

Hands-On Computer Science

Nicolas Meseth

8. September 2025

Inhaltsverzeichnis

Vorwort	5
Was macht dieses Buch besonders?	5
Tipps für die Lektüre	6
Experimente	8
Hands-On Programmieren lernen	8
Spannende Experimente	9
Frust ist dein Freund – zumindest ein bisschen	10
Voraussetzungen	12
Visual Studio Code	12
Python	12
Software von Tinkerforge	12
Tinkerforge-Bibliothek für Python	12
Brick Daemon	12
Brick Viewer	13
1 Farben	14
Zusammenfassung	14
Experimentaufbau	14
Hardware	14
Erste Schritte	15
1.1 Erstes Programm: LED ansteuern	15
1.1.1 Programme	20
1.1.2 Boilerplate Code	20
1.1.3 Bibliotheken	20
1.1.4 Klassen und Objekte	21
1.1.5 Schlüsselwörter	22
1.1.6 Objekte erzeugen	22
1.1.7 Methoden	22
1.1.8 Ein Objekt für die LED	23
1.1.9 Zusammenfassung unseres ersten Programms	23
1.1.10 Und jetzt?	23
1.2 Licht und Farben	24
1.2.1 Blick auf die Physik	24

1.2.2	Additive Farbmischung	26
1.2.3	Subtraktive Farbmischung	26
1.3	Pulsierende LED	30
1.3.1	Abzählbare Wiederholungen	30
1.3.2	Bedingte Wiederholungen	33
1.4	Farbkreise	34
1.5	Regenbogen-LED	37
1.5.1	Runde für Runde	38
1.5.2	Geschwindigkeit steuern	39
2	Zahlen	43
	Zusammenfassung	43
	Experimentaufbau	43
	Hardware	43
2.1	Erste Schritte mit dem Drehknopf	43
2.2	Zähler auslesen	46
2.3	Kontrollstrukturen	49
2.4	LED-Dimmer 1.0	49
2.5	Zahlensysteme	51
2.5.1	Unser Dezimalsystem	52
2.5.2	Das Oktalsystem	53
2.5.3	Das Binärsystem	53
2.5.4	Andere Systeme	57
2.6	Bits & Bytes	57
2.6.1	Zwei Zustände	57
2.6.2	Acht Bits macht ein Byte	58
2.6.3	Kilo, Mega, Giga	60
2.7	LED-Dimmer 2.0	61
2.7.1	<code>min()</code> und <code>max()</code>	61
2.7.2	Helligkeit entkoppeln	62
2.7.3	Konstanten	64
2.8	Druckknopf auslesen	65
2.9	LED-Dimmer 3.0	66
2.9.1	Farbe per Variable steuern	67
2.9.2	Farbe per Knopfdruck ändern	68
2.10	Funktionen	70
3	Texte	74
	Zusammenfassung	74
	Experimentaufbau	74
	Hardware	74
	Erste Schritte mit dem Abstandssensor	74
3.1	Lichtschranke	77

3.2	Hinderniserkennung	80
3.3	Universelles Eingabegerät	81
3.4	Texte kodieren	84
3.4.1	Wie viele Bits benötigen wir?	84
3.4.2	Wörterbücher	90
3.5	ASCII-Code	92
3.5.1	Unicode	94
3.6	LED-Dimmer 4.0	95
4	Bilder	99
	Zusammenfassung	99
	Experimentaufbau	99
	Hardware	99
	Erste Schritte mit	99
4.1	Bitmaps	99
4.2	Komprimierte Bilder	99
4.3	Hexadezimal	99
4.4	Der RGB-Code	99
5	Ton	100
	Zusammenfassung	100
	Experimentaufbau	100
	Hardware	100
	Erste Schritte mit	100
	Literaturverzeichnis	101

Vorwort

Glückwunsch, ihr seid angekommen! Egal, wie euer Weg hierher aussah: Ihr habt dieses Buch geöffnet. Vielleicht studiert ihr an der Hochschule Osnabrück und wurdet (zu eurem Glück) dazu verpflichtet, oder ihr seid ganz bewusst hier gelandet und freut euch darauf, etwas Neues zu lernen – genau wie ich.

Dieses Buch entstand ursprünglich, um meinen Veranstaltungen an der Hochschule Osnabrück eine verständliche und praxisnahe Grundlage zu geben. Es dient als Hauptlektüre für meine Vorlesungen und als Nachschlagewerk für alle, die eine Sitzung verpasst haben oder Themen eigenständig vertiefen möchten. Besonders willkommen sind Quereinsteiger, Wiederholer und neugierige Menschen, die bisher noch gar keinen Kontakt mit der [Hochschule Osnabrück](#) hatten.

Hier bekommt ihr keine trockene Theorie, sondern eine praxisnahe Einführung in die Grundlagen moderner Computer und unserer digitalen Welt. Das Fach dahinter heißt auf Deutsch Informatik, international Computer Science. Der Titel Hands-On Computer Science verrät bereits, worum es geht: Es wird praktisch, und zwar von Anfang an.

Was macht dieses Buch besonders?

Lehrbücher zur Informatik gibt es reichlich. Viele sind großartig, aber kaum eines passt genau zu dem, was ich mit meinen Studierenden vorhave. Woran liegt das?

Viele klassische Informatikbücher versuchen, das gesamte Fach möglichst umfassend abzubilden. Das ist sinnvoll für angehende Informatikerinnen und Informatiker, aber meine Zielgruppe seid ihr: Studierende in Studiengängen wie [Management nachhaltiger Ernährungssysteme](#), [Lebensmittelproduktion](#) oder [Agrarsystemtechnologien](#) – oder ihr seid gar nicht an der Hochschule, sondern wollt euch einfach einen besseren Zugang zur digitalen Welt erarbeiten.

Kurz gesagt: Dieses Buch ist für alle gedacht, die in die digitale Welt eintauchen wollen, ohne sich mit komplizierten Details zu überfordern. Dafür braucht ihr kein allumfassendes Nachschlagewerk, sondern einen klaren roten Faden, der euch Schritt für Schritt an die grundlegenden Konzepte heranführt.

Viele Bücher versprechen Praxisnähe, doch oft endet sie in nüchternen Übungsaufgaben am Kapitelende. Genau hier setzt Hands-On Computer Science an und macht drei Dinge anders:

1. Ihr lernt informatische Konzepte anhand spannender Experimente mit Microcontrollern, Sensoren, Buttons, LEDs und Displays.
2. Wir führen wichtige Konzepte früh ein, gehen aber erst nach und nach ins Detail und wiederholen sie regelmäßig.
3. Theorie und Praxis sind nicht getrennt, sondern eng miteinander verbunden – Programmieren und informatische Grundlagen lernt ihr gleichzeitig.

Schon ab Kapitel 1 beginnt ihr zu programmieren, und zwar nicht abstrakt, sondern konkret mit Hardware-Bauteilen wie einer LED. Im Laufe des Buches lernt ihr Schritt für Schritt neue Hardwarekomponenten kennen, die direkt mit relevanten informatischen Konzepten verknüpft sind. So schließt ihr nicht nur eine Reihe Experimente erfolgreich ab, sondern baut fast nebenbei ein solides Fundament in Informatik und Programmierung auf. Wenn alles gut läuft, merkt ihr kaum, wie schnell ihr gelernt habt.

Tipps für die Lektüre

Weil es in diesem Buch viel ums Programmieren geht, findet ihr viele Codeblöcke und Code-Snippets. Als Einstiegssprache verwenden wir Python. Warum ausgerechnet Python? Das erfahrt ihr später.

