E Einführung

E.1 Erste Erfahrungen mit Zahlen und Zeichenketten

Wir schließen unser ESP32-Board an den Rechner an und starten die Thonny-IDE. In dem Terminal-Fenster (s. Kapitel V, Abb. 9) wird die Micropython-Version angegeben; darunter erscheint das Python-Prompt: >>>

Hinter das Prompt-Zeichen geben wir nun ein:

```
23+8
```

Dann betätigen wir die Enter-Taste. Daraufhin sieht das Terminalfenster so aus:

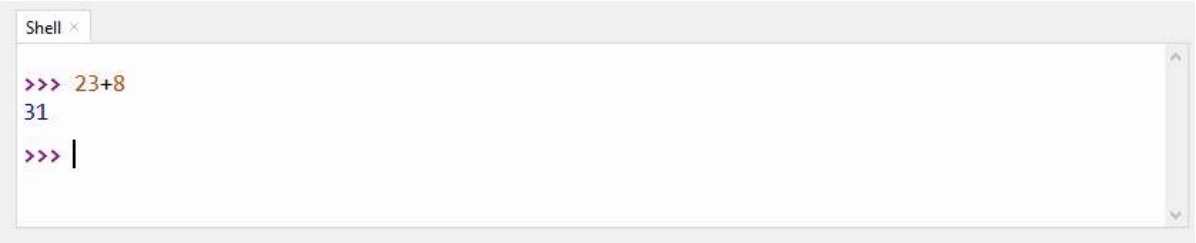

Abb. 1

Unter der Rechenaufgabe steht das Ergebnis, darunter eine Leerzeile, und darunter ist wieder das Prompt-Zeichen zu sehen: Micropython wartet auf eine neue Eingabe.

Ein Blick hinter die Kulissen: Tatsächlich war es nicht das Terminal, das hier gerechnet hat. Wie schon im Kapitel V dargelegt, hat das Terminal die eingegebene Aufgabe an das Micropython-System auf dem Mikrocontroller weitergeleitet. Und dieses Micropython-System hat die Aufgabe bearbeitet und das Ergebnis an das Terminal gesendet. Anschließend wird auf eine neue Eingabe gewartet... Diese Schleife wird auch als Read-Evaluate-Print-Loop (kurz: **REPL**) bezeichnet.

Betrachten wir eine weitere Aufgabe:

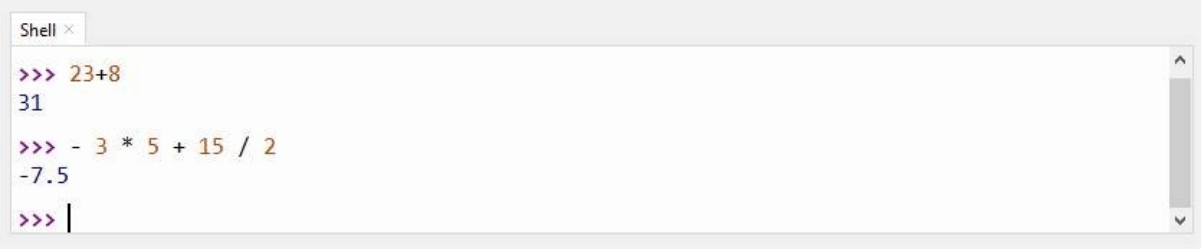

Abb. 2

Wir schließen:

- In den Term können wir Leerzeichen einfügen. Micropython ignoriert sie.
- Micropython kann auch mit negativen Zahlen rechnen.
- Micropython beherrscht die Regel "Punktrechnung vor Strichrechnung".
- Micropython kann auch mit Dezimalzahlen umgehen.

Prüfen Sie nun selbst einmal nach, ob Micropython auch mit (runden) Klammern umgehen kann.

Zahlen können in **Variablen** gespeichert werden; dazu schreibt man hinter den Variablennamen ein Gleichheitszeichen, gefolgt von dem zu speichernden Wert. Man sagt auch: Mit `a = 22` wird **der Variablen a der Wert 22 zugewiesen**. Variablennamen können aus mehreren Zeichen bestehen. Allerdings sind nur folgende Zeichen zulässig: Groß- und Kleinbuchstaben (ohne deutsche Sonderzeichen wie ä, Ä, ..., ß), Ziffern und der Unterstrich (_). Ein Variablenname darf nicht mit einer Ziffer beginnen. Testen Sie selbst einige Zeichenfolgen aus.

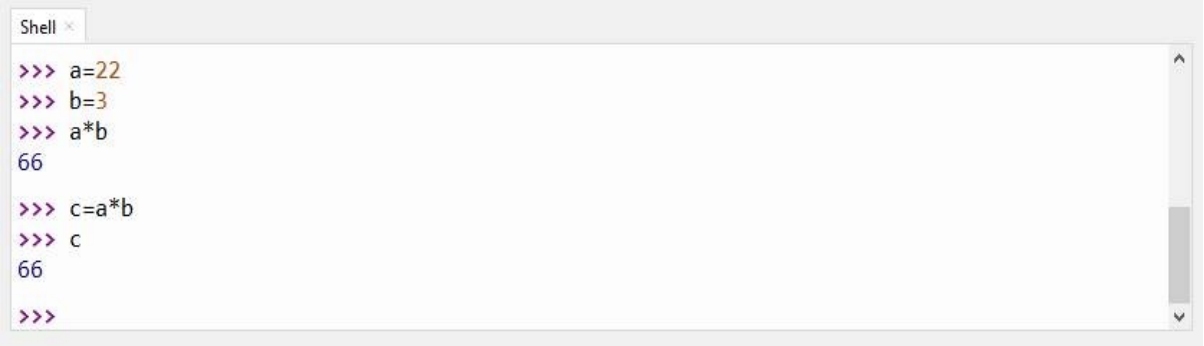

Abb. 3

Geben wir hinter dem Prompt eine Variable ein, wird nach dem Betätigen der Enter-Taste der Wert der Variable ausgegeben. Wie reagiert Micropython, wenn Sie auf diese Weise versuchen, den Wert einer Variablen auszugeben, der wir zuvor noch keinen Wert zugewiesen haben? Versuchen Sie es selbst aus!

