# Python & MicroPython: Programmieren lernen für Einsteiger

## Ablaufsteuerung und Programmstrukturen

Teil 4

**Bislang lag der Hauptfokus dieser Artikelserie auf der Ansteuerung von Hardware. Die Kontrolle von IO-Pins und das Blinken von LEDs können in Python mit wenigen, nahezu selbsterklärenden Anweisungen umgesetzt werden. Allerdings kommt man irgendwann an den Punkt, an dem auch tiefere Programmierkenntnisse erforderlich sind. Selbst wenn heute die Programmiertätigkeiten häufig darin bestehen, passende Programmteile oder universelle Bibliotheken zusammenzufügen, ist immer ein gewisses Grundverständnis der elementaren Programmstrukturen erforderlich. Andernfalls lassen sich komplexere Projekte kaum realisieren.**

In diesem Beitrag sollen die folgenden Programmstrukturen näher beleuchtet werden:

- Kommentare
- Print()
- Einrückungen und Blöcke
- Elementare Programmstrukturen wie if, while, for usw.
- Definition von Funktionen

Damit werden dann auch Anweisungen und Befehle klar, die in früheren Beispielen bereits erforderlich waren, aber noch nicht im Detail diskutiert wurden.

## Befehle und Anweisungen im Detail

Im Gegensatz zum klassischen C/C++, bei dem ein Quellcode erst nach einem Kompilierungsprozess als ausführbarer Code zur Verfügung steht, wird Python interpretiert. Der Code wird also unmittelbar ausgeführt, d. h., es wird keine für das betreffende System ausführbare Datei generiert. Das geht natürlich zulasten der Ausführungsgeschwindigkeit. In Python kann man zwei unterschiedliche Varianten wählen, um den Interpreter aufzurufen:

- Interaktiver Modus
- Skript-Modus

Im interaktiven Modus gibt man die Befehle direkt ein und erhält nach Betätigung der Return-Taste unmittelbar das Ergebnis. Den interaktiven Modus erkennt man an den drei Größer-als-Zeichen (>>>) in Thonny (Bild 1). Hier werden Eingaben direkt verarbeitet:

```
>>> print("Good Morning, Dr. Falken")
Good Morning, Dr. Falken
```

Auch eine LED (z. B. am Port 24 eines Raspberry Pi) kann auf diese Weise ein- bzw. ausgeschaltet werden (Bild 2). Das kann sehr nützlich sein, um Hardware schnell und sicher zu überprüfen, ohne dass ein spezielles Programm erstellt werden muss.

Umfangreichere Programme dagegen werden im Programmfenster eingegeben. Sie können mit der Taste F5 gestartet werden (Bild 3). Dies ist das übliche Vorgehen, wenn längere Code-Sequenzen abgearbeitet werden müssen.

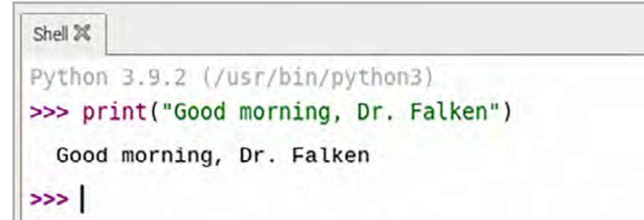


Bild 1: Arbeiten in der Shell

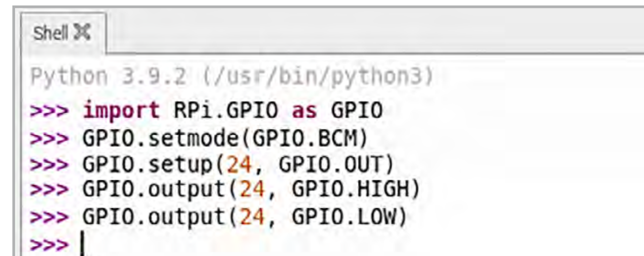


Bild 2: Schalten einer LED über die Shell

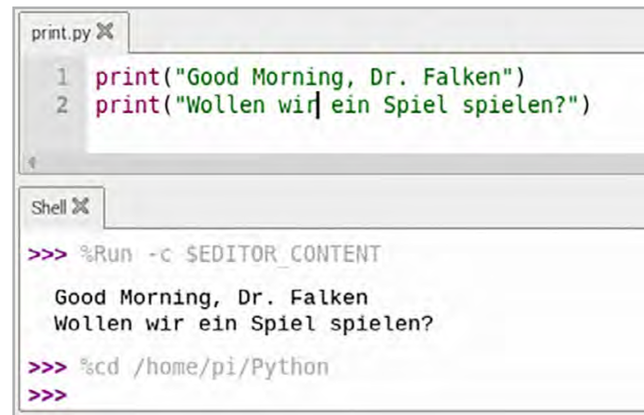


Bild 3: Abarbeiten eines Programms im Editor

## Kommentare helfen, Programme zu verstehen

In jeder Programmiersprache sind erläuternde Kommentare wichtig. Nur so weiß man auch Monate oder Jahre später noch, was ein bestimmter Programmabschnitt bewirkt, ohne dass man sich jedes mal wieder neu in alle Details vertiefen muss.

Es ist nicht notwendig, jede Programmzeile einzeln zu kommentieren. Erfahrene Codierer sollten in der Lage sein, einzelne Anweisungen auch ohne Kommentar zu verstehen. Nur bei besonderen Konstrukten oder ungewöhnlichen bzw. innovativen Code-Zeilen empfiehlt sich ein einzelner Zeilenkommentar. Bei Unterprogrammen oder ganzen logischen Programmabschnitten dagegen darf eine kurze Erläuterung zur Funktionsweise nicht fehlen.