Codeblöcke sind deutlich vom übrigen Text abgehoben, grau hinterlegt und in einer Schreibmaschinenschrift dargestellt, etwa Courier New oder Consolas. Hier ein kleines Beispiel:

```
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.

Die Annotationen sind mit einer kleinen Zahl versehen. Wenn ihr das Buch online lest und mit der Maus über diese Zahl fahrt, erscheint ein Tooltip, der die jeweilige Codezeile erklärt. Das funktioniert nur online, nicht in der PDF- oder Druckversion. Dort sind die Erläuterungen unter dem Codeblock aufgeführt.

Weil wir in jedem Kapitel ein Programm Schritt für Schritt entwickeln und es dadurch immer länger wird, lasse ich gelegentlich Stellen im Code weg, um den Fokus auf neu hinzugekommene

Zeilen zu legen. Diese Auslassungen markiere ich in den Codeblöcken mit drei Punkten (...). Keine Sorge: Den gesamten Code findet ihr stets am Ende eines Abschnitts. Außerdem liegt jedes Programm im zu diesem Buch gehörigen GitHub-Repository unter:

<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 direkt in eure Zwischenablage, und ihr könnt ihn anschließend in euer geöffnetes Visual Studio Code oder eine andere IDE einfügen und ausprobieren. In der Online-Version lassen sich manche Codeblöcke einklappen, damit ihr weniger scrollen müsst.

Alles klar? Dann schauen wir uns an, was uns in diesem Buch erwartet!

Experimente

Hands-On Programmieren lernen

Hast du dich schon einmal gefragt, wie man Informationen über Licht übertragen kann? Oder wie man mit Licht den Puls messen kann? Oder wie man mit zwei einfachen Kabeln einen Wasserstandssensor baut? Das alles klingt vielleicht weit hergeholt, ist aber tatsächlich machbar – und wie genau, das wollen wir in diesem Buch herausfinden! Dabei werden wir nicht nur die digitale Welt der Computer und Programmierung kennenlernen, sondern auch mit spannenden Geräten in der analogen Welt arbeiten. In jedem Kapitel arbeiten wir mit anderen Geräten, die dir unterschiedliche Facetten der digitalen Welt näherbringen und gleichzeitig ermöglichen, das Programmieren spielerisch zu erlernen.

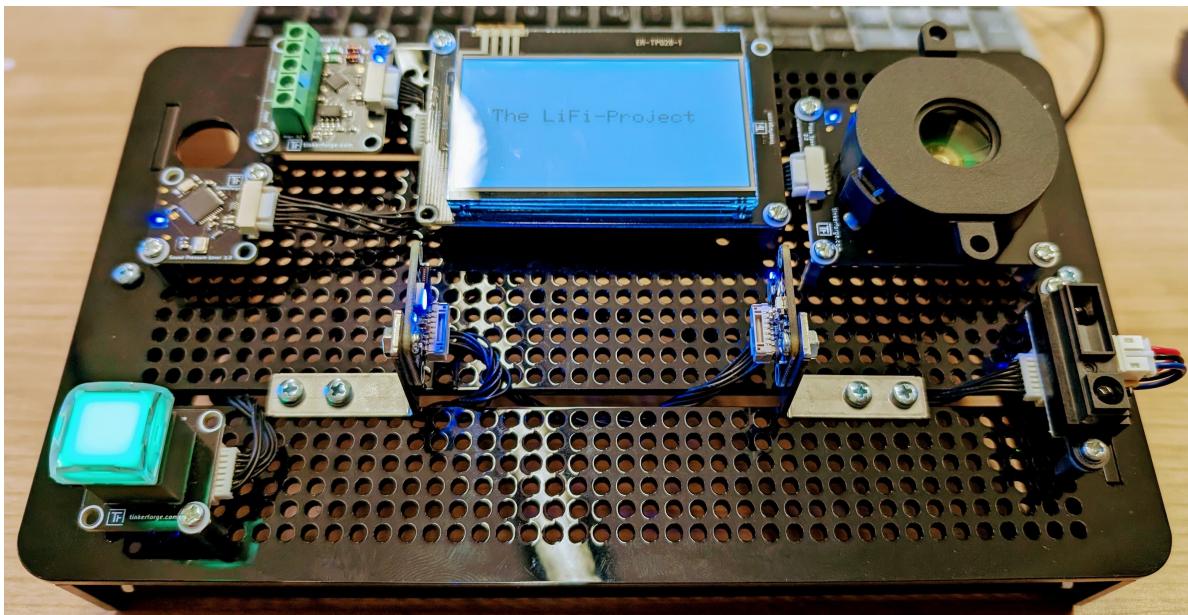


Abbildung 1: Tinkerforge Workbench mit vielen Geräten

Hier ein Überblick über die Geräte, mit denen wir gemeinsam experimentieren werden. Zusammengefasst kosten alle Komponenten 249 €. Aber keine Sorge: Wenn du das Buch im Rahmen meines Moduls „Digitalisierung und Programmierung“ an der Hochschule Osnabrück liest, erhältst du für das gesamte Semester ein komplettes Hardware-Kit.

Was?	Bauteil	Anzahl	Preis pro Stück
Bunte LED	RGB LED Bricklet 2.0	1	8 €
Button mit integrierter, bunter LED	RGB LED Button Bricklet	1	15 €
Licht- und Farbsensor	Color Bricklet 2.0	1	17 €
LCD Touchdisplay	LCD 128x64 Bricklet	1	33 €
Piezo Lautsprecher	Piezo Speaker Bricklet 2.0	1	19 €
Infrarot-Entfernungsmesser	Distance IR 4-30cm Bricklet 2.0	1	20 €
Analoger Spannungssensor	Analog In Bricklet 3.0	1	14 €
Schalldruckpegelsenor	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 €

Spannende Experimente

Kapitel für Kapitel werden wir an unterschiedlichen Experimenten arbeiten. Dabei lernst du nicht nur, wie man Hardware-Komponenten miteinander verbindet, sondern vor allem auch, wie man Computer – diese universellen Problemlösungsmaschinen – für eigene Ideen und Lösungen programmieren kann. Hier ist der Überblick, was dich in diesem Buch erwartet:

Kapitel	Experiment(e)
Kapitel 1	Wir lassen eine LED einen Regenbogenfarbverlauf über die Zeit erzeugen.
Kapitel 3	Wir lernen, wie man Texte umständlich und ohne Tastatur eingeben kann – über Handgesten.

Kapitel	Experiment(e)
Kapitel 4	Wir verbinden Tabellenkalkulation mit Bildern und Touchdisplays
?@sec-codes	Wir lernen Morse-Code und wie wir diesen über einen Lautsprecher ausgeben können.
?@sec-conversion	Wir verwenden einfache Kippschalter, um analoge Werte in digitale Werte umzuwandeln.
?@sec-information	
?@sec-sensors	Wir bauen einen Wasserstandssensor mit einem analogen Spannungssensor.
?@sec-signals	Wir basteln einen Pulsmesser aus einem Farbsensor
?@sec-protocols	Wir übertragen Nachrichten über Lichtsignale
?@sec-encryption	Wir verstecken geheime Botschaften in harmlosen Nachrichten
?@sec-algorithms	
?@sec-compression	Wir entwickeln ein Verfahren, um Information zu komprimieren.
?@sec-computer	Wir entwickeln eine Rechenmaschine, die zwei Bytes addieren kann, mit nur einem einem Bauteil.
?@sec-problems	

Frust ist dein Freund – zumindest ein bisschen

Eins möchte ich gleich vorwegnehmen: Beim Programmierenlernen ist eine gewisse Portion Frust unvermeidbar. Klingt unangenehm? Ist es auch! Aber es ist zugleich Teil eines enorm wertvollen Lernprozesses. Jeder Fehler, den du machst, ist eine Gelegenheit, um zu verstehen, wie Computer wirklich funktionieren – nämlich absolut präzise und ohne jede Toleranz für Fehler.

Computer sind gnadenlose Lehrer. Sie zeigen dir sofort und unerbittlich, wenn etwas nicht stimmt – sei es ein vergessener Punkt, ein falscher Buchstabe oder ein simpler Zahlendreher. Das kann frustrieren, aber genau dieses direkte und sofortige Feedback hilft dir auch, schnell und effektiv zu lernen. Sobald du verstehst, wie du aus Fehlermeldungen sinnvolle Schlüsse ziehst und deine Programme entsprechend korrigierst, wirst du belohnt – mit Erfolgserlebnissen und einer steilen Lernkurve.

Also, wenn mal etwas nicht klappt: Nimm es nicht persönlich, sondern sieh es als Herausforderung. Atme tief durch, mach dir klar, dass Fehler unvermeidbar und sogar wichtig sind, und probier es noch einmal. Ich verspreche dir: Es lohnt sich!



Abbildung 2: Ein frustrierter Frosch

Voraussetzungen

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

Visual Studio Code

Eine beliebte Entwicklungsumgebung, die ihr kostenlos herunterladen könnt. [Hier geht's zur Download-Seite.](#)

Python

Eine Programmiersprache, die für die Experimente verwendet wird. Ihr könnt Python von der offiziellen Website herunterladen: [Python Download](#).

Software von Tinkerforge

Tinkerforge-Bibliothek für Python

Eine Sammlung von Funktionen, die die Programmierung der Tinkerforge-Hardware erleichtern. Ihr könnt die Bibliothek mit dem folgenden Befehl installieren: `pip install tinkerforge`.

Brick Daemon

Ein Hintergrundprozess, der die Kommunikation mit der Tinkerforge-Hardware ermöglicht. Ihr könnt den Brick Daemon von der Tinkerforge-Website herunterladen: [Brick Daemon Download](#).

Brick Viewer

Ein Tool, das eine grafische Benutzeroberfläche für die Interaktion mit der Tinkerforge-Hardware bietet. Ihr könnt den Brick Viewer von der Tinkerforge-Website herunterladen: [Brick Viewer Download](#).

1 Farben

Das erste Kapitel hat es gleich in sich: Wir lernen etwas über Farben und wie sie im Computer funktionieren. Gleichzeitig steigen wir in die Programmierung ein und schreiben unser erstes Programm. Dabei nutzen wir eine LED und erzeugen einen Regenbogenfarbverlauf.

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. Um dahin zu kommen, lernen wir in Abschnitt 1.1 zuerst, wie man überhaupt eine LED aus einem Programm heraus steuern kann. In Abschnitt 1.2 werfen wir einen kurzen Blick auf Farben im allgemeinen und wie diese im Computer erzeugt werden. Dabei unterscheiden wir die additive von der subtraktiven Farbmischung. Danach bereiten wir uns in Abschnitt 1.3 auf den Regenbogenfarbverlauf vor, indem wir zuerst eine pulsierende LED programmieren. Wir lernen dann in Abschnitt 1.4, wie man mittels RGB-Farbcodes überhaupt einen Regenbogen erzeugen kann und warum das für uns Menschen nützlich ist. Am Ende in Abschnitt 1.5 schreiben wir schließlich das Programm, das die LED in allen Farben des Regenbogens leuchten lässt.

Unsere wichtigsten Lernziele in diesem Kapitel sind:

1. Wir verstehen, wie Farben im Computer funktionieren und beschrieben werden.
2. Wir entwickeln unser erstes Programm am Beispiel einer LED-Steuerung.
3. Wir lernen Schleifen als wichtiges Konzept in der Programmierung kennen.

Experimentaufbau

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.

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.

Erste Schritte

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!

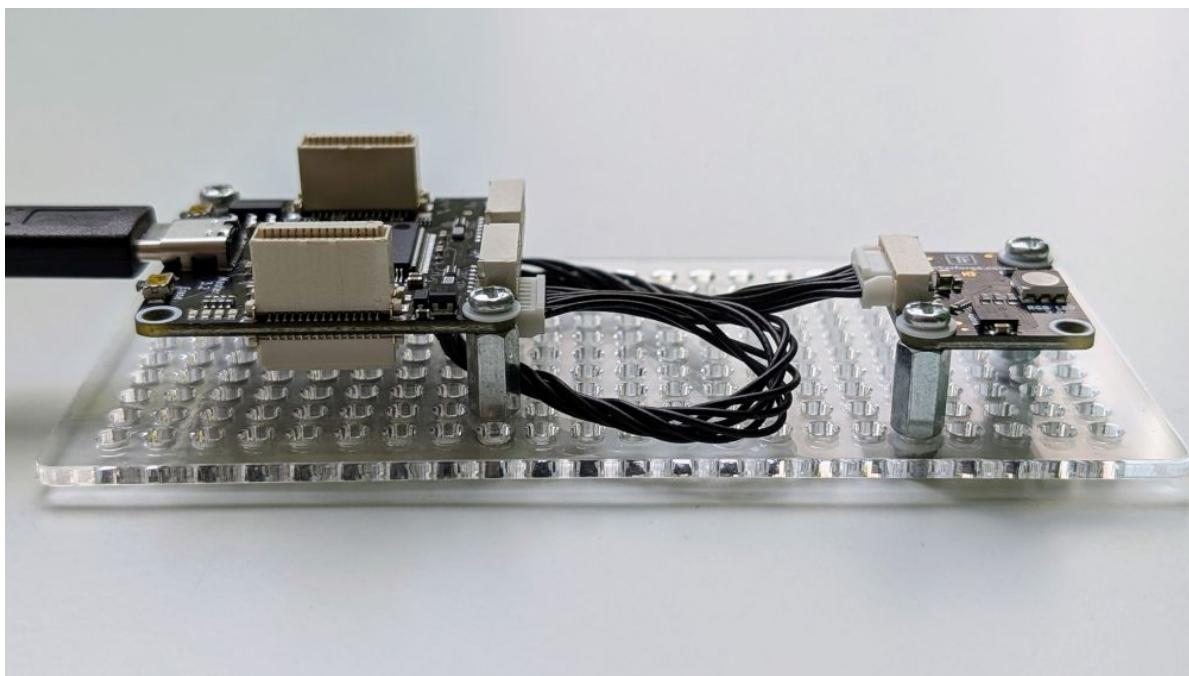
1.1 Erstes Programm: LED ansteuern

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

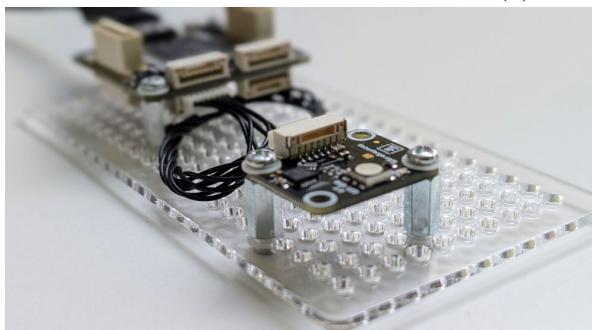
Listing 1.1 Der Boilerplate-Code für die Verbindung mit den Geräten am Beispiel der RGB LED.

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

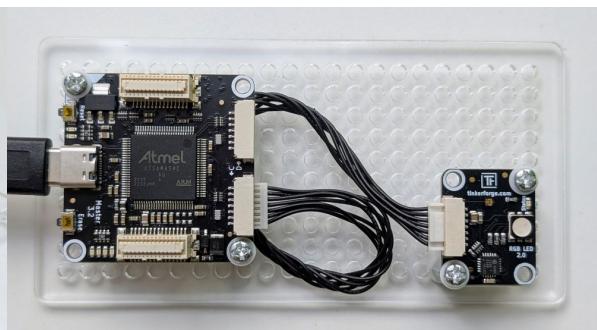
ipcon = IPConnection()                                     ③
ipcon.connect("localhost", 4223)                            ④
led = BrickletRGBLEDV2("ZEP", ipcon)                      ⑤
```