In der Zeile `c = a*b` wird der Variablen `c` nicht die Zeichenfolge "`a*b`" zugewiesen. Vielmehr wird zunächst der Ausdruck `a*b` aus den zugewiesenen Werten berechnet und anschließend das Ergebnis dieser Rechnung in der Variablen `c` gespeichert.

Können auch Dezimalzahlen auf die gleiche Weise in Variablen gespeichert werden? Was geschieht mit den Variablen, wenn die Stopp-Schaltfläche (unterhalb der Menü-Zeile) oder `Ausführen > Soft-Reboot` angeklickt wird? Testen Sie es selbst aus!

Wenn Sie in Abb. 3 `c = A * b` statt `c = a * b` geschrieben hätten, wäre es zu einer Fehlermeldung gekommen. Micropython unterscheidet nämlich zwischen Groß- und Klein-Buchstaben. Der Fachman sagt: **Micropython ist case-sensitive**. Dies gilt nicht nur für Variablennamen, sondern generell!

Auch **Zeichenketten** (englisch: **Strings**) können in Variablen gespeichert werden. Damit eine Zeichenkette von Micropython nicht als Variablenname gedeutet wird, muss sie unbedingt mit Anführungszeichen markiert werden. Dabei ist es gleichgültig, ob sie dafür ‘ oder “ benutzen. Allerdings muss am Ende der Zeichenfolge dasselbe Anführungszeichen stehen wie am Anfang. Testen Sie selbst aus:

```
>>> s1 = 'Hallo'
>>> s2 = "Welt"
>>> s1
'Hallo'
>>> s1+s2
'HalloWelt'
```

Überlegen Sie einmal: Wie ist hier das Plus-Zeichen zwischen `s1` und `s2` zu deuten? Wie kann man dafür sorgen, dass zwischen den Wörtern Hallo und Welt in der letzten Zeile ein Leerzeichen steht?

Die Bedeutung des Plus-Zeichens hängt davon ab, von welchem Typ der Inhalt der Variablen ist, zwischen denen es steht. Bei vielen Programmiersprachen muss der Programmierer vor der Benutzung einer Variablen eine so genannte **Variablendeklaration** vornehmen, bei der der Typ der Variablen festgelegt wird. Eine solche Deklaration ist bei Micropython nicht erforderlich. Am Beispiel des Pluszeichens haben wir gesehen, dass Micropython hier selbstständig den Typ erkennt.

E.2 Mit Objekten arbeiten: LEDs ein- und ausschalten

An unser ESP32-Board schließen wir eine LED an Pin 25 wie in Abb. 4 dargestellt an. Es ist übrigens nicht gleichgültig, an welchen Pin sie angeschlossen wird, weil nicht alle Pins als Ausgänge benutzt werden können. Achten Sie auch bitte auf die Polung der LED!

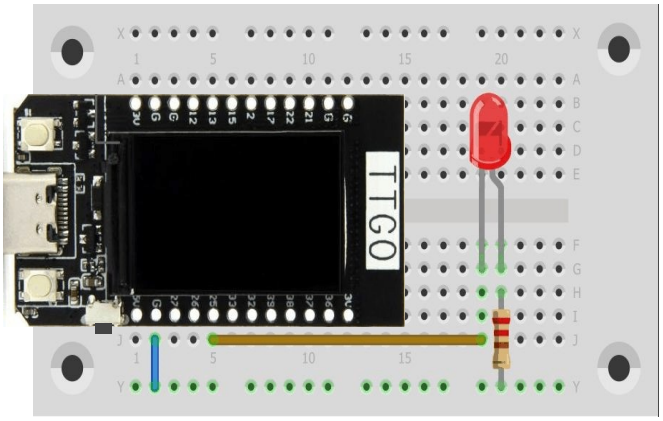

Abb. 4

Über das Terminal soll die LED nun ein- und ausgeschaltet werden. Im Internet finden wir die folgende Zeile zum Einschalten:

```
led.value(1)
```

Sie kommt uns etwas suspekt vor, weil nirgendwo eine Pin-Nummer auftaucht. Wir geben trotzdem den Befehl einmal im Terminal ein. Das Ergebnis ist in Abb. 5 zu sehen. Es macht klar: Die Angelegenheit ist wohl nicht ganz so einfach. Tatsächlich werden wir hier etwas weiter ausholen müssen; dabei werden wir mit wichtigen Konzepten bekannt gemacht: Module, Klassen, Objekte mit ihren Methoden und Eigenschaften. Das klingt erst einmal recht abstrakt. An unserem LED-Beispiel lässt sich aber gut verdeutlichen, wie man damit umgeht. Das wollen wir jetzt zeigen.

