

Basteln mit dem Raspberry Pi Pico und MicroPython

Vorab-Handout DGHK Ostercamp 2025

Jochen Brinkmann

Inhalt

Vorbereitung auf das Camp	3
Was braucht Ihr?	3
Installation der Software.....	3
Variante 1: alle Systeme – manueller Download	3
Variante 2: Windows winget	3
Python	4
Hardware	4
Python Kurzeinführung.....	5
Programmieren mit Python.....	5
Variablen.....	6
Beispiele	6
Globale Variablen	6
Ausgabe von Texten auf dem Bildschirm	7
Bedingungen	8
Vergleichsoperatoren für Bedingungen	9
Beispiele	9
Schleifen.....	10
while-Schleife ("Wiederhole, bis Bedingung nicht mehr erfüllt").....	10
for-Schleife ("Für jeden Wert in..." bzw. "Von 0 bis n erreicht ist...").....	10
Funktionen	11
Module nutzen	12
Häufig genutzte Module	12
Beispiel	12
Code-Kommentare.....	13
Literatur und Links	13
Links	13
Bücher & Zeitschriften	13

Vorbereitung auf das Camp

Was braucht Ihr?

Wichtig ist erstmal der Rechner:

- **Notebook** mit Windows 10/11
Linux und Macbook gehen natürlich auch, da kann ich aber nicht immer helfen.

Große Ansprüche an die Leistung gibt es nicht, allerdings sollte das Betriebssystem auf einem aktuellen Stand sein, ihr solltet mit dem Umgang vertraut sein, und die USB-Ports sollten funktionieren.

Auf dem Rechner solltet ihr zuhause bereits folgendes vorinstalliert haben:

- **Python 3.x** (aktuell ist 3.13)
- Die Entwicklungsumgebung **Thonny** 4.x (aktuell 4.16 oder 4.17)

Außerdem braucht ihr:

- **MicroUSB-Kabel** zum Verbinden des Raspberry Pi Pico mit dem Rechner
(das sind die alten USB-Kabel mit dem flachen trapezförmigen Stecker!)
- **USB-Stick**, um eure Programme zu sichern und auszutauschen

Installation der Software

Variante 1: alle Systeme – manueller Download

- Python gibt es als Installer hier: <https://www.python.org/downloads/>
- Thonny gibt es hier: <https://thonny.org/>

Ladet die Files für euer System runter und installiert zuerst Python und dann Thonny.

Auf den Seiten gibt es systemspezifische Instruktionen, folgt diesen und holt euch notfalls bei euren Eltern Hilfe (für Python benötigt ihr Admin-Rechte!).

Variante 2: Windows winget

Neuere Windowssysteme kommen ähnlich wie Linux mit einem Paketmanager vorinstalliert. Einfach mal eine Kommandozeile öffnen und **winget**¹ eingeben – wenn ihr danach keine Fehlermeldung, sondern eine Befehlsliste bekommt, dann gebt ihr nacheinander jeweils die folgenden Befehle in eine neue Zeile ein (jeweils mit einem **Enter** abschließen):

- **winget install Python.Python.3.13**
- **winget install AivarAnnamaa.Thonny**

⚠️ Wichtig: erst abwarten, bis das jeweilige Tool fertig installiert ist. Dabei poppt auch mal ein Installationsdialog auf, den ihr gegebenenfalls mit der Maus bestätigen müsst. Ihr benötigt auch auf diesem Weg Admin-Rechte für die Python-Installation!

¹ Für mehr Details siehe <https://learn.microsoft.com/de-de/windows/package-manager/winget/>

Python

Python ist eine beliebte Programmiersprache, die sich aufgrund eines reichhaltigen Angebots an Bibliotheken (fertige Funktionspaketen für jeglichen Bedarf) und einer guten Zugänglichkeit großer Beliebtheit erfreut. Auch der Raspberry Pi Pico und viele andere Microcontroller unterstützen diese Sprache, so dass es erstaunlich einfach ist, diese zu programmieren.