(a) Seitenansicht.



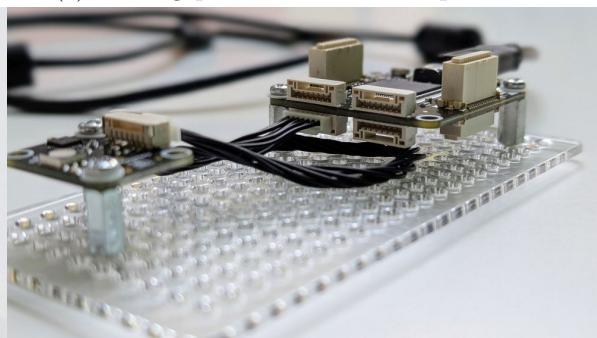
(b) Nahaufnahme der LED.



(c) Montageplatte mit allen Komponenten.



(d) Untenansicht.



(e) Ansicht der vier Steckplätze.

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

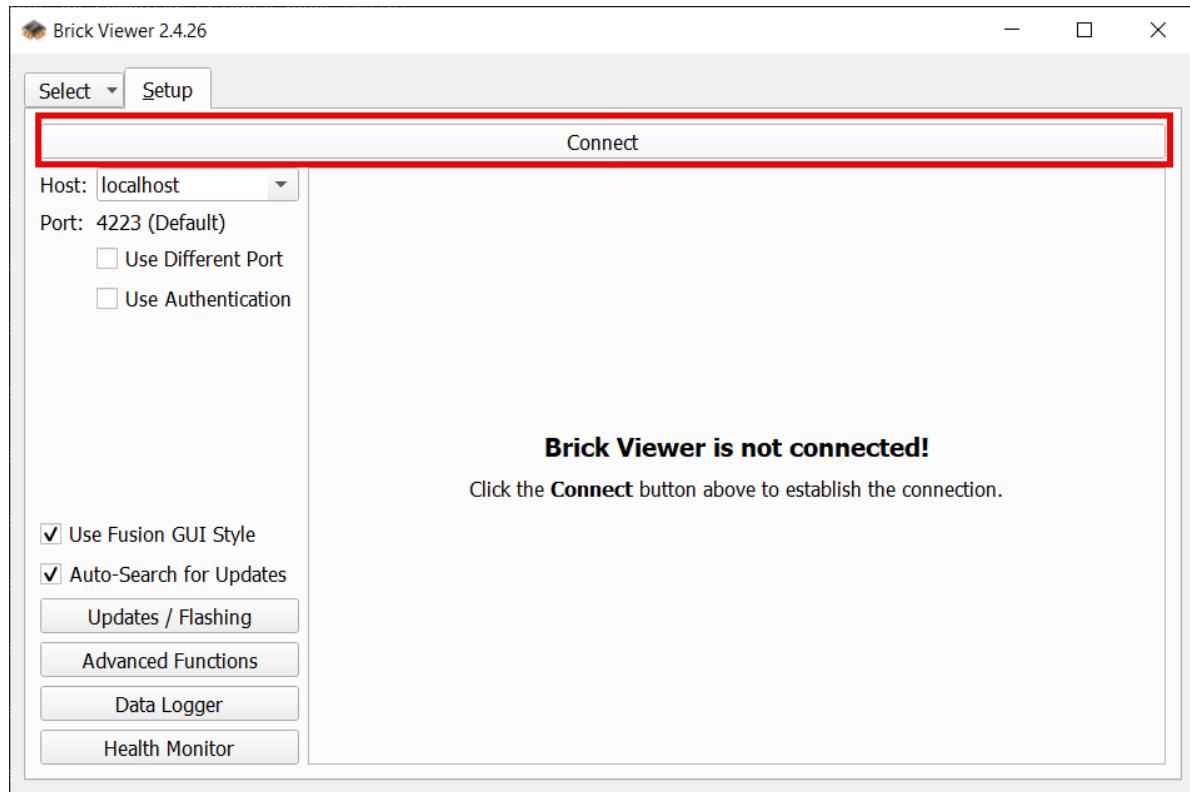


Abbildung 1.2: Über den Connect-Button verbindet ihr den Brick Viewer mit dem angeschlossenen Master Brick.

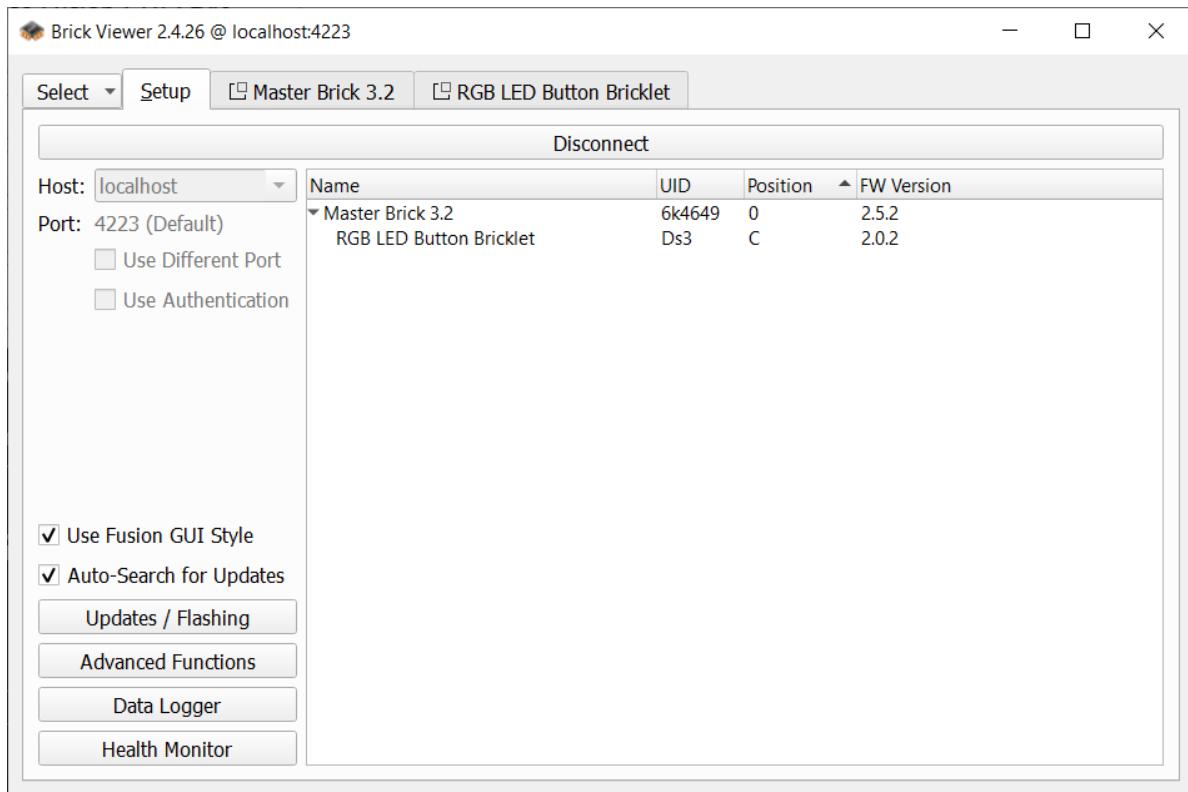


Abbildung 1.3: Der Brick Viewer, nachdem ihr mit dem Master Brick verbunden seid.