Abb. 5

Ganz wichtig ist zunächst: Standardmäßig hält Micropython nur eine relativ geringe Anzahl von Befehlen bereit. Ähnliche Situationen gibt es im alltäglichen Leben: In Ihrem Haus wird sich nicht zu jedem Thema ein passendes Buch finden. Was machen Sie? Sie besorgen sich aus einer Bücherei oder dem Internet ein oder mehrere "Dokumente" (Bücher, Zeitschriften, Webseiten…) mit den gewünschten Themen. Für alle Eventualitäten alle erdenklichen Dokumente immer zu Hause vorzuhalten wäre einfach unpraktisch.

Ähnlich verfahren wir bei Python: Es gibt sehr wohl Befehle, mit denen man LEDs kontrollieren kann. Man muss sie sich allerdings besorgen. Micropython besitzt eine Sammlung von Modulen; solch eine Sammlung wird auch eine Bibliothek (englisch: **library**) genannt. In der Firmware unseres ESP32 befindet sich bereits eine solche Bibliothek; je nach benutzter Firmware können die einzelnen Module sich in Umfang und Funktionsweise geringfügig unterscheiden.

Jedes **Modul** liefert uns nun Hilfsmittel zu einem bestimmten Thema. Das Modul, welches uns bei dem LED-Problem helfen kann, hat den Namen `machine`. Dieses Modul müssen wir uns aus der Bibliothek holen (**importieren**). Dies geschieht mit dem Kommando

```
import machine
```

Um uns einen Überblick über den Inhalt dieses Moduls zu verschaffen, lassen wir und das Inhaltsverzeichnis (directory) anzeigen:

```
dir(machine)
```

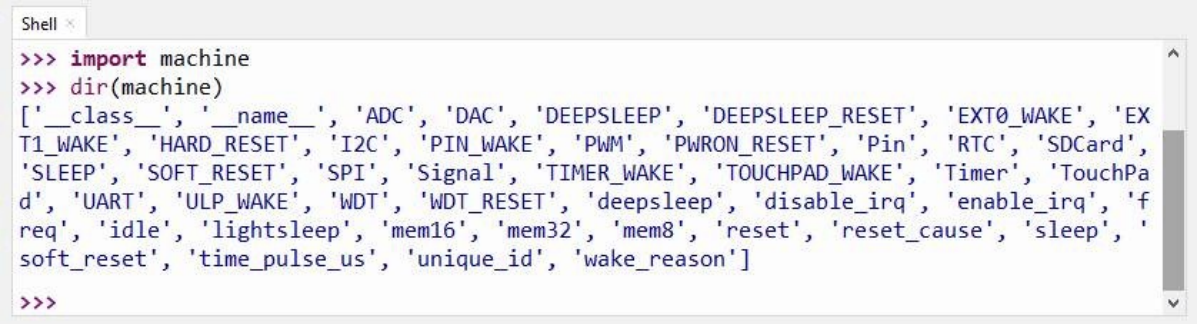

Abb. 6

Einige "Inhalte" wie ADC, I2C, PWM und UART kommen uns vielleicht bekannt vor. Am ehesten für unsere Zwecke geeignet scheint aber `Pin` zu sein; immerhin wollen wir ja unsere LED über einen Pin des Mikrocontrollers ansteuern. Was hat es aber damit auf sich?

Kann uns eine Internet-Recherche weiterhelfen? Mit den Stichworten "Micropython" und "pin" erhalten wir als ersten Link:

`https://docs.micropython.org/en/latest/library/machine.Pin.html`

Dort erfahren wir, dass es sich hier um eine Klasse handelt, die zur Kontrolle der I/O-Pins dient. Eine **Klasse** kann man sich als einen Bauplan vorstellen. Nach diesem Bauplan können **Objekte** (oft auch als **Instanzen** der Klasse bezeichnet) erzeugt werden. Beispielsweise wird durch die Befehlszeile

```
r1 = Rechteck(laenge, breite, ... )
```

eine Instanz der Klasse `Rechteck` mit den entsprechenden Werten für die Eigenschaften `laenge` bzw. `breite` erzeugt (vorausgesetzt, dass diese Klasse bereits existiert) und weist sie der Variablen `r1` zu.

Da in unserem Fall das Pin-Objekt zur Steuerung unserer LED dienen soll, wählen wir als Variablennamen `led`.

```
led = machine.Pin(25, machine.Pin.OUT)
```

Der erste Parameter legt hier die Nummer des Pins (GPIO-Nr) fest, der zweite gibt an, dass dieser Pin als Ausgang konfiguriert wird (`OUT` ist eine Eigenschaft der Klasse `Pin`). Damit Micropython "weiß", in welchem Modul die Klasse `Pin` zu finden ist, wird dem Klassennamen der Modulname vorangestellt, getrennt durch einen Punkt.

Die für uns wichtige **Methode** heißt `value`: Mit `led.value(1)` wird der Ausgang von Pin25 auf High gelegt; die LED geht dann an. Ausgeschaltet wird mit `led.value(0)`.

Das sollten Sie jetzt unbedingt austesten!