Einfache Kommentare beginnen mit # und enden mit dem Zeilenende:

```
>>> print("hello RasPi")      # this is a comment

hello RasPi
```

Mehrzeilige Kommentare können auch mit einem 3-fachen "-Zeichen (""") gekennzeichnet werden. Dieselbe Zeichenfolge beendet dann den Kommentar:

```
"""
erste Kommentarzeile
zweite Kommentarzeile
"""
```

In der Praxis kann das so aussehen:

```
'''
This is a multi-line comment.
Prints hello world.
'''
print("hello world")
```

Alternativ kann in Thonny auch die Kommentarfunktion verwendet werden. Sie erlaubt es, mehrere Zeilen gleichzeitig mit dem #-Zeichen als Kommentare zu kennzeichnen (Bild 4).

Die Kommentarfunktion eignet sich auch sehr gut dazu, bestimmte Programmteile „auszukommentieren“. Wenn beispielsweise beim Test eines umfangreicheren Codes einzelne Abschnitte probeweise nicht ausgeführt werden sollen, kann man diese mit Kommentarzeichen versehen. Die Zeilen werden dann vom Interpreter nicht mehr beachtet. Ein aufwendiges Löschen und späteres Wiedereinfügen der Programmteile entfällt.

Kommentare helfen insbesondere auch Einsteigern, ein besseres Verständnis für den Programmaufbau zu erlangen. Von der technischen Seite her gesehen haben sie keinerlei Einfluss auf den Programmablauf.

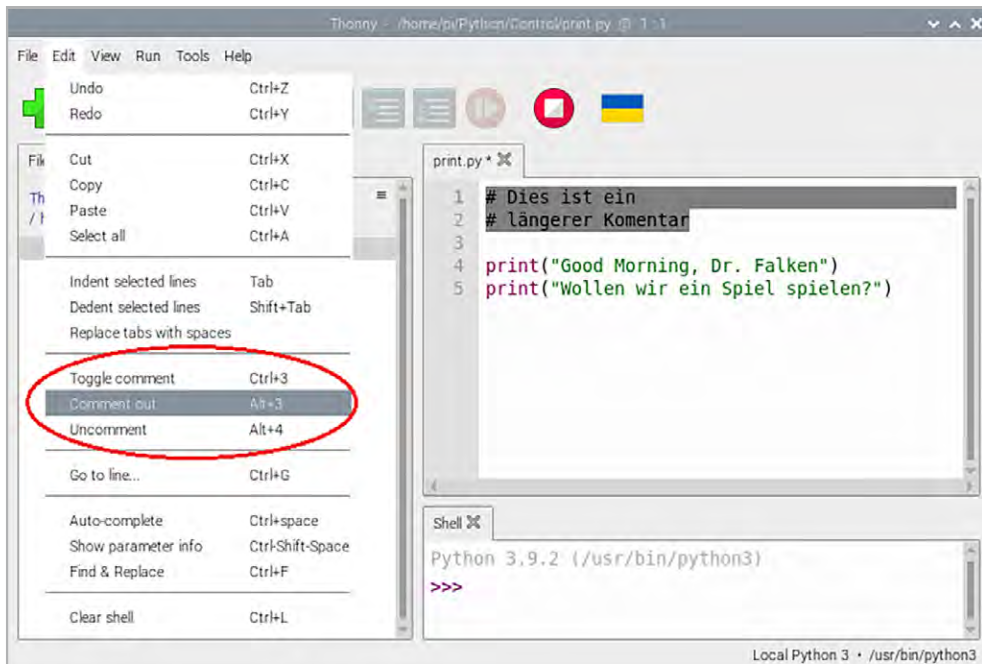


*Bild 4: Kommentarfunktion in Thonny*

## Wie gedruckt: Die Print()-Anweisung

Über die Print()-Anweisungen können Informationen auf das Terminal ausgegeben werden. Der Befehl ist zum einen direkt im Terminal ausführbar, zum anderen dient er in Programmen zur Ausgabe textbasierter Informationen.

Print() kann mehrere Zeichenfolgen haben, die durch "," geteilt werden:

```
>>> print("hello", "world!")
hello world!
```

Die Print()-Anweisung führt standardmäßig einen Zeilenwechsel aus. Mit

```
end = ""
```

kann dieser unterdrückt werden:

```
print("hello", end=" ")
print("world")
```

Dies führt zu folgendem Ergebnis:

```
hello world
```

## Klare Struktur: Einrückungen und Blöcke

Python unterscheidet verschiedene Blöcke durch Einrückung. Es ist nicht erforderlich, geschweifte Klammern („{}“) oder Ähnliches zu verwenden. Dies ist einer der wichtigsten Unterschiede zu den meisten anderen Sprachen wie C, Pascal, Basic etc. Der Vorteil dieser Methode ist, dass man zu einem gewissen Maß an Programmstruktur gezwungen wird:

```
if True:
  # block 01
  print ("True")
else:
  # block 02
  print ("False")
```

Die Anzahl der Leerzeichen für Einrückungen ist variabel, aber derselbe Block muss immer die gleiche Anzahl an Leerzeichen für Einrückungen haben. Andernfalls wird eine Fehlermeldung ausgegeben:

```
if True:
  print ("Answer")
  print ("True")
else:
 print ("Answer")
    print ("False")   # The different indentation leads to a runtime error
```

Dies führt zu folgendem Hinweis:

```
>>> %Run -c $EDITOR_CONTENT
Traceback (most recent call last):
  File "<stdin>", line 6
IndentationError: unexpected indent
```

In Bedingungen und Schleifen (s. u.) werden Blöcke auf identische Art gebildet.