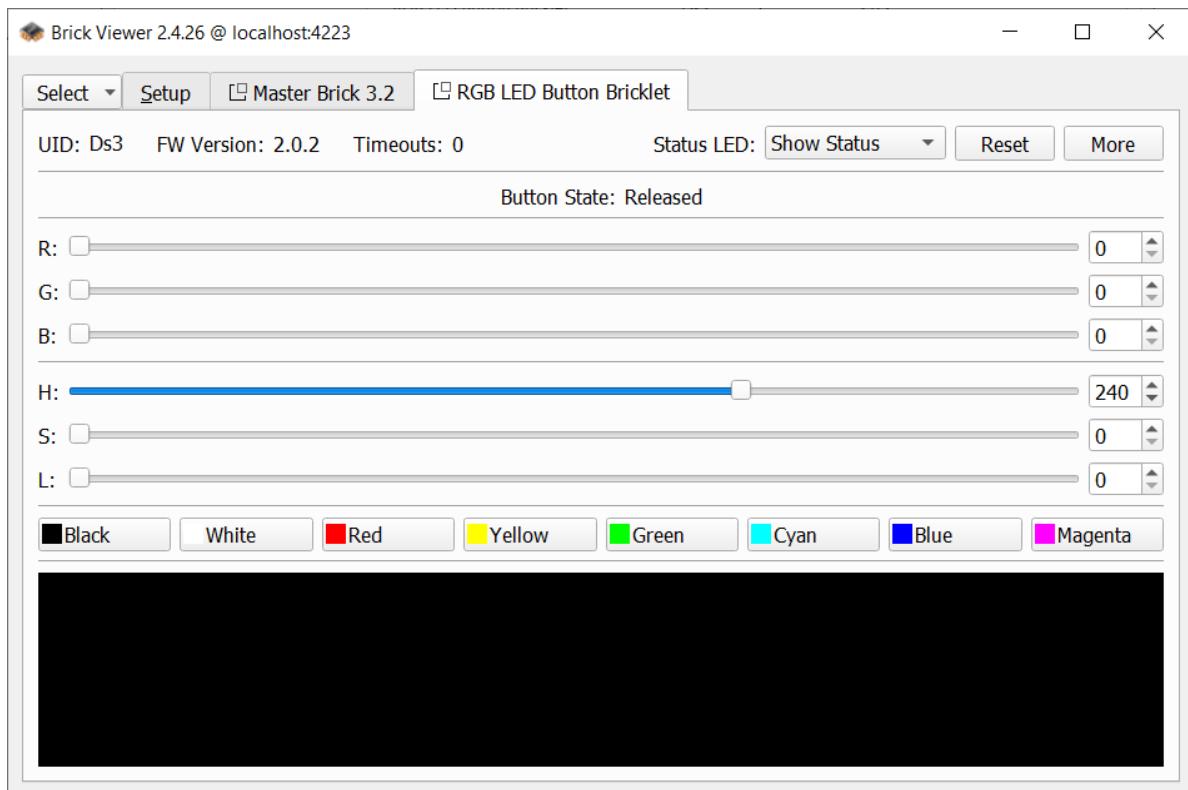


Abbildung 1.4: Die Ansicht für die RGB LED im Brick Viewer, in der ihr alle Funktionen per Klick im Zugriff habt.

- 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.

1.1.1 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.

1.1.2 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.

1.1.3 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.

1.1.4 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.

1.1.5 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.

1.1.6 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.

1.1.7 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.

1.1.8 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](#).

1.1.9 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.

1.1.10 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-Farbkodeierung*. 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](#).

1.2 Licht und Farben

1.2.1 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.

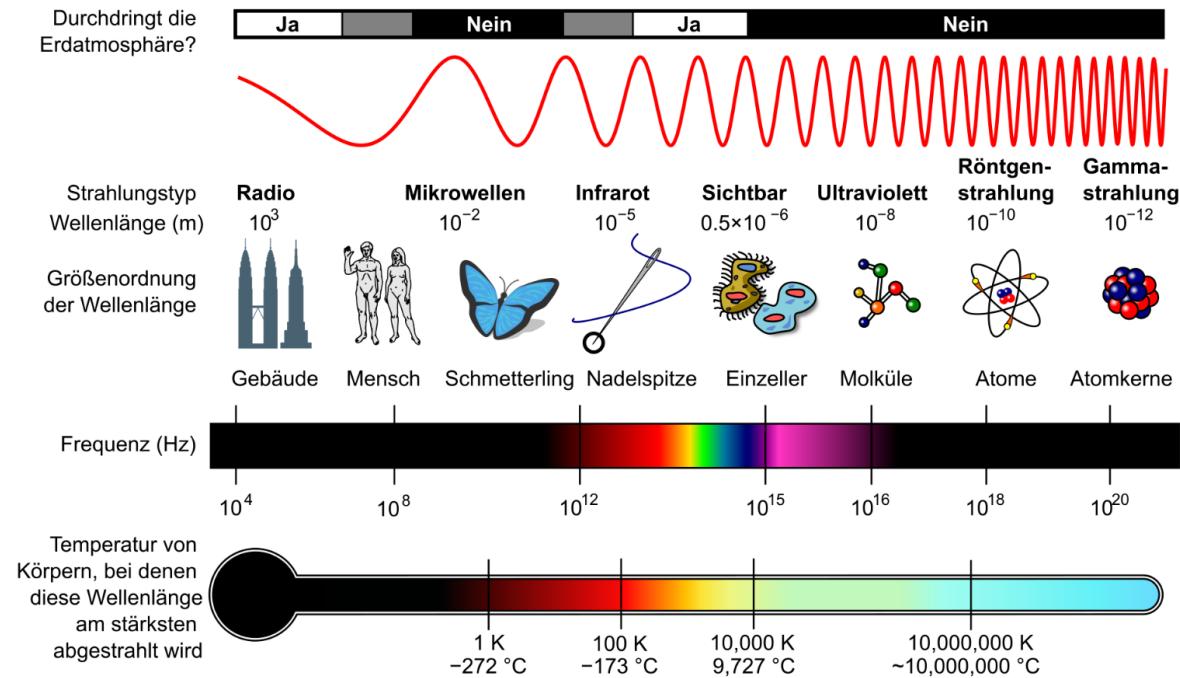


Abbildung 1.5: Das elektromagnetische Spektrum, von dem das sichtbare Licht ein kleiner Teil ist. (Quelle: [Wikipedia](#))

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 besteht, 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. Genaueres 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!

1.2.2 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)
```

1.2.3 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.

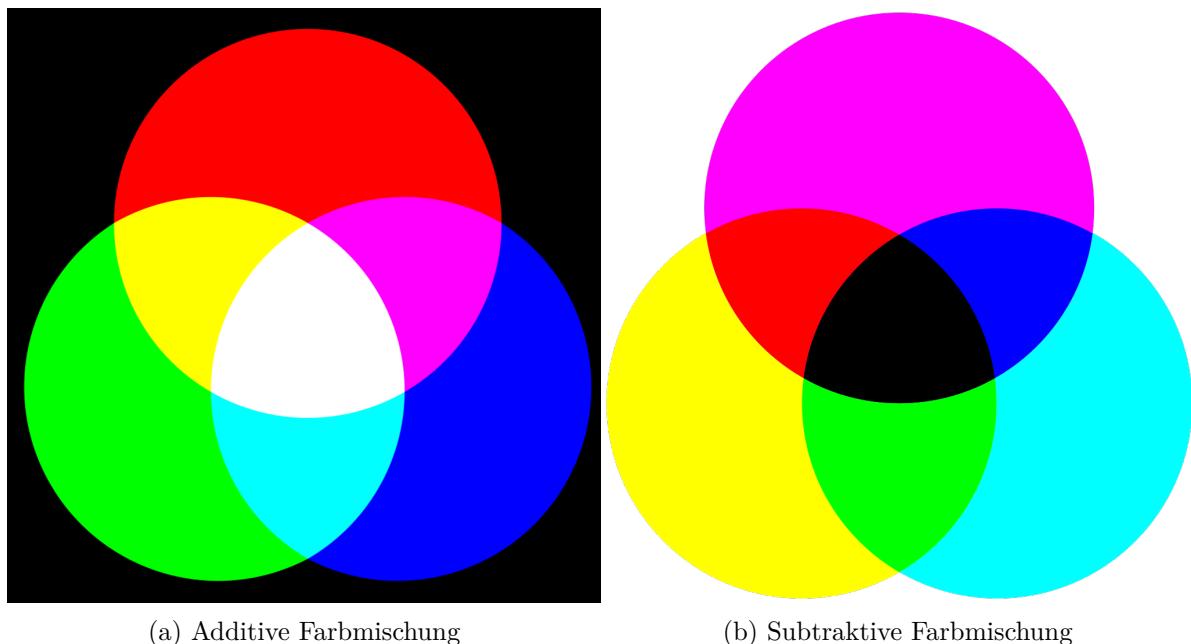


Abbildung 1.6: Additive und subtraktive Farbmischung.

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 Farldrucker 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 einfach 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!



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

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

1.3.1 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?

1.3.2 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 Pulsierzyklus 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 Wiederholungen 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](#).

1.4 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.

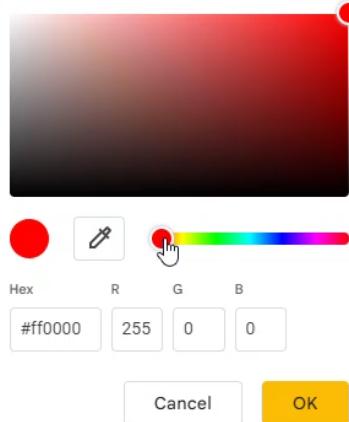


Abbildung 1.8: Die Farbauswahl in Google Slides funktioniert über den Hue-Farbkreis.

In `?@fig-hue-color-picker` sieht 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 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.

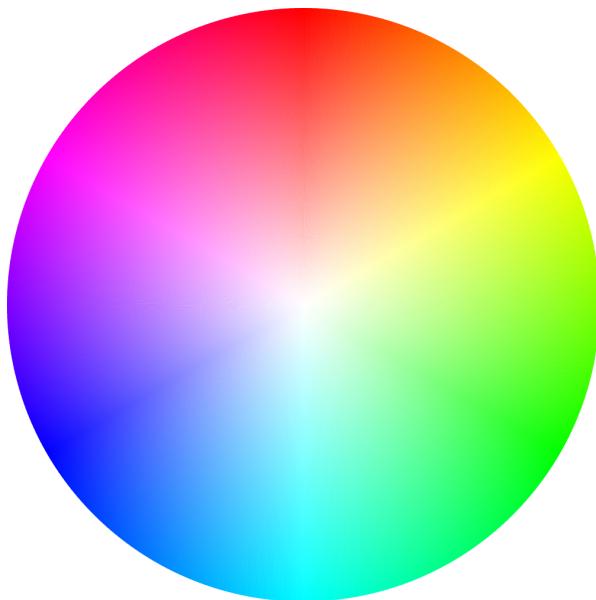


Abbildung 1.9: Der Hue-Farbkreis mit HSV-Werten.

Ein Blick auf die Animation in `?@fig-hue-color-picker` 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
6. Rot = 255, Grün = 0, Blau sinkt linear

Dann beginnt der Zyklus von vorn.

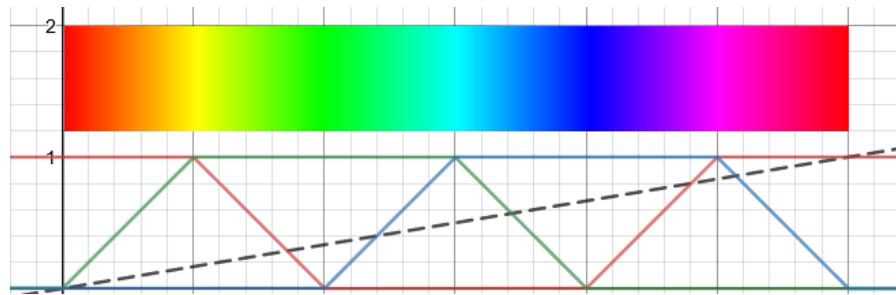


Abbildung 1.10: Der Hue-Farbverlauf mit den Veränderungen der RGB-Werte (Quelle: [Ronja's Tutorials](#)).

1.5 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.

1.5.1 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.

1.5.2 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('ZEP', 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)    42
```