Wer möchte, importiert nicht das komplette `machine`-Modul, sondern nur die Klasse `Pin` aus diesem Modul. Das erreicht man mit der Anweisung:

```
from machine import Pin
```

Das hat u. A. den Vorteil, dass die Anweisung zum Erzeugen unseres Objekts `led` jetzt kürzer ausfällt:

```
led = Pin(25, Pin.OUT)
```

E.3 Programmieren

Für diesen Abschnitt stecken wir uns das folgende Ziel: Unsere LED aus dem letzten Abschnitt soll blinken. Und das ohne, dass wir selbst am Terminal dafür aktiv eingreifen. Dazu muss die LED selbstständig vom Mikrocontroller ein- und ausgeschaltet werden. Genauer gesagt muss der Mikrocontroller folgende "Befehle" der Reihe nach durchführen.

```
einschalten
warten
ausschalten
warten
einschalten
warten
ausschalten
warten
u.s.w
```

Eine derartige Abfolge von Befehlen bezeichnet man als **Programm**.

Micropython-Programme werden im Editor (vgl. Abb. 9 aus Kapitel V) eingegeben. Die benötigten Micropython-Befehle kennen wir zum großen Teil schon; was uns noch fehlt, ist ein Befehl, der den Mikrocontroller warten lässt. Dies geht mit der `sleep`-Funktion aus dem Modul `time`. Mit

```
sleep(1.5)
```

lassen wir den Mikrocontroller z. B. 1,5 s lang warten. Ohne solche Warte-Befehle würde so rasch ein- und ausgeschaltet, dass das Auge dies nicht mehr wahrnehmen kann.

Das Programm sieht nun so aus wie in Abb. 7. Wir öffnen ein neues Editor-Fenster und tippen es dort ein; dabei wird der Programmtext automatisch formatiert: Schlüsselwörter wie `from` und `import` werden z. B. fett angezeigt.

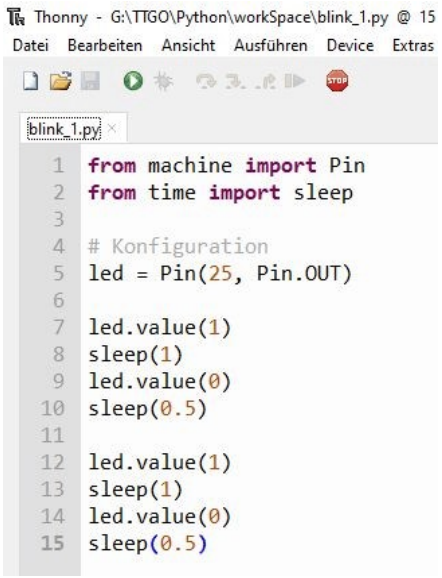

Abb. 7

Mit `Datei > Save as...` wollen wir es unter dem Namen `blink_1.py` abspeichern. Als Speicherort werden uns zwei Optionen angeboten: `This computer` oder `Micropython device`. Wir wählen die erste Möglichkeit und speichern dann das Python-Programm in einem Ordner unserer Wahl ab. Die Thonny-IDE fügt dabei automatisch die Extension `.py` an den Dateinamen an. Außerdem wird der benutzte Dateiname beim Speichern in den Reiter des Editierfeldes eingetragen.

Das Programm können wir mit `Ausführen > Run current script` starten. Einfacher geht das allerdings mit der Funktionstaste F5 oder mit der Schaltfläche  . Die LED blinkt nun zweimal auf!

Ein Blick hinter die Kulissen: Bei den meisten Programmierumgebungen wird das in einer Hochsprache wie z. B. Basic, C oder Pascal geschriebene Programm zunächst **kompiliert**; das dabei erzeugte **Maschinenprogramm** wird anschließend auf den Mikrocontroller geladen und kann dann dort gestartet werden. Bei unsere Thonny-Micropython-Umgebung läuft es hingegen so ab: Das in Micropython geschriebene Programm wird durch die REPL (s. o.) an das Micropython-System auf dem ESP32 übertragen und dort **interpretiert**; d. h. von dem Micropython-Quelltext wird ein erster Abschnitt in Bytecode übersetzt und gleich ausgeführt, anschließend wird auf dieselbe Weise mit dem nächsten Abschnitt verfahren u.s.w. Daran würde sich auch nichts wesentlich ändern, wenn wir den Micropython-Quelltext auf dem ESP32 gespeichert hätten.

Jetzt wollen Sie sicherlich die LED häufiger blinken lassen. Natürlich kann man die vier Zeilen für das Ein- und Ausschalten (inkl. Pausen) noch einige Male unten im Programm anfügen. Es geht aber auch eleganter mit einer **Endlos-Schleife**:

```
while True:
      led.value(1)
      sleep(1)
      led.value(0)
      sleep(0.5)
```

Beachten Sie: Sobald Sie hinter dem Doppelpunkt die Enter-Taste gedrückt haben, springt der Cursor in die nächste Zeile und rückt dabei ein. Alle vier Zeilen des Schleifen-Körpers müssen auf die gleiche Weise eingerückt werden. Später werden wir auf die Bedeutung dieses Einrückens (engl.: indenting) noch ausführlich eingehen.

Speichern Sie das neue Programm unter dem Namen `blink_2.py` ab und starten Sie es. Es lässt die LED solange blinken, bis Sie das Programm unterbrechen, z. B. indem Sie die Stopp-Schaltfläche betätigen.

Nun wollen wir, dass der ESP32 das Blink-Programm auch ohne Rechner ausführt. Dafür speichern wir das Programm auf dem ESP32 unter dem Dateinamen `main.py` ab. Der Grund dafür ist folgender: Nach einem Hard-Reset (z. B. RST-Taste betätigt oder den ESP32 gerade an elektrische Quelle angeschlossen) wird automatisch die Datei `boot.py` und anschließend die Datei `main.py` auf dem ESP32 ausgeführt.