## Zeitsteuerung und Sleep

Im Beispiel zur blinkenden LED wurde bereits die Delay-Anweisung verwendet. Diese ist im Modul „time“ enthalten und wird über

```
import time
```

verfügbar. Mit der Anweisung

```
time.sleep(seconds)
```

kann eine feste Verzögerungszeit in Sekunden eingestellt werden. Alternativ kann auch nur der Sleep-Befehl selbst importiert werden:

```
from time import sleep
```

Dann kann die Anweisung zu

```
sleep(seconds)
```

verkürzt werden. Obwohl der Befehl auch für Sekundenbruchteile genutzt werden kann, empfiehlt es sich, für sehr kurze Verzögerungszeiten die Anweisung

```
time.sleep_ms(Milliseconds)
```

zu verwenden, da diese für kleine Zeiten eine verbesserte Präzision aufweist.

Der Nachteil dieser Funktionen ist, dass sie blockierend arbeiten. Das heißt, der Controller kann während der Wartezeit keine anderen Aufgaben ausführen, da er mit dem Zählen von Prozessorzyklen ausgelastet ist. Als Alternative bietet sich die Verwendung von Interrupts oder anderen Programmiertechniken an. Details dazu finden sich in späteren Beiträgen zu dieser Reihe.

Für Zeitabfragen stehen die beiden folgenden Routinen zur Verfügung:

```
time.ticks_ms()
time.ticks_us()
```

Sie geben die aktuelle Systemlaufzeit in Milli- bzw. Mikrosokunden an. Eine klassische Anwendung ist die Messung von Programmlaufzeiten. Mit dem folgenden Code (math.py) kann man so z. B. zeigen, dass mathematische Operationen einen gewissen Zeitaufwand erfordern:

```
import time
import math

while(True):
 start = time.time()
 time.sleep(1)
 #x = math.cos(math.tan(math.exp(math.sin(22.5))))
 stop = time.time()
 print((stop-start-1)*1e6, "µs")
```

Das Programm liefert auf einem Rasperry Pi 4 Laufzeiten von
1073,1220245361328 µs
1065,969467163086 µs
1074,3141174316406 µs

Wird das Kommentarzeichen vor der Berechnung entfernt, steigt die angezeigte Ablaufgeschwindigkeit an auf
1102,447509765625 µs
1122,9515075683594 µs
1103,8780212402344 µs

Die Laufzeit nimmt also um ca. 30 µs zu. Die Berechnung der Funktion

math.cos(math.tan(math.exp(math.sin(22.5))))

beansprucht also einen nicht zu vernachlässigenden Zeitrahmen.

Oftmals findet man in Microcontroller-Programmen statt des Moduls „time" auch die „utime"-Bibliothek. Beide sind in Python z. B. für den Raspberry Pi Pico verfügbar und werden für die Zeitsteuerung in Programmen verwendet.

Der Hauptunterschied zwischen den beiden Modulen besteht darin, dass „utime" eine Low-Level-Zeitbibliothek ist, die speziell für Mikrocontroller-Plattformen wie den Raspberry Pi Pico optimiert wurde. Time hingegen ist eine Standard-Python-Zeitbibliothek, die auch auf größeren Computer-Plattformen wie PCs verfügbar ist.

Ein weiterer Unterschied besteht darin, dass „utime" eine höhere Genauigkeit bietet als „time", da es auf dem internen Systemtakt des Raspberry Pi Pico basiert.

Für einfache Anwendungen können jedoch beide Module gleichberechtigt eingesetzt werden.

## Wichtige Werte: Variablen und Konstanten

In Python ist es besonders einfach, Variablen zu erstellen. Es ist nicht erforderlich, den Datentyp der Variablen während der Zuweisung anzugeben. Dies ist ein wesentlicher Unterschied zu anderen Sprachen. Dort müssen Variablen stets explizit mit einem bestimmten Typen initialisiert werden (z. B. int a = 1234...).

Variablen sind auch in der Konsole einsetzbar:

```
>>> a = 17
>>> print(a)
17
```

Für Variablennamen sind die 26 Grundbuchstaben, also die Buchstaben von A bis Z ohne deutsche Sonderzeichen, verwendbar. Diese Grundbuchstaben können sowohl als Klein- als auch als Großbuchstaben eingesetzt werden. Wie in vielen professionellen Programmiersprachen wie C/C++, Java oder auch C# wird auch in Python bei Funktionen, Methoden, Klassen, Schlüsselwörtern etc. zwischen Klein- und Großschreibung unterschieden.

Der Fachbegriff dafür lautet „case-sensitive". Über denn Sinn gleichlautender Variablennamen, die sich nur in der Klein- bzw. Großschreibung von Buchstaben unterscheiden, kann man streiten. Die Erfahrung lehrt jedoch, dass die Verwendung gleicher Namen in unterschiedlichen Schreibweisen keine Vorteile bringt, sondern eher zu Fehlern führt.

In Python gelten für die Vergabe von Variablennamen diese Regeln:

- Der Variablenname darf nur Zahlen, Buchstaben und Unterstriche enthalten
- Das erste Zeichen einer Variablen muss ein Buchstabe oder ein Unterstrich sein
- Der Variablenname unterscheidet zwischen Groß- und Kleinschreibung

Variablen können Werte verschiedener Typen zugewiesen werden. Die Typen in MicroPython umfassen Zahlen, Zeichenfolgen, Listen, Wörterbücher, Tupel usw. Mit type() kann der Datentyp von Variablen und Konstanten überprüft werden, z. B.

```
>>> a = 17
>>> print(type(a))
<class 'int'>

>>> a, b, c, d = 17, 1.5, True, 5+7j
>>> print(type(a), type(b), type(c), type(d))

<class 'int'> <class 'float'> <class 'bool'> <class
'complex'>
```

Zahlen wie 10, 100 oder Zeichenfolgen wie „Hello World!" sind Konstanten.

MicroPython bietet das Schlüsselwort „const" an, mit dem auch einer Variablen ein unveränderlicher Wert zugewiesen wird:

```
from micropython import const

a = const(33)
```

Tabelle 1 zeigt einige Beispiel für erlaubte und nicht erlaubte Variablennamen.

Tabelle 1

| Variable | erlaubt? | Kommentar |
|---|---|---|
| a = 42 | ja | sehr kurz, z. B. für Laufvariablen in Schleifen |
| MeineVariable = 'Tst!' | ja | |
| _messwert = 2233.44 | ja | |
| das_ist_c = 299792.458 | ja | |
| _321_123 = 'Nicht sinnvoll' | ja | aber nicht selbsterklärend |
| 42terWert = 1972 | nein, da mit einer Ziffer beginnend | |
| c$ = -111.222 | nein, da mit Sonderzeichen | |
| No@1 = 'Erster!' | nein, da mit Sonderzeichen | |
| Analoger Messwert22 = 12.345 | nein, weil mit Leerzeichen | |

## Reservierte Wörter und Namen

Bei der Namensvergabe kann man also seiner Kreativität nahezu freien Lauf lassen. Lediglich Schlüsselwörter wie

```
print, for, while
```

dürfen nicht verwendet werden. Zu beachten ist, dass diese Schlüsselwörter in Python stets kleingeschrieben werden. Daneben haben sich einige Konventionen eingebürgert. Diese haben sich in der Praxis bewährt und sind zu einem Quasi-Standard geworden:

- Der Großbuchstabe „i" wird möglichst nicht verwendet, da er mit der Ziffer 1 verwechselt werden könnte.
- Der Großbuchstabe „O", wird im Allgemeinen nicht verwendet, da er mit der Ziffer 0 verwechselt werden könnte.
- Variablennamen werden durch Unterstriche strukturiert, z. B. meine_allerbeste_variable.

## Verzweigungen und Schleifen

In Python sind alle auch bei anderen Sprachen üblichen Kontrollfluss-Anweisungen verfügbar. Allerdings gibt es teilweise wichtige Unterschiede zu den klassischen Varianten.

### If-Anweisung

Einer der wichtigsten Befehle ist die „if-else-elif-Anweisung". Das folgende Beispiel stellt fest, ob eine Zahl gerade oder ungerade ist:

```
x = 9

if x%2 == 0:
    print(x,'ist gerade')
else:
    print(x,'ist ungerade')
```

Eine If-Struktur kann keine, eine oder mehrere Elif-Teile enthalten. Das Schlüsselwort „elif" ist eine Kurzschreibweise für „else if" und kann als Ersatz für Switch- oder Case-Anweisungen in anderen Sprachen dienen. Elif ermöglicht es, Variablen auf mehrere Werte zu überprüfen:

```
x = 7

if x%2 == 0:
    print(x,'ist durch 2 teilbar')
elif x%3 == 0:
    print(x,'ist durch 3 teilbar')
else:
    print(x,'ist weder durch 2 noch durch 3 teilbar')
```

Das Struktogramm in Bild 5 veranschaulicht den Einsatz von if und elif. Um festzustellen, welche Art von if-Anweisung verwendet werden soll, sollte man die folgenden Richtlinien beachten:

- **if-else-Anweisung:** Alternativen schließen sich gegenseitig aus. Das bedeutet, wenn eine Alternative wahr ist, müssen die anderen Alternativen falsch sein.

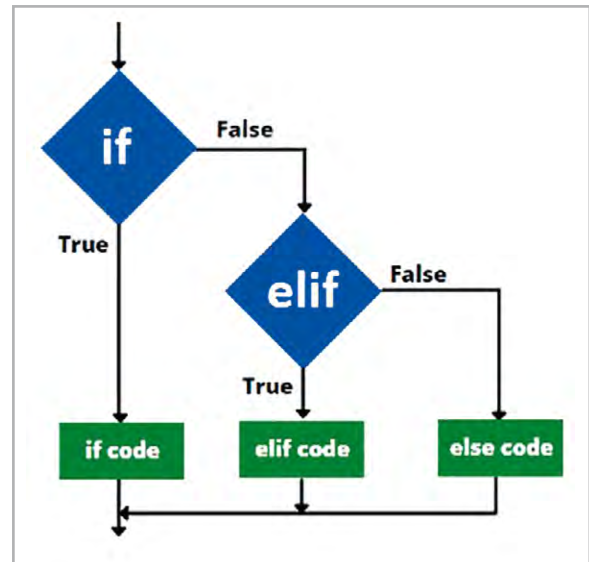


*Bild 5: If und elif*

- **If-elif-else-Anweisung:** Alternativen schließen sich nicht gegenseitig aus. Es ist also möglich, dass mehr als eine Alternative gleichzeitig wahr ist. Eine Zahl kann z. B. sowohl durch zwei als auch durch drei teilbar sein.

### While

Die While-Schleife ermöglicht es, einen Block so lange auszuführen, wie eine bestimmte Bedingung wahr ist:

```
while Bedingung:
        # Code-Block, der wiederholt ausgeführt wird,
        solange die Bedingung wahr ist
```

Die Bedingung wird vor jedem Durchlauf der Schleife überprüft. Wenn die Bedingung wahr ist, wird der Code-Block innerhalb der Schleife ausgeführt. Wenn die Bedingung falsch ist, wird der Code-Block übersprungen. Innerhalb des Code-Blocks kann man Variablen ändern, die die Bedingung beeinflussen. Dadurch kann man die Schleife beenden, wenn eine bestimmte Bedingung erfüllt wird.

Dabei ist es wichtig, sicherzustellen, dass sich die Bedingung irgendwann ändert, damit die Schleife nicht unendlich lang („Endlosschleife") ausgeführt wird.

Das folgende Programm lässt eine LED mithilfe der While-Anweisung genau 10x an Port 24 blinken (blink_10.py):

```
import RPi.GPIO as GPIO
import time

LedPin = 24
GPIO.setmode(GPIO.BCM)
GPIO.setup(LedPin, GPIO.OUT)

counter = 0

while counter < 10:
    print("Counter:", counter)
    counter += 1
    GPIO.output(LedPin, GPIO.LOW)
    time.sleep(0.5)
    GPIO.output(LedPin, GPIO.HIGH)
    time.sleep(0.5)

print("Schleife beendet!")
GPIO.output(LedPin, GPIO.LOW)
```

**For-Anweisungen**

Die For-Anweisung in Python unterscheidet sich von den Versionen in C oder Pascal. Anstatt stets über eine arithmetische Folge von Zahlen zu iterieren oder eine Schrittweite und Endbedingung zu definieren, iteriert die For-Anweisung in Python über die Elemente einer Sequenz (z. B. einer Liste oder eines Strings):

```
a = ['Hans', 'Peter', 'Maximilian']
for x in a:
    print( x, len(x))

>>>
Hans 4
Peter 5
Maximilian 10
```

Die Verwendung von Listen als Laufindex ermöglicht auch die einfache Umsetzung komplexer Schleifen, die so in anderen Programmiersprachen nicht möglich wären (list.py):

```
list_of_lists = [ [1, 2, 3], ['a', 'b', 'c'], ['Anna', 'Berta', 'Claudia']]
for list in list_of_lists:
    for x in list:
        print(x)
```

liefert

```
>>>
1
2
3
a
b
c
Anna
Berta
Claudia
```

**Die Range()-Funktion**

Um über die klassische Folge von Zahlen zu iterieren, kann die Funktion range() verwendet werden. Sie erzeugt Listen, die arithmetischen Aufzählungen entsprechen:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

**Wichtig:** Der angegebene Endpunkt ist nie Teil der erzeugten Liste. Range(10) erzeugt eine Liste von 10 Werten, also exakt die gültigen Indizes für Elemente einer Sequenz der Länge 10. Es ist möglich, den Bereich bei einer anderen Zahl starten zu lassen oder eine andere ganzzahlige Schrittweite anzugeben, wobei hier auch negative Schrittweiten zulässig sind:

```
>>> range(5, 10)
[5, 6, 7, 8, 9]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(-10, -100, -30)
[-10, -40, -70]
```

Um die Indizes einer Sequenz zu durchlaufen, kombiniert man range() und len() wie folgt:

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
print i, a[i]
0 Mary 1 had 2 a 3 little 4 lamb
```

## Praxisübung: SOS mit if_elif und for

Im folgenden Programm werden die in den vorstehenden Abschnitten vorgestellten Strukturen praktisch umgesetzt (SOS.py):

```
import RPi.GPIO as GPIO
import time

led_pin = 18
GPIO.setmode(GPIO.BCM)
GPIO.setup(led_pin, GPIO.OUT)

dot_duration = 0.2
dash_duration = 0.5
pause_duration = 0.3

sos_morse = "... --- ..."

try:
    for _ in range(3):
        for symbol in sos_morse:
            if symbol == ".":
                GPIO.output(led_pin, GPIO.HIGH)
                time.sleep(dot_duration)
            elif symbol == "-":
                GPIO.output(led_pin, GPIO.HIGH)
                time.sleep(dash_duration)
            else:
                time.sleep(pause_duration)
            GPIO.output(led_pin, GPIO.LOW)
            time.sleep(pause_duration)
        time.sleep(pause_duration * 7)

except KeyboardInterrupt:
    pass

finally:
    GPIO.cleanup()
```

Das Programm verwendet die RPi.GPIO-Bibliothek, um auf die GPIO-Pins des Raspberry Pi zuzugreifen. Um die Ausgabe sichtbar zu machen, muss eine LED mit Vorwiderstand, ein LED-Modul oder der LED-Cluster an Pin 18 angeschlossen werden (Bild 6).

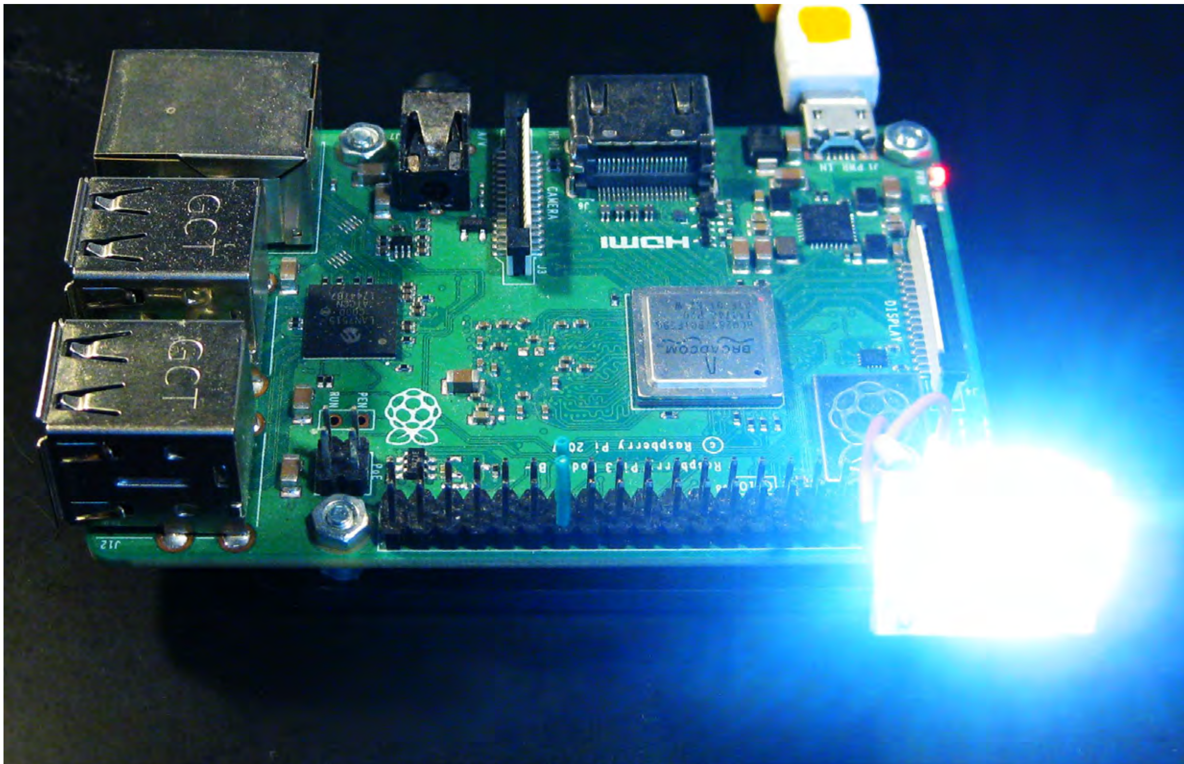


*Bild 6: Ein LED-Cluster erzeugt ein weithin sichtbares SOS-Signal!*

Das Programm definiert die Dauer für einen Punkt, einen Strich und die Pausen zwischen den Morsezeichen.
sos_morse = "... --- ..."

Mithilfe der If/Elif-Anweisung werden die Zeichen interpretiert:
„." liefert ein kurzes Leuchtzeichen (Punkt)
„-" liefert ein langes Leuchtzeichen (Strich)

Schließlich wird der SOS-Morsecode dreimal wiederholt, mit einer Pause zwischen den „Wörtern".
Das Programm wird durch Drücken von Strg+C beendet und die GPIO-Einstellungen werden zurückgesetzt.

## Break- und Continue-Anweisungen

Die Break-Anweisung bricht aus der kleinsten umgebenden For- oder While-Schleife aus. Die Continue-Anweisung setzt die Schleife mit der nächsten Iteration fort. Sie wird ausgeführt, wenn die Schleife vollständig durch die Liste gelaufen ist (mit for) oder wenn die Bedingung falsch wird (mit while), aber nicht, wenn die Schleife durch eine Break-Anweisung terminiert wird. Dies wird bei der folgenden Schleife deutlich, die die Position des ersten negativen Elements in einer Liste findet:

```
numbers = [10, 20, -5, 30, 40, -3, 50]

for i, num in enumerate(numbers):
    if num < 0:
        print("Das erste negative Element befindet sich an der Position:", i)
        break
```

Wenn die Break-Anweisung fehlt, werden fälschlicherweise alle negativen Elemente ausgegeben.

Das folgende Programm soll alle geraden Zahlen von 0 bis 9 außer 4 ausgeben:

```
print("Gerade Zahlen von 0 bis 9, außer 4:")
for i in range(10):
    if i == 4:
        continue  # Überspringe die aktuelle Iteration und fahre mit der nächsten fort
    if i % 2 == 0:
        print(i)
```

In diesem Beispiel wird eine For-Schleife verwendet, um Zahlen von 0 bis 9 zu durchlaufen. Wenn die Variable i den Wert 4 erreicht, wird die Anweisung continue ausgeführt. Dadurch wird die aktuelle Iteration übersprungen und die Schleife fährt mit der nächsten Iteration fort, ohne den nachfolgenden Code auszuführen. Dadurch wird die Zahl 4 nicht gedruckt, die anderen geraden Zahlen von 0 bis 8 werden jedoch korrekt ausgegeben.

## Pass-Anweisungen

Die Pass-Anweisung tut nichts. Sie kann benutzt werden, wenn eine Anweisung syntaktisch notwendig ist, ohne dass das Programm wirklich etwas tun muss. Beispiel:

```
print("Endlos...")
while 1:
    pass
```

Dieses Programm läuft solange, bis es mit CTRL-C unterbrochen wird. Auch zur Messung von Zeiten kann „pass" verwendet werden. Das folgende Programm gibt z. B. die Zeit zwischen zwei Tastereignissen (an Port 24) aus und kann als einfache Stoppuhr verwendet werden (pass.py):

```
import RPi.GPIO as GPIO
import time

GPIO.setwarnings(False)
GPIO.setmode(GPIO.BCM)
readoutPort=24
GPIO.setup(readoutPort, GPIO.IN, pull_up_down=GPIO.PUD_UP)

while 1:
    tStart = time.time()
    while GPIO.input(readoutPort) > 0:
       pass
    tStop = time.time()
    T = tStop-tStart
    print(f"Zeit seit letztem Tastendruck: {T:.{2}f} s")
    time.sleep(1)
```

Das Pass-Statement wird oft verwendet, um eine Stelle in einem Code zu kennzeichnen, an der syntaktisch eine Anweisung erforderlich ist, aber zunächst noch keine Anweisungen bekannt sind:

```
# Eine Schleife, die vorübergehend keine Operationen enthält
for i in range(5):
    pass  # Hier könnte später Code hinzugefügt werden
```

## Eingabe mit Input

Die Input()-Funktion ermöglicht es dem Benutzer, Daten von der Konsole einzugeben:

```
import math

eingabe = input("Bitte geben Sie eine Zahl ein: ")

# Konvertiere die Eingabe in eine Ganzzahl
zahl = float(eingabe)

print("Die eingegebene Zahl ist:", zahl)
print("Die Quadratwurzel der eingegebene Zahl ist:", math.sqrt(zahl))
```

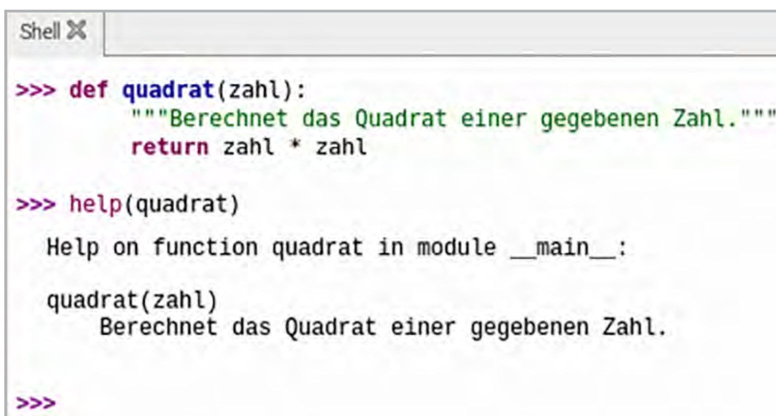


*Bild 7: Hilfefunktion mit Docstrings*

In diesem Beispiel wird der Benutzer aufgefordert, eine Zahl einzugeben. Die Eingabe wird als Zeichenfolge (str) von input() zurückgegeben. Dann wird die eingegebene Zeichenfolge in eine Zahl (float) konvertiert, damit sie für Berechnungen verwendet werden kann.

## Definition von Funktionen und „Docstrings"

Funktionen sind benannte Code-Blöcke, die eine bestimmte Aufgabe ausführen. Sie sind ein wesentliches Konzept in der Programmierung, da sie Code wiederverwendbar machen, die Lesbarkeit verbessern und komplexe Programme in kleinere, handhabbare Teile aufteilen.

Eine Funktion wird in Python mit dem Def-Schlüsselwort festgelegt, gefolgt vom Funktionsnamen und optionalen Parametern in runden Klammern. Abschließend ist ein Doppelpunkt erforderlich:

```
def grüße(name):
    print("Hallo,", name)
```

Funktionen können Parameter akzeptieren, die als Eingabe für die Funktion dienen. Diese Parameter sind Platzhalter für Werte, die der Funktion übergeben werden, wenn sie aufgerufen wird.
Beispiel:

```
def quadrat(zahl):
    return zahl * zahl
```

Eine Funktion wird mit ihrem Namen aufgerufen. Die erforderlichen Argumente werden in runden Klammern übergeben:

```
grüße("Max")
ergebnis = quadrat(5)
```

Man kann sogenannte Dokumentationszeichenfolgen („Docstrings") verwenden, um eine Funktion zu dokumentieren, damit andere Benutzer verstehen, was die Funktion tut und wie sie verwendet werden soll:

```
def quadrat(zahl):
    """Berechnet das Quadrat einer gegebenen Zahl."""
    return zahl * zahl
```

Docstrings sind Zeichenfolgen, die als Dokumentation direkt innerhalb des Quellcodes von Python-Funktionen platziert werden. Ihr Zweck ist es, dem Benutzer Informationen über die Verwendung und Funktionalität des Codes bereitzustellen.

Docstrings werden unmittelbar nach der Definition einer Funktion platziert und in dreifachen Anführungszeichen (""" """) eingeschlossen. Sie enthalten im Allgemeinen eine Beschreibung der Funktion sowie Informationen über Parameter und Rückgabewerte.

In oben stehendem Beispiel ist der Docstring """Berechnet das Quadrat einer gegebenen Zahl.""" die Kurzdokumentation für die Funktion quadrat(). Wenn zu dieser Funktion Hilfe benötigt wird, kann man die integrierte Help()-Funktion verwenden, um den Docstring anzuzeigen (Bild 7).

Die Help()-Funktion ruft den Docstring der Funktion auf und gibt ihn im Terminal aus. Das Hinzufügen von Docstrings zu einem Code ist bewährte Praxis, um die Wartbarkeit von Funktionen zu verbessern.

Das folgende Beispiel zeigt die praktische Anwendung einer mathematischen Funktion. Diese liefert die pythagoreische Summe zweier Zahlen, also die Länge der Hypotenuse eines rechtwinkligen Dreiecks, wenn die beiden anderen Seitenlängen bekannt sind (pytago.py):

```
import math

def pythagoreische_summe(a, b):
    """Berechnet die pythagoreische Summe zweier Zahlen."""
    ergebnis = math.sqrt(a**2 + b**2)
    return ergebnis

x = 3
y = 4

print("Die pythagoreische Summe von", x, "und", y, "ist:",
pythagoreische_summe(x, y))
```

Die Funktion kann nun an allen Stellen, an denen eine pythagoreische Summe erforderlich ist, einfach ersetzt werden durch den Funktionsaufruf pythagoreische_summe.

Auch der rekursive Aufruf von Funktionen ist möglich. Das bedeutet, dass Funktionen sich selbst aufrufen können. Ein wichtiges Beispiel für einen rekursiven Funktionsaufruf ist die Erzeugung von Fibonacci-Zahlen. Dabei handelt es sich um eine Sequenz von Zahlen, bei der jede Zahl die Summe der beiden vorherigen Zahlen ist.

Die ersten Fibonacci-Zahlen lauten: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 ... Fibonacci-Zahlen sind in verschiedenen Bereichen der Mathematik, Informatik und Naturwissenschaften von Bedeutung. Die Fibonacci-Folge lässt sich am einfachsten mit einer rekursiven Funktion erzeugen (Fibonacci.py):

```
def fibonacci_rekursiv(n):
    if n <= 1:
        return n
    else:
        return fibonacci_rekursiv(n - 1) + fibonacci_rekursiv(n - 2)

for n in range(10):
    print("Die", n+1, "-te Fibonacci-Zahl ist:", fibonacci_rekursiv(n))
```

Man erkennt, dass sich die Funktion für n-1 und n-2 selbst aufruft, um zum Ergebnis für n zu kommen.

Ein anderes Beispiel ist die Berechnung der Quadratwurzel einer Zahl (heron.py):

```
def quadratwurzel_von_2_iterativ():
    # Startwert für die Iteration
    x = 1.0
    # Anzahl der Iterationen
    iterations = 10

    # Iterationsschleife
    for _ in range(iterations):
        # Heron-Verfahren: Verbessere die Schätzung der Quadratwurzel
von 2
        x = 0.5 * (x + 2 / x)

    return x

ergebnis = quadratwurzel_von_2_iterativ()
print("Quadratwurzel von 2 (iterativ):", ergebnis)
```

Dieser rekursive Algorithmus ist als Heron-Verfahren bekannt und liefert sehr schnell gute Näherungen für Quadratwurzeln.

Abschließend sei noch auf das Schlüsselwort „global" hingewiesen. Dieses erlaubt es, eine Variable innerhalb einer Funktion als globale Variable zu behandeln. Das bedeutet, dass die betreffende Variable auch außerhalb der Funktion gelesen und modifiziert werden kann.

Im folgenden Beispiel teilt „global x" Python mit, dass x auf die globale Variable x verweist und nicht auf eine lokale Variable innerhalb der Funktion. Daher wirkt sich eine Änderung an x innerhalb der Funktion auch auf die globale Variable x aus.

```
x = 10

def meine_funktion():
    global x
    x += 5
    print("Innerhalb der Funktion:", x)

meine_funktion()
print("Außerhalb der Funktion:", x)
```

## Übungen und Anregungen

- Schalten Sie eine LED genau 7x ein und aus:
  - einmal interaktiv in der Shell
  - einmal programmgesteuert
- Nähern Sie die Zahl Pi als Quotienten zweier Integervariablen an!
- Ersetzen Sie eine Elif-Konstruktion im SOS-Programm durch reine If/Else-Schleifen!
- Ist das immer möglich? - Wo liegt der Vorteil von „elif"?
- Berechnen Sie die Quadratwurzel von 2024 mit einem iterativen Funktionsaufruf!

## Ausblick

In diesem Beitrag wurden die wichtigsten Programmstrukturen von Python erläutert und praktisch erprobt. Nach dem Durcharbeiten des Artikels ist man nun bestens vorbereitet, auch etwas anspruchsvollere Aufgaben zu lösen.

Im nächsten Beitrag sollen die neuen Erkenntnisse eingesetzt werden, um Python in der Elektronikpraxis effizient und nutzbringend einzusetzen.

Eine der wichtigsten Aufgaben von digitalen Systemen ist die Erfassung von Mess- und Umweltdaten. Vor ihrer digitalen Verarbeitung müssen diese in eine binäre Form konvertiert werden. Dies kann mit Analog-Digital-Umsetzern erreicht werden. Der nächste Artikel wird sich eingehend mit der Technik der Analog-Digital-Umsetzung und ihrer Umsetzung mit Python befassen. ELV

| Material | Artikel-Nr. |
|---|---|
| z. B. Raspberry Pi 4 Model B, 8 GB RAM | 250567 |
| LEDs und LED-Cluster sind enthalten im Prototypenadapter-Set PAD6 (CMOS-Logik) | 158980 |

**Zum Download-Paket**