2 Zahlen

Unser Regenbogen aus dem ersten Kapitel war nur der Anfang. Doch wie speichert der Computer eigentlich die RGB-Werte für all diese Farben? Das führt uns zum Binärsystem: Mit nur Einsen und Nullen, verpackt in kleinen Päckchen – den Bytes –, kann er jeden Farnton unseres Regenbogens abbilden. Und noch viel mehr.

Zusammenfassung

Unsere wichtigsten Lernziele in diesem Kapitel sind:

- Wir verstehen wie ein Computer Zahlen darstellt und speichert.
- Wir führen das Bit als zentrale Informationseinheit im Computer ein.
- Wir lernen Kontrollstrukturen kennen, mit denen wir unser Programm steuern können.
- Wir fügen Funktionen zu unserem Werkzeugkasten hinzu, um besseren Code zu schreiben.

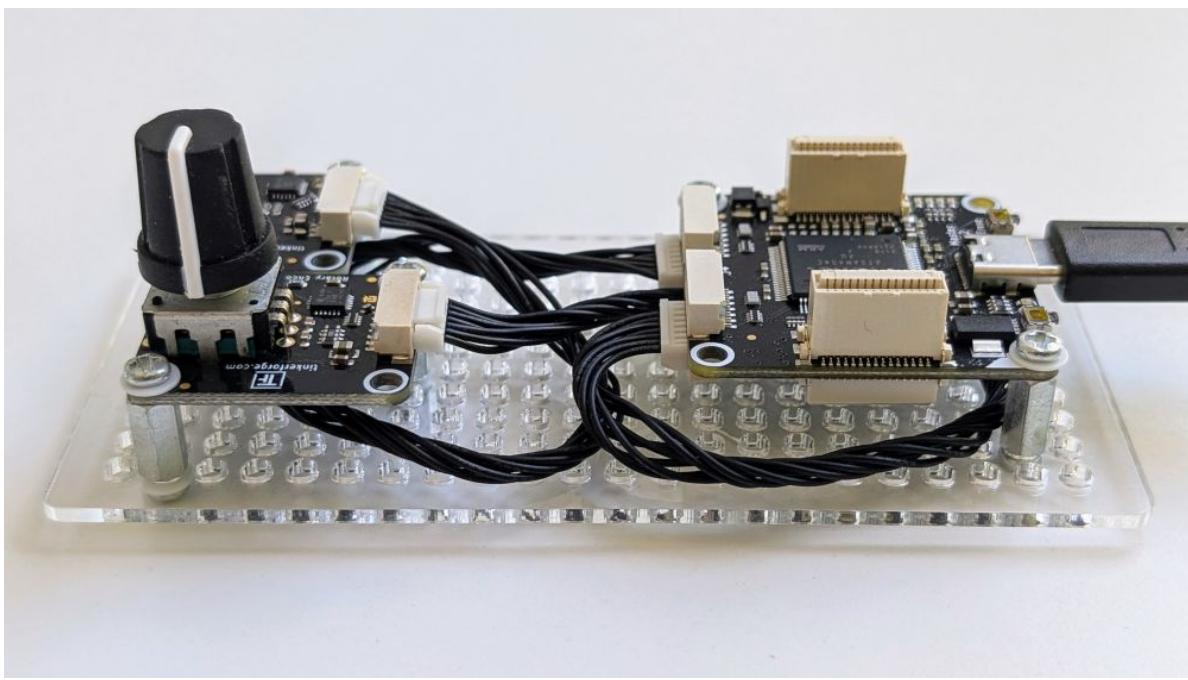
Experimentaufbau

Hardware

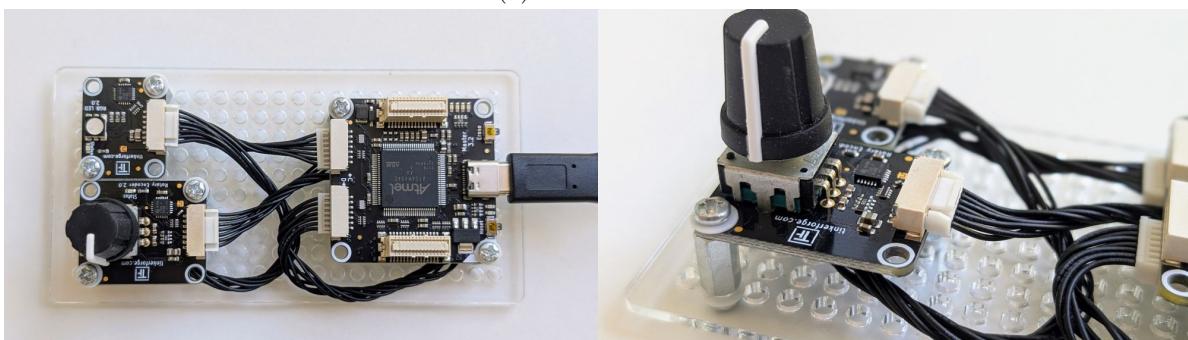
Das erste Experiment in Kapitel 1 war ein guter Einstieg! In diesem Kapitel legen wir noch eine Schippe drauf: Unsere Hardware bekommt ein neues Bauteil – einen Drehknopf ([Rotary Encoder Bricklet 2.0](#)). Das montiert ihr einfach neben der LED, wie in Abbildung 2.1 gezeigt.

2.1 Erste Schritte mit dem Drehknopf

Wie bei der LED werfen wir zuerst einen Blick auf den neuen Drehknopf im Brick Viewer. Schließt dazu euren Master Brick per USB an, startet den Brick Viewer und klickt auf Connect.

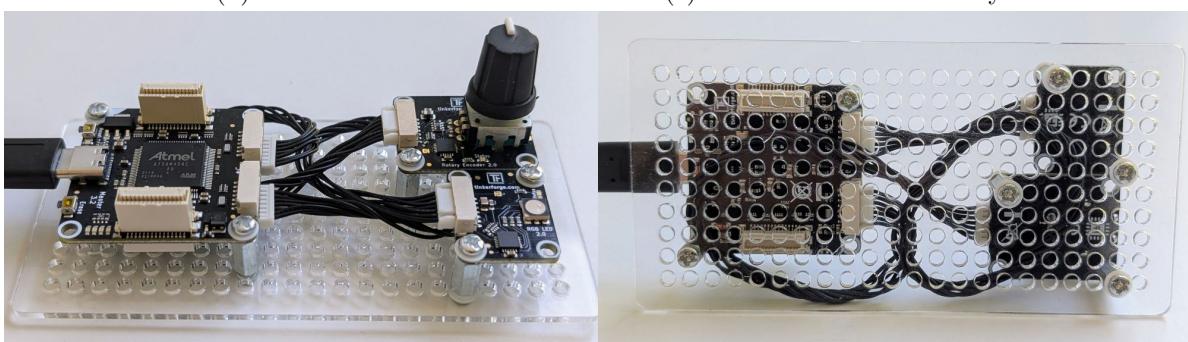


(a) Seitenansicht.



(b) Draufsicht.

(c) Nahaufnahme des Rotary Encoders.



(d) Seitenansicht.

(e) Untenansicht.