Nachdem wir unser Blink-Programm unter dem Namen `main.py` auf dem ESP32 gespeichert haben, lassen wir uns mit `Ansicht > Dateien` die Dateien auf dem ESP32 und auch auf dem Rechner in getrennten Fenstern anzeigen (vgl. Abb. 8). Diese beiden Fenster besitzen zwar keine Drag&Drop-Funktionen; über die rechte Maustaste lässt sich aber jeweils ein Kontext-Menü öffnen. Mit diesem kann man z. B. auch Dateien vom Rechner auf den ESP32 übertragen.

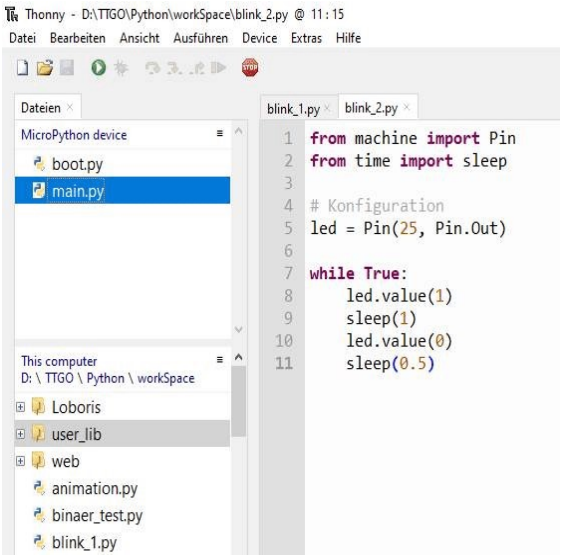

Abb. 8

Nun können wir einmal den Reset-Knopf an dem Board betätigen. Im Terminal erkennen wir einige Aktionen, die dabei ausgeführt werden. Anschließend beginnt die LED zu blinken, ohne dass wir das Programm explizit starten mussten.

Wer eine Powerbank zur Verfügung hat, trennt einmal das ESP32-Board vom Rechner und schließt es stattdessen an die Powerbank an. In dem Augenblick, in dem das Board wieder mit Strom versorgt wird, werden automatisch auch `boot.py` und `main.py` ausgeführt und unsere LED blinkt - diesmal auch ohne Kontakt mit dem Rechner.

E.4 Vergleichen und Verzweigen

Wir bleiben bei unserer LED: Diesmal soll sie nicht selbstständig blinken, sondern mit einem der Taster unseres Boards geschaltet werden. Wir wählen den linken Taster in der Abb. 1 aus dem Kapitel V. Ganz bewusst benutzen wir den Ausdruck ***Taster*** und nicht *Schalter*; denn wie bei einer Türklingel soll die LED nicht dauerhaft ein- oder ausgeschaltet werden; vielmehr soll die LED dann und nur dann leuchten, wenn der Taster gedrückt ist.

In Abb. 9 sehen wir, wie dieser Taster mit dem Pin `GPIO0` verbunden ist. Wegen des Pullup-Widerstands liegt am Punkt X ein hohes Potential (high) an, wenn der Taster nicht gedrückt ist. Solange er gedrückt ist, liegt dort ein niedriges Potential (low) an. Ob hohes oder niedriges Potential vorliegt, kann über den Pin `GPIO0` ermittelt werden. Dazu muss dieser Pin jetzt als Eingang konfiguriert werden.

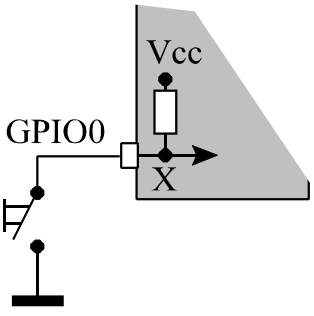

Abb. 9

Beim Erzeugen unseres Pin-Objekts muss der zweite Parameter deswegen jetzt nicht `Pin.OUT`, sondern `Pin.IN` lauten. Außerdem muss der interne Pullup-Widerstand an diesem Pin "aktiviert" werden; dies geschieht mit einem dritten Parameter `Pin.PULL_UP`. Das erzeugte Objekt nennen wir jetzt sinnvollerweise `taster`.

```
taster = Pin(0, Pin.IN, Pin.PULL_UP)
```

Ob der Zustand an dem Pin nun high oder low ist, kann mit der Methode `value` ermittelt werden:

```
zustand = taster.value()
```

Entscheidend ist hier, dass hier die Methode `value` kein Argument besitzt. In diesem Fall wird an dem Pin der Zustand High oder Low nicht gesetzt, sondern er wird gemessen und als Rückgabewert in der Variable zustand gespeichert. Liegt am Pin der Zustand High vor, ist dieser Rückgabewert 1, ansonsten 0.

Unser Programm muss nun immer wieder den Zustand am Taster bestimmen. Wenn dann `zustand` gleich 0 ist, dann muss die LED eingeschaltet werden, ansonsten muss sie ausgeschaltet werden. Das Micropython-Programm sieht dann so aus:

```
from machine import Pin

led = Pin(25, Pin.OUT)
taster = Pin(0, Pin.IN, Pin.PULL_UP)
while True:
      zustand = taster.value()
      if zustand == 0:
            led.value(1)
      else:
            led.value(0)
```

Die Grundstruktur der **Verzweigung** ist

```
if <Bedingung>:
      Block für den Wenn-Teil
else:
      Block für den Sonst-Teil
```

In vielen Programmiersprachen werden Anfang und Ende eines Blockes durch spezielle Klammern oder Schlüsselwörter wie `begin` und `end` markiert. In Micropython werden **Blöcke** lediglich durch **Einrücken** gekennzeichnet: Alle Befehle **desselben** Blockes werden **auf die dieselbe Weise eingerückt**. In unserem Fall befindet sich der Block für den Wenn-Teil innerhalb des Blockes für die Endlos-Schleife. Der untergeordnete Block (hier z. B. Wenn-Teil) muss dann weiter nach rechts eingerückt sein. Wie wir bereits bei der Endlos-Schleife im letzten Abschnitt festgestellt haben, unterstützt uns die Thonny-IDE beim Einrücken.

Die Bedingung, nach der das Programm seine Entscheidung fällen soll, besteht meist aus einem Vergleich, z. B.

```
if wert > 8:
if x == y:
```

Bitte beachten Sie: Beim **Überprüfen auf Gleichheit** benutzt man ein **doppeltes Gleichheitszeichen**; das einfache Gleichheitszeichen ist für die Wertzuweisung reserviert! Weitere Vergleichsoperatoren sind >=, < und <=.

Allgemein kann als Bedingung ein Wahrheitswert oder ein entsprechender Ausdruck angegeben werden. Solche Ausdrücke können z. B. auch Verknüpfungen von Wahrheitswerten sein:

```
if a>5 and b:
```

Micropython wertet dabei zunächst die einzelnen Ausdrücke aus und weist ihnen Wahrheitswerte zu, True oder False. Diese Wahrheitswerte werden dann zu einem einzigen Wahrheitswert verknüpft, z. B. `True and False` → `False`. (Für das Ergebnis bewertet Micropython dabei nicht unbedingt alle einzelnen Vergleiche, sondern verwendet eine so genannte "Kurzschluss-Auswertung".)

Ein Blick hinter die Kulissen: Zwar müssen wir bei Micropython Variablen nicht deklarieren, das bedeutet aber nicht, dass es bei Micropython keine unterschiedlichen Datentypen gibt. Bislang haben wir folgende (primitive) **Datentypen** kennen gelernt:

Variablentyp	Bedeutung
int	ganze Zahlen (integer)
float	Fließkommazahlen
string	Zeichenketten
bool	Wahrheitswerte (Boolesche Werte): True, False

An dieser Stelle sollte auch klar werden, wie unsere Endlos-Schleife funktioniert. Sie stellt einen Spezialfall dar von:

```
while <Bedingung>:
```
`Block` (welcher solange wiederholt wird, wie die Bedingung den Wert True hat)

Wenn nun die Bedingung nur aus dem Wert True besteht, wird der Block endlos wiederholt.

Überlegen Sie nun einmal, was die beiden folgenden Programme machen. Testen Sie es ggf. mit Ihrem ESP32-Board aus.

```
# Programm 1
from machine import Pin
from time import sleep

led = Pin(25, Pin.OUT)
taster = Pin(0, Pin.IN, Pin.PULL_UP)

while taster.value() == 1:
      led.value(1)
      sleep(0.1)
      led.value(0)
      sleep(0.1)


# Programm 2
from machine import Pin

led = Pin(25, Pin.OUT)
taster = Pin(0, Pin.IN, Pin.PULL_UP)
while True:
      led.value(1-taster.value())
```

E.5 Listen, Tupel, Strings und Iteratoren

Im nächsten Kapitel wollen wir das ESP32-Board nach Wlans scannen lassen. Ohne hier schon auf eine inhaltliche Analyse einzugehen, schauen wir uns das Ergebnis eines solchen Scans an (s. Abb. 10). Uns geht es in diesem Abschnitt darum, die einzelnen dort auftauchende Strukturen kennen zu lernen.

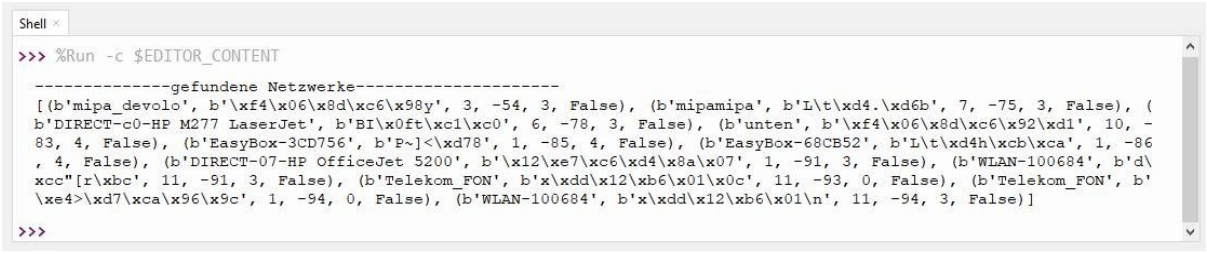

Abb. 10

Beginnen wir mit den so genannten Listen. Durch

```
quadratzahlen = [4, 9, 16, 25, 36, 49]
```

wird in der Variablen `quadratzahlen` eine Liste von 6 Quadratzahlen abgespeichert. **Listen**

bestehen aus einer (geordneten) Reihe von Daten; diese werden auch **Elemente** genannt. Die Elemente einer Liste stehen in **eckigen Klammern** und werden jeweils durch ein Komma getrennt; die Elemente einer Liste müssen nicht unbedingt denselben Datentyp besitzen.