Wem es auf Geschwindigkeit ankommt und wer das letzte aus der begrenzten Hardware rauskitzeln möchte, der greift zu Programmiersprachen wie C, C++ oder Rust - die aber auch viel mehr Vorkenntnisse und Einarbeitung voraussetzen. Für uns und die meisten Bastelprojekte reicht die Performance von Python allemal – und der deutlich leichtere Einstieg führt zu schnellen Resultaten. Von daher ist diese einsteigerfreundliche Sprache genau die richtige für uns.

 Vorkenntnisse in Python sind von Vorteil, weil ihr dann weniger mit dem Lernen der Sprache beschäftigt seid, und euch mehr auf die Ansteuerung der Hardware sowie das kreative Ausprobieren konzentrieren könnt. Vielleicht habt ihr ja die Gelegenheit nebenbei schonmal etwas in die Sprache reinzuschauen! Aber keine Sorge: wir brauchen nur Grundlagen, und die gehen wir alle nebenbei auch nochmal durch!

Im nächsten Teil findet ihr eine Kurzeinführung in Python. Damit könnt ihr gerne schonmal etwas rumspielen bis das Camp losgeht – und es beim Kurs dann als Spickzettel nutzen!

Hardware

Wir werden mit dem Raspberry Pi Pico arbeiten. Das ist ein kleiner Microcontroller, der neben einer einfachen Programmierung via Python eine Menge Schnittstellen bietet, um darüber Sensoren, LEDs, Taster, Motoren oder ähnliches anzusteuern.



Abbildung 1 Der Raspberry Pi Pico (Serie 1)

Die Materialien (Breadboards, Kabel, Sensoren, LEDs...) für den Kurs werden leihweise von mir gestellt, den Pico selbst könnt ihr am Ende des Kurses mitnehmen und zuhause damit weiterbasteln.

Python Kurzeinführung

Programmieren mit Python

Diese Kurzeinführung für Python soll keine vollwertige Anleitung sein, sondern mehr eine Schnellreferenz, die sich auf die wenigen und grundlegenden Teile von Python beschränkt, welche wir im Kurs brauchen. Jeden einzelnen Punkt könnte man noch um viele Details und Besonderheiten ergänzen – und vieles lassen wir ganz aus. Wer tiefer einsteigen möchte findet weiter hinten entsprechende Links und Literaturhinweise.

Um ein Programm in Python zu erstellen, braucht man eigentlich nur einen beliebigen Texteditor. Wer es etwas bequemer haben möchte, der nimmt eine spezielle Entwicklungsumgebung – in unserem Fall **Thonny**. Darin schreibt man Zeile für Zeile die Anweisungen, welche ausgeführt werden sollen und führt das Programm dann mit einem Python-Interpreter (= "Übersetzer") aus. Das ist ein Programm, welches die zuvor geschriebenen Anweisungen ausliest, abarbeitet und für den Computer in ausführbare Instruktionen übersetzt. Der Python-Interpreter auf dem Raspberry Pi Pico heißt **MicroPython** und bringt unter anderem spezielle Anpassungen für diesen Mikrocontroller (z.B. Unterstützung der Pins und Schnittstellen) mit sich.

Beim Schreiben der Programme nutzt man **Variablen**, um sich Werte zu merken, sowie **Bedingungen**, **Schleifen** und **Funktionen**, um den Programmablauf zu strukturieren und zu steuern. Zusätzliche, von anderen Leuten zur Verfügung gestellte Funktionalitäten lädt man über sogenannte **Module** hinzu, um sie für die eigenen Programme zu verwenden. Diese Programmbausteine werden hier, zusammen mit ein paar nützlichen Grundfunktionalitäten (z.B. Textausgabe, Kommentare) kurz vorgestellt.

Eine Besonderheit – und häufige Fehlerquelle - bei Schreiben von Python-Code ist die Tatsache, dass die Sprache es mit Einrückungen im Text sehr genau nimmt.

Egal ob eine bedingte Ausführung, Schleife oder eine Funktionsdefinition einen Codeblock einleitet – der Inhalt wird mit Tab eingerückt (oder 4 Leerzeichen), und wenn die Einrückung endet, dann endet auch der Codeblock. Das erhöht die Lesbarkeit des Codes enorm.

Ein **Codeblock** ist eine aufeinanderfolgende Reihe von Zeilen im Quellcode, die nacheinander abgearbeitet werden und zusammengehören. Ein Codeblock kann aus einer einzelnen Zeile oder vielen bestehen und auch selbst wieder Codeblöcke enthalten. Beispiele für Codeblöcke sind das Innere einer Funktion, Schleife oder Bedingung.

Variablen

Variablen sind Namen für Werte, welche im Vorfeld nicht bekannt sind (z.B. ein Messwert, der Status eines Schalters, ein Name oder Wert, welchen der Nutzer erst während des Programmablaufs eingibt), und die **variabel** (= veränderlich) sind. Im Laufe des Programmverlaufs kann der Variablen immer wieder ein neuer Wert zugewiesen werden. Für die Zuweisung wird das einfache Gleichheitszeichen verwendet ('=').

Beispiele

- Einer Variablen namens `a` den Wert 3 zuweisen: `a = 3`
- Einer Variablen namens `stadt` den Text (auch **String** genannt) „Hamburg“ zuweisen:
`stadt = 'Hamburg'` ← wichtig: nur einfache Anführungsstriche!
- Einer Variablen `loesung` ein Rechenergebnis zuweisen:
`loesung = 20 * (a - 2)` ← es gelten die Klammerregeln wie in Mathe!

<code>a + b</code> addiert zwei Zahlen <code>a</code> und <code>b</code>	<code>a * b</code> multipliziert <code>a</code> mit <code>b</code>
<code>a - b</code> zieht den Wert <code>b</code> von <code>a</code> ab	<code>a / b</code> teilt <code>a</code> durch <code>b</code>

- Fließkommazahlen werden einfach mit einem Punkt (statt Komma) definiert, also `2.0` oder `27.33` (keinen Punkt als Trenner für die Tausender verwenden!):
`ergebnis = 2.5 * 2.5 * 3.14`
- Einer Variablen den Wert WAHR (`True`) oder FALSCH (`False`) zuweisen:
`echt = True`
- Einer Variablen eine Liste von mehreren Werten zuweisen:
`obst = ["Apfel", "Birne", "Traube"]`

⚠️ Variablennamen, aus denen man Sinn und Zwecke der Variable klar erkennen kann, erhöhen die Lesbarkeit eures Codes enorm! Vermeidet Namen wie `a`, `b`, `x`, `y`, `var`,...

Globale Variablen

Bestimmte Variablen werden immer wieder gebraucht, auch in Funktionen, wo diese eigentlich nicht verfügbar sind. Normalerweise können sie dort nur gelesen werden. Über das Schlüsselwort `global` kann ihnen aber trotzdem einen Wert zuweisen:

```
externeVariable = False
```

```
def meineFunktion():
    global externeVariable
    externeVariable = True

print(externeVariable)
meineFunktion()
print(externeVariable)
```

Der obige Code gibt zuerst `False`, dann `True` aus - die Funktion verändert die Variable.

Ohne die Variable mit dem Stichwort `global` innerhalb der Funktion freizugeben, würde die ursprüngliche Variable nicht verändert und das Ergebnis wäre eine Ausgabe von `False` und `False`... (innerhalb der Funktion würde aber der zugewiesene Wert verwendet!)

Ausgabe von Texten auf dem Bildschirm

Um zu sehen, was das Programm gerade macht, kann man in Python unter anderem Texte auf dem Bildschirm ausgeben, das funktioniert mit der eingebauten `print()`-Funktion. Der Befehl

```
print('Hallo!')
```

gibt zum Beispiel

Hallo!

auf dem Bildschirm aus.

Strings verknüpfen

Durch Kombinieren von Text und Variablen kann man komplexe und veränderbare Ausgaben erzeugen:

```
print('Hallo ' + name + ', Du bist ' + str(alter) + ' Jahre alt!')
```

Wenn hier die Variable `name` den Wert 'Anton' und `alter` den Wert 12 zugewiesen hat, lautet die Ausgabe:

Hallo Anton, Du bist 12 Jahre alt!

Das `+` wirkt hier als Verknüpfung zwischen den vorgegebenen und den variablen Teilen der Ausgabe. Damit der Zahlenwert an den String angehängt werden kann, muss er erst selbst in einen Text umgewandelt werden – das übernimmt die Funktion `str()`. Die Leerzeichen für den Abstand zwischen den Variablenwerten und dem fixen Text fügt man einfach bei letzterem bereits hinzu. Es gibt auch eine formatierte Ausgabe, die ist für unsere Zwecke aber nicht nötig.

Formatierte Ausgabe

Alternativ kann man auch Platzhalter ('{}') in einen String einbauen und diesen dann über die String-Funktion `format()` um die einzusetzenden Werte ergänzen.

```
textMitPlatzhaltern = 'Hallo {}, Du bist {} Jahre alt!'
textMitErsetzung = textMitPlatzhaltern.format(name, alter)
print(textMitErsetzung)
```

Die Ausgabe ist wie oben, nur muss man den eigentlichen String nicht einzeln zusammensetzen oder die Zahlen erst in Strings konvertieren. Noch einfacher geht es, wenn man `format` direkt am String aufruft:

```
print('Hallo {}, Du bist {} Jahre alt!'.format(name, alter))
```

Was man verwendet, ist Geschmackssache - wobei die formatierte Ausgabe deutlich mächtiger ist und zusätzlich weitreichende Formatierungsmöglichkeiten bietet (z.B. feste Anzahl Nachkommastellen für Fließkommawerte).

Auf dem Raspberry Pi Pico funktioniert die normale Textausgabe mangels Bildschirms natürlich im normalen Betrieb nicht - aber solange er am Rechner angeschlossen ist, kann man die Textausgabe wunderbar zum Testen und bei der Fehlersuche verwenden.

Bedingungen

Bedingungen sind wichtig, um im Laufe des Programms Entscheidungen zu treffen und das Verhalten des Programms an verschiedene Situationen anzupassen. Sie funktionieren nach einer simplen **WENN ... DANN ...**-Muster. Wenn nötig kann dies durch ein oder mehrere **ANSONSTEN WENN ... DANN ...**-Entscheidungen und/oder ein **ANSONSTEN ...** ergänzt werden. In Python ist das auf Englisch und verkürzt (**if**, **elif** und **else**) und das DANN wird durch Doppelpunkt und nachfolgende Einrückung angezeigt.

Umgangssprachlich	Python	Wann verwenden?
WENN <Bedingung ist WAHR>, DANN <mache dies und jenes>	if <Bedingung>: <mache dies> <mache jenes>	Immer: Anfang einer bedingten Ausführung. Nur wenn die Bedingung erfüllt ist, soll etwas bestimmtes getan werden
ANSONSTEN WENN <Bedingung ist WAHR>, DANN <mache was anderes>	elif <andere Bedingung>: <mache was anderes>	Optional: Wenn die vorhergehende Bedingung nicht erfüllt ist, und dafür abhängig von einer anderen Bedingung etwas anderes getan werden soll
ANSONSTEN <mache was ganz anderes>	else: <mache was ganz anderes>	Optional: Wenn die vorhergehenden Bedingungen nicht erfüllt sind und für diesen Fall etwas bestimmtes passieren soll

Die Bedingung prüft dabei immer einen 'boolschen Ausdruck'. Das ist eine Variable, eine Funktion oder eine Vergleichsoperation, die als Ergebnis oder Inhalt einen Boolean, also einen "Wahrheitswert" ergibt, welcher entweder **True** (WAHR) oder **False** (FALSCH) ist. Mehrere boolsche Ausdrücke können mit **and** (= logisches *UND*), **or** (logisches *ODER*) verknüpft oder mit **not** (= logisches *NICHT*) umgekehrt werden.

Bedingungen können dabei geschachtelt werden. Dabei hat sowohl die Reihenfolge wie auch die Klammersetzung Einfluss auf die Bedeutung eines Ausdrucks:

- **a and b or c** ↔ a und b müssen wahr sein – oder nur c oder alle drei
- **a and (b or c)** ↔ a und entweder b oder c müssen wahr sein - oder alle drei

Es empfiehlt sich durch Klammern vorzugeben, wie der Ausdruck bewertet werden soll.

Vergleichsoperatoren für Bedingungen

Python	Bedeutung
A < B	A kleiner als B (es gibt auch ' \leq ' für kleiner oder gleich)
A > B	A größer als B (es gibt auch ' \geq ' für größer oder gleich)
A == B	A hat den gleichen Wert wie B ⚠️ Wichtig: doppeltes Gleichheitszeichen ('==') für Vergleiche!!!
BedingungA or BedingungB	Logisches <i>ODER</i> Bedingung A und/oder Bedingung B ist erfüllt. Nur falsch, wenn keine der beiden Bedingungen erfüllt ist!
BedingungA and BedingungB	Logisches <i>UND</i> Bedingung A und Bedingung B sind beide erfüllt. Falsch, wenn mindestens eine der Bedingungen nicht erfüllt ist!
not BedingungA	Logisches <i>NICHT</i> Bedingung A ist nicht erfüllt. Das logische NICHT kehrt die Bedeutung einer Bedingung um – aus false wird true und umgekehrt!
Bedingung A and (BedingungB or BedingungC)	Klammern können genutzt werden, um mehrere Bedingungen zu einer zusammenzufassen und mit anderen zu kombinieren. Hier: Bedingung A muss erfüllt sein UND Bedingung B ODER C müssen erfüllt sein. Ohne die Klammer wurde dieser Ausdruck anders interpretiert.

Beispiele

```
ergebnis = 'unbekannt'
if a == b:
    ergebnis = 'gleich'
elif a < b:
    ergebnis = 'kleiner als'
else:
    ergebnis = 'groesser als'
print('A ist ' + ergebnis + ' B')
```

if, **elif** oder **else** sind nicht eingerückt, die Zeile endet aber jeweils auf einen Doppelpunkt (`:`). Der Codeblock mit den dazugehörigen Befehlen ist eingerückt (ein Tab bzw. 4 Leerzeichen gegenüber der Bedingungszeile).

Geschachtelte Bedingungen werden einfach immer weiter eingerückt:

```
if a > b:
    print('A ist groesser')
    if c == 3:
        print('C ist DREI!')
    else:
        print(C ist' + c)
else:
    print('A ist kleiner!')
```

Schleifen

Schleifen erlauben es einen Codeblock wiederholt auszuführen, entweder

- so lange, bis eine Bedingung nicht mehr erfüllt ist
- alle Elemente einer Liste abgearbeitet wurden
- bis der Code eine bestimmte Anzahl mal ausgeführt wurde

Ein Beispiel ist z.B. eine Schleife, in der der Wert eines Sensors immer wieder abgefragt wird, bis ein bestimmter Wert erreicht oder überschritten wurde, oder eine Schleife, welche eine LED immer wieder ein und ausschaltet, so dass diese blinkt.

while-Schleife ("Wiederhole, bis Bedingung nicht mehr erfüllt")

Bei der **while**-Schleife (übersetzt etwa **WÄHREND**) wird der zugehörige Codeblock so lange wiederholt, wie die Bedingung erfüllt ist. Auch hier ist die Einrückung wichtig!

```
temperatur = temperatur_auslesen()
while temperatur > 2.0:
    if knopf_gedrueckt():
        break
    temperatur = temperatur_auslesen()
```

Die Schleife kann auch mittendrin abgebrochen werden (die **break**-Anweisung oben im Beispiel), oder man kann mit **continue** direkt zurück an den Anfang der Schleife für den nächsten Durchlauf springen.

for-Schleife ("Für jeden Wert in..." bzw. "Von 0 bis n erreicht ist...")

Eine Liste Element für Element abarbeiten

Im Gegensatz zur **while**-Schleife kann man mit **for...in...** (= 'für...in...') für jedes Element einer Liste den Code der Schleife ausführen. Beispielsweise hat man eine Liste mit Früchten und möchte für jede einen Text ausgeben:

```
obst = ['Äpfel', 'Birnen', 'Trauben']
for frucht in obst:
    print('Lecker, ich mag ' + frucht + '!')
```

Dabei muss man keine Bedingung prüfen oder Variablen hochzählen – die Variable **frucht** wird automatisch bei jeder Runde auf den nächsten Wert in der Liste gesetzt und kann dann verwendet werden. Wenn alle Elemente der Liste abgearbeitet sind, endet die Schleife.

Eine Schleife genau n -mal ausführen

Um einen Codeblock genau eine bestimmte Anzahl mal auszuführen, nutzt man die Funktion **range(<Anzahl>)** – diese gibt eine Liste mit der gewünschten Anzahl von Zahlen von 0 aufwärts zurück, die man dann wie oben gezeigt in einer **for**-Schleife durchlaufen kann.

Auch hier kann mit **break** die Schleife abgebrochen oder es mit **continue** direkt weiter in die nächste Runde gehen.

```
for i in range(5):
    print('Schritt ' + str(i))
```

Funktionen

Programme können leicht unübersichtlich werden, und bestimmte Funktionalitäten werden immer wieder gebraucht. Anstatt nun jedes Mal den gleichen Code erneut zu schreiben, wird er in einen separaten Teil des Programmes ausgelagert, der dann über einen Namen von überall im Code aufgerufen werden kann. Dabei können diesem Codeteil Variablen und Werte als **Parameter** übergeben werden, die für die Ausführung benötigt werden. Und als Ergebnis kann die Funktion einen Wert zurückgeben, den **Rückgabewert**.

```
def printHallo():
    print('Hallo!')
```

Dies ist eine einfach Funktionsdefinition. Die Methode hat keine Parameter und keinen Rückgabewert, sie enthält auch nur eine Zeile Code, aber es braucht oft auch nicht.

Die Funktion kann nun im Code wie ein normaler Befehl verwendet werden:

```
if not schonBegruesst:
    printHallo()
    schonBegruesst = true;
```

Funktionen mit **Parametern** und **Rückgabewert** sehen wie folgt aus:

```
def calcKreisflaeche(radius):
    flaeche = 3.14159 * radius * radius
    return flaeche
```

Die Parameter sind als Variablennamen in der Klammer definiert, die Rückgabe eines Wertes erfolgt über den **return** Befehl. Und nein, Pi sollte man nicht als Zahl im Code eintragen – die gibt es als Konstante mit deutlich höherer Genauigkeit im Modul **math** (**math.pi**).

Will man nun eine Fläche berechnen, so ruft man einfach die Funktion auf:

```
kreisflaeche = calcKreisflaeche(2.5)
```

Statt Werten könnten hier auch Variablen übergeben werden, deren aktueller Wert dann in der Funktion verwendet wird, z.B. ein Messwert oder eine Benutzereingabe. Wichtig ist dabei nur, die Werte der Parameter müssen natürlich zur Funktion passen. Würde man hier einen Text übergeben, dann würde das zu einem Fehler führen.

⚠️ Für Funktionen gilt das gleiche, wie für Variablen: vergeb Nom, welche sofort erkennen lassen, was die Aufgabe der Funktion ist. Nutzt entweder Großbuchstaben (sogenanntes Camel Case, z.B. `'retteDieWelt()'`) oder Unterstriche (z.B. `'rette_die_welt()'`) zur Worttrennung und beginnt den Nom mit einem Verb – die Funktion macht ja etwas bestimmtes, das soll beschrieben werden!

Module nutzen

Nicht alles kann oder muss man selbst schreiben – sei es aus Zeitmangel oder fehlendem Fachwissen. Dafür gibt es Module, das sind Sammlungen an Objekttypen, Funktionen und Konstanten (feste Werte), die man in seinem Projekt nutzen kann. Dies macht man über den Befehl **import**:

```
import time
```

Der obige Befehl importiert (= macht verfügbar) das Modul **time**, welches verschiedene Funktionen zu Zeit und Timern zur Verfügung stellt. Wenn man das Modul importiert hat, dann kann man fortan die enthaltenen Funktionen in seinem Programm nutzen - z.B. den Befehl **time.sleep(zeitInSekunden)**, um das Programm eine bestimmte Zeit warten zu lassen.

Häufig genutzte Module

Python bringt bereits viele nützliche Module von Haus aus mit:

Modulname	Wofür?
time	Funktionalitäten rund um Zeit, aber auch Pausieren
machine	Modul für die speziellen Funktionalitäten des Raspberry Pi Pico
random	Zufallszahlen erzeugen
math	Mathematische Funktionen
sys	Verschiedene Systeminfos (aber auch z.B. die Parameter mit denen das Programm gestartet wurde)
network	Wifi-Funktionalität (nur Raspberry Pi Pico W)
bluetooth	Bluetooth-Funktionalität (nur Raspberry Pi Pico W)

Beispiel

In der Praxis sieht das dann beispielsweise wie folgt aus (kleiner Vorgeschmack auf den Kurs):

```
import time
import machine
# Hole Pin 15 als Ausgabe und weise ihn der Variablen led zu
led = machine.Pin(15, machine.Pin.OUT) # aus Modul machine
# Endlosschleife, LED an PIN 15 wird alle 200ms umgeschaltet (an/aus)
while True:
    led.toggle()
    time.sleep(0.2) # aus Modul time
```

Wenn an Pin 15 eine LED angeschlossen ist, dann würde sich diese nun bis zum Beenden des Programms alle 200 Millisekunden an- und im nächsten Durchlauf wieder ausschalten – sie würde blinken!

Hier werden erst die beiden benötigten Module importiert, dann wird einer der Pins des Raspberry Pi Pico als Ausgabe-Pin konfiguriert (aus dem Modul **machine**) und einer Variablen namens **led** zugewiesen. Die Variable ist vom Typ Pin und besitzt eine Funktion namens **toggle()**, über die das Signal an diesem Pin umgeschaltet werden kann. Über **time.sleep(0.2)** (aus dem Modul **time**) pausiert das Programm 0.2 Sekunden (= 200ms) bevor es weitergeht.

Code-Kommentare

Um auch später noch gut nachvollziehen zu können, was das gerade geschriebene Programm genau macht, hilft es dieses möglichst gut zu dokumentieren. Der einfachste Weg hierfür ist es Code-Kommentare einzufügen. Das sind Zeilen im Programm, die vom Interpreter ignoriert werden und einen beliebigen Text enthalten können. In Python beginnen Kommentarzeilen mit einer Raute ('#').

```
# Dieser Kommentar könnte erklären, was die Funktion macht...
def zaehleBisWert(maximum):
    for schritt in range(maximum):
        print('Schritt {}...'.format(schritt)) # Auch ein Kommentar
    print('Ziel von {} erreicht'.format(maximum))
```

Alles nach der Raute wird in der jeweiligen Zeile vom Interpreter ignoriert, Kommentare funktionieren daher auch am Ende einer Zeile.

Es macht Sinn fleißig zu kommentieren: was macht diese Funktion, was bedeutet diese Variable, was sind gültige Werte für einen Parameter... - alles, was hilft den Code auch noch nach Monaten auf einen Blick zu verstehen, ist gut!

Literatur und Links

Es gibt massig gute Quellen im Netz, wenn man Python lernen möchte - und natürlich auch Bücher, teilweise sogar speziell für Kinder und Jugendliche. Hier eine kleine Auswahl.

Links

- <https://www.python-lernen.de/>
Eine sehr gute Seite, um Python zu lernen. Von den absoluten Grundlagen hin zu fortgeschrittenen Themen, alles gut erklärt und mit vielen Beispielen.
- <https://docs.python.org/3/library/>
Die Python-Standardbibliothek – hier finden sich detaillierte Informationen zu den häufig in Programmen genutzten Modulen (z.B. `time`, `math`, `random`, ...), die Python von Haus aus mitbringt. In vielen Sprachen verfügbar, leider nicht auf Deutsch.

Bücher & Zeitschriften

- **Python 3 – Das fundierte und praxisorientierte Handbuch**, *Simon Fläig, A/S Verlag*
2020 - 126 Seiten kompakte und verständliche Einführung in die Grundlagen von Python und der Programmierung.
- **Python für Kids: Programmieren lernen ohne Vorkenntnisse**, *Hans-Georg Schumann, mitp Verlag* 2022 - gute Einführung in die Python-Programmierung für Kinder und Jugendliche.
- **Python 3 – Das umfassende Handbuch**, *Johannes Ernesti & Peter Kaiser, 7. aktualisierte und korrigierte Auflage, Rheinwerk Verlag Bonn 2024* - fast zwei Kilogramm geballtes Wissen auf 1100 Seiten. Eher kein Einsteigerbuch, sondern mehr was zum Nachschlagen - und wenn man es genauer wissen muss