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

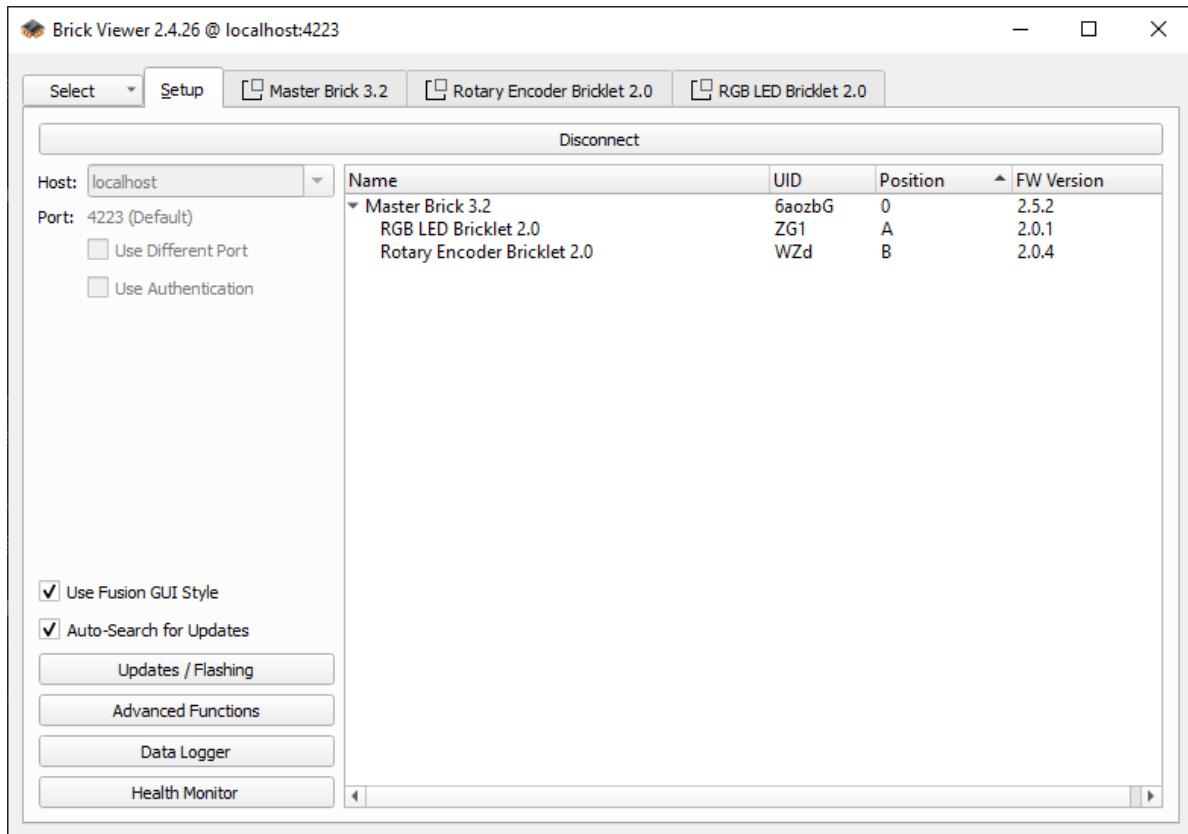


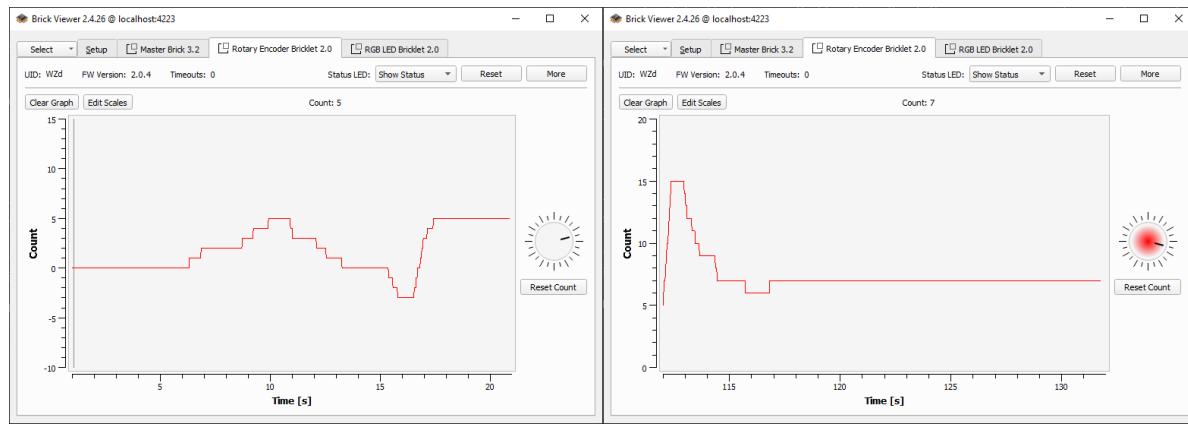
Abbildung 2.2: Der Brick Viewer nach dem Connect.

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.



(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.

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 coolen 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.



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 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 `if`, gefolgt von einer Bedingung.

Neu ist hier die Kontrollstruktur `if`. Sie prüft eine Bedingung, die – wie ihr schon aus Kapitel 1 kennt – nur `True` oder `False` sein kann. 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('WZd', ipcon)
led = BrickletRGBLEDV2('ZG1', 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 254 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:

```
#| code-line-numbers: false
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 $1 \cdot 100 = 100$. Damit haben wir $100 + 20 + 3 = 123$. Ganz einfach und intuitiv.

1	2	3
<hr/>	<hr/>	<hr/>
10^2	10^1	10^0
100	10	1

1	2	3
<hr/>	<hr/>	<hr/>
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

¹Hände hoch, wer daran noch glaubt!

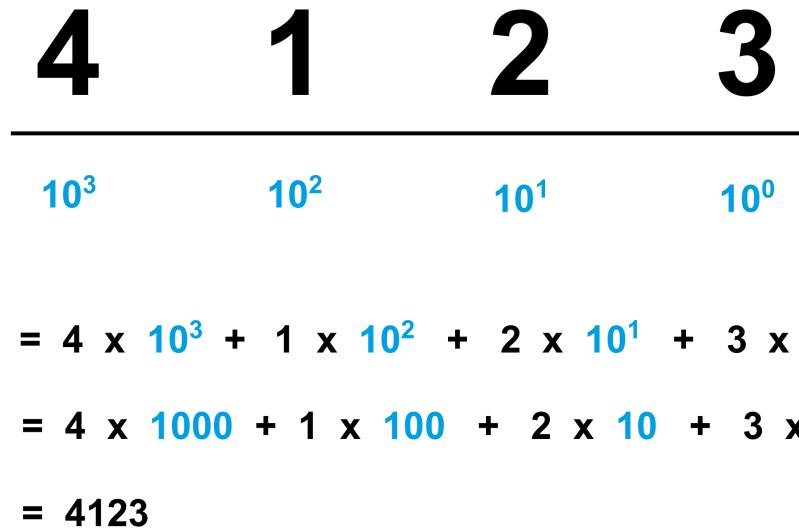


Abbildung 2.6: Jede Stelle steht für eine höhere Potenz der Basis 10.

dafür gesorgt, dass wir dezimal denkende Wesen werden. Warum? Eine Theorie besagt, dass die menschliche Anatomie, insbesondere die Anzahl der Finger, einen Einfluss auf unser Zahlensystem hatte. Zählt einfach mal anhand eurer Finger durch.

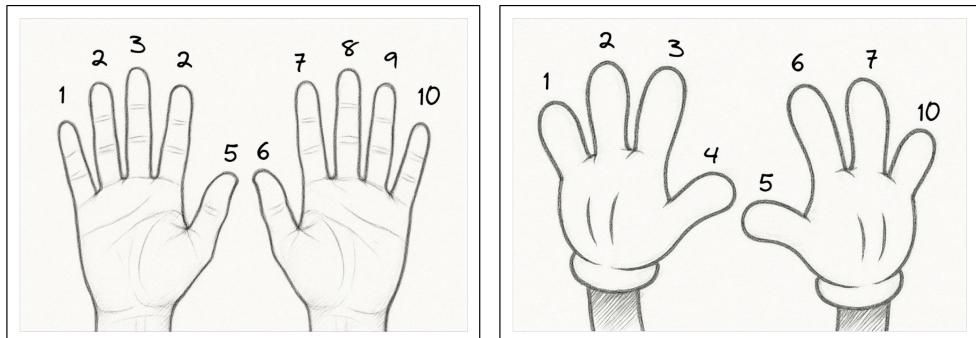
2.5.2 Das Oktalsystem

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!

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.



human hand

cartoon character's hand

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

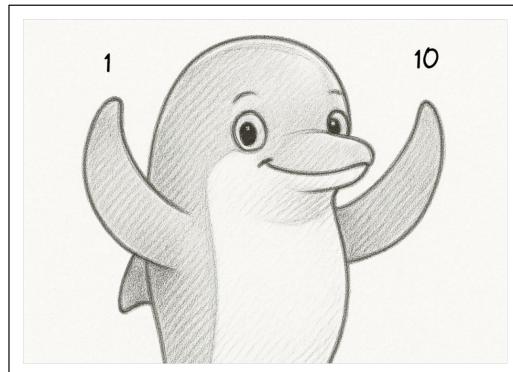
Das Wort “binär” stammt aus dem Lateinischen und bedeutet “paarweise” oder “zu zweit”. Von diesem Wort stammt auch der Name des Stellenwertsystems mit der Basis 2 – und das nicht ohne Grund. Im Binärsystem gibt es für jede Stelle genau zwei Möglichkeiten: 0 oder 1. Ein anderer Begriff ist übrigens Dualsystem,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

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

(octal)

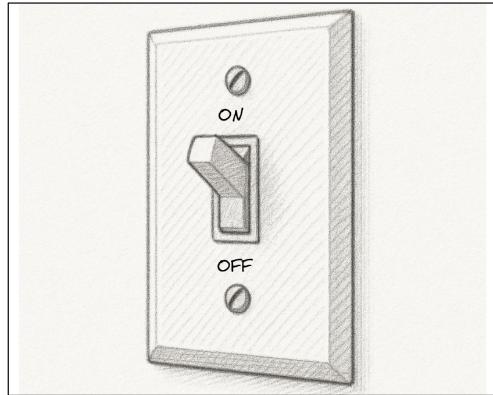
$$\begin{aligned}
 &= 1 \times 8^2 + 2 \times 8^1 + 3 \times 8^0 \\
 &= 1 \times 64 + 2 \times 8 + 3 \times 1 \\
 &= 83 \text{ (decimal)}
 \end{aligned}$$

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.

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

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

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

(a)

(b)

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

place value systems

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

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

n = number of digits
 R = base

2.5.4 Andere Systeme

- Römische Zahlen
- Tally-Schreibweise

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.

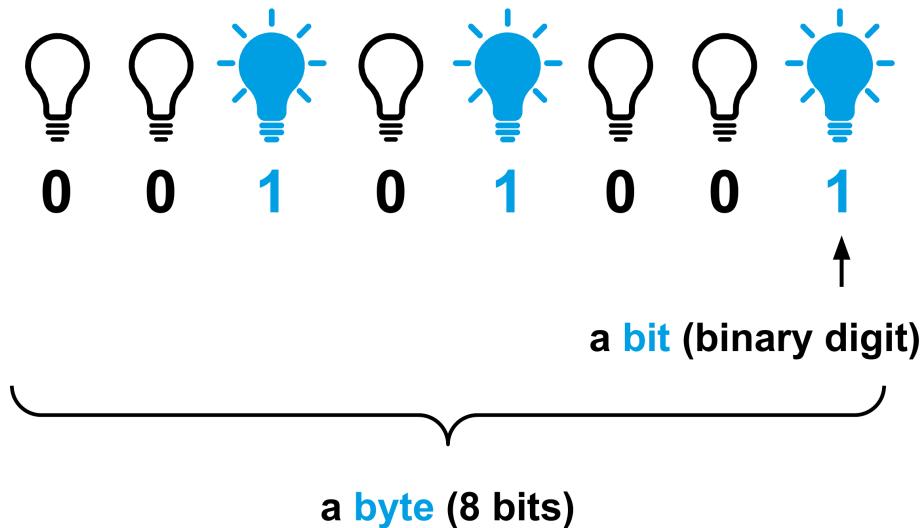


Abbildung 2.12: Ein Byte könnt ihr euch vorstellen wie acht Glühbirnen nebeneinander.

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.

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,

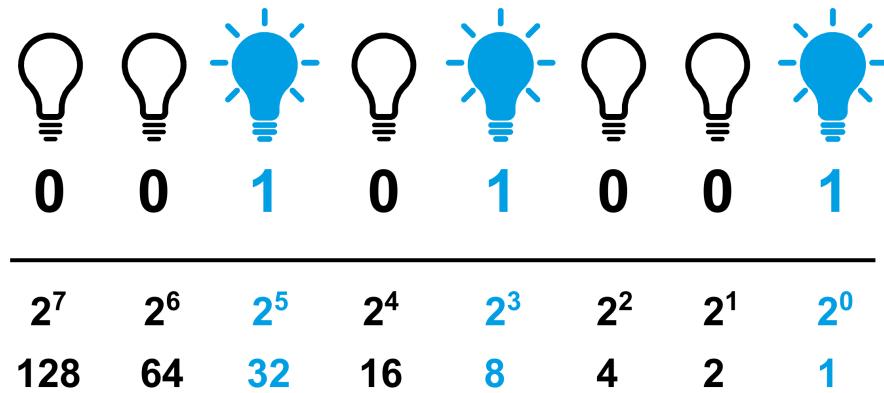


Abbildung 2.13: Jede Glühbirne steht für eine Stelle aus dem Binärsystem.

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 2.1 seht ihr eine Übersicht über die verschiedenen Größenordnungen.

Tabelle 2.1: 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

Potenz (Bytes)	Ausgeschrieben	Bezeichnung (Abkürzung)	Entspricht ca.
10^{21}	Trilliard	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 2.1 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 9 hilft uns dabei. Nachdem wir den neuen Wert des Zählers in der Variable `last_count` gespeichert haben (Zeile 6), 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 9 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 ②

        # Adjust brightness
        brightness += diff
        brightness = max(0, min(255, brightness)) ③

        # 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). 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.

- 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.

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
        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}")
```

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 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, 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 - aber es sollte funktionieren. Gehen wir es einmal durch: Wenn der Button losgelassen wurde (Zeile 5) gelangen wir zur Prüfung der if-Statements. Im ersten Fall wird geprüft, ob die LED gerade Weiß leuchtet (color == "white"). Ist das der Fall, dann wechseln wir jetzt auf Gelb. Im zweiten Schritt sehen wir ein elif, 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 auf `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?

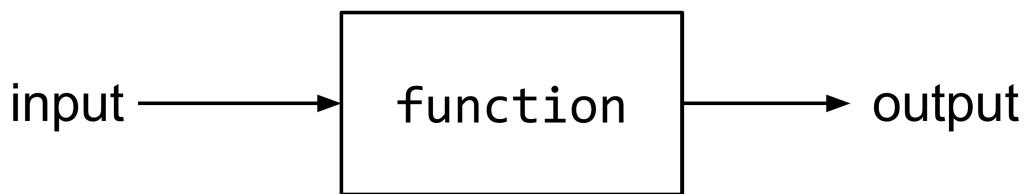


Abbildung 2.14: Funktionen folgen einem Eingabe-Verarbeitung-Ausgabe (EVA) Schema.

```
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.

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('WZd', ipcon)
led = BrickletRGBLEDV2('ZG1', 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:          73
        diff = new_count - last_count
        last_count = new_count

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

3 Texte

Wir nutzen Computer ständig, um Texte zu schreiben – von der WhatsApp-Nachricht über E-Mails und Einladungen bis hin zur Bachelorarbeit. Wir tippen auf dem Smartphone, Tablet oder Computer. Aber habt ihr euch schon einmal gefragt, wie das genau funktioniert? Um das zu verstehen, machen wir einen kleinen Umweg.

Zusammenfassung

Unsere wichtigsten Lernziele in diesem Kapitel sind:

- Wir lernen, wie Computer Texte darstellen und speichern.
- Wir führen ASCII und Unicode als Standards für die Textdarstellung ein.
- Wir verstehen Binärcodierung und wie sie im Computer zum Einsatz kommt.

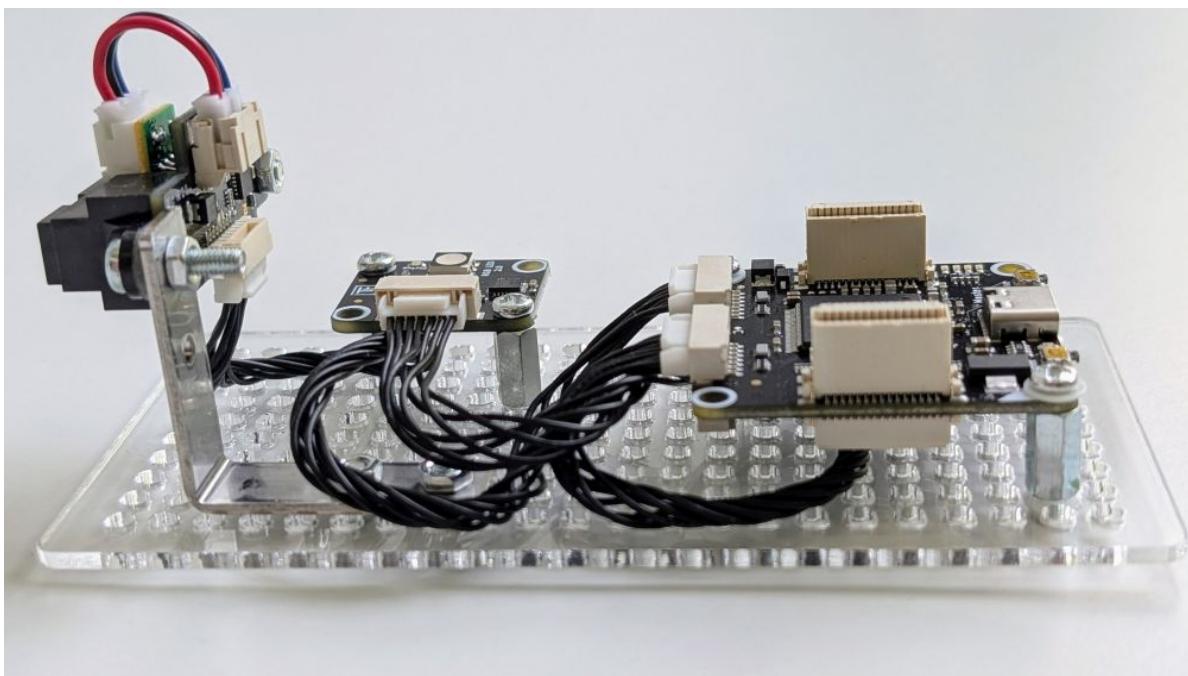
Experimentaufbau

Hardware

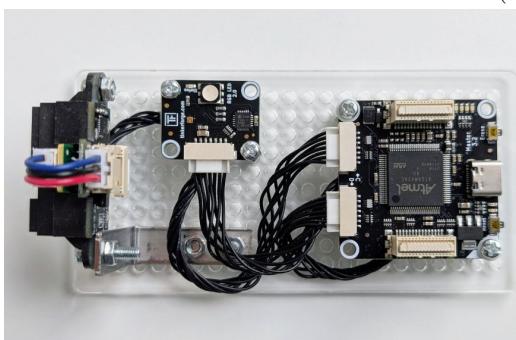
Für dieses Kapitel benötigen wir erneut die LED ([RGB LED Bricklet 2.0](#)), ersetzen aber den Drehknopf aus Kapitel [2](#) durch einen Infrarot-Abstandssensor ([Distance IR 4-30cm Bricklet 2.0](#)). Beide Geräte schließen wir an den Mikrocontroller ([Master Brick 3.2](#)) an und fixieren sie auf einer 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.

Erste Schritte mit dem Abstandssensor

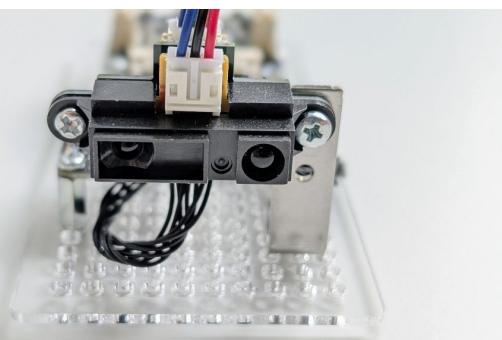
Wie bei der LED werfen wir zuerst einen Blick auf den neuen Abstandssensor im Brick Viewer. Schließt dazu euren Master Brick per USB an, startet den Brick Viewer und klickt auf Connect. Im Tab Setup sollten nun neben der LED auch der Abstandssensor erscheinen. Denkt daran: Dort findet ihr auch die UID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



(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.

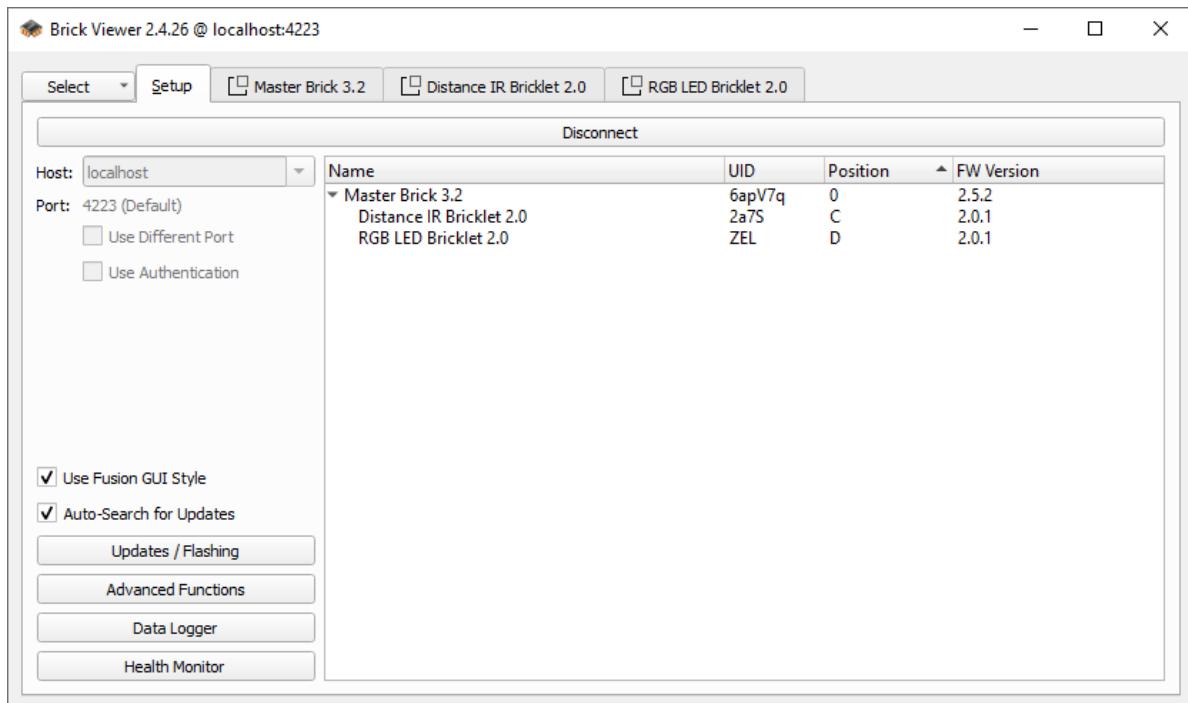


Abbildung 3.2: Nach erfolgreicher Verbindung erscheint der Infrarot-Entfernungsmeßgeräte in der Übersicht des Brick Viewers.

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?

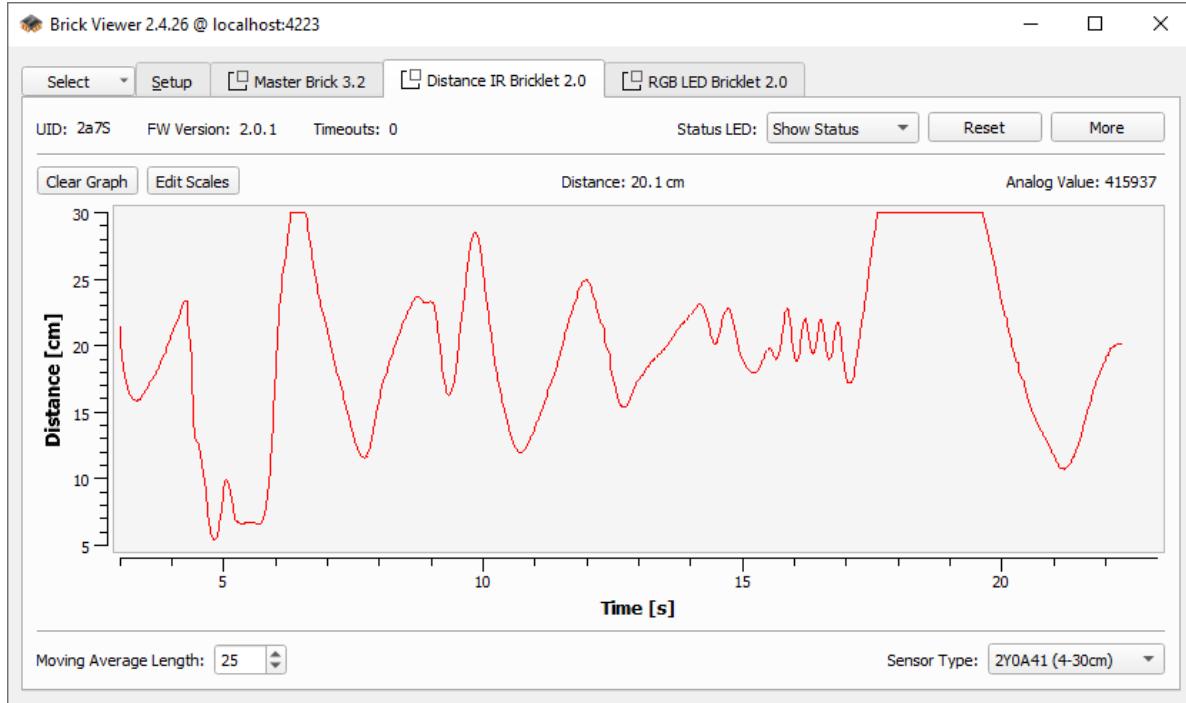


Abbildung 3.3: Der Brick Viewer zeigt die aktuelle Entfernung im Zeitverlauf an.

3.1 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 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:

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('2a7S', ipcon)                                ①
led = BrickletRGBLEDV2('ZEL', ipcon)                                     ②
```

- 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:

```

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

3.2 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:

- 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.

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
```

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.

3.3 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 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: ①
                print(f"1 bei {distance} mm")
                receiving = False
            elif distance <= 170:
                print(f"0 bei {distance} mm")
                receiving = False
        else: ②
            if distance >= 300:
                receiving = True ③

```

```

    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.

3.4 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.

3.4.1 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 hält. 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, 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 3.1: 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}")

    last_distance = distance
```

```

        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 3.2: Anzahl Bits und mögliche Kodierungen.

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

3.4.2 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 = {}
SYMBOLS[0] = "A"                                     ①
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.

3.5 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.

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	01000000	130	58	X	120	01100000	170	78	x
25	00011001	031	19	EM	57	00110001	071	39	9	89	01000001	131	59	Y	121	01100001	171	79	y
26	00011010	032	1A	SUB	58	00110010	072	3A	:	90	01000010	132	5A	Z	122	01100010	172	7A	z
27	00011011	033	1B	ESC	59	00110011	073	3B	:	91	01000011	133	5B	[123	01100011	173	7B	{
28	00011100	034	1C	FS	60	00110000	074	3C	<	92	01000000	134	5C	\	124	01100000	174	7C	
29	00011101	035	1D	GS	61	00110001	075	3D	=	93	01000001	135	5D]	125	01100001	175	7D)
30	00011110	036	1E	RS	62	00110010	076	3E	>	94	01000010	136	5E	^	126	01100010	176	7E	~
31	00011111	037	1F	US	63	00110011	077	3F	?	95	01000011	137	5F	-	127	01100011	177	7F	DEL

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

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
    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.

3.5.1 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?

3.6 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('2a7S', ipcon)

led = BrickletRGBLEDV2('ZEL', 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 Programmierskills sind gewachsen!

Im folgenden Kapitel [4](#) 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

Unsere wichtigsten Lernziele in diesem Kapitel sind:

- Wir führen das Hexadezimalsystem als ein Zahlensystem mit einer Basis > 10 ein

💡 Experiment

- Einführung in die Idee einer Pixelmatrix aus Farbwerten als Bild
- Verwendung des LCD-Displays zur Veranschaulichung, auch wenn nur schwarz/weiß
- Verwendung von Zeichen auf dem LCD, um Pixel-Bitmaps für Schriftarten hervorzuheben
- Studierende nutzen Bitmap-Sheet, um ein eigenes Logo zu entwerfen und auf dem LCD anzuzeigen

Experimentaufbau

Hardware

Erste Schritte mit ...

4.1 Bitmaps

4.2 Komprimierte Bilder

4.3 Hexadezimal

4.4 Der RGB-Code

5 Ton

Das erste Kapitel hat es gleich in sich: Wir lernen etwas über Farben und wie sie im Computer funktionieren. Gleichzeitig steigen wir in die Programmierung ein und schreiben unser erstes Programm. Dabei nutzen wir eine LED und erzeugen einen Regenbogenfarbverlauf.

Zusammenfassung

Experimentaufbau

Hardware

Erste Schritte mit ...

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.
- 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‘, [21](#)
‘IPConnection‘, [21](#)

Bibliothek, [21](#)
binär, [54](#)
Binärsystem, [53](#)
Byte, [58](#)

elektromagnetische Strahlung, [24](#)

HSV-Modell, [35](#)

Klasse, [21](#)
Konstante, [64](#)

Methode, [22](#)

Objekt, [21](#), [22](#)
Objektinstanz, [22](#)

Programm, [20](#)

RGB-Farbkodeierung, [24](#)

Schleife, [31](#)
Schlüsselwort, [22](#)
Stellenwertsystem, [52](#)

Zählerschleife, [31](#)