Entscheidend ist, dass wir mit Hilfe eines Index auf die einzelnen Elemente zugreifen können; der Index nummeriert die einzelnen Elemente der Reihe nach durch; dabei beginnt die Zählung bei 0. Bei unserer Liste `quadratzahlen` erfolgt der Zugriff auf das Element mit dem Index 4 z. B. durch

```
quadratzahlen[4]
```

Das Ergebnis ist in diesem Fall 36. Nicht nur auf einzelnen Elemente der Liste kann man zugreifen, sondern auch auf **Teillisten**: Die Eingabe

```
quadratzahlen[2:5]
```

liefert z. B. die folgende Teilliste:

```
[16, 25, 36]
```

Beachten Sie: Es werden hier Elemente aus der Liste `quadratzahlen` "herausgeschnitten", und zwar vom Index 2 (**einschließlich**) bis zum Index 5 (**ausschließlich**).

Mit einer **Schleife** kann man die Elemente einer Liste zeilenweise ausgeben (oder auch anderweitig bearbeiten):

```
i = 0
while i<6:
    print(quadratzahlen[i])
    i += 1
```

Die letzte Zeile ist eine abkürzende Schreibweise für die Anweisung `i=i+1`; der Index `i` wird hier also um 1 erhöht. Mit den folgenden zwei Zeilen lässt sich die zeilenweise Ausgabe allerdings deutlich einfacher gestalten:

```
for quadratzahl in quadratzahlen:
    print(quadratzahl)
```

Ein wichtiger Vorteil hierbei ist auch, dass die Anzahl der Schleifendurchläufe nicht explizit bestimmt werden muss.

Wegen des Schlüsselwortes `for` spricht man hier auch von einer `for`-Schleife. Anders als bei manchen anderen Programmiersprachen taucht hier ein Index nicht (explizit) auf, vielmehr wird direkt mit den Elementen gearbeitet, die hier durch die **Schleifenvariable** `quadratzahl`

repräsentiert werden. Hinter dem Schlüsselwort `in` müssen sogenannte **iterierbare** Objekte stehen; dazu gehören z. B. Listen und Strings.

> *Ein Blick hinter die Kulissen:* Durch das Schlüsselwort `for` wird zu unserer Liste `quadratzahlen` ein neues Objekt erzeugt, **Iterator** genannt. Dieser Iterator besitzt einen Zeiger, über den auf die einzelnen Elemente unserer Liste zugegriffen werden kann. Bei jedem Durchlauf der `for`-Schleife wird dieser Zeiger automatisch auf das nächste Element der Liste gerichtet; dieses Element wird sodann der Schleifenvariablen `quadratzahl` zugewiesen. Das geschieht so lange, bis das Ende der Liste erreicht ist; dann wird die Schleife automatisch abgebrochen.

Ein ähnlicher Datentyp wie die Liste ist das **Tupel**: Äußerlich unterscheiden sich Tupel von Listen dadurch, dass sie nicht mit eckigen, sondern mit **runden Klammern** gekennzeichnet werden. Im Gegensatz zu den Listen sind die Elemente eines Tupels **nicht veränderbar**. Ansonsten lassen sie sich aber wie Listen behandeln. Insbesondere kann auch bei ihnen ein Iterator zur Anwendung kommen.

Dasselbe wie für Tupel gilt auch für **Zeichenketten (Strings)**: Versuchen Sie einmal selbst von der Zeichenkette 'Hallo Welt!' einen Teil auszuschneiden oder die einzelnen Zeichen zeilenweise auszugeben.

Werfen wir noch einmal einen Blick auf Abb. 10, so entdecken wir Objekte, die Zeichenketten sehr ähnlich sind; allerdings steht vor dem ersten Anführungszeichen jeweils ein `b`. Beispiele sind `b'mipa_devolo'` und `b'\xf4\x06\x8d\xc6\x98y'`. Hier handelt es sich um Objekte vom Datentyp `bytes`; manche bezeichnen sie auch als **Byte-Strings**. Ein solches Objekt besteht aus einer **Folge von Bytes**. Es besteht **nicht aus einer Folge von Zeichen**, auch wenn einzelne Bytes auf dem Terminal manchmal in Form von Zeichen dargestellt werden. Dies ist genau dann der Fall, wenn es sich bei dem Byte gerade um den Code eines *darstellbaren* Zeichens des Standard-ASCII-Zeichensatzes handelt: Der Code 65 steht im ASCII-Zeichensatz für das Zeichen "A". Taucht das Byte 65 in einem Byte-String auf, wird es also auf dem Terminal mit dem Zeichen "A" dargestellt. Der Code 6 steht für ein so genanntes Steuerzeichen, nämlich für das Signal ACK ("Acknowledge", das bedeutet: "Verstanden"). Solche Steuerzeichen sind nicht darstellbar. Die Standard-ASCII-Zeichensatz-Tabelle besitzt nur Codes zwischen 0 und 127. Bytes mit einem größeren Wert können somit auch nicht durch ein Standard-ASCII-Zeichen dargestellt werden.

