

# Micropython mit ESP32

<b>Einleitung</b>	<b>3</b>
<b>Lektion 1: Was dich in diesem Kurs erwartet</b>	<b>4</b>
Musst du irgendwelche Voraussetzungen erfüllen?	4
Wie weit möchtest du in die Materie eintauchen?	4
Wie gehen wir vor?	4
Brauchst du exakt die vorgeschlagene Hardware?	5
Wo bekommst du Unterstützung?	5
<b>Lektion 2: Die Hardware</b>	<b>6</b>
Spezifikationen	6
Zur Verfügung stehende Pins	7
Pinout	8
Das Schema	9
<b>Lektion 3: Die Software</b>	<b>10</b>
Die Entwicklungsumgebung	10
Installation	10
Erste Schritte	10
<b>Lektion 4: Digitale Ein- und Ausgänge, Delay</b>	<b>12</b>
Digitale Ausgänge	12
Digitale Eingänge	12
sleep (delay) und ticks (millis)	13
<b>Lektion 5: Das Display</b>	<b>14</b>
Initialisierung	14
Direkte Befehle	14
Grafikbefehle	15
Textbefehle	15
Ausgabe an Zeichenposition	16
<b>Lektion 6: Die Stoppuhr</b>	<b>17</b>
Die Verdrahtung	17
Der Ablauf	17
Initialisierung der Komponenten	18
Eine Funktion zum Abfragen der Tasten	18
Was machen wir, wenn Start gedrückt wird?	19
Was machen wir, wenn die Uhr aktiv ist und wie reagieren wir auf Stopp?	19
Anzeige auf dem Display und die format() - Anweisung	19
Deine Übungsaufgabe	20
<b>Lektion 7: Eine mögliche Lösung</b>	<b>21</b>
Die Stoppuhr soll automatisch ausgeführt werden. Wie geht das?	21
<b>Lektion 8: Druck, Temperatur und Luftfeuchtigkeit (BME280)</b>	<b>22</b>
Überblick	22
Der Versuchsaufbau	22
Die Software	22
<b>Lektion 9: Die Thonny-Version 3.2.3</b>	<b>24</b>
Installation von Thonny	24
Aktualisieren der Micropython - Version auf dem Board	25
Neue Arbeitsweise mit Thonny 3.2.3	26
Der File - Browser	27

<b>Lektion 10: Neue Heltec-Version</b>	<b>28</b>
<b>Lektion 11: Nochmals BME280</b>	<b>30</b>
<b>Lektion 12: WLAN - Verbindung</b>	<b>31</b>
Verbindung aufbauen	31
Die Zeit abfragen	32
<b>Lektion 13: Modularisierung und eine Lösung der Übungsaufgabe</b>	<b>33</b>
Die Struktur unseres Programmes	33
geheim.py	34
netzwerk.py	34
display.py	34
Das Hauptprogramm (Zeitanzeige)	35
<b>Lektion 14: Zugriff auf Webservice</b>	<b>36</b>
Erstellen eines Accounts und eines API - Keys	36
Erste Abfrage im Browser	36
Abruf der Daten mit Python	38
Auslesen der Temperatur aus den Wetterdaten	38
Übungsaufgabe	38
<b>Lektion 15: Eine bessere Darstellung</b>	<b>39</b>
Die Lösung der Übungsaufgabe	39
Ein verbessertes Netzwerkmodul	40
Der Webserver	41
Mehrere Webseiten	42
Darstellung des Wetters	43
<b>Lektion 16: Automatische Aktualisierung und Zeitanzeige</b>	<b>45</b>
Anzeige der Zeit im Log	45
Wetterdaten automatisch aktualisieren	46
Wann wurden die Wetterdaten gemessen?	46
Erweiterte Funktionen und Fehlertoleranz	46
<b>Lektion 17: Speichern von Daten in einer Datei</b>	<b>48</b>
Erzeugen einer Datei, schreiben in eine Datei	48
Lesen einer Datei	49
<b>Lektion 18: Das OneCall API</b>	<b>50</b>
Aufruf	50
Woher bekommst du die Koordinaten?	50
Notwendige Programmänderungen	51
webseiten.py	51
netzwerk20.py	51
openweathermap20.py	51
Garbage Collection (Das Modul gc)	51

## Einleitung

Dieses Handbuch begleitet den Videokurs **Micropython mit ESP32**. Die dazugehörigen Videos sind auf Youtube frei zugänglich. Ich gehe davon aus, dass du die jeweilige Lektion gesehen und nachvollzogen hast.

Mit jeder Lektion wächst das Handbuch um ein Kapitel. Die neuste Version findest du immer im Begleitmaterial zur Lektion. In diesem Begleitmaterial findest du auch alle Source - Codes der Experimente.

Support gibt es ausschliesslich über das Forum.

Auf <https://community.hobbyelektroniker.ch/wbb/index.php?board/40-die-lektionen-zu-micropython-mit-esp32/> findest du für jede Lektion einen eigenen Bereich in dem du Fragen stellen kannst.

Selbstverständlich kannst du dort auch auf Fragen anderer Zuschauer antworten.

Auf Micropython wird soweit eingegangen, wie es für das Verständnis der Experimente notwendig ist. Wenn du eine systematischere Einführung möchtest, empfehle ich dir den parallel laufenden Kurs **Micropython Grundlagen** zu verfolgen. Auch hier gibt es einen Support - Bereich im Forum: <https://community.hobbyelektroniker.ch/wbb/index.php?board/45-die-lektionen-zu-micropython-grundlagen/>

Es wird auch zwischendurch immer wieder einmal eine Übungsaufgabe geben. Versuche diese selbstständig zu lösen. Wenn du nicht weiterkommst, suche im Internet die notwendigen Informationen. Wenn das nichts hilft, dann stelle eine Frage im Forum und diskutiere das Problem mit anderen Kursteilnehmern. Erst ganz zum Schluss solltest du dir die Musterlösung anschauen. Diese Lösung ist immer nur ein Vorschlag. Vielleicht ist deine Lösung sogar besser.

Also, dann kann es los gehen. Ich wünsche dir viel Spass!

## Lektion 1: Was dich in diesem Kurs erwartet

Wir werden uns mit einem etwas weiterentwickelten Mikrokontroller beschäftigen, dem ESP 32. Anders als im Arduino werden wir nicht in der Sprache C oder C++ programmieren, sondern in Micropython. Micropython ist ein auf Mikrokontroller angepasstes Python 3. Du wirst also die Sprache Micropython lernen.

### Musst du irgendwelche Voraussetzungen erfüllen?

Der Kurs ist nicht für absolute Mikrokontroller - Neulinge oder Programmieranfänger geeignet. Wenn du aber schon eine Led an einem Arduino zum Leuchten gebracht hast und den Unterschied von Strom und Spannung kennst, solltest du den Kurs mitverfolgen können. Programmierseitig solltest du wissen, was Variablen sind und wozu Funktionen gut sind. Der Arduinokurs auf meinem Youtube - Kanal ist eine sehr gute Voraussetzung. Python oder ESP32 - Kenntnisse sind nicht notwendig. Ein beliebiger Mikrokontroller und eine beliebige Programmiersprache bilden eine gute Grundlage.

### Wie weit möchtest du in die Materie eintauchen?

Am meisten lernst du natürlich, wenn du alles was ich zeige nachvollziehst und nicht locker lässt, bevor du es vollständig verstanden hast. Dabei wirst du durch gelegentliche Übungsaufgaben unterstützt. Du wirst dann danach in der Lage sein selbstständig ganz andere Projekte zu realisieren.

Teilweise gehe ich aber auf Details ein, die dich vielleicht gar nicht so interessieren. Wir sind hier nicht in der Schule, du kannst also auch Dinge auslassen. Da alle verwendeten Programme zum Download bereit stehen, kannst du sie auch benutzen, ohne deren innere Arbeitsweise vollständig zu verstehen.

Dieser Kurs ist auf Praxis ausgelegt. Wir werden Experimente aufbauen und dabei jeweils die dazugehörigen Befehle von Micropython besprechen. Das wird genügen, falls du schon Kenntnisse von Micropython hast oder gar nicht allzu tief in die Details eintauchen möchtest. Für alle Anderen wird parallel dazu auch der Kurs **Micropython Grundlagen** angeboten. Hier liegt dann der Schwerpunkt auf der Programmierung.

### Wie gehen wir vor?

Alles beginnt mit der Beschaffung der Hardware und der Installation der notwendigen Software. Ich zeige dir die Installation für Mac und Windows. Unter Linux sollte es mehr oder weniger identisch sein.

Danach gibt es einige Grundlagen zum Thema Micropython. Dabei machen wir auch erste Experimente mit unserem ESP32.

Bald schon wird es aber konkret. Wir starten mit dem Projekt 'Wetterstation', das uns den ganzen Kurs begleiten wird. Du lernst, wie die verschiedenen Sensoren abfragt und die gemessenen Werte ausgegeben werden. Die Ausgabe wird zuerst auf die Konsole, später aber auf ein kleines Display erfolgen.

Da der ESP32 WLAN-tauglich ist, werden wir auch eine Ausgabe über einen Webserver erstellen. Damit kannst du mit jedem Computer, jedem Handy oder Tablet im WLAN deine Wetterdaten über den Browser abfragen.

Was ist schon heutzutage eine Wetterstation ohne Wettervorhersagen? Also werden wir über das Internet Wettervorhersagen beziehen und diese ebenfalls darstellen. Das ist aber noch nicht alles, einige Ideen werden sicher noch dazukommen.

Mit jedem Schritt wirst du neue Sprachelemente von Micropython kennenlernen, so dass du am Schluss in der Lage sein wirst, eigene Ideen zu verwirklichen.



## **Brauchst du exakt die vorgeschlagene Hardware?**

Nein, eigentlich nicht. Das hängt aber von deinen Vorkenntnissen ab. Es gibt sehr viele geeignete ESP32 - Module. Wenn du in die notwendige Dokumentation beschaffen kannst und sie auch verstehst, dann ist das kein Problem. Du musst in der Lage sein andere Pins zu verwenden und natürlich dadurch auch die Verdrahtung und die Programme anzupassen. Wenn du hier unsicher bist, dann verwende einfach die vorgeschlagene Hardware.

Bei vollständig anderen Sensoren ist allerdings Vorsicht angebracht. Oft werden diese auf ganz andere Art programmiert und du bist dann auf dich allein gestellt.

## **Wo bekommst du Unterstützung?**

Im Forum selbstverständlich. Die normalen Kommentare auf Youtube eignen sich nicht für ausführlich Antworten auf Fragen.

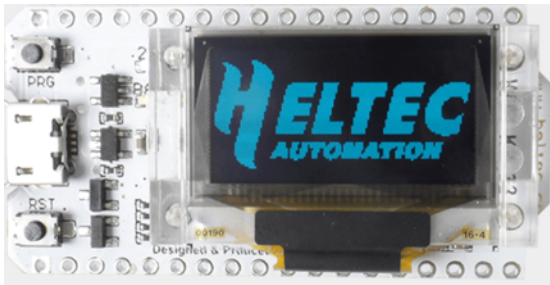
<https://community.hobbyelektroniker.ch/wbb/index.php?board/40-die-lektionen-zu-micropython-mit-esp32/>

und

<https://community.hobbyelektroniker.ch/wbb/index.php?board/45-die-lektionen-zu-micropython-grundlagen/>

Zu jeder Lektion wird es einen Bereich geben, in dem Fragen beantwortet werden. Auch deine Antworten sind selbstverständlich immer sehr erwünscht.

## Lektion 2: Die Hardware



### **Heltec WiFi Kit 32**

<https://heltec.org/project/wifi-kit-32/>

Ich empfehle dir eine Bestellung direkt beim Hersteller, Bangood oder Ali-Express. Mit 2 - 3 Wochen Lieferzeit musst du aber rechnen. Bestelle mindestens 2 Stück. So hast du ein Board zum Experimentieren und ein zweites zum fixen Einbau in deine Wetterstation.

Das Board bekommst du vielleicht auch über Amazon oder Ebay. Achte darauf, dass du nicht die LoRa - Variante bestellst. Auf diesem Board sind viele Pins bereits belegt.

### **Sensor BME280 mit I2C (SCL, SDA) für Luftdruck, Feuchte und Temperatur**

Die sind sehr verbreitet, du solltest sie an vielen Orten bekommen. Wichtig ist einfach, dass es sich um eine I2C - Variante handelt.

Ich werde auch den **Sensor BME680** ausprobieren ( <https://www.aliexpress.com/item/32902672818.html> ), dieser wird aber nicht Voraussetzung für den Bau der Wetterstation sein.

Ich werde auch Experimente mit **anderen Displays** machen. Welche das genau sind, steht noch nicht fest. Dazu werde ich in meiner Bastelkiste graben und hoffen, dass ich etwas Passendes finde. Ob es dann in der Wetterstation verwendet wird, entscheidet sich erst später. Die vollständige Anzeige wird sowieso nur über einen Browser zu sehen sein.

Der Rest (Steckbrett, Anschlussdrähte, USB - Kabel usw.) sollte ja schon in deiner Bastelkiste vorhanden sein.

Andere ESP32 - Board können verwendet werden, wenn du damit zurecht kommst. Eine andere Pin - Belegung wird dann von dir immer Anpassungen an der Verdrahtung und dem Programm verlangen. Sorge auf jeden Fall dafür, dass du vom Hersteller oder Lieferanten die entsprechenden Informationen bekommst.

Beim Sensor BME280 empfehle ich dir dringen, dich daran zu halten. Sonst wirst du auf dich alleine gestellt sein.

## **Spezifikationen**

- ESP32 (240MHz Tensilica LX6 dual-core + 1 ULP, 600 DMIPS)
- 520KB SRAM
- Wi-Fi, dual mode Bluetooth
- 3 x UART, 2 x SPI, 2 x I2C, 1 x I2S
- 29 x general GPIO
- 2 x 12 bit ADC, an 18 Pins
- 2 x 8 bit DAC
- 4 MB (32 MBits) Flash
- 1 x Micro - USB
- USB to UART: CP2102 (Treiber von [silabs.com](http://silabs.com) )
- Battery connector: 3.7V Lithium
- Display: OLED 0.96 inch, 128 x 64

## Zur Verfügung stehende Pins

Diese Pins werden beziehen sich auf das Heltec - Board. Falls du ein anderes Board verwendest, musst du die Liste anpassen.

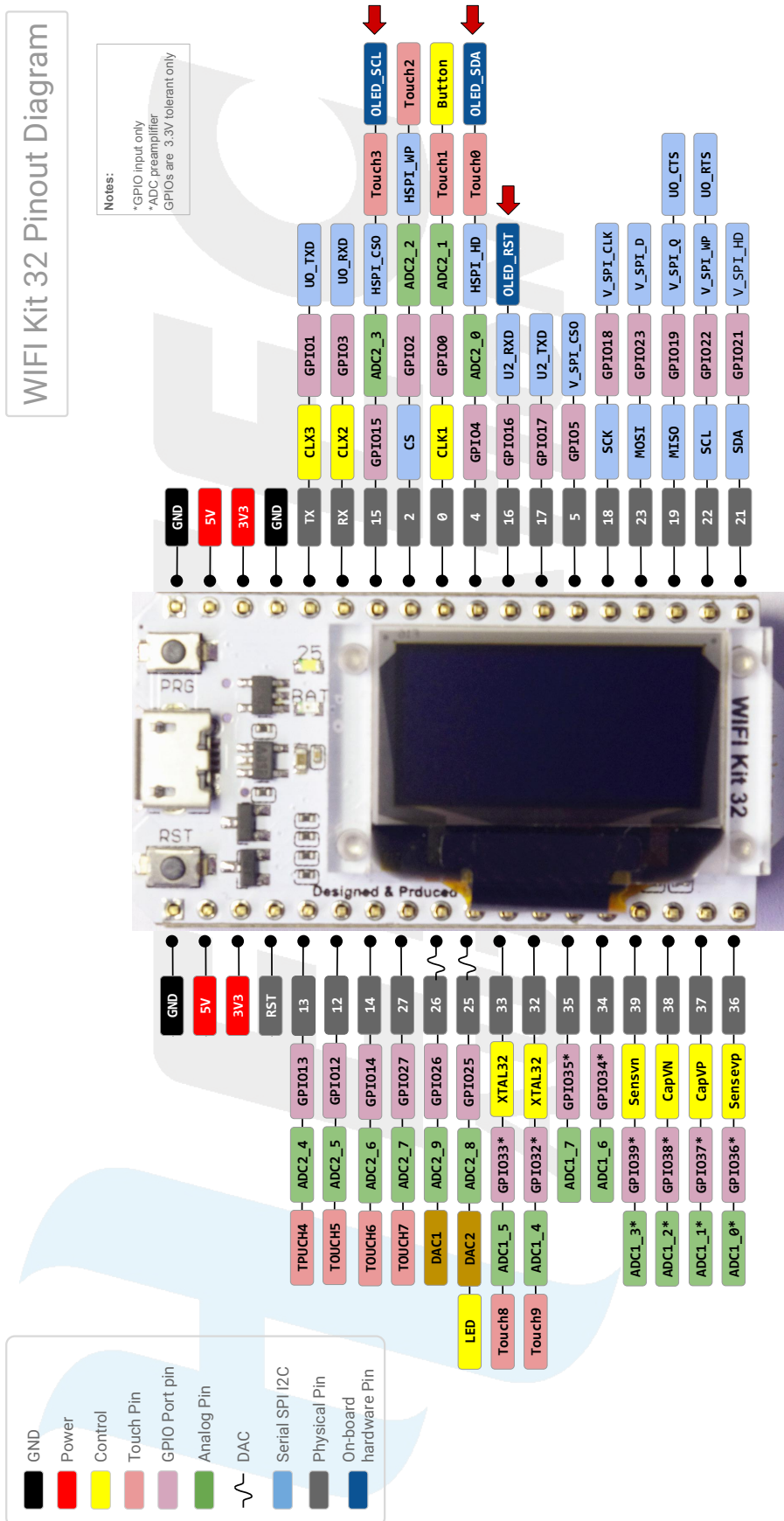
**Achtung: Alle Pins nur 3.3V, sie sind nicht 5V tolerant !!!**

GPIO 0	Eingebauter Button	
GPIO 2	frei	
GPIO 4	OLED	SDA
GPIO 5	frei	
GPIO 12	frei	
GPIO 13	frei	
GPIO 14	frei	
GPIO 15	OLED	SCL
GPIO 16	OLED	muss HIGH gesetzt werden
GPIO 17	frei	
GPIO 18	frei	SCK (SPI)
GPIO 19	frei	MISO (SPI)
GPIO 21	frei	SDA (I2C)
GPIO 22	frei	SCL (I2C)
GPIO 23	frei	MOSI (SPI)
GPIO 25	Eingebaute LED	
GPIO 26	frei	
GPIO 27	frei	
GPI 34	frei	nur Input, kein PULL_UP/PULL_DOWN
GPI 35	frei	nur Input, kein PULL_UP/PULL_DOWN
GPI 36	???	nur Input, kein PULL_UP/PULL_DOWN
GPI 37	???	nur Input, kein PULL_UP/PULL_DOWN
GPI 38	???	nur Input, kein PULL_UP/PULL_DOWN
GPI 39	???	nur Input, kein PULL_UP/PULL_DOWN

**Inzwischen gibt es auch neue Versionen des Heltec - Boards. Hinweise dazu findest du [hier](#).**

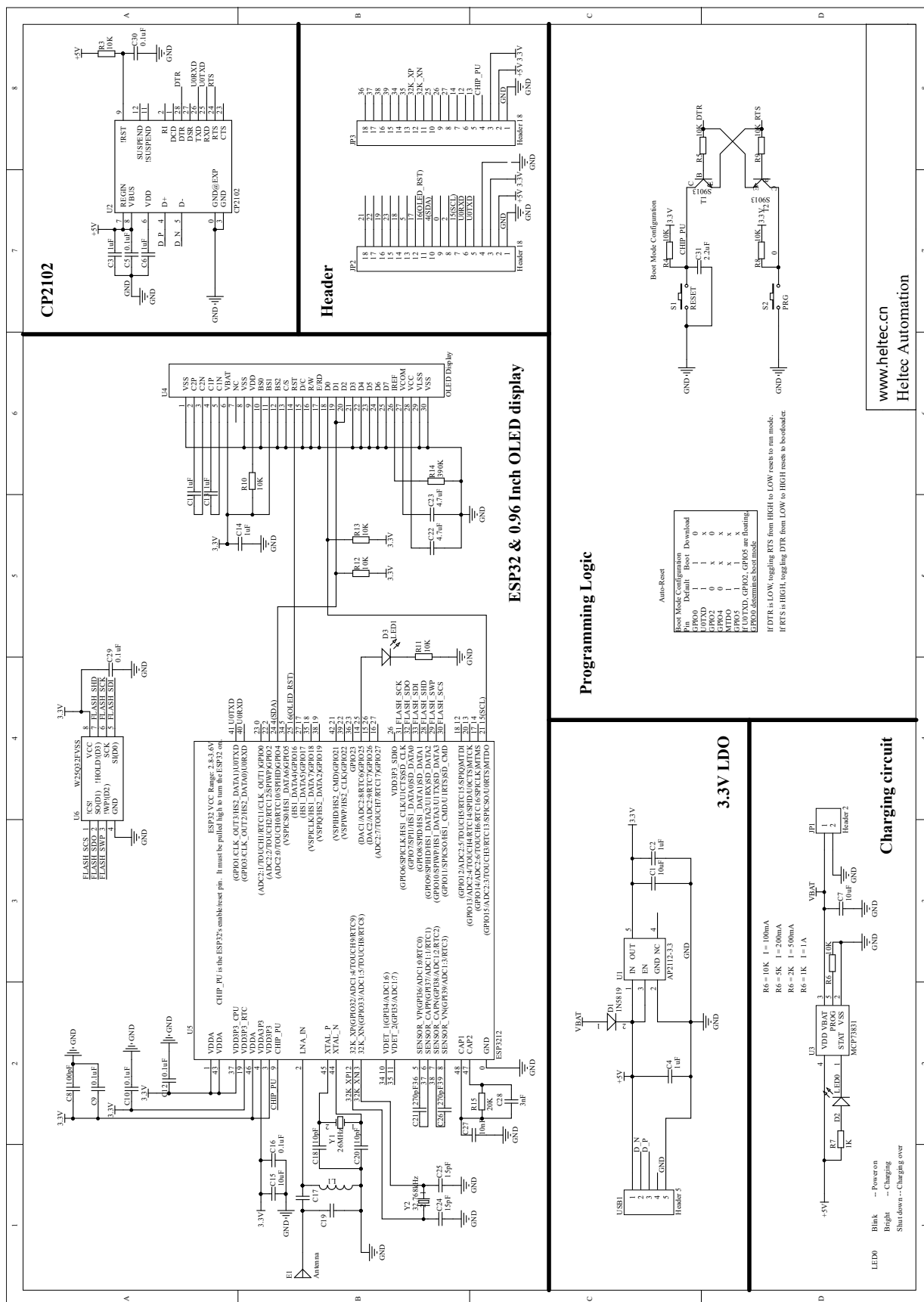
## Pinout

[https://github.com/Heltec-Aaron-Lee/WiFi\\_Kit\\_series/blob/master/PinoutDiagram/WIFI%20Kit%2032.pdf](https://github.com/Heltec-Aaron-Lee/WiFi_Kit_series/blob/master/PinoutDiagram/WIFI%20Kit%2032.pdf)



# Das Schema

[https://github.com/Heltec-Aaron-Lee/WiFi\\_Kit\\_series/blob/master/SchematicDiagram/WIFI\\_Kit\\_32\\_Schematic\\_diagram.PDF](https://github.com/Heltec-Aaron-Lee/WiFi_Kit_series/blob/master/SchematicDiagram/WIFI_Kit_32_Schematic_diagram.PDF)



## Lektion 3: Die Software

### Die Entwicklungsumgebung

Wir werden mit der Thonny - IDE arbeiten. Sie ist sehr einfach aufgebaut und lässt sich leicht installieren. Falls du die entsprechenden Kenntnisse hast, kannst du selbstverständlich auch grosse Entwicklungsumgebungen wie PlatformIO oder PyCharm verwenden. Das ist aber gar nicht so einfach, wie es auf den ersten Blick aussieht.

Ich verwende im Kurs Thonny ( <https://thonny.org/> ). Die Beschreibung hier bezieht sich auf die Version **3.1.2**. Inzwischen gibt es neuere Versionen, die die Installation von thonny-esp überflüssig machen.

**Hinweise zur neuen Version findest du hier.**

### Installation

**1. Treiber für USB-Port herunterladen und installieren:**

<https://www.silabs.com/products/development-tools/software/usb-to-uart-bridge-vcp-drivers>

**2. Download Micropython für ESP32:**

<https://micropython.org/download>

**3. Thonny herunterladen und installieren:**

<https://thonny.org> oder die Links im vorherigen Abschnitt.

**4. ESP32 - Board anschliessen**

**5. Thonny starten**

**6. Tools / Manage plug-ins...**

- Suchen nach esptool, installieren
- Suchen nach thonny-esp, installieren (*nicht mehr notwendig*)

**Die aktuelle Version benötigt ein anderes Vorgehen. Hinweise dazu findest du hier.**

**7. Thonny beenden und neu starten**

**8. Preferences / Interpreter**

- MicroPython on ESP32
- Port: CP2102

**9. Device / Erase ESP8266/ESP32 flash ausführen**

**10. Device / Install Micropython to ESP8266/ESP32 ausