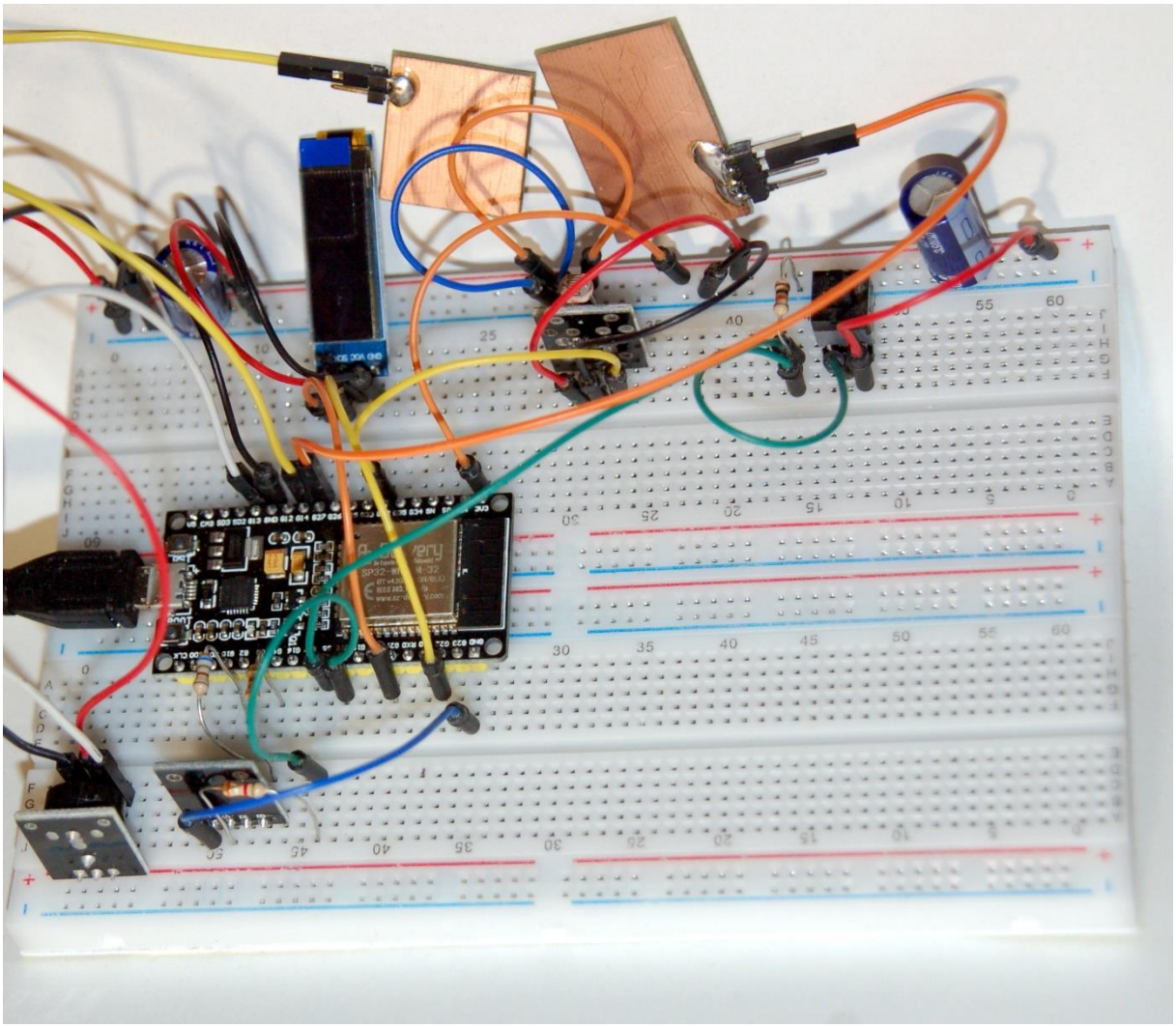


Micropython mit dem ESP32 / ESP8266 – Teil4

Module und Klassen

Herzlich willkommen zum vierten Teil von MicroPython mit dem ESP32/ESP8266. Dieses Mal werden wir den Einsatz weiterer Hardware besprechen und vor allem das Thema Module und Klassen näher beleuchten, eine sehr interessante Materie, wie ich meine. Es wird ein OLED-Display vorgestellt und dazu ein Modul zur vereinfachten Textausgabe und zur Erzeugung von Balkendiagrammen programmiert. Ein Hardwaretimer steuert einen aktiven Buzzer nebst LED. Natürlich dient eine selbst erstellte Klasse dem einfacheren Einsatz in diversen weiteren Programmen.

Bei der Besprechung der Hausaufgaben aus Teil 3 können Sie ein Reaktionszeit-Messgerät mit Touchpad oder Taster programmieren und dabei Ihrer Phantasie vollen Lauf lassen.

Das Kapitel über die Verwendung von esptool.py zum Flashen von Firmware habe ich auf den nächsten Beitrag verschoben, weil dieser Teil einfach zu umfangreich geworden wäre.

Aber bevor ich mit dem eigentlichen Teil 4 starte, etwas ganz anderes.

Raspberry PI Pico – Charakter und Bewertung

Vor ein paar Tagen kam der Raspberry Pi Pico auf den Markt. Es handelt sich dabei um ein Microcontrollerboard der Raspberry.Pi Foundation, die für dieses Board auch den ersten eigenen Chip entwickelt hat, den RP2040 Microcontroller. Es ist ein Dual-core ARM Cortex M0+ 32bit-Processor, der bis 133MHz getaktet werden kann. Weshalb dieses Teil hier im Blog auftaucht, liegt daran, dass in der Firmware bereits **MicroPython** mit integriert ist. Das war zu erwarten, weil Python auch die Muttersprache der Raspberry Pi-Familie ist.

Die Tabelle vergleicht die Eigenschaften des Pico mit denen des ESP32, damit man sich ein Bild von der Leistungsfähigkeit machen kann.

Feature	Raspi Pico	ESP32
Flash	2MB	4MB + 448kB ROM
RAM	264kB	520kB + 16kB RTC
Maße	51x21mm	53x28mm
Befestigung	4x2,1mm	4x2,5mm
Vcc	3,3V	3,3V
Spannungsversorgung extern	1,8..5,0V mit zusätzlicher Hardware	3,3..12V
Takt	133MHz	240MHz
GPIO	23+3 analog	28+6 analog
SPI	2	2
I^2C	2	2
UART	3	3
ADC	3 x 12bit	3 x 12bit
DAC	-	2 x 8bit
PWM	16	16
I^2S	-	ja
USB	1.1	2.0
Touchsensoreingänge	-	10
IR-Controller	-	ja
Pulszähler	-	8 Kanäle, 7 Modi
Bluetooth	-	ja
SD/MMC Controller	ja	ja
RTC	-	ja
Randomnumber-Generator	-	hard- + softwaremäßig
Ethernetcontroller	-	ja mit zusätzlicher Hardware
Hallsensor intern	-	ja

Temperatursensor intern	-	ja
Hardwaretimer	1 mit 4 Alarmeinheiten	2 je 2 Kanäle
Watchdogtimer	-	ja mit extra Takt
WLAN	-	ja Station- und Accesspoint-Mode 802.11 b/g/n
Micropython	in Firmware	ladbar
NodeMCU-LUA	?	ladbar
C/C++	?	Arduino-IDE
AT-Firmware	-	ladbar

Die Tabelle offenbart auf den ersten Blick zwei wesentliche Dinge.

Erstens, der Pico hat von vornherein nur halb so viel Speicher wie der ESP32, Flash und RAM betreffend.

Zweitens, dem Pico fehlt jegliche Netzwerkkonnektivität.

Nun könnte man natürlich über einen UART-Port wenigstens einen ESP8266-01 anbinden, aber das Gelbe vom Ei ist das auch nicht. Und ein zusätzlicher ESP32 kommt für diesen Zweck erst gar nicht in Frage, dann kann ich mir nämlich den Pico sparen.

Die Eigenschaften des RP2040 reichen aber auch bei der Taktfrequenz nur halb an den ESP32 heran. Gut, einige Schnittstellen beider Systeme sind vergleichbar, dennoch fehlen dem Pico einige Features vom ESP32. Der gravierendste Mangel ist die nicht vorhandene Netzwerkausstattung. Das verweist den Pico in die Niederungen eines Arduino, der dafür ebenfalls zusätzliche Hardware benötigt. Das bedeutet eine gewaltige Einschränkung für die Einsetzbarkeit der Hardware im praktischen Leben.

Es ist wunderbar, dass der Pico aus 1,8V bis 5,0V seine Betriebsspannung von 3,3V durch einen Buck-Boost-Converter herstellen kann. Im gleichen Atemzug muss aber auch erwähnt werden, dass man bei Verwendung einer externen Spannungsquelle zusätzlich eine Schutzdiode oder einen P-MOSFET-Transistor verbauen muss.

Auf den ersten Blick, meint man, oh wie schön, die Platine passt ja in eine 40-polige DIL-Fassung. Weit gefehlt, die Pins sind leider um eine Rastereinheit zu weit auseinander. Wenigstens kann man Standardstiftleisten einlöten und damit das Board auf ein Breadboard stecken. Die Halblochkantenkontakte ermöglichen immerhin eine direkte Montage als Modul auf einer Anwenderplatine.

Fazit:
Bei der Riesenauswahl an Microcontrollern für jeden Zweck auf dem Markt mutet die Einführung des Pico seltsam an. Was soll dieses Produkt an Neuem bringen? Der niedrigere Preis kann den Mangel an Performance nicht aufwiegen. Natürlich steigert er das Ego der Rasp.Pi-Foundation – "Wir können auch Microcontroller herstellen."

Nach diesem Ausflug geht es jetzt wieder ans Eingemachte!

An Hardware kommt folgendes Material zum Einsatz, einen Teil davon haben Sie ja bereits, falls Sie die ersten drei Blogbeiträge gelesen und vor allem aktiv umgesetzt haben.

1	ESP32 NodeMCU Module WLAN WiFi Development Board oder
1	ESP-32 Dev Kit C V4 oder
1	NodeMCU Lua Lolin V3 Module ESP8266 ESP-12F mit CH340
1	0,91 Zoll OLED I2C Display 128 32 Piel für Arduino und Raspberry Pi oder
1	0,96 Zoll OLED I2C Display 128 64 Piel für Arduino und Raspberry Pi
1	KY-012 Buzzer Modul aktiv
2	LED (Farbe egal) und
2	Widerstand 330 Ohm für LED oder
1	KY-011 Bi-Color LED Modul 5mm und
2	Widerstand 560 Ohm für LED oder
1	KY-009 RGB LED SMD Modul und
1	Widerstand 330 Ohm für blaue LED
1	Widerstand 680 Ohm für rote LED
1	Widerstand 3,9k Ohm für grüne LED
1	KY-004 Taster Modul Sensor Taste oder
1	keypad-ttp224-14-kapazitiv
1	KY-018 Foto LDR Widerstand
2	Mini Breadboard 400 Pin mit 4 Stromschienen für Arduino und Jumper Kabel
1	Jumper Wire Kabel 3 40 STK. je 20 cm M2M/ F2M / F2F
2	Blech ca. 20 20 mm (nicht Aluminium!) oder Platinenreste
	einige Steckstifte 0,60,612mm

Anmerkung zu den "Touchpads":
An die Blech- oder Platinenstücke müssen Steckstifte für den Anschluss von Jumperkabeln gelötet werden. Deshalb sollte hier auch kein Aluminium hergenommen werden, weil sich das nur mit einem Trick verlöten lässt, der den Einsatz von Chemie nötig macht. Also Kupfer, Messing oder blankes Weißblech nehmen.

Nachdem ein Schwerpunkt in diesem Beitrag neben der Einbindung von Hardware auf dem Erstellen und dem Einsatz von Modulen liegen soll, starten wir gleich mit ein paar Experimenten zu diesem Thema.

Zur Wiederholung ein Statement:
Module sind Sammlungen thematisch zusammengehöriger Daten und Funktionen in einer eigenen Datei. Daten und Funktionen können innerhalb des Moduls zu Klassen zusammengefasst werden.

Zusatzbemerkung: Klassen können auch innerhalb eines Programms erstellt werden.

Weshalb Module verwenden?

Ein paar Fragen, die sich vielleicht im Verlauf des letzten Beitrags, das Statement betreffend, ergeben haben, will ich gleich zu Anfang beantworten.

Frage1:
Warum sollte ich überhaupt Module und Klassen verwenden?

Antwort:
Für manche Dinge in Ihrem Programm werden Sie um die Verwendung von Modulen und Klassen nicht herumkommen, wenn Sie nicht das Rad noch einmal erfinden und z. B. den Zugriff auf GPIO-Pins, Timer, den ADC usw. selbst noch einmal in C/C++ codieren wollen. Das betrifft die bereits in MicroPython enthaltenen Klassen und Module. Bei der Arduino-IDE verwenden Sie sicher auch die vielfältigen Libraries, die für alle möglichen Sensoren angeboten werden. In diesem Sinn haben wir in den vorangehenden Teilen schon regen Gebrauch von Klassen und Modulen gemacht und sogar schon eine eigene Klasse erstellt.

Frage2:
Klar, eingebaute Module zu verwenden ist sinnvoll, aber wozu soll ich selbst so etwas erstellen? Geht das nicht auch ohne? Kann ich so etwas überhaupt?

Antwort:
Bei Programmen mit ein paar Zeilen wird man keine eigenen Module erstellen. Mit zunehmendem Umfang des Programms werden Sie aber folgende Dinge feststellen.

- Es wird immer nerviger zwischen Textteilen am Anfang und weiter hinten hin- und her zu springen
- Je länger der Programmtext wird, desto länger dauert das Hochladen auf den ESP32
- Je länger ein Programm wird, desto mehr schwindet der Überblick
- Je umfangreicher eine Programmdatei wird, desto schwieriger wird es, sie zu pflegen, wenn Sie nach längerer Zeit etwas ändern wollen
- Sie kopieren öfters dieselben Teile eines Programms, die zur Bedienung bestimmter Hardware gedacht sind, in andere Programme

Vielleicht haben Sie deswegen auch schon zum Einsatz von eigenen Funktionen gegriffen, denn es reicht ja, wiederholt verwendete, gleiche Sequenzen nur einmal zu codieren, eben in Funktionen und Prozeduren und bei Bedarf aufzurufen. Bei der Verwendung von Modulen haben Sie erst einmal die Möglichkeit, Ihr Projekt über mehrere Dateien und damit Editorfenster zu verteilen. Statt zu scrollen wechseln Sie das Fenster. Es ist sogar problemlos möglich, mehrere beliebige Texteditoren parallel zu verwenden. Das geht schneller und hat zudem den Vorteil, dass Sie bei Änderungen nur einen Teil ihres Projekts neu von Thonny oder µPyCraft aus auf den ESP hochladen müssen; das spart wiederum Zeit.

Frage3:
Reicht es denn nicht, wenn ich Variablen und Funktionen einfach am Anfang meines Programms platziere?

Antwort:
Das ist ein guter erster Ansatz, der allerdings neben dem wachsenden Umfang des Programms und den damit verbundenen Nachteilen (Antwort zu Frage2) einen weiteren entscheidenden Nachteil aufweist. Was wäre, wenn Sie die gleichen Funktionen auch noch in weiteren Programmen einsetzen möchten. Nehmen Sie als Beispiel die Ansteuerung einer OLED-Anzeige. Freilich können Sie dann den Text aus Datei1 in die Datei2 kopieren. Damit haben Sie aber einen weiteren Grundsatz nicht beachtet, Sie haben einen redundanten Datenbestand erzeugt.
Das bedeutet:

- Dass mit jedem erneuten Kopieren die gleiche Menge an Speicherplatz auf dem lokalen Medium benötigt wird
- Dass Änderungen am Programmtext der Funktionen in jeder bereits bestehenden Dateien durchgeführt werden müssen
- Dass Sie bald den Überblick verlieren, was wo wie programmiert wurde.

Andernfalls reicht es, die Änderungen in einer Moduldatei durchzuführen. Durch den Import werden die Änderungen alle bestehenden und künftigen Programme erreichen.

Frage4:
Welchen Vorteil bietet der Einsatz von Modulen?

Antwort:
Im Wesentlichen habe ich das bereits zu Frage2 und Frage3 beantwortet. Sagen wir es anders herum, positiv formuliert.

- Module als Sammlung von immer wieder benötigten Konstanten, Variablen und Funktionen entlasten jede Programmdatei vom Umfang her.
- Die Verwendung von Modulen erleichtert die Pflege der Software.
- Module können in beliebig viele Programme importiert werden.
- Der zentrale Charakter von Modulen vermeidet Redundanz und erleichtert so die Pflege der Software.
- In MicroPython enthaltene Module erleichtern den Umgang mit portspezifischer Hardware und sind daher unverzichtbar

Frage5:
Was ist der Unterschied zwischen einem Modul und einer Klasse?

Antwort:
In einem Modul können prinzipiell Daten und Funktionen von völlig verschiedener Bedeutung und Verwendung gesammelt werden, so wie in einer Bücherei Bücher zu allen möglichen Themen stehen. Aber in jeder Bücherei sind die Bücher auch thematisch sortiert, was nicht zuletzt die Auffindbarkeit und den Überblick über den Bestand fördert. In MicroPython übernehmen diese Sortierung die Klassen. Ein Modul kann eine oder mehrere Klassen enthalten, muss das aber nicht unbedingt tun.

Ein Modul kann von sich aus weitere Module oder Klassen importieren. Eine Klasse kann dabei von einer anderen Klasse Attribute und Methoden in den eigenen Namensraum übernehmen, man spricht dann von Vererbung.

Frage 6:
OK, wenn das so praktisch ist, wie werden nun Module und Klassen korrekt erstellt?

Antwort:
Dazu schauen wir uns gleich ganz konkrete Beispiele an. Dabei werden wir das eine oder andere Highlight von Modulen und Klassen näher durchleuchten. Am Ende dieses Blogbeitrags sind Sie Modulmogul.

Module und Klassen experimentell

Wir beginnen mit ganz einfachen Aktionen, bei denen es aber darauf ankommt, Seiteneffekte zu vermeiden. Solche Effekte entstehen zum Beispiel dadurch, dass sich MicroPython alles merkt, was wir über REPL, also die Kommandozeile im interaktiven Modus, eingegeben haben oder was bereits gelaufene Programme hinterlassen. Um dieses Verhalten zu umgehen, kann man entweder den ESP32 ausschalten oder einfach nur RST drücken.

Eine Sache, die wir bisher einfach so benutzt haben, schauen wir uns jetzt etwas näher an. Es geht um das Thema Funktionen im weiteren und Variablen im engeren Sinn. Die Kernfrage lautet: Wie kann ich wo und warum Variablen ansprechen und warum gibt es manchmal Probleme damit?

Die erste Sequenz erläutert deshalb das wichtige Thema **Namensraum**. Darunter versteht man den Bereich, in dem Namen von Konstanten, Variablen und Funktionen gültig sind. Die englische Bezeichnung dafür ist **scope**. Dieser Name suggeriert die Bedeutung von 'Sichtbarkeit', denken Sie an den Namen 'Mikro**skop**'. Man könnte Namensraum auch so deuten, dass darin die Daten und Funktionen "sichtbar" sind, also angesprochen werden können. Von der Arduino-IDE her kennen Sie vielleicht die Fehlermeldung "… is not declared in this scope!", wenn Sie vergessen haben, eine Variable zu deklarieren, bevor Sie ihr einen Wert zuweisen oder sie referenzieren.

In MicroPython müssen Sie eine Variable nicht durch eine Typzuweisung deklarieren wie in der Arduino-IDE, aber Sie müssen einer Variablen einen Wert zuweisen, bevor die Variable abgefragt wird, zum Beispiel in einem Formelterm oder einer Anweisung wie print.

MicroPython unterscheidet im Wesentlichen zwei Gültigkeitsbereiche, den **globalen** und den **lokalen**. Als **global** wird alles betrachtet, was im Hauptprogramm (main()) oder von REPL aus deklariert wird. Als lokal ist der Namensraum von Funktionen anzusehen. **Alles, was neu in einer Funktion deklariert wird, ist nur innerhalb dieser Funktion sichtbar** und kann vom Hauptprogramm aus nicht angesprochen werden. Wir sprechen hier von **Kapselung**. Das gilt auch, wenn Variablen innerhalb und außerhalb der Funktion den gleichen Namen haben.

Erzeugen Sie jetzt mittels New in µPyCraft oder Thonny, ich verwende im Weiteren als Oberbegriff die Bezeichnung **Editor**, ein neues Dokument. Diese Datei können Sie immer wieder verwenden, wenn Sie nach dem Test den Inhalt löschen, ergänzen oder überschreiben. Das ist einfacher und übersichtlicher als jedes Mal eine neue Datei anzulegen.

Geben Sie den Programmtext ein, speichern sie im workSpace ab, transferieren Sie das Programm zum ESP32. Drücken Sie RST und starten Sie das Programm. Diese Vorgehensweise wiederholen Sie bitte für jedes weitere der folgenden Beispiele.

Versuche mit Variablen

Variablen werden in MicroPython durch drei Dinge charakterisiert, Name, Wert und Identität. Der Name verweist auf einen Speicherplatz, in dem der Wert gespeichert ist. Werte kann man vergleichen mit ==, !=, <, > <= und >=. Damit MicroPython Variablen eindeutig identifizieren kann, tragen diese außerdem eine Identitätsnummer, das ist eine ganze Zahl. Haben zwei Variablen den gleichen Wert, dann heißt das nicht unbedingt, dass sie dieselbe Identität haben müssen, aber wenn sie dieselbe Identität haben, müssen sie denselben Wert haben. Verrückt? Ja, ein bisschen. Es liegt daran, dass im letzteren Fall eine Variable lediglich zwei verschiedene Namen hat, die aber beide auf denselben Wert zeigen.

```
>>>a = 2
>>>b = 2
>>>c = 1+1
```

Es ist a == b True und natürlich auch a == c True. Das muss so sein, denn alle drei Variablen haben die gleiche Identität, die man mit der Funktion id() überprüfen kann

id(a), id(b) und id(c) liefert stets den gleichen Wert. Bei meinem Versuch war es 5. Erst wenn man der Variablen b, zum Beispiel, den Wert 3 zuweisen würde, änderte sich ihre Identität. a, b und c zeigen also alle drei ursprünglich auf die gleiche Speicherstelle, in welcher der Wert 2 abgelegt ist. OK? Ich will hier nicht weiter ausholen, weil diese Art von Speicherverwaltung sehr seltsame Blüten treiben kann. Vielleicht komme ich in einem anderen Beitrag darauf noch zurück. Jetzt zum Gegenteil.

```
>>> d = 2.0
>>> d == a     True
Aber
id(a)       5
id(d)       77
```

a und d sind wertgleich, klar, aber identisch sind sie nicht, denn die ganze Zahl 2 und die Fließkommazahl 2.0 haben einen unterschiedlichen Typ und können von daher schon nicht identisch sein. Auch OK? Sicher!

Hier gibt es jetzt zwei Variablen mit dem gleichen Namen a. Außerhalb der Funktion ist a global und hat den Wert 10, innerhalb hat ein **ganz anderes**, lokales a den Wert 7. Denken Sie an die Bedeutung des Begriffs Kapselung.

Download: aaa.py

```
def show():
  a=7
  print(a)
  print("id:",id(a))


a=10
show()
print(a)
print("id:",id(a))
```

Ausgabe:
7
id: 15
10
id: 21

Die 7 stammt aus der Funktion, die 10 hat sich über den Funktionsaufruf hinaus im a des Hauptprogramms erhalten. Lassen Sie sich innerhalb und außerhalb der Funktion die Identität von a durch einen zusätzlichen print-Befehl ausgeben, dann sehen Sie, dass es wirklich zwei verschiedene Variablen sind.

Um einer Funktion Daten zur Verarbeitung zu überreichen, können Sie die Übergabe durch Parameter verwenden. Parameter werden in einer Aufzählung nach dem Funktionsnamen in runden Klammern angegeben. Die Namen der Parameter sollten sich wegen der Eindeutigkeit und Klarheit des Programms von den Namen unterscheiden, die im aufrufenden Programm verwendet werden. Mein Parameter heißt hier b und bekommt beim Aufruf als Argument a zugewiesen. Das a im Funktionskörper ist wieder lokal.

Download: aab.py

```
def show(b):
  a=b+7
  print(a,b)

a=10
show(a)
print(a)
```

Ausgabe:
17 10
10

Um ein Ergebnis von der Funktion zurück zu erhalten, verwenden Sie **return**. Wenn Sie **return** nicht angeben, bekommen Sie den Wert **None** zurück.

Download: aac.py

```
def show(b):
  a=b+7
  print(a,b)
  return a

a=10
a= show(a)
print(a)
```

Ausgabe:
17 10
17

Sehr wohl kann aber eine globale Variable innerhalb einer Funktion referenziert (abgefragt) werden, wenn innerhalb der Funktionsdefinition nicht eine Variable gleichen Namens erzeugt wird, wie in den bisherigen Beispielen.

Download: aad.py

```
def show():
  c=a+7
  print(a,c)
  return c

a=10
x=show()
print(a,x)
```

Ausgabe:
10 17
10 17

Folgendes Konstrukt führt aber zu einem Fehler, wenn Sie versuchen, den Wert von a innerhalb der Funktion zu ändern. Der Fehler tritt nicht auf, wenn Sie die Zeile a=20 vor der Zeile c=a+7 platzieren. Dann hat das aber trotzdem nicht den gewünschten Effekt, nämlich, dass der Wert der **globalen** Variable a geändert wird.

Download: aae.py

```
def show():
  c=a+7
  print(a,c)
  a=20
  return c

a=10
x=show()
print(a,x)
```

Ausgabe:
```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 8, in <module>
  File "<string>", line 2, in show
NameError: local variable referenced before assignment
>>>
```

Diese Meldung kommt dadurch zustande, dass der Interpreter beim Übersetzen des Textes der, aus seiner Sicht, lokalen Variable a lediglich einen Speicherplatz zuweist aber noch keinen Wert. Wird jetzt während des Programmlaufs die Funktion aufgerufen, dann hat a in Zeile 2 noch keinen Wert, der zu 7 addiert werden könnte. Die Wertzuweisung an a passiert halt erst in Zeile 5.

Um der Funktion zu sagen, dass das a aus dem globalen Namensraum in der Funktion verwendet werden soll, muss das Ganze so aussehen. **global** weist den Interpreter an, die Variable aus dem übergeordneten Namensraum zu verwenden. Jetzt funktionieren die Addition in Zeile 3 und auch die Wertänderung in Zeile 5 mit dem a-Wert 10 aus dem Hauptprogramm oder von der Kommandozeile.
Download: aaf.py

```
def show():
  global a
  c=a+7
  print(a,c)
  a=20
  return c

a=10
x=show()
print(a,x)
```

Ausgabe:
```
10 17
20 17
```

Kaffeepause? OK!

Hat der Kaffee gemundet? Dann schauen wir uns jetzt noch einige grundsätzliche Dinge zu selbst erstellten Modulen und Klassen an. Das Verständnis dafür werden Sie gleich danach im Hardwareteil benötigen.

Die vier wesentlichsten Versionen des Modul- und Klassenimports

Ähnlich wie bei den Funktionen verhält es sich mit dem Namensraum von Modulen und Klassen. Bevor wir weitere eigene Module und Klassen erstellen, werfen wir noch kurz einen Blick auf die Importvarianten. Für die folgenden Beispiele reicht die Eingabe über die Kommandozeile.

Variante 1

Weil ich mathematische Funktionen und Konstanten einsetzen möchte, muss ich das Modul **math** importieren. Das mache ich auf viererlei Weise und bekomme jedes Mal andere Ergebnisse, wenn ich einen Wert für die Kreiszahl pi =3,14… erhalten möchte.

Neustart (RST), erster Versuch

```
>>>import math
>>> pi
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pi' isn't defined
>>> math.pi
3.141593
>>>
```

import math importiert also, wie es aussieht alle Strukturen aus dem Modul **math** in einem eigenen, gleichnamigen Scope. Die Konstante **pi** ist REPL nicht bekannt. Erfolg habe ich dann, wenn ich den Namensraum **math** benutze. Jetzt ist der gerundete Wert der Kreiszahl **pi** bekannt.

Variante 2

Neustart (RST), nächster Versuch

```
>>> import math as m
>>> pi
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pi' isn't defined
>>> math.pi
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'math' isn't defined
>>> m.pi
3.141593
```

pi ist REPL nicht bekannt, aber auch math.pi kennt REPL jetzt nicht mehr. Dafür ist **m.pi** nun bekannt. Das liegt daran, dass wir den Namensraum math jetzt zu m umbenannt haben. **math** ist damit nicht mehr erreichbar. Ein Alias, wie m in diesem Beispiel, kann man benutzen, um sich Schreibarbeit zu sparen, oder wie in der zweiten Folge, um zwei verschiedene Module mit dem gleichen Namen ansprechen zu können.

Variante 3

Neustart (RST), dritte Variante

```
>>>from math import *
>>> pi
3.141593
>>> math.pi
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'math' isn't defined
>>>
```

Mit **from math import *** importieren wir den gesamten Namensraum von math in den globalen Namensraum. Beide Bereiche verschmelzen also quasi miteinander. Damit ist jeder Bezeichner aus math global verfügbar, aber der Name math selbst ist damit gestorben. Beachten Sie bitte in diesem Zusammenhang auch die Warnung bei der vierten Variante.

Version 4

Neustart (RST), vierte Variante

```
>>>from math import sin, pi
>>> pi
3.141593
>>> sin(pi/2)
1.0
>>> math.sin(math.pi)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'math' isn't defined
>>>
```

Weil ich cos, tan usw. nicht brauche, importiere ich nur die sin()-Funktion und die Konstante pi. Auch hier wirkt die Verwendung von **from** so, dass die importierten Bezeichner in den globalen Namensraum übernommen werden. Sie können sich also merken

- **import … und import as… importiert alle Members von Modulen als eigenen Namensraum. Mit as wird dieser umgetauft.**
- **from … import … verschmilzt den externen mit dem globalen oder übergeordneten Namensraum.**
- **Werden nur Teile eines Moduls mit from importiert, dann sind die restlichen Members des Moduls außerhalb der Klasse nicht verfügbar.**

Aber, Achtung, das passiert unter Umständen auch:

Neustart (RST),

```
>>> pi=3.14
>>> from math import sin, pi      oder  >>> from math import *
>>> pi
3.141593
```

Der Import von Attributen, Konstanten und Funktionen in den aktuellen Namensraum überschreibt dort definierte gleichnamige Objekte. Im Beispiel sind das der vor dem Import definierte Name und der Wert für pi. Dass es sich um verschiedene Objekte handelt, zeigt wieder der Wert der Identität. Sie bekommen sicher für id andere Zahlenwerte.

Neustart (RST)

```
>>> pi=3.14
>>> id(pi)
1073633840
>>> from math import sin, pi
>>> pi
3.141593
>>> id(pi)
1061192072
```

Diese Dinge sind sehr wichtig. Die Betrachtung über die Funktionen und das Verständnis zum Import von Modulen helfen Ihnen, weniger Wutzettel zerknüllt in die Ecke zu donnern, wenn der ESP32 mal wieder nichts als Fehlermeldungen beim Import bringt oder sich völlig unerwartete Rechenergebnisse einstellen.

Eigene Module und Klassen - etwas genauer hingeschaut

Mit dem bisherigen Wissen können wir uns an das Basteln eines Moduls machen, bei dem wir ein wenig weiter hinter die Kulissen schauen, als wir das bei touch.py im 3. Teil getan haben.

Zweckfrei und just for fun habe ich folgenden Programmtext erstellt, unter **mathe1.py** gespeichert und zum ESP32 übertragen. Es werden ein paar Variablen deklariert und drei Funktionen definiert. **rezi(n)** bildet den Kehrwert von n, **sum(n)** die Summe der Zahlen von 1 bis n und **fak(n)** berechnet n!, das Produkt der Zahlen von 1 bis n. Die Funktion des Codes muss ich, glaube ich, nicht weiter erläutern. Wichtiger ist das, was wir mit dem Code anstellen werden. Wenn Sie das **Programm auf dem ESP32 starten**, passiert visuell nicht viel. Im Hintergrund kennt MicroPython aber jetzt den stark gerundeten Wert der Eulerschen Zahl **e** und einen Wert für **Pi** und **PI**. Und der ESP32 weiß, was er tun muss, wenn Sie über REPL die Anweisung **rezi**(8) geben. Probieren Sie es selbst über die Kommandozeile aus.

Download: mathe1.py

```
e =  2.71

def rezi(n):
  if n !=0:
    return 1/n
  else:
    print("Fehler: Versuchte Division durch 0!")
    return None

Pi =3.14
PI = 0

def sum(n):
  summe=0
  for i in range(n+1):
    summe+= i
  return summe

def fak(n):
  nfak = 1
  for i in range(2,n+1):
    nfak *= i
  return nfak
```

```
Pi
3.14
>>> e
2.71
>>> rezi(8)
0.125
>>> sum(20)
210
>>> fak(8)
40320
```

Neustart (RST) neue Version

```
>>>import mathe1
>>> Pi
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'Pi' isn't defined
>>> mathe1.Pi
3.14
>>>
```

Das Modul als solches ist komplett und einsatzfähig für den Import an der Kommandozeile und in Programmen. Unser Vorteil, Änderungen am Text von mathe1 müssen nur an einer Stelle, eben der Datei mathe1.py und nicht in jedem einzelnen Programm durchgeführt werden. Modul ändern, speichern, hochladen und das Testprogramm neu starten. Außerdem können Sie Modul- und Testprogramm in jeweils einem eigenen Fenster parallel zueinander editieren. Ständiges Rauf- und Runterscrollen entfällt. Ein ganz einfaches Beispiel:

Download: mathe1a.py

```
import mathe1

print("Kreiszahl:",mathe1.Pi)
n = 100
print("Die Summe von 1 bis",n,"ist",mathe1.sum(n))
```

```
Kreiszahl: 3.14
Die Summe von 1 bis 100 ist 5050
```

Also das Modul hätten wir erstellt und eingesetzt.

Dann basteln wir jetzt daraus eine Klasse, die wir danach im Einsatz testen werden.

Die Definition einer Klasse beginnt mit dem Schlüsselwort **class,** gefolgt vom Namen der Klasse und dem obligatorischen Doppelpunkt - wissen wir bereits. Beachten Sie bitte wieder, dass, ähnlich wie bei anderen Programmstrukturen, all das, was zum Klassenkörper gehören soll, zusätzlich eingerückt werden muss. Hier ist der Programmtext.

Download: mathe2.py

```
e =  2.71

def rezi(n):
  if n !=0:
    return 1/n
  else:
    return None

class MH:
  Pi =3.14
```

```
PI = 0

def sum(n):
  summe=0
  for i in range(n+1):
    summe+= i
  return summe

def fak(n):
  nfak = 1
  for i in range(2,n+1):
    nfak *= i
  return nfak
```

Speichern Sie den Text unter mathe2.py im workSpace und schicken Sie die Datei zum ESP.

Neustart (RST)

Jetzt wird es spannend. Sicher haben Sie bemerkt, dass ich e und rezi nicht mit in die Klasse aufgenommen habe. Weshalb? Das werden wir gleich alles ausprobieren.

```
>>>import mathe2
>>> mathe2.e
2.71
>>> mathe2.rezi(100)
0.01
```

Sehr gut, scheint alles zu funktionieren – denkste!

```
>>> mathe2.sum(1000)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'module' object has no attribute 'sum'
```

OK, wir haben ja innerhalb des Moduls eine neue Struktur mit einem eigenen Namensraum eingerichtet, dann eben so:

```
>>> mathe2.MH.sum(1000)
500500
>>> mathe2.MH.Pi
3.14
```

Puh, noch mal Glück gehabt. Aber im Grunde hat sich im Vergleich zu vorher am Aufbau fast nichts geändert.

Sollte man nicht von einer Klasse Objekte ableiten können, wie wir es zum Beispiel bereits mit **machine.Pin** getan haben?

```
>>>from machine import Pin
>>>taste = Pin(5,Pin.IN)
```

Versuch:

```
>>> from mathe2 import MH
>>> m=MH
```

Keine Fehlermeldung, sehr gut, funktioniert tatsächlich. Natürlich lassen sich auch die Funktionen, die jetzt in der Klasse als Methoden bezeichnet werden, aufrufen.

```
>>> m.sum(1000)
500500

>>>m.Pi
3.14
```

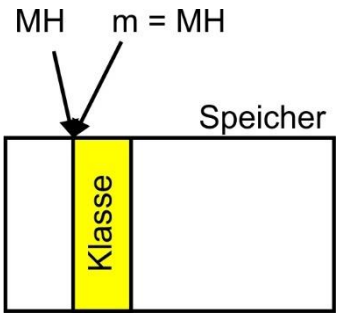
Aber ist **m** jetzt wirklich eine **Instanz** der Klasse **MH**? Der id-Test bringt die Wahrheit ans Licht.

```
>>> id(m)
1073635408
>>> id(MH)
1073635408
```

Und der Aufruf von m sagt ein Übriges.

```
>>> m
<class 'MH'>
```

m und MH sind identisch oder anders gesagt, m ist ein Aliasname für MH, und m kann daher **keine Instanz** von MH bezeichnen.



Mit einem einfachen Befehl können Sie nachschauen, welche Bezeichner für eine bestimmte Klasse oder ein Modul verfügbar sind. Das sagt auch viel über die Art des Aufrufs von Methoden und die Referenzierung von Variablen und zeigt in diesem Fall, dass hinter beiden Namen die dieselbe Struktur steht.

```
>>> dir(m)
['__class__', '__module__', '__name__', '__qualname__', 'sum', '__bases__',
'__dict__', 'Pi', 'PI', 'Rezi', 'fak']

>>> dir(MH)
['__class__', '__module__', '__name__', '__qualname__', 'sum', '__bases__',
'__dict__', 'Pi', 'PI', 'Rezi', 'fak']


>>> dir(mathe2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'mathe2' isn't defined
```

Die Scopes von m und Klasse MH sind identisch, während der Namensraum von mathe2 nicht importiert wurde. Was außerhalb von MH liegt kann nicht im Scope von MH enthalten sein. MH ist ein Member von mathe2 und nicht umgekehrt.

Was ich weiter oben beim Thema Variablen schon sagte, gilt auch für andere Objekte, Funktionen und, wie hier, Klassen. Wir wollen das verallgemeinern.

- **Zwei Objekte sind dann identisch, wenn sie denselben Speicherbereich ansprechen.**
- **Identität schließt damit Wertgleichheit oder gleiche Funktionalität ein.**
- **Verschiedene Namen beziehen sich bei identischen Objekten stets auf denselben Speicherbereich.**

Die Identität zweier Objekte kann man, außer über die Ausgabe des Identitätswerts ganz einfach über das Schlüsselwort **is** feststellen.

```
>>> m is MH
True
```

Neustart (RST)

Jetzt, da wir eine Klasse gebaut haben, sollten wir auch Objekte davon ableiten können. Frisch ans Werk, noch einmal.

```
>>>from mathe2 import MH
>>>m= MH
```

Halt, - dass wir mit dieser Zuweisung keine Objekte erzeugen können, sondern nur einen weiteren Namen für die Klasse MH legt den Schluss nahe, dass da noch etwas Wichtiges fehlt.

Zum Ableiten von Objekten braucht unsere Klasse noch den sogenannten Konstruktor, eine Anweisung, nach der ein Objekt erstellt werden kann. Diese besondere Methode haben Sie bereits im Teil 3 des Blogs kennengelernt. Hier erkennen Sie jetzt den Zweck, den die Methode __init__ erfüllen muss. Das Script **mathe3**.py stellt die Zusammenfassung der behandelten Fälle dar und enthält auch

einen ganz einfachen Konstruktor, der lediglich ein Instanzattribut PI erzeugt und melden kann, dass er aufgerufen wurde. Außerdem müssen aus den beiden bisherigen **Klassenmethoden** sum() und fak() **Instanzmethoden** gemacht werden. Aus diesem Grund wurde in die Parameterliste dieser Methoden als erster Parameter ein **self** hinzugefügt. Das hat bislang gefehlt. Aus dem 3. Teil wissen wir, dass dieses **self** den Bezug des Objekts, das wir von der Klasse ableiten, zu sich selbst herstellt.

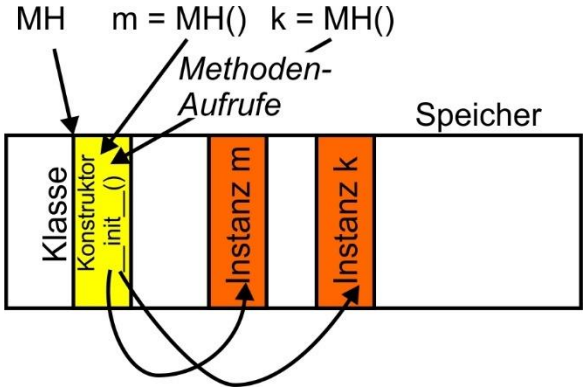
Download: mathe3.py

```
e =  2.71

def rezi(n):
  if n !=0:
    return 1/n
  else:
    return None

class MH:
  Pi =3.14
  PI = 0

  def __init__(self,k=3.1415):
    self.PI = k
    print("PI=",self.PI,"wurde angelegt.")

  def sum(self,n):
    summe=0
    for i in range(n+1):
      summe+= i
    return summe

  def fak(self,n):
    nfak = 1
    for i in range(2,n+1):
      nfak *= i
    return nfak
```

```
>>> from mathe3 import MH
>>> k=MH()
PI= 3.1415 wurde angelegt.
>>> m=MH()
PI= 3.1415 wurde angelegt.
>>> m is k
False
```

Der syntaktische Unterschied zur "Instanziierung" im vorherigen Beispiel ist das Paar runder Klammern nach **MH**(). Ohne diese Klammern wird lediglich die Klasse referenziert und derselbe Speicherplatz einem weiteren Namen zugeordnet. Erst durch Aufruf des Konstruktors, der Methode __init__, über den Klassennamen MH wird eine Instanz, ein Objekt erzeugt. Hier passiert also keine einfache Zuweisung wie m=MH sondern ein Funktionsaufruf m=MH(), der mehr bewirkt als die

Zuweisung. Dafür, dass der Interpreter das erkennen kann, ist das Klammerpaar wie bei jeder anderen Funktion nötig. Dass sich verschiedene Instanzen selbst unterscheiden, wird durch den Identitätsvergleich mit **is** deutlich.



Sie können jetzt auch die Konstante PI von m auf einen anderen Wert setzen. k wird dadurch nicht beeinflusst.

```
>>> m.PI = 3.14
>>> k.PI
3.1415
>>> m.PI
3.14
```

Ebenso wie Attribute kann man auch Funktionen in einer Instanz überschreiben. Ähnlich läuft das auch bei der Vererbung von Klassen. Aber dazu ein anderes Mal mehr.

```
>>> def reziSum(n):
        summe=0
        for i in range(1,n+1):
            summe += 1/i
            print(1/i)
        return summe

>>> reziSum(4)
1.0
0.5
0.3333333
0.25
2.083333
>>> from mathe3 import MH
>>> m=MH()
PI= 3.1415 wurde angelegt.
>>> m.sum(4)
10
>>> m.sum = reziSum
>>> m.sum(4)
1.0
0.5
0.3333333
```

0.25
2.083333

Übrigens, selbst ohne die Definition einer __init__-Methode erzeugt die Anweisung m=**MH()** eine Instanz aus der Klasse MH des Moduls mathe2 mit den Methoden sum() und fak(). Es stehen dann nur keine Instanzattribute zur Verfügung, wohl aber die Klassenattribute Pi und PI. Probieren Sie es aus!

Kaffeepause!

Zurück? Dann machen wir uns jetzt an die Behandlung der neuen Hardware.

Hardware

Timereinsatz und Piepen – der Piezosummer

In MicroPython gibt es drei Arten, um zu warten.

- schlafen und sonst nichts tun
- warten und während dessen immer die gleiche Sache erledigen
- den Wecker stellen, beliebige andere Dinge tun und wenn der Wecker schellt, zwischendurch ganz was anderes erledigen.

Die erste Variante kennen wir vom Blinkerprogramm aus Teil1. Die zweite Variante hatten wir im Einsatz, um im Teil 3 die Geschwindigkeit des ADC zu messen. Und die dritte Art erkunden wir jetzt.

Ich brauche für mein Projekt die Möglichkeit, für eine bestimmte (kurze) Zeit einen Piezosummer ertönen zu lassen. Weil ich mich nicht auch noch um die Erzeugung der Signalfrequenz kümmern möchte, wähle ich einen aktiven Piezosummer, der nur durch eine digitale 1 angesteuert werden muss. Den Pfeifton erzeugt er selber. Während der Summer aktiv ist, soll zeitgleich eine LED aufblinken. Das Einschalten aktiviere ich selbst, das Ausschalten soll, egal womit sich mein Programm gerade beschäftigt, automatisch passieren. In der Zwischenzeit kann das Hauptprogramm beliebige andere wichtige Dinge tun.

Solche Fälle erledigt man gerne durch sogenannte **Interrupts**, wörtlich übersetzt Unterbrechungen. Das Programm wird dann auf Kaffeepause geschickt, die Interruptserviceroutine übernimmt die Steuerung, und wenn sie fertig ist, bekommt das unterbrochene Programm das Ruder zurück. **Unterbrechungen** können durch externe Quellen oder interne Prozesse des ESP32/8622 ausgelöst werden. So kann der Controller auf die Änderung des Logikpegels an einem Pin reagieren oder wie in meinem Fall auf das Ablaufen eines Timers.

Die Methoden und Werte für diese Funktionalität stellt die Klasse **Timer** aus dem Modul **machine** bereit. Interessant, dass **Timer** nicht im Modul **time** wohnt. Tatsache ist aber, dass **time** softwaremäßig implementiert ist, während **Timer** die Hardware des ESP32 betrifft und somit logischerweise in **machine** beheimatet sein muss. Timer sind Hardware-Baugruppen, deren Zählerregister von einem eigenen Takt ohne Prozessorbeteiligung hochgezählt werden und beim Erreichen eines vorgegebenen Zählerstands eine Unterbrechung auslösen können. Der ESP32/8266 besitzt 4 Stück davon.

Ein Timer muss für die Verwendung erst einmal instanziiert werden. Dann ist es wichtig, wie lange es dauern soll, bis der Wecker klingelt und es muss bekannt sein, ob der Wecker nur einmal rasseln soll oder wiederholt. In diesem Fall muss man den Wecker auch ausschalten können. Die Initialisierung übernimmt die Methode **init**(). Die Syntax dafür ist sehr überschaubar.

Erzeugen Sie nun ein Timerobjekt t, bis zu 4 **Timer** sind möglich. Die Nummer wird dem Konstruktor der Klasse übergeben.

>>>from machine import Timer

t = **Timer**(0)

Das Timerobjekt t muss initialisiert werden, gleich im Beispiel für einmalige Aktion.

callback ist eine Funktion, die aufgerufen wird, wenn der Wecker klingelt.

callback = lambda f: aktion

Mit dem Schlüsselwort lambda wird eine anonyme Funktion erzeugt, welche die Aktion ausführt. Das kann eine print-Anweisung sein oder ein arithmetischer Ausdruck oder auch ein Funktionsaufruf wie hier. **period** erhält die Zeitdauer in Millisekunden. **ONE_SHOT** (einmal) und **PERIODIC** (wiederholt) sind Konstanten der Klasse Timer. Alles zusammen ergibt:

t.init(period=5000, mode=Timer.ONE_SHOT, callback=lambda f: Aktion)

t1 = Timer(1)
t1.init(period=2000, mode=Timer.PERIODIC, callback=lambda f: Aktion)

t1.deinit()

deinit() wird benötigt, um den periodisch initialisierten Timer t1 zu stoppen. Bei ONE_SHOT ist das nicht nötig, weil das mit Ablauf des Timers automatisch passiert.

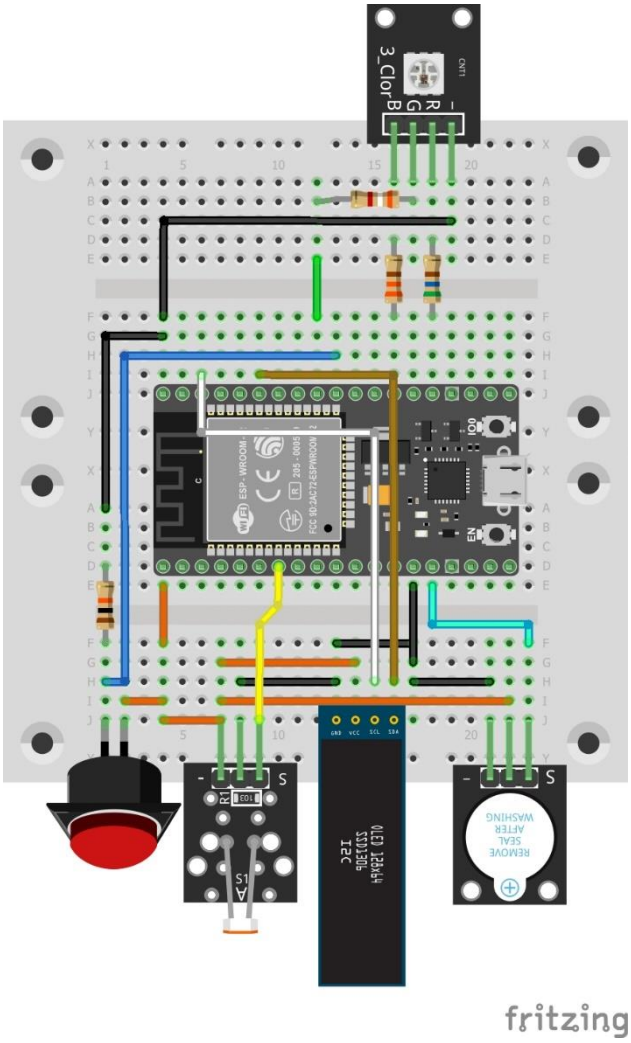
Weil wir jetzt gleich Nägel mit Köpfen machen wollen, basteln wir auch gleich ein Modul beep mit der Klasse BEEP zur Demonstration und zur Vertiefung des erworbenen Wissens.

Download: beep.py

```
from machine import Pin, Timer
import os

DAUER   =     const(15)

class BEEP:
  dauer   =     DAUER

  def __init__(self, led, buzz, duration=dauer):
    self.ledPin=Pin(led, Pin.OUT)
    self.buzzPin=Pin(buzz, Pin.OUT)
    self.beepOff()
    self.tim=Timer(0)
    self.dauer = duration
    print("Konstruktor von BEEP")
    print("LED:{}, Buzz:{}, dauer={}".format(led, buzz, self.dauer))

  def beepOff(self):
    self.ledPin.value(0)
```

```
  self.buzzPin.value(0)

def beep(self, puls=None):
  if puls == None:
    tick = self.dauer
  else:
    tick = puls
  self.buzzPin.value(1)
  self.ledPin.value(1)
  self.tim.init(mode=Timer.ONE_SHOT,period=tick,callback=lambda t: self.beepOff())

def setDuration(self, duration=dauer):
  self.dauer=duration

def getDuration(self):
  return self.dauer
```

Der Anschluss für die rote LED an GPIO2 und den Buzzer an GPIO13 sieht so aus.



Im Modulbereich, außerhalb der Klasse importieren wir von **machine Pin** und **Timer** sowie **os**. Dann wird die Konstante **DAUER** definiert. Danach beginnt die Klassendefinition von **BEEP**.

Wir weisen den Wert der statischen Konstante **DAUER** dem Klassenattribut **dauer** zu – das sind unterschiedliche Objekte, MicroPython ist **case sensitiv!** Jetzt kommt der interessanteste Teil des Moduls, die Konstruktorfunktion.

Weil die einleitenden Phrasen schon bekannt sind, konzentriere ich mich als Erstes auf die Parameterliste (**self, led, buzz, duration=dauer**).

Zur Wiederholung: Wir befinden uns in der Definition einer Klasse und daher hat jede Instanz-Methode als ersten Parameter self. Der verweist auf das Objekt, welches später von der Klasse abgeleitet wird, nicht etwa auf den Inhalt des Funktionskörpers oder gar die Klasse. Die Angabe dieses Parameters ist obligatorisch, sprich notwendig. Gleiches gilt für das Präfix self bei Instanzattributen.

Es folgen zwei sogenannte Positionsparameter, **led** und **buzz**. Diese werden vom Interpreter beim Aufruf der Funktion mit den, an dieser Position angegeben Argumenten belegt. **Positionsparameter müssen beim Aufruf angegeben werden** und zwar **in genau der Abfolge der Definition in der Parameterliste**.

duration = dauer ist ein **optionaler Parameter**. Er kann beim Aufruf einfach nur durch einen Wert angegeben oder gezielt durch ein Paar Name=Wert gesetzt werden. Er kann aber auch weggelassen werden, dann wird der Default-Wert aus der Funktionsdefinition eingesetzt. Die Beispiele zeigen das. Aber **Positionsparameter müssen stets vor den optionalen Parametern** aufgeführt werden!

```
>>>from beep import BEEP

>>> b = BEEP(2,13,700)
Konstruktor von BEEP
LED:2, Buzz:13, dauer=700

>>> c = BEEP(2, 13, duration=1000)
Konstruktor von BEEP
LED:2, Buzz:13, dauer=1000
```

Im folgenden Beispiel wurde die Reihenfolge vertauscht; das funktioniert nicht.

```
>>> a = BEEP(2, duration=1000, 13)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
SyntaxError: non-keyword arg after keyword arg
```

Positionsparameter stehen in der Parameterliste stets am Anfang. Optionale Parameter können folgen, müssen das aber nicht unbedingt, und auch deren Reihenfolge ist frei, sofern die Paare Name= Wert angegeben werden. Der Aufruf der Timer-Initialisierung ist ein Beispiel dafür.

```
self.tim.init(mode=Timer.ONE_SHOT,period=puls,callback=lambda t: self.beepOff())
```

Was für normale Funktionen gilt, trifft natürlich auch für den Konstruktor von BEEP zu. Die GPIO-Nummer des LED- und des Buzzer-Pins müssen zwingend angegeben werden, die beep-Dauer ist optional. Wird sie auf keine der zulässigen Arten beim Aufruf von BEEP() angegeben, setzt die __init__-Funktion sie auf 15. Damit man sehen kann, ob der Konstruktor gelaufen ist und was er getan hat, habe ich ihm die beiden Print-Anweisungen eingebaut.

Was tut der Konstruktor sonst noch? Er definiert die beiden Pin-Instanzen **ledPin** und **buzzPin**, die Timer-Instanz **tim0** und setzt die Beepdauer des erzeugten BEEP-Objekts auf die Voreinstellung oder das übergebene Argument. Außerdem schaltet er durch die Instanzmethode **beepOff** Buzzer und LED aus und erzeugt so einen definierten Ausgangszustand.

Sie erkennen hier genau, welches Attribut und welche Funktion an eine Instanz der Klasse BEEP gebunden ist. Sie alle werden durch den Vorsatz (aka Prefix) **self** gekennzeichnet. Das funktioniert prächtig bei Aufrufen von Methoden und Referenzen auf Variablen innerhalb der Definition der Klasse. Die Methoden **beep**() und **beepOff**() machen davon Gebrauch. Eine Übergabe von Instanzattributen als Parameter einer Funktion ist grundsätzlich nicht möglich und auch nicht nötig, da ein Objekt natürlich seine hauseigenen Variablen auch innerhalb von hauseigenen Funktionen nutzen kann. Einen besonderen Fall stelle ich nach den ersten Experimenten vor.

Im Gebrauch sieht das dann so aus, wir kennen derartige Aufrufe ja bereits aus verschiedensten Anwendungen.

Die Dauer wird beim Konstruktoraufruf weggelassen, daher wird automatisch die Vorgabe der statischen Variable DAUER verwendet.

```
>>> from beep import BEEP
>>> c=BEEP(2,13)
Konstruktor von BEEP
LED:2, Buzz:13, dauer=15
```

Den folgenden Instanzen werden andere Werte zugewiesen.

```
>>> b=BEEP(2,13, 400)
Konstruktor von BEEP
LED:2, Buzz:13, dauer=400

>>> d=BEEP(2,13, duration=1000)
Konstruktor von BEEP
LED:2, Buzz:13, dauer=1000

>>> c.dauer
15
```

Zum folgenden Test gibt es auch eine Hausaufgabe am Ende des Beitrags.

```
>>> from beep import BEEP
>>> b=BEEP(2,13, 400)
Konstruktor von BEEP
LED:2, Buzz:13, dauer=400

400ms-Signal
>>> b.beep()

>>> b.setDuration(800)

800ms-Signal
>>> b.beep()

>>> b.getDuration()
800

800ms überschreiben
>>> b.beep(6000)

Vor Ablauf der 6 Sekunden
>>> b.beepOff()

>>> b.getDuration()
800

Klassendefaultwert 15ms setzen
>>> b.setDuration()
>>> b.beep()
>>>
```

Das Modul ist für ESP32 und ESP8266 gleichermaßen geeignet, da keine Ressourcen verwendet wurden, die nur auf dem ESP32 vorhanden sind. Die Pins auf dem ESP8266 müssen nur so gewählt werden, dass sie keine anderen Funktionen beeinträchtigen.

Werfen wir noch einen Blick auf die Methode beep()

```
def beep(self, puls=None):
  if puls == None:
    tick = self.dauer
  else:
    tick = puls
  self.buzzPin.value(1)
  self.ledPin.value(1)
  self.tim.init(mode=Timer.ONE_SHOT,period=tick,callback=lambda t: self.beepOff())
```

Ohne Argument soll der in self.dauer hinterlegte Wert der Instanz verwendet werden. Der kann aber nicht in die Parameterliste aufgenommen werden, auch nicht als optionaler Parameter. Alternativ soll ein angegebenes Argument als Pulsdauer

dienen. Der Trick besteht nun darin, dass in der Parameterliste als Default None (auch 0 wäre möglich) vorgelegt wird. Im if-Konstrukt wird darauf getestet. Statt None wird also der eigentlich gewünschte Standardwert gesetzt und sonst das übergebene Argument.

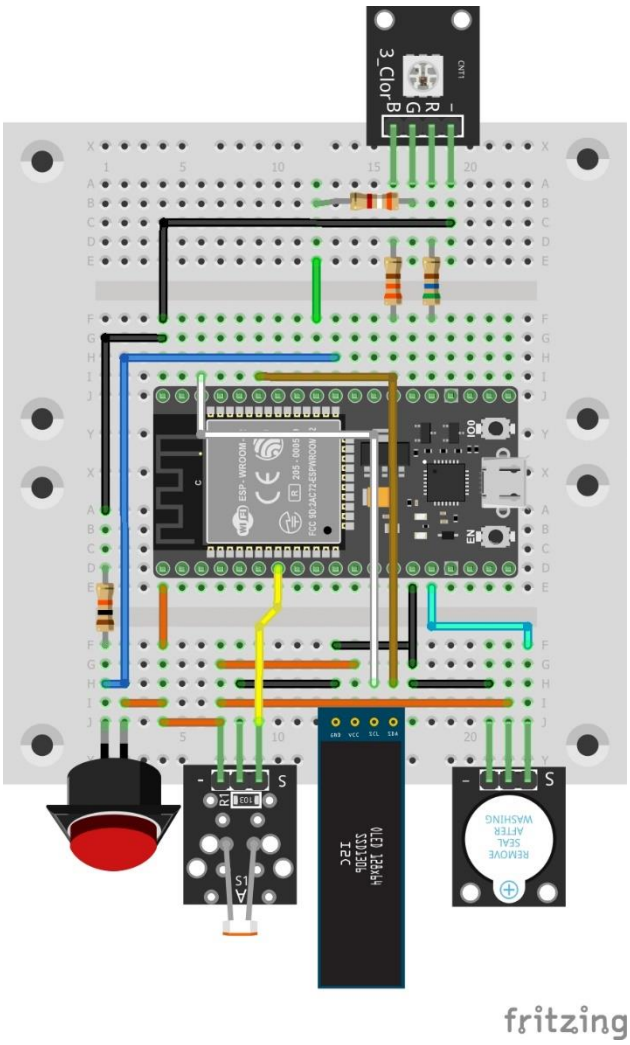
Weil man an Variablen und dergleichen nicht unmittelbar herumpfriemeln soll, wird hier übrigens der Wert von self.dauer durch Methoden gesetzt und abgefragt, so wie es sich gehört.

OLED am ESP32/ESP8266

Damit sich der ESP langsam verselbständigt, verpassen wir ihm jetzt noch ein OLED-Display. Mit so einem sind wir nicht mehr unbedingt auf die Ausgabe von Meldungen über das Terminal angewiesen. Zwei preisgünstige Displays sind diese:

- 0,91 Zoll OLED I2C Display 128 x 32 Pixel für Arduino und Raspberry Pi oder
- 0,96 Zoll OLED I2C Display 128 x 64 Pixel für Arduino und Raspberry Pi

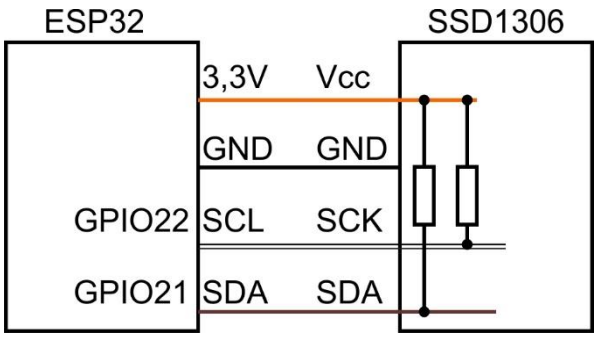
Der Grund, weshalb ich mich für ein OLED-Display entschieden habe und nicht für ein 2x16-LCD, ist ganz einfach der, dass ich mit einem OLED-Display platzsparend auch einfache Grafik darstellen kann. Das geht mit einem rein alphanumerischen LCD nicht. Außerdem gibt es für die OLEDs der Reihe 1306 ein Modul, dessen ich mich bedienen werde.



Hier ist die Verdrahtung auf dem Breadboard. Mit enthalten sind noch das LDR-Modul, das RBG-LED-Modul und der aktive Buzzer sowie der Taster aus Folge 1. Zur Schonung der Augen und ESP-Pins habe ich die Widerstände an der RGB-LED erhöht, um die Helligkeit abzusenken.

Die genannten Displays werden über den I2C-Bus angesteuert. Er besteht aus zwei Leitungen, der Taktleitung SCL (weiß) und der Datenleitung SDA (braun). Beide Leitungen müssen mit einem 4,7 kOhm- bis 10 kOhm-Widerstand gegen Vcc beschaltet werden. Beim SSD1306 ist das auf dem Board durch den Hersteller bereits geschehen. Die Versorgungsspannung kann sich zwischen 3,3V und 5V bewegen. Wir legen Vcc des Displays an den 3,3V-Ausgang des ESP32. Dadurch wird verhindert, dass über die Pullup-Widerstände der SCL- und

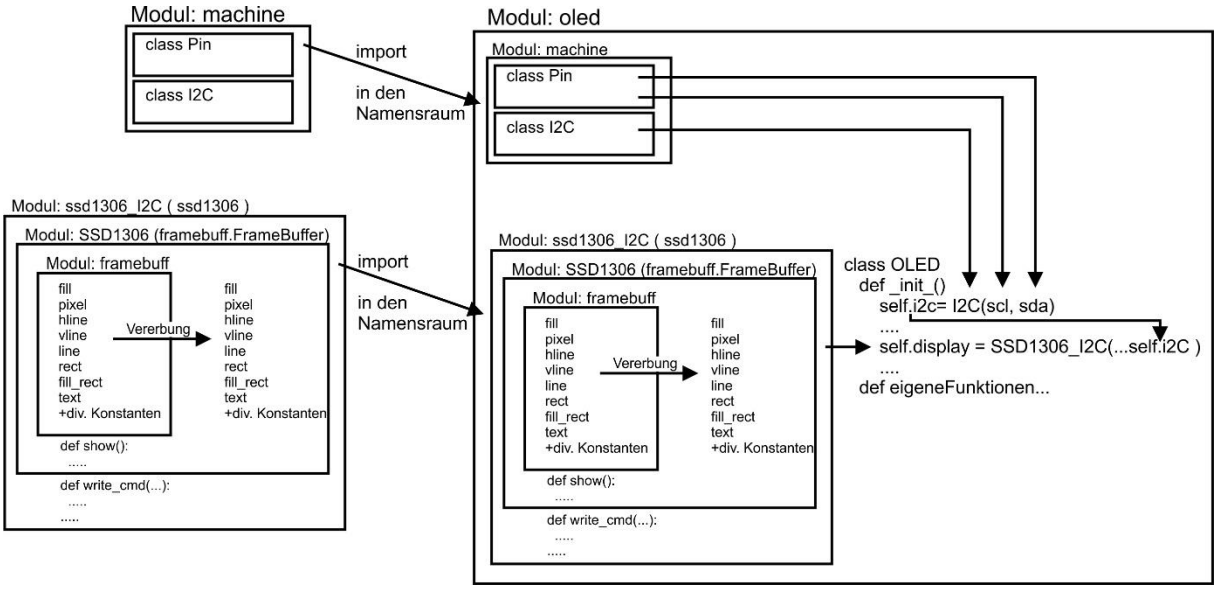
SDA-Leitungen mehr als 3,3V an den Eingängen des ESP32 liegen. GND geht natürlich an GND vom ESP32. Der Strombedarf des Displays ist sehr gering, sodass der Regler auf dem ESP32-Board das noch gut verkraftet. Je nach Controller werden SCL und SDA an unterschiedliche GPIO-Pins gelegt. Beachten Sie bitte die Pinbelegung Ihres OLED-Moduls, die von der in der Schaltskizze abweichen kann.



Controller	SDA	SCL
ESP32	21	22
ESP8266	4 (D2)	5 (D1)

In µPyCraft ist bereits ein Modul **ssd1306** enthalten. Es befindet sich im Ordner **uPy_lib**. Für mein Modul **oled.py** habe ich aber eine andere Quelldatei benutzt. Im Prinzip verwenden beide Varianten dieselbe ursprüngliche Klasse **FrameBuffer** aus dem Modul **framebuf** und stellen auch für den Nutzer letztlich die gleiche API zur Verfügung, aber der Weg von A nach B ist unterschiedlich. Variante 1 importiert framebuffer und erzeugt daraus umständlich neue, gleichnamige Methoden.

Die von mir verwendete ssd1306.py verfügt bereits durch Vererbung über den Namensraum von FrameBuffer und muss keine neuen Methoden definieren. Ich habe versucht, das in einem Diagramm darzustellen.



Laden Sie nun von github die Datei ssd1306.py herunter und kopieren Sie diese im Windows Explorer in Ihren Workspace. Öffnen Sie die Datei dort im Editor und entfernen Sie am Ende der Datei diese Klassendefinition

class SSD1306_**SPI**(SSD1306):

inklusive aller Zeilen bis zum Dateiende. Was Sie entfernt haben, ist das SPI-Interface, welches wir nicht benötigen, weil das Display ja über I2C angesprochen wird. So sparen wir Speicherplatz. Weil wir den I2C-Bus nur mittelbar über das ssd1306-Modul benutzen und nicht direkt ansprechen, gehe ich auch nicht näher auf die Funktionen dieser Art Datenübertragung via I2C ein. Allein das wäre ein Thema für einen eigenen Blogbeitrag.

Damit Sie ssd1306.py als Modul einbinden können, muss es nach dem Speichern im Workspace ins **device directory** des ESP32 hochgeladen werden.

Werfen wir jetzt einen Blick auf das Modul oled.py. Ich werde einzelne Teile genau besprechen, der Rest folgt ansonsten genau dem gleichen Muster.

Mein Ziel war vornehmlich eine zeilen- und spaltenorientierte Textausgabe wie bei einem alphanumerischen LCD. Dazu kam der Wunsch, Textzeilenbereiche gezielt löschen zu können und variable Säulen ab der Grundlinie für ein Diagramm zur Verfügung zu haben, optional x- und y-Achse. Die eingebauten Methoden von ssd1306 waren dabei hilfreich, aber selbst nicht ausreichend. Folgende Methoden habe ich nach meinen Wünschen definiert.

OLED([sclw=SCLPin],[sdaw=SDAPin],[widthw=[64|128]],[heightw=[32|64]])
writeAt(string,xpos,ypos)
clearFT(xv,yv[,xb=spalte][,yb=zeile])
clearAll()
pillar(xpos,breite,hoehe)
setKontrast(wert)
xAxis(), yAxis()
switchOn(), switchOff()

Einige Module bzw. Klassen werden importiert.

```
from machine import Pin, I2C
# ssd1306.py muss sich im device directory befinden
from ssd1306 import SSD1306_I2C
```

Es lohnt sich übrigens, einfach mal einen Blick in die Datei ssd1306.py zu werfen. Man kann dadurch sehr schön nachvollziehen, wie Vererbung und der Import von Namensräumen funktionieren. Eigene Versuche an der Kommandozeile sind dazu hilfreich.

Es folgt der Beginn der Klassendefinition, und ein paar Konstanten werden festgelegt. Dabei gilt es die unterschiedlichen Belegungen für ESP32 und ESP8266 zu berücksichtigen, am besten durch eine Automatik in Form der if-Struktur. Man kann hier übrigens keine Konstanten an SD und SC zuweisen wie bei WIDTH und HEIGHT, weil sonst eine "seltsame" Fehlermeldung erscheint über deren Gründe man hier https://forum.micropython.org/viewtopic.php?t=7989 nachlesen kann.

```
class OLED:
  device = sys.platform
  if device == 'esp32':
    SD =    21
    SC =    22
  elif device == 'esp8266':
    SD =    4
    SC =    5
  else:
    print("Unbekannter Controller!")
    sys.exit()
  WIDTH = const(128)       # Pixelbreite des Displays
  HEIGHT = const(32)       # Pixelhöhe
```

Das Wesentlichste der Konstruktormethode

```
  def __init__(self, sclw=SC, sdaw=SD, widthw=WIDTH, heightw=HEIGHT):
    #ESP32 Pin assignment
    self.columns = widthw // 8
    self.rows = heightw // 10
    self.sd = sdaw
    self.sc = sclw
    self.i2c = I2C(-1, scl=Pin(sclw), sda=Pin(sdaw))
    self.width = widthw
    self.height = heightw
    self.display = SSD1306_I2C(widthw, heightw, self.i2c)
    self.display.contrast(0x3f) # Maximum ist 0xff
    self.display.fill(0)
    print("Konstruktor von OLED")
    print("SDA:{}, SCL:{}, Size:{}x{}".format(self.sd, self.sc, self.width, self.height))
```

Der Konstruktor verwendet 4 optionale Parameter. Das erlaubt uns, die Reihenfolge der Parameter beim Aufruf zu ändern, wenn diese als Name=Wert-Paare übergeben werden. Man kann auch einen Teil der Parameter oder sogar alle weglassen, dann werden die Vorgaben der Kopfzeile verwendet. Ein Aufruf wie der folgende ist möglich. Er richtet den ESP32 mit den Standardwerten für die I2C-Pins ein. Analog verhält es sich beim ESP8266.

```
>>> from oled import OLED
>>> d=OLED()
Konstruktor von OLED
SDA:21, SCL:22, Size:128x32
```

Ein I2C-Objekt wird erzeugt und die Anzahl von Textzeilen und -Spalten wird berechnet. Wir erzeugen ein Displayobjekt für das SSD1306, setzen den Kontrast beziehungsweise die Helligkeit und löschen den Displayinhalt.

Die Methode **clearFT** () löscht den Bereich von Textspalte x / -Zeile y bis Textspalte xb / -Zeile yb.

```
def clearFT(self,x,y,xb=maxcol,yb=maxrow):
  xv = x * 8
  yv = y * 10
  if xb >= self.columns:
    xb = self.columns*8
  else:
    xb = (xb+1) *8
  if yb >= self.rows:
    yb = self.rows*10
  else:
    yb = (yb + 1)*10
  self.display.fill_rect(xv,yv,xb-xv,yb-yv,0)
  self.display.show()
```

Es gibt zwei Positionsparameter, die angegeben werden müssen, das ist die obere linke Ecke im Spalten-Zeilen-Raum. Für die restlichen beiden Parameter müssen keine Werte angegeben werden, sie sind optional. Weiter können für die beiden einfach auch nur Zahlen (Variablen) angegeben werden, dann bleibt es bei der Reihenfolge xb, yb. Will man die Reihenfolge verändern, muss man Name=Wert-Paare angeben. Lässt man die beiden letzten Argumente weg, wird als Default bis zur rechten unteren Displayecke gelöscht.

Damit keine unzulässigen Werte an das Display weitergegeben werden, erfolgt zunächst eine Plausibilitätsabfrage. Dann werden die Spalte-Zeile-Informationen in Pixeldaten umgerechnet und schließlich das entsprechende Rechteck mit der **fill_rect**()-Methode , die letztlich aus der Klasse **FrameBuffer** stammt, zum Löschen **vorbereitet**. Die Methode **show**() bringt die Löschung **zur Anzeige**. **show** ist immer aufzurufen, wenn eine oder mehrere Änderungen am Framebufferinhalt vorgenommen wurden**. Denken Sie an diesen Satz, wenn das Display schwarz bleibt oder die Anzeige sich nicht ändert. In der Regel ist nicht das Display defekt, sondern man hat nur vergessen, show() aufzurufen**. Bei allen Befehlen in der Klasse OLED kann das nicht passieren, denn der Befehl **show**() ist bereits überall integriert. Allerdings ist das mit einer Verlängerung der Laufzeit der schreibenden OLED-Methoden verbunden. Die Hausaufgaben 3. und 4. beschäftigen sich damit.

Während die Methoden aus FrameBuffer displayübergreifend funktionieren, sind show() und einige weitere Methoden für jeweils ganz spezielle Familien von Displays zuständig und deshalb in der Klasse SSD1306 definiert. Die Art, wie Befehle und Daten an das Display gesendet werden, hängt aber nicht nur vom verwendeten Bus ab. Das ist hier der I2C-Bus. Dafür steht die Klasse SSD1306_I2C bereit, die von SSD1306 erbt. Die Vererbung von Klassen werden wir in einem der nächsten Beiträge anschauen.

Die umfangreichste Methode von OLED ist die folgende. Das relativiert sich wieder, wenn man die sechs Kommentarzeilen weglässt.

```
def writeAt(self,s,x,y):
  if x >= self.columns or y >= self.rows: return None
  text = s
  length = len(s)
  xp = x * 8
  yp = y * 10
  if x+length < self.columns:
    b = length * 8
  else:
    b = (self.columns - x) * 8
    text = text[0:self.columns-x]
  self.display.fill_rect(xp,yp,b,9,0)
  # loesche length Zeichen von xp bis xp+length*8-1 incl
  # Das 1. Zeichen steht an Position xp, xp * 8
  # Zeile ist an Position yp und ist 9 Pixel hoch
  # zu loeschende Pixelpositionen: b
  # Loeschen = Farbe 0
  self.display.text(text,xp,yp)
  self.display.show()
  #print("textuebergabe: {0}, at {1},{2}".format(text,x,y))
  return text
```

Alle drei Parameter sind Positionsparameter und müssen beim Aufruf angegeben werden. **s** enthält den auszugebenden String. Zahlen oder Bytesobjekte müssen vor der Übergabe in Strings verwandelt oder durch Formatbefehle im übergebenen String integriert werden.

Sind **x** oder **y** nicht im zulässigen Spalten- oder Zeilenrahmen, wird die Ausgabe abgebrochen und None zurückgegeben, eben nichts. Sonst erfolgt die Rückgabe des tatsächlich ausgegebenen Textes.

Wir bestimmen die Länge **length** des Strings und die Pixelpositionen für das Löschen des Bereichs, in welchem der String landen soll. Dann wird geprüft, ob der String in die restlichen Spalten ab Spalte x passt und wir berechnen die Breite **b** des Löschrechtecks = Anzahl gültiger Zeichenpositionen mal 8 Pixel Zeichenbreite. Notfalls wird der String der Zeichen beraubt, die über den Displayrand hinausgehen würden (Slicing). Dann löschen wir den Zeilenbereich, schicken den moderierten Text in den Framebuffer mit **display.text()**, lassen den Framebuffer mit **show**() anzeigen, geben zur Kontrolle alles evtl. noch am Terminal aus und liefern den moderierten Text zurück. Das aufrufende Programm kann daran durch Vergleich mit dem Argument beim Aufruf erkennen, ob der Text komplett oder nur verstümmelt ausgegeben wurde.

Die letzte Methode, die ich bespreche ist **pillar**(). Pillar stammt aus dem englischen Sprachbereich und bedeutet Säule. Das ist auch die Aufgabe dieser Methode, sie soll eine Säule in ein Diagramm zeichnen.

```
def pillar(self,x,b,h):
  if x+b > self.width or b<=2 or x < 0: return None
  if h > self.height: h=self.height
  y=self.height-h
```

```
    self.display.fill_rect(x,y,b,h,1)
    self.display.show()
    return h
```

Alle drei Positionsparameter, die **müssen** …, richtig, angegeben werden. x ist die Position der linken unteren Ecke, **b** die Breite und **h** die gewünschte Höhe der Säule oder besser des Rechtecks, das die Säule darstellen soll.

Position **x** und Breite **b** werden auf Plausibilität und Anwendbarkeit geprüft. Ist die Position zu weit rechts oder links oder ist die Breite zu mickerig, wird nichts gezeichnet und None zurückgegeben.

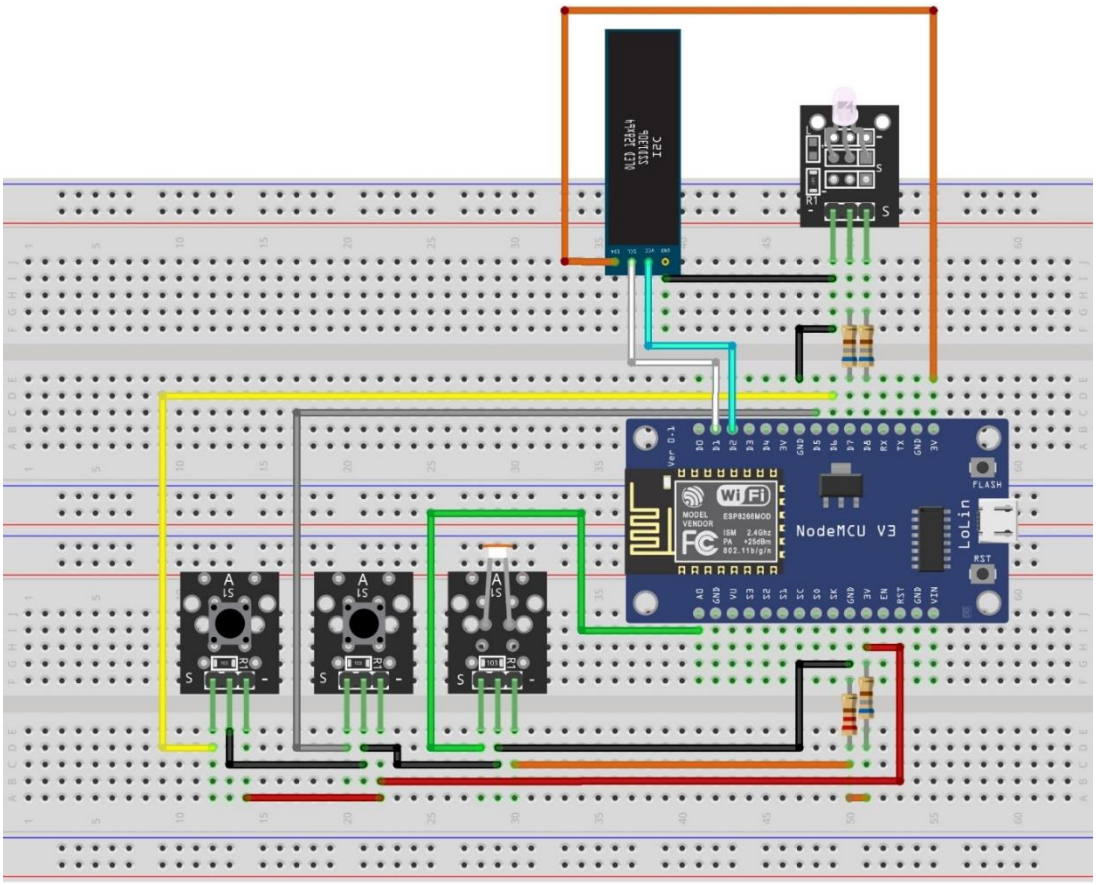
Die Höhe wird notfalls auf die maximale Höhe zurechtgestutzt. Dann lassen wir die richtigen Bits im Framebuffer setzen, das macht wieder **fill_rect**() und die Methode **show**() schickt die Daten zum Display.

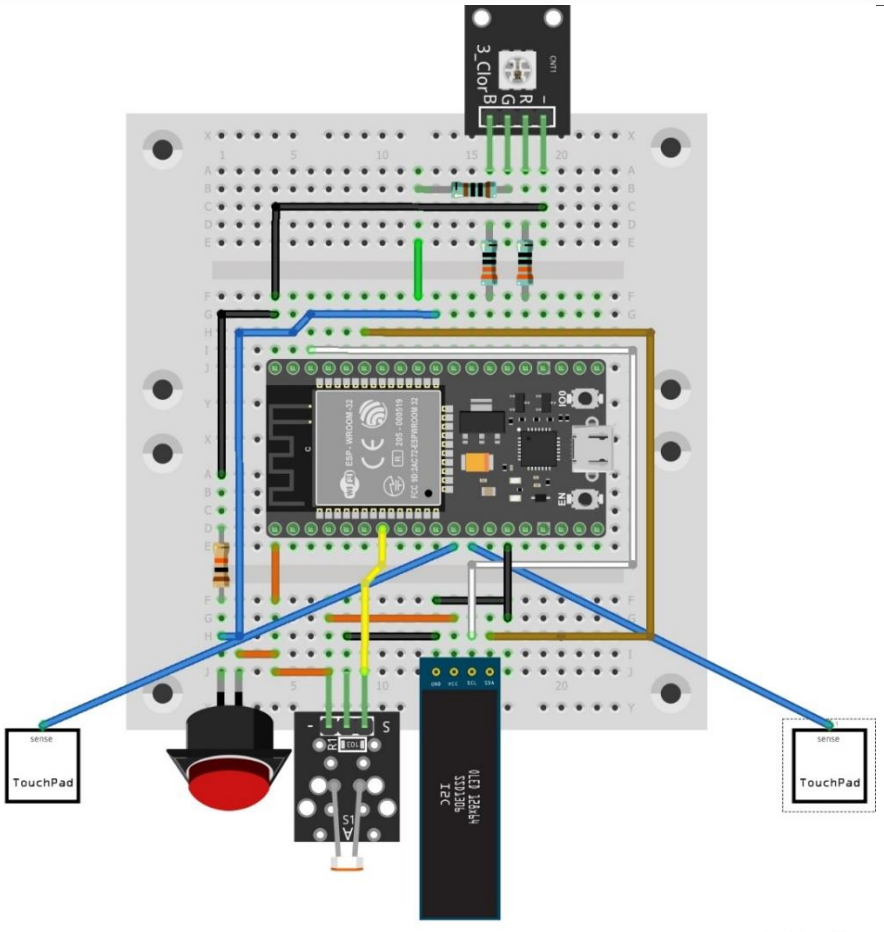
Der Rest der Methoden ist so kurz und übersichtlich, dass eine Erläuterung wohl nicht mehr nötig ist. Die Datei **oled.py** können Sie herunterladen. Eine Beschreibung der Grafikbefehle in **FrameBuffer** finden Sie in der Dokumentation von MicroPython.

Abschließend stellen wir mit Hilfe zweier Touchpads den Kontrast des OLED-Displays ein. Alternativ können wir die Umgebungshelligkeit mit dem LDR erfassen und diesen Wert, entsprechend angepasst, als Stellgröße für eine automatische Kontraststeuerung des Displays einsetzen, wie man sie vom Smartphone kennt. "Ganz nebenbei" erfahren Sie, wie man Touchpads sehr gut als Ersatz für mechanische Taster verwenden kann.

Das Programm touchtest.py, das ich dafür verwende, können Sie als Ganzes herunterladen. Stellen Sie bitte sicher, dass die folgenden Dateien im device Directory des ESP liegen.
touch.py bzw touch8266.py
oled.py
ssd1306.py
Transferieren Sie nun auch touchtest.py dort hin.

Jetzt kommt die Hardware. Das OLED-Display muss noch am ESP8266 / ESP32 angeschlossen werden, die beiden Touchpads / Taster und der LDR sind vermutlich noch mit dem ESP verbunden. Die Abbildungen zeigen dennoch die Teile und Verbindungen für dieses Experiment.


NodeMCU V3
Wi Fi
FLASH
RST
fritzing


3_ClorB
R
G
B
sense
TouchPad
sense
TouchPad
fritzing

Wir gehen einmal den umgekehrten Weg und schauen uns zuerst an, was das Programm tut. Danach untersuchen wir, wie es das macht.

Boardauswahl
Damit das Programm sowohl für ESP32 als auch ESP8266 funktioniert, gleich am Beginn die automatische Auswahl des Boards mit der Initialisierung der Pads beziehungsweise Taster.

Touchpad-Funktionstest
Der erste Teil ist ein Test ob die Touchpads richtig funktionieren. Zehn Sekunden lang kann man die Pads berühren oder Taster drücken. Dabei werden die eingelesenen Werte am Terminal ausgegeben.

Programmfortführung nach Berührung
Das Programm wartet bis zu 10 Sekunden auf eine Aktion an der "Up"-Einheit, um das Programm fortzusetzen.

Kontrast über die Pads einstellen
Die Einheit "Up" erhöht bei Betätigung den Kontrastwert um 10, die andere, "Down", senkt ihn um 10. Erfolgt 5 Sekunden lang keine Aktion, arbeitet das Programm weiter.

Zum Schluss kann man über Abdunkeln oder Beleuchten des LDR das Display dunkler oder heller machen.

Starten Sie also das Programm touchtest.py und probieren Sie es aus.

Jetzt zur Arbeitsweise.

```
import sys
from machine import Pin, ADC
device = sys.platform
if device == 'esp32':
  from touch import TP
  up = TP(27)
  down = TP(14)
  ldr = ADC(Pin(32))
  ldr.atten(ADC.ATTN_11DB) #Full range: 3.3v
  ldr.width(ADC.WIDTH_10BIT)
elif device == 'esp8266':
  from touch8266 import TP
  up = TP(14)    # D5
  down = TP(12)  # D6
  ldr = ADC(0)
else:
  print("Unbekannter Controller")
  sys.exit()
from time import sleep, time
from oled import OLED

d=OLED()
```

Wir importieren diverse Klassen zur Bedienung der Eingabeeinheiten, für OLED, ADC und Zeitsteuerung. Die Touchpad-Instanzen **up** und **down** an GPIO27 und 14 am ESP32 beziehungsweise die Tastereingänge an GPIO14 und GPIO12 am ESP8266 sowie ein Displayobjekt **d** werden erzeugt. Der analoge Anschluss wird zugewiesen und beim ESP32 die gleiche Auflösung wie beim ESP8266 eingestellt.

```
print ("Touchpad-Funktionstest")
i = 0
while i < 10:
  print(up.getTouch(),down.getTouch(),"up - down", 10-i)
  sleep(1)
  i += 1
```

Im Sekundenabstand lesen wir die Eingabewerte ein und schicken sie zusammen mit einem Countdown ans Terminal.

```
print("Warten auf Berührung")
wert = up.waitForTouch(10)
print(wert)
```

Die Klasse TP erleichtert die Programmführung durch das Berühren von Touchpads. Schalter oder Tasten können das auch leisten, unterliegen aber einer mechanischen Abnutzung und sie prellen. Beim ESP8266 müssen wir dennoch darauf zurückgreifen. Alternativ können Sie Touchpad-Module verwenden. In der Materialliste habe ich eines aufgelistet, das keypad-ttp224-14-kapazitiv.

Aber durch die umsichtige Programmierung der Module touch.py und touch8266.py ist nur ein Testprogramm für beide Controllerfamilien nötig.

Der umfangreichste Teil des Programms ist die manuelle Helligkeitssteuerung.

```
k=32
d.setKontrast(k)
delay=5
schritt=10
start=time()
end=start+delay
current=start
while current < end:
  plus=up.getTouch()
  minus=down.getTouch()
  if up.threshold:
    plus = (plus < up.threshold)
  if down.threshold:
    minus = (minus < down.threshold)
  if plus:
    k=(k+schritt if k<255-schritt else 255)
    end=time()+delay
  elif minus:
```

```
    k=(k-schritt if k>schritt else 0)
    end=time()+delay
  d.setKontrast(k)
  print(k)
  sleep(0.2)
  current=time()
```

Nach dem Setzen der Anfangswerte geht es in eine Zeitschleife. Die Eingabeeinheiten werden abgefragt Während für den ESP8266 die Tasten bereits ein boolsches Ergebnis, 0 = False oder 1 = True, liefern, muss das in diesem Fall für die Fuzzy-Logic der Touchpads erst hergestellt werden. Falls also der threshold-Wert der Eingänge nicht None (= False) ist, nämlich für die Touchpads, wird der eingelesene Wert in ein True überführt, wenn er zunächst unter dem Grenzwert lag, das Pad also berührt wurde.

Danach wird je nach Eingabe der für die Helligkeit in 10-er Stufen erhöht oder abgesenkt. Die Konditionalausdrücke begrenzen den Bereich auf 0 .. 255. War eine Eingabe erfolgt, wird außerdem die Endezeit der Schleifendurchläufe neu gesetzt, damit man ohne Stress die Einstellung tätigen kann.

Zum Abschluss wird die Helligkeit auf den letzten Wert eingestellt, dieser ausgegeben und nach dem Ausbremsen der Durchlaufgeschwindigkeit die laufende Zeit aktualisiert. Wenn 5 Sekunden lang keine Eingabe mehr erfolgte, bricht die Schleife ab und der letzte Teil des Programms mit der Abfrage des ADC-Werts und seiner Umsetzung auf den Bereich des Kontrastwerts läuft an.

Fassen wir zusammen, was Sie in dieser Folge gelernt haben.

- Sie wissen mehr über Namensräume von Variablen, Funktionen und Klassen.
- Sie können Timer in Ihren Programmen einsetzen
- Sie können ein OLED-Display wie ein herkömmliches LCD benutzen.
- Sie haben eigene Module und Klassen definiert und dabei etwas über die Speichernutzung von MicroPython erfahren
- Sie können abschätzen, wann es besser ist, ganze Module zu importieren und wann Sie selektiv Klassen oder einzelne Members importieren sollten.
- Sie können Objekte vergleichen und deren Identität prüfen
- Sie haben gesehen, wie man Touchpads oder Taster in Verbindung mit einem Modul zur Steuerung von Prozessen einsetzen kann

In der nächsten Folge bauen wir die Hardware für das finale Projekt. OLED, Taster, Touchpads, LEDs und Tweeter werden wieder zum Einsatz kommen. Und das System kann autonom aufgebaut werden. Das heißt, der ESP startet automatisch durch, erlaubt einige Aktionen zur Vorbereitung und wartet dann auf den Start in die Messschleife. Was gemessen wird? – Bleiben Sie gespannt!

Sie können diese Folge des Blogs auch als PDF herunterladen.

Wenn Sie eine Folge verpasst haben, hier sind die Links zu den Folgen eins bis drei:
Folge 1
Folge 2
Folge 3

Neue Hausaufgaben

1. Nehmen wir an, Sie führen folgende Eingaben über die Kommandozeile aus:
   >>> from beep import BEEP
   >>> BEEP.dauer = 5000

   >>> BEEP.dauer
   5000

   >>> b=BEEP(2,13)
   ???
   ???
   >>> b.beep()
   ???

   Wie reagiert der ESP auf den letzten Befehl? Können Sie das Verhalten begründen?

2. Lassen Sie die Ausgaben in der Datei touchtest.py auf dem OLED-Display ausgeben, statt am Terminal. Wann macht das Sinn, wann nicht?

3. Messen Sie die Geschwindigkeit, die mit der Ausgabe am Terminal und am OLED-Display erreichbar ist.

4. Macht es für die Geschwindigkeit einen Unterschied, wie viele Zeichen am Display und am Terminal pro Durchgang ausgegeben werden?

5. Erzeugen Sie mit dem pillar()-Befehl und dem Zufallszahlengenerator aus der Hausaufgabenlösung ein Säulendiagramm aus 10 Balken, das die Anzeigefläche eines Displays möglichst voll nutzt. Denken Sie dran, dass es Displays mit verschiedener Breite gibt.

6. Bereiten Sie zwei verschiedene Ausgaben für das OLED-Display vor, zum Beispiel einen Textschirm und ein Säulendiagramm. Lassen Sie beides im Wechsel von 3 Sekunden anzeigen, während (also gleichzeitig!) im Terminal die Ausgabe der Zahlen von 1 bis 30 nach folgendem Schema im Dauerlauf erfolgt.
   1
   12
   123
   1234
   12345
   …

7. Bauen Sie den Reaktionstester aus der Hausaufgabenlösung so um, dass bei jedem Durchgang eine andere LED scharf ist. Natürlich müssen sie dem Spieler über das OLED sagen, welche Farbe das jeweils ist.

8. Können Sie es einrichten, dass in einem neuen Modul touch die Variante ESP32 und die Variante ESP8266 vorgehalten werden und je nach Controllertyp beim Import automatisch die richtige zum Einsatz kommt?

Lösungen der Hausaufgaben aus Teil 3

1. Schreiben Sie eine Sequenz, die in der Methode __init__() den Parameter grenzwert auf gültige Angaben überprüft. Der Wert muss ganzzahlig, positiv und kleiner als 256 sein.

Eine erste Trockenübung könnte wie folgt aussehen. Sie können eine Programmdatei anlegen oder einfach nur REPL nutzen.

Download: hausi1a.py

```
# hausi1a.py                                                    #1
grenzwert = 345                                                 #2
grenzwert = int(grenzwert)                                      #3
grenzwert = (grenzwert if grenzwert >0 and grenzwert <256 else 64) #4
print("Als Grenzwert wird {} verwendet.".format(grenzwert))        #5

#1 Dateiname[, Version, Autor, Zweck]
#2 Grenzwert zum Test angeben
#3 Ganzzahligen Wert erzwingen
#4 Bereich abchecken
#5 Ergebnis mitteilen
```

Eine evtl. angegebene Fließkommazahl wird durch Zeile 3 zur Ganzzahl gestutzt. Der bedingte Ausdruck (aka conditional expression) in Zeile 4 weist der Variable grenzwert ihren Wert wieder zu, wenn er im richtigen Bereich liegt, sonst wird 64 zugewiesen.
Zeile 5 zeigt das Ergebnis.

Download: hausi1b.py

```
# hausi1b.py                                                    #1
import sys                                                      #2
grenzwert = "abcd"                                              #3
try:                                                            #4
  grenzwert = int(grenzwert)                                    #5
except ValueError:                                              #6
  print("Falsche Angabe: {}\nBitte eine Ganzzahl angeben".format(grenzwert)) #7
  sys.exit()                                                    #8
grenzwert = (grenzwert if grenzwert >0 and grenzwert <256 else 64)  #9
print("Als Grenzwert wird {} verwendet.".format(grenzwert))        #10

#1 Dateiname[, Version, Autor, Zweck]
#2 Modul sys importieren fuer exit()-Funktion
#3 Grenzwert zum Test angeben
#4 Versuch der Umwandlung in einen ganzzahligen Wert
#5 Umwandlung
#6 Ausnahmebehandlung, falls der Versuch misslingt
#7 Fehlermeldung
#8 Programmabbruch
#9 Falls die Umwandlung erfolgreich war, wird der Exceptblock Ã¼bersprungen
#10 Meldung des Ergebnisses
```

Die zweite Variante fängt einen Fehler ab, wenn versucht wird, eine Angabe in eine Ganzzahl zu verwandeln, für die das nicht möglich ist, z. B. im Fall eines Strings. Es wird dann eine Fehlermeldung ausgegeben und das Programm abgebrochen. Sie sollten dieses Beispiel als Datei ausführen, damit Sie sehen, wie es läuft.

Die Variante 3 meldet auch für die Bereichsüberschreitung einen Fehler und bricht ab.

Download: hausi1c.py

```
# hausi1c.py                                                    #1
import sys                                                      #2
grenzwert = 45.9                                                #3
try:                                                            #4
  grenzwert = int(grenzwert)                                    #5
except ValueError:                                              #6
  print("Falsche Angabe: {}\nBitte eine Ganzzahl angeben".format(grenzwert)) #7
  sys.exit()                                                    #8
if grenzwert <0 or grenzwert >255:                              #9
  print("Falsche Angabe: {} liegt nicht im Bereich zwischen 0(incl.) und
256(excl.)".format(grenzwert))
  sys.exit()                                                    #11
print("Als Grenzwert wird {} verwendet.".format(grenzwert))        #12

#1 Dateiname[, Version, Autor, Zweck]
#2 Modul sys importieren fuer exit()-Funktion
#3 Grenzwert zum Test angeben
#4 Versuch der Umwandlung in einen ganzzahligen Wert
#5 Umwandlung
#6 Ausnahmebehandlung, falls der Versuch misslingt
#7 Fehlermeldung
#8 Programmabbruch
#9 Bereichspruefung
#10 Fehlermeldung
#11 Abbruch
#12 Meldung des Ergebnisses
```

Abschließend die Lösung für touch.__init__()

Download: hausi1.py

```
# hausi1.py
# Ausschnitt aus touch.TOUCH.__init__()
# Ueberpruefung des Grenzwerts
  # touch related methods
  # *********************************************
  def __init__(self, pinNbr, grenzwert=Grenze):
    self.number = pinNbr
    self.tpin = TouchPad(Pin(pinNbr))
    gw = int(grenzwert)                                         #3
    gw = (gw if gw >0 and gw <256 else 64) #4
    print("Als Grenzwert wird {} verwendet.".format(gw))        #5
    self.threshold = gw
```

2. Erstellen Sie eine Plausibilitätskontrolle in __init__, welche die Gültigkeit der GPIO-Nummer für den Touchpin überprüft bevor das Objekt erzeugt wird. Definieren Sie hierfür eine Liste mit den gültigen Pinnummern. Ob die Eingabe einem Wert der Liste entspricht, prüfen Sie mit dem Schlüsselwort in nach. Beispiel für die Kommandozeile:

```
>>> eingabe = 4
>>> touchliste = [15,2,0,4,13,12,14,27,33,32]
>>> eingabe in touchliste
True
```

Hier ein Beispiel als eigenständiges Testprogramm:

Download: hausi2a.py

```
from machine import Pin,TouchPad
import sys
pinNbr=7
touchliste = [15,2,0,4,13,12,14,27,33,32]                    #7
if not pinNbr in touchliste:                                 #8
  print("{} ist keine gueltige GPIO-Nummer fuer Touchpins".format(pinNbr)) #9
  sys.exit()                                                 #10
tpin = TouchPad(Pin(pinNbr))
print("Pin({}) als Touchpin eingerichtet".format(pinNbr))
```

Die Konstruktormethode in touch.TP sieht dann so aus:

Download: hausi2.py

```
# hausi2.py
# Ausschnitt aus touch.TOUCH.__init__()
# Ueberpruefung des Grenzwerts
  # touch related methods
  # *********************************************
import sys
  def __init__(self, pinNbr, grenzwert=Grenze):
    touchliste = [15,2,0,4,13,12,14,27,33,32]                  #7
    if not pinNbr in touchliste:                               #8
      print("{} ist keine gueltige GPIO-Nummer fuer Touchpins".format(pinNbr)) #9
      sys.exit()                                               #10
    self.number = pinNbr
    self.tpin = TouchPad(Pin(pinNbr))
    gw = int(grenzwert)
    gw = (gw if gw >0 and gw <256 else 64)
    print("Als Grenzwert wird {} verwendet.".format(gw))
    self.threshold = gw
```

#7 Liste gueltiger Touchpins
#8 Pruefung, falls pinNbr nicht iin der Liste vorkommt,
#9 Fehlermeldung und
#10 Abbruch

3. Können Sie waitForTouch so verändern, dass bei Berührung innerhalb der Laufzeit statt dem Wert des Touchpads die Verzögerung von Start der Methode bis zum Zeitpunkt des Berührens zurückgegeben wird?

Ich habe für meine Lösung etwas ausgeholt und die Zeitmessung in Sekunden, im Hinblick auf Aufgabe 4, durch eine in Millisekunden ersetzt. Das Beispielprogramm zeigt außerdem den Einsatz einer Klasse direkt im Programm. Deswegen entfällt hier der Import der Klasse TP. Die Lösungen zu Aufgabe 1 und 2 sind bereits in der Klassendefinition berücksichtigt.

Download: hausi3a.py

```
from machine import Pin, TouchPad
from time import time, ticks_ms

class TP:
  # touch related values
  # Default-Grenzwert fuer Beruehrungsdetermination
  # ***********************************************
  Grenze = const(150)

  # touch related methods
  # ***********************************************
  def __init__(self, pinNbr, grenzwert=Grenze):
    touchliste = [15,2,0,4,13,12,14,27,33,32]                    #7
    if not pinNbr in touchliste:                                 #8
      print("{} ist keine gueltige GPIO-Nummer fuer Touchpins".format(pinNbr)) #9
      sys.exit()                                                 #10
    self.number = pinNbr
    self.tpin = TouchPad(Pin(pinNbr))
    gw = int(grenzwert)
    gw = (gw if gw >0 and gw <256 else 64)
    print("Als Grenzwert wird {} verwendet.".format(gw))
    self.threshold = gw

  # Liest den Touchwert ein und gibt ihn zurueck. Im Fehlerfall wird
  # None zurueckgegeben.
  def getTouch(self):
    # try to read touch pin
    try:
      tvalue = self.tpin.read()
    except ValueError:
      print("ValueError while reading touch_pin")
      tvalue = None
    return tvalue

  # delay = 0 wartet ewig und gibt gegf. einen Wert < threshold zurueck
  # delay <> 0 wartet delay Sekunden, wird bis dann kein Touch bemerkt,
  # wird None zurueckgegeben, sonst ein Integer, der der Beruehrung enspricht
  def waitForTouch(self, delay):
    laufzeit = delay*1000
    start = ticks_ms()
```

```
    end = (start + laufzeit if delay >0 else start+10000)
    current = start
    while current < end:
      val = self.getTouch()
      if (not val is None) and val < self.threshold:
        return current-start
      current = ticks_ms()
      if delay==0:
        end=current+10000
    return None

t=TP(27)
zeit=t.waitForTouch(5)
print("{} ms".format(zeit))
```

Die fettformatierten Zeilen müssen geändert beziehungsweise ergänzt werden. Das Touchpad ist an GPIO27 angeschlossen.

4. Bauen Sie Aufgabe 3 zu einem Reaktionstestgerät aus, indem Sie zu Beginn der Laufzeit eine LED aufleuchten lassen, die nach dem Berühren wieder ausgemacht wird.

Die letzten drei Zeilen aus dem Beispiel hausi3a.py werden durch folgende Zeilen ersetzt.

Download: hausi4a.py

```
ledG=Pin(18,Pin.OUT)
ledG.off()
t=TP(27)
sleep(4)
ledG.on()
zeit=t.waitForTouch(5)
ledG.off()
print("{} ms".format(zeit))
```

Nach einem Vorlauf von 4 Sekunden leuchtet die grüne LED auf (Aufbau aus Teil3 mit RGB-LED). Ab dann läuft die Zeit bis zum Touch.

Steigerungsform:
Lassen Sie zufallsgesteuert eine von drei LEDs aufleuchten wobei nur bei einer getouched werden darf. Tippt man bei einer falschen LED, wird man degradiert. Die richtigen Touches werden mitgezählt.

Ersetzen Sie die drei letzten Programmzeilen aus hausi3a.py wie folgt:

Download: hausi4b.py

```
import os
runden = 0
ledG=Pin(18,Pin.OUT)
ledG.off()
ledR=Pin(2,Pin.OUT)
```

```
ledR.off()
ledB=Pin(4,Pin.OUT)
ledB.off()
ledList=[ledG,ledR,ledB]              #60
t=TP(27)                              #61
testLed=2                             #62
for i in range(11):                   #63
  led = os.urandom(1)[0] & 0x03       #64
  led = (led if led != 3 else 2)      #65
  ledx =ledList[led]                  #66
  sleep(4)                            #67
  ledx.on()
  #print(led)
  if led==testLed:                    #70
    zeit=t.waitForTouch(3)
    if zeit and zeit < 500:
      runden += 1
      print("{} ms".format(zeit))
    else:
      print("Leider zu langsam")
  else:                               #77
    zeit=t.waitForTouch(2)
    if zeit:
      runden -= 1
      print("Falsch getouched, Punktabzug")
    else:
      runden += 1
      pass
  ledx.off()                          #84
print("Treffer: {}".format(runden))
```

Zur Funktion des Programms:
#60 Die drei LEDs sind definiert, ich übernehme die Objekte in die Liste ledList.
#61, #62 Touchpin definieren und die "scharfe" LED festlegen
#63 10 mal wiederholen
#64, #65 Eine Zufallszahl 0 <= led <=2 würfeln.
#66 Das LED-Objekt aus der Liste holen
#67 Warten und LED an
#70ff Touchpad freigeben, Zeit messen, evtl Treffer erhöhen oder trösten
#77ff falsch getoucht, Punktabzug, sonst Score erhöhen
#84 Ergebnis der Testreihe

5. Was passiert, beim ESP32 und beim ESP8266, wenn Sie die letzte Zeile in touchtest8266.py entkommentieren?

Der erkannte Chip wird im Terminalfenster gemeldet, mit jeder Definition der Touchpads wird der Grenzwert ausgegeben und das war's, wenn sie einen ESP32 verwenden.

Im Fall eines ESP8266 wird als Fehler gemeldet, dass das Objekt t kein Attribut threshold besitzt.

6. Finden Sie eine Möglichkeit, den Fehler aufzufangen, der beim Abfragen des Attributs t.threshold auftritt, wenn man das Modul touch8266 benutzt.

Fügen Sie in der touch.TP.__init__() folgende Zeile ein:

self.threshold = None

Damit wird selbst dann, wenn sie

print(t.threshold)

verwenden, None, eben gar nichts, ausgegeben. Aber es gibt auch keine Fehlermeldung und diese Lösung hat den Nebeneffekt, dass Sie jederzeit im Programm über das Threshold-Attribut abfragen können ob Tasten oder Touchpads initialisiert sind. Das ist übrigens nicht nur eine Sache des Controllertyps, denn auch am ESP32 kann man Touchpads natürlich durch Tasten ersetzen.