Wenn nun ein Byte größer als 127 ist oder zu einem nicht darstellbaren Zeichen gehört, dann stellt Micropython es auf dem Terminal in **hexadezimaler Form** dar. Dabei wird die Hex-Zahl durch `\x` eingeleitet. Im Beispiel `b'\xf4\x06\x8d\xc6\x98y'` ist das erste Byte größer als 127; es kann also nicht durch ein Zeichen dargestellt werden. Das zweite Byte hat den Wert 6; wir wissen schon, dass es einem Steuerzeichen entspricht. Beide Bytes werden daher hexadezimal dargestellt. Lediglich das letzte Byte in diesem Beispiel gehört zu einem darstellbaren Zeichen (nämlich "y") und wird dann entsprechend auf dem Terminal ausgegeben.

Selbstverständlich kann man sämtliche Bytes hexadezimal eingeben, auch wenn es sich um darstellbare ASCII-Codes handelt: Gibt man am Terminal `print(b'\x41\x42\x43')` ein, wird jedoch `b'ABC'` ausgegeben. Dies ist für den Nutzer natürlich viel einfacher zu lesen als die Folge von Hexadezimalzahlen.

Die einzelnen Bytes, aus denen sich ein Bytes-Objekt zusammensetzt, kann man auch in eine Listenform bringen und umgekehrt; dazu benutzt man die Funktionen `list()` bzw. `bytes()`. Die Eingabe

```
>>> list(b'ABC')
```

hat z. B. das Ergebnis:

```
[65, 66, 67]
```

Umgekehrt kann man mit der `bytes`-Funktion eine Liste von einzelnen Bytes in ein bytes-Objekt umwandeln. Das Ergebnis von `bytes([72, 97, 108, 108, 111])` ist z. B. `b'Hallo'`.

Ganz lehrreich ist es, Bytes-Objekte mit deutschen Sonderzeichen wie z. B. `b = b'Käse'` zu betrachten. Wenn man die Länge dieses Objekts mit `len(b)` bestimmen lässt, erhält man den Wert 5. Auf den ersten Blick scheint das widersprüchlich zu sein, besteht das Wort 'Käse' doch nur aus 4 Buchstaben. Lassen wir uns das Objekt noch einmal anzeigen, erkennt man, dass das Objekt in Wirklichkeit so aussieht:

```
b'K\xc3\xa4se'
```

Die zugehörige Liste ist `[75, 195, 164, 115, 101]`. Offensichtlich wird das **Zeichen** "ä" hier **durch zwei Zahlen kodiert**, nämlich 195 = $C3 und 164 = $A4. Das ist die **UTF-8-Kodierung** des Zeichens "ä". Mehr Informationen dazu findet man z. B. unter https://www.utf8-zeichentabelle.de/ und https://realpython.com/python-encodings-guide/.

Den Datentyp `bytes` darf man nicht mit dem Datentyp `string` verwechseln. Einen ersten Unterschied erkennt man, wenn man von dem String `s = 'Käse'` mit `len(s)` die Länge bestimmen lässt. Hier ist der Wert 4; d. h. bei einem string-Objekt sind die **Zeichen** selbst relevant und nicht die **Codes**, mit denen sie gespeichert werden. Mit der `bytes`-Funktion lässt sich unser String `s` in den Typ `bytes` (unter Angabe der benutzten Zeichen-Kodierung) umwandeln:

```
b = bytes(s, 'UTF-8')
```

`b` hat jetzt wie erwartet den Inhalt `b'K\xc3\xa4se'`. Um eine Variable `b` vom Datentyp `bytes` in einen String umzuwandeln, wendet man die `str`-Funktion an:

```
s = str(b, 'UTF-8')
```

Schauen wir uns Abb. 10 ein letztes Mal an: Die beiden eckigen Klammern "[" und "]" tauchen jeweils nur einmal auf. Hier liegt demnach auch nur eine einzige Liste vor. Die Elemente dieser Liste bestehen aber ihrerseits aus Tupeln. Und die Elemente dieser Tupel besitzen verschiedene Datentypen. Insgesamt liegt also schon eine recht komplexe Struktur vor. Die Bedeutung der einzelnen Elemente werden wir im nächsten Kapitel klären.

E.6 Funktionen

Wenn eine Folge von bestimmten Befehlen im Verlauf eines Programms häufiger benutzt wird, bietet es sich an, diese zu einer Einheit zusammenzufassen. Eine solche Einheit wird in Micropython als **Funktion** bezeichnet. Die folgende Funktion `summe` soll als Beispiel für die Definition einer Funktion dienen:

```
def summe(liste):
  s = 0
  for l in liste:
    s += l
  return s
```

Sie bestimmt zu einer Liste, die als Parameter übergeben wird, deren Summe; diese Summe wird mit dem Schlüsselwort `return` zurückgegeben. Schreiben wir unter diese Funktion jetzt den Befehl `print(summe([12, 5, 18]))`, erhalten wir im Terminal als Ausgabe die Zahl 35.

Drei wichtige Bemerkungen noch zum Abschluss dieses kurzen Abschnitts über Funktionen:

- Wenn eine Funktion keine Parameter besitzt, werden die Klammern trotzdem hingeschrieben; sie bleiben dann leer.

- Funktionen können auch ohne Rückgabewert auskommen. In manchen Programmiersprachen würde man sie dann als Prozeduren bezeichnen. Mikropython macht hier keinen Unterschied: Wenn kein Rückgabewert erforderlich ist, lässt man einfach die `return`-Zeile weg.

- Alle Variablen, welche innerhalb der Definition einer Funktion auftauchen, sind (standardmäßig) **lokal**; von außerhalb kann man also nicht auf deren Werte zugreifen.