
3.7 LED-Dimmer 4.0	98
4 Bilder	101
Zusammenfassung	101
4.1 Experimentaufbau	101
4.1.1 Hardware	101
4.1.2 Erste Schritte mit dem Display	102
4.2 Pixel	103
4.3 Bitmaps	106
4.3.1 Quadrate und andere Formen	106
4.3.2 Buchstaben	110
4.4 Vektorgrafiken	114
4.5 Von Bits zum Bild	116
4.5.1 Excel mit Python einlesen	118
4.5.2 Zeile für Zeile die Pixelwerte extrahieren	119
4.5.3 Hexadezimale Farbwerte	121
4.5.4 Die Liste mit Bits erstellen	124
4.5.5 Anzeige auf dem Display	124
4.5.6 Eine Bitmap speichern	127
4.6 Farbige Bilder	129
4.6.1 Bitmaps im RGB-Format	129
4.6.2 Struktur einer Bitmap-Datei	133
4.7 Bewegte Bilder	135
4.8 Transformationen	137
4.8.1 Graustufen	138
4.8.2 Schwarzweiß	144
4.8.3 Informationsverlust	147
4.9 Pacman	147
4.9.1 Von Farbe zu Schwarzweiß	147
4.9.2 Pacman-Animation	151

4.9.3	Bewegung	152
Literaturverzeichnis		154

Vorwort

Glückwunsch, ihr seid angekommen! Egal, wie euer Weg hierher aussah: Ihr habt dieses Buch geöffnet – und damit den ersten Schritt getan. Vielleicht studiert ihr an der Hochschule Osnabrück und seid im Modul dabei, vielleicht seid ihr aus Neugier hier. So oder so: Willkommen!

Dieses Buch ist aus meiner Lehrpraxis an der Hochschule Osnabrück entstanden. Es dient als Hauptlektüre in meinen Veranstaltungen, als Nachschlagewerk für verpasste Sitzungen und als kompakte, praxisnahe Einführung für alle, die sich eigenständig in die digitale Welt einarbeiten wollen. Die wissenschaftliche Disziplin dahinter heißt auf Deutsch Informatik, international Computer Science. Der Titel Hands-On Computer Science verrät schon den Ansatz: Wir lernen praktisch – von Anfang an.

Für wen ist dieses Buch gedacht?

Lehrbücher zur Informatik gibt es viele. Doch nicht jedes passt zu dem, was ich mit meinen Studierenden erreichen möchte. Meine Zielgruppe seid ihr:

- Studierende aus Studiengängen wie [Management nachhaltiger Ernährungssysteme](#), [Lebensmittelproduktion](#) oder [Agrarsystemtechnologien](#).
- Quereinsteiger:innen, Wiederholer:innen und neugierige Menschen – auch ohne Bezug zur [Hochschule Osnabrück](#).

Kurz gesagt: Dieses Buch ist für alle, die in die digitale Welt eintauchen wollen, ohne sich in Details zu verlieren. Ihr braucht keinen dicken Wälzer, sondern einen roten Faden, der euch Schritt für Schritt zu den grundlegenden Konzepten führt – mit Spaß und Erfolgserlebnissen.

Was macht dieses Buch anders?

Viele Bücher versprechen Praxisnähe, enden aber in abstrakten Übungen am Kapitelende. Hier gehen wir es anders an:

1. Ihr entdeckt informatische Konzepte über Experimente mit Microcontrollern, Sensoren, Buttons, LEDs und Displays.

2. Wichtige Ideen führen wir früh ein, vertiefen sie nach und nach und wiederholen sie regelmäßig.
3. Theorie und Praxis gehören zusammen: Ihr lernt Programmieren und Informatik-Grundlagen gleichzeitig.

So lernen wir: Hands-on von Anfang an

Habt ihr euch schon einmal gefragt, wie man Informationen mit Licht überträgt? Wie man mit Licht den Puls messen kann? Oder wie man mit zwei einfachen Kabeln einen Wasserstandssensor baut? Genau solche Fragen beantworten wir – nicht nur theoretisch, sondern praktisch.

In jedem Kapitel erproben wir ein neues Experiment und lernen dabei eine Facette der digitalen Welt kennen. Gleichzeitig wächst euer Programmierwissen organisch mit. Wenn alles gut läuft, merkt ihr kaum, wie schnell ihr vorankommt.

Experimente im Überblick

Kapitel für Kapitel arbeitet ihr an Experimenten, die Hardware und Software verbinden. Dabei geht es um mehr als das Zusammenschrauben von Komponenten: Ihr lernt, Computer als universelle Problemlösungsmaschinen zu nutzen – für eure eigenen Ideen.

Kapitel	Experiment(e)
Kapitel 1	Wir lassen eine LED einen Regenbogenfarbverlauf über die Zeit erzeugen.
Kapitel 2	Wir entwickeln einen Dimmer für die LED, der über einen Drehknopf gesteuert wird.
Kapitel 3	Wir lernen, wie man Texte ganz ohne Tastatur eingeben kann – über Handgesten.
Kapitel 4	Wir verbinden Tabellenkalkulation mit Bildern und Displays

Die Hardware: unser Experimentier-Set

Hier seht ihr die Geräte, mit denen wir experimentieren. Zusammen kosten alle Komponenten ca. 249 €. Keine Sorge: Wenn ihr dieses Buch im Rahmen meines Moduls „Digitalisierung und Programmierung“ an der Hochschule Osnabrück nutzt, erhaltet ihr für die Zeit des Semesters ein komplettes Hardware-Kit kostenlos.

Was?	Bauteil	Anzahl	Preis pro Stück
Bunte LED	RGB LED Bricklet 2.0	1	8 €
Drehknopf mit Zählerfunktion	Rotary Encoder Bricklet	1	8 €
Infrarot-Entfernungsmesser	Distance IR 4-30cm Bricklet 2.0	1	20 €
OLED Display	OLED 128x64 Bricklet	1	25 €
Button mit integrierter, bunter LED	RGB LED Button Bricklet	1	15 €
Licht- und Farbsensor	Color Bricklet 2.0	1	17 €
Piezo Lautsprecher	Piezo Speaker Bricklet 2.0	1	19 €
Analoger Spannungssensor	Analog In Bricklet 3.0	1	14 €
Schalldruckpegelsensor	Sound Pressure Level Bricklet	1	35 €
Mikrocontroller	Master Brick 3.2	2	35 €
Anschlusskabel 15 cm	Bricklet Kabel 15cm (7p-7p)	8	1 €
USB-A-auf-USB-C-Kabel	USB-A auf USB-C Kabel 100 cm	1	6 €
Montageplatte	Montageplatte 22x22 (12x12cm)	2	7 €
Schrauben, Abstandshalter und Muttern	Befestigungskit 12mm	4	2 €

So nutzt ihr dieses Buch am besten

Weil es hier viel ums Programmieren geht, findet ihr zahlreiche Codebeispiele. Wir verwenden als Einstiegssprache Python. Warum Python? Weil es in der Praxis weit verbreitet ist, eine klare, leicht lesbare Syntax hat und viele nützliche Bibliotheken den Einstieg erleichtern.

Codeblöcke sind im Text deutlich abgesetzt (grau hinterlegt, Schreibmaschinenschrift). Ein Beispiel mit Annotationen:

```

led.set_rgb_value(0, 0, 0)                                ①
led.set_rgb_value(255, 255, 255)                            ②

print("Diese Zeile hat keine Annotation")

# Lasse die LED blau aufleuchten                         ③
led.set_rgb_value(0, 0, 255)

```

- ① Schaltet die LED aus, weil der RGB-Code (0, 0, 0) die Farbe Schwarz ergibt.
- ② Schaltet die LED auf weißes Licht, weil dreimal 255 die Farbe Weiß ergibt.
- ③ Auch Kommentare sind für kurze Erläuterungen nützlich.

Wenn ihr das Buch online lest, erscheinen zu den kleinen Ziffern im Code beim Darüberfahren Tooltips mit Erklärungen. In der PDF- oder Druckversion stehen die Erläuterungen unter dem Codeblock.

Damit der Fokus beim schrittweisen Entwickeln auf den neuen Teilen liegt, lasse ich gelegentlich Abschnitte im Code aus und markiere das mit drei Punkten (...). Den vollständigen Code findet ihr am Ende eines Abschnitts und im GitHub-Repository zum Buch:

<https://github.com/winf-hsos/hands-on-computer-science-code>

Noch ein Tipp: Wenn ihr mit der Maus über einen Codeblock fahrt, erscheint rechts oben ein Clipboard-Symbol. Ein Klick darauf kopiert den Code in eure Zwischenablage – ideal, um ihn in Visual Studio Code oder eine andere IDE einzufügen. In der Online-Version lassen sich manche Codeblöcke einklappen, damit ihr weniger scrollen müsst.

Fehler als Lernmotor: Frust gehört dazu

Eins vorweg: Beim Programmierenlernen ist Frust normal – und nützlich. Computer sind präzise, gnadenlose Lehrer. Ein vergessener Punkt, ein Buchstabe zu viel, ein Zahlendreher: Sofort gibt es Feedback. Das kann nerven, beschleunigt aber euren Lernprozess enorm. Sobald ihr Fehlermeldungen als Hinweise versteht und gezielt damit umgeht, kommen die Erfolgsergebnisse schnell.

Wenn etwas nicht klappt: Atmet durch, nehmt es nicht persönlich und versucht es erneut. Fehler sind unvermeidbar – und wichtig. Es lohnt sich!

Empfehlenswerte Bücher

Auch wenn dieses Buch einen eigenen Weg geht, haben mich andere Werke inspiriert:



Abbildung 1: Ein frustrierter Frosch

- Code: The Hidden Language of Computer Hardware and Software von Charles Petzold: Ein Klassiker, der von einfachen Konzepten schrittweise zu komplexeren Themen führt – bis ihr gedanklich einen Computer nachgebaut habt.
- Abenteuer Informatik von Jens Gallenbacher.
- Computer Science: An Overview von J. Glenn Brookshear und Dennis Brylow: Ein umfassendes Lehrbuch mit breitem Überblick und klarer Struktur.
- The Way Things Work von David Macaulay: Erklärt Technologien unterhaltsam. Der Teil „The Digital Domain“ mit den Abschnitten „Making Bits“, „Storing Bits“, „Processing Bits“ und „Sending Bits“ hat die Struktur dieses Buches beeinflusst.

Auf geht's

Alles klar? Dann machen wir uns startklar für die Experimente! Dazu müssen wir ein paar Dinge auf unserem Rechner installieren.

Vorbereitung

Benötigte Software

Um die Experimente in diesem Buch selbst durchführen zu können, benötigt ihr folgende Software auf eurem Computer.

Software	Was ist das?	Download-Link
Visual Studio Code	Eine beliebte und kostenlose Entwicklungsugbung. Damit schreiben wir unsere Programme.	Download-Seite
Python	Die Programmiersprache, die wir in diesem Buch verwenden. Ebenfalls weit verbreitet und kostenlos.	Download-Seite
Git	Ein kleines Werkzeug, mit dem wir Codestände verwalten können. Damit kommt ihr am einfachsten an die Codebeispiele aus diesem Buch.	Download Seite
Brick Daemon	Ein Hintergrundprozess, der die Kommunikation mit der Tinkerforge-Hardware ermöglicht.	Download-Seite
Brick Viewer	Ein Tool, das eine grafische Benutzeroberfläche für die Interaktion mit der Tinkerforge-Hardware bietet.	Download-Seite

Klickt jeweils auf den Downloadlink, speichert die Installationsdatei auf eurem Computer, und installiert das Programm. Sobald ihr alles fertig installiert habt, macht ihr hier weiter.

Erweiterung für Visual Studio Code

Damit Visual Studio Code mit Python umgehen kann, benötigt ihr noch die Python-Erweiterung. Öffnet Visual Studio Code, klickt links auf das Erweiterungssymbol (das Quadrat-Symbol) und sucht nach "Python". Installiert die Erweiterung von Microsoft. Solltet ihr sie nicht finden, könnt ihr auch [hier](#) klicken.

Beispielcode herunterladen

Ich stelle für dieses Buch die fertigen Programme nach jedem Abschnitt auf GitHub bereit. Ihr könnt euch den Code von dort ganz einfach auf euren Rechner herunterladen und jederzeit darauf zugreifen. Um den Code herunterzuladen, gibt es zwei Wege.

Code über Git herunterladen (empfohlen)

Wenn ihr Git erfolgreich installiert habt, könnt ihr den Code über die Kommandozeile (Terminal) herunterladen. Öffnet dazu ein Terminalfenster (unter Windows: PowerShell oder Command Line, unter macOS: Terminal), wechselt in ein Verzeichnis, in dem ihr den Code speichern möchtet, und gebt folgenden Befehl ein:

```
git clone https://github.com/winf-hsos/hands-on-computer-science-code
```

Bei funktionierender Internetverbindung wird der Code heruntergeladen und in einem neuen Ordner namens `hands-on-computer-science-code` gespeichert. Ihr könnt diesen Ordner dann in Visual Studio Code öffnen, um auf die Codebeispiele zuzugreifen.

Code als ZIP-Datei herunterladen

Ihr könnt den aktuellen Stand von GitHub über herunterladen, indem ihr [hier](#) klickt. Speichert die ZIP-Datei auf eurem Computer und entpackt sie an einen Ort eurer Wahl. Anschließend könnt ihr den Ordner mit den Codebeispielen in Visual Studio Code öffnen.

Virtuelle Python-Umgebung einrichten (optional)

Um die Abhängigkeiten für die Codebeispiele zu installieren, ist es eine gute Idee, eine virtuelle Python-Umgebung zu verwenden. Dadurch wird sichergestellt, dass die benötigten Pakete isoliert von anderen Projekten installiert werden. Hier sind die Schritte, um eine virtuelle Umgebung einzurichten:

1. Öffnet ein Terminalfenster in Visual Studio Code (Terminal > Neues Terminal)
2. Navigiert zu dem Verzeichnis, in dem ihr den Code gespeichert habt.
3. Führt den folgenden Befehl aus, um eine virtuelle Umgebung zu erstellen:

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

Wenn ihr einen Mac-Rechner verwendet, müsst ihr anstelle des `python`-Befehls `python3` verwenden:

```
python3 -m venv .python-env
```

4. Aktiviert die virtuelle Umgebung mit dem folgenden Befehl:

Unter Windows:

```
.python-env\Scripts\activate
```

Unter macOS und Linux:

```
source .python-env/bin/activate
```

Abhängigkeiten installieren

Die Codebeispiele in diesem Buch verwenden (unter anderem) die Tinkerforge-Python-Bibliothek, um mit der Hardware zu kommunizieren. Diese Bibliothek müsst ihr einmalig installieren. Öffnet dazu ein Terminalfenster in Visual Studio Code (Terminal > Neues Terminal) und gebt folgenden Befehl ein:

```
pip install tinkerforge
```

Alternativ könnt ihr auch folgenden Befehl verwenden:

```
pip install -r requirements.txt
```

Damit installiert ihr alle Abhängigkeiten, die in der Datei `requirements.txt` aufgelistet sind. Damit seid ihr für die Experimente in diesem Buch bestens gerüstet.

Auch hier unterscheidet sich der Befehl auf einem Mac-Rechner. Dort müsst ihr `pip3` anstelle von `pip` verwenden:

```
pip3 install tinkerforge
```

Alternativ könnt ihr auch folgenden Befehl verwenden:

```
pip3 install -r requirements.txt
```

Umgebung testen

Um sicherzustellen, dass alles korrekt eingerichtet ist, könnt ihr ein kleines Testprogramm schreiben und ausführen. Erstellt eine neue Python-Datei in Visual Studio Code im Unterordner `workspace` und benennt sie `my_first_program.py`. Öffnet die neue Datei mit einem Doppelklick und fügt folgenden Code ein:

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

Speichert die Datei und führt sie aus, indem ihr im Terminal folgenden Befehl eingebt:

```
python my_first_program.py
```

Und für alle Mac-Nutzer unter uns:

```
python3 my_first_program.py
```

Es sollte nun die Ausgabe `Hello, World!` im Terminal erscheinen. Wenn das funktioniert, seid ihr bereit für die nächsten Schritte!

Los geht's!

Jetzt seid ihr bereit, mit den Experimenten zu starten! Viel Spaß beim Lernen und Experimentieren!

1 Farben

Zusammenfassung

Im ersten Kapitel steigen wir gleich voll ein und schreiben unser erstes Programm. Unser Ziel ist es, eine LED nacheinander in allen Farben des Regenbogens leuchten zu lassen.

Auf dem Weg dorthin gehen wir die folgenden Schritte.

#	Was?	Wo?
1	Wir lernen, wie man eine LED aus einem Programm heraus steuert.	Abschnitt 1.2
2	Wir werfen einen kurzen Blick auf Farben und wie sie im Computer erzeugt werden. Dabei lernen wir den Unterschied zwischen additiver und subtraktiver Farbmischung kennen.	Abschnitt 1.3
3	Wir programmieren eine pulsierende LED als erster Schritt in Richtung Regenbogenverlauf.	Abschnitt 1.4
4	Wir lernen den Hue-Farbverlauf kennen, weil wir den für einen schönen Regenbogen benötigen.	Abschnitt 1.5
5	Endlich - wir schreiben das Programm für den Regenbogenverlauf.	Abschnitt 1.6

1.1 Experimentaufbau

1.1.1 Hardware

Bereit für euer erstes Hardware-Experiment? Perfekt! Ihr braucht dafür eine LED ([RGB LED Bricklet 2.0](#)) und einen Mikrocontroller ([Master Brick 3.2](#)). Befestigt beide Bauteile mit Abstandshaltern auf einer Montageplatte, wie in Abbildung 1.1 gezeigt. Zwei Schrauben pro Gerät reichen völlig. Denkt an die kleinen, weißen Unterlegscheiben aus Kunststoff. Sie schützen eure Platinen vor Druckstellen.

Die vollsändige Hardwareliste für dieses Kapitel sieht so aus:

- 1 x [Master Brick 3.2](#)
- 1 x [RGB LED Bricklet 2.0](#)

- 1 x Montageplatte 22x10
- 1 x Brickletkabel 15cm (7p-7p)
- 1 x Befestigungskit 12 mm

Neben der Hardware benötigt ihr auch die passende Software. Diese solltet ihr bereits installiert haben. Falls nicht, schaut im [Abschnitt zu den Voraussetzungen](#) vorbei. Dort ist alles genau beschrieben. Im Folgenden gehe ich davon aus, dass ihr alles am Laufen habt.

1.1.2 Erste Schritte mit der LED

Im ersten Schritt wollen wir die LED und ihre Funktionen testen! Das geht ganz leicht mit dem Brick Viewer. Schließt zuerst den Master Brick über das USB-Kabel an euren Computer an und öffnet den Brick Viewer. Klickt dann auf den Connect-Button.

Wenn alles geklappt hat, zeigt euch der Brick Viewer alle angeschlossenen Geräte in Tabs an. Schaut euch Abbildung [1.3](#) an – so etwa sollte es aussehen.

Wechselt nun zum Tab der RGB LED. Hier könnt ihr auf unterschiedlichen Wegen die Farbe der LED einstellen. Mehr kann eine LED nicht!

Mit den drei Schiebereglern steuert ihr die einzelnen Farbkanäle – Rot, Grün, Blau. Der Wertebereich: 0 bis 255. Warum gerade diese Farben und diese Zahlen? Gute Frage. Die Antwort kommt weiter unten.

Fazit: Der Brick Viewer ist ideal zum Ausprobieren. Aber wenn ihr echte Projekte umsetzen wollt, müsst ihr programmieren lernen. Also los!

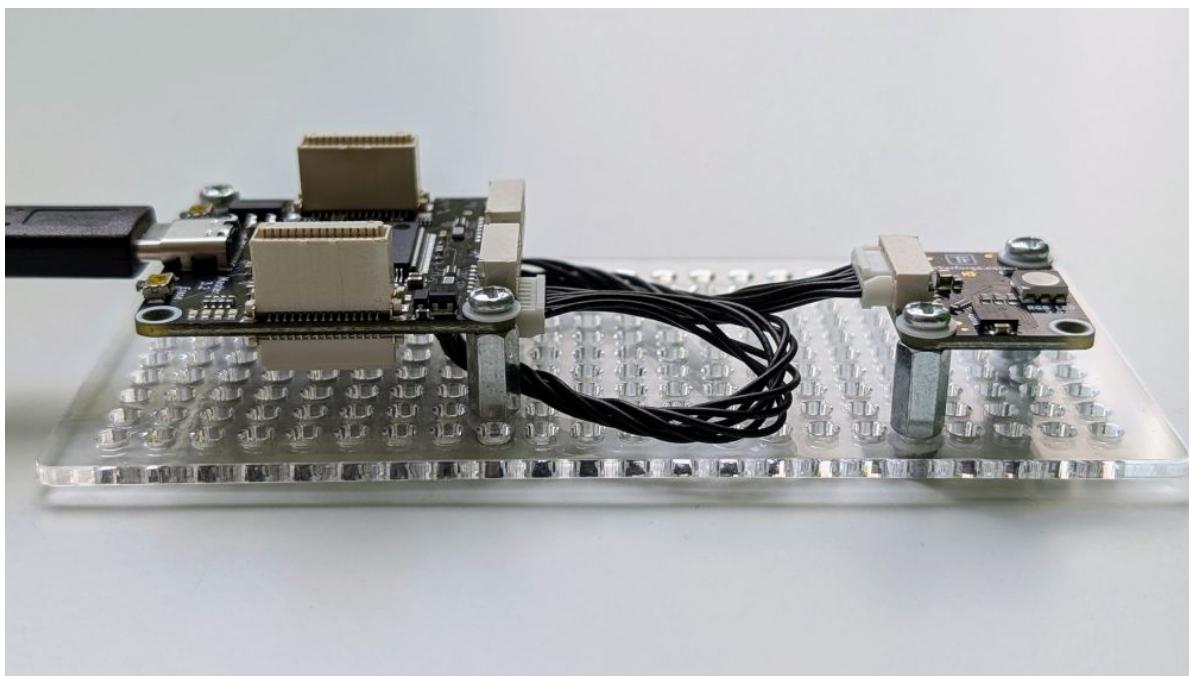
1.2 Erstes Programm: LED ansteuern

Wie verbinden wir uns über ein Programm mit der LED und setzen ihre Farbe? Die Antwort darauf findet ihr im folgenden kurzen Codebeispiel.

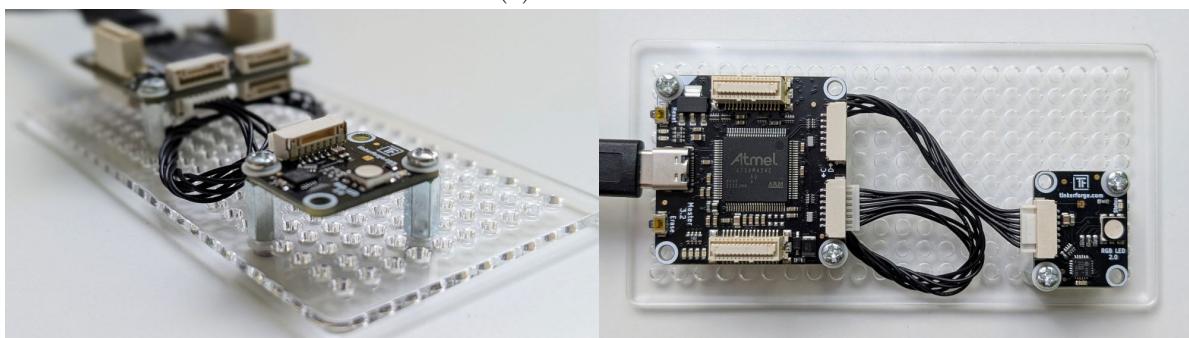
Listing 1.1 Der Boilerplate-Code für die Verbindung mit den Geräten am Beispiel der RGB LED.

```
from tinkerforge.ip_connection import IPConnection          ①
from tinkerforge.bricklet_rgb_led_v2 import BrickletRGBLEDV2    ②

ipcon = IPConnection()                                     ③
ipcon.connect("localhost", 4223)                            ④
led = BrickletRGBLEDV2("<YOUR_LED_UID>", ipcon)           ⑤
```

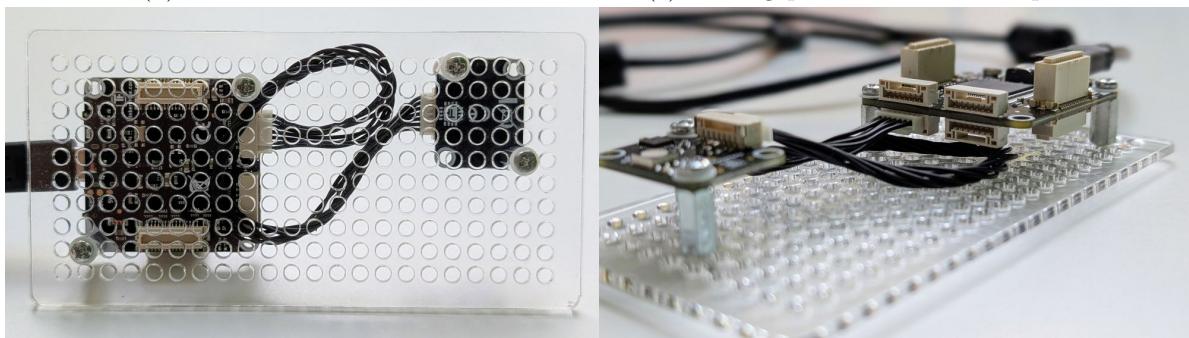


(a) Seitenansicht.



(b) Nahaufnahme der LED.

(c) Montageplatte mit allen Komponenten.



(d) Untenansicht.

(e) Ansicht der vier Steckplätze.

Abbildung 1.1: Einfaches Setup mit einem Mikrocontroller und einer LED.

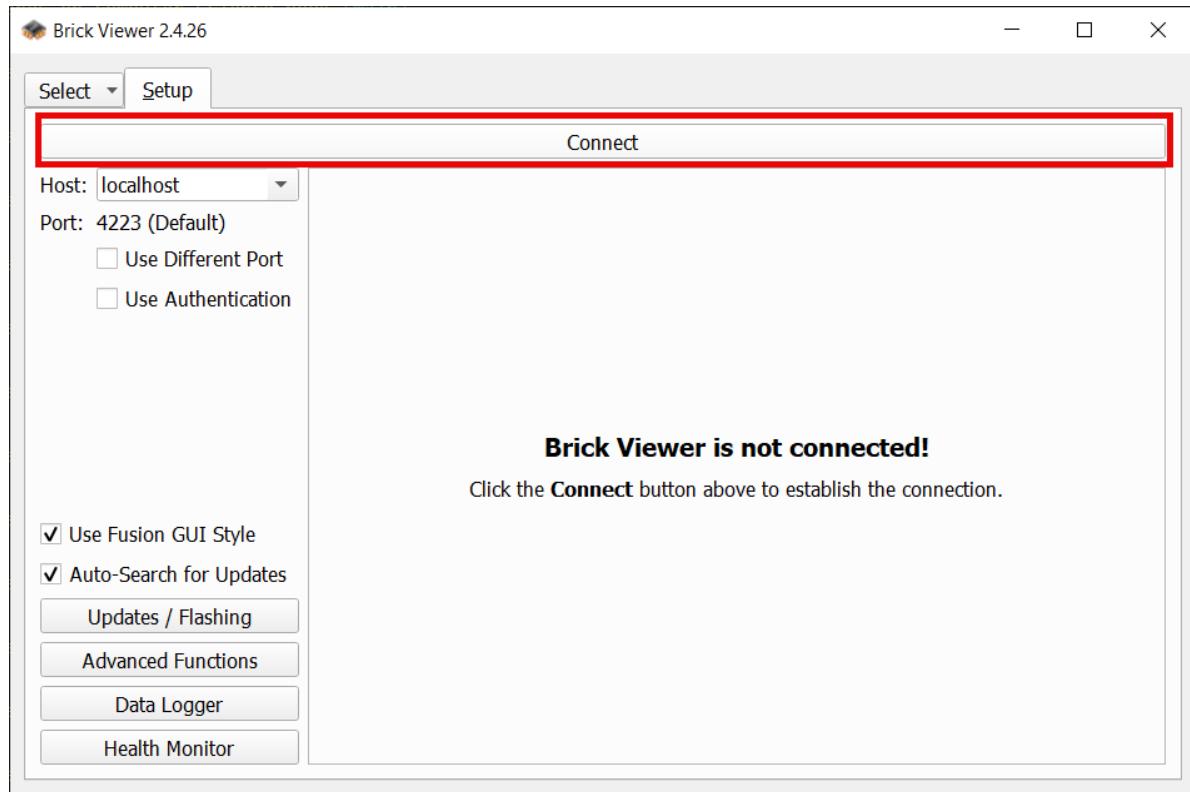


Abbildung 1.2: Über den Connect-Button verbindet ihr den Brick Viewer mit dem angeschlossenen Master Brick.

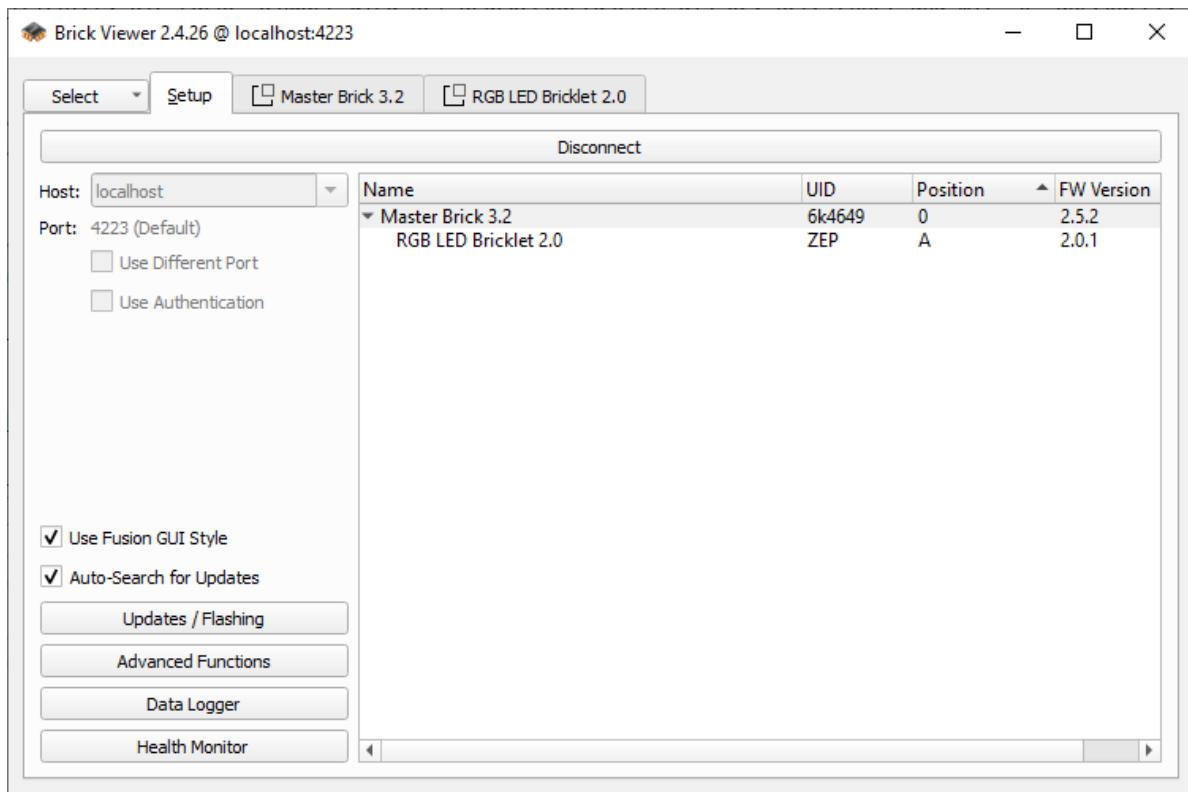


Abbildung 1.3: Der Brick Viewer, nachdem ihr mit dem Master Brick verbunden seid.

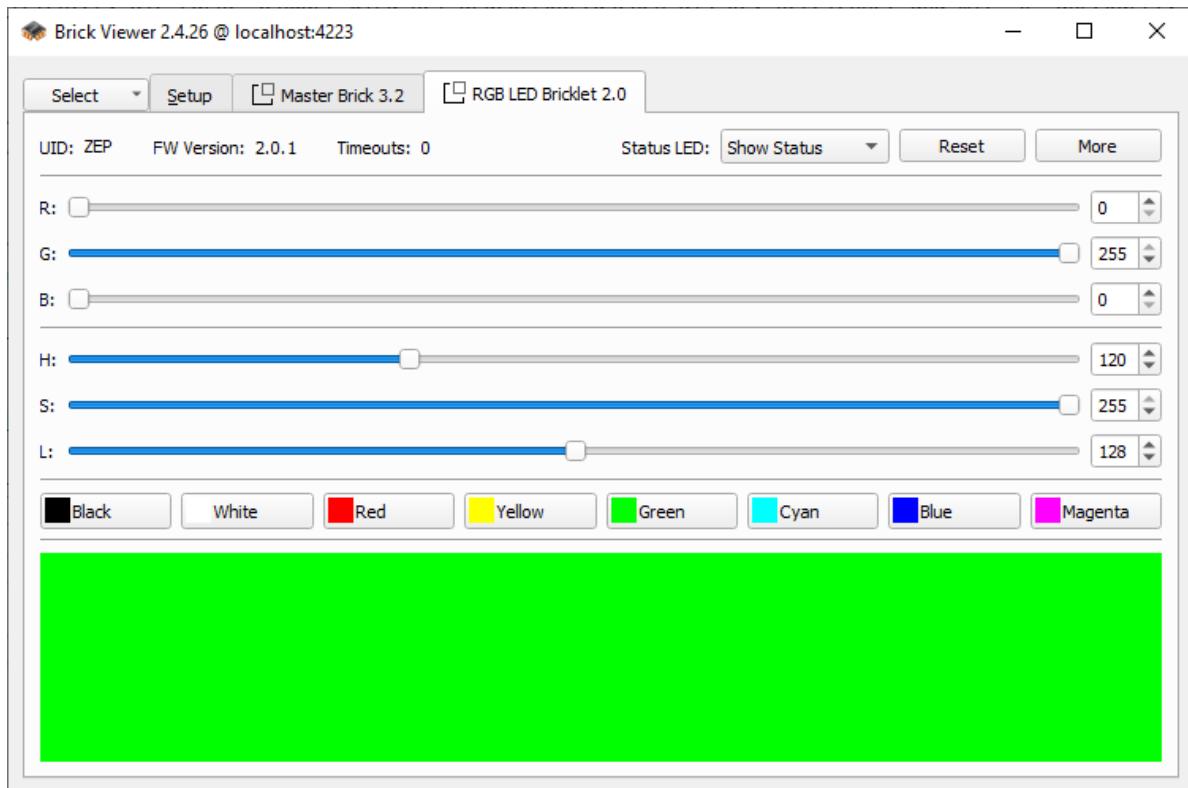


Abbildung 1.4: Die Ansicht für die RGB LED im Brick Viewer, in der ihr alle Funktionen per Klick im Zugriff habt.

- ① Hier importieren wir ein Objekt aus einer Bibliothek zum Herstellen einer Verbindung mit dem Master Brick.
- ② Hier importieren wir ein weiteres Objekt, das wir zur Darstellung der LED als Python-Objekt benötigen.
- ③ Die Verbindung erfolgt über eine sogenannte IP-Verbindung, die wir hier als Objekt erstellen.
- ④ Mit `connect` stellen wir eine Verbindung zum angeschlossenen Master Brick her.
- ⑤ Schließlich holen wir uns eine virtuelle Instanz des RGB LED Bricklets, indem wir die UID nennen und sagen, welche Verbindung (`ipcon`) genutzt werden soll.

1.2.1 Programme

Zunächst klären wir den Begriff Programm. Ein Programm ist eine Abfolge von Anweisungen, die ein Computer ausführt, um eine bestimmte Aufgabe zu erledigen. In unserem Fall ist das Programm später dafür zuständig, mit der LED zu interagieren und sie in verschiedenen Farben leuchten zu lassen. Programme werden in Programmiersprachen geschrieben, die es uns ermöglichen, dem Computer präzise Anweisungen zu geben. Wir verwenden in diesem Buch die Programmiersprache Python, die sich besonders gut für Einsteiger eignet und gleichzeitig mächtig genug ist, um komplexe Aufgaben zu lösen.

Wenn wir ein Programm ausführen, arbeitet der Computer die Anweisungen Schritt für Schritt von oben nach unten ab. Die Nummerierung der Zeilen verdeutlicht das sehr schön. Der Computer beginnt bei Zeile 1 und arbeitet die Befehle Zeile für Zeile bis nach unten ab.

Es gibt Befehle, die den Computer von dieser linearen Abfolge abweichen lassen, etwa Schleifen oder Verzweigungen. Diese lernen wir später kennen.

1.2.2 Boilerplate Code

Der Codeausschnitt in Listing 1.1 wird uns im Verlauf dieses Buches immer wieder begegnen. Wir benötigen ihn, um uns am Anfang des Programms mit den Geräten zu verbinden, die wir für unsere Anwendung benötigen. In der Informatik nennen wir solchen Code, den wir häufig in der gleichen Form benötigen und fast eins zu eins kopieren können, auch Boilerplate-Code. Wundert euch also nicht, wenn ich diesen Begriff ab und an mal verwende. Jetzt wisst ihr, was gemeint ist.

1.2.3 Bibliotheken

Beginnen wir in den ersten beiden Zeilen. Hier seht ihr zwei sehr ähnliche Befehle, die mit dem Schlüsselwort `from` beginnen. Nach dem Schlüsselwort `from` folgt der Name einer Bibliothek, aus der wir ein für unser Programm benötigtes Objekt importieren. Die Kombination der

Schlüsselwörter `from ... import` lässt sich also wörtlich übersetzen: "Aus der Bibliothek X importiere das Objekt Y".

Eine *Bibliothek* in einer Programmiersprache ist die Bündelung und Bereitstellung von Funktionen, Klassen oder Konstanten. Eine Bibliothek könnt ihr euch vorstellen wie einen Werkzeugkasten: Sie enthält fertige Werkzeuge (Funktionen und Klassen), damit ihr nicht alles von Grund auf selbst programmieren müsst. Tinkerforge stellt uns genau solche Werkzeuge bereit, damit wir schnell und unkompliziert mit den Geräten loslegen können. Für jedes Gerät gibt es in der Tinkerforge-Bibliothek eine eigene Klasse, über die wir auf die Funktionen jedes Geräts zugreifen können.

1.2.4 Klassen und Objekte

Mit `from ... import` importieren wir also etwas aus einer Bibliothek. Soweit so gut. Aber was bedeutet das genau? Mit *importieren* ist konkret gemeint, dass wir dem Programm mitteilen, dass wir vorhaben, die genannten Dinge in unserem Programm zu verwenden, und dass sie deshalb am besten schon einmal geladen werden sollten. Ob wir diese Dinge später wirklich nutzen, steht auf einem anderen Blatt.

In dem Fall der ersten beiden Zeilen unseres Programms von oben sind es zwei Klassen, deren Verwendung wir ankündigen. Die erste Klasse heißt `IPConnection` und die zweite `BrickletRGBLEDV2`. Der Begriff Klasse ist hier analog zum Begriff *Kategorie* zu verstehen. Wir können zu einer Klasse gehörige Objekte erzeugen, und alle Objekte derselben Klasse verhalten sich gleich und haben die gleichen Funktionen. Das verstehen wir am besten an einem einfachen Beispiel.

Stellt euch vor, ihr habt eine Klasse namens `Auto`. Diese Klasse beschreibt alle Eigenschaften und Funktionen, die ein Auto haben kann, wie etwa `fahren()`, `bremsen()` oder `tanken()`. Diese Dinge sollen für jedes Auto gleich ablaufen. Jedes konkrete Auto in der Welt ist ein Objekt dieser Klasse. Wir können also sagen: "Mein Auto ist ein Objekt der Klasse `Auto`." Jedes `Auto` hat neben den Funktionen die gleichen Eigenschaften wie Farbe, Marke und Modell. Aber jedes Auto kann andere Werte für diese Eigenschaften haben.

Genauso verhält es sich mit den Klassen, die Tinkerforge für uns bereitgestellt hat. Die Klasse `IPConnection` beschreibt, wie wir eine Verbindung zu einem Mikrocontroller herstellen können, und die Klasse `BrickletRGBLEDV2` beschreibt, wie wir mit der LED interagieren können. Wenn wir ein Objekt dieser Klasse erstellen, können wir alle Funktionen nutzen, die in der Klasse definiert sind. Eine LED muss nicht fahren oder bremsen wie ein Auto. Dafür hat sie andere Funktionen, wie etwa `set_rgb_value()`, die uns erlaubt, die Farbe der LED zu ändern. Eine Eigenschaft jeder LED ist ihre UID, die eindeutig ist und uns hilft, sie im System zu identifizieren.

1.2.5 Schlüsselwörter

Soeben haben wir mit `from` und `import` unsere ersten beiden Schlüsselwörter in Python kennengelernt! Aber was bedeutet das genau? Ein Schlüsselwort, das wir im Englischen auch *keyword* oder *reserved keyword* nennen, ist ein Begriff, der in der jeweiligen Programmiersprache eine feste Bedeutung hat und deshalb nicht anderweitig verwendet werden darf. Wir werden gleich noch sehen, dass wir bei der Programmierung auch häufig Namen vergeben müssen, etwa für Variablen oder Funktionen. Diese Namen dürfen nicht wie ein Schlüsselwort lauten, ansonsten funktioniert unser Programm nicht wie gewünscht. Welche Schlüsselwörter es in Python gibt, könnt ihr [hier](#) nachschauen.

Im Codeausschnitt oben laden wir zuerst das Objekt für die Verbindung zum angeschlossenen Mikrocontroller, die über eine IP-Verbindung hergestellt wird. Was das genau ist? Später mehr dazu. Zusätzlich zur `IPConnection` laden wir anschließend noch die benötigten Klassen für die Geräte, die wir in unserem aktuellen Setup verwenden wollen. In diesem Kapitel ist das nur die LED, in späteren Experimenten werden es auch mal mehrere Geräte sein.

1.2.6 Objekte erzeugen

In Listing 1.1 in Zeile 4 erzeugen wir ein Objekt der Klasse `IPConnection`. Die fertige Instanz – so nennen wir ein Objekt, das aus einer Klasse erzeugt wurde – speichern wir auf einer *Variable* mit dem Namen `ipcon`. Diesen Namen haben wir uns selbst ausgedacht, damit wir später darauf zugreifen können. Wir hätten auch einen anderen Namen wählen können. Eine Variable ist also ein Platzhalter für einen Wert, den wir später im Programm verwenden wollen. In diesem Fall ist `ipcon` der Platzhalter für die Verbindung zu unserem Mikrocontroller. Was eine Variable technisch ist, lernen wir später noch genauer kennen.

1.2.7 Methoden

Über das Objekt `ipcon` können wir nun eine Verbindung zu unserem Mikrocontroller herstellen. Das geschieht in Zeile 5 mit der Methode `connect()`. Eine Methode ist eine Funktion, die zu einem Objekt gehört – wie etwa `fahren()` oder `bremsen()` in unserem Auto-Beispiel.

Wir können Methoden aufrufen, um eine bestimmte Aktion auszuführen. In diesem case stellen wir eine Verbindung zum Mikrocontroller her, indem wir die Adresse und den Port angeben, über den die Verbindung hergestellt werden soll. In unserem Fall ist das “localhost”, was für die lokale Maschine steht, und Port 4223, der durch den Brick Daemon standardmäßig so konfiguriert ist. Der Aufruf einer Methode erfolgt immer mit dem Punkt `.` nach dem Objekt, gefolgt vom Namen der Methode und den Klammern `()`, in denen wir eventuell benötigte Parameter angeben.

Eine Methode ist letztlich eine Funktion, die zu einem Objekt gehört. Zu einem späteren Zeitpunkt schreiben wir unsere eigenen Funktionen und lernen dann noch viel mehr darüber.

1.2.8 Ein Objekt für die LED

In Zeile 6 erzeugen wir schließlich ein Objekt der Klasse `BrickletRGBLEDV2`. Dieses Objekt repräsentiert unsere LED und ermöglicht es uns, mit ihr zu interagieren. Wir nennen das Objekt `led`, was kurz und klar ist. Auch hier haben wir uns den Namen selbst ausgedacht, um später darauf zugreifen zu können. Auch wenn wir grundsätzlich Variablennamen frei wählen können, sollten sie immer so gewählt werden, dass sie den Inhalt der Variable beschreiben. Das macht es später einfacher, den Code zu verstehen. Gleichzeitig gibt es in Python einige Regeln, die wir bei der Benennung von Variablen beachten müssen. Dazu gehören etwa, dass Variablennamen nicht mit einer Zahl beginnen dürfen und keine Leerzeichen enthalten dürfen. Eine ausführliche Liste der Regeln findet ihr [hier](#).

1.2.9 Zusammenfassung unseres ersten Programms

Damit haben wir unser erstes Programm von oben nach unten erläutert und dabei schon viele wichtige Konzepte der Programmierung kennengelernt:

Programme	Abfolge von Anweisungen, die nacheinander ausgeführt werden.
Boilerplate Code	Standard-Code, den man immer wieder braucht.
Importieren von Bibliotheken	Sammlung von fertigen Code-Elementen.
Schlüsselwörter	Reservierte Begriffe der Programmiersprache.
Klassen und Objekte	Kategorien und deren konkrete Instanzen.
Methoden und Funktionen	Funktionen, die zu einem Objekt gehören.
Variablen	Platzhalter für Werte.

1.2.10 Und jetzt?

Wir haben nun eine digitale Repräsentation unserer LED in Python. Wir können die LED jetzt zum Leuchten bringen, indem wir eine Methode der Klasse `BrickletRGBLEDV2`, die `set_rgb_value()` heißt, verwenden. Diese Methode erwartet drei Parameter: Rot, Grün und Blau. Mit diesen Parametern können wir die Farbe der LED einstellen.

```
led.set_rgb_value(0, 255, 0)
```

(1)

(1) Setzt die LED auf grün. R = 0, G = 255, B = 0. Logisch, oder?

Moment mal ... Wo steht hier eigentlich *grün*? Steht da gar nicht. Stattdessen drei Zahlen. Willkommen bei der RGB-Farbcodierung. Jede Farbe besteht aus drei Werten zwischen 0 und

255: Rot, Grün, Blau. Null ist nix. 255 ist volle Power. Alles 0? Schwarz. Alles 255? Weiß. Nur Grün auf 255? Na klar: grün.

Aber warum machen wir das mit Zahlen? Weil Computer nun mal mit Zahlen arbeiten. Das ist einer der zentralen Gedanken dieses Buches: Wie übersetzen wir die Welt in etwas, das ein Computer versteht?

Warum aber ist das so? Warum kodieren wir in der Informatik jede Farbe mit *drei* Zahlen? Warum überhaupt mit Zahlen? Hier kommen wir zu einer zentralen Frage dieses Buches: Wie bilden Computer Informationen ab?

Vorher müssen wir aber kurz zurück in die Schule.

Tipp

Das Codebeispiel aus diesem Abschnitt findet ihr auf [GitHub](#).

1.3 Licht und Farben

1.3.1 Blick auf die Physik

Physik ist vielleicht schon eine Weile her. Erinnern wir uns dennoch kurz, was Licht ist und wie Farben damit zusammenhängen. Licht ist elektromagnetische Strahlung. Das bedeutet, es handelt sich um gekoppelte Schwingungen elektrischer und magnetischer Felder, die sich mit Lichtgeschwindigkeit ausbreiten. Vereinfacht können wir uns Licht als Wellen vorstellen, die sich durch den Raum bewegen. Diese Wellen haben unterschiedliche Frequenzen und Wellenlängen. Das sichtbare Licht ist nur ein kleiner Teil des gesamten elektromagnetischen Spektrums, das von Radiowellen über Infrarotstrahlung bis hin zu Röntgenstrahlen und Gammastrahlen reicht.

Bei Wellen unterscheiden wir zwischen der Frequenz (wie oft die Welle pro Sekunde schwingt) und der Wellenlänge (der Abstand zwischen zwei aufeinanderfolgenden Wellenbergen). Die Frequenz und die Wellenlänge sind umgekehrt proportional: Je höher die Frequenz, desto kürzer die Wellenlänge und umgekehrt.

Frequenzen messen wir in Hertz (Hz), wobei 1 Hz einer Schwingung pro Sekunde entspricht. Das sichtbare Licht hat Frequenzen im Bereich von etwa 430 THz (Terahertz) bis 750 THz. Die Wellenlängen des sichtbaren Lichts liegen zwischen etwa 400 nm (Nanometer) für violettes Licht und etwa 700 nm für rotes Licht. Um sich das vorzustellen: Ein Nanometer ist ein Milliardstel Meter. Zum Vergleich: Ein menschliches Haar hat einen Durchmesser von etwa 80.000 bis 100.000 Nanometern. Die Abstände zwischen den Wellenlängen des sichtbaren Lichts sind also extrem klein.

Was bedeutet das nun für eine LED? Eine LED (Light Emitting Diode) ist ein Halbleiterbauelement, das Licht erzeugt, wenn elektrischer Strom hindurchfließt. Die Farbe des Lichts hängt von Eigenschaften des Halbleitermaterials ab, aus dem die LED besteht. Verschiedene Materialien emittieren Licht bei unterschiedlichen Wellenlängen, was zu verschiedenen Farben führt. Zum Beispiel emittiert eine rote LED Licht mit einer Wellenlänge von etwa 620–750 nm, während eine grüne LED Licht mit einer Wellenlänge von etwa 495–570 nm emittiert.

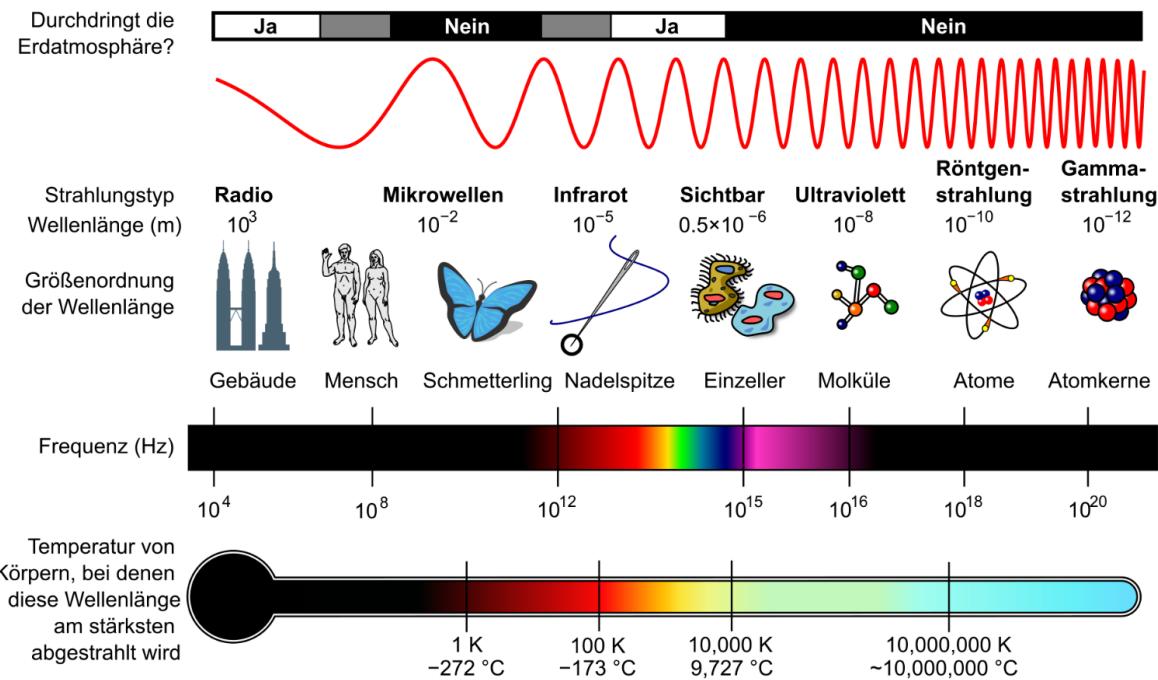


Abbildung 1.5: Das elektromagnetische Spektrum, von dem das sichtbare Licht ein kleiner Teil ist. (Quelle: [Wikipedia](#))

Die RGB LED besteht in Wirklichkeit aus drei einzelnen LEDs: einer roten, einer grünen und einer blauen. Jede dieser LEDs kann unabhängig voneinander angesteuert werden, um verschiedene Farben zu erzeugen. Mehr Stromstärke bedeutet mehr Intensität der jeweiligen Farbe. Durch die Kombination der drei Grundfarben Rot, Grün und Blau in unterschiedlichen Intensitäten können wir eine Vielzahl von Farben mischen. Das ist das Prinzip der additiven Farbmischung: Wenn wir alle drei Farben mit voller Intensität leuchten lassen, erhalten wir Weiß. Wenn wir keine Farbe leuchten lassen, erhalten wir Schwarz. Klar, die LEDs sind dann alle aus.

Jetzt wissen wir, warum die Methode `set_rgb_value()` drei Parameter erwartet: Rot, Grün und Blau. Diese Parameter sind die Intensitäten der jeweiligen Farbe, die wir in unserem Programm angeben. Mit den Werten 0 bis 255 können wir jede Farbe im sichtbaren Spektrum erzeugen.

Ein Farbwert im Computer besteht also aus drei Zahlen bestehend, die jeweils zwischen 0 und 255 liegen. Das gilt für unsere LED, aber auch für Pixel in TVs, Smartphones, digitalen Fotos oder Monitoren. Wie kommt es aber zu der merkwürdigen Zahl 255? Warum nicht einfach 0 bis 100? Das liegt daran, wie ein Computer grundsätzlich Werte speichert und wie dieser Speicher organisiert ist. Genauer erfahren wir schon in Kapitel 2.

Klingt alles theoretisch sehr gut. Aber wie sieht es mit der Praxis aus? Probieren wir es aus und mischen zwei Farben mit voller Intensität!

```
led.set_rgb_value(255, 255, 0)
```

Was macht der Befehl? Welche Farbe kommt dabei heraus? Probiert es einfach mal aus!

1.3.2 Additive Farbmischung

Ihr solltet alle eure LEDs in Gelb aufleuchten sehen. In der additiven Farbmischung* mischen wir Rot und Grün und erhalten dadurch Gelb. Gelb ist heller als die beiden Farben Rot und Grün, was kein Zufall ist. Das ist das Prinzip der additiven Farbmischung: Wenn wir zwei Farben mit voller Intensität leuchten lassen, erhalten wir eine neue Farbe, die stets heller ist als die Ursprungsfarben. Wir fügen mehr Licht hinzu. Wenn wir alle drei Farben mit voller Intensität mischen, erhalten wir schließlich Weiß.

```
led.set_rgb_value(255, 255, 255)
```

Am anderen Ende des Spektrums erzeugen drei Nullen die Farbe Schwarz:

```
led.set_rgb_value(0, 0, 0)
```

1.3.3 Subtraktive Farbmischung

Ihr könnt euch merken, dass wir im Kontext von Computern oft von *additiver Farbmischung* sprechen, weil Bildschirme Licht erzeugen. Durch das Mischen der drei Farbkanäle entstehen neue Farben gemäß der additiven Farbmischung, also stets heller als ihre Grundfarben. Daneben gibt es aber noch die subtraktive Farbmischung. Sie funktioniert anders, nämlich genau umgekehrt. Statt beim Mischen Licht hinzuzufügen, nehmen wir Licht weg.

Erinnert ihr euch an euren Farbkasten aus der Grundschule? Dort habt ihr auch Farben gemischt, um neue Farben zu erzeugen, die euer Farbkasten nicht direkt mitgeliefert hat. Was hat im Farbkasten die Mischung aus Rot und Grün ergeben? Sicher nicht Gelb – eher Braun. Eine dunklere Farbe. Das liegt daran, dass wir es hier nicht mit additiver, sondern mit subtraktiver Farbmischung zu tun haben. Bei der subtraktiven Farbmischung mischen wir Pigmente,

die Licht *absorbieren* und *reflektieren*. Das Mischen von Farben fungiert hier wie ein Filter: Bestimmte Teile des Lichtspektrums werden nicht mehr reflektiert, sondern absorbiert und sind damit nicht mehr sichtbar. Das Ergebnis einer Mischung zweier Farben ergibt in der subtraktiven Farbmischung also stets eine dunklere Farbe – genau umgekehrt zur additiven Farbmischung.

Was passiert mit dem absorbierten Licht? Es wird in eine andere Form der Energie umgewandelt, nämlich Wärme. Deshalb wird eine schwarze Oberfläche auch besonders heiß, wenn die Sonne darauf knallt. Sie absorbiert das gesamte Lichtspektrum und wandelt es in Wärme um. Dagegen wirken weiße Oberflächen fast wie Klimaanlagen. Es ist kein Zufall, dass wir in sonnigen Erdteilen viele weiße Fassaden sehen.

Wenn wir alle Farben mischen, ergibt die subtraktive Farbmischung Schwarz, weil kein Licht mehr reflektiert wird. Alles Licht wird aufgesogen und nichts kommt mehr zurück. Das ist ein anderes Prinzip als bei der additiven Farbmischung, bei der wir Lichtquellen *kombinieren*, um neue Farben zu erhalten.

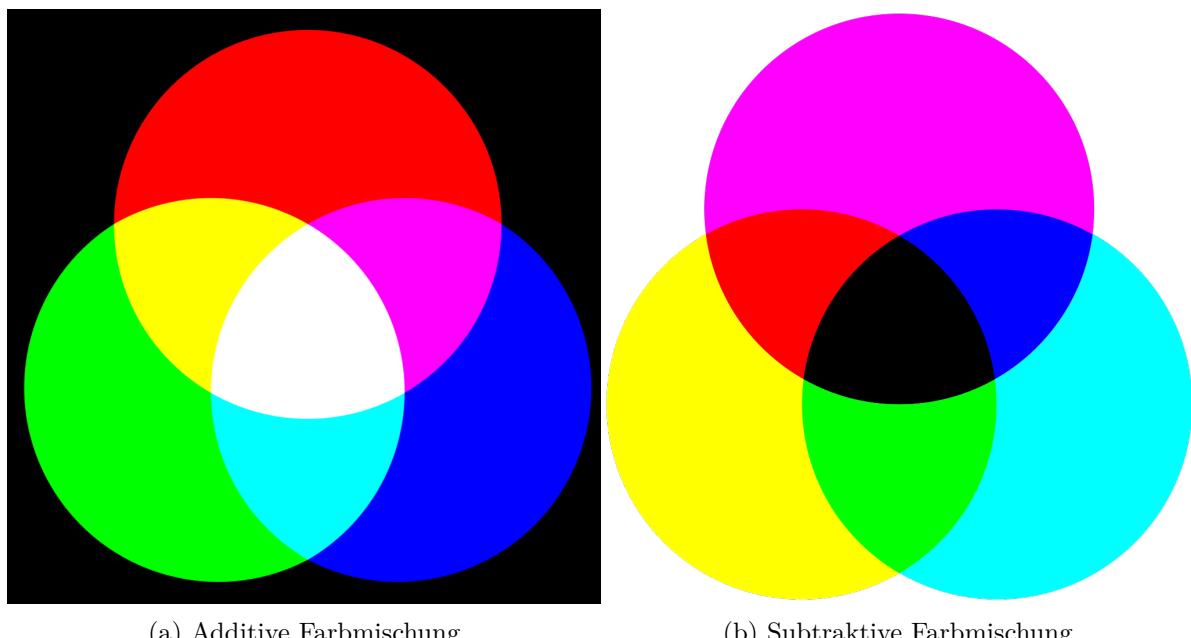


Abbildung 1.6: Additive und subtraktive Farbmischung.

In Abbildung 1.6 sehen wir die beiden Farbmischungsarten im Vergleich. In Abbildung 1.6b sehen wir die drei Grundfarben, die wir bei der subtraktiven Variante benötigen, um daraus alle weiteren Farben zu erhalten. Das sind Cyan, Magenta und Gelb. Im Englischen ist die Abkürzung CMY, wo das “Y” für *Yellow* steht. In der additiven Farbmischung sind es, wie oben schon gesehen, Rot, Grün und Blau. Wenn ihr Abbildung 1.6a betrachtet, dann erkennt ihr, dass genau diese drei Farben durch das Mischen jeweils zweier Grundfarben in der additiven Farbmischung entstehen. Und umgekehrt gilt das gleiche Prinzip! Ob das Zufall ist?

In der additiven Farbmischung entsteht Gelb durch das Mischen von Rot und Grün, wobei Blau fehlt. Im Umkehrschluss bedeutet das: Gelbes Licht enthält keine blaue Komponente, es reflektiert also kein Blau. In der subtraktiven Farbmischung (wie beim Farbkasten) wird Gelb erzeugt, indem Blau aus weißem Licht herausgefiltert wird – Gelb reflektiert also kein Blau, sondern absorbiert es. Gelb kann also auch als Blaufilter gesehen werden. Das erklärt, warum ein gelber Gegenstand unter blauem Licht dunkel erscheint: Er kann das blaue Licht nicht reflektieren.

Jetzt können wir auch erklären, warum Farbdrucker vier unterschiedliche Kartuschen benötigen (Abbildung 1.7). Mit den Grundfarben der subtraktiven Farbmischung Cyan, Magenta und Gelb können wir jede beliebige Farbe mischen. Zusätzlich haben Drucker eine Kartusche für Schwarz, um erstens ein sattes Schwarz drucken zu können und zweitens die Farbkartuschen zu schonen. Denn schließlich müssen alle drei Farben gemischt werden, um Schwarz zu erhalten. Und weil viele Drucksachen Schwarz enthalten (oder sogar ausschließlich), ist eine schwarze Kartusche einfacher effizienter. Die Farbe Schwarz wird bei Druckerkartuschen als Key bezeichnet und mit "K" abgekürzt. Wir sprechen daher auch von CMYK.

Farben spielen eine so wichtige Rolle bei der Arbeit mit Computern. Deshalb lohnt es sich, ein wenig über die Hintergründe von Farben und deren Mischung zu verstehen. Wir werden später noch lernen, wie Bildschirme Farben darstellen. Spätestens dann wird uns das Thema der Farbmischung wieder begegnen.

Ab jetzt wollen wir weiter mit unserer LED experimentieren und den RGB-Code, mit dem Computer Farben abbilden, praxisnah verstehen. Bisher haben wir gelernt, dass wir die Farbe der LED über die Methode `set_rgb_value()` verändern können, wenn wir wissen, welcher RGB-Code unsere gewünschte Farbe repräsentiert. Da wir jetzt mehr über die Farbmischung wissen, können wir die LED also ganz einfach in der Farbe Magenta leuchten lassen:

```
led.set_rgb_value(255, 0, 255)
```

Gemäß der Theorie der additiven Farbmischung (Abbildung 1.6a) müssten Rot und Blau Magenta ergeben. Probiert es aus!

1.4 Pulsierende LED

Das deklarierte Ziel unseres ersten Experiments ist es, einen Regenbogenfarbverlauf zu erzeugen. Dazu müssen wir die Farbe der LED kontinuierlich ändern, sodass sie von Rot über Gelb, Grün, Cyan, Blau und Violett wieder zurück zu Rot wechselt.

Lasst uns aber möglichst einfach anfangen und uns dem Regenbogen schrittweise annähern. Zunächst wäre es schön, wenn wir die LED einfach Rot pulsieren lassen könnten. Dazu müssen wir nämlich nur den Rot-Kanal und nicht alle drei Kanäle der LED ansteuern. Gleichzeitig



Abbildung 1.7: Ein typisches Set mit CMY-Druckerkartuschen inklusive Schwarz.

lernen wir schon hier ein Problem kennen, das uns in der Programmierung häufig begegnet und für das es eine elegante Lösung gibt.

1.4.1 Abzählbare Wiederholungen

Was bedeutet es, die LED pulsieren zu lassen? Und was müssen wir dafür tun? Pulsieren bedeutet, dass die LED über einen kurzen Zeitraum immer heller wird, kurz in der vollen Helligkeit verweilt und dann sofort wieder kontinuierlich dunkler wird. Sobald sie schwarz ist, fängt der Zyklus von vorne an.

Den Ausdruck *immer heller werden* können wir bezogen auf die LED so übersetzen, dass wir den Anteil des Rot-Kanals schrittweise erhöhen. Wenn die LED zu Beginn aus ist, also alle Kanäle auf 0 stehen, können wir den Rot-Kanal von 0 auf 255 erhöhen und so die LED immer heller in Rot aufleuchten lassen.

Wir beginnen also mit einer schwarzen LED:

```
led.set_rgb_value(0, 0, 0)
```

Anschließend setzen wir den Wert für Rot auf 1:

```
led.set_rgb_value(1, 0, 0)
```

Und erhöhen ihn schrittweise:

```
led.set_rgb_value(2, 0, 0)
led.set_rgb_value(3, 0, 0)
led.set_rgb_value(4, 0, 0)
# ...
```

Wenn wir nach diesem Muster fortfahren, hätten wir bis zum vollen Rot 255 Zeilen Code geschrieben, eine Zeile für jeden Erhöhungsschritt. Und anschließend das Gleiche nochmal rückwärts, damit wir wieder zu Schwarz kommen. Mit 510 Zeilen Code hätten wir dann einen Pulsierungszyklus durchlaufen. Wollen wir die LED öfter pulsieren lassen, vervielfacht sich unser Code entsprechend. Das kann nicht die Lösung für ein so einfaches Problem sein.

Und tatsächlich gibt es in der Programmierung eine bessere Möglichkeit, um sich wiederholende Abläufe abzubilden: die Schleife. In einem Fall, bei dem wir genau wissen, wie oft wir etwas wiederholen wollen, bietet sich eine Zählerschleife an:

```
for r in range(256):
    led.set_rgb_value(r, 0, 0)
```

Voilà! Unsere 510 Zeilen Code können wir mit einer Schleife auf zwei Zeilen reduzieren! Dazu müssen wir im Kopf der Schleife (`for ... in ...`) festlegen, wie oft der eingerückte Codeblock nach dem Doppelpunkt ausgeführt werden soll. In Python funktioniert das über die Angabe einer Folge, für die jedes Element einmal durchlaufen wird. Das aktuelle Element ist in der Schleife als `r` verfügbar. Und `r` nimmt nacheinander jeden Wert der Folge an, die nach dem Schlüsselwort `in` folgt. Diese Folge erzeugt hier die Funktion `range(256)`, die – wie der Name preisgibt – eine Zahlenfolge von 0 bis zum angegebenen Wert minus eins erzeugt. In unserem Fall also von 0 bis 255.

Um das besser nachvollziehen zu können, geben wir den Wert für `r` einfach mal aus:

```
for r in range(256):
    led.set_rgb_value(r, 0, 0)
    print(r) ①
```

- ① Mit `print()` geben wir einen Wert auf der Konsole aus.

Jetzt wird es deutlich: Mit jedem Durchlauf der Schleife wird ein neuer Wert für `r` gesetzt und ausgegeben. Und zwar jeweils um eins erhöht. Die Funktion `range(256)` erzeugt genau gesagt eine sortierte Reihe mit den Zahlen von 0 bis 255. Das sieht in Python dann so aus:

```
list_of_numbers = range(256)
print(list(list_of_numbers)) ①
```

- ① Mit der `list()`-Funktion wandeln wir die von `range()` erzeugte Folge in eine Liste um, die wir dann ausgeben können.

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]
```

Rückwärts erreichen wir das gleiche Ergebnis mit einer weiteren Schleife, deren Folge wir umkehren, sodass sie von 255 bis 0 geht:

```
for r in range(255, -1, -1):
    led.set_rgb_value(r, 0, 0)
```

Warum hat `range()` auf einmal drei Argumente? Ganz einfach: Standardmäßig erstellt die Funktion eine Folge von 0 bis zur angegebenen Zahl minus eins. Wir können die Folge aber beeinflussen, indem wir einen Startwert und einen Schrittwert angeben. In unserem Fall oben beginnen wir bei 255 (erster Parameter) und gehen bis -1 (zweiter Parameter), wobei wir in jedem Schritt um -1 verringern (dritter Parameter). Warum zählen wir bis -1, wo wir doch eigentlich die 0 als kleinste Zahl benötigen? Das liegt daran, dass die Folge von `range()` immer

bis zum zweiten Parameter minus eins geht. Wenn wir also 0 als kleinste Zahl benötigen, müssen wir bis -1 zählen.

Fassen wir unsere Erkenntnis zusammen und lassen die LED pulsieren:

```
import time

# Increase red step by step
for r in range(256):
    led.set_rgb_value(r, 0, 0)
    time.sleep(0.001)

# Stay at full brightness for a bit
time.sleep(0.25)

# Decrease red step by step
for r in range(255, -1, -1):
    led.set_rgb_value(r, 0, 0)
    time.sleep(0.001)
```

Soweit bekannt? Fast – eine kleine Neuerung habe ich gerade eingebaut, nämlich die Funktion `time.sleep()`. Diese Funktion pausiert das Programm für die angegebene Zeit in Sekunden. In unserem Fall pausieren wir für 0,001 Sekunden, also 1 Millisekunde. Dadurch wird die LED langsamer heller und dunkler, was den Puls-Effekt verstärkt. Ohne diese Pause würde die LED so schnell aufleuchten, dass es für das menschliche Auge nicht mehr wahrnehmbar wäre. Tatsächlich würden wir auch die Hardware überfordern, weil die LED gar nicht so schnell die Farbe wechseln kann. Das Programm würde abstürzen.

Am Höhepunkt warten wir erneut – dieses Mal eine Viertelsekunde –, bevor wir die LED langsam ausgehen lassen und den Rotanteil schrittweise wieder auf Null setzen. Dann endet unser Programm, leider viel zu früh. Die LED soll doch eigentlich weiter pulsieren, bis ... ja, bis wann überhaupt?

1.4.2 Bedingte Wiederholungen

Beim Lösen von Problemen stoßen wir häufig auf Situationen, in denen wir bestimmte Schritte wiederholt ausführen möchten, aber nur unter bestimmten Bedingungen. Hier kommen bedingte Wiederholungen ins Spiel, die es uns ermöglichen, Schleifen zu erstellen, die nur dann fortgesetzt werden, wenn eine bestimmte Bedingung noch erfüllt ist.

Das können wir auf unsere pulsierende LED anwenden. Sie soll ihren Pulszyklus Dunkel–Hell–Dunkel wiederholen, solange der Benutzer nicht unterbricht. Das ist zumindest ein pragmatisches Abbruchkriterium für unseren Fall. Wir definieren also hier keine feste Anzahl Wie-

derholungen wie bei der `for ... in`-Schleife, sondern wir wollen festlegen, unter welcher *Bedingung* die Schleife fortgesetzt wird. Wir könnten also sagen: *solange* die Bedingung X erfüllt ist, wiederhole die aufgeführten Schritte. Und weil Programmiersprachen für Menschen gemacht sind, klingt es im echten Programm auch so ähnlich:

```
while 1==1:  
    print("This condition is always true")  
    time.sleep(1)
```

Das Schlüsselwort `while` führt eine bedingte Schleife ein, gefolgt von der Bedingung, die die Schleife steuert. Die Bedingung wird vor jedem neuen Schleifendurchlauf geprüft (auch vor dem ersten) und sollte sie falsch (`false`) sein, wird die Schleife beendet.

Wann wird die Schleife oben also beendet? Richtig – niemals. Die Bedingung `1==1` ist immer wahr, die Schleife läuft somit endlos. Wir sprechen auch von einer Endlosschleife, die wir in der Programmierung unbedingt vermeiden wollen, es sei denn, sie ist explizit gewollt und nicht versehentlich entstanden. Das kurze Programm oben schreibt also in Abständen von einer Sekunde den Text “This condition is always true” auf die Konsole.

Eine Bedingung ist in Python und anderen Programmiersprachen ein wichtiges Konzept, das es uns ermöglicht, Entscheidungen zu treffen und den Programmfluss zu steuern. In unserem Fall könnte die Bedingung lauten: “Solange der Benutzer nicht stoppt, wiederhole den Pulsierzyklus”. Eine Bedingung hat die Eigenschaft, dass sie jederzeit ausgewertet werden kann und entweder den Wert wahr (`true`) oder falsch (`false`) annimmt. Wie aber drücken wir das in Python aus?

```
while True:  
    print("I will loop forever")  
    time.sleep(1)
```

Die einfachste Möglichkeit ist es, das Ergebnis der Evaluation direkt hinzuschreiben. Die obige Schleife prüft in jedem Durchgang, ob der Wert `True` wahr ist - was er natürlich ist. Das ist also so ähnlich wie bei der Schleife weiter oben, die die Bedingung `1==1` geprüft hat. Die ist ebenfalls immer `True` oder wahr.

Wir lernen im Laufe des Buches noch viele echte Bedingungen kennen, deren Ergebnis nicht von Vornherein bekannt ist. Für unsere pulsierende LED reicht es aber aus, wenn wir eine gewollte Endlosschleife verwenden. Denn auch eine Endlosschleife können wir jederzeit verlassen, indem wir das Programm mit der Tastenkombination Strg+C abbrechen.

Wenn wir jetzt unseren Pulsierzyklus von oben in die neue bedingte `while`-Schleife einfügen, sind wir schon am Ziel. Der Pulsierzyklus wird wiederholt, solange das Programm nicht abgebrochen wird:

```

while True:

    # Increase red step by step
    for r in range(256):
        led.set_rgb_value(r, 0, 0)
        time.sleep(0.001)

    # Stay at full brightness for a bit
    time.sleep(0.25)

    # Decrease red step by step
    for r in range(255, -1, -1):
        led.set_rgb_value(r, 0, 0)
        time.sleep(0.001)

    # Stay at full dark for a bit
    time.sleep(0.25)

```

Der Vollständigkeit halber das Ganze inklusive des [Boilerplate-Codes](#) für die Verbindung zu den Tinkerforge-Geräten:

 Tipp

Das Codebeispiel aus diesem Abschnitt findet ihr auf [GitHub](#).

1.5 Farbkreise

Das RGB-Farbschema ist für Computer optimal, weil sich damit mit nur drei Zahlen jede beliebige Farbe kodieren lässt. Zahlen sind schließlich die Sprache, mit der Computer am besten umgehen können. Für uns Menschen ist dieses Schema jedoch weniger intuitiv. Oder könntet ihr auf Anhieb sagen, welche Farbe hinter der Kombination (67, 201, 182) steckt?

Um Farben für uns leichter wählbar zu machen, wurde der sogenannte Hue-Farbkreis entwickelt. Er ordnet die Farben auf einer Skala von 0 bis 360 Grad an – ähnlich wie die Winkel auf einem Kreis. Neben dem Farbton (Hue) lassen sich zusätzlich die Sättigung und die Helligkeit einstellen: Der Farbton bestimmt die eigentliche Farbe, die Sättigung, wie kräftig oder blass sie wirkt, und die Helligkeit, wie hell oder dunkel sie erscheint.

In Abbildung 1.8 seht ihr, wie die Farbauswahl in Google Slides funktioniert. Mit dem Slider in der Mitte bestimmt ihr den Farbton. Habt ihr einen passenden Ton gefunden, könnt ihr im

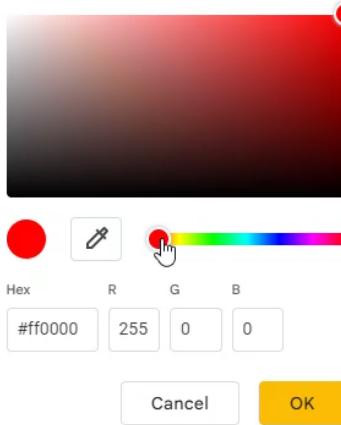


Abbildung 1.8: Die Farbauswahl in Google Slides funktioniert über den Hue-Farbkreis.

Rechteck darüber durch Verschieben des kleinen Kreises die Sättigung und Helligkeit anpassen.

Beobachtet ihr dabei die RGB-Werte, erkennt ihr die Systematik der Farbton-Skala: Ausgehend von reinem Rot wird Schritt für Schritt Grün hinzugefügt – so entstehen Orange und Gelb. Danach nimmt der Rotanteil ab, während Blau hinzukommt. Über Cyan gelangen wir zu reinem Blau. Schließlich wird wieder Rot beigemischt, wodurch Violett bis Pink entstehen. Auf diese Weise bildet der Kreis den gesamten Regenbogen ab.

Da die Skala am Ende wieder bei Rot ankommt, lässt sich der Farbverlauf nahtlos wiederholen – ohne harte Übergänge. Genau deshalb wird der Hue-Verlauf meist als Kreis dargestellt.

Abbildung 1.9 zeigt den Hue-Farbkreis im HSV-Modell. HSV steht für Hue, Saturation, Value (Farbton, Sättigung, Helligkeit). Der Wert Value gibt die Helligkeit auf einer Skala von 0 bis 100 % an. Im Bild ist die Helligkeit konstant bei 100 %, während die Sättigung von innen nach außen zunimmt. In der Mitte sehen wir deshalb Weiß, während am äußeren Rand die Farben ihre volle Intensität haben.

Wenn wir ein Programm schreiben, das die gesamte Hue-Farbskala durchläuft und die LED jeweils in der passenden Farbe aufleuchten lässt, erhalten wir unser Regenbogenprogramm. Da die LED RGB-Werte benötigt, müssen wir den Verlauf des Hue-Farbkreises in RGB umsetzen. Ein Blick auf die Animation in Abbildung 1.8 hilft: Der Farbverlauf lässt sich in sechs Phasen unterteilen, wie Abbildung 1.10 zeigt:

1. Rot = 255, Blau = 0, Grün steigt linear
2. Rot sinkt linear, Grün = 255, Blau = 0
3. Rot = 0, Grün = 255, Blau steigt linear
4. Rot = 0, Grün sinkt linear, Blau = 255
5. Rot steigt linear, Grün = 0, Blau = 255

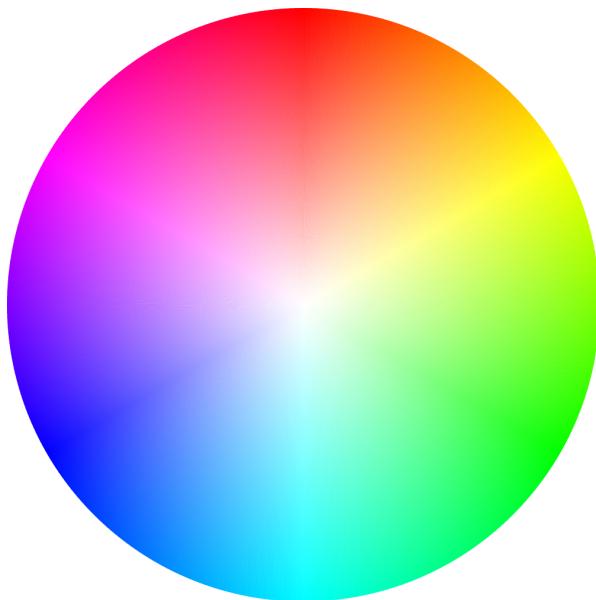


Abbildung 1.9: Der Hue-Farbkreis mit HSV-Werten.

6. Rot = 255, Grün = 0, Blau sinkt linear

Dann beginnt der Zyklus von vorn.

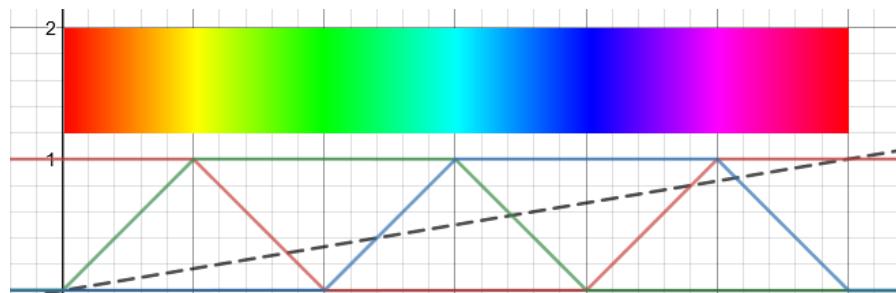


Abbildung 1.10: Der Hue-Farbverlauf mit den Veränderungen der RGB-Werte (Quelle: [Ronja's Tutorials](#)).

1.6 Regenbogen-LED

Mit dem Wissen können wir uns an das erklärte Ziel unseres Experiments machen: die LED nacheinander in allen Farben des Regenbogens aufleuchten zu lassen. Beginnen wir mit der ersten Phase und schreiben dafür ein Python-Programm:

```

for green in range(256):
    led.set_rgb_value(255, green, 0)
    time.sleep(0.01)

```

Weil wir genau wissen, wie oft wir die Schleife durchlaufen wollen, verwenden wir eine **for**-Schleife. Innerhalb der Schleife erhöhen wir die Variable `green` jeweils um 1, was effektiv den Grünanteil unseres RGB-Wertes erhöht. Mit jedem Durchlauf fügen wir somit mehr Grün hinzu, während Rot und Blau konstant bleiben. Phase 1 ist damit abgeschlossen – machen wir weiter mit Phase 2:

```

for red in range(255, -1, -1):
    led.set_rgb_value(red, 255, 0)
    time.sleep(0.01)

```

Wie wir mit einer **for**-Schleife rückwärts zählen, haben wir schon weiter oben kennengelernt. In Phase 2 verringern wir schrittweise den Rotanteil, während die anderen beiden Farben konstant bleiben. Damit kommen wir zu Phase 3:

```

for blue in range(256):
    led.set_rgb_value(0, 255, blue)
    time.sleep(0.01)

```

Ich glaube, ihr habt das Prinzip verstanden. Indem wir die sechs Phasen jeweils in einer eigenen Schleife abarbeiten, erhalten wir das vollständige Regenbogenprogramm:

```

# phase 1
for green in range(256):
    led.set_rgb_value(255, green, 0)
    time.sleep(0.01)

# phase 2
for red in range(255, -1, -1):
    led.set_rgb_value(red, 255, 0)
    time.sleep(0.01)

# phase 3
for blue in range(256):
    led.set_rgb_value(0, 255, blue)
    time.sleep(0.01)

# phase 4

```

```

for green in range(255, -1, -1):
    led.set_rgb_value(0, green, 255)
    time.sleep(0.01)

# phase 5
for red in range(256):
    led.set_rgb_value(red, 0, 255)
    time.sleep(0.01)

# phase 6
for blue in range(255, -1, -1):
    led.set_rgb_value(255, 0, blue)
    time.sleep(0.01)

```

Wie schön Eine Sache fehlt aber noch.

1.6.1 Runde für Runde

Der Regenbogen soll am Ende wieder von vorne beginnen. Wie schon beim Pulsieren der Farben können wir auch hier eine `while`-Schleife verwenden und die Phasen kontinuierlich abspielen – solange, bis der Benutzer die Escape-Taste drückt:

```

while True:

    # phase 1
    for green in range(256):
        led.set_rgb_value(255, green, 0)
        time.sleep(0.01)

    # phase 2
    for red in range(255, -1, -1):
        led.set_rgb_value(red, 255, 0)
        time.sleep(0.01)

    # phase 3
    for blue in range(256):
        led.set_rgb_value(0, 255, blue)
        time.sleep(0.01)

    # phase 4
    for green in range(255, -1, -1):

```

```

    led.set_rgb_value(0, green, 255)
    time.sleep(0.01)

    # phase 5
    for red in range(256):
        led.set_rgb_value(red, 0, 255)
        time.sleep(0.01)

    # phase 6
    for blue in range(255, -1, -1):
        led.set_rgb_value(255, 0, blue)
        time.sleep(0.01)

```

Wir haben es fast geschafft! Eine Kleinigkeit wollen wir an unserem Programm noch verbessern.

1.6.2 Geschwindigkeit steuern

Vielleicht habt ihr gemerkt, dass die Geschwindigkeit, mit der unsere LED den gesamten Regenbogen einmal durchläuft, nicht sehr hoch ist. Ich würde das gerne beschleunigen. Die Zeit steuern wir über die `time.sleep()`-Funktion, sodass wir einfach den Wert in jedem Funktionsaufruf verringern könnten. Das wäre aber nicht sehr effizient, weil wir ihn an sechs Stellen anpassen müssen. Wenn wir danach merken, dass es zu schnell ist, müssten wir den Wert erneut überall editieren. Das geht einfacher!

Der Trick liegt darin, den Wert für die Wartedauer als Variable zu definieren und nur an einer Stelle zu ändern.

```

pause_duration = 0.01
while not keyboard.is_pressed('esc'):

    # phase 1
    for green in range(256):
        led.set_rgb_value(255, green, 0)
        time.sleep(pause_duration)

    # etc.

```

Schon besser! Wir gehen aber noch einen Schritt weiter. Statt dieses kleinteiligen Werts für eine Pause zwischen zwei kleinen Farbveränderungen möchte ich die Gesamtdauer für den Durchlauf eines Regenbogens angeben. Der Wert `pause_duration` soll dann auf dieser Basis

errechnet werden. Dazu müssen wir nur die Anzahl der Pausen insgesamt kennen; in jeder der sechs Phasen sind es 256. Macht also:

$$6 \times 256 = 1536$$

Im Programm setzen wir die Pausendauer also auf die Gesamtdauer in Sekunden geteilt durch 1536:

```
rainbow_duration = 5
pause_duration = rainbow_duration / 1536

while not keyboard.is_pressed('esc'):

    # phase 1
    for green in range(256):
        led.set_rgb_value(255, green, 0)
        time.sleep(pause_duration)

    # etc.
```

Und schon können wir unseren Regenbogen beliebig zeitlich steuern. Damit sind wir am Ende des Kapitels angekommen. Wir schließen es mit dem vollständigen Code für unseren Regenbogenverlauf in Listing 1.3 ab. Vergesst nicht, den Wert für die UID eurer LED anzupassen, damit es auch bei euch funktioniert:



Tipp

Das Codebeispiel aus diesem Abschnitt findet ihr auf [GitHub](#).

Seid ihr bereit für das nächste Experiment?

Listing 1.2 Das fertige Programm, das die LED rot pulsieren lässt.

```
import time
from tinkerforge.ip_connection import IPConnection
from tinkerforge.bricklet_rgb_led_v2 import BrickletRGBLEDV2

ipcon = IPConnection()
ipcon.connect('localhost', 4223)
led = BrickletRGBLEDV2('ZEP', ipcon)

# Turn LED off initially
led.set_rgb_value(0, 0, 0)

while True:

    # Increase red step by step
    for r in range(256):
        led.set_rgb_value(r, 0, 0)
        time.sleep(0.001)

    # Stay at full brightness for a bit
    time.sleep(0.25)

    # Decrease red step by step
    for r in range(255, -1, -1):
        led.set_rgb_value(r, 0, 0)
        time.sleep(0.001)

    # Stay at full dark for a bit
    time.sleep(0.25)
```

Listing 1.3 Das fertige Regenbogenprogramm.

```
import time
from tinkerforge.ip_connection import IPConnection
from tinkerforge.bricklet_rgb_led_v2 import BrickletRGBLEDV2

ipcon = IPConnection()
ipcon.connect("localhost", 4223)
led = BrickletRGBLEDV2("<YOUR_LED_UID>", ipcon)

# Turn LED off initially
led.set_rgb_value(0, 0, 0)

rainbow_duration = 5
pause_duration = rainbow_duration / 1536

while True:

    # phase 1
    for green in range(256):
        led.set_rgb_value(255, green, 0)
        time.sleep(pause_duration)

    # phase 2
    for red in range(255, -1, -1):
        led.set_rgb_value(red, 255, 0)
        time.sleep(pause_duration)

    # phase 3
    for blue in range(256):
        led.set_rgb_value(0, 255, blue)
        time.sleep(pause_duration)

    # phase 4
    for green in range(255, -1, -1):
        led.set_rgb_value(0, green, 255)
        time.sleep(pause_duration)

    # phase 5
    for red in range(256):
        led.set_rgb_value(red, 0, 255)
        time.sleep(pause_duration)

    # phase 6
    for blue in range(255, -1, -1):
        led.set_rgb_value(255, 0, blue)
        time.sleep(pause_duration)    43
```

2 Zahlen

Zusammenfassung

Im zweiten Kapitel spendieren wir der LED aus Kapitel 2 eine Dimmfunktion, die wir über einen Drehknopf steuern können.

Auf dem Weg dorthin gehen wir die folgenden Schritte.

#	Was?	Wo?
1	Wir machen uns mit dem Drehknopf vertraut.	Abschnitt 2.1
2	Wir lesen den Zählerstand des Drehknopfs aus einem Programm heraus aus.	Abschnitt 2.2
3	Wir führen Kontrollstrukturen ein (<code>if</code>).	Abschnitt 2.3
4	Wir erstellen eine erste Version eines Dimmers für die LED.	Abschnitt 2.4
5	Wir beschäftigen uns mit Zahlensystemen.	Abschnitt 2.5
6	Wir lernen, was ein Bit und ein Byte ist.	Abschnitt 2.6
7	Wir erweitern den LED-Dimmer zur Version 2.	Abschnitt 2.7
8	Wir setzen uns mit den Druckknopf auseinander.	Abschnitt 2.8
9	Wir bauen Version 3 des LED-Dimmers.	Abschnitt 2.9
10	Wir modularisieren unseren Code mit Funktionen.	Abschnitt 2.10

2.1 Experimentaufbau

2.1.1 Hardware

Das erste Experiment in Kapitel 1 war ein guter Einstieg! In diesem Kapitel legen wir noch eine Schippe drauf: Unsere Hardware bekommt ein neues Bauteil – einen Drehknopf. Das montiert

ihr einfach neben der LED, wie in Abbildung 2.1 gezeigt.

Die vollständige Hardwareliste für dieses Kapitel sieht so aus:

- 1 x Master Brick 3.2
- 1 x RGB LED Bricklet 2.0
- 1 x Rotary Encoder Bricklet 2.0
- 1 x Montageplatte 22x10
- 2 x Brickletkabel 15cm (7p-7p)
- 2 x Befestigungskit 12 mm

2.1.2 Erste Schritte mit dem Drehknopf

Wie bei der LED werfen wir zuerst einen Blick auf den neuen Drehknopf im Brick Viewer. Schließe dazu euren Master Brick per USB an, startet den Brick Viewer und klickt auf Connect. Im Setup-Tab sollte nun neben der LED auch der Rotary Encoder auftauchen. Denkt daran: Dort findet ihr auch die UID eurer Geräte – die braucht ihr gleich für euer Programm.

Wechselt nun in den Tab für den Drehknopf, wo ihr ihn direkt testen könnt: Ihr seht den aktuellen Zählerwert. Der kann positiv oder negativ sein – je nachdem, wie oft und in welche Richtung ihr gedreht habt. Daneben zeigt ein Diagramm die zeitliche Entwicklung.

Doch der Knopf kann mehr als nur zählen: Ihr könnt ihn auch drücken. Achtet mal auf den kleinen Kreis im Brick Viewer. Wird er gedrückt, leuchtet er rot. Noch löst das Drücken keine Aktion aus, aber wir überlegen später, welche Funktion wir damit verbinden wollen.

Und zuletzt: der Button Reset Count. Damit setzt ihr den Zähler zurück – eine praktische Funktion, die wir später ebenfalls ins Programm einbauen können.

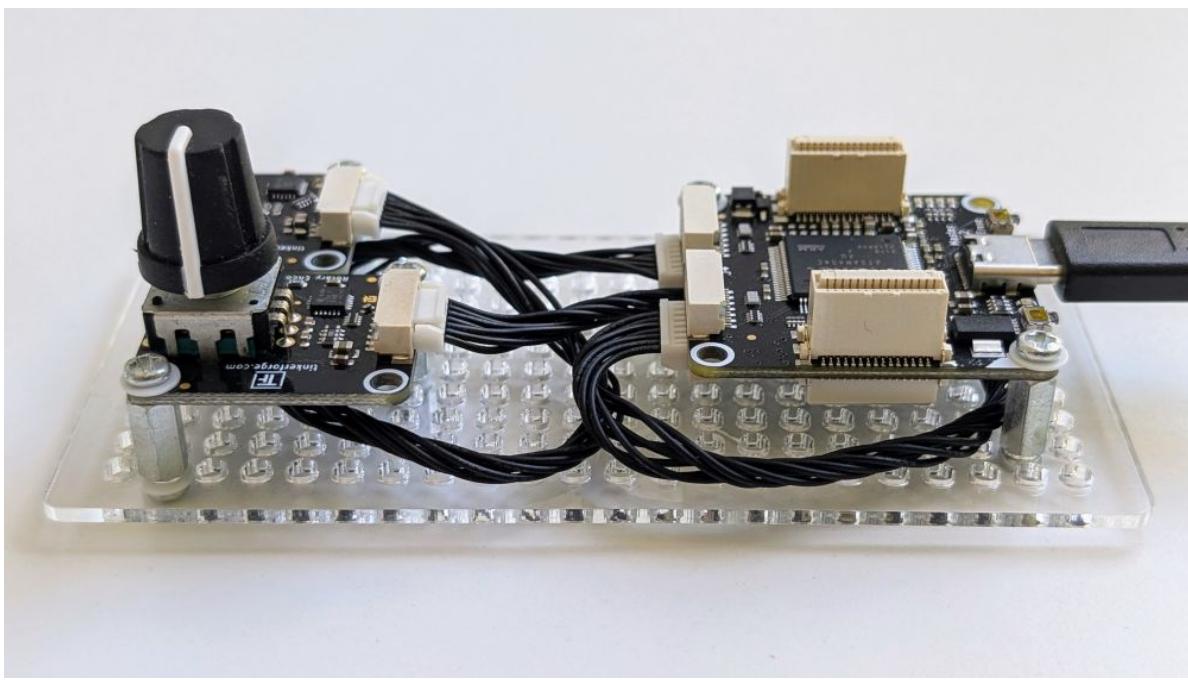
Fassen wir zusammen, was unser Drehknopf (Rotary Encoder) draufhat:

1. Er zählt – vorwärts und rückwärts
2. Er merkt, wenn ihr ihn drückt
3. Er kann seinen Zähler zurücksetzen

Zeit also, das Ganze in Python auszuprobieren und zu sehen, welche spannenden Anwendungen wir damit bauen können.

2.2 Zähler auslesen

Der Drehknopf funktioniert ähnlich wie der Lautstärkeregler einer Stereoanlage (siehe Abbildung 2.4): Dreht ihr nach rechts, wird es lauter – nach links, leiser.

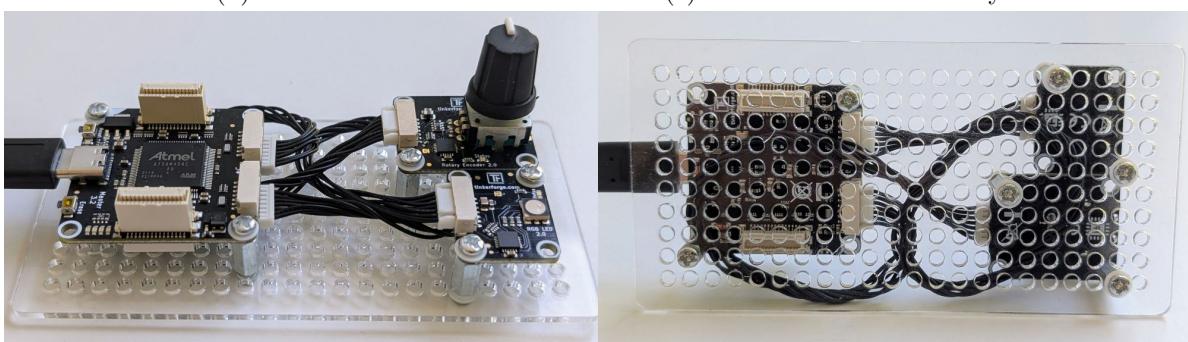


(a) Seitenansicht.



(b) Draufsicht.

(c) Nahaufnahme des Rotary Encoders.



(d) Seitenansicht.

(e) Untenansicht.

Abbildung 2.1: Einfaches Setup mit einem Mikrocontroller, LED und einem Drehknopf.

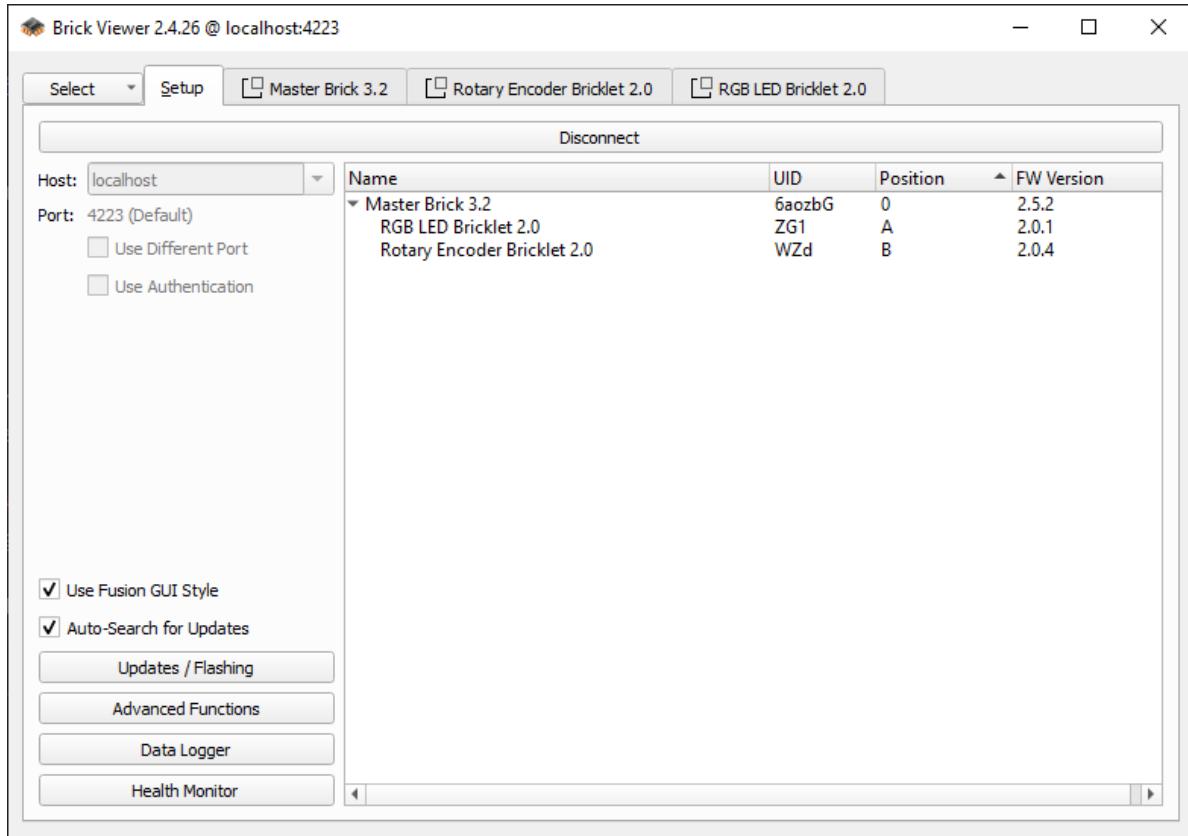
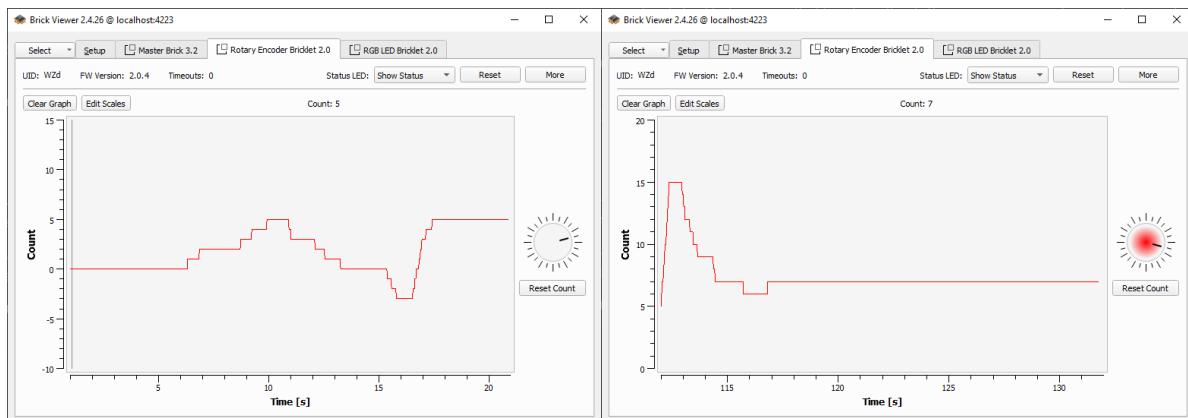


Abbildung 2.2: Der Brick Viewer nach dem Connect.



(a) Das Diagramm zeigt den aktuellen Wert an.

(b) Der Button wird rot, wenn er gedrückt ist.

Abbildung 2.3: Die Funktionen des Rotary Encoders im Brick Viewer.



Abbildung 2.4: Die gute alte Stereoanlage mit Drehknopf! Wer kennt sie nicht mehr? (Quelle: [Wikimedia](#))

Im Hintergrund verändert sich bei jeder Drehung der Wert, den der Knopf sendet – mal höher, mal niedriger, je nach Ausgangszustand. Im Brick Viewer habt ihr das schon gesehen. Probieren wir es jetzt in einem Programm aus: Der folgende Code verbindet sich mit dem Drehknopf, liest den aktuellen Wert und gibt ihn in der Konsole aus. Denkt daran, eure eigene UID einzutragen:

```
from tinkerforge.ip_connection import IPConnection
from tinkerforge.bricklet_rotary_encoder_v2 import BrickletRotaryEncoderV2

ipcon = IPConnection()
ipcon.connect('localhost', 4223)
knob = BrickletRotaryEncoderV2('WZd', ipcon)          ①
count = knob.get_count(reset=False)                   ②
print(count)
```

- ① Wir erstellen eine virtuelle Repräsentation des Drehknopfs in unserem Programm, um seine Funktionen verwenden zu können.
- ② Über die virtuelle Repräsentation des Drehknopfs können wir mittels `get_count()` den aktuellen Wert abfragen. Der Parameter `reset` bestimmt, ob der Zähler nach dem Auslesen zurückgesetzt werden soll oder nicht.

Die Ausgabe sollte mit dem Wert übereinstimmen, den ihr auch im Brick Viewer seht – kein Wunder, beide nutzen dieselbe Programmierschnittstelle. Damit habt ihr die erste Funktion des Drehknopfs erfolgreich aus Python getestet.

Dreht ihr den Knopf und startet das Programm erneut, erscheint natürlich ein anderer Wert. Klar! Aber jedes Mal neu starten? Das geht besser. Die Lösung kennt ihr schon aus Kapitel 1: eine Schleife, die das Programm so lange wiederholt, bis wir es beenden:

```
while True:
    count = knob.get_count(reset=False)
    print(count)
```

Zur Erinnerung: `while True` erzeugt eine Endlosschleife. Normalerweise wollen wir so etwas vermeiden – außer, wir brauchen es genau dafür. Endlosschleifen sind praktisch, wenn wir kontinuierlich Daten lesen oder auf Ereignisse warten. Und keine Sorge: Mit Strg+C könnt ihr das Programm jederzeit beenden, wenn die Konsole aktiv ist. Klickt also einmal in die Konsole und drückt Strg+C, dann hat der Spuk ein Ende.

Wenn ihr das Programm ausführt, werdet ihr direkt ein Problem erkennen. Die Schleife rennt förmlich und gibt nacheinander immer wieder denselben Wert aus. Nur wenn wir am Knopf drehen, ändert sich der Wert – wird aber von der Schleife x-mal auf die Konsole geschrieben. Wie könnten wir das verbessern?

2.3 Kontrollstrukturen

Wie wäre es hiermit?

```
last_count = None
while True:
    new_count = knob.get_count(reset=False)

    if new_count != last_count:
        last_count = new_count
        print(last_count)
```

Gehen wir durch, was hier passiert: Zuerst weisen wir der Variable `last_count` vor dem ersten Schleifendurchlauf den Wert `None` zu. Anschließend wird in jedem Durchlauf der aktuelle Zählerstand ausgelesen und in der Variable `new_count` gespeichert. Danach prüfen wir, ob sich der neue Wert im Vergleich zum alten unterscheidet. Da `last_count` im ersten Durchlauf `None` ist, wird die Bedingung in Zeile 5 beim Start immer `True` sein. Somit geben wir den Wert zu Beginn auf jeden Fall aus – genau so, wie es für die Anwendung sinnvoll ist.

In den folgenden Schleifendurchläufen wird nur dann etwas ausgegeben, wenn sich der Wert verändert hat, ihr also tatsächlich am Drehknopf gedreht habt. Ansonsten bleibt die Ausgabe unverändert.

Die Prüfung, ob der aktuelle Wert (gespeichert in `new_count`) sich vom alten Wert unterscheidet, erfolgt in Zeile 5. Hier lernen wir auch ein neues Konzept der Programmierung kennen: die Kontrollstruktur, eingeleitet mit dem Schlüsselwort `if`, gefolgt von einer Bedingung. Eine Bedingung, kann – wie ihr schon aus Kapitel 1 wisst – nur `True` oder `False` sein. Ist sie wahr (`True`), läuft der eingerückte Code darunter. Ist sie falsch (`False`), passiert nichts.

Übertragen auf unser Programm heißt das: `print(last_count)` läuft nur dann, wenn sich der Wert tatsächlich verändert hat. In diesem Fall merken wir uns den neuen Wert und aktualisieren `last_count`. Beim nächsten Schleifendurchlauf prüfen wir wieder, ob sich etwas getan hat. Meistens ist das nicht so – und genau deshalb sehen wir nur dann eine Ausgabe, wenn wir wirklich am Knopf drehen. Ziemlich effizient, oder?

2.4 LED-Dimmer 1.0

Wenden wir das Gelernte an und bauen einen praktischen LED-Dimmer. Dafür holen wir unsere LED aus Kapitel 1 mit dazu und kombinieren sie mit dem Drehknopf.

Die Idee ist simpel: Der Zähler des Knopfs steuert die Helligkeit der LED. Dreht ihr nach rechts, wird sie heller, nach links, dunkler. Wie bei der Stereoanlage in Abbildung 2.4.

Bevor wir uns an die eigentliche Anwendungslogik machen, brauchen wir Zugriff auf beide Geräte – LED und Drehknopf. Dazu erweitern wir den bekannten Boilerplate-Code und speichern die Geräte in eigenen Variablen:

```
from tinkerforge.ip_connection import IPConnection
from tinkerforge.bricklet_rotary_encoder_v2 import BrickletRotaryEncoderV2
from tinkerforge.bricklet_rgb_led_v2 import BrickletRGBLEDV2

ipcon = IPConnection()
ipcon.connect("localhost", 4223)
knob = BrickletRotaryEncoderV2("<YOUR_ROTARY_UID>", ipcon)
led = BrickletRGBLEDV2("<YOUR_LED_UID>", ipcon)
```

Dieser Teil muss immer am Anfang unseres Programms stehen, damit alles funktioniert. In den kommenden Beispielen setzen wir ihn als gegeben voraus und wiederholen ihn nicht jedes Mal.

Als Startpunkt nehmen wir den Code von oben, der den Zählerwert auf der Konsole ausgibt. Schließlich brauchen wir genau diese Information – wann sich der Wert ändert und wie er aktuell steht – auch, um die LED zu steuern.

Damit wir die LED von aus bis volle Helligkeit dimmen können, legen wir uns zuerst auf eine Farbe fest. Ich bin zwar kein Fan von weißem LED-Licht, aber für dieses Kapitel ist es am einfachsten: voll aufgedreht leuchtet die LED weiß, ausgedreht ist sie schwarz – klar! Später kümmern wir uns darum, wie wir das Licht wärmer machen können.

Erinnern wir uns also: Was bedeuten die Zustände An und Aus im RGB-Farbraum?

```
# White
led.set_rgb_value(255, 255, 255)

# Black (off)
led.set_rgb_value(0, 0, 0)
```

Damit haben wir die beiden Extremzustände festgelegt. Doch was passiert dazwischen, wenn die LED gedimmt ist? Ganz einfach: Wir lassen die drei RGB-Werte gemeinsam von 1 bis 255 hoch- oder runterlaufen. Höhere Werte ergeben ein helleres Weiß, niedrigere ein dunkleres.

Setzen wir diese Erkenntnisse in Programmcode um und weisen den Zählerwert den RGB-Werten der LED zu. Spoiler-Alert: Das ist etwas naiv, aber lässt uns mal schauen, was passiert und wo möglicherweise Probleme auftreten:

```

knob.reset()
last_count = 0

while True:
    new_count = knob.get_count(reset=False)

    if new_count != last_count:
        last_count = new_count
        print(last_count)

    # Setze RGB-Werte auf den Zählerwert
    led.set_rgb_value(last_count, last_count, last_count)

```

Lasst es mal laufen und dreht voll auf oder runter! Beobachtet dabei den Wert für `last_count`. Was passiert, wenn er kleiner als Null wird? Oder wenn er größer als 255 wird? Bumm! Das Programm stürzt ab!

Warum? Auf der Kommandozeile bekommen wir eine lange Fehlermeldung mit der folgenden Aussage ganz am Ende:

```
struct.error: ubyte format requires 0 <= number <= 255
```

Wenn man die Fehlermeldung googelt oder ChatGPT fragt, bekommt man Hilfe. Offensichtlich wird für einen RGB-Wert, den wir der Funktion `set_rgb_value()` übergeben, ein bestimmter Datentyp erwartet, der `ubyte` heißt. Das steht für “unsigned byte” und bedeutet, dass der Wert zwischen 0 und 255 liegen muss.

Moment – was hat jetzt das Byte mit 0 bis 255 zu tun? Bisher dachten wir doch, das wäre wegen des RGB-Codes? Stimmt auch, aber der RGB-Code liegt nicht zufällig im Wertebereich von 0 bis 255.

Um das zu verstehen, müssen wir das Binärsystem kennen. Also los!

2.5 Zahlensysteme

Eigentlich ist es schnell erklärt. Das Binärsystem ist wie das Dezimalsystem, mit dem wir alltäglich unterwegs sind – nur nutzt es statt der Basis 10 die Basis 2. Einfach, oder? Wenn nicht, lest weiter – das hier soll schließlich ein Einführungsbuch sein.

2.5.1 Unser Dezimalsystem

Wir wenden das Dezimalsystem täglich intuitiv an. Es fragt sich wahrscheinlich niemand von euch, was die Systematik dahinter ist, oder? Und doch habt ihr es alle einmal in der Schule gelernt, und wir müssen es an dieser Stelle etwas auffrischen. Solltet ihr mit Stellenwertsystemen noch 100 % vertraut sein, könnt ihr diesen Abschnitt getrost überspringen.

Nehmen wir eine Zahl wie die 123 als Beispiel. Wir haben sofort ein Gefühl für die Zahl, wir wissen etwa, wie groß sie ist. Und wenn wir es etwas genauer erklären müssen, können die meisten von euch sicher erläutern, wofür – also für welchen Wert – jede Ziffer steht. Wir beginnen mit der kleinsten Wertigkeit, also der Ziffer ganz rechts: der 3. Sie steht für die Einserstelle, und davon haben wir 3. Die nächste Stelle steht für die Zehner, und weil dort eine 2 steht, sind es zwanzig. Also $3 + 20 = 23$. Schließen wir auch die dritte und letzte Ziffer in unsere Erläuterung ein: Die 1 steht für die Hunderterstelle, also $1 * 100 = 100$. Damit haben wir $100 + 20 + 3 = 123$. Ganz einfach und intuitiv.

1	2	3
10^2	10^1	10^0
100	10	1

1	2	3
10^2	10^1	10^0
$= 1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0$ $= 1 \times 100 + 2 \times 10 + 3 \times 1$ $= 123$		

- (a) Im Dezimalsystem hat jede Stelle einen anderen Wert.
(b) Durch Ausmultiplizieren errechnen wir den Wert der Zahl.

Abbildung 2.5: Das Dezimalsystem ist ein Stellenwertsystem.

Das Ganze funktioniert nicht nur mit dreistelligen Zahlen, sondern prinzipiell mit beliebig langen Zahlen. Wir wissen, dass die nächste Ziffer, die wir links im Beispiel in Abbildung 2.6 sehen, für die Tausenderstelle steht. Die nächste Stelle würde für die Zehntausenderstelle stehen – und so weiter. Warum fällt es uns so leicht?

Erstens, weil wir damit jeden Tag umgehen. Das Dezimalsystem ist das System, das wir am häufigsten verwenden, und wir haben es von klein auf gelernt. Es ist intuitiv und einfach zu verstehen. Zweitens aber auch, weil wir die Systematik kennen: Jede Stelle ist 10-mal so viel wert wie die vorherige.

Wurde uns das Dezimalsystem von Gott gegeben? Vielleicht – wenn man an die Schöpfung glaubt¹ und daran, dass Gott uns so geschaffen hat, wie wir sind, dann hat er implizit dafür gesorgt, dass wir dezimal denkende Wesen werden. Warum? Eine Theorie besagt, dass die

¹Hände hoch, wer daran noch glaubt!

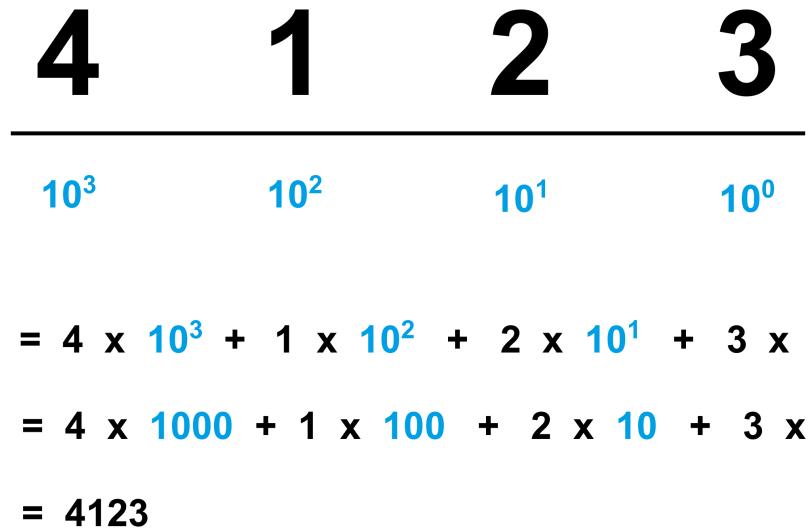


Abbildung 2.6: Jede Stelle steht für eine höhere Potenz der Basis 10.

menschliche Anatomie, insbesondere die Anzahl der Finger, einen Einfluss auf unser Zahlen- system hatte. Zählt einfach mal anhand eurer Finger durch.

2.5.2 Das Oktalsystem

Nun gibt es auch Wesen mit weniger als zehn Fingern (und auch mit mehr?). Nehmt mal einen Cartoon-Charakter wie Mickey Mouse als Beispiel. In Abbildung 2.7 seht ihr, wie hier wahrscheinlich gezählt wird. Hätte ein Volk von Mickey-Mäusen sich auch für das Dezimalsystem entschieden?

Vermutlich nicht! Das Oktalsystem ist wie das Dezimalsystem ein Stellenwertsystem, nur mit einer anderen Basis. Das bedeutet, dass jede Stelle statt eine Zehnerpotenz eine Achterpotenz darstellt. Die verfügbaren Symbole oder Ziffern sind 0 bis 7. Warum? Weil es nur acht Finger gibt, und die mit zehn Fingern ist der achte Finger die Zahl 10. Im Oktalsystem stellt das die Dezimalzahl 8 dar, weil die zweite Ziffer von rechts für die Achterstelle steht.

Die 123, die ihr in Abbildung 2.8 seht, können wir - wie jede Zahl in einem beliebigen Stellen- wertsystem - mit dem einfachen Ausmultiplizieren in das Dezimalsystem umrechnen. Die drei ganz rechts steht für $3 \cdot 1$. Weil jede Zahl hoch Null eine Eins ergibt, steht die erste Stelle steht in jedem System für die Eins. Die zweite Stelle steht für 8^2 , also für die Achter (also $2 \cdot 8 = 16$)

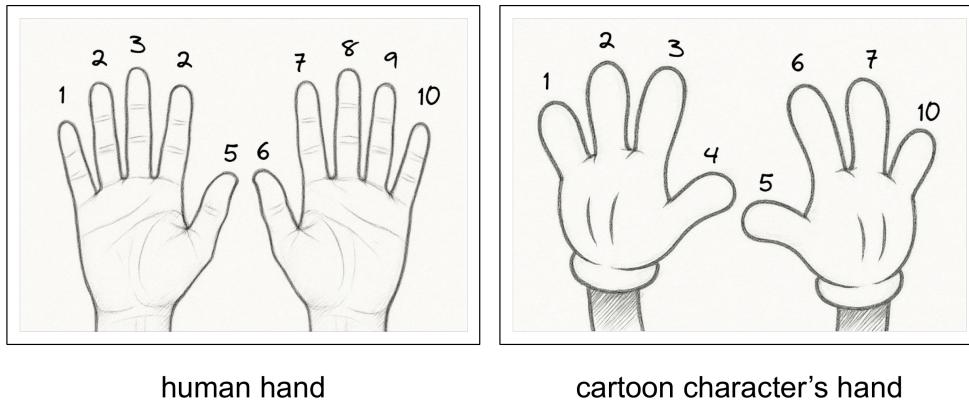


Abbildung 2.7: Cartoon-Charaktere haben nur acht Finger. Quelle: Erstellt mit ChatGPT nach Petzold (2022)

und die dritte Stelle 8^3 , was Vierunsechzig ergibt. Somit geht sie mit $1 \cdot 64$ in die Berechnung des Dezimalwertes ein. Zusammen ergibt das $64 + 16 + 3 = 83$ im Dezimalsystem.

2.5.3 Das Binärsystem

Treiben wir es noch ein wenig weiter auf die Spitze und nehmen ein paar Finger weg – sagen wir bis auf zwei. Dann wären wir vielleicht bei einem Delfin mit zwei Flossen, wie ihr ihn in Abbildung 2.9 seht. Delfine haben sich vermutlich auf ein System geeinigt, das auch für unsere heutigen Computer die Grundlage darstellt: das Binärsystem.

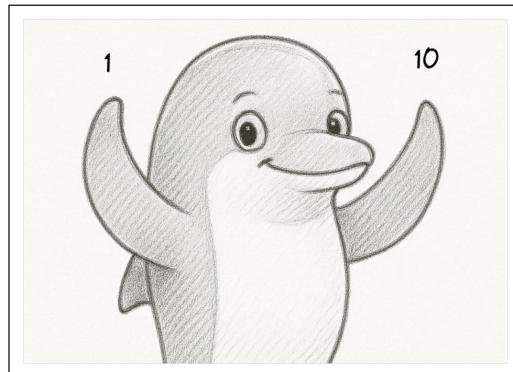
Das Wort “binär” stammt aus dem Lateinischen und bedeutet “paarweise” oder “zu zweit”. Von diesem Wort stammt auch der Name des Stellenwertsystems mit der Basis 2 – und das nicht ohne Grund. Im Binärsystem gibt es für jede Stelle genau zwei Möglichkeiten: 0 oder 1. Ein anderer Begriff ist übrigens Dualsystem, was genau das Gleiche meint. Auch das Wort “dual” kommt von den Römern und heißt so viel wie “zwei enthaltend”.

Zwei Möglichkeiten, das erinnert an einen Lichtschalter, der entweder an oder aus sein kann. In Abbildung 2.10 ist das bildlich dargestellt. Entweder leuchtet die Lampe (1) oder sie ist aus (0). Mehr geht nicht. Das Binärsystem ist also wirklich simpel.

$$\begin{array}{r}
 1 \quad 2 \quad 3 \\
 \hline
 8^2 \quad 8^1 \quad 8^0
 \end{array}$$

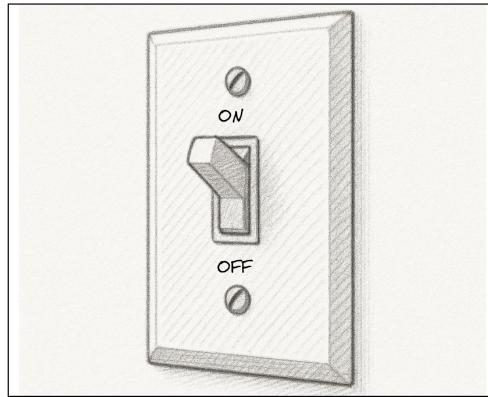
$$\begin{aligned}
 &= 1 \times 8^2 + 2 \times 8^1 + 3 \times 8^0 \\
 &= 1 \times 64 + 2 \times 8 + 3 \times 1 \\
 &= 83 \text{ (decimal)}
 \end{aligned}$$

Abbildung 2.8: Das Oktalsystem funktioniert wie das Dezimalsystem. Nur die Basis ist 8 statt 10.



what now?

Abbildung 2.9: Delfine würden anders zählen. Eher wie Computer.



a binary number is like a switch

Abbildung 2.10: Eine Binärziffer ist vergleichbar mit einem Lichtschalter, der an oder aus sein kann.

Und auch das Rechnen im Binärsystem ist nichts Besonderes. Es funktioniert wie im Dezimalsystem oder Oktalsystem auch, wir tauschen einfach die Basis aus. Jede Stelle stellt nun eine Potenz von zwei dar. Statt vieler Symbole gibt es nur die 0 und die 1, um Zahlen darzustellen. Das macht es im Binärsystem sogar noch einfacher, als in Systemen mit einer höheren Basis. Denn letztlich müssen wir nur die 2er-Potenzen, an denen eine 1 steht, addieren. Alles, wo eine Null steht, können wir ignorieren.

$$\begin{array}{r}
 1 \quad 1 \quad 0 \\
 \hline
 2^2 \quad 2^1 \quad 2^0
 \end{array}
 \quad (\text{binary})$$

$$\begin{array}{r}
 1 \quad 1 \quad 0 \\
 \hline
 2^2 \quad 2^1 \quad 2^0
 \end{array}
 \quad (\text{binary})$$

$$\begin{aligned}
 &= 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 \\
 &= 1 \times 4 + 1 \times 2 + 0 \times 1 \\
 &= 6 \text{ (decimal)}
 \end{aligned}$$

(a)

(b)

Abbildung 2.11: Das Binärsystem funktioniert wie alle anderen Stellenwertsysteme auch.

place value systems

$$N = d_n * R^{n-1} + \dots + d_2 * R^1 + d_1 * R^0$$

$$d \in \{0, 1, \dots, R-1\}$$

$$\begin{aligned}
 n &= \text{number of digits} \\
 R &= \text{base}
 \end{aligned}$$

2.5.4 Andere Systeme

Stellenwertsysteme sind nicht die einzigen Zahlensysteme. Es zum Beispiel das römische Zahlensystem, bei dem die Position der Ziffer keine Rolle spielt. Stattdessen werden verschiedene Symbole verwendet, um Zahlen darzustellen. Ein anderes Beispiel ist die Tally-Schreibweise, bei der Striche und Strichpakete verwendet werden, um Zahlen zu darzustellen.

2.6 Bits & Bytes

2.6.1 Zwei Zustände

Warum haben wir uns Zahlensysteme angeschaut, und was hat das mit Computern zu tun? Ganz einfach: Computer denken binär. Das bedeutet, sie kennen nur zwei Zustände: an oder aus, 0 oder 1.

Wir kommen später noch einmal ausführlich darauf zurück, aber so viel schon vorweg: Eine Binärziffer nennen wir im Englischen “binary digit”, kurz “bit”. Jetzt klingelt es, oder?

Ein Bit ist eine Informationseinheit. Nicht irgendeine, sondern die kleinste, die es gibt. Die Erklärung, warum das so ist, folgt später. Wir wollen uns an dieser Stelle die Frage stellen, was wir mit einem Bit alles anstellen können.

Ein Bit ist alleine ziemlich einsam und eingeschränkt. Wenn sich ein Computer mit einem Bit lediglich merken kann, ob eine Lampe an oder aus ist, dann sind das genau zwei Möglichkeiten. Nicht besonders viel. Wir kamen aber von den Farben über Zahlensysteme zu den Bits – und unsere ursprüngliche Frage war, wie ein Computer mit seinen Mitteln – also Nullen und Einsen (oder eben Bits) – so viele unterschiedliche Farben abbilden und speichern kann. Zwei würden gerade einmal für Schwarz/Weiß ausreichen.

Ihr ahnt es vielleicht schon: Wir gesellen zum ersten ein zweites Bit hinzu. Und schon können wir vier unterschiedliche Zustände abbilden: 00, 01, 10 und 11. Damit könnten wir zum Beispiel die Farben Schwarz, Blau, Grün und Cyan darstellen. Etwas willkürlich (warum gerade diese Farben), aber denkbar.

Was passiert, wenn wir ein drittes Bit hinzunehmen? Sind es nun sechs Zustände? Nein, es sind acht: 000, 001, 010, 011, 100, 101, 110 und 111. Damit könnten wir die Farben Schwarz, Blau, Grün, Cyan, Rot, Magenta, Gelb und Weiß darstellen (oder jede andere Kombination, die wir uns wünschen).

Mit jedem zusätzlichen Bit können wir also nicht plus zwei mehr Zustände abbilden, sondern wir verdoppeln unsere Möglichkeiten. Also müssen wir mal zwei – und nicht plus zwei – rechnen. Das ist eine gute Nachricht, denn die Anzahl der Farben, die wir mit jedem zusätzlichen Bit darstellen können, verdoppelt sich jedes Mal.

Das halten wir fest, aber schauen wir zurück auf unsere RGB-Farben und die Fehlermeldung, die wir zuletzt bekommen haben. Der Wert für eine Farbe aus dem RGB-Farbcde muss zwischen 0 und 255 liegen. Wir haben somit inklusive der Null 256 Möglichkeiten für jede der drei RGB-Grundfarben. Wie viele Bits benötigen wir dafür? Rechnen wir es aus:

$$\begin{aligned}
 2^0 &= 1 \\
 2^1 &= 2 \\
 2^2 &= 4 \\
 2^3 &= 8 \\
 2^4 &= 16 \\
 2^5 &= 32 \\
 2^6 &= 64 \\
 2^7 &= 128 \\
 2^8 &= 256
 \end{aligned}$$

Stopp! $2^8 = 256$, das genügt uns völlig. Mit 8 Bits können wir somit 256 Zustände abbilden – genau passend für 256 Rot-, Grün- oder Blauanteile.

2.6.2 Acht Bits macht ein Byte

Und das ist kein Zufall: 8 Bits sind für Computer eine besondere Größe. Wir nennen eine Gruppe von 8 Bits ein Byte. Und jetzt dürfte es erneut klingeln.

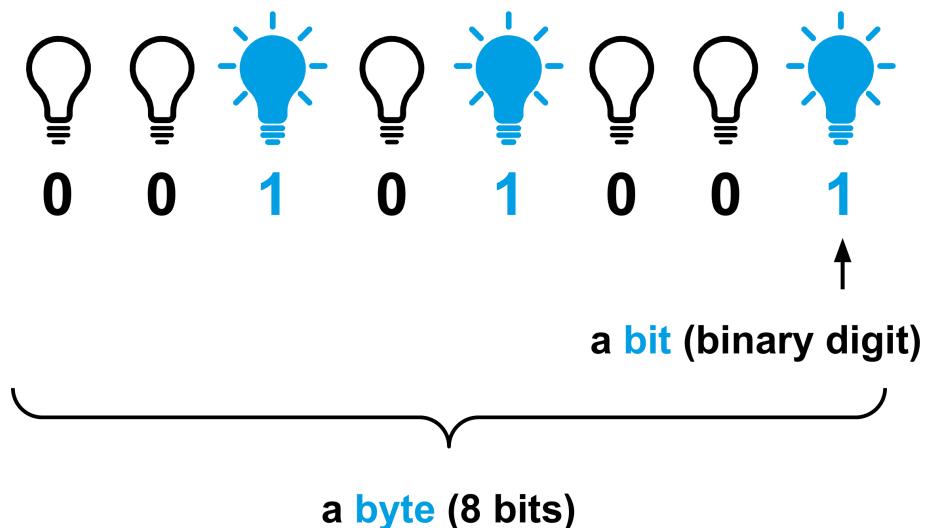


Abbildung 2.12: Ein Byte könnt ihr euch vorstellen wie acht Glühbirnen nebeneinander.

In Abbildung 2.12 ist ein Byte als Reihe von acht Glühbirnen dargestellt. Ihr könnt euch vorstellen, dass Bits mit dem Wert 1 leuchten und Bits mit dem Wert 0 aus sind. Um den Wert zu ermitteln, den das Byte gerade darstellt, könnt ihr jeder Glühbirne von rechts nach links die entsprechenden Wertigkeiten der Stellen aus dem Binärsystem zuweisen und die Werte addieren. Stellen, an denen die Glühbirne leuchtet, werden addiert, die anderen werden ausgelassen (Abbildung 2.13). Das Byte im gezeigten Beispiel steht somit für:

$$32 + 8 + 1 = 41$$

Wofür steht das Byte, wenn alle Lampen leuchten? Oder anders gefragt: Was ist die größte Zahl, die wir mit einem Byte darstellen können?

$$128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255$$

Die Antwort überrascht uns nicht, denn schließlich haben wir es ja schon herausgefunden: Ein Byte erlaubt uns, Werte zwischen 0 (alle Glühbirnen aus) und 255 (alle Glühbirnen an) darzustellen. Insgesamt also 256 Möglichkeiten. Somit können wir mit 8 Glühbirnen die Intensität einer der drei Grundfarben im RGB-Code darstellen.

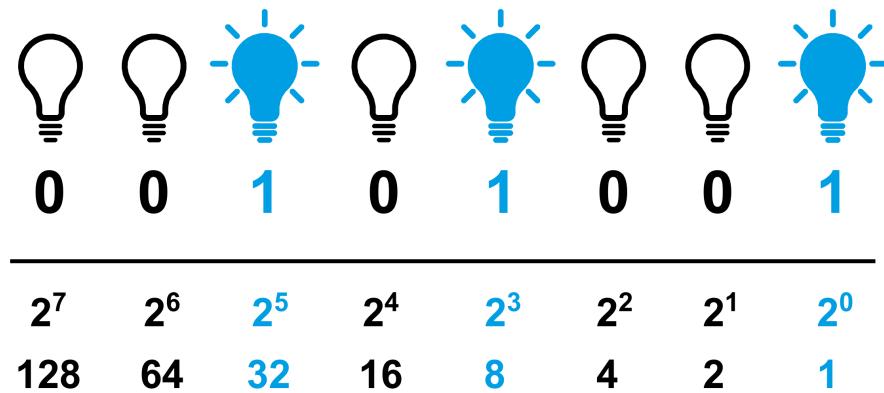


Abbildung 2.13: Jede Glühbirne steht für eine Stelle aus dem Binärsystem.

Das erklärt auch die Fehlermeldung von oben: Ein Byte kann Werte zwischen 0 und 255 darstellen. Wir haben im Experiment den Drehknopf voll nach oben oder nach unten gedreht,

wodurch der Wert entweder größer als 255 oder kleiner als 0 wurde. Und damit ist es kein gültiger Wert im Sinne eines Bytes mehr.

2.6.3 Kilo, Mega, Giga

Ein Byte besteht aus 8 Bits. Wenn wir also von Bytes sprechen, reden wir oft auch von Kilobytes, Megabytes, Gigabytes et cetera. Diese Begriffe sind wichtig, um die Größe von Daten zu beschreiben. In Tabelle 2.2 seht ihr eine Übersicht über die verschiedenen Größenordnungen.

Tabelle 2.2: Verschiedene Mengeneinheiten für Bytes und deren ungefähre Entsprechung.

Potenz (Bytes)	Ausgeschrieben	Bezeichnung (Abkürzung)	Entspricht ca.
10^3	Tausend	Kilobyte (KB)	kleine Textdatei
10^6	Million	Megabyte (MB)	Digitales Foto
10^9	Milliarde	Gigabyte (GB)	Film (DVD 4,7 GB)
10^{12}	Billion	Terabyte (TB)	Gängige Festplattenkapazität
10^{15}	Billiarde	Petabyte (PB)	Speichervolumen Rechenzentrum
10^{18}	Trillion	Exabyte (EB)	Internetverkehr pro Tag
10^{21}	Trilliarde	Zettabyte (ZB)	Datenbestand weltweit (>100 ZB)
10^{24}	Quadrillion	Yottabyte (YB)	keine Entsprechung

Jetzt, da ihr wisst, was mit einem Byte gemeint ist, könnt ihr eine ungefähre Vorstellung für die Größenordnungen von Datenmengen entwickeln. Die ersten drei Zeilen aus Tabelle 2.2 könnt ihr selbst einmal nachvollziehen. Schaut euch dazu mal eine Textdatei an, notiert deren Größe und rechnet aus, wie viele Glühbirnen für die Speicherung gebraucht werden. Denkt daran: Ein Byte entspricht acht Glühbirnen.

Wir kommen in den späteren Kapiteln immer wieder auf die Bits und Bytes zurück, weil wir in Computern letztlich überall mit diesen Einheiten arbeiten. Es ist somit gut, wenn ihr schon an dieser Stelle ein grundlegendes Verständnis für diese Konzepte entwickelt.

2.7 LED-Dimmer 2.0

Zurück zu unserem eigentlichen Vorhaben. Wir waren gerade dabei, einen Dimmer für unsere LED zu basteln, als uns die Zahlensysteme dazwischen gekommen sind. Dafür haben wir

jetzt ein besseres Verständnis dafür, wie ein Computer Farben sieht – nämlich als lange Sequenz aus Nullen und Einsen. Und zwar 24 davon, weil jede Grundfarbe ein Byte an Speicher verwendet.

2.7.1 `min()` und `max()`

Was müssen wir also in unserem Programm verändern, jetzt, da wir wissen, was zuvor das Problem war? Genau! Wir müssen sicherstellen, dass die Werte, die wir an die LED senden, im gültigen Bereich für ein Byte liegen – und zwar zwischen 0 und 255.

```
knob.reset()
last_count = 0

while True:
    new_count = knob.get_count(reset=False)

    if new_count != last_count:
        last_count = new_count

        # Clamp last_count to valid byte range
        last_count = max(0, min(255, last_count)) ①

        print(last_count)

        # Setze RGB-Werte auf den Zählerwert
        led.set_rgb_value(last_count, last_count, last_count)
```

- ① Die Funktionen `min()` und `max()` sorgen dafür, dass der Wert von `last_count` immer zwischen 0 und 255 bleibt. Wenn `last_count` kleiner als 0 ist, wird er auf 0 gesetzt. Wenn er größer als 255 ist, wird er auf 255 gesetzt.

Die neue Logik in Zeile 11 hilft uns dabei. Nachdem wir den neuen Wert des Zählers in der Variable `last_count` gespeichert haben (Zeile 8), wenden wir eine geschickte Kombination der beiden Funktionen `max()` und `min()` an, um sicherzustellen, dass der Wert im gültigen Bereich bleibt. Wie funktioniert das genau? Dazu gehen wir die Zeile Schritt für Schritt durch.

Zunächst einmal der Ausdruck `min(255, last_count)`. Die Funktion `min()` gibt einfach den kleineren der beiden Werte zurück, die ihr übergeben werden. Wenn `last_count` also größer als 255 ist, wird 255 zurückgegeben. Andernfalls wird `last_count` zurückgegeben. Das Ergebnis dieser Auswertung ist gleichzeitig der zweite Wert, den wir der Funktion `max()` übergeben.

Die Funktion `max()` macht genau das Gegenteil. Sie gibt den größeren der beiden ihr übergebenen Werte zurück. Zur Auswahl stehen ihr der Wert 0 und das Ergebnis der `min()`-Funktion.

Das bedeutet, dass `max()` sicherstellt, dass der endgültige Wert von `last_count` niemals kleiner als 0 ist.

Und voilà! Nach Zeile 11 kann der Wert von `last_count` nur noch zwischen 0 und 255 liegen. Problem gelöst!

Probiert es am besten direkt aus und dreht mal voll auf! Es sollte nun kein Fehler mehr auftreten.

2.7.2 Helligkeit entkoppeln

Vielleicht habt ihr es auch bemerkt, aber so richtig toll funktioniert unser Dimmer immer noch nicht. Zwar erscheint keine Fehlermeldung mehr, wenn wir endlos aufdrehen. Jedoch wird die LED auch nicht gedimmt, wenn wir wieder in die andere Richtung drehen. Der Grund dafür ist einfach: Die Helligkeit der LED hängt in unserem Programm direkt vom Zählerstand des Drehknopfes ab. Wenn der über 255 kommt, wird die Helligkeit zwar auf 255 gedeckelt, der Zähler wird aber im Hintergrund trotzdem weiter hochgezählt. Wenn wir die LED wieder dimmen, also einen Helligkeitswert von weniger als 255 erreichen möchten, dann müssen wir zunächst mit dem Drehknopf wieder bis unter die 255 kommen.

Viel schöner wäre es, wenn wir zwar endlos überdrehen könnten, aber mit der ersten Drehung in die andere Richtung die Helligkeit der LED sofort verringern. Ein einfacher Weg wäre, für den Zählerstand des Drehknopfes analog zu `last_count` nur Werte zwischen 0 und 255 zu erlauben. Dazu könnten wir den Zähler – genau wie `last_count` – manuell auf 0 oder 255 setzen, je nachdem, ob wir größer als 255 oder kleiner als 0 waren. Leider bietet der Drehknopf über seine Programmierschnittstelle keine solche Funktion an. Wir können den Wert zwar auslesen, aber nicht programmatisch verändern.

Wir müssen also einen Workaround entwickeln. Eine Möglichkeit wäre, die Helligkeit unabhängig vom Zählerstand zu verwalten und dafür eine eigene Variable `brightness` einzuführen. Wir könnten den Wert von `brightness` dann erhöhen oder verringern, wenn wir eine Drehung in die eine oder andere Richtung erkannt haben.

Um zu erkennen, ob und in welche Richtung der Drehknopf gedreht wurde, können wir die Differenz zwischen dem aktuellen und dem letzten Zählerstand betrachten. Sie gibt uns direkt Aufschluss: Ist die Differenz positiv, wurde der Knopf nach oben gedreht, ist sie negativ, wurde er nach unten gedreht.

```
knob.reset()
last_count = 0

brightness = 0
led.set_rgb_value(brightness, brightness, brightness) (1)
```

```

while True:
    new_count = knob.get_count(reset=False)

    if new_count != last_count:
        diff = new_count - last_count
        last_count = new_count

        # Adjust brightness
        brightness += diff
        brightness = max(0, min(255, brightness)) ②

    # Setze RGB-Werte auf den Zählerwert
    led.set_rgb_value(brightness, brightness, brightness)

    print(f"Brightness / Counter: {brightness} / {new_count}") ④

```

- ① Die neue Variable `brightness` zu Beginn mit 0 initialisieren. Die LED soll aus sein.
- ② Hier ermitteln wir die Differenz zwischen dem aktuellen und dem letzten Zählerstand und speichern sie in der Variable `diff`.
- ③ Wir passen die Helligkeit an, indem wir `brightness` um `diff` erhöhen oder verringern. Dabei stellen wir sicher, dass der Wert zwischen 0 und 255 bleibt.
- ④ Zur Überprüfung geben wir beide Variablen aus. Wenn wir den Wertebereich 0–255 verlassen, gehen die Werte der beiden Variablen auseinander.

2.7.3 Konstanten

Das sieht schon sehr gut aus! Unser Dimmer ist fast fertig, die grundlegende Funktionalität läuft robust. Eine Kleinigkeit stört mich noch: Der Dimmer reagiert nur sehr langsam, und wir müssen scheinbar endlos drehen, um die LED auf die volle Helligkeit zu bekommen. Können wir das beschleunigen?

Das ist natürlich eine rhetorische Frage – in der Programmierung können wir so gut wie alles umsetzen. Und in diesem Fall ist es sogar recht einfach. Damit die LED schneller hell oder dunkel wird, wenn wir am Drehknopf drehen, können wir die Anpassung der Helligkeit einfach verstärken. Momentan wird die Variable `brightness` um die Differenz des Zählerstands erhöht oder verringert. Wir könnten stattdessen einen festen, höheren Schrittwert definieren, um die Helligkeit schneller zu ändern.

Dazu definieren wir eine neue Variable, die eine Besonderheit hat. Wir geben ihr den Namen `STEP`, der nur aus Großbuchstaben besteht (Zeile 3 in Listing 2.1). Gemäß der [Regeln für die Bennung von Variablen in Python](#) werden Namen in GROSSBUCHSTABEN üblicherweise für Konstanten verwendet – und tatsächlich ist `STEP` genau genommen auch keine Variable, sondern eine **Konstante**.

Eine Konstante unterscheidet sich dadurch, dass ihr Wert einmal festgelegt wird und sich danach nicht mehr ändert. In unserem Fall wollen wir, dass STEP immer den Wert 10 hat. Konstanten definieren wir typischerweise zu Beginn eines Python-Programms, damit man einen schnellen Überblick über alle definierten Konstanten und ihre Werte bekommen kann.

Es ist wichtig zu verstehen, dass der fixe Wert einer Konstante sich nur auf die Ausführung des Programms bezieht. Zwischen mehreren Ausführungen desselben Programms kann der Wert einer Konstante geändert werden. Zum Beispiel könnten wir als Hersteller des LED-Dimmers für eine neue Version entscheiden, dass dieser sich noch schneller dimmen lassen soll, und wir erhöhen den Wert für STEP auf 20. Oder der Benutzer könnte diesen Wert über die Einstellungen der hypothetischen Dimmer-App anpassen.

Wenn wir – wie in Zeile 15 gezeigt – die Differenz des Zählers mit der Schrittgröße multiplizieren, können wir die Anpassung der Helligkeit verstärken.

Listing 2.1 Der fertige LED-Dimmer (ohne Boilerplate-Code)

```
knob.reset()
brightness = 0
STEP = 10
led.set_rgb_value(brightness, brightness, brightness) ①

last_count = 0
while True:
    new_count = knob.get_count(reset=False)

    if new_count != last_count:
        diff = new_count - last_count
        last_count = new_count

        # Adjust brightness
        brightness += diff * STEP ②
        brightness = max(0, min(255, brightness))

        # Setze RGB-Werte auf den Zählerwert
        led.set_rgb_value(brightness, brightness, brightness)

    print(f"Brightness / Counter: {brightness} / {new_count}")
```

- ① Hier definieren wir eine Konstante STEP und weisen ihr den Wert 10 zu.
- ② Die Helligkeit wird nun um `diff * STEP` angepasst, was bedeutet, dass jede Drehung des Knopfes einen größeren Einfluss auf die Helligkeit hat.

Mit dem LED-Dimmer haben wir die zentrale Funktion des Drehknopfes zur Genüge kennengelernt. Das Gerät hat aber noch eine andere Funktion.

2.8 Druckknopf auslesen

Neben dem Zähler besitzt der Drehknopf (der Name sagt es schon) noch eine Funktion, nämlich die eines einfachen Druckknopfes. Wir haben es weiter oben in Abschnitt 2.1.2 mit dem Brick Viewer schon ausprobiert: Der Drehknopf lässt sich drücken und erzeugt eine haptische Rückmeldung, ein leichtes Knacken. Im Brick Viewer wurde der kleine Kreis auf der rechten Seite dann rot eingefärbt.

Selbstverständlich können wir den Zustand des Buttons auch aus einem Programm heraus abfragen. Dazu bietet uns der Drehknopf eine Methode `is_pressed()` an:

```
while True:
    if knob.is_pressed():
        print("Button pressed")
    else:
        print("Button not pressed")
```

Die Funktion liefert `True` zurück, wenn der Button gerade gedrückt ist, und ansonsten `False`. Das können wir wunderbar nutzen und darüber eine Bedingung formulieren, um entweder “Button pressed” oder “Button not pressed” auf der Konsole auszugeben. Ihr erinnert euch bestimmt an das `if`-Statement aus Abschnitt 2.3. Das ist genau das, was wir jetzt brauchen!

Listing 2.2 Ein erster Test des Drehknopf-Buttons.

```
button_pressed_before = False                                ①
while True:
    button_pressed_after = knob.is_pressed()

    if button_pressed_before == True and button_pressed_after == False: ②
        print("Button was pressed and released")                      ③

    button_pressed_before = button_pressed_after                  ④
```

- ① Wir initialisieren eine Variable `button_pressed_before`, die den vorherigen Zustand des Buttons speichert. Am Anfang gehen wir mal davon aus, dass er nicht gedrückt ist.
- ② Mit dem `if`-Statement überprüfen wir, ob der Button losgelassen wurde. Dazu muss der vorherige Zustand `True` und der aktuelle Zustand `False` sein.

- ③ Wenn der Button soeben losgelassen wurde, geben wir eine entsprechende Information auf der Konsole aus.
- ④ Am Ende der Schleife aktualisieren wir den vorherigen Zustand `button_pressed_before`, damit er den aktuellen Zustand für die nächste Iteration speichert.

Das reicht fürs Erste – der Button kann tatsächlich nicht mehr als das. Reicht aber auch: Damit können wir unserem Dimmer schon einen zusätzlichen Mehrwert verleihen. Schließlich kann unsere LED nicht nur weiß leuchten.

2.9 LED-Dimmer 3.0

Wäre es nicht praktisch, wenn wir das Licht der LED nicht nur dimmen, sondern auch den Farbton verändern könnten? Weißes Licht ist am Abend bekanntlich nicht empfehlenswert, ein wärmerer Farbton verbessert den Schlaf und grünes Licht soll beruhigend wirken.

Lasst uns unseren Dimmer so erweitern, dass per Knopfdruck der Farbton gewechselt werden kann. Fürs Erste wollen wir die Farben Weiß, Gelb und Grün anbieten. Das lässt sich später beliebig erweitern.

2.9.1 Farbe per Variable steuern

Der Ausgangspunkt für unser dimmbares Stimmungslicht ist der Dimmer aus Listing 2.1. Von hier aus fügen wir Schritt für Schritt die Logik für den Farbwechsel per Button ein. Lasst uns aber zunächst ganz ohne Button versuchen, die Farbe der LED zu ändern.

Bisher haben wir es uns einfach gemacht und die LED in Weiß leuchten lassen. Dazu mussten wir nur jeden der drei RGB-Farbkanäle auf den gleichen Wert setzen. Wenn wir neben Weiß auch Gelb und Grün anbieten wollen, müssen wir die Farbkanäle unterschiedlich ansteuern. Für Gelb setzen wir den roten und den grünen Kanal auf den gleichen Wert, während der blaue Kanal auf 0 bleibt. Für Grün setzen wir den grünen Kanal auf den gleichen Wert und die anderen beiden auf 0. Um so eine Logik umzusetzen, haben wir das passende Instrument bereits in unserem Werkzeugkasten: [Kontrollstrukturen](#).

Nehmen wir mal an, wir hätten eine Variable `color`, auf der die aktuelle Farbe gespeichert ist, in der die LED leuchten soll. Sie könnte also die Werte “white”, “yellow” oder “green” annehmen. Dann könnten wir mit `if`-Statements die notwendige Logik umsetzen:

```
if color == "white":
    led.set_rgb_value(brightness, brightness, brightness)
if color == "yellow":
    led.set_rgb_value(brightness, brightness, 0)
```

```
if color == "green":  
    led.set_rgb_value(0, brightness, 0)
```

Erinnert euch, dass der Code nach einem `if` nur dann ausgeführt wird, wenn die vorangegangene Bedingung erfüllt ist. Da die Variable `color` zu einem Zeitpunkt nur einen der drei Werte annehmen kann, muss genau eine der drei Bedingungen erfüllt sein und alle anderen entsprechend nicht.

Wenn wir jetzt zu Beginn unseres Programms `color` auf einen der drei Werte setzen, können wir die Logik schnell mal testen:

```
color = "white"  
  
if color == "white":  
    led.set_rgb_value(brightness, brightness, brightness)  
if color == "yellow":  
    led.set_rgb_value(brightness, brightness, 0)  
if color == "green":  
    led.set_rgb_value(0, brightness, 0)
```

Alles sollte so sein wie zuvor, die LED leuchtet weiß.

```
color = "yellow"
```

Jetzt sollte beim Start des Programms die LED gelb leuchten. Dasselbe probiert mal mit “green” aus, das dürfte auch funktionieren.

2.9.2 Farbe per Knopfdruck ändern

Die aktuelle Farbe in einer Variable zu speichern ist eine gute Idee gewesen. Darauf können wir aufbauen und den Button für den Wechsel der Farbe nutzen. Aber wie?

Zunächst erinnern wir uns an die Logik aus Listing 2.2, in dem wir den Button bereits aus einem Programm heraus getestet haben. Dort haben wir eine Logik gebastelt, die erkennt, wenn der Button gedrückt und wieder losgelassen wird. Wenn das der Fall war, wurde der Wert “Button gedrückt” auf der Konsole ausgegeben. Könnten wir diese Logik nicht verwenden, um statt etwas auszugeben einfach die Farbe zu wechseln?

Natürlich können wir das. Passen wir den Code entsprechend an:

```

button_pressed_before = False
while True:
    button_pressed_after = knob.is_pressed()

    if button_pressed_before == True and button_pressed_after == False:
        if color == "white":
            color = "yellow"
        elif color == "yellow":
            color = "green"
        elif color == "green":
            color = "white"

    button_pressed_before = button_pressed_after

```

Wieder ein Haufen voller `ifs` - wir sprechen auch von verschachtelten `ifs` - aber es sollte funktionieren. Gehen wir es einmal durch: Wenn der Button losgelassen wurde, also das erste `if` in Zeile 5 `True` zurückgibt, gelangen wir zur Prüfung des `if`-Statements auf zweiter Ebene. Hier wird im ersten Fall geprüft, ob die LED gerade Weiß leuchtet (`color == "white"`, Zeile 6). Ist das der Fall, dann wechseln wir jetzt auf Gelb. Im zweiten Schritt sehen wir ein `elif` (Zeile 8), das sehr ähnlich zu einem `if` ist, mit der Einschränkung, dass es nur überhaupt geprüft wird, wenn das vorherige `if` nicht schon wahr war. Das macht in diesem Fall einen großen Unterschied (im Vergleich zu weiter oben, als wir einfache `if`-Statements verwendet haben, um die Farbe der LED mit `set_rgb_color` zu setzen). Überlegt mal, was passieren würde, wenn wir hier folgenden Code einsetzen würden:

```

if color == "white":
    color = "yellow"
if color == "yellow":
    color = "green"
if color == "green":
    color = "white"

```

Geht das mal im Kopf durch. Wenn `color` aktuell den Wert `white` hat, dann wird durch das erste `if` der Wert `yellow` gesetzt. Anschließend wird das zweite `if` geprüft, das jetzt wahr ist, und der Wert wird auf `green` gesetzt. Das dritte `if` wird dann also ebenfalls wahr sein, und der Wert wird wieder auf `white` gesetzt. Im Endeffekt haben wir also nichts gewonnen, die LED bliebe weiß. Hier ist die Verwendung von `elif` entscheidend. Denn ein `elif` wird nur geprüft, wenn das vorherige `if` oder `elif` nicht wahr war. Nach der ersten Anpassung wäre hier also Schluss und die Farbe ist wie gewünscht Gelb.

Fügen wir alles zusammen - die neue Logik zum setzen der Farben basierend auf der Variable `color` und die Logik zum Ändern der Variable, sowie die Logik des Dimmers aus Listing 2.1:

Schaut euch den Code in Ruhe an und prüft, ob ihr ihn Zeile für Zeile nachvollziehen könnt. An dieser Stelle hat unser Programm schon eine beträchtliche Größe angenommen, und so langsam wird es unübersichtlich. Versuchen wir also, Struktur hineinzubringen. Im Wesentlichen besteht das Programm aus drei Teilen, jeden habe ich mit einem vorangestellten Kommentar markiert:

1. Hauptschleife, um das Programm am Laufen zu halten
2. Logik für Farbwechsel bei Tastenfreigabe
3. Logik zur Helligkeitsanpassung

In der Hauptschleife wird am Anfang immer wieder der aktuelle Zählerstand und der Zustand des Buttons abgefragt und auf jeweils einer Variable gespeichert. Diese Werte benötigen wir, um zu entscheiden, ob wir die Farbe ändern oder die Helligkeit anpassen müssen.

Um einen potenziellen Farbwechsel kümmert sich der zweite Block, der mit dem `if button_pressed_before == True ...` beginnt. Die Bedingung prüft, ob der Button gerade aus dem gedrückten Zustand in den nicht gedrückten Zustand wechselt, der Benutzer ihn also gerade losgelassen hat. In diesem Moment soll die Farbe gewechselt werden. Die Logik dafür haben wir gerade entwickelt.

Um die Helligkeitsanpassung kümmert sich dann der dritte und letzte größere Block. Er beginnt mit `if new_count != last_count`, was prüft, ob der Drehknopf betätigt wurde. Wenn ja, dann wird die Helligkeit entsprechend der Differenz angepasst. Diese Logik haben wir in Abschnitt [2.7](#) zusammen entwickelt.

Wer von euch jetzt ganz genau hinsieht, der erkennt, dass die Blöcke 2 und 3 zum Teil identischen Code ausführen. In der Programmierung ist das eine rote Flagge ! Lasst uns darüber sprechen, warum!

2.10 Funktionen

In der Programmierung möchten wir Wiederholungen um jeden Preis vermeiden. Wir sprechen auch vom DRY-Prinzip, was für *Don't Repeat Yourself* steht. Wenn wir feststellen, dass wir denselben Code an mehreren Stellen verwenden, sollten wir darüber nachdenken, etwas zu verändern. Warum? Und was?

Nehmen wir in unserem Beispiel an, wir führen eine vierte Farbe ein, sagen wir Blau. Dann müssten wir den Code in den Blöcken 2 und 3 anpassen, um die neue Farbe zu berücksichtigen. Das bedeutet, dass wir den gleichen Code an mehreren Stellen ändern müssten, was fehleranfällig und mühsam ist. Zwei mag noch nicht nach einem Problem klingen, aber selbst hier zeigt sich das Problem der Wiederholung. Wird eine Stelle vergessen, ist der Code inkonsistent und funktioniert nicht mehr wie gewünscht.

Die Lösung liegt darin, häufig verwendeten Code in Funktionen auszulagern. Funktionen sind ein mächtiges Werkzeug in der Programmierung. Sie ermöglichen es uns, Codeblöcke zu definieren, die wir immer wieder verwenden können, ohne sie jedes Mal neu schreiben zu müssen. Funktionen helfen uns dabei, unseren Code sauberer, übersichtlicher und wartbarer zu gestalten.

Im Listing 2.3 wird dieser Teil an zwei Stellen wiederholt:

```
if color == "white":  
    led.set_rgb_value(brightness, brightness, brightness)  
if color == "yellow":  
    led.set_rgb_value(brightness, brightness, 0)  
if color == "green":  
    led.set_rgb_value(0, brightness, 0)
```

Zeit, diesen Code nur einmal zu schreiben! Machen wir daraus eine Funktion. Wie das geht? Im Prinzip müssen wir vier Dinge klären:

1. Was soll die Funktion tun?
2. Wie sieht das Ergebnis aus?
3. Was benötigt die Funktion, um ihre Aufgabe zu erledigen?
4. Wie heißt die Funktion?

```
def set_led_color(color, brightness):  
    if color == "white":  
        led.set_rgb_value(brightness, brightness, brightness)  
    if color == "yellow":  
        led.set_rgb_value(brightness, brightness, 0)  
    if color == "green":  
        led.set_rgb_value(0, brightness, 0)
```

In Listing 2.4 sieht ihr den fertigen Code für den Dimmer mit Farbwechsel per Knopfdruck.

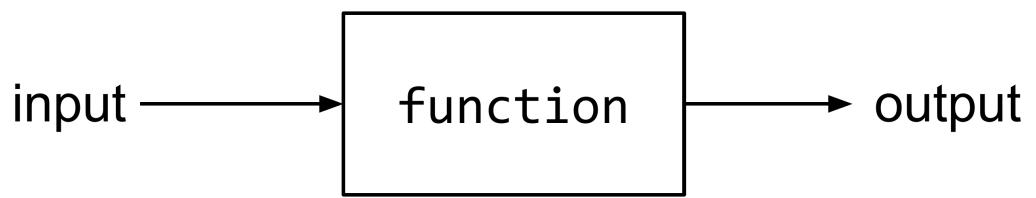


Abbildung 2.14: Funktionen folgen einem Eingabe-Verarbeitung-Ausgabe (EVA) Schema.

Listing 2.3 Farbwechsel und Helligkeitsanpassung in einem Programm.

```
button_pressed_before = False

# 1. Main loop to keep program running
while True:
    new_count = knob.get_count(reset=False)
    button_pressed_after = knob.is_pressed()

    # 2. Logic for color change on button release
    if button_pressed_before == True and button_pressed_after == False:
        if color == "white":
            color = "yellow"
        elif color == "yellow":
            color = "green"
        elif color == "green":
            color = "white"

        # Update LED to reflect new color
        if color == "white":
            led.set_rgb_value(brightness, brightness, brightness)
        if color == "yellow":
            led.set_rgb_value(brightness, brightness, 0)
        if color == "green":
            led.set_rgb_value(0, brightness, 0)

    button_pressed_before = button_pressed_after

    # 3. Logic for brightness adjustment
    if new_count != last_count:
        diff = new_count - last_count
        last_count = new_count

        # Adjust brightness
        brightness += diff * STEP
        brightness = max(0, min(255, brightness))

        # Update LED to reflect new color
        if color == "white":
            led.set_rgb_value(brightness, brightness, brightness)
        if color == "yellow":
            led.set_rgb_value(brightness, brightness, 0)
        if color == "green":
            led.set_rgb_value(0, brightness, 0)

    print(f"Brightness / Counter: {brightness} / {new_count}")
```

Listing 2.4 Der fertige Dimmer mit Farbwechsel per Knopfdruck.

```
from tinkerforge.ip_connection import IPConnection
from tinkerforge.bricklet_rotary_encoder_v2 import BrickletRotaryEncoderV2
from tinkerforge.bricklet_rgb_led_v2 import BrickletRGBLEDV2

ipcon = IPConnection()
ipcon.connect("localhost", 4223)
knob = BrickletRotaryEncoderV2("<YOUR_ROTARY_UID>", ipcon)
led = BrickletRGBLEDV2("<YOUR_LED_UID>", ipcon)

knob.reset()
brightness = 0
STEP = 10
led.set_rgb_value(brightness, brightness, brightness)
last_count = 0

color = "white"
button_pressed_before = False

def set_led_color(color, brightness):
    if color == "white":
        led.set_rgb_value(brightness, brightness, brightness)
    if color == "yellow":
        led.set_rgb_value(brightness, brightness, 0)
    if color == "green":
        led.set_rgb_value(0, brightness, 0)

while True:
    new_count = knob.get_count(reset=False)
    button_pressed_after = knob.is_pressed()

    # If button changes from pressed to not pressed
    if button_pressed_before == True and button_pressed_after == False:
        if color == "white":
            color = "yellow"
        elif color == "yellow":
            color = "green"
        elif color == "green":
            color = "white"

        print(f"Color changed to: {color}")
        set_led_color(color, brightness)

    button_pressed_before = button_pressed_after

    if new_count != last_count:          75
        diff = new_count - last_count
        last_count = new_count

        # Adjust brightness
        brightness += diff * STEP
        brightness = max(0, min(255, brightness))
```

3 Texte

Zusammenfassung

Im dritten Kapitel basteln wir ein universelles Eingabegerät für Informationen. Dafür verwenden wir einen Infrarot-Abstandssensor, mit dem wir Bits kodieren können. Diese Bits könnten dann etwa Texte repräsentieren. Oder ganz andere Dinge.

Der Weg dahin führt über folgende Schritte:

#	Was?	Wo?
1	Wir machen uns mit dem Infrarot-Abstandssensor vertraut.	Abschnitt 3.1
2	Wir bauen eine einfache Lichtschranke.	Abschnitt 3.2
3	Wir erweitern die Lichtschranke zu einer Hinderniserkennung für Roboter.	Abschnitt 3.3
4	Wir erschaffen ein universelles Eingabegerät für Binärcodes.	Abschnitt 3.4
5	Wir geben den Binärcodes eine Bedeutung und übermitteln Texte.	Abschnitt 3.5
6	Wir schauen auf ein verbreitetes Codesystem für Texte: ASCII.	Abschnitt 3.6
7	Wir setzen den Infrarot-Abstandssensor in den LED-Dimmer ein.	Abschnitt 3.7

3.1 Experimentaufbau

3.1.1 Hardware

Für dieses Kapitel benötigen wir erneut die LED ([RGB LED Bricklet 2.0](#)), ersetzen aber den Drehknopf aus Kapitel 2 durch einen Infrarot-Abstandssensor ([Distance IR 4-30cm Bricklet 2.0](#)). Beide Geräte schließen wir an den Mikrocontroller ([Master Brick 3.2](#)) an und fixieren sie auf einer Montageplatte. Wie in der Abbildung gezeigt, befestigen wir den Abstandssensor mit einem Metallwinkel so, dass er bündig mit der Platte ist und nach vorn zeigt.

Die vollständige Hardwareliste für dieses Kapitel sieht so aus:

- 1 x [Master Brick 3.2](#)
- 1 x [RGB LED Bricklet 2.0](#)
- 1 x [Distance IR 4-30cm Bricklet 2.0](#)

- 1 x Montageplatte 22x10
- 2 x Brickletkabel 15cm (7p-7p)
- 2 x Befestigungskit 12 mm

3.1.2 Erste Schritte mit dem Abstandssensor

Wie bei der LED werfen wir zuerst einen Blick auf den neuen Abstandssensor im Brick Viewer. Schließt dazu euren Master Brick per USB an, startet den Brick Viewer und klickt auf Connect. Im Tab Setup sollten nun neben der LED auch der Abstandssensor erscheinen. Denkt daran: Dort findet ihr auch die UID eurer Geräte – die braucht ihr gleich im Programm.

Wechselt in den Tab für den Abstandssensor, wo ihr ihn direkt testen könnt: Ihr seht die aktuelle Entfernung, die der Sensor misst, in Echtzeit oben in der Mitte (in Zentimetern). Darunter zeigt ein Kurvendiagramm den zeitlichen Verlauf. Bewegt eure Hand vor dem Sensor, um ein Gefühl für sein Verhalten zu bekommen. Was passiert, wenn ihr sehr nah vor dem Sensor seid? Und was, wenn ihr eure Hand weiter weg bewegt?

3.2 Lichtschranke

Einen Infrarot-Abstandssensor verwenden wir in der Praxis zur berührungslosen Messung des Abstands zu einem Objekt. Der Sensor sendet einen unsichtbaren Infrarotlichtstrahl aus und misst das Licht, das vom Objekt zurückgeworfen wird. Anhand der Intensität oder des Winkels des zurückkehrenden Lichts berechnet der Sensor die Entfernung. Diese Sensoren sind aus der Praxis nicht wegzudenken und kommen in vielen Szenarien zum Einsatz, zum Beispiel:

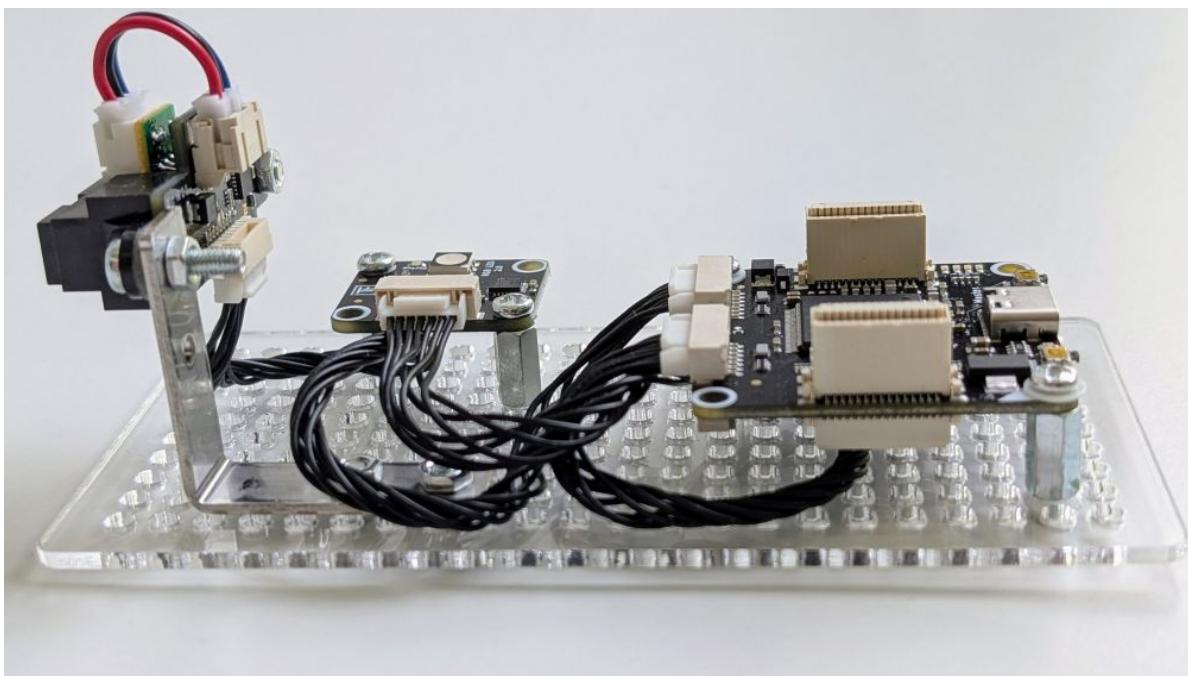
- Objekterkennung an einem Fließband
- Füllstandsmessung in Behältern
- Hinderniserkennung für Roboter, etwa bei Staubsaugerrobotern
- Positionierung von Werkstücken in Maschinen
- Einparkhilfe beim Auto

Letztlich lassen sich alle Anwendungsfälle auf zwei Fragen reduzieren:

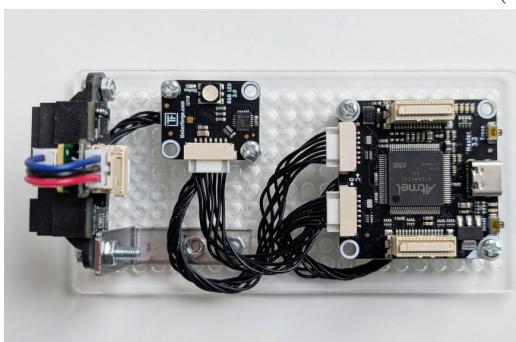
1. Befindet sich ein Objekt in der Sicht des Sensors?
2. Wie weit ist ein Objekt vom Abstandssensor entfernt?

Wir starten mit der einfacheren der beiden Fragen: Befindet sich ein Objekt in der Sicht des Sensors? Das beschreibt im Kern die Funktion einer Lichtschranke.

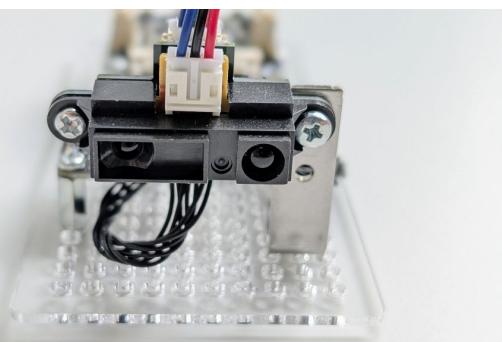
Wie ihr beim Ausprobieren im Brick Viewer festgestellt habt, misst der Sensor Entfernung zwischen 4 und 30 cm. Das bedeutet: Objekte außerhalb dieses Bereichs werden nicht erkannt – weder näher als 4 cm noch weiter als 30 cm. Eine Lichtschranke hat somit die Aufgabe zu



(a) Seitenansicht.



(b) Draufsicht.



(c) Nahaufnahme des IR-Abstandssensors.



(d) Seitenansicht, leicht schräg.

Abbildung 3.1: Einfaches Setup mit einem Mikrocontroller, LED und einem Infrarot-Abstandssensor.

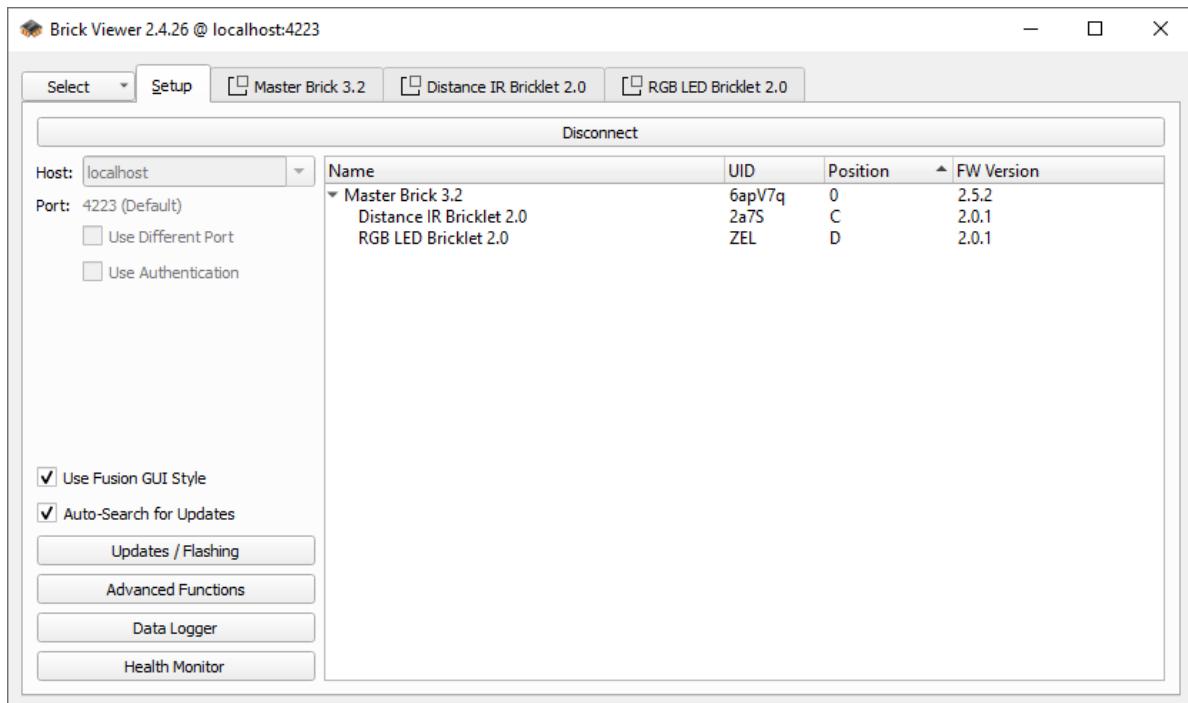


Abbildung 3.2: Nach erfolgreicher Verbindung erscheint der Infrarot-Entfernungsmeister in der Übersicht des Brick Viewers.

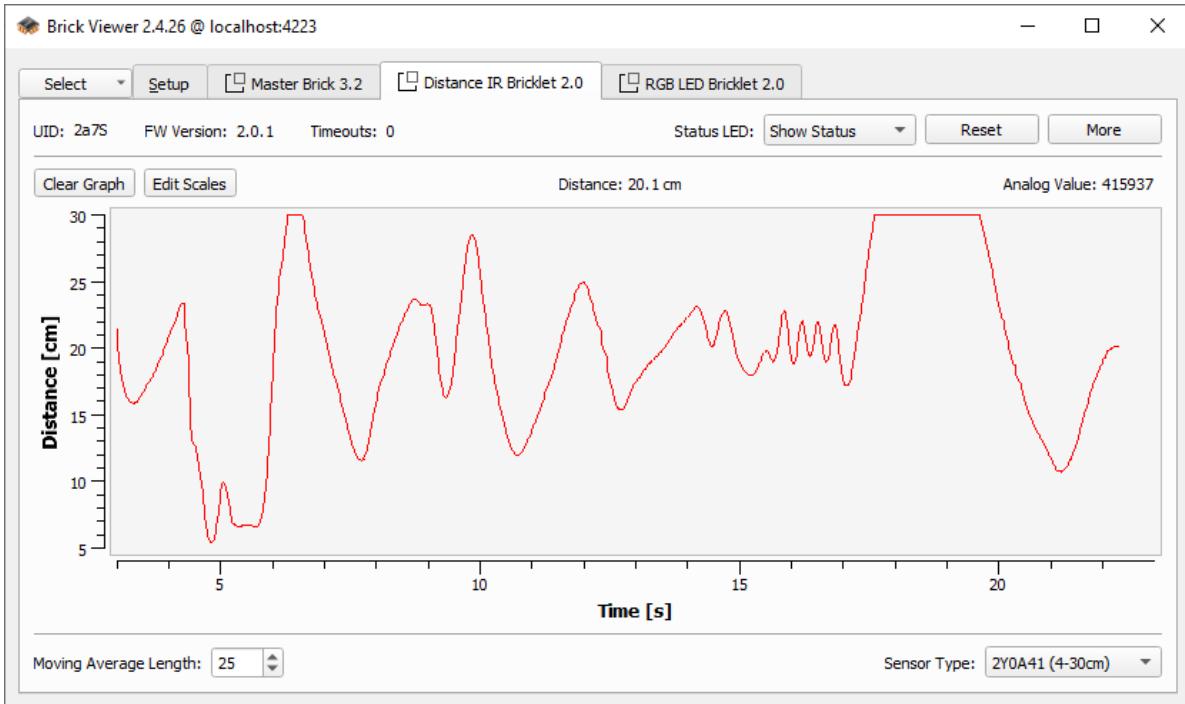


Abbildung 3.3: Der Brick Viewer zeigt die aktuelle Entfernung im Zeitverlauf an.

prüfen, ob der IR-Abstandssensor einen Wert kleiner als 30 cm misst. Dann befindet sich ein Objekt in seiner Sichtlinie.

Verbinden wir uns mit dem Sensor und schreiben das notwendige Grundgerüst. Tragt eure UIDs ein und legt beide Bricklets an, Sensor und LED:

- ① Tragt hier die UID eures IR-Sensors ein.
- ② Tragt hier die UID eurer RGB-LED ein.

Der Sensor ist danach in der Variablen `ir` gespeichert. Über sie können wir [seine Funktionen](#) aufrufen. Eine davon erlaubt uns das Auslesen des aktuell gemessenen Werts:

```
distance = ir.get_distance()
```

Ein Blick in die Dokumentation verrät, dass der Rückgabewert in Millimetern angegeben wird. Ein Wert an der oberen Grenze (rund 300 mm) signalisiert typischerweise, dass sich kein Objekt innerhalb der Reichweite befindet. Damit wir besser testen können, lesen wir den Wert kontinuierlich aus und schreiben ihn auf die Konsole:

Listing 3.1 Boilerplate-Code für IR-Abstandssensor und RGB-LED

```
from tinkerforge.ip_connection import IPConnection
from tinkerforge.bricklet_distance_ir_v2 import BrickletDistanceIRV2
from tinkerforge.bricklet_rgb_led_v2 import BrickletRGBLEDV2
import time

ipcon = IPConnection()
ipcon.connect("localhost", 4223)

ir = BrickletDistanceIRV2("<YOUR_IR_UID>", ipcon)          ①
led = BrickletRGBLEDV2("<YOUR_LED_UID>", ipcon)           ②

while True:
    distance = ir.get_distance()
    print(f"Aktuelle Entfernung: {distance} mm")
```

Ihr werdet sehen, dass wir sehr viele Ausgaben erzeugen, weil in jedem Schleifendurchlauf der Wert ausgegeben wird – auch wenn er sich nicht verändert hat. Geben wir erneut nur die Veränderungen aus, um die Ausgaben zu reduzieren:

```
last_distance = 0
while True:
    distance = ir.get_distance()
    if distance != last_distance:
        print(f"Aktuelle Entfernung: {distance} mm")
        last_distance = distance
```

Ihr erinnert euch: Dasselbe Prinzip haben wir in Abschnitt [2.7](#) verwendet, um nur bei einer Änderung des Drehknopfs eine Ausgabe zu erzeugen. Wir merken uns den letzten Wert und vergleichen ihn mit dem aktuell gemessenen. Ist er gleich, passiert nichts. Hat er sich verändert (`distance != last_distance`), geben wir den neuen Wert aus und aktualisieren den letzten Wert (`last_distance = distance`).

Wir sind unserem Etappenziel einer Lichtschranke schon sehr nah. Anhand der Ausgabe könnten wir entscheiden, ob ein Objekt in der Sicht des Sensors ist oder nicht. Das soll aber unser Programm automatisch erledigen. Dazu fügen wir eine weitere Bedingung mit einer `if`-Anweisung hinzu:

```

last_distance = 0
while True:
    distance = ir.get_distance()
    if last_distance != distance:
        if distance < 300:
            print(f"Objekt erkannt: {distance} mm")
        else:
            print("Kein Objekt in Reichweite")

    last_distance = distance

```

Die neue `if`-Anweisung prüft, ob der Abstand kleiner als 300 mm ist. Dann befindet sich etwas vor dem Sensor. In diesem Fall geben wir einen entsprechenden Hinweis aus. Andernfalls kommt der Hinweis „Kein Objekt in Reichweite“. Diesen anderen Fall bildet der optionale `else`-Teil ab. Code hinter `else` wird immer dann ausgeführt, wenn keine der vorher definierten Bedingungen über `if` oder `elif` zutrifft.

Damit unsere Lichtschranke auch ohne Blick auf die Konsole funktioniert, bringen wir im letzten Schritt die LED ins Spiel. Sie soll rot aufleuchten, wenn ein Objekt erkannt wird. Die LED haben wir bereits kennengelernt, den Code können wir übernehmen:

```

last_distance = 0
while True:
    distance = ir.get_distance()
    if last_distance != distance:
        if distance < 300:
            print(f"Objekt erkannt: {distance} mm")
            led.set_rgb_value(255, 0, 0)                                ①
        else:
            print("Kein Objekt in Reichweite")
            led.set_rgb_value(0, 0, 0)                                  ②

    last_distance = distance

```

- ① Die LED leuchtet rot, wenn ein Objekt erkannt wird.
- ② Die LED schaltet sich aus, wenn kein Objekt mehr erkannt wird.

Probiert es aus – die Lichtschranke sollte funktionieren und Objekte innerhalb von 30 cm zuverlässig erkennen.

3.3 Hinderniserkennung

Die Lichtschranke leuchtet immer rot, sobald ein Objekt erkannt wird. Dabei spielt die Entfernung keine Rolle. Lasst uns die Idee zu einer Hinderniserkennung für einen hypothetischen Staubsaugerroboter erweitern. Die LED soll anzeigen, ob sich ein Objekt bereits nahe am Roboter befindet oder ob es noch weit genug entfernt ist.

Nehmen wir an, dass Objekte, die 17 cm oder näher am Roboter sind, als Gefahr gelten. Alles, was zwischen 17 und 30 cm Abstand hält, sieht der Roboter nicht als bedrohlich an. Die drei Zustände wollen wir über die Farbe der LED abbilden:

- Gelb: Objekt in mittlerer Entfernung erkannt ($17 \text{ cm} < \text{Abstand} < 30 \text{ cm}$)
- Rot: Nahes Objekt erkannt ($\leq 17 \text{ cm}$)
- Aus: Kein Objekt vor dem Sensor ($\geq 30 \text{ cm}$)

Wir können dazu den Code der Lichtschranke erweitern:

Listing 3.2 Der Code für die zweistufige Hinderniserkennung.

```
last_distance = 0
while True:
    distance = ir.get_distance()
    if last_distance != distance:
        if distance > 170 and distance < 300:
            led.set_rgb_value(255, 255, 0)                                ①
        elif distance <= 170:
            led.set_rgb_value(255, 0, 0)                                    ②
        else:
            led.set_rgb_value(0, 0, 0)                                     ③

    last_distance = distance
```

- ① Die LED leuchtet gelb, wenn ein Objekt in mittlerer Entfernung erkannt wird.
- ② Die LED leuchtet rot, wenn ein Objekt in naher Entfernung erkannt wird.
- ③ Die LED schaltet sich aus, wenn kein Objekt mehr erkannt wird.

Ihr erinnert euch an die additive Farbmischung aus Abbildung 1.6a? Gelb entsteht durch die Kombination von Rot und Grün. Wir verwenden hier eine `elif`-Anweisung, die eine weitere Bedingung prüft, wenn die vorherige `if`-Bedingung nicht zutrifft. So können wir mehrere Bedingungen hintereinander prüfen. Der letzte `else`-Teil fängt alle Fälle ab, in denen kein Objekt erkannt wurde.

Unser Staubsaugerroboter könnte den Code oben verwenden, um Hindernisse zu erkennen und bei zu großer Nähe ein Ausweichmanöver zu starten.

3.4 Universelles Eingabegerät

Eine andere Möglichkeit, den Abstandssensor und den Code aus Listing 3.2 zu verwenden, ist ein Eingabegerät für Informationen in unseren Computer. Wie soll ein Abstandssensor als Eingabegerät für Informationen fungieren?

Wie wir später noch sehen werden, benötigen wir für die Darstellung von Informationen unterschiedliche Zustände – mindestens zwei. Genau das verkörpert das Bit, das wir in Abschnitt 2.6 kennengelernt haben: Ein Bit hat zwei Zustände, an oder aus, 0 oder 1.

Was wäre, wenn wir die beiden Bereiche „nah“ und „weit genug entfernt“ nicht länger als Entfernung interpretieren, sondern einfach als zwei Zustände? Sagen wir, der Bereich „weit genug entfernt“ steht für die 1 und der Bereich „nah“ für die 0. Dann könnten wir über die bewusste Platzierung eines soliden Gegenstands und anschließende Messung der Entfernung den Zustand eines Bits kodieren:

```
last_distance = 0
while True:
    distance = ir.get_distance()
    if last_distance != distance:
        if distance > 170 and distance < 300:
            print("1")
        elif distance <= 170:
            print("0")

    last_distance = distance
```

Wenn ihr den Code ausführt, werdet ihr ein Problem erkennen: Platzieren wir unsere Hand nahe am Sensor, um eine 0 zu kodieren, gibt das Programm nacheinander sehr viele Nullen aus. Dasselbe gilt für Einsen. Dabei wollen wir mit einer Handgeste jeweils nur eine 1 oder 0 übermitteln, nicht eine ganze Reihe. Das liegt daran, dass sich unsere Hand ständig minimal bewegt – ein Millimeter reicht.

Wir müssen unser Programm so anpassen, dass es nicht erneut auf eine Änderung reagiert, solange die Hand nicht wieder weggenommen wurde. Das erkennen wir daran, dass der Sensor die maximale Entfernung von 30 cm misst. Erst wenn dieses Ereignis wieder auftritt, soll ein neuer Zustand kodiert werden.

Eine Lösung besteht darin, dass wir uns merken, ob unser Eingabegerät aktuell empfangsbereit ist oder nicht. Wir führen dafür die Variable `receiving` ein, die den Zustand unseres Eingabegeräts beschreibt. Ist sie `True`, ist das Gerät bereit, eine Eingabe zu empfangen. Ist sie `False`, ignorieren wir alle Änderungen, bis die Hand wieder weggenommen wurde.

```

last_distance = 0
receiving = False
while True:
    distance = ir.get_distance()
    if last_distance != distance:

        if receiving:
            if distance > 170 and distance < 300:
                print(f"1 bei {distance} mm")
                receiving = False
            elif distance <= 170:
                print(f"0 bei {distance} mm")
                receiving = False
        else:
            if distance >= 300:
                receiving = True
                print("Bereit für den nächsten Code")

    last_distance = distance

```

- ① Nur im Empfangsmodus wird anhand der Entfernung ein Bit kodiert.
- ② Wenn wir nicht im Empfangsmodus sind, prüfen wir, ob die Hand wieder weggenommen wurde (≥ 30 cm).
- ③ Wenn die Hand weg ist, schalten wir wieder in den Empfangsmodus.

Das sieht schon gut aus. Probiert aber einmal aus, eure Hand sehr langsam von oben nach unten vor den Sensor zu bewegen, und zwar im nahen Bereich, sodass eigentlich eine 0 kodiert werden sollte. In manchen Fällen erkennt das Programm fälschlicherweise eine 1 statt der 0. Warum ist das so? Der Sensor hat bei der Messung eine leichte zeitliche Verzögerung. Wenn er aktuell 30 cm Abstand misst und wir unsere Hand langsam nach unten bewegen, misst der Sensor zunächst einen Abstand knapp unter 30 cm. Das Programm reagiert sofort und kodiert eine 1, obwohl wenig später der Sensorwert in den Bereich der 0 kommt (zum Beispiel 8 cm).

Wir können das Problem umgehen, indem wir einen kleinen Verzug einbauen, sobald ein Unterschied erkannt wurde. Nach diesem zeitlichen Verzug messen wir erneut, um sicherzugehen, die korrekte Position der Hand zu erwischen. So könnte das im Code aussehen:

```

last_distance = 0
receiving = False
while True:
    distance = ir.get_distance()
    if last_distance != distance:

```

```

# 100 ms warten und erneut messen
time.sleep(0.1)
distance = ir.get_distance()

if receiving:
    if distance > 170 and distance < 300:
        print(f"1 bei {distance} mm")
        receiving = False
    elif distance <= 170:
        print(f"0 bei {distance} mm")
        receiving = False
else:
    if distance >= 300:
        receiving = True
    print("Bereit für den nächsten Code")

last_distance = distance

```

Testet es jetzt: Unser Eingabegerät erkennt die Zustände 0 und 1 zuverlässig.

3.5 Texte kodieren

Können wir unser universelles Eingabegerät dazu verwenden, dem Computer Texte zu diktieren? Schließlich bedeutet „universell“, dass es für viele Zwecke einsetzbar ist. Und die Antwort lautet: ja! Wenn wir ein Gerät entwickeln, mit dem wir Bits kodieren können, können wir damit alles eingeben, was ein Computer darstellen kann.

In Kapitel 2 haben wir gesehen, wie wir mit 8 Bits den Wert einer der drei Grundfarben im RGB-Code darstellen können. Wenn wir unser Eingabegerät einsetzen, um hintereinander 24 Bits zu übermitteln und diese als einen RGB-Farbcde zu interpretieren, könnten wir damit unsere LED in einer beliebigen Farbe aufleuchten lassen. Das versuchen wir später. Jetzt kümmern wir uns um eine ebenso wichtige Form der Information: Texte.

Texte bestehen allgemein aus Zeichen. Die meisten Zeichen in Texten sind Buchstaben, die wir in Klein- und Großbuchstaben unterscheiden. Dazu kommen Zahlen und Satzzeichen. Schaut auf das Keyboard eures Computers – dort findet ihr die meisten Zeichen, die ihr für Texte benötigt.

3.5.1 Wie viele Bits benötigen wir?

Genau wie bei den Farben, für die wir 24 Bits benötigen (jeweils 8 pro Farbe im RGB-Code), stellt sich bei Texten die Frage, wie viele Bits wir benötigen, um ein Zeichen darzustellen. Die Antwort hängt von der Anzahl der benötigten Zeichen ab.

Nähern wir uns von der anderen Seite und erweitern unseren Zeichencode Bit für Bit. Wir beginnen klein und fangen mit einem Bit an. Wenn wir Bits als Text interpretieren, wie viele Zeichen (oder Buchstaben) können wir dann mit einem einzigen Bit darstellen? Richtig: zwei!

```
last_distance = 0
receiving = False
while True:
    distance = ir.get_distance()
    if last_distance != distance:

        # 100 ms warten und erneut messen
        time.sleep(0.1)
        distance = ir.get_distance()

        if receiving:
            if distance > 170 and distance < 300:
                print("B")
                receiving = False
            elif distance <= 170:
                print("A")
                receiving = False
        else:
            if distance >= 300:
                receiving = True

    last_distance = distance
```

Lasst das Programm laufen und legt eure Hand einmal nahe vor den Sensor, dann zweimal weiter weg und wieder nah. Das habt ihr gerade geschrieben: „ABBA“.

Neben der bekannten schwedischen Band lassen sich mit den Buchstaben A und B jedoch nicht viele andere Wörter bilden. Wir sind also gut beraten, mindestens ein zweites Bit hinzuzunehmen. Die Anzahl Bits, die wir für einen Buchstaben benötigen, erhöht sich damit auf zwei. Wie bilden wir das im Programm ab?

Am einfachsten, indem wir uns die Bits zunächst merken, sie also hintereinander in eine Zeichenkette schreiben. Sobald eine vorher definierte Länge einer Nachricht – hier zunächst zwei

Bits – erreicht ist, dekodieren wir die Bitfolge und erhalten den passenden Buchstaben. Danach geht es wieder von vorn los und unsere Bit-Zeichenkette ist wieder leer.

```
MESSAGE_LENGTH = 2  
bits = ""  
text = ""  
last_distance = 0  
receiving = False  
  
while True:  
    distance = ir.get_distance()  
    if last_distance != distance:  
  
        time.sleep(0.1)  
        distance = ir.get_distance()  
  
        if receiving:  
            if distance > 170 and distance < 300:  
                print("1")  
                bits += "1"  
                receiving = False  
            elif distance <= 170:  
                print("0")  
                bits += "0"  
                receiving = False  
  
            if len(bits) == MESSAGE_LENGTH:  
                print(f"Bits: {bits}")  
                letter = decode_letter(bits)  
                print(f"Buchstabe: {letter}")  
                text += letter  
                print(f"Text: {text}")  
                bits = ""  
        else:  
            if distance >= 300:  
                receiving = True  
  
    last_distance = distance
```

- ① Die Länge einer Nachricht. So viele Bits müssen wir sammeln, bis wir die Nachricht entschlüsseln können.
- ② Wir erstellen eine leere Zeichenkette `bits`, in der wir jedes empfangene Bit speichern.
- ③ `text` sammelt die dekodierten Buchstaben zu einem Text.

- ④ Wir merken uns das Bit, indem wir es an das Ende von `bits` hinzufügen – hier eine „1“.
- ⑤ Dasselbe für eine „0“.
- ⑥ Wenn wir genug Bits zusammen haben, dekodieren wir die Bitfolge.
- ⑦ `decode_letter(bits)` wandelt die Bitfolge in einen Buchstaben um. Die Funktion implementieren wir gleich.
- ⑧ Wir fügen den Buchstaben dem bisherigen Text an.
- ⑨ Danach setzen wir die Bit-Zeichenkette zurück.

Direkt nach dem Start wartet das Programm darauf, dass ihr eure Hand vor den Sensor haltet. Jede erkannte Änderung wird nach 100 Millisekunden noch einmal gemessen, um den Wert zu stabilisieren. Befinden wir uns im Empfangsmodus (`receiving` ist `True`), schreiben wir je nach Abstand eine „1“ (weit) oder „0“ (nah) ans Ende der Zeichenkette `bits` und schalten den Empfang vorübergehend aus. Sobald die Länge von `bits` der erwarteten `MESSAGE_LENGTH` entspricht, rufen wir `decode_letter(bits)` auf, erhalten den passenden Buchstaben, hängen ihn an `text` an und leeren `bits`. Erst wenn der Sensor wieder mindestens 30 cm misst, schalten wir den Empfang erneut frei, damit die nächste Eingabe beginnen kann.

Okay – probieren wir es aus. Unser Programm sammelt das erste Bit, dann das zweite und dann...

```
NameError: name 'decode_letter' is not defined
```

Was ist das? Eine Fehlermeldung (wie sprechen auch von Bugs), die uns sagt: Die Funktion `decode_letter()` ist nicht definiert. Wir müssen sie also noch implementieren. Wir haben die Funktion zwar schon namentlich genannt, aber es gibt nirgends eine Definition. Das holen wir jetzt nach.

Erinnert euch an Abschnitt 2.10: Wir müssen wissen, was die Funktion tun soll, was sie dafür benötigt und was sie zurückgibt. Einen Namen haben wir bereits: `decode_letter`.

Die Funktion soll unsere Zeichenkette voller Bits der Länge zwei, also so etwas wie „00“, „01“, „10“ oder „11“, in einen Buchstaben umwandeln. Die Eingabe ist `bits` und die Ausgabe ein Buchstabe, den diese Bitfolge kodiert. Unsere Funktion könnte so aussehen:

```
def decode_letter(bits):
    if bits == "00":
        return "A"
    elif bits == "01":
        return "B"
    elif bits == "10":
        return "C"
    elif bits == "11":
        return "D"
```

Mit einem `if`-Statement, begleitet von drei `elif`-Zweigen, prüfen wir, welchem der möglichen Werte die Zeichenkette `bits` entspricht, und geben einen Buchstaben A, B, C oder D zurück. Da es mit zwei Bits insgesamt vier Möglichkeiten gibt, können wir auch nur vier Buchstaben damit abbilden. Wir erweitern das weiter unten, damit wir alle Buchstaben des Alphabets abdecken können.

Tabelle 3.2: Unser aktuelles Codesystem für vier Buchstaben.

Bitfolge	Dezimalzahl	Buchstabe
00	0	A
01	1	B
10	2	C
11	3	D

Die Tabelle fasst unser Codesystem zusammen. In der zweiten Spalte haben wir zur Bitfolge die entsprechende Dezimalzahl eingetragen. Erinnert euch: Das Binärsystem ist ein Stellenwertsystem wie jedes andere auch, nur eben zur Basis 2. Wie ihr die entsprechende Dezimalzahl zu einer Binärzahl errechnet, haben wir in Abschnitt 2.5.3 gelernt.

Fügen wir die Funktion in unser Programm ein. Wichtig ist: Eine Funktion muss vor ihrer Verwendung definiert sein.

```
MESSAGE_LENGTH = 2
bits = ""
text = ""
last_distance = 0
receiving = False

def decode_letter(bits):
    if bits == "00":
        return "A"
    elif bits == "01":
        return "B"
    elif bits == "10":
        return "C"
    elif bits == "11":
        return "D"

while True:
    distance = ir.get_distance()
    if last_distance != distance:
```

```

time.sleep(0.1)
distance = ir.get_distance()

if receiving:
    if distance > 170 and distance < 300:
        print("1")
        bits += "1"
        receiving = False
    elif distance <= 170:
        print("0")
        bits += "0"
        receiving = False

    if len(bits) == MESSAGE_LENGTH:
        print(f"Bits: {bits}")
        letter = decode_letter(bits)
        print(f"Buchstabe: {letter}")
        text += letter
        print(f"Text: {text}")
        bits = ""

else:
    if distance >= 300:
        receiving = True

last_distance = distance

```

Cool, neben „ABBA“ können wir jetzt auch „ADAC“ schreiben. Wir wollen aber natürlich noch mehr, und bevor wir Bit für Bit hinzufügen, überlegen wir, wie viele Bits wir eigentlich benötigen.

Es gibt 26 Buchstaben im Alphabet, und vielleicht wollen wir auch ein Leerzeichen kodieren. Die Unterscheidung zwischen Klein- und Großbuchstaben lassen wir an dieser Stelle einmal weg – sie wäre aber für ein praxistaugliches Codesystem wichtig. Somit sind es 27 Zeichen, die wir kodieren wollen. Mit jedem zusätzlichen Bit verdoppeln wir unsere Möglichkeiten, das haben wir in Abschnitt 2.6 gelernt. Rufen wir uns noch einmal die Tabelle in den Sinn, um zu erkennen, wie viele Bits wir benötigen.

Tabelle 3.3: Anzahl Bits und mögliche Kodierungen.

Anzahl Bits	Mögliche Kodierungen
1	$2^1 = 2$
2	$2^2 = 4$

Anzahl Bits	Mögliche Kodierungen
3	$2^3 = 8$
4	$2^4 = 16$
5	$2^5 = 32$
6	$2^6 = 64$
7	$2^7 = 128$
8	$2^8 = 256$

Demnach reichen uns fünf Bits aus, denn damit können wir insgesamt 32 Kodierungen vornehmen. Wir hätten somit noch fünf freie Plätze, die wir vielleicht für Satzzeichen wie Punkt oder Komma verwenden.

Um das in unserem Programm zu reflektieren, müssen wir die Funktion `decode_letter` anpassen und gleichzeitig die Länge einer Nachricht auf 5 Bits erhöhen. Damit wir es etwas einfacher haben und die Buchstaben den Dezimalzahlen von 0–25 zuordnen können, wandeln wir die Bitfolge zuerst in eine Dezimalzahl um:

```
def decode_letter(bits):

    # In Dezimalzahl umwandeln
    decimal = int(bits, 2) ①

    if decimal == 0:
        return "A"
    elif decimal == 1:
        return "B"
    elif decimal == 2:
        return "C"
    elif decimal == 3:
        return "D"
    elif decimal == 4:
        return "E"
    ...
    elif decimal == 25:
        return "Z"
    else:
        return "?"
```

- ① Die Funktion `int` wandelt die Bitfolge in eine Dezimalzahl um. Der erste Parameter ist die Bitfolge als String, der zweite die Basis (hier 2 für Binärzahlen).

Für die Umwandlung der Bitfolge verwenden wir die Funktion `int()`, die uns später noch öfter begegnen wird. Sie wandelt Zeichenketten in ganze Zahlen um, und wenn wir als zweiten Parameter die Basis des Zahlensystems angeben, funktioniert das auch mit Binärzahlen.

Die Lösung funktioniert, allerdings ist sie nicht besonders elegant. Wir müssen für jeden Buchstaben einen eigenen `if/elif`-Zweig schreiben, was schnell unübersichtlich wird. Zudem wird unser Code extrem lang – im Codeblock oben deutet ... bereits an, dass es noch viele weitere Buchstaben zwischen E und Z gibt. Glücklicherweise geht das eleganter, und zwar mit einem Wörterbuch.

3.5.2 Wörterbücher

Ein Dictionary (deutsch: Wörterbuch) ist in der Programmierung eine Sammlung von Schlüssel-Wert-Paaren. Über einen Schlüssel – zum Beispiel eine Zahl – greifen wir direkt auf den zugehörigen Wert zu, etwa einen Buchstaben. Stellt euch das wie ein Telefonbuch vor, bei dem ihr über den Namen die Nummer herausfindet. Das ist ideal, wenn wir Bitfolgen zuerst in Dezimalzahlen umwandeln und dann schnell den passenden Buchstaben nachschlagen möchten. Statt viele `if/elif`-Zweige zu schreiben, legen wir einmalig eine Nachschlagetabelle an. Das macht den Code kürzer, übersichtlicher und leichter erweiterbar.

Für unsere Zeichendekodierung können wir ein Dictionary nutzen, das die Dezimalwerte 0–25 auf „A“–„Z“ abbildet und zum Beispiel 26 als Leerzeichen reserviert. Damit wird `decode_letter` deutlich kompakter und leichter zu pflegen.

```
# Nachricht auf 5 Bits erweitern
MESSAGE_LENGTH = 5

# Dictionary mit 0-25 => A-Z und 26 => Leerzeichen
SYMBOLS = {}  
①
SYMBOLS[0] = "A"  
②
SYMBOLS[1] = "B"
SYMBOLS[2] = "C"
SYMBOLS[3] = "D"
SYMBOLS[4] = "E"
...
SYMBOLS[25] = "Z"
SYMBOLS[26] = " "

def decode_letter(bits):
    decimal = int(bits, 2)
    return SYMBOLS.get(decimal, "?")
```

- ① Mit geschweiften Klammern erzeugen wir ein leeres Dictionary.

- ② Mit den eckigen Klammern können wir einem Schlüssel einen Wert zuweisen. Wenn es den Eintrag nicht gibt, wird er neu angelegt. Andernfalls wird er überschrieben.

Wenn ihr später weitere Zeichen (z. B. Punkt oder Komma) ergänzen wollt, könnt ihr sie einfach hinzufügen:

```
SYMBOLS[27] = ","
SYMBOLS[28] = "."
```

Wie ihr an der Schreibweise von `SYMBOLS` erkennen könnt, handelt es sich um eine Konstante. Logisch, schließlich verändert sich unser Codesystem für die Symbole im Verlauf des Programms nicht. Wir initialisieren ein leeres Dictionary mit geschweiften Klammern (Zeile 5). Die Zuweisung der Werte erfolgt dann über die eckigen Klammern, wobei in den eckigen Klammern der Schlüssel (englisch: Key) steht und der Wert, den wir dem Schlüssel zuweisen möchten, hinter dem Gleichheitszeichen folgt.

Prinzipiell könnten wir mit den eckigen Klammern auch Werte abfragen. Wenn wir zum Beispiel `SYMBOLS[0]` schreiben, erhalten wir den Wert "A" zurück. Wenn wir einen Schlüssel abfragen, der nicht existiert, bekommen wir jedoch einen Fehler. Eine bessere Möglichkeit zum Abfragen von Werten bietet daher die `get()`-Methode. Sie liefert den Wert für den Schlüssel zurück, den wir als erstes Argument übergeben. Wenn dieser Schlüssel nicht existiert, bekommen wir den Wert `None` zurück, was robuster ist, als wenn das Programm mit einem Fehler abbräche. Zudem können wir als zweites Argument einen Standardwert angeben, der zurückgegeben wird, wenn der Schlüssel nicht existiert. In unserem Fall ist das ein Fragezeichen "?", das signalisiert, dass die Bitfolge keinem bekannten Buchstaben zugeordnet werden kann.

Überprüft euch selbst: Welcher Wert kommt bei folgendem Aufruf zurück?: `SYMBOLS.get(5)`

Tipp

Das Codebeispiel aus diesem Abschnitt findet ihr auf [GitHub](#).

Klont das Repository und öffnet den Ordner in eurem Visual Studio Code, um es schnell ausführen zu können.

3.6 ASCII-Code

Im vorigen Abschnitt haben wir zusammen überlegt, wie wir Texte kodieren und als Bitfolge über ein universelles Eingabegerät basierend auf einem IR-Abstandssensor übertragen können. Ziemlich cool, findet ihr nicht? Dabei haben wir jedem Buchstaben eine eindeutige Bitfolge zugewiesen und sind letztlich auf unser eigenes 5-Bit-Kodierungssystem gekommen.

Weil eine der ersten Anwendungen mit dem Computer die Verarbeitung von Texten war, haben sich darüber, wie man Texte kodieren kann, schon andere sehr schlaue Leute Gedanken

gemacht. Das Ergebnis ist der ASCII-Code, den es seit den 1960er-Jahren gibt und der weltweit standardisiert ist.

Der ASCII-Code basiert auf sieben Bits und kann somit 128 verschiedene Symbole verwalten. Die vollständige Liste an Symbolen seht ihr in Abbildung 3.4. Die Tabelle enthält fünf Spalten, wobei die ersten vier den Zahlencode für das kodierte ASCII-Zeichen in unterschiedlichen Zahlensystemen angeben. Ganz links seht ihr die Dezimalzahl, daneben die Binärdarstellung. Zusätzlich wird noch die Oktalzahl und die Hexadezimalschreibweise mit angezeigt. Oktal kennen wir bereits, was es mit Hexadezimal auf sich hat, lernen wir später in Kapitel 4.

Wenn ihr genau hinseht, erkennt ihr, dass die Binärzahl nicht aus 7, sondern aus 8 Bits besteht. Die erste Ziffer ganz links ist aber immer 0. Das liegt daran, dass Computer häufig in 8-Bit-Einheiten – also in Bytes – denken. Auch der Speicher eines Computers ist so organisiert. Deshalb benötigt ein ASCII-Symbol in der Praxis statt sieben, acht Bits auf unserem Computer.

Was passiert mit dem vermeintlich „verschwendeten“ Bit? Damit könnten wir doch immerhin 128 weitere Symbole abbilden. Und das wird auch gemacht: Es gibt verschiedene lokale Erweiterungen des ASCII-Codes, die zusätzliche Zeichen definieren. Für den deutschen Sprachraum etwa die Umlaute ä, ö, ü und das scharfe S (ß). Für andere Sprachen gibt es ähnliche Erweiterungen. Wir nennen diese Erweiterungen auch Codepages.

Die Codepage, die wir üblicherweise verwenden, nennt sich Windows-1252 (CP-1252) und ist eine Erweiterung des ASCII-Codes, die 256 Zeichen umfasst. Die ersten 128 Zeichen sind identisch mit dem ASCII-Code, die weiteren 128 enthalten zusätzliche Zeichen, die in westeuropäischen Sprachen benötigt werden.

Der ASCII-Code – das steht für American Standard Code for Information Interchange – beinhaltet ein paar nette Eigenschaften. So können wir zum Beispiel einen Großbuchstaben in einen Kleinbuchstaben umwandeln, indem wir 32 zu seinem Dezimalcode addieren. Umgekehrt funktioniert das natürlich auch.

Für unsere Texteingabe über den IR-Abstandssensor bedeutet das: Wir benötigen überhaupt keinen eigenen Code, sondern können einfach den ASCII-Code verwenden. Allerdings müssen wir unsere Nachricht auf die Länge 7 erweitern, was mehr Aufwand bei der Eingabe macht. Dafür verwenden wir einen Standard. Argument genug – passen wir den Code an.

Die Änderungen finden im Wesentlichen in der Funktion `decode_letter()` statt. Zudem ändern wir den Wert der Konstante `MESSAGE_LENGTH`:

```
MESSAGE_LENGTH = 7 # Anzahl Bits pro Buchstabe  
...  
  
def decode_letter(bits):  
    # Links eine 0 hinzufügen, damit es 8 Bits sind
```

Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII
0	00000000	000	00	NUL	32	00100000	040	20	SP	64	01000000	100	40	@	96	01100000	140	60	'
1	00000001	001	01	SOH	33	00100001	041	21	!	65	01000001	101	41	A	97	01100001	141	61	a
2	00000010	002	02	STX	34	00100010	042	22	"	66	01000010	102	42	B	98	01100010	142	62	b
3	00000011	003	03	ETX	35	00100011	043	23	#	67	01000011	103	43	C	99	01100011	143	63	c
4	00000100	004	04	EOT	36	00100100	044	24	\$	68	01000100	104	44	D	100	01100100	144	64	d
5	00000101	005	05	ENQ	37	00100101	045	25	%	69	01000101	105	45	E	101	01100101	145	65	e
6	00000110	006	06	ACK	38	00100110	046	26	&	70	01000110	106	46	F	102	01100110	146	66	f
7	00000111	007	07	BEL	39	00100111	047	27	'	71	01000111	107	47	G	103	01100111	147	67	g
8	00001000	010	08	BS	40	00101000	050	28	(72	01001000	110	48	H	104	01101000	150	68	h
9	00001001	011	09	HT	41	00101001	051	29)	73	01001001	111	49	I	105	01101001	151	69	i
10	00001010	012	0A	LF	42	00101010	052	2A	*	74	01001010	112	4A	J	106	01101010	152	6A	j
11	00001011	013	0B	VT	43	00101011	053	2B	+	75	01001011	113	4B	K	107	01101011	153	6B	k
12	00001100	014	0C	FF	44	00101100	054	2C	,	76	01001100	114	4C	L	108	01101100	154	6C	l
13	00001101	015	0D	CR	45	00101101	055	2D	-	77	01001101	115	4D	M	109	01101101	155	6D	m
14	00001110	016	0E	SO	46	00101110	056	2E	/	78	01001110	116	4E	N	110	01101110	156	6E	n
15	00001111	017	0F	SI	47	00101111	057	2F	\	79	01001111	117	4F	O	111	01101111	157	6F	o
16	00010000	020	10	DLE	48	00100000	060	30	0	80	01000000	120	50	P	112	01100000	160	70	p
17	00010001	021	11	DC1	49	00100001	061	31	1	81	01000001	121	51	Q	113	01100001	161	71	q
18	00010010	022	12	DC2	50	00100010	062	32	2	82	01000010	122	52	R	114	01100010	162	72	r
19	00010011	023	13	DC3	51	00100011	063	33	3	83	01000011	123	53	S	115	01100011	163	73	s
20	00010100	024	14	DC4	52	00100100	064	34	4	84	01000100	124	54	T	116	01100100	164	74	t
21	00010101	025	15	NAK	53	00100101	065	35	5	85	01000101	125	55	U	117	01100101	165	75	u
22	00010110	026	16	SYN	54	00100110	066	36	6	86	01000110	126	56	V	118	01100110	166	76	v
23	00010111	027	17	ETB	55	00100111	067	37	7	87	01000111	127	57	W	119	01100111	167	77	w
24	00011000	030	18	CAN	56	00110000	070	38	8	88	01010000	130	58	X	120	01110000	170	78	x
25	00011001	031	19	EM	57	00110001	071	39	9	89	01010001	131	59	Y	121	01110001	171	79	y
26	00011010	032	1A	SUB	58	00110010	072	3A	:	90	01010010	132	5A	Z	122	01110010	172	7A	z
27	00011011	033	1B	ESC	59	00110011	073	3B	:	91	01010011	133	5B	[123	01110011	173	7B	{
28	00011100	034	1C	FS	60	00110000	074	3C	<	92	01010000	134	5C	\	124	01110000	174	7C	
29	00011101	035	1D	GS	61	00110001	075	3D	=	93	01010001	135	5D]	125	01110001	175	7D	}
30	00011110	036	1E	RS	62	00110010	076	3E	>	94	01010010	136	5E	^	126	01110010	176	7E	~
31	00011111	037	1F	US	63	00110011	077	3F	?	95	01010011	137	5F	-	127	01110011	177	7F	DEL

Abbildung 3.4: Die ursprüngliche ASCII-Codetabelle kodiert die Symbole als 7-Bit-Binärzahlen

```

bits = "0" + bits

# Binärstring in Dezimalzahl und dann in ASCII-Zeichen umwandeln
decimal = int(bits, 2)
return chr(decimal)

```

Die `decode_letter()`-Funktion fügt nun zunächst dem übergebenen Bit-String, der aus 7 Bits bestehen sollte, eine 0 an den Anfang hinzu. Damit haben wir die 8 Bits aus der ASCII-Tabelle in Abbildung 3.4 komplettiert. Anschließend erfolgt – wie zuvor – die Konvertierung von binär nach dezimal. Die wesentliche Änderung steht in der Zeile darunter: Wir geben das Ergebnis der Funktion `chr()` zurück, der wir den Dezimalwert unseres kodierten Symbols übergeben. Brauchen wir also kein Dictionary mehr?

Ganz genau! Es gibt bereits eine Funktion, die die passenden Symbole für Codes liefern kann. Wenn wir in die offizielle Dokumentation der Funktion `chr()` schauen, dann steht dort:

Return the string representing a character with the specified Unicode code point. For example, `chr(97)` returns the string ‘a’, while `chr(8364)` returns the string ‘€’. This is the inverse of `ord()`.

Die Funktion gibt also die Repräsentation des Codes als Zeichen zurück. Aber was steht da? Mit dem angegebenen Unicode-Codepunkt? Was ist denn jetzt schon wieder Unicode? Wir haben doch gerade über ASCII gesprochen.

3.6.1 Unicode

Unicode ist ein internationaler Standard, der jedes Zeichen aus praktisch allen Schriftsystemen der Welt eindeutig beschreibt. Während ASCII nur 128 Symbole umfasst und damit vor allem die englische Sprache abdeckt, definiert Unicode einen gemeinsamen Zeichensatz mit weit über einer Million möglichen Codepunkten. Ein Codepunkt ist dabei eine Nummer, die einem Zeichen zugeordnet ist, zum Beispiel hat der Buchstabe „A“ den Codepunkt U+0041 und das Eurozeichen „€“ den Codepunkt U+20AC.

Wichtig ist die Unterscheidung zwischen Zeichensatz und Kodierung: Unicode ist der Zeichensatz (die Menge aller Zeichen mit ihren Codepunkten), während Formate wie UTF-8, UTF-16 oder UTF-32 beschreiben, wie diese Codepunkte als Bits und Bytes gespeichert oder übertragen werden. UTF-8 ist heute die am weitesten verbreitete Kodierung im Web. Sie ist variabel lang und hat eine zentrale Eigenschaft: Die ersten 128 Codepunkte (0–127) entsprechen exakt dem ASCII-Code. Dadurch ist UTF-8 vollständig rückwärtskompatibel zu ASCII. Eine reine ASCII-Datei ist zugleich gültiges UTF-8, und Funktionen wie `chr()` und `ord()` in Python arbeiten mit Unicode-Codepunkten. Wenn ihr also `chr(65)` aufruft, erhaltet ihr „A“ – das passt sowohl in ASCII als auch in Unicode. Für Zeichen außerhalb des ASCII-Bereichs verwendet UTF-8 mehr als ein Byte, bleibt aber weiterhin eindeutig und effizient.

UTF-8 verwendet je nach Zeichen unterschiedlich viele Bytes – zwischen einem und vier. Häufige Zeichen wie die ASCII-Buchstaben brauchen nur 1 Byte. Ein „A“ hat den Codepunkt U+0041 und wird in UTF-8 als 0x41 gespeichert. Zeichen mit Akzenten benötigen oft 2 Bytes, zum Beispiel „ä“ (U+00E4) als 0xC3 0xA4. Das Eurozeichen „€“ (U+20AC) braucht 3 Bytes: 0xE2 0x82 0xAC. Ein Emoji wie „ “ (U+1F60A) benötigt 4 Bytes: 0xF0 0x9F 0x98 0x8A.

Eine hilfreiche Analogie: Stellt euch UTF-8 wie einen Paketdienst mit vier Paketgrößen (S, M, L, XL) vor. Die meisten Sendungen (ASCII-Zeichen) passen in Größe S und sind damit sehr platzsparend. Für seltener oder komplexere Zeichen wird automatisch eine größere Paketgröße gewählt. Am „Adressaufkleber“ – den ersten Bits des ersten Bytes – erkennt der Empfänger sofort, wie groß das Paket ist und wie viele Folgebytes er einlesen muss. So bleibt Text kompakt, und reine ASCII-Texte sind automatisch gültiges UTF-8.

Auf diese Weise wird sichergestellt, dass die häufigsten Zeichen möglichst wenig Speicherplatz benötigen, während dennoch alle Zeichen der Welt eindeutig kodiert werden können. Das macht UTF-8 zur bevorzugten Wahl für die Textverarbeitung in modernen Anwendungen und im Web. Ihr könnt das übrigens selbst einmal überprüfen: Erstellt zwei leere Textdateien und speichert beide als UTF-8. Fügt in die erste einen Text nur aus Buchstaben, Zahlen und Leerzeichen ein und in die zweite einen Text mit Sonderzeichen und Emojis. Dabei sollte in

jeder Datei die gleiche Anzahl Zeichen stehen. Wie unterscheidet sich die Größe der Dateien im Dateiexplorer?

3.7 LED-Dimmer 4.0

In Kapitel 2 haben wir bereits drei Versionen eines LED-Dimmers gebaut. Eine letzte, vierte Variante kommt noch dazu: Lasst uns schauen, ob wir die Helligkeit mit dem Abstandssensor steuern können.

Das war wieder einmal eine rhetorische Frage, natürlich können wir das! Mit Computern lässt sich so gut wie jedes Problem lösen, es geht nur um das Wie. Vielleicht habt ihr schon eine Idee, nachdem wir den Abstandssensor in diesem Kapitel schon intensiv kennengelernt haben.

Der Dimmer aus Kapitel 2 lässt die LED in verschiedenen Stufen heller und dunkler leuchten. Über den Drehknopf haben wir zunächst 1er-Schritte umgesetzt, was bedeutete, dass wir 256 Ticks des Drehknopfs benötigten, um die LED auf volle Helligkeit zu schalten. Später haben wir dann eine Konstante STEP eingeführt, um die Schritte zu vergrößern, sodass nur noch eine Umdrehung notwendig war.

Wie lässt sich das auf den Abstandssensor übertragen? Im Gegensatz zum Drehknopf, den wir beliebig lange in eine Richtung drehen können, hat der Abstandssensor einen festen Messbereich zwischen 4 und 30 cm. Das macht unsere Aufgabe einfacher, denn eine Transformation des Messbereichs in den Helligkeitsbereich der LED ist ausreichend. Wenn der Abstandssensor nun einen Wert von 4 cm misst, soll die LED auf 0 % Helligkeit dimmen, und bei 30 cm auf 100 % Helligkeit.

Nehmen wir an, der aktuelle Messwert ist in `distance` gespeichert, dann könnten wir den Helligkeitswert `brightness` so berechnen:

```
brightness = (distance - 40) / (300 - 40) * 255
```

Vergewissern wir uns, dass die Formel richtig ist. Wenn `distance` den Wert 4 cm hat, dann sollte `brightness` 0 sein. Das passt, denn der erste Teil der Formel wird dann 0. Und 0 geteilt durch egal was ergibt 0. Wenn `distance` = 300, dann sollte `brightness` 255 sein. Das passt ebenfalls, denn der erste Teil der Formel wird dann 260 und 260 geteilt durch 260 ergibt 1. Multipliziert mit 255 ergibt 255. Sieht also gut aus.

Der folgende Code baut die Berechnungslogik in den LED-Dimmer aus Abschnitt 2.7 ein, der nur eine Farbe (Weiß) beherrscht. Wir könnten ihn aber genau wie in Abschnitt 2.9 um weitere Farben erweitern:

```

from tinkerforge.ip_connection import IPConnection
from tinkerforge.bricklet_distance_ir_v2 import BrickletDistanceIRV2
from tinkerforge.bricklet_rgb_led_v2 import BrickletRGBLEDV2

ipcon = IPConnection()
ipcon.connect("localhost", 4223)
ir = BrickletDistanceIRV2("<YOUR_IR_UID>", ipcon)
led = BrickletRGBLEDV2("<YOUR_LED_UID>", ipcon)
led.set_rgb_value(0, 0, 0) # Anfangszustand: aus

last_distance = 0
while True:
    distance = ir.get_distance()
    if last_distance != distance:
        last_distance = distance

    # Abstand (40-300 mm) auf LED-Helligkeit (0-255) abbilden
    brightness = int((distance - 40) / (300 - 40) * 255)
    led.set_rgb_value(brightness, brightness, brightness) # LED-Helligkeit setzen

```

Je näher wir mit der Hand an den Sensor kommen, desto dunkler wird die LED – und umgekehrt.

Damit schließen wir dieses Kapitel ab. Wir haben gelernt, wie Computer Texte kodieren. Der Umweg über unser eigenes Eingabegerät für Binärcodes hat sich gelohnt: Eure Programmier-skills sind gewachsen!

Im folgenden Kapitel beschäftigen wir uns mit Bildern, wie Computer sie sehen, und wie ein Bild auf den Bildschirm kommt.

Listing 3.3 Der vollständige Code für die Texteingabe mit dem Infrarot-Abstandssensor.

```
from tinkerforge.ip_connection import IPConnection
from tinkerforge.bricklet_distance_ir_v2 import BrickletDistanceIRV2
from tinkerforge.bricklet_rgb_led_v2 import BrickletRGBLEDV2
import time

ipcon = IPConnection()
ipcon.connect('localhost', 4223)

ir = BrickletDistanceIRV2('2a7S', ipcon)
led = BrickletRGBLEDV2('ZEL', ipcon)

MESSAGE_LENGTH = 5 # Anzahl Bits pro Buchstabe
bits = ""
text = ""
last_distance = 0
receiving = False

# Dictionary mit 0-25 => A-Z und 26 => Leerzeichen
SYMBOLS = {}
SYMBOLS[0] = "A"
SYMBOLS[1] = "B"
SYMBOLS[2] = "C"
SYMBOLS[3] = "D"
SYMBOLS[4] = "E"
SYMBOLS[5] = "F"
SYMBOLS[6] = "G"
SYMBOLS[7] = "H"
SYMBOLS[8] = "I"
SYMBOLS[9] = "J"
SYMBOLS[10] = "K"
SYMBOLS[11] = "L"
SYMBOLS[12] = "M"
SYMBOLS[13] = "N"
SYMBOLS[14] = "O"
SYMBOLS[15] = "P"
SYMBOLS[16] = "Q"
SYMBOLS[17] = "R"
SYMBOLS[18] = "S"
SYMBOLS[19] = "T"
SYMBOLS[20] = "U"
SYMBOLS[21] = "V"
SYMBOLS[22] = "W"
SYMBOLS[23] = "X"
SYMBOLS[24] = "Y"
SYMBOLS[25] = "Z"
SYMBOLS[26] = " "

def decode_letter(bits):
    decimal = int(bits, 2)
    return SYMBOLS.get(decimal, "?")
```

4 Bilder

Zusammenfassung

Im vierten Kapitel...

Der Weg dahin führt über folgende Schritte:

#	Was?	Wo?
1	Wir machen uns mit dem Display vertraut.	Abschnitt 4.1
2	Wir lernen das Pixel kennen und schalten sie im Display an und aus.	Abschnitt 4.2
3	Wir führen die Bitmap als eine Sammlung von Pixelwerten ein.	Abschnitt 4.3
4	Wir lernen, wie man Buchstaben als Bitmaps darstellen kann.	Abschnitt 4.3.2
5	Wir lernen eine Alternative zu Bitmaps kennen.	Abschnitt 4.4
6	Wir zeigen ein Bild als Bitfolge aus dem Display an.	Abschnitt 4.5
7	Wir lernen, was eine Animation im Computer ist.	Abschnitt 4.7
9	Wir animieren Pacman und erwecken ihn auf dem Display zum Leben.	Abschnitt 4.9

4.1 Experimentaufbau

4.1.1 Hardware

In den Experimenten dieses Kapitels verwenden wir ein einfaches Display, das wir an unseren Master Brick anschließen. Das Display kann einzelne Pixel weiß aufleuchten lassen und damit einfache Bilder und auch Texte darstellen. Insgesamt stehen uns 128 x 64 Pixel zur Verfügung, also 8192 einzelne Bildpunkte. Damit können wir arbeiten.

Die vollständige Hardwareliste für dieses Kapitel sieht so aus:

- 1 x Master Brick 3.2
- 1 x Distance IR 4-30cm Bricklet 2.0
- 1 x OLED 128x64 Bricklet 2.0
- 1 x Montageplatte 22x10
- 2 x Brickletkabel 15cm (7p-7p)

4.1.2 Erste Schritte mit dem Display

Wie immer geht es zuerst in den Brick Viewer. Verbindet euch mit dem Master Brick und ihr solltet die angeschlossenen Geräte sehen können. Den Infrarotsensor kennen wir bereits aus dem letzten Kapitel. Wir konzentrieren uns deshalb auf das neue Display, das ihr im Tab “OLED 128x64 Bricklet 2.0” findet.

Die Oberfläche für das Display seht ihr in Abbildung 4.1. Im Wesentlichen kann das Display drei Dinge:

1. Einzelne Pixel ein- und ausschalten
2. Seinen gesamten Inhalt löschen
3. Text anzeigen, was ein Spezialfall von Punkt 1 ist

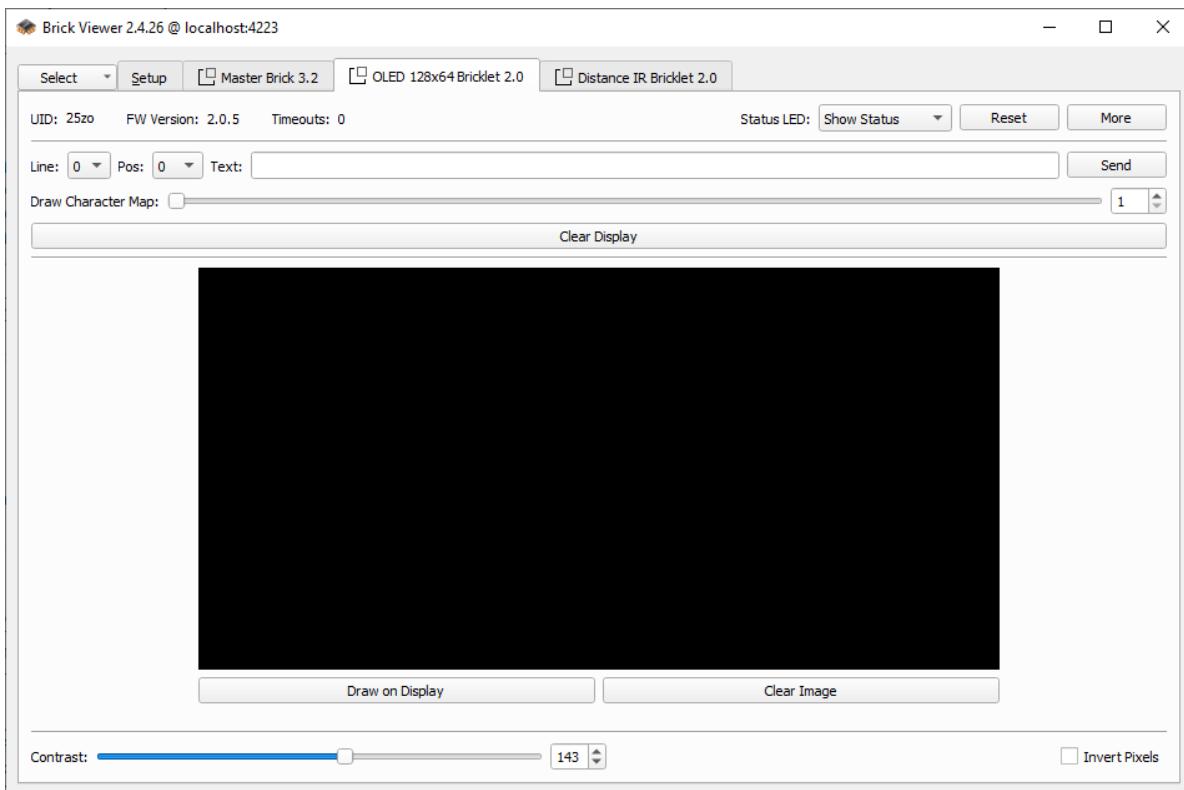
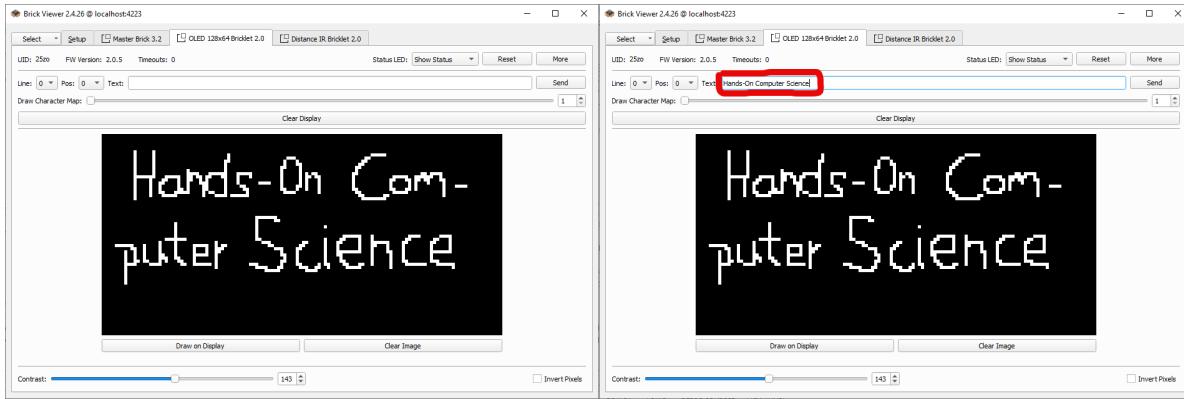


Abbildung 4.1: Im Brick Viewer können wir auf dem Display Text anzeigen oder zeichnen.

Alle genannten Funktionen können wir direkt im Brick Viewer ausprobieren. Mit der Maus können wir auf der schwarzen Fläche freihändig zeichnen und unser Resultat mit “Draw on Display” auf das Display übertragen. Mit “Clear Display” löschen wir den gesamten Inhalt wieder. Alternativ können wir Text in das Textfeld eingeben und mit “Send” anzeigen lassen.

Für Text können wir die Position über die Angabe der Zeile (Line) sowie der Position in der Zeile (Pos) bestimmen. Im Dropdown steht ihr, dass die Zeilen von 0 bis 7 und die Positionen von 0 bis 21 nummeriert sind. Das Display teilt die 128 Pixel Breite in 22 Zeichenpositionen und die 64 Pixel Höhe in 8 Textzeilen auf. Die eingebaute Schrift nutzt ein 5x8-Pixel-Raster pro Zeichen und fügt passende Abstände ein.



- (a) Mein Versuch, den Titel des Buches mit der Maus zu zeichnen.
(b) Das Display unterstützt auch die direkte Textausgabe.

In Abbildung 4.3 könnt ihr den Unterschied zwischen meinem kläglichen Versuch, den Titel des Buches mit der Maus zu zeichnen, und der Textausgabe sehen. Das Display ist zwar nicht besonders groß, aber für einfache Grafiken und Texte reicht es allemal. Wie ihr seht, ist der Text "Hands-On Computer Science" zu lang für eine Zeile und wird daher abgeschnitten.

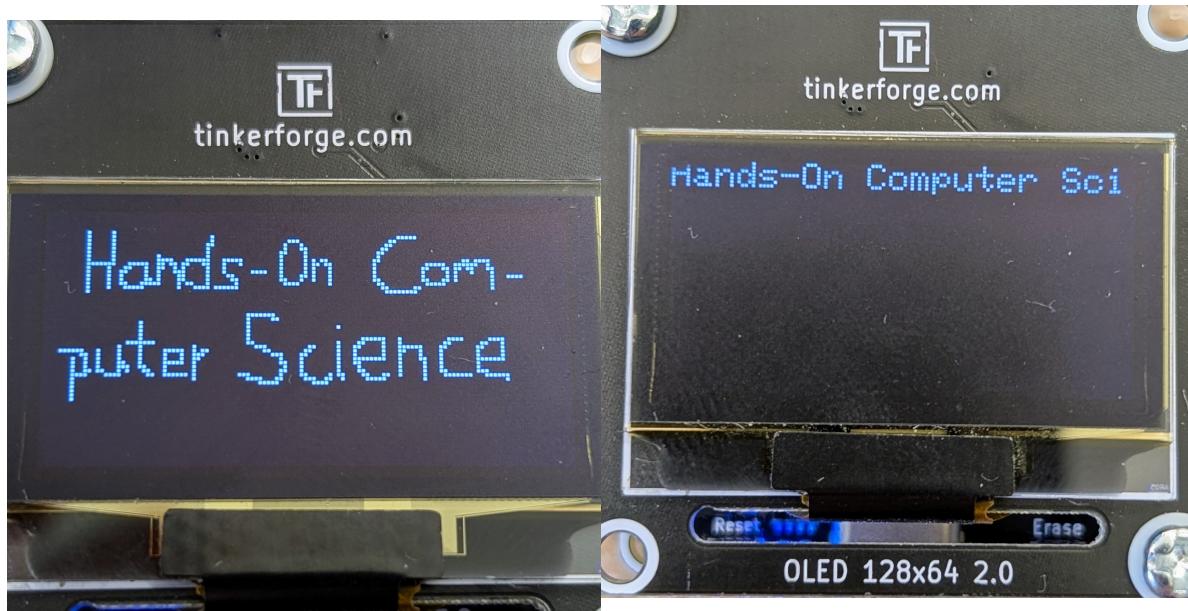
4.2 Pixel

Nachdem wir das Display im Brick Viewer ausprobiert haben, wollen wir uns seine Funktionen in Python anschauen. Dazu stellen wir wie immer die Verbindung her und erzeugen eine Instanz des Display-Objekts:

```
from tinkerforge.ip_connection import IPConnection
from tinkerforge.bricklet_oled_128x64_v2 import BrickletOLED128x64V2      ①

ipcon = IPConnection()
ipcon.connect('localhost', 4223)
oled = BrickletOLED128x64V2('25zo', ipcon)                                ②
oled.clear_display()                                                       ③
```

- ① Wir importieren die Klasse `BrickletOLED128x64V2` aus der Tinkerforge-Bibliothek, die uns die Funktionen des Displays zur Verfügung stellt.



(a) So sieht meine Zeichnung auf dem Display aus.
 (b) Es geht auch direkt als Text, allerdings abgeschnitten.

Abbildung 4.3: Gezeichneter vs. getippter Text auf dem Display.

- ② Denkt dran, eure eigene UID einzutragen.
- ③ Wir löschen den Display-Inhalt, damit wir mit einem leeren Display starten. Die Funktion `clear_display()` erledigt das.

Über die Variable `oled` können wir von nun an die verschiedenen Funktionen des Displays verwenden. Eine davon seht ihr bereits im Codebeispiel, nämlich das Löschen des Displays mit `clear_display()`. Die andere wesentliche Funktion ist das An- und Ausschalten einzelner Pixel.

Das Wort Pixel wird vom englischen “picture element” abgeleitet, also Bildelement. Ein Pixel ist der kleinste darstellbare Punkt auf einem Display. In unserem Fall kann ein Pixel entweder schwarz oder weiß sein. Auf anderen Bildschirmen, wie dem eures Smartphones oder Fernsehers, können Pixel auch farbig sein. Das schauen wir uns später noch genauer an. In diesem Kapitel konzentrieren wir uns auf die einfache schwarzweiße Darstellung. Schließlich kann unser Display nur Schwarzweiß.

Das Display, das wir verwenden, hat eine Auflösung von 128 x 64 Pixeln. Das bedeutet, dass es 128 Pixel in der Breite und 64 Pixel in der Höhe hat. Insgesamt ergibt das somit 8192 Pixel, die wir individuell an- oder ausschalten können.

Beginnen wir, indem wir das Pixel in der linken oberen Ecke des Displays einschalten. Damit wir mit einzelnen Pixel sprechen können, hat jedes eine eigene Koordinate, die einfach die

Position als Zeile und Spalte angibt. Genau wie in einer Excel-Tabelle, in der die Zelle in der dritten Spalte und vierten Zeile mit C4 adressiert würde. Nur verwenden wir bei Pixel statt Buchstaben nur Zahlen.

Die Koordinaten für das Pixel links oben lauten (0, 0), wobei die erste Zahl die x-Koordinate (horizontal) und die zweite Zahl die y-Koordinate (vertikal) ist. Um dieses Pixel einzuschalten, verwenden wir die Funktion `write_pixels()` und rufen sie wie folgt auf:

```
oled.write_pixels(0, 0, 0, 0, [1])
```

Aber warum denn so viele Argumente? Es reicht doch, wenn wir die Koordinaten des Pixels angeben, oder? Der Grund ist, dass die Funktion `write_pixels()` nicht nur einzelne Pixel ansteuern kann, sondern auch rechteckige Flächen von Pixeln. Deshalb müssen wir die Funktion so aufrufen, dass sie eine rechteckige Fläche beschreibt. Abbildung 4.4 zeigt das anhand des Beispiels des linken oberen Pixels.

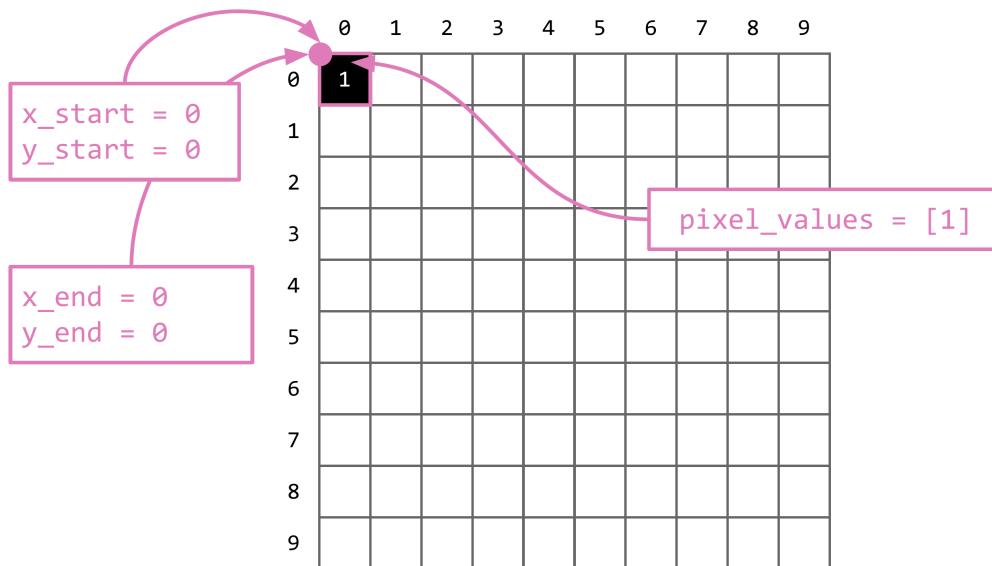


Abbildung 4.4: Die Funktion `write_pixels()` benötigt die Angabe eines Rechtecks und die entsprechenden Pixelwerte als 0 oder 1. Für ein einziges Pixel sind beide Punkte des Rechtecks identisch.

In der Abbildung seht ihr die Werte, die die Funktion `write_pixels()` übergeben bekommt, um ein Pixel anzusteuern. Allgemein lautet die Signatur der Funktion so:

```
write_pixels(x_start, y_start, x_end, y_end, pixel_values)
```

Die ersten vier Argumente definieren die zwei Punkte die rechteckige Fläche als Linksoben und Rechtsunten beschreiben. Die ersten beiden Argumente sind die x- und y-Koordinaten des oberen linken Punkts, die zweiten beiden Argumente sind die x- und y-Koordinaten des unteren rechten Punkts. Diese Eckpunkte sind inklusive, das heißt, die Breite ergibt sich aus `x_end - x_start + 1` und die Höhe aus `y_end - y_start + 1`. In unserem Fall sind beide Punkte `(0, 0)`, wir sprechen also genau ein Pixel an. Der letzte Parameter ist eine Liste von Werten, die angibt, ob die Pixel in der definierten Fläche ein- oder ausgeschaltet werden sollen. Ein Wert von 1 bedeutet weiß, 0 bedeutet schwarz. Da wir nur ein Pixel ansprechen, enthält die Liste nur einen Wert: `[1]`.

Auch wenn es im Beispiel nur ein Wert ist, stellt die `[1]` dennoch eine Liste imn Python dar. Darauf weisen die eckigen Klammern hin. Innerhalb der Klammern können beliebig viele Werte durch Kommas getrennt angegeben werden. Es kann aber auch nur ein Wert sein, wie in unserem Fall.

Wir können das Pixel wieder ausschalten, indem wir den Wert in der Liste auf 0 ändern. Sobald ihr Enter drückt, sollte das Pixel in der linken oberen Ecke wieder ausgehen:

```
oled.write_pixels(0, 0, 0, 0, [1])
input("Drücke Enter um das Pixel auszuschalten...")
oled.write_pixels(0, 0, 0, 0, [0])
```

Einzelne Pixel lassen sich auf die oben gezeigte Weise an- und ausschalten. Aber was ist, wenn wir mehrere Pixel gleichzeitig ansteuern wollen? Zum Beispiel ein kleines Quadrat in der Mitte des Displays? Dafür müssen wir die Koordinaten der beiden Punkte anpassen, die die rechteckige Fläche definieren, und eine Liste mit den entsprechenden Werten für alle Pixel in dieser Fläche erstellen. Wir sprechen hier von einer Bitmap, also einer Matrix von Pixelwerten, die zusammen ein Bild ergeben.

4.3 Bitmaps

4.3.1 Quadrate und andere Formen

Sagen wir, wir wollen ein 2x2 großes Quadrat in der Mitte des Displays zeichnen. Dazu müssen wir zuerst berechnen, wo die Mitte des Displays ist. Da das Display 128 Pixel breit und 64 Pixel hoch ist, läge die Mitte rechnerisch bei `(64, 32)`. Weil wir bei 0 zu zählen beginnen, korrigieren wir auf `(63, 31)`. Um ein 2x2-Quadrat zu zeichnen, setzen wir die Koordinaten des oberen linken Punkts auf `(62, 30)` und die Koordinaten des unteren rechten Punkts auf `(63,`

31). Die Liste der Werte für die Pixel in dieser Fläche muss 4 Werte enthalten, alle auf 1 gesetzt, um das Quadrat anzuzeigen.

```
oled.write_pixels(62, 30, 63, 31, [1,1,1,1])
```

Genau genommen ist die Liste eine flache Struktur, sie wird aber als 2x2-Matrix interpretiert. Die Bitmap sieht also so aus:

```
1 1  
1 1
```

Abbildung 4.5 zeigt das Konzept der Bitmap für unser 2x2-Quadrat. Der Startpunkt oben links unterscheidet sich in diesem Beispiel von dem Endpunkt unten rechts, weil wir nicht nur ein einzelnes Pixel ansteuern, sondern eine quadratische Fläche.

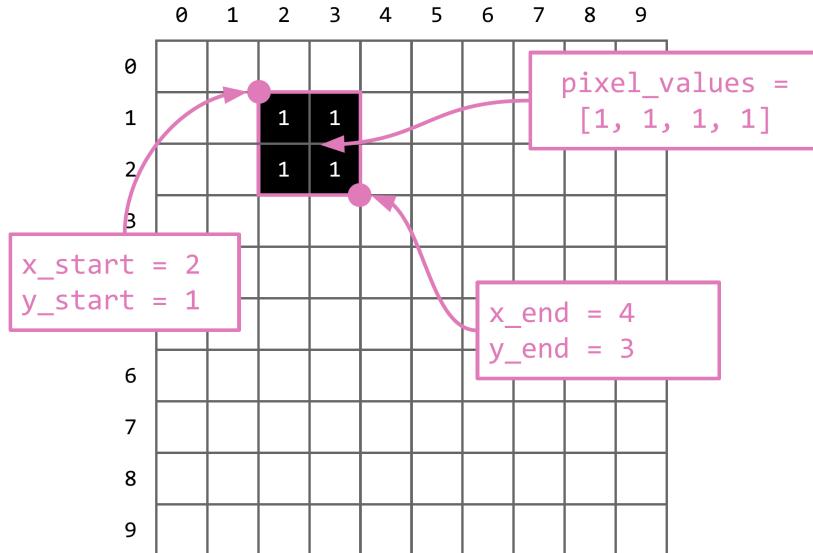


Abbildung 4.5: Ein 2x2 Quadrat über die Angabe der linken oberen sowie der unteren rechten Ecke.

Was ist, wenn wir das Quadrat auf 3x3 vergrößern wollen? Dann ändern wir den unteren rechten Punkt auf (64, 32) oder alternativ den oberen linken Punkt auf (61, 29). Die Liste der Werte erweitern wir auf 9 Einträge:

```
oled.write_pixels(61, 29, 63, 31, [1]*9)
```

Die Python-Syntax `[1]*9` erzeugt eine Liste mit 9 Einsen. Das ist eine praktische Abkürzung, um lange Listen mit gleichen Werten zu erstellen.

Die Idee von `write_pixels()` ist also, dass wir eine rechteckige Fläche definieren und dann eine Liste von Werten bereitstellen, die angibt, ob jedes Pixel in dieser Fläche ein- oder ausgeschaltet wird. Die Länge der Liste muss genau der Anzahl der Pixel in der Fläche entsprechen. Die Werte in der Liste werden zeilenweise von links nach rechts und von oben nach unten gelesen.

In Abbildung 4.6 sehen wir ein weiteres Beispiel, ebenfalls für eine Bitmap mit 3x3 Pixeln. Jedoch sind dieses Mal nicht alle Pixel angeschaltet, sondern nur die Pixel, die ein Kreuzmuster ergeben. Als Liste sieht das so aus:

```
[0, 1, 0, 1, 1, 1, 0, 1, 0]
```

Als Matrix dargestellt, wie es auch auf dem Display erscheint, sieht das so aus:

```
0 1 0  
1 1 1  
0 1 0
```

In der Matrixdarstellung erkennt man schnell das Kreuzmuster, das wir gezeichnet haben.

Wenn wir dieses Kreuzmuster öfters zeichnen wollen, dann wäre es eine gute Idee, die Liste der Pixelwerte in einer Variablen zu speichern. So können wir sie immer wieder verwenden, ohne die Liste jedes Mal neu schreiben zu müssen. Das sieht dann so aus:

```
cross_bitmap = [  
    0, 1, 0,  
    1, 1, 1,  
    0, 1, 0  
]
```

Jetzt können wir das Kreuzmuster einfach zeichnen, indem wir die Variable `cross_bitmap` an die Funktion `write_pixels()` übergeben:

```
oled.write_pixels(0, 0, 2, 2, cross_bitmap)
```

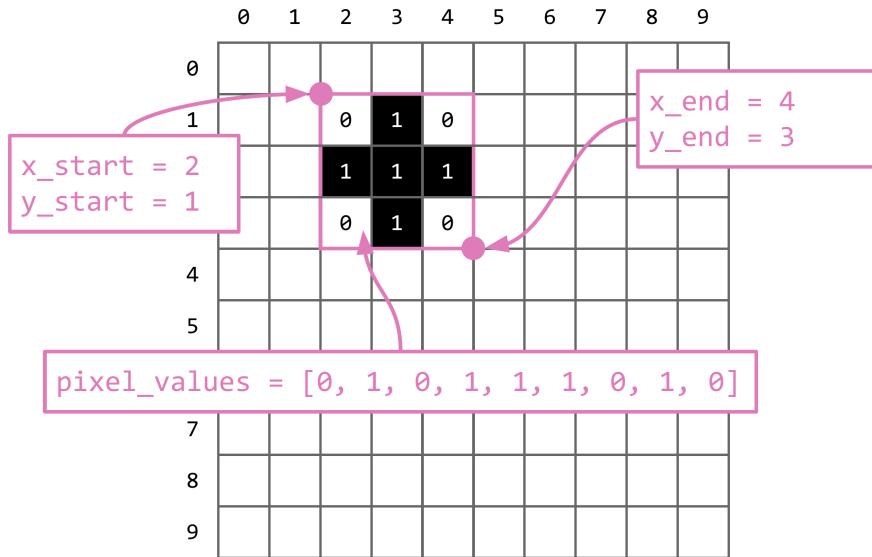


Abbildung 4.6: Über die Pixelwerte lassen sich beliebige Symbole darstellen, hier ein Kreuz.

Da wir als Quadrat (0, 0) bis (2, 2) angegeben haben, wird das Kreuz in der linken oberen Ecke des Displays gezeichnet. Wir können die x- und y-Koordinaten anpassen, um das Kreuz an einer anderen Position zu zeichnen. Zum Beispiel direkt daneben noch eins, mit einem Pixel Abstand dazwischen:

```
oled.write_pixels(0, 0, 2, 2, cross_bitmap)
oled.write_pixels(4, 0, 6, 2, cross_bitmap)
```

Was, wenn wir Kreuze über das gesamte Display zeichnen wollen? Ein Kreuz inklusive Abstand benötigt 3 Pixel auf der x-Achse. Da wir 128 Pixel in der Breite haben, können wir also 42 Kreuze in eine Zeile des Displays zeichnen.

```
oled.write_pixels(0, 0, 2, 2, cross_bitmap)
oled.write_pixels(4, 0, 6, 2, cross_bitmap)
oled.write_pixels(8, 0, 10, 2, cross_bitmap)
...
oled.write_pixels(120, 0, 122, 2, cross_bitmap)
```

Stellt euch statt „...“ 38 weitere Zeilen Code vor, die jeweils ein Kreuz zeichnen. Das wäre ziemlich mühsam. Zum Glück kennen wir dafür schon eine bessere Lösung, die Schleife. Wir

können eine Schleife verwenden, um die x-Koordinate automatisch zu berechnen und die Kreuze in einer einzigen Schleife zu zeichnen. So sieht das aus:

```
for x in range(0, 128, 4):
    oled.write_pixels(x, 0, x + 2, 2, cross_bitmap)
```

Erinnert euch: Die `range()`-Funktion erzeugt eine Folge von Zahlen. In diesem Fall starten wir bei 0, enden bei 128 und erhöhen die Zahl in jedem Schritt um 4. Dadurch erhalten wir die x-Koordinaten 0, 4, 8, ..., 124. In jedem Schleifendurchlauf zeichnen wir ein Kreuz an der aktuellen x-Position.

Denken wir das Ganze ein Stück weiter und erzeugen ein Display voller Kreuze. Dazu könnten wir unsere Schleife kopieren und die y-Koordinate anpassen, um die Kreuze in mehreren Reihen zu zeichnen. Aber auch hier können wir wieder eine Schleife verwenden, um die y-Koordinate automatisch zu berechnen. So sieht das aus:

```
for y in range(0, 64, 4):
    for x in range(0, 128, 4):
        oled.write_pixels(x, y, x + 2, y + 2, cross_bitmap)
```

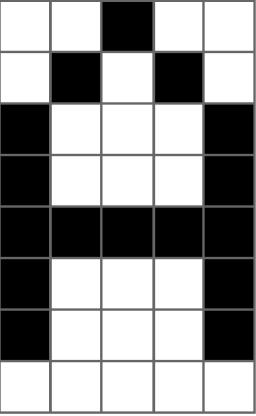
Wenn ihr den Code ausführt, könnt ihr dem Display beim Zeichnen zuschauen. Es füllt sich nach und nach mit Kreuzen, beginnend oben rechts nach unten links, bis das gesamte Display bedeckt ist. Schneller ginge es, wenn wir alle Pixelwerte in einer einzigen Liste speichern und dann auf einmal zeichnen. Das könnt ihr als Übung selbst ausprobieren.

Wenn wir Kreuze als Bitmaps darstellen und zeichnen können, wie sieht es dann mit anderen Symbolen, wie etwa Buchstaben aus?

4.3.2 Buchstaben

Auch Buchstaben auf dem Bildschirm sind nichts anderes als Pixelmuster. Wie bereits erwähnt, ist ein Buchstabe auf dem Display von Tinkerforge 5 Pixel breit und 8 Pixel hoch. Das bedeutet, dass wir für jeden Buchstaben eine Bitmap mit 40 Werten benötigen. Tinkerforge, der Hersteller des Displays, stellt eine Übersicht der unterstützten Zeichen und deren Pixelmuster auf [seiner Webseite](#) bereit. Daraus habe ich den Buchstaben "A" als Bitmap in eine einfache Tabelle übertragen und Pixel, die an sind, schwarz eingefärbt. Das Ergebnis seht ihr in Abbildung 4.7.

Anhand dieser Darstellung können wir die Werte für die Liste mit Einsen und Nullen ableiten, die wir benötigen, um den Buchstaben "A" auf dem Display darzustellen. Die Liste sieht so aus:



	0	1	2	3	4
0	0	0	1	0	0
1	0	1	0	1	0
2	1	0	0	0	1
3	1	0	0	0	1
4	1	1	1	1	1
5	1	0	0	0	1
6	1	0	0	0	1
7	0	0	0	0	0

Abbildung 4.7: Der Buchstabe “A” als Bitmap.

```
letter_a_bitmap = [
    0, 0, 1, 0, 0,
    0, 1, 0, 1, 0,
    1, 0, 0, 0, 1,
    1, 0, 0, 0, 1,
    1, 1, 1, 1, 1,
    1, 0, 0, 0, 1,
    1, 0, 0, 0, 1,
    0, 0, 0, 0, 0
]
```

Wer gute Augen hat, erkennt in der Matrix den Buchstaben “A” sofort. Welcher Buchstabe verbirgt sich in der folgenden Bitmap?

```
letter_unknown_bitmap = [
    0, 1, 1, 1, 0,
    1, 0, 0, 0, 1,
    1, 0, 0, 0, 0,
    0, 1, 1, 1, 0,
    0, 0, 0, 0, 1,
```

```

1, 0, 0, 0, 1,
0, 1, 1, 1, 0,
0, 0, 0, 0, 0
]

```

Zeichnen wir den Buchstaben auf dem Display, um es herauszufinden. Wir kennen die Dimensionen (5x8 Pixel), haben die Bitmap als Liste und müssen nur noch die Position bestimmen, an der der Buchstabe gezeichnet werden soll. Ich habe mich für die Position (6, 10) als oberen linken Punkt entschieden:

```
oled.write_pixels(6, 10, 10, 17, letter_unknown_bitmap)
```

Und? Seht ihr auch ein großes “S”?

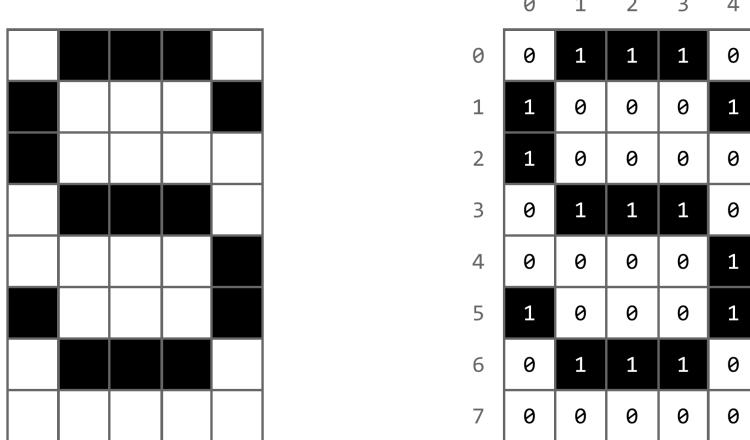


Abbildung 4.8: Der Buchstabe “S” als Bitmap.

Versuchen wir, davor noch das “A” zu schreiben:

```
oled.write_pixels(1, 10, 5, 17, letter_a_bitmap)
```

Wir haben richtig gerechnet: Das “A” soll vor dem “S” stehen, also müssen wir mit der x-Koordinate 5 Pixel nach links gehen. Die y-Koordinate bleibt gleich, weil beide Buchstaben

auf der gleichen Höhe stehen sollen. Lasst uns noch ein Pixel Platz zwischen beiden Buchstaben lassen. Das bedeutet, dass wir die x-Koordinate des “A” um 1 Pixel weiter nach links setzen, also auf 0. Die x-Koordinate des rechten unteren Punkts des “A” nimmt dann den Wert 4 an. Der Code für das “A” sieht so aus:

```
oled.write_pixels(0, 10, 4, 17, letter_a_bitmap)
```

Und nun noch ein “S” ans Ende, damit wir ein sinnvolles Wort geschrieben haben:

```
oled.write_pixels(0, 10, 4, 17, letter_a_bitmap)
oled.write_pixels(6, 10, 10, 17, letter_unknown_bitmap)
oled.write_pixels(12, 10, 16, 17, letter_unknown_bitmap)
```

Sucht euch aus, ob ihr nun eine Spielkarte, den Namen eines Schmerzmittels oder das englischsprachige Wort für Ar*** geschrieben habt. In jedem Fall sind es drei Buchstaben, die wir mit Bitmaps auf dem Display dargestellt haben. Und das ganz ohne die Textfunktion des Displays zu verwenden.

Das wäre übrigens auch einfacher gegangen, wäre aber nicht so lehrreich gewesen:

```
oled.write_line(0, 0, "Ass")
```

Wenn ihr beides hintereinander ausführt, steht oben “Ass” per Textfunktion und darunter “ASS” als eigene Bitmaps. Das Schöne an der `write_line()`-Funktion ist, dass sie automatisch die richtigen Bitmaps für die Buchstaben verwendet. Das Display hat eine eingebaute Schriftart, die es nutzt, wenn wir Text anzeigen wollen. Aber wie ihr gesehen habt, können wir auch eigene Bitmaps verwenden, um Buchstaben oder andere Symbole zu zeichnen, die nicht im Zeichensatz des Displays enthalten sind.

Schriftarten für Pixel-Displays sind im Kern nichts anderes als Sammlungen von Bitmaps: Für jedes Zeichen wird festgelegt, welche Pixel im 5x8-Raster leuchten. Unser “A” oben ist genau so definiert, und das OLED-Bricklet bringt eine solche Tabelle für viele Zeichen bereits mit. Wenn ihr eine andere Schrift wollt, erstellt ihr einfach eine neue Bitmap-Tabelle, etwa eine fette oder eine schmale Variante, und verwendet diese beim Zeichnen. Das funktioniert hervorragend in festen Rastergrößen, stößt aber an Grenzen, sobald sich die Größe der Buchstaben ändern muss, wenn etwa die Schrift vergrößert werden soll. Hier kommen Vektorgrafiken ins Spiel, die Formen nicht als Pixelraster, sondern als Linien und Kurven beschreiben.

4.4 Vektorgrafiken

Während Bitmaps jedes Pixel explizit speichern, beschreiben Vektorgrafiken Objekte über geometrische Formen, etwa “eine Linie von A nach B” oder “ein Kreis mit Mittelpunkt M und Radius r”. Eine Vektor-Schriftart (wie TrueType) enthält keine 5x8-Raster pro Zeichen, sondern Pfade für die Konturen von “A”, “S” etc. Der Vorteil: Diese Formen lassen sich beliebig vergrößern oder verkleinern, ohne dass Treppeneffekte entstehen. Das ist ideal für hochauflösende Displays und Druck. Auf unserem kleinen OLED-Display arbeiten wir dennoch mit Bitmaps, weil sie direkt den Pixeln entsprechen und ohne aufwendiges Rasterisieren angezeigt werden können.

Schaut euch zur Verdeutlichung der Problematik einmal die beiden “a” aus der Abbildung 4.9 an. Das linke “a” ist eine TrueType-Schriftart, die als Vektorgrafik beschrieben wird. Das rechte “a” ist über eine Bitmap definiert und wurde stark vergrößert. Während das rechte “a” pixelig wirkt - wir sprechen auch vom Treppeneffekt, ist die als Vektorgrafik beschriebene Variante gestochen scharf, auch in großen Größen.



Source: [Wikipedia](#)

Abbildung 4.9: Das linke “a” ist eine TrueType-Schriftart, das rechte “a” eine Bitmap. Quelle: [Wikipedia](#).

Wie funktioniert das? Dazu betrachten wir ein anderes Beispiel für eine Vektorgrafik in einem speziellen Format mit dem Namen Scalable Vector Grafik, oder kurz SVG. Kopiert einfach mal den folgenden Code in eine Textdatei und benennt sie `vector_graphics.svg`. Öffnet die Datei anschließend in einem Webbrowser.

```

<svg width="440" height="220" xmlns="http://www.w3.org/2000/svg">
  <circle cx="60" cy="60" r="50" stroke="#0085C7" stroke-width="10" fill="none" />
  <circle cx="180" cy="60" r="50" stroke="#000000" stroke-width="10" fill="none" />
  <circle cx="300" cy="60" r="50" stroke="#DF0024" stroke-width="10" fill="none" />
  <circle cx="120" cy="110" r="50" stroke="#FFD500" stroke-width="10" fill="none" />
  <circle cx="240" cy="110" r="50" stroke="#009F3D" stroke-width="10" fill="none" />
</svg>

```

Ihr solltet ein Bild wie in Abbildung 4.10 sehen. Zoomt nun einmal ganz nach hinhein, indem ihr Strg + Plus (Windows) oder Cmd + Plus (Mac) drückt. Ihr werdet feststellen, dass die Kreise auch bei starker Vergrößerung scharf bleiben, ohne dass Treppeneffekte auftreten. Das liegt daran, dass Vektorgrafiken mathematisch beschrieben werden und nicht auf eine feste Pixelauflösung angewiesen sind.

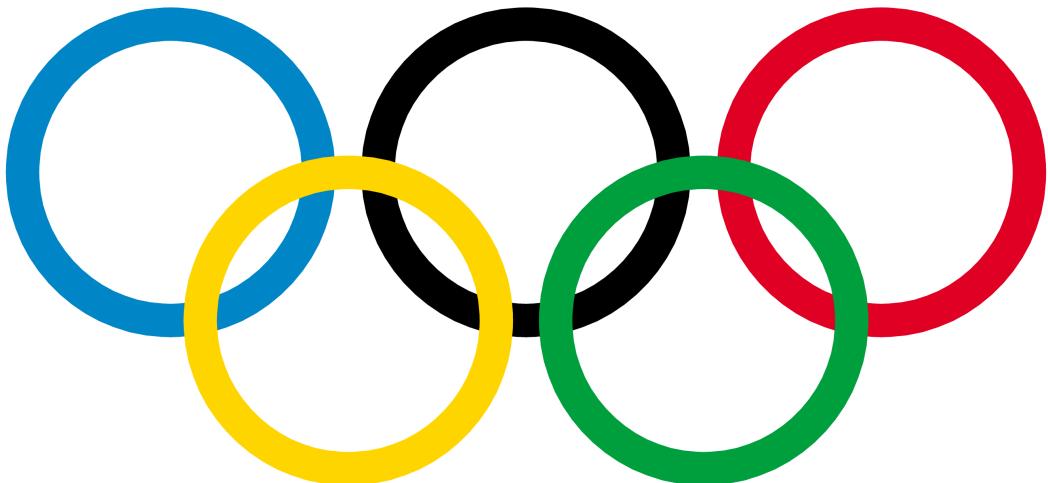


Abbildung 4.10: Ein Versuch der Olympischen Ringe als Vektorgrafik, die man ohne Qualitätsverlust beliebig vergrößern kann.

Aber Moment, wenn Vektorgrafiken auf einem Bildschirm angezeigt werden, müssen sie dann nicht auch als Pixel dargestellt werden? Schließlich besteht doch jedes Bild im Endeffekt aus Pixeln wenn wir es anzeigen, oder? Und das stimmt. Während Vektorgrafiken das, was auf dem Bildschirm erscheinen soll, über geometrische Formen beschreiben, muss das Bild letztlich in eine Bitmap umgewandelt werden, damit es auf dem Bildschirm angezeigt werden kann. Dieser Prozess wird als Rasterisierung bezeichnet.

In Abbildung 4.11 seht ihr die Vektorgrafik von oben, die in eine Bitmap mit niedriger Auflösung (100 x 50 Pixel) umgewandelt wurde. Wenn man wie im Bild hereinzoomt oder das Bild auf eine höhere Auflösung wie 1000 x 500 Pixel bringen möchte, weil man die Ringe vielleicht in einen Großformat auf ein Werbeplakat drucken möchte, dann erkennt man die Treppeneffekte deutlich. Die Auflösung einer Bitmap ist somit entscheidend für die Bildqualität. Je höher die Auflösung, desto mehr Pixel stehen zur Verfügung, um Details darzustellen, und desto schärfer wirkt das Bild. Bei Vektorgrafiken spielt das hingegen keine Rolle: Wir können für ein 3 x 2 m Werbeplakat einfach eine entsprechende hochauflösende Bitmap generieren, ohne dass die Qualität leidet, weil die Vektorgrafik immer die gleichen geometrischen Formen *beschreibt*.

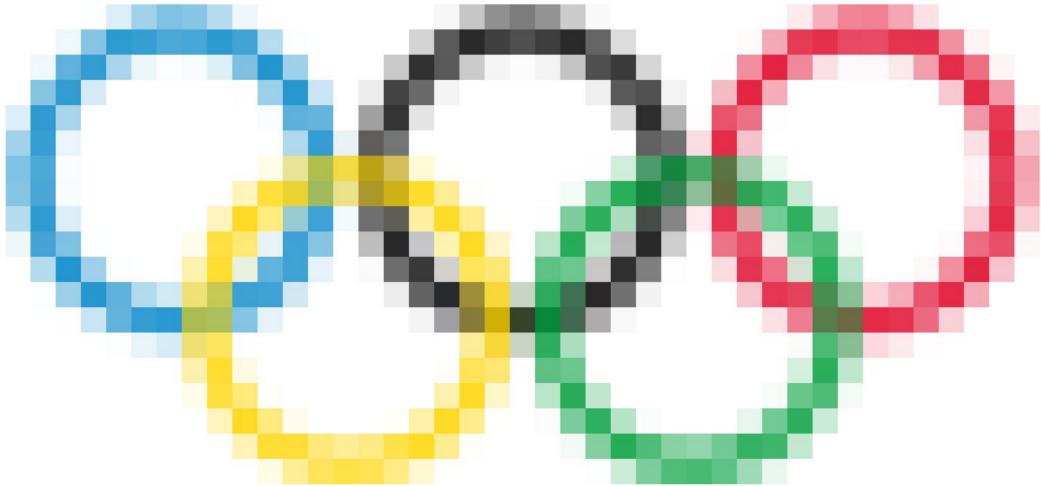


Abbildung 4.11: Die Beschreibung der geometrischen Objekte muss irgendwann in ein Bild aus Pixeln überführt werden. Dann spielt die Auflösung die entscheidende Rolle.

Aufgrund ihrer Eigenschaften werden Vektorgrafiken insbesondere für Logos, Icons und Schriftarten verwendet, die in verschiedenen Größen dargestellt werden müssen. Designagenturen nutzen sie häufig, um sicherzustellen, dass ihre Designs in jeder Größe gut aussehen. Für komplexe Bilder mit vielen Farben und Details, wie Fotos, sind Bitmaps jedoch besser geeignet, weil eine Beschreibung der geometrischen Formen zu aufwendig wäre.

4.5 Von Bits zum Bild

Wir bleiben in diesem Kapitel bei Bitmaps, weil auch eine Vektorgrafik am Ende auf dem Bildschirm nichts anderes als eine Bitmap. Ein Bildschirm kennt nur Pixel.

Egal ob es vorher eine Vektorgrafik war oder direkt als Bitmap vorliegt: Um ein Bild auf unserem Display anzuzeigen, müssen wir es in eine Liste von Pixelwerten umwandeln, die wir dann mit der `write_pixels()`-Funktion zeichnen können. Im Beispiel unseres Displays reicht eine Liste mit Binärwerten (0 und 1) aus, um jedes Pixel als ein- oder ausgeschaltet zu kennzeichnen. Für ein farbiges Display, wie wir es typischerweise bei Computerbildschirmen, Smartphones oder Fernseher finden, wären mehrere Bits pro Pixel notwendig, um die verschiedenen Farben darzustellen. Ihr erinnert euch an den RGB-Farbc ode aus Kapitel 1? Der kann für Farbbilder verwendet werden und mündet im Endeffekt auch in einer Liste von Nullen und Einsen, die die Farbwerte jedes Pixels repräsentieren. Nur ist diese Liste eben deutlich länger als bei Schwarzweiß-Bildern.

Betrachtet einmal das Bild in Abbildung 4.12. Ihr kennt bestimmt, was es zeigt. Es handelt sich um eine Bitmap-Darstellung von Darth Vaders Kopf aus Star Wars. Das Bild ist 27 Pixel breit und 24 Pixel hoch, also insgesamt 648 Pixel. Jedes Pixel ist entweder schwarz oder weiß, was locker auf unser Display passt. Wie aber bekommen wir es nun auf unser Display?

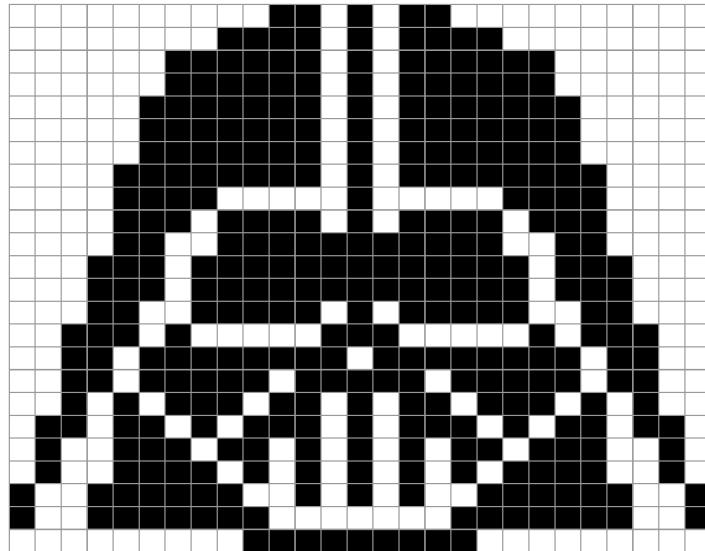


Abbildung 4.12: Darth Vaders Kopf aus Star Wars als Bitmap

Nehmen wir an, die Liste aus Nullen und Einsen liegt uns noch nicht vor. Unser erstes Ziel sollte somit sein, diese Liste zu erstellen. Das könnten wir per Hand versuchen, indem wir Zeile für Zeile von links nach rechts durch das Bild gehen und für jedes weiße Pixel eine 0 und für jedes schwarze Pixel eine 1 notieren. Das wäre aber sehr mühsam und fehleranfällig. Glücklicherweise liegt das Bild bereits digital vor, allerdings nicht als Bildformat, sondern als Excel-Tabelle.

Bitte was? Ein Bild als Excel-Tabelle? Die Idee habe ich aus dem [CS50-Kurs der Harvard University](#) übernommen. Dort erstellen Studierende mithilfe von Excel-Tabellen Pixelbilder, indem sie die Zellen entsprechend einfärben. Jede Zelle entspricht dabei einem Pixel, das entweder schwarz oder weiß ist. Das Ergebnis kann dann als Bitmap interpretiert werden. Die Excel-Datei mit Darth Vaders Maske könnt ihr euch [herunterladen](#) und das Ganze einmal selbst ausprobieren.

Da wir bereits eine digitale Form haben (wenn auch kein Bildformat), können wir das Problem im Sinne des EVA-Modells auffassen: Wir haben Eingabedaten (die Excel-Tabelle) und wollen eine Ausgabe (die Liste von Nullen und Einsen). Der Prozess dazwischen ist die Verarbeitung, die dafür sorgt, dass die Excel-Tabelle gelesen und die Pixelwerte extrahiert werden. Das Ganze wollen wir natürlich als Python-Programm umsetzen, weil das unsere Programmiersprache der Wahl ist. Na dann los!

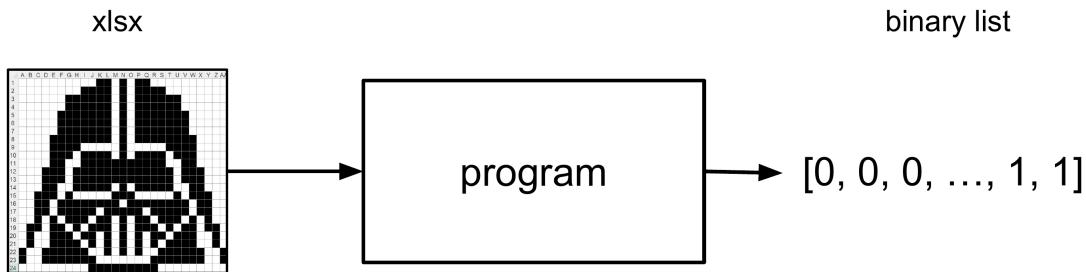


Abbildung 4.13: Die Überführung einer Excel-basierten Bitmap in eine Liste aus Bits im EVA-Modell.

4.5.1 Excel mit Python einlesen

Um Excel-Dateien in Python zu lesen müssen wir das Rad nicht neu erfinden. Es gibt verschiedene Bibliotheken, eine der beliebtesten und einfachsten ist `openpyxl`. Stellt sicher, dass ihr die Bibliothek installiert habt. Das könnt ihr mit dem folgenden Befehl in eurer Kommandozeile tun. Denkt daran, dass MacOS-Nutzer `pip3` anstelle von `pip` verwenden müssen:

```
pip install openpyxl
```

Die Bibliothek `openpyxl` stellt uns eine Funktion mit dem Namen `load_workbook()` zur Verfügung, der wir einfach den Pfad der Excel-Datei übergeben können. Wenn ihr vorher die Excel-Datei mit Darth Vaders Maske heruntergeladen habt, könnt ihr sie so laden:

```
from openpyxl import load_workbook  
workbook = load_workbook("Darth Vader Pixel Art.xlsx")
```

- ① Wir kündigen an, dass wir die Funktion `load_workbook()` aus der Bibliothek `openpyxl` verwenden möchten.
- ② Wir laden die Excel-Datei und speichern das Ergebnis in der Variable `workbook`.

Da ein Excel-Dokument aus mehreren Tabellenblättern bestehen kann, müssen wir nach dem Laden das Blatt auswählen, das wir verwenden wollen. Am sichersten geht das über den Namen des Tabellenblatts, das in unserem Fall "Darth Vader" heißt:

```
sheet = workbook["Darth Vader"]
```

4.5.2 Zeile für Zeile die Pixelwerte extrahieren

Um aus der Excel-Darstellung zu einer Liste mit 0 und 1 zu kommen, können wir ein Programm schreiben, das genau das Vorgehen simuliert, das wir per Hand machen würden: Wir gehen Zeile für Zeile durch die Tabelle und lesen die Farbe jeder Zelle aus. Wenn die Zelle schwarz ist, fügen wir eine 1 zur Liste hinzu, wenn sie weiß ist, eine 0.

Das Programmierkonzept, das wir dazu verwenden, haben wir schon kennengelernt. Erinnert ihr euch? Wir wollen etwas mehrfach ausführen (prüfen ob die Zelle schwarz oder weiß ist) für eine bekannte Menge von Elementen (alle Zellen in der Tabelle). Für die mehrfache Ausführung desselben Codes verwenden wir Schleifen. Und wenn die Menge der Elemente bekannt ist, verwenden wir eine `for`-Schleife.

Einer `for`-Schleife müssen wir mitteilen, für welche Menge an Elementen sie den selben Code mehrmals ausführen soll. Als wir die `for`-Schleife eingeführt haben, hat diese Aufgabe die `range()`-Funktion erledigt. Sie hat uns eine Sequenz (oder Menge) an Zahlen erzeugt, über die wir iterieren können.

Diese Aufgabe übernimmt jetzt die Funktion `iter_rows()`, die die `openpyxl`-Klasse für das Tabellenblatt uns bereitstellt. Die Funktion gibt uns für ein Tabellenblatt die Menge aller Zeilen in der Tabelle zurück. Also erneut eine Menge an Elementen, wie von der `for`-Schleife gewünscht, wenn auch dieses Mal keine Zahlen. Für jede Zeile in dieser Menge können wir dann wiederum über die einzelnen Zellen iterieren, einfach indem wir `for cell in row:` schreiben.

Es braucht also zwei Schleifen, die ineinander verschachtelt sind: Die erste Schleife geht Zeile für Zeile vor, und die zweite geht für jede Zeile die Spalten (oder Zellen) durch:

```
for row in sheet.iter_rows():
    for cell in row:
        ...
```

①
②
③

- ① Die äußere Schleife iteriert über jede Zeile im Tabellenblatt.
- ② Die innere Schleife iteriert über jede Zelle in der aktuellen Zeile. Die Variable `row` ist eine Liste von Zellen, und `cell` repräsentiert die aktuelle Zelle in dieser Liste.
- ③ Hier müssen wir gleich den Code ergänzen, der die Farbe der Zelle prüft und die entsprechende 0 oder 1 zur Liste hinzufügt.

Was passiert nun für jede Zelle in der inneren Schleife? Wir müssen die Farbe der Zelle auslesen. Wir können auf die Füllfarbe einer Zelle über das `cell.fill`-Attribut zugreifen. Die Farbe selbst ist im `fgColor`-Attribut gespeichert, und der RGB-Wert der Farbe ist im `rgb`-Attribut zu finden. Etwas kompliziert, aber so funktioniert es nun mal. Man kann das alles in der Dokumentation der Bibliothek [nachlesen](#). Auch ChatGPT & Co. helfen hier weiter. Auswendig wissen muss man das nicht.

Probieren wir, die Farbe der Zelle auszulesen:

```
for row in sheet.iter_rows():
    for cell in row:
        color = getattr(cell.fill.fgColor, "rgb", None)
        print(color)
```

①
②

- ① Das müssen wir nicht auswendig wissen, das schlagen wir in der Dokumentation der Bibliothek nach.
- ② Wir geben die Farbe der Zelle aus, um zu sehen, welchen Wert sie hat.

Wenn ihr das Programm bis hierher ausführt, dann solltet ihr auf der Konsole eine lange Liste von Werten sehen. Aber wie sehen die denn aus und was soll das sein?

```
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
```

00000000
FF000000
FF000000
...
...

4.5.3 Hexadezimale Farbwerte

Im vorigen Kapitel Kapitel 2 haben wir das Binärsystem kennengelernt. Das Binärsystem ist nur eines von vielen Stellenwertsystemen und es verwendet als Basis die 2. Wir haben auch gesehen, dass andere Systeme mit anderen Basen gibt, wie das Oktalsystem zur Basis 8 oder das Dezimalsystem zur Basis 10, das wir im Alltag verwenden. Ein weiteres, in der Informatik wichtiges System, ist das Hexadezimalsystem zur Basis 16. Es verwendet die Ziffern 0 bis 9, und weil es dann keine weiteren Zahlen mehr gibt, anschließend die Buchstaben A bis F, um die Werte von 10 bis 15 darzustellen. Hexadezimale Zahlen werden häufig in der Informatik verwendet, um Bytes kompakt darzustellen. Wie ihr gleich sehen werdet, passt ein Byte nämlich wunderschön in zwei hexadezimale Ziffern.

Aber eins nach dem anderen. Beginnen wir mit dem Hexadezimalsystem an sich. Wir können das bekannte Schema der Stellenwertsysteme anwenden, um zu verstehen, welche Wertigkeit jede Ziffer in einer Hexadezimalzahl hat:

1	A	D
16 ²	16 ¹	16 ⁰
$= 1 \times 16^2 + 10 \times 16^1 + 13 \times 16^0$		
$= 1 \times 256 + 10 \times 16 + 13 \times 1$		
$= 429$ (decimal)		

Abbildung 4.14: Das Stellenwertschema für das Hexadezimalsystem zur Basis 16. Funktioniert wie jedes andere Stellenwertsystem.

In Abbildung 4.14 sehen wir die Wertigkeit der einzelnen Stellen in einer Hexadezimalzahl. Die rechte Stelle hat den Wert $16^0 = 1$, die nächste Stelle links davon hat den Wert $16^1 = 16$, dann $16^2 = 256$ und so weiter. Um den Gesamtwert einer Hexadezimalzahl zu berechnen, multiplizieren wir jede Ziffer mit ihrer jeweiligen Wertigkeit und addieren die Ergebnisse. Der einzige Unterschied zu den Stellenwertsystemen, die wir bisher kennengelernt haben, ist die Verwendung von Buchstaben für die Werte 10 bis 15. Da wir keine arabischen Ziffern für diese Werte haben, es aber ein Symbol für die 10, 11, 12, 13, 14 und 15 geben muss, verwenden wir die Buchstaben A, B, C, D, E und F. Ganz einfach, wenn auch gewöhnungsbedürftig.

Warum ist das Hexadezimalsystem überhaupt interessant? Wie schon erwähnt, können wir über zwei Hexadezimalziffern ein Byte kompakt darstellen. Um zu verstehen, warum das so ist, schauen wir uns eine Hexadezimalzahl mit nur einer Ziffer genauer an. Mit einer Ziffer können wir Werte von 0 bis 15 darstellen. Im Binärsystem benötigen wir dafür ganze vier Ziffern, also 4 Bits. Ihr erinnert euch an die Darstellung eines Bytes aus Kapitel 2:

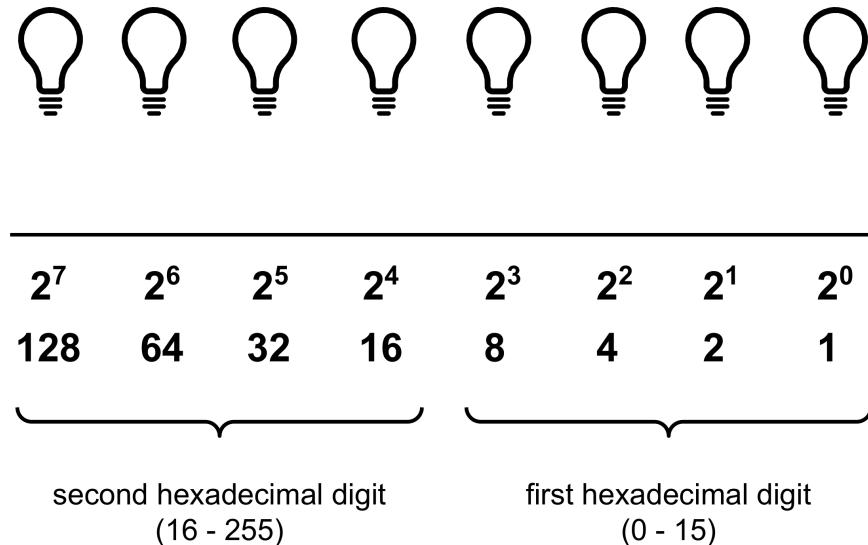
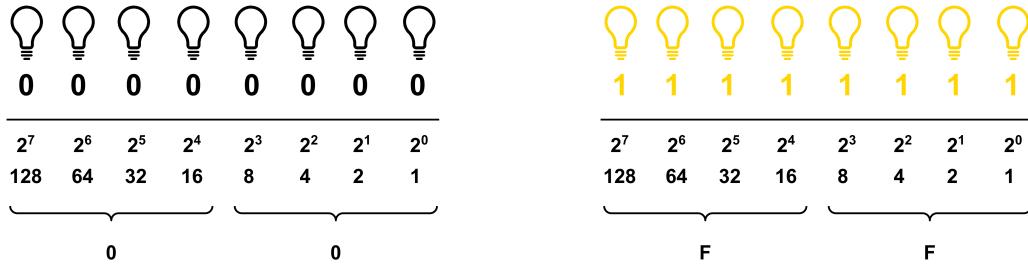


Abbildung 4.15: Ein Byte lässt sich schön kompakt mit zwei Hexadezimalziffern darstellen.

Mit vier Bits können wir ebenfalls Werte von 0 bis 15 darstellen, also genauso viele wie mit einer Hexadezimalziffer. Das bedeutet wir können die erste Hälfte eines Bytes, was wir ein Nibble nennen, mit einer Hexadezimalziffer darstellen. Vergewissern wir uns nun, für welche Wertigkeit die zweite Ziffer im Hexadezimalsystem steht und welchen Wertebereich wir damit darstellen können: Die zweite Stelle steht für $16^1 = 16$. Mit dieser Stelle können wir also Werte von 0 bis 15 mal 16 darstellen, also von 0 bis 240 in Schritten von 16. Kombinieren wir nun beide Stellen, also die erste und die zweite Ziffer, dann können wir Werte von 0 bis

15 (erste Ziffer) plus 0 bis 240 (zweite Ziffer) darstellen. Das ergibt einen Gesamtwertebereich von 0 bis 255, also genau den Wertebereich, den ein Byte abdeckt.



(a) 0 entspricht dem Hexadezimalwert 0x00. (b) 255 entspricht dem Hexadezimalwert 0xFF.

Abbildung 4.16: Darstellung von Bytes als Hexadezimalzahlen.

Kleine Randnotiz: In der Informatik wird eine Hexadezimalzahl häufig mit einem vorangestellten **0x** gekennzeichnet, um klarzumachen, dass es sich um eine Hexadezimalzahl handelt. So wird aus der Zahl 255 im Dezimalsystem die Zahl **0xFF** im Hexadezimalsystem.

Alpha-Werte

Zurück zu der merkwürdigen Ausgabe von oben. Was auf dem Terminal gesehen haben sind also Hexadezimalzahlen, die jeweils den Farbwert einer Zelle in der Excel-Tabelle repräsentieren. Aber Moment: Wir haben gelernt, dass ein RGB-Wert aus drei Bytes besteht, je ein Byte für einen der Farbkanäle Rot, Grün und Blau. Wenn wir ein Byte mit zwei Hexadezimalziffern darstellen können, warum sehen wir dann in der Ausgabe Hexadezimalzahlen mit 8 Ziffern?

Die Antwort ist einfach: Die ersten zwei Ziffern repräsentieren die Transparenz (Alpha-Kanal), gefolgt von den von uns erwarteten sechs Ziffern für Rot, Grün und Blau. In unserem Fall sind die Farben entweder komplett schwarz (**FF000000**) oder komplett weiß (**FFFFFF**). Hätten wir in unsere Excel-Tabelle einen Zellenhintergrund mit Transparenz eingefügt, dann würde sich der Alpha-Wert entsprechend ändern. Er kann also ebenfalls Werte von 00 (vollständig transparent) bis FF (vollständig undurchsichtig) annehmen und damit 256 mögliche Transparenzstufen repräsentieren.

Macht doch mal den Test und färbt die obere linke Zelle in Rot ein und lässt das Programm erneut laufen. Achtung: Speichert die Excel-Datei und schließt sie, bevor ihr das Python-Programm erneut ausführt, sonst kann `openpyxl` die Datei nicht lesen:

```
FFFF0000
00000000
00000000
```

00000000

Die erste Hexadezimalzahl FFFF0000 repräsentiert jetzt die Farbe Rot (oder welche Farbe ihr in Excel als Zellenhintergrund gewählt habt): FF für den Alpha-Kanal (vollständig sichtbar), FF für den Rot-Kanal (volle Intensität), 00 für den Grün-Kanal (keine Intensität) und 00 für den Blau-Kanal (keine Intensität). Die anderen Zellen sind weiterhin schwarz.

4.5.4 Die Liste mit Bits erstellen

Damit haken wir das Thema der Hexadezimalzahlen ab, aber es war notwendig, um die Farbwerte in der Excel-Tabelle zu verstehen. Jetzt können wir nämlich die Farbe jeder Zelle prüfen und entsprechend eine 0 oder 1 zur Liste hinzufügen. Wenn die Farbe FF000000 (schwarz) ist, fügen wir eine 1 hinzu, andernfalls eine 0:

```
from openpyxl import load_workbook

workbook = load_workbook("Darth Vader Pixel Art.xlsx")
sheet = workbook["Darth Vader"]

bits = []
for row in sheet.iter_rows():
    for cell in row:
        color = getattr(cell.fill.fgColor, "rgb", None)
        if color == "FF000000":
            bits.append(1)
        else:
            bits.append(0)

print(f"Bitmap with {len(bits)} bits: {bits}")
```

Die Aufgabe, schwarz von weiß zu unterscheiden, erledigt hier die `if`-Anweisung. Je nachdem, ob die Bedingung `color == "FF000000"` wahr oder falsch ist, wird entweder eine 1 oder eine 0 zur Liste `bits` hinzugefügt. Das Hinzufügen eines Elements an das Ende einer Liste funktioniert mit der `append()`-Methode, der wir das neue Element direkt übergeben. Am Ende haben wir eine Liste mit 648 Einträgen, die die Pixelwerte von Darth Vaders Maske repräsentieren. Jetzt bleibt nur noch die Frage, wie wir Darth Vader auf das Display bekommen.

4.5.5 Anzeige auf dem Display

Wie wir einzelne Pixel auf dem Display anzeigen können, haben wir bereits weiter oben in Abschnitt 4.2 gelernt. Dieses Vorgehen können wir jetzt nutzen und unsere Liste mit Bits in

der Variable `bits` an die `write_pixels()`-Funktion übergeben, um Darth Vaders Maske auf dem Display anzuzeigen. Wir müssen nur noch die Koordinaten des oberen linken und unteren rechten Punkts angeben. Dazu müssen wir entscheiden, wo wir das Bild auf dem Display platzieren wollen. Ich habe mich für die Position (50, 20) als oberen linken Punkt entschieden, das ist ungefähr die Mitte. Der untere rechte Punkt ergibt sich aus der Breite und Höhe des Bildes. Da das Bild 27 Pixel breit und 24 Pixel hoch ist, setzen wir den unteren rechten Punkt auf (76, 43):

```
oled.write_pixels(50, 20, 76, 43, bits)
```

Natürlich müssen wir zuvor den Boilerplate-Code zum Initialisieren des Displays in unser Programm kopieren. Das komplette Programm sieht dann so aus:

```
from openpyxl import load_workbook
from tinkerforge.ip_connection import IPConnection
from tinkerforge.bricklet_oled_128x64_v2 import BrickletOLED128x64V2

ipcon = IPConnection()
ipcon.connect('localhost', 4223)
oled = BrickletOLED128x64V2('25zo', ipcon)
oled.clear_display()

workbook = load_workbook("xlsx/Darth Vader Pixel Art.xlsx")
sheet = workbook["Darth Vader"]

bits = []
for row in sheet.iter_rows():
    for cell in row:
        color = getattr(cell.fill.fgColor, "rgb", None)
        if color == "FF000000":
            bits.append(1)
        else:
            bits.append(0)

print(f"Bitmap with {len(bits)} bits: {bits}")
oled.write_pixels(50, 20, 76, 43, bits)
```

Und Voilà! Darth Vader erscheint auf dem Display!

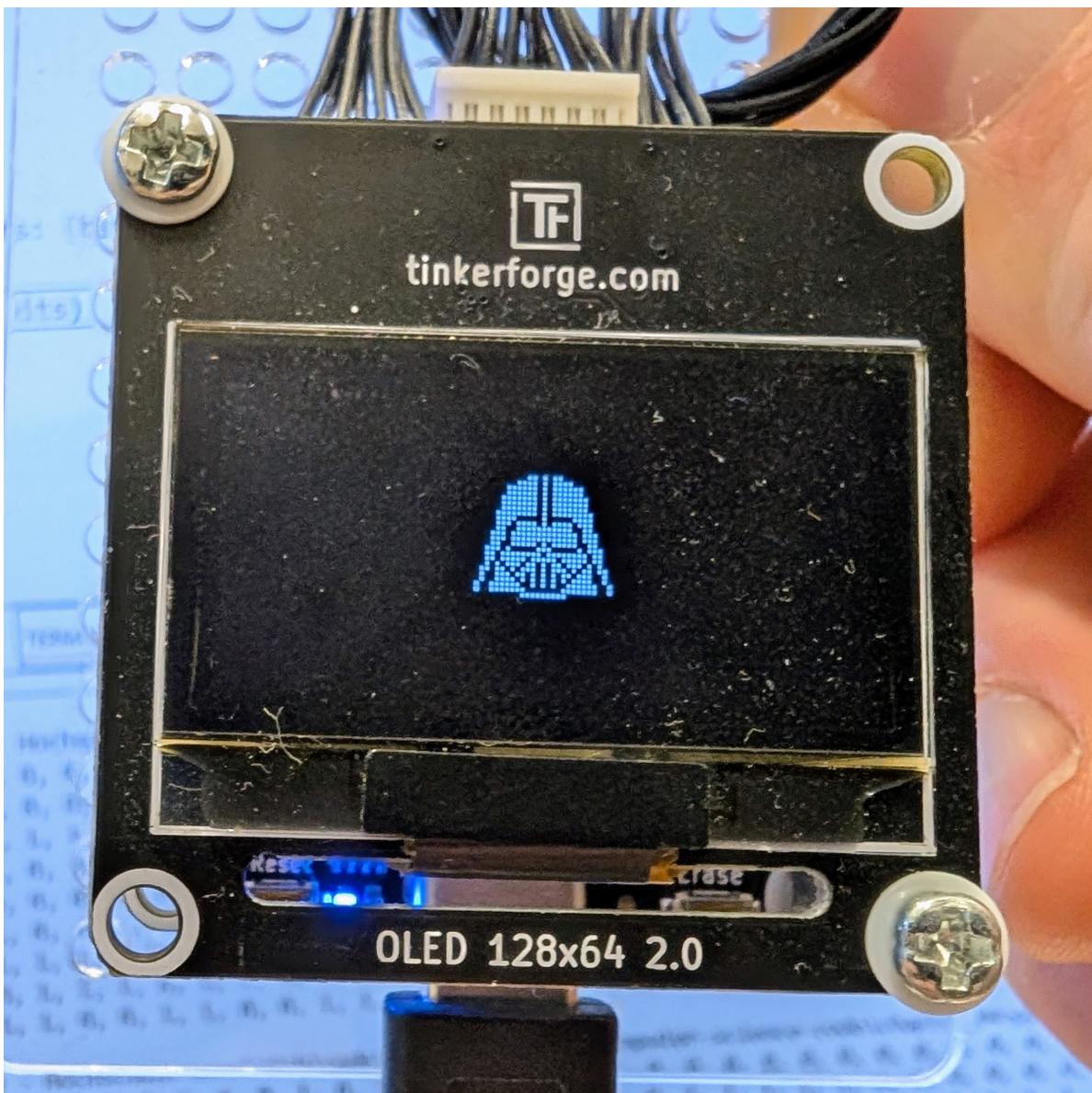


Abbildung 4.17: Darth Vaders Maske als Bitmap aus dem OLED-Display.

4.5.6 Eine Bitmap speichern

Das hat gut geklappt! Wir haben ein Programm geschrieben, das eine Bitmap aus Excel ausliest und auf dem Display anzeigt. Aber ist Excel wirklich ein geeignetes Format für die Speicherung von Bildern?

Sicher nicht, auch wenn es für den Zweck des Verstehens gut funktioniert. Für Bilder, die als Pixelmatrix gespeichert werden, gibt es bessere Alternativen, wie das Bildformat BMP (Bitmap). Es ist ein einfaches, unkomprimiertes Format, das die Pixelwerte direkt als Abfolge von Binärziffern speichert. Dieses Format erhält man auch, wenn an ein Foto mit einer digitalen Kamera schießt: Es wird jeder Farbwert vom Fotosensor gelesen und als Pixelwert gespeichert. Nicht als Excel-Datei, sondern als Bitmap-Datei mit der Endung `.bmp`.

Lasst uns versuchen, unser Bild auch als Bitmap zu speichern. Wenn wir das händisch machen wollten, müssten wir uns mit dem Aufbau des BMP-Formats beschäftigen. Denn eine BMP-Datei besteht nicht nur aus den Pixelwerten, sondern enthält zusätzlich einen Header mit Informationen über das Bild, wie die Breite, Höhe und Farbtiefe.

Wie ihr schon ahnt, müssen wir das Rad auch hier nicht neu erfinden. Es gibt Bibliotheken, die uns die Arbeit abnehmen. Eine der bekanntesten ist die Bibliothek `Pillow`, die vielseitige Möglichkeiten für die Arbeit mit Bildern in Python mitbringt. Mit `Pillow` können wir ganz einfach Bilder laden, bearbeiten und speichern. Installiert die Bibliothek, falls noch nicht geschehen:

```
pip install Pillow
```

MacOS-Nutzer verwenden wieder `pip3` anstelle von `pip`.

Um unser Bild als Bitmap zu speichern, können wir die `Image`-Klasse aus der PIL-Bibliothek verwenden, die wir am Anfang einmal ankündigen müssen. Am Ende, wenn wir die Bitmap aus Excel eingelesen haben, erstellen wir ein neues Bildobjekt mit der gewünschten Größe von 27x24 Pixel und dem Modus “1”, der steht für Schwarzweiß-Bilder. Dann setzen wir die Pixelwerte entsprechend unserer Liste `bits` und speichern das Bild als BMP-Datei:

```
from PIL import Image  
...  
image = Image.new('1', (27, 24))  
image.putdata(bits)  
image.save("xlsx/darth_vader.bmp")
```

- ① Wir importieren die `Image`-Klasse aus der PIL-Bibliothek, die Teil von `Pillow` ist.
- ② Wir erstellen ein neues Bildobjekt mit der Größe 27x24 Pixel im Modus “1” für Schwarzweiß-Bilder. Modus “RGB” wäre für Farbbilder im RGB-Code.

- ③ Wir setzen die Pixelwerte des Bildes mit unserer Liste `bits`. Die Werte werden zeilenweise von links nach rechts und von oben nach unten erwartet.
- ④ Wir speichern das Bild als BMP-Datei mit dem Namen `darth_vader.bmp` im Unterverzeichnis `xlsx`.

Schaut jetzt einmal in das Verzeichnis `xlsx`. Dort solltet ihr die Datei `darth_vader.bmp` finden. Öffnet sie mit einem Bildbetrachter eurer Wahl, und ihr solltet Darth Vaders Maske sehen.

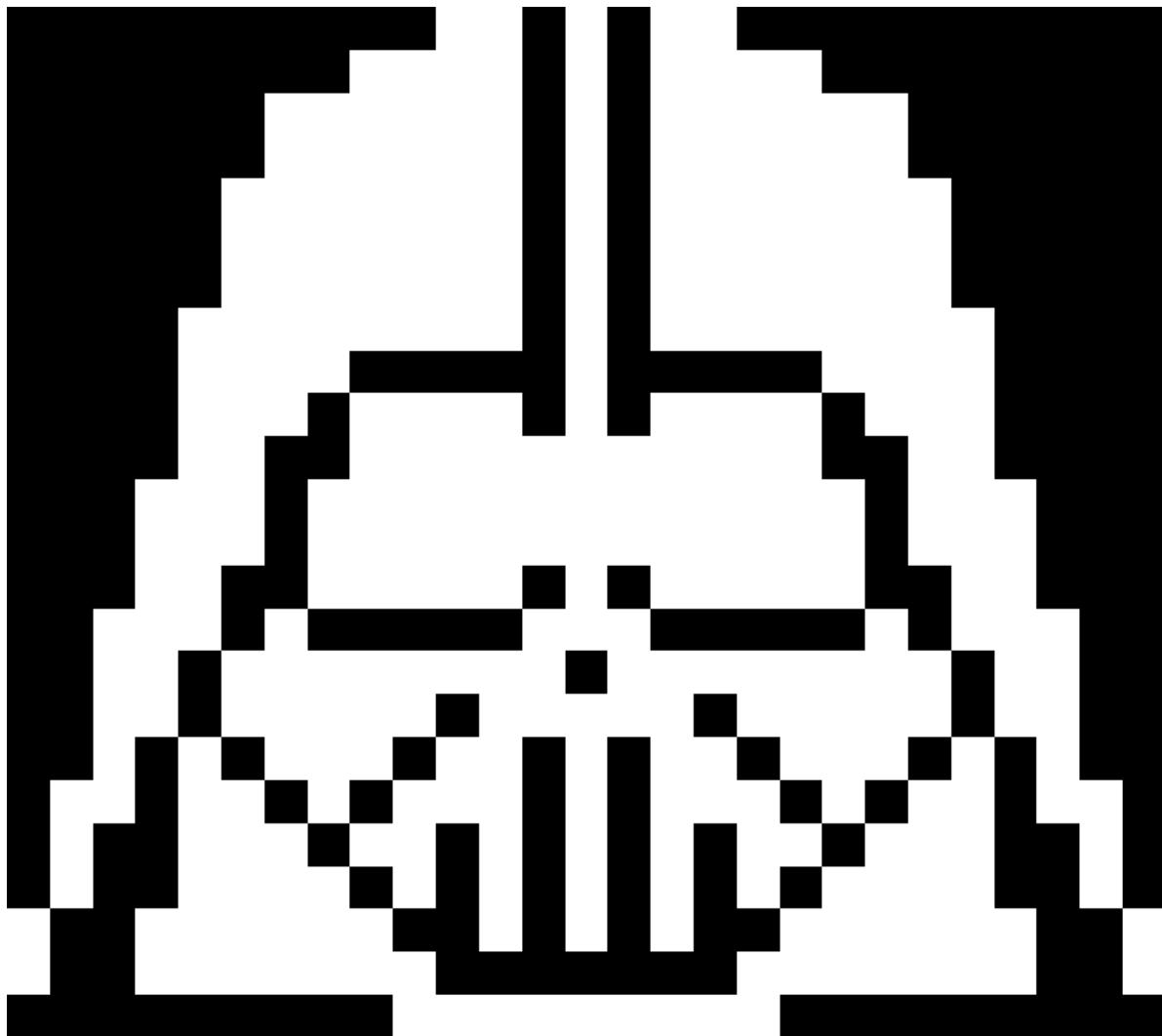


Abbildung 4.18: Darth Vader als Bitmap-Datei auf eurer Festplatte.

Bestimmt ist euch auffallen, dass wir mit dem Display nur Bilder im Schwarz-Weiß-Format anzeigen können. Darth Vader ist daher ein gutes Beispiel. Um aber zu verstehen, wie Computer Bilder repräsentieren, speichern und verarbeiten, müssen wir uns auch mit Farbbildern

beschäftigen. Damit beginnen wir im nächsten Abschnitt und rufen dazu eine berühmte Spielfigur aus der Videospielgeschichte auf den Plan: Super Mario aus den 1980er-Jahren. In Farbe!

4.6 Farbige Bilder

Bisher haben wir uns nur mit Bitmaps beschäftigt, deren Pixel entweder schwarz oder weiß sind. Das hat mit unserem Display zu tun, das eben nicht mehr kann. Trotzdem sollten wir verstehen, wie das Ganze mit farbigen Bildern funktioniert, denn das Prinzip ist dasselbe. Nur dass wir für jedes Pixel nicht nur einen Binärwert (0 oder 1) speichern, sondern einen Farbwert, der aus mehreren Bits besteht. Die Grundlagen dafür, nämlich den RGB-Code, haben wir schon in Kapitel 1 kennengelernt.

4.6.1 Bitmaps im RGB-Format

In Abbildung 4.19 seht ihr eine farbige Bitmap von Super Mario, wie sie in meiner Kindheit häufig über den Bildschirm (damals noch Röhrenfernseher) geflimmert ist. Stundenlang habe ich versucht, den kleinen Klempner durch die Level zu steuern und Prinzessin Peach zu retten. Die Grafik war damals natürlich sehr pixelig, weil die Auflösung der Nintendo Entertainment System (NES) Konsole nur 256 x 240 Pixel betrug. Das hat dem Spielspaß aber keinen Abbruch getan.

Wie unterscheidet sich nun Super Mario in Farbe von Darth Vader, der nur schwarzweiß dargestellt wurde? Nun, zunächst fällt auf, dass die Dimensionen beider Bitmaps unterschiedlich sind. Super Mario ist 16 Pixel breit und 16 Pixel hoch, also insgesamt 256 Pixel. Darth Vader war mit 27x24 Pixeln deutlich größer. Das hat aber nichts mit der Farbe zu tun.

Ein anderer wesentlicher Unterschied ist, dass ein Pixel in Super Marios Bitmap mehr Speicherplatz benötigt, weil es in Farbe dargestellt wird. Jedes Pixel in Super Marios Bitmap wird durch einen RGB-Wert repräsentiert, der aus drei Bytes besteht, je eines für Rot, Grün und Blau. Das bedeutet, dass die gesamte Bitmap von Super Mario $256 \text{ Pixel} \times 3 \text{ Bytes pro Pixel} = 768 \text{ Bytes}$ Speicherplatz benötigt. Im Gegensatz dazu benötigt die Bitmap von Darth Vader nur $648 \text{ Pixel} \times 1 \text{ Bit pro Pixel} = 81 \text{ Bytes}$ Speicherplatz (da jedes Pixel nur schwarz oder weiß ist). Größeres Bild braucht weniger Speicher, hat dafür aber keine Farben.

Ich habe für dieses Experiment Super Mario ebenfalls als Excel-Tabelle erstellt, genau wie bei Darth Vader. Ihr könnt die Datei [hier herunterladen](#). Um Mario in eine Liste von Bits zu überführen, können wir unser Programm von oben grundsätzlich wiederverwenden, müssen es aber ein wenig anpassen. Insbesondere müssen wir jetzt statt 0 oder 1 die RGB-Werte der Zellen in die Liste einfügen.

Ladet zunächst die Excel-Datei herunter und ersetzt im Programm den Dateinamen und den Namen des Tabellenblatts:

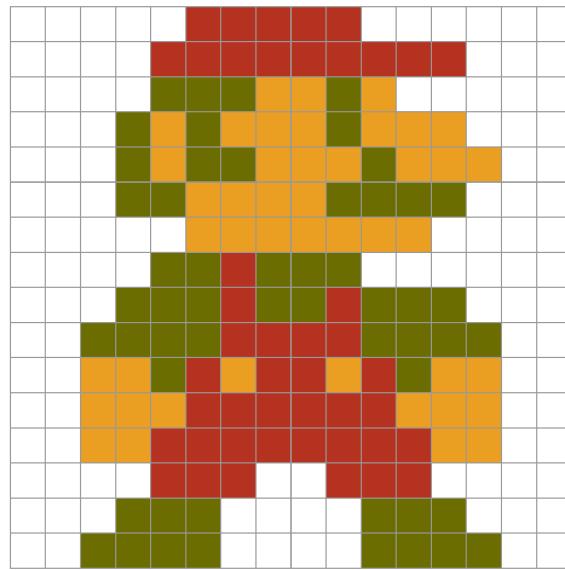


Abbildung 4.19: Super Mario aus der 8-Bit Nintendo-Spielkonsole NES als Bitmap

```
workbook = load_workbook("Super Mario Pixel Art.xlsx")
sheet = workbook["Super Mario"]
```

Es folgen die beiden verschachtelten Schleifen, um jede Zelle auszulesen. Hier müssen wir die Anpassung vornehmen. Gehen wir wie immer Schritt für Schritt vor. Wie wir bereits wissen, kommt der Farbcode zusammen mit dem Alpha-Kanal als Hexadezimalzahl mit 8 Ziffern daher. Für uns sind aber nur die letzten 6 Ziffern interessant. Schneiden wir also die ersten beiden Ziffern ab:

```
bitmap = []
for row in sheet.iter_rows():
    for cell in row:
        color = getattr(cell.fill.fgColor, "rgb", None)
        color = color[2:]                                ①
        print(color)                                     ②
```

- ① Wir schneiden die ersten beiden Ziffern ab, um nur den RGB-Teil der Farbe zu erhalten.
- ② Wir geben den RGB-Farbcode in der Konsole aus, um zu überprüfen, ob wir die richtigen Werte extrahiert haben.

Lasst es einmal laufen und prüft, dass wirklich 6-stellige RGB-Werte in Hexadezimalform ausgegeben werden:

```
FFFFFF
FFFFFF
FFFFFF
FFFFFF
FFFFFF
B53120
B53120
B53120
B53120
B53120
...
...
```

Wenn es bei euch auch so aussieht, dann ist soweit alles gut. Für die Speicherung im RGB-Format benötigen wir einzelne Bytes, was bedeutet, dass wir den RGB-Code in seine drei Bestandteile zerlegen müssen. Dazu erwartet die `putdat()`-Methode der `Image`-Klasse eine Liste von Dreiertupeln jeweils als Dezimalwerte. Also beispielsweise für die ersten drei Pixel, die alle weiß sind:

```
bitmap = [(255, 255, 255), (255, 255, 255), (255, 255, 255)]
```

Wir müssen also zwei Schritte in unser Programm integrieren: Erstens das Zerlegen des RGB-Codes in seine Einzelteile (`#FFFFFF` wird zu `FF, FF, FF`) und zweitens die Umwandlung der Hexadezimalwerte in Dezimalwerte (`FF` wird zu `255`). Beides können wir mit Python-Funktionen erledigen. Um den RGB-Code zu zerlegen, können wir die String-Slicing-Methode verwenden. Dabei können wir angeben, welchen Teilstring wir aus einer Zeichenkette extrahieren möchten. Für die Umwandlung von Hexadezimal- in Dezimalwerte gibt es die eingebaute Funktion `int()`, der wir den Hexadezimalstring und die Basis 16 übergeben können und sie spuckt die entsprechende Dezimalzahl aus. Das Ganze sieht dann so aus:

```
...
color = getattr(cell.fill.fgColor, "rgb", None)
color = color[2:]
r = int(color[0:2], 16)                                ①
g = int(color[2:4], 16)                                ②
b = int(color[4:6], 16)                                ③
print(r, g, b)                                         ④
```

- ① Wir extrahieren die ersten beiden Ziffern für den Rot-Kanal und wandeln sie mit `int()` in eine Dezimalzahl um.

- ② Wir extrahieren die mittleren beiden Ziffern für den Grün-Kanal und wandeln sie mit `int()` in eine Dezimalzahl um.
- ③ Wir extrahieren die letzten beiden Ziffern für den Blau-Kanal und wandeln sie mit `int()` in eine Dezimalzahl um.
- ④ Wir geben die Dezimalwerte für Rot, Grün und Blau in der Konsole aus, um zu überprüfen, ob die Umwandlung korrekt ist.

Die Ausgabe sollte jetzt so aussehen:

```
255 255 255
255 255 255
255 255 255
255 255 255
255 255 255
181 49 32
181 49 32
...

```

Die ersten fünf Pixel sind weiß, gefolgt von den ersten rötlichen Pixeln von Marios Mütze. Passt also! Nun noch die RGB-Werte als Dreiertupel zur Liste `bitmap` hinzufügen:

```
rgb_tuple = (r, g, b)                                ①
bitmap.append(rgb_tuple)                            ②
```

- ① Wir erstellen ein Dreiertupel mit den RGB-Werten.
- ② Wir fügen das Dreiertupel mit den RGB-Werten zur Liste `bitmap` hinzu.

Ein Tupel in Python ist sehr ähnlich wie eine Liste, nur dass wir eine Liste verändern können, ein Tupel nicht. Anstelle der eckigen Klammern [] für Listen verwenden wir für Tupel runde Klammern (). Ansonsten funktioniert das Hinzufügen eines Tupels zu einer Liste genauso wie zuvor mit der `append()`-Methode.

Hier nun der gesamte Code bis hierher (ohne Boilerplate-Code zum Initialisieren des Displays und das Öffnen des Excel-Dokuments):

```
bitmap = []
for row in sheet.iter_rows():
    for cell in row:
        color = getattr(cell.fill.fgColor, "rgb", None)
        color = color[2:]
        r = int(color[0:2], 16)
        g = int(color[2:4], 16)
        b = int(color[4:6], 16)
```

```
rgb_tuple = (r, g, b)  
bitmap.append(rgb_tuple)
```

(4)

(5)

- ① Wir lesen die Farbe der Zelle aus.
- ② Wir schneiden die ersten beiden Ziffern ab, um nur den RGB-Teil der Farbe zu erhalten, ohne den Alpha-Wert.
- ③ Wir extrahieren die Rot-, Grün- und Blau-Komponenten und wandeln sie in Dezimalzahlen um.
- ④ Wir erstellen ein Dreiertupel mit den RGB-Werten.
- ⑤ Wir fügen das Dreiertupel mit den RGB-Werten zur Liste `bitmap` hinzu.

Mit den RGB-Werten in der Liste `bitmap` können wir jetzt ein farbiges Bild von Super Mario erstellen und speichern. Dazu verwenden wir wieder die `Image`-Klasse aus der PIL-Bibliothek. Diesmal erstellen wir ein Bild im Modus “RGB”, der Dimensionierung 16x16 und übergeben wieder die Liste `bitmap` an die `putdata()`-Methode. Dadurch, dass wir den Modus auf “RGB” gesetzt haben, erwartet die Methode jetzt eine Liste von Dreiertupeln mit den RGB-Werten. Zum Schluss speichern wir das Bild als BMP-Datei:

```
image = Image.new('RGB', (16, 16))  
image.putdata(bitmap)  
image.save("xlsx/super_mario_color.bmp")
```

Öffnet jetzt einen Dateiexplorer und navigiert in das Verzeichnis, in dem ihr die Daten gespeichert habt. Ihr könnt die Datei öffnen und solltet Super Mario in Farbe sehen. Zum Abschluss dieses Abschnitts möchte ich aber mit euch einmal auf die Größe der Datei schauen.

4.6.2 Struktur einer Bitmap-Datei

Um die Größe der Datei anzuzeigen könnt ihr entweder im Datei-Explorer auf die Datei klicken. Unten in der Fuszeile sollte die genaue Größe in Bytes angegeben werden. Alternativ geht es das auch über das Terminal in Visual Studio Code. Wechselt dazu mit `cd` in das richtige Verzeichnis und gebt den Befehl `dir` (Windows) oder `ls -lh` (MacOS, Linux) ein. In der Ausgabe sollte die Datei und ihre Größe angezeigt werden:

```
28.10.2025 19:41      822 super_mario_color.bmp
```

Die Zahl von Interesse ist hier unmittelbar vor dem Dateinamen. In meinem Fall sind es 822 Bytes. Rein rechnerisch müssten wir aber auf:

$$16 \cdot 16 \cdot 3 = 768$$

Bytes kommen. Schließlich haben wir $16 \times 16 = 256$ Pixel und jedes Pixel benötigt 3 Bytes Speicherplatz. Warum ist die Datei also größer als erwartet?

Die Differenz ist 54 Bytes. Die Erklärung dafür ist, dass eine Bitmap-Datei (und so gut wie jede andere Datei auch), neben den eigentlichen Informationen (hier: Pixelwerte im RGB-Format) auch noch sogenannte Metainformationen enthält. Metainformationen sind Informationen über ein Objekt, hier die Datei oder das Bild. In einer Bitmap-Datei sind das zum Beispiel Informationen über die Breite und Höhe des Bildes oder die Farbtiefe (also wie viele Bits pro Pixel verwendet werden). All diese Informationen werden im sogenannten Header der Bitmap-Datei gespeichert, der am Anfang steht. Der Header ist notwendig, damit ein Programm, das die Bitmap-Datei öffnet, weiß, wie es die Pixelwerte interpretieren soll, wo die Pixelwerte anfangen und enden, und so weiter.

Die wichtigsten Strukturelemente einer Bitmap-Datei sind in Abbildung 4.20 dargestellt. Wir sehen hier die Bitmap-Datei in einem sogenannten Hexadezimal-Editor, mehr dazu weiter unten. Der kleine gelbe Bereich am Anfang ist 14 Bytes lang und stellt den sogenannten Datei-Header dar. Hier wird zum Beispiel ganz zu Beginn mitgeteilt, dass es sich überhaupt um eine Bitmap-Datei handelt (die ersten beiden Bytes 42 und 4D enthalten die ASCII-Zeichen "BM" für Bitmap). Außerdem wird hier die Gesamtgröße der Datei gespeichert (36 03 00 00, was 822 Bytes entspricht; da das Little-Endian Format verwendet wird, sind die Bytes in umgekehrter Reihenfolge gespeichert) und die Position, an der die eigentlichen Pixelwerte beginnen. Das ist die 36 im Bild, was dezimal 54 ist. Die eigentlichen Pixelwert beginnen also mit dem 54. Byte in der Datei, was der hellgrüne Bereich ist.

Direkt danach folgt der größere rosafarbene Bereich, der 40 Bytes lang ist und den sogenannten DIB-Header (Device Independent Bitmap) enthält. In diesem Header sind Metadaten über das Bild gespeichert, wie die Breite, Höhe, Farbtiefe und so weiter. Letztere wird durch die 18 angegeben, was dezimal der 24 entspricht. Das steht für 24 Bits Farbtiefe. Ein Programm, das diese Bitmap-Datei liest, weiß also anhand des DIB-Header, wie es die Pixelwerte lesen muss, um das Bild korrekt darzustellen.

Erst nach den beiden Headern folgen die eigentlichen Pixelwerte, die in Abbildung 4.20 grünlich dargestellt sind und die in unserem Fall 768 Bytes lang sind. Das Bild zeigt nur einen Auschnitt aus der gesamten Datei, es folgen weiter unten noch weitere Pixelwerte. Probiert es am besten einmal selbst, die Datei im Hexcode-Editor anzuschauen. Dann bekommt ihr ein Gefühl dafür, wie die Datei tatsächlich aussieht und auf eurem Rechner gespeichert wird. Schließlich geht es in diesem Buch darum, wie Computer funktionieren, und da gehört es dazu, dass wir verstehen, wie Daten im Computer gespeichert werden. Das Beispiel der Bitmap ist dafür sehr zugänglich und anschaulich.

Öffnet dazu einfach die Webseite hexed.it in eurem Browser. Dahinter verbirgt sich ein Editor für Dateien, wie ihr ihn auch von Word, Notepad oder Visual Studio Code kennt. Der Unterschied ist, dass dieser Editor die Datei im Binärformat anzeigt, also die Nullen und Einsen, die eurer Computer tatsächlich speichert. Damit die Schreibweise etwas kompakter wird, werden die Bits in Bytes gruppiert und als Hexadezimalzahlen dargestellt. Jedes Byte wird also

super_mario_color.bmp *																
00000000	42	4D	36	03	00	00	00	00	00	00	36	00	00	00	28	00
00000010	00	00	10	00	00	00	10	00	00	00	01	00	18	00	00	00
00000020	00	00	00	03	00	00	C4	0E	00	00	C4	0E	00	00	00	00
00000030	00	00	00	00	00	00	FF	FF	FF	FF	FF	FF	00	6D	6B	00
00000040	6D	6B	00	6D	6B	00	6D	6B	FF							
00000050	FF	FF	FF	FF	FF	00	6D	6B	00	6D	6B	00	6D	6B	00	6D
00000060	FF															
00000070	6D	6B	00	6D	6B	00	6D	6B	FF							
00000080	FF	FF	FF	FF	FF	00	6D	6B	00	6D	6B	00	6D	6B	FF	FF
00000090	FF															
000000A0	FF	FF	20	31	B5	20	31	B5	20	31	B5	FF	FF	FF	FF	FF
000000B0	FF	20	31	B5	20	31	B5	20	31	B5	FF	FF	FF	FF	FF	FF
000000C0	FF	22	9E	EA												
000000D0	9E	EA	20	31	B5	20	31									
000000E0	B5	20	31	B5	20	31	B5	20	31	B5	22	9E	EA	22	9E	EA
000000F0	FF	22	9E	EA												
00000100	9E	EA	22	9E	EA	20	31	B5	20	31	B5	20	31	B5	20	31
00000110	B5	20	31	B5	20	31	B5	22	9E	EA	22	9E	EA	22	9E	EA
00000120	FF	22	9E	EA												

Abbildung 4.20: Die Struktur einer Bitmap-Datei am Beispiel der super_mario_color.bmp von oben

als zwei Hexadezimalziffern angezeigt. Wenn ihr die super_mario_color.bmp-Datei im Editor öffnet, solltet ihr eine ähnliche Ansicht wie in Abbildung 4.21 sehen. Dort sind die Farbwerte der ersten drei Pixel zum besseren Verständnis rot umrandet.

Es passt also im Endeffekt doch alles zusammen: Die Bitmap-Datei hat 14 Bytes für den Datei-Header, 40 Bytes für den DIB-Header und 768 Bytes für die Pixelwerte, was zusammen die 822 Bytes gibt, die wir im Terminal oder Explorer sehen. Wir können alle beruhigt schlafen und haben wieder was gelernt!

4.7 Bewegte Bilder

Viele Anwendungen, wozu auch das Videospiel Super Mario gehört, kommen mit statischen Bildern nicht aus. Mario kann laufen, hüpfen und Feuerkugeln auf seine Gegner schleudern. Auch bei Videosoclips auf YouTube oder Filem auf Netflix stehen bewegte Bilder im Vordergrund. Aber wie funktionieren bewegte Bilder eigentlich?

Das Verständnis, wie statische Bilder in einem Computer gespeichert werden, ist die Grundvoraussetzung, um bewegte Bilder - oder Animationen - zu verstehen. Denn eine Animation besteht im Grunde aus einer schnellen Abfolge von Einzelbildern, die nacheinander angezeigt werden. Wenn die Bilder schnell genug wechseln, dann entsteht der Eindruck von Bewegung.

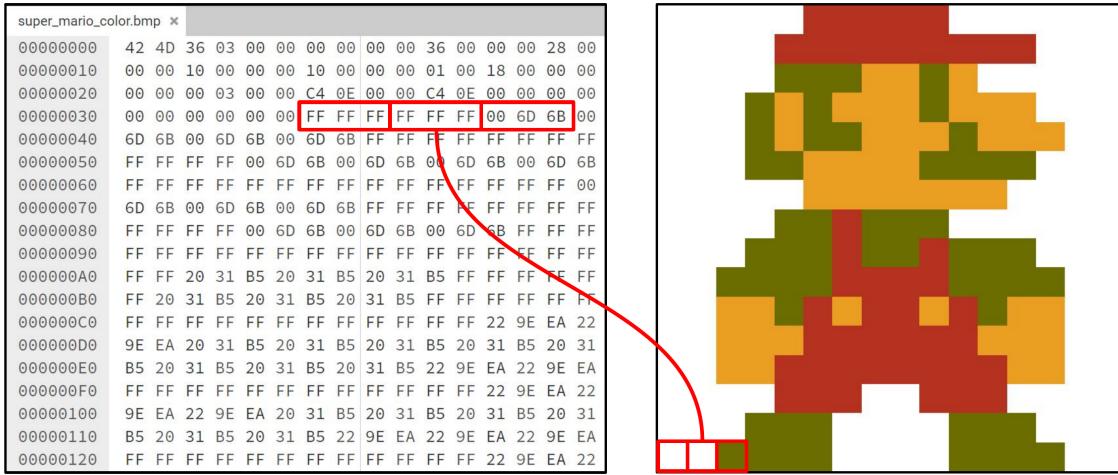
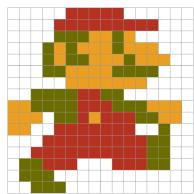


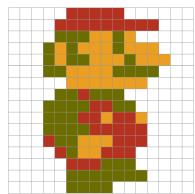
Abbildung 4.21: Marios Bitmap im Hex-Editor. Hier wird jedes Byte als Hexadezimalzahl angezeigt.

Unser Gehirn kann nämlich nicht so schnell unterscheiden, ob es sich um einzelne Bilder handelt oder ob sich etwas bewegt.

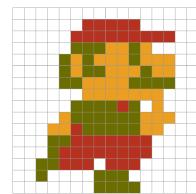
Um auch das einmal Hands-On zu erfahren, versuchen wir in diesem Abschnitt, Mario zum Laufen zu bringen. Dazu benötigen wir nicht nur ein Bild, sondern mehrere Bilder, die nacheinander die unterschiedlichen Posen abbilden, die Mario beim Laufen einnimmt. In Abbildung 4.22 seht ihr drei solcher Bilder. Diese stammen 1:1 aus den Sprites des 1908er Spiels "Super Mario Bros." für die NES-Konsole.



(a) Erste Frame



(b) Zweiter Frame



(c) Dritter Frame

Abbildung 4.22: Drei Bilder - oder Frames - um Mario zum Laufen zu bringen.

Ein Sprite ist ein kleines Bild, das in Videospielen verwendet wird, um Figuren oder Objekte darzustellen. In der Regel sind Sprites Teil einer größeren Grafikdatei, die alle benötigten

Bilder für ein Spiel enthält. Diese Grafikdatei wird als Sprite-Sheet bezeichnet. Jedes Bild im Sprite-Sheet repräsentiert eine bestimmte Pose oder Aktion der Figur. Indem das Spiel schnell zwischen diesen Bildern wechselt, entsteht der Eindruck von Bewegung. Das aber nur am Rande.

Wichtig ist, dass wir jetzt drei Bilder von Mario haben, die wir nacheinander anzeigen können, um den Eindruck zu erwecken, dass Mario läuft. Das fertige Ergebnis sieht ihr in der Animation in `?@fig-images-super-mario-animation`. Es handelt sich hierbei um ein so genannte GIF, was für Graphics Interchange Format steht. Ein GIF ist ein Bildformat, das mehrere Einzelbilder in einer Datei speichern kann. Jedes Einzelbild wird als Frame bezeichnet, und die Frames werden in schneller Abfolge abgespielt, um den Eindruck von Bewegung zu erzeugen. Aufgrund ihrer Einfachheit und der breiten Unterstützung in Webbrowsersn sind GIFs besonders beliebt für kurze Animationen und Memes im Internet. Vielleicht habt ihr selbst schon einmal ein GIF versendet, zum Beispiel über WhatsApp oder anderen Chatprogrammen. Dort sind sie ebenfalls sehr beliebt.

Wir können das selbe Prinzip hinter GIFs nutzen, um eine Animation auf unser kleines OLED-Display zu bringen. Dazu müssen wir Bilder, die zu einer Animation gehören, nacheinander auf dem Display anzeigen und dazwischen jeweils eine kurze Pause einlegen. Wenn wir das schnell genug machen, dann entsteht der Eindruck eines bewegten Bildes. Dazwischen steht noch ein Problem: Das Display kann keine Farben anzeigen.

4.8 Transformationen

Der große Vorteil von digitalen Bildern ist es, dass wir sie mit einfacher Arithmetik fast beliebig bearbeiten können. In unserem Fall müssten wir die Bilder von Mario in eine Schwarzweiß-Darstellung umwandeln, damit wir sie auf dem Display darstellen können. Sicher habt ihr sowas ähnliches schon einmal selbst gemacht: Einen sogenannten Filter auf ein Foto angewendet, das euer Bild in Graustufen verwandelt, oder einer, der die Farben knalliger erscheinen lässt. Filter sind heute gängige Funktionen von Social Media Apps wie Instagram, Snapchat oder TikTok, man braucht dazu kein Photoshop mehr. Aber wie funktioniert das eigentlich? Und was, wenn wir nicht einmal Graustufen, sondern nur Schwarzweiß wollen?

Auf den Punkt gebracht können wir aus einem Farbild mit zwei einfachen Schritten, die reine Arithmetik benötigen, ein Graustufenbild erstellen:

1. Wir transformieren jedes Pixel in einen Grauwert. Dadurch erhalten wir ein Bild, das nur noch aus Grautönen, also Helligkeitswerten, besteht.
2. Wir wandeln die Grautöne in Schwarzweiß um, indem wir einen Schwellenwert festlegen. Alle Grautöne, die heller als der Schwellenwert sind, werden zu Weiß, alle dunkleren Grautöne zu Schwarz.

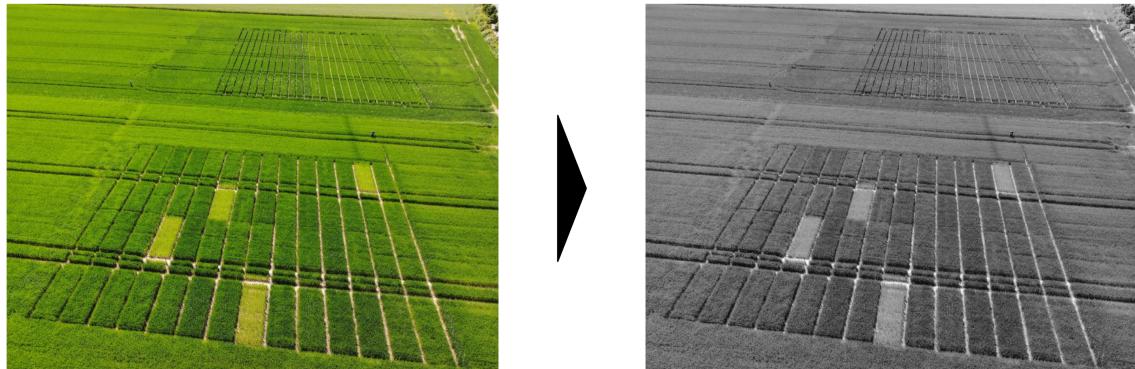
4.8.1 Graustufen

In Abbildung 4.23 seht ihr das Beispiel einer Drohnenaufnahme in Farbe und in Graustufen. Um das Farbbild in ein Graustufenbild zu verwandeln, müssen wir für jedes Pixel einen Grauwert berechnen. Wenn wir uns vor Augen führen, was ein Graustufenbild essentiell ausmacht, nämlich dass es nur Helligkeitswerte enthält, dann liegt die Lösung nahe: Wir müssen aus dem RGB-Wert eines Pixels einen einzigen Helligkeitswert, auch Luminanz genannt, berechnen. Eine einfache Möglichkeit ist es, pro Pixel den Durchschnitt der drei Farbwerte zu bilden:

$$\text{luminance} = \frac{R + G + B}{3}$$

Damit bekommen wir für jedes Pixel einen Wert zwischen 0 und 255, der die Helligkeit des Pixels repräsentiert. 0 steht für schwarz, 255 für weiß, und alle Werte dazwischen für Grautöne. Das wäre eine Möglichkeit, um ein Graustufenbild zu erstellen. Allerdings entspricht diese Methode nicht exakt der menschlichen Wahrnehmung von Helligkeit. Man hat herausgefunden, dass unser Auge empfindlicher ist für Grün- und Rot-Töne als für Blau-Töne. Daher wird häufig eine gewichtete Summe verwendet, um die Luminanz genauer zu berechnen:

$$\text{luminance} = 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B$$



30

Abbildung 4.23: Ein Drohnenbild in Farbe und in Graustufen.

Wir wollen Mario möglichst für Menschen optimieren, deshalb verwenden wir die gewichtete Summe. In Python können wir das so umsetzen:

```
luminance = 0.299 * r + 0.587 * g + 0.114 * b  
luminance = round(luminance)
```

- ① Wir berechnen die Luminanz als gewichtete Summe der RGB-Werte.
- ② Wir wandeln den Luminanzwert durch Rundung mit der `round()`-Funktion in eine Ganzzahl um, da die RGB-Werte Ganzzahlen sein müssen.

Bevor wir ein Pixel von Farbe in Graustufen umwandeln können, müssen wir zuerst die Bitmap-Datei laden und jedes Pixel einzeln auslesen. Das Laden funktioniert ganz einfach mit der `Pillow`-Bibliothek, die wir bereits weiter oben verwendet haben:

```
from PIL import Image  
image = Image.open("super_mario_color.bmp")
```

- ① Mit der `open()`-Methode der `Image`-Klasse laden wir die Bitmap-Datei und erstellen ein Bildobjekt auf der Variable `image`.

Denkt daran den Pfad zu eurem Bild anzupassen, solltet ihr es zum Beispiel in einem Unterordner gespeichert haben. Nach diesen beiden Zeilen Code haben wir Zugriff auf das Bild über die Variable `image`. Darüber können wir für jedes beliebige Pixel mit der `getpixel()`-Methode den Farbwert auslesen. Machen wir den Test und prüfen, ob das Pixel in der linken oberen Ecke wirklich weiß ist:

```
pixel = image.getpixel((0, 0))  
print(pixel)
```

Die Funktion erwartet ein Tupel mit den Koordinaten des Pixels, das wir auslesen wollen. Die linke obere Ecke hat die Koordinaten $(0, 0)$. Die Ausgabe sollte jetzt so aussehen:

`(255, 255, 255)`

Wie es aussieht erhalten wir ein Tupel zurück, das die RGB-Werte des Pixels enthält. Perfekt! Jetzt können wir die Luminanz berechnen und das Pixel in Graustufen umwandeln. Dazu extrahieren wir zunächst die einzelnen Farbwerte aus dem Tupel:

```
pixel = image.getpixel((0, 0))  
r = pixel[0]  
g = pixel[1]  
b = pixel[2]
```

Jetzt können wir die Luminanz berechnen, wie oben beschrieben. Wir geben sie auch gleich mit aus, um es auf der Konsole überprüfen zu können:

```
luminance = 0.299 * r + 0.587 * g + 0.114 * b
luminance = round(luminance)
print(f"R: {r}, G: {g}, B: {b}, Luminance: {luminance}")
```

So sieht es bis hierhin alles zusammen aus:

```
pixel = image.getpixel((0, 0))
r = pixel[0]
g = pixel[1]
b = pixel[2]
luminance = 0.299 * r + 0.587 * g + 0.114 * b
luminance = round(luminance)
print(f"R: {r}, G: {g}, B: {b}, Luminance: {luminance}")
```

Für das Pixel in oben links hat es also geklappt. Schauen wir uns Abbildung 4.19 noch einmal an. Das sechste Pixel in der ersten Reihe müsste einen rotähnlichen Wert haben. Das wollen wir mit unserem kleinen Programm überprüfen. Dazu müssen wir nur die Koordinaten in der `getpixel()`-Methode anpassen, alles andere bleibt gleich:

```
pixel = image.getpixel((5, 0))                                         ①
r = pixel[0]
g = pixel[1]
b = pixel[2]
luminance = 0.299 * r + 0.587 * g + 0.114 * b
luminance = round(luminance)
print(f"R: {r}, G: {g}, B: {b}, Luminance: {luminance}")
```

- ① Wir passen die Koordinaten an, um das sechste Pixel in der ersten Reihe auszulesen. Der erste Wert ist die Spaltennummer (5), der zweite Wert die Zeilennummer (0).

Warum 5 und nicht 6? Weil die Zählung bei 0 anfängt, ihr erinnert euch. Die Ausgabe sollte jetzt so aussehen:

```
R: 181, G: 49, B: 32, Luminance: 87
```

Nun würde ich das gerne auch noch für ein grünliches und ein gelbliches Pixel ausprobieren, möchte aber vorher eine kleine Verbesserung in unserem Code einbauen. Wie ihr vielleicht gemerkt habt, müssen wir für jedes Pixel den gesamten Code für die Berechnung des Graustufenwerts wiederholen. Das ist nicht sehr elegant und führt zu viel doppeltem Code, was wir in

der Programmierung unbedingt vermeiden wollen. Besser wäre es, wenn wir den Code in eine Funktion packen könnten, die wir dann für jedes Pixel aufrufen:

```
def rgb_to_luminance(rgb_tupel):  
    r = rgb_tupel[0]  
    g = rgb_tupel[1]  
    b = rgb_tupel[2]  
  
    luminance = 0.299 * r + 0.587 * g + 0.114 * b  
    luminance = round(luminance)  
    return luminance
```

- ① Mit der `def`-Anweisung definieren wir eine Funktion namens `rgb_to_luminance`, die ein Tupel mit den RGB-Werten als Eingabeparameter erwartet. Alles, was für die Berechnung der Luminanz notwendig ist, wird in den Rumpf der Funktion geschrieben. Am Ende geben wir den berechneten Luminanzwert mit der `return`-Anweisung zurück.

Jetzt können wir diese Funktion für jedes Pixel aufrufen und müssen den Code für die Berechnung der Luminanz nicht mehrmals schreiben:

```
pixel = image.getpixel((4, 0))  
luminance = rgb_to_luminance(pixel)  
print(f'Luminance of pixel (4,0): {luminance}')  
  
pixel = image.getpixel((4, 1))  
luminance = rgb_to_luminance(pixel)  
print(f'Luminance of pixel (4,1): {luminance}')  
  
pixel = image.getpixel((4, 2))  
luminance = rgb_to_luminance(pixel)  
print(f'Luminance of pixel (4,2): {luminance}')  
  
pixel = image.getpixel((4, 3))  
luminance = rgb_to_luminance(pixel)  
print(f'Luminance of pixel (4,3): {luminance}')
```

Die Ausgabe sieht so aus:

```
Luminance of pixel (4,0): 255  
Luminance of pixel (4,1): 87  
Luminance of pixel (4,2): 96  
Luminance of pixel (4,3): 167
```

Klasse! Wir haben jetzt probeweise vier Pixel aus Marios Bitmap betrachtet und die Graustufenwerte berechnet. In Abbildung Abbildung 4.24 sind die Pixel, die wir in unserem Programm im Blick haben, rot umrandet dargestellt.

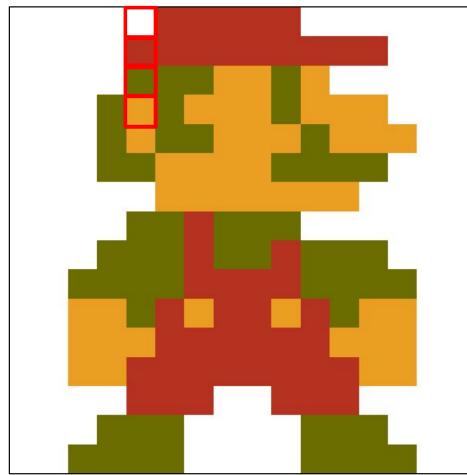


Abbildung 4.24: Vier Pixelproben aus Marios Bitmap bestätigen unser Vorgehen.

Jetzt sind wir soweit und können nicht nur vier Proben, sondern alle Pixel und damit das gesamte Bild in Graustufen umwandeln. Dazu bedienen wir uns wieder eines wichtigen Programmierkonzepts, nämlich... richtig, den Schleifen. Genau gesagt den `for`-Schleifen, mit denen wir bereits weiter oben gearbeitet haben. Wir können nämlich die Breite und Höhe des Bildes mit `image.size` auslesen, was einen Tupel mit zwei Werten zurückgibt. Der erste steht für die Breite und der zweite für die Höhe des Bildes in Pixel. Damit können wir zwei verschachtelte `for`-Schleifen schreiben, eine für die Zeilen und eine für die Spalten, und so über alle Pixelkoordinaten iterieren:

```
w, h = image.size

for y in range(h):
    for x in range(w):
        r, g, b = image.getpixel((x, y))
        luminance = rgb_to_luminance((r, g, b)) ②
        print(f"Pixel at ({x}, {y}): R={r} G={g} B={b} Luminance={luminance}")
```

Vergessen wir nicht das ursprüngliche Ziel: Wir wollen das Graustufenbild erstellen. Dazu

benötigen wir eine neue Liste, in der wir die Graustufenwerte speichern können. Am Ende können wir dann wieder die Pillow-Bibliothek verwenden, um aus der Liste ein neues Bild zu erstellen und es als Bitmap-Datei zu speichern. Sammeln wir zunächst die Graustufenwerte in einer Liste:

```
w, h = image.size
grayscale_values = []
for y in range(h):
    for x in range(w):
        r, g, b = image.getpixel((x, y))
        luminance = rgb_to_luminance((r, g, b))
        grayscale_values.append(luminance)
```

Wenn wir uns die Liste nach Ausführung des Codes anschauen, dann sieht das so aus:

```
Grayscale bitmap with 256 pixel values: [255, 255, 255, 255, ... 96, 96, 96, 96, 255, 255]
```

Unsere Liste hat 256 Werte, was 16×16 Pixeln entspricht. Jeder Wert ist ein Graustufenwert zwischen 0 und 255. Jetzt können wir die Liste verwenden, um ein neues Bild im Graustufenmodus zu erstellen und es als Bitmap-Datei zu speichern:

```
grayscale_image = Image.new("L", (w, h))
grayscale_image.putdata(grayscale_values)
grayscale_image.save("super_mario_grayscale.bmp")
```

Das Ergebnis sieht ihr in Abbildung 4.25. Wir haben es geschafft, Mario in ein Graustufenbild zu verwandeln! Herzlichen Glückwunsch!

Den fertigen Code für die Umwandlung von Farbe in Graustufen seht ihr hier noch einmal komplett:

```
from PIL import Image
image = Image.open("xlsx/super_mario_color.bmp")

def rgb_to_luminance(rgb_tupel):  
    r = rgb_tupel[0]  
    g = rgb_tupel[1]  
    b = rgb_tupel[2]  
  
    luminance = 0.299 * r + 0.587 * g + 0.114 * b  
    luminance = round(luminance)①
```

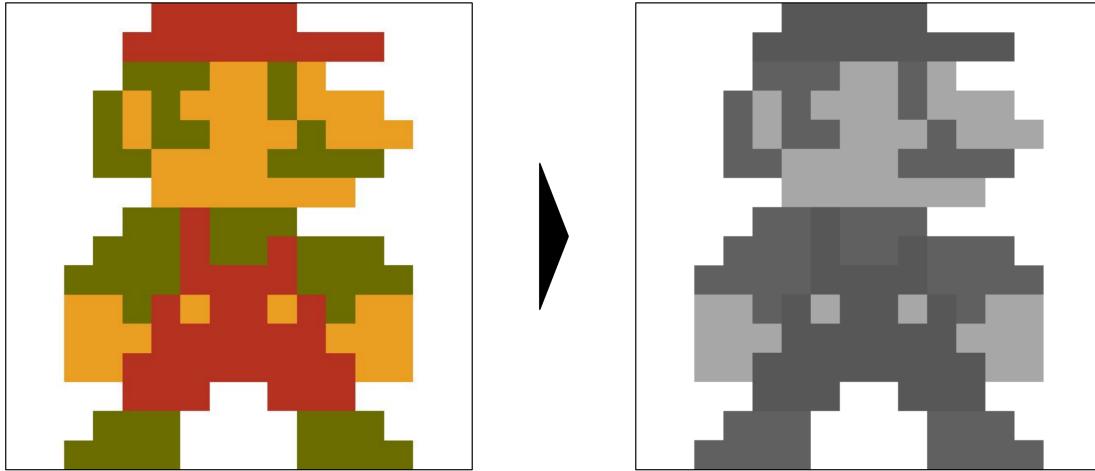


Abbildung 4.25: Mario als Graustufenbild, das unser Programm erzeugt hat.

```

    return luminance

w, h = image.size
grayscale_values = []
for y in range(h):
    for x in range(w):
        r, g, b = image.getpixel((x, y))
        luminance = rgb_to_luminance((r, g, b))
        grayscale_values.append(luminance)

print(f"Grayscale bitmap with {len(grayscale_values)} pixel values: {grayscale_values}")

grayscale_image = Image.new("L", (w, h))
grayscale_image.putdata(grayscale_values)
grayscale_image.save("xlsx/super_mario_grayscale.bmp")

```

4.8.2 Schwarzweiß

Ein Etappenziel ist erreicht, wir müssen aber noch einen Schritt weiter gehen. Unser Display kann nur Schwarzweiß darstellen, also müssen wir das Graustufenbild noch in ein Schwarzweiß-

bild umwandeln. Das funktioniert noch einfacher, als die Umwandlung von Farbe zu Graustufen. Bei Schwarzweiß haben wir nur zwei Möglichkeiten: Entweder schwarz (0) oder weiß (1). Wir müssen also für jeden Graustufenwert entscheiden, ob er zu schwarz oder weiß wird. Dazu ist es gängige Methode, einen Schwellenwert festzulegen, zum Beispiel 128. Alle Graustufenwerte, die kleiner als 128 sind, werden zu schwarz (0), alle Werte größer oder gleich 128 werden zu weiß (1). Da Funktionen mittlerweile zu unserem Repertoire gehören, können wir die Umwandlung direkt als Funktion umsetzen:

```
def luminance_to_bw(luminance, threshold=128):  
    if luminance < threshold:  
        return 0  
    else:  
        return 1
```

①

- ① Wir definieren eine Funktion namens `luminance_to_bw`, die einen Luminanzwert und einen optionalen Schwellenwert als Eingabeparameter erwartet. Standardmäßig ist der Schwellenwert auf 128 gesetzt. Die Funktion gibt 0 zurück, wenn der Luminanzwert kleiner als der Schwellenwert ist, und 1 sonst.

Unsere Funktion bekommt als Eingabe den Luminanzwert sowie den Schwellenwert, um darauf basierend zu entscheiden, ob das Pixel schwarz oder weiß wird. Dabei lernen wir ein weiteres Detail zu Funktionen kennen: Wir können Parameter mit einem Standardwert versehen. In unserem Fall ist der Schwellenwert standardmäßig auf 128 gesetzt. Wenn wir die Funktion also ohne zweiten Parameter aufrufen, wird automatisch 128 verwendet. Wir können den Schwellenwert aber auch explizit angeben, wenn wir möchten. Dieses Vorgehen erspart uns die Übergabe eines Parameters, bei dem in den meisten Fällen sowieso ein Standardwert verwendet wird. Wir haben aber jederzeit die Möglichkeit, den Wert zu überschreiben.

Wir sparen uns an dieser Stelle die Probe einzelner Pixel und wenden die Funktion direkt auf alle Graustufenwerte aus dem Bild an. Dazu laden wir es wieder mit `Pillow` und definieren zwei verschachtelte `for`-Schleifen, alles wie gehabt. Hier das vollständige Programm:

```
from PIL import Image  
image = Image.open("xlsx/super_mario_grayscale.bmp")  
  
def luminance_to_bw(luminance, threshold=128):  
    if luminance < threshold:  
        return 0  
    else:  
        return 1  
  
w, h = image.size  
bw_values = []
```

①

```

for y in range(h):
    for x in range(w):
        grayscale_value = image.getpixel((x, y))
        bw = luminance_to_bw(grayscale_value, 128)
        bw_values.append(bw)

print(f"Black and white bitmap with {len(bw_values)} pixel values: {bw_values}")

bw_image = Image.new("1", (w, h))
bw_image.putdata(bw_values)
bw_image.save("xlsx/super_mario_bw.bmp")

```

Neben der Funktion `luminance_to_bw`, die wir jetzt statt `if` verwenden, machen wir hier nichts Neues. Wir laden das Graustufenbild, wandeln jedes Pixel in Schwarzweiß um und speichern das Ergebnis als Bitmap-Datei. Das kennen wir schon aus Abschnitt 4.5.6. Das fertige Schwarzweißbild seht ihr in Abbildung 4.26.

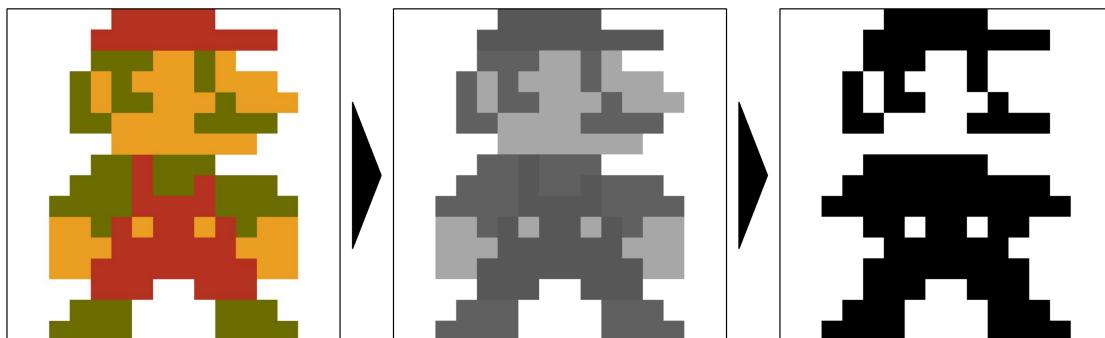


Abbildung 4.26: Mario in Schwarzweiß, was offensichtlich ein großer Informationsverlust ist.

4.8.3 Informationsverlust

Was fällt euch bei der Kaskade von Farbe über Graustufen zu Schwarzweiß auf? Richtig, es geht eine Menge Information verloren. In Farbe haben wir 256 verschiedene Farbtöne pro Kanal, also insgesamt über 16 Millionen Farben. In Graustufen haben wir immerhin noch 256 verschiedene Helligkeitswerte. In Schwarzweiß hingegen nur noch zwei Zustände: schwarz oder weiß.

Das ist ein enormer Informationsverlust, der sich auch im Bild bemerkbar macht. Mario ist zwar am Ende noch erkennbar, aber viele Details sind verloren gegangen. Speziell die Stellen an Marios Körper, an denen man seine Haut sehen sollte, sind jetzt komplett weiß. Das liegt daran, dass die Graustufenwerte dort über dem Schwellenwert von 128 lagen und somit zu weiß umgewandelt wurden. Wir können versuchen, den Schwellenwert anzupassen, um ein besseres Ergebnis zu erzielen. Probiert es gerne selbst aus, indem ihr den Wert in der Funktion `luminance_to_bw` ändert und das Programm erneut ausführt. Es darf aber bezweifelt werden, dass wir mit nur zwei Farben ein wirklich gutes Ergebnis erzielen können.

Eine wichtige Erkenntnis in Bezug auf Bildern können wir an dieser Stelle festhalten: Die Anzahl an Informationen, die unser Bild enthält, bestimmt maßgeblich die Qualität des Bildes. Zu den Informationen in einem Bild zählen die Anzahl der Farben (Farbtiefe) und die Auflösung (Anzahl der Pixel). Je mehr Farben und je höher die Auflösung, desto detaillierter und realistischer wirkt das Bild. Allerdings steigt mit zunehmender Informationsmenge auch die Dateigröße, was bei der Speicherung und Übertragung von Bildern berücksichtigt werden muss. Diesen Trade-off zwischen Qualität und Größe finden wir nicht nur bei Bildern, sondern grundsätzlich bei der Digitalisierung von Dingen der realen Welt wieder. Da kommen wir später im Buch noch öfter drauf zurück.

Um zum Ende dieses Kapitels dennoch eine funktionierende Animation auf unser Display zu zaubern, verwenden wir eine Figur, die sich auch in Schwarzweiß gut erkennen lässt: Pacman.

4.9 Pacman

4.9.1 Von Farbe zu Schwarzweiß

Am Beispiel von Mario haben wir oben schon einen Prozess kennengelernt, mit dem wir Farbbilder in Schwarzweißbilder umwandeln können. Das Ergebnis der drei Sprites seht ihr in Abbildung 4.27.

Für Mario, der in seinem echten Klempnerdasein farbig ist, funktioniert die schwarzweiße Welt nicht gut. Wir nutzen deshalb für die Animation auf unserem Display mit Pacman jemanden, der sowieso in Unifarbe daher kommt. Wenn ihr aber trotzdem mal ausprobieren wollt, wie Mario in Schwarzweiß auf dem Laufsteg aussieht, könnt ihr das gerne versuchen. Ihr findet die drei Bilder in dem zu diesem Buch gehörigen [GitHub-Repository](#).

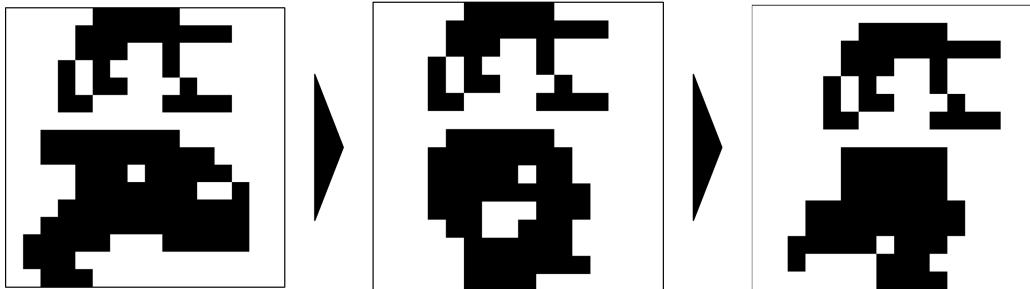


Abbildung 4.27: Super Marios Laufbewegung als Sequenz dreier Schwarzweiß-Bilder.

Pacman ist ein weiterer Spielesklassiker aus den 1980er Jahren. Ich selbst habe mit Pacman eine eigene Geschichte: Mein erstes größeres Programmierprojekt war die Entwicklung eines Pacman-Spiels in Turbo Pascal. Das ist 1998 während meines Ausslandsaufenthaltes in North Carolina in der elften Schulkasse entstanden. Für das Projekt habe ich Pacman ebenfalls Pixel für Pixel gezeichnet und animiert.

Das Spiel ist schnell erklärt. Pacman ist eine kreisförmige, gelbe Figur, die der Spieler mit dem Joystick durch ein Labyrinth steuert und dabei möglichst viele Punkte sammelt bzw. frisst. Pacman ist aber nicht alleine, er wird von mehreren Monstern gejagt, die ihn fangen wollen. Einen Eindruck, wie das Spiel früher ausgesehen hat, bekommt ihr in Abbildung 4.28.

Wenn Pacman läuft, öffnet und schließt sich sein Mund im Wechsel. In Abbildung 4.29 sieht ihr diese Mundbewegung in drei Bildern. Diese wollen wir im Folgenden auf unser Display übertragen und dort hintereinander anzeigen.

Pacman von Farbe zu Schwarzweiß umzuwandeln ist denkbar einfach. Da es mit Gelb nur eine Farbe gibt, können wir einfach jedes gelbe Pixel zu weiß (1) und alle anderen Pixel zu schwarz (0) umwandeln. Dazu laden wir wieder die Bitmap-Datei mit `Pillow` und iterieren über alle Pixel. Wir müssen nicht einmal den genauen Farbwert kennen: Für jedes Pixel prüfen wir nur, ob es sich von Weiß (#FFFFFF) unterscheidet. Wenn ja, fügen wir eine 1 zur Liste hinzu, sonst eine 0. Am Ende speichern wir die Liste als Schwarzweiß-Bitmap-Datei:

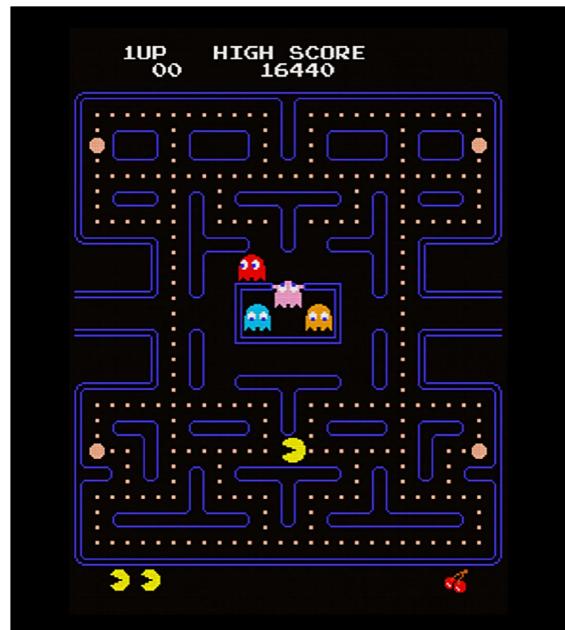


Abbildung 4.28: So sah Pacman früher als Arcade-Game aus.

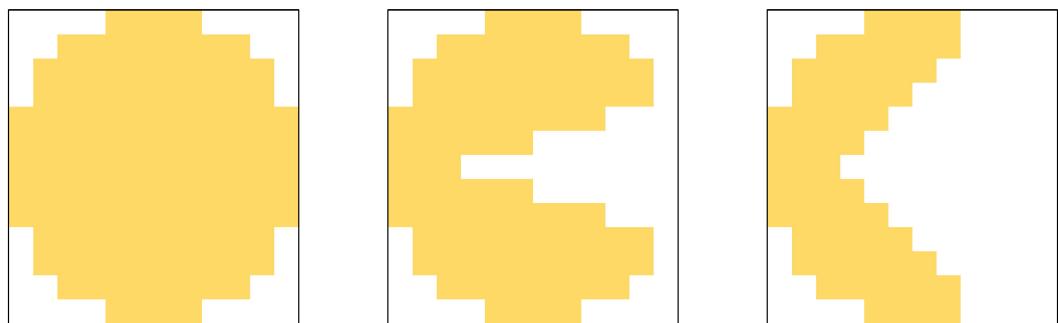


Abbildung 4.29: Pacman öffnet den Mund in 3 Bildern

```

from PIL import Image
image = Image.open("bmp/pacman_closed.bmp") ①

w, h = image.size
bw_values = [] ②
③

for y in range(h):
    for x in range(w):
        r, g, b = image.getpixel((x, y))

        if r == 255 and g == 255 and b == 255: ④
            bw_values.append(0)
        else:
            bw_values.append(1) ⑤

print(bw_values) ⑥

```

- ① Wir laden die Bitmap-Datei, hier als Beispiel die von Pacman mit geschlossenem Mund.
- ② Wir lesen die Breite und Höhe des Bildes aus, damit wir wissen, über welche Pixelkoordinaten wir iterieren müssen.
- ③ Wir erstellen eine leere Liste, in der wir die Schwarzweiß-Werte speichern.
- ④ Wir verwenden zwei verschachtelte `for`-Schleifen, um über alle Pixelkoordinaten zu iterieren. Einmal über die Zeilen (`y`) und einmal über die Spalten (`x`).
- ⑤ Wir lesen den RGB-Wert des aktuellen Pixels an der Koordinate `x` und `y` aus.
- ⑥ Wir prüfen, ob das Pixel weiß ist (alle Farbwerte 255). Wenn ja, fügen wir eine 0 zur Liste hinzu, sonst eine 1.

Die Ausgabe auf der Konsole sieht so aus:

```
[0, 0, 0, 0, 1, 1, 1, 1, ... 1, 1, 1, 1, 1, 0, 0, 0, 0]
```

In der Variable `bw_values` stehen nun die Pixelwerte des Schwarzweiß-Bildes von Pacman mit geschlossenem Mund. Jeder Wert ist entweder eine 1 (weiß) oder eine 0 (schwarz). Wunderbar, genau das Format benötigen wir auch, um ein Bild auf dem Display darzustellen. Wenn wir den entsprechenden Boilerplate-Code ergänzen, können wir das Bild direkt auf dem Display anzeigen lassen.

```

from tinkerforge.ip_connection import IPConnection
from tinkerforge.bricklet_oled_128x64_v2 import BrickletOLED128x64V2

ipcon = IPConnection()
ipcon.connect('localhost', 4223)

```

```

oled = BrickletOLED128x64V2('25zo', ipcon)
oled.clear_display()

# Code zum Laden und Umwandeln des Bildes in Schwarzweiß wie oben
...

oled.write_pixels(10, 10, 21, 22, pacman_closed)

```

Wir setzen Pacmans obere linke Ecke auf die Koordinate (10, 10) auf dem Display. Der zweite Punkte für die rechte untere Ecke ergibt sich aus der Breite und Höhe der Bitmap. Diese ist 12x13 Pixel, dementsprechend ist die Koordinate (21, 22). Der letzte Parameter ist die Liste mit den Schwarzweiß-Pixelwerten, die wir oben erstellt haben. Wenn wir das Programm ausführen, dann sollte Pacman mit geschlossenem Mund auf dem Display erscheinen (s. Abbildung 4.30a).

4.9.2 Pacman-Animation

Das selbe Vorgehen können wir exakt eins zu eins mit den anderen beiden Bildern anwenden. Dazu könnten wir den Code von oben kopieren, den Pfad zur Bitmap-Datei anpassen und die Variable für die Pixelwerte umbenennen. Aber das wäre wieder doppelter Code, was wir vermeiden wollen. Besser ist es, wenn wir den Code in eine Funktion packen, die wir dann für jedes Bild aufrufen können:

```

def convert_rgb_to_bw(image_path):
    image = Image.open(image_path)
    w, h = image.size

    bw_values = []
    for y in range(h):
        for x in range(w):
            r, g, b = image.getpixel((x, y))

            if r == 255 and g == 255 and b == 255:
                bw_values.append(0)
            else:
                bw_values.append(1)

    return bw_values

```

Die Funktion `convert_rgb_to_bw` nimmt den Pfad eines beliebigen Bildes über den Parameter `image_path` entgegen, lädt es (Zeile 2) und speichert es auf der Variable `image`. Danach

ermitteln wir die Dimensionsn (Zeile 3) und iterieren durch jedes Pixel mithilfe zweier verschachtelter `for`-Schleifen (Zeilen 6 und 7). Im Inneren der Schleifen füllen wir eine Liste mit den Schwarzweiß-Pixelwerten nach der oben beschriebenen Logik. Am Ende gibt die Funktion die Liste zurück (Zeile 15). Diese Funktion können wir für jedes der drei Pacman-Bilder aufrufen und die Pixelwerte in Variablen speichern:

```
pacman_half = convert_rgb_to_bw("bmp/pacman_half.bmp")
pacman_closed = convert_rgb_to_bw("bmp/pacman_closed.bmp")
pacman_open = convert_rgb_to_bw("bmp/pacman_open.bmp")
```

Wenn wir jetzt die drei Variablen `pacman_half`, `pacman_closed` und `pacman_open` hintereinander der `write_pixels()`-Methode übergeben, können wir die Bilder nacheinander auf dem Display anzeigen lassen, und so eine Animation erzeugen. Zwischen jeden Frame fügen jeweils eine kurze Pause mit der `time.sleep()`-Funktion ein, damit das menschliche Auge die einzelnen Bilder wahrnehmen kann. Das Ganze lassen wir in einer Endlosschleife laufen, damit die Animation immer wiederholt wird:

```
wait_time = 0.1
while True:
    oled.write_pixels(10, 10, 21, 22, pacman_closed)
    time.sleep(wait_time)
    oled.write_pixels(10, 10, 21, 22, pacman_half)
    time.sleep(wait_time)
    oled.write_pixels(10, 10, 21, 22, pacman_open)
    time.sleep(wait_time * 2)
    oled.write_pixels(10, 10, 21, 22, pacman_half)
    time.sleep(wait_time)
```

Vergesst nicht, ganz oben in eurem Programm die `time`-Bibliothek zu importieren, damit wir die `sleep()`-Funktion verwenden können. Über die Variable `wait_time` können wir die Geschwindigkeit der Animation steuern. Je kleiner der Wert, desto schneller wechselt Pacman die Bilder. In der letzten Zeile der Schleife habe ich die Wartezeit verdoppelt, damit Pacman mit offenem Mund etwas länger zu sehen ist. Schließlich muss er ja die Punkte auch fressen können.

So sieht es als bewegtes Bild auf dem Display aus (s. [?@fig-images-pacman-animation-display](#)):

4.9.3 Bewegung

Im echten Spiel bewegt Pacman nicht nur seinen Mund sondern er bewegt sich auch fort. Das Laufen können wir nicht alleine durch das schnelle Anzeigen von Bildern in einer Abfolge



(a) Erste Frame

(b) Zweiter Frame

(c) Dritter Frame

Abbildung 4.30: Die drei Pacman-Frames auf dem OLED-Display.

umsetzen. Stattdessen müssen wir zusätzlich zur Fressanimation die x-Koordinate von Pacman verändern. Und zwar am besten erhöhen wir sie, denn dann würde sich Pacman nach rechts auf dem Bildschirm bewegen, passend zu seiner Blickrichtung.

Anstatt die x-Koordinate fest auf den Wert 10 zu setzen, führen wir eine Variable `x_pos` ein und erhöhen sie in jedem Schleifendurchlauf um einen festen Wert, zum Beispiel 2 Pixel:

Literaturverzeichnis

- Adami, Christoph. 2016. „What is Information?“ *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 374 (2063): 20150230. <https://doi.org/10.1098/rsta.2015.0230>.
- Brookshear, J. Glenn, und Dennis Brylow. 2020. *Computer science: an overview*. 13th edition, global edition. NY, NY: Pearson.
- Gallenbacher, Jens. 2020. *Abenteuer Informatik: IT zum Anfassen für alle von 9 bis 99, vom Navi bis Social Media*. 4. Auflage, korrigierte Publikation. Berlin Heidelberg: Springer.
- Petzold, Charles. 2022. *Code: the hidden language of computer hardware and software*. 2. Aufl. Hoboken: Microsoft Press.
- Pólya, George, und John Horton Conway. 2004. *How to solve it: a new aspect of mathematical method*. Expanded Princeton Science Library ed. Princeton science library. Princeton [N.J.]: Princeton University Press.
- Scott, John C. 2009. *But how do it know?: the basic principles of computers for everyone*. Oldsmar, FL: John C. Scott.

Index

‘BrickletRGBLEDV2‘, [22](#)

‘IPConnection‘, [22](#)

Bibliothek, [22](#)

Binärsystem, [55](#)

Konstante, [65](#)

Objekt, [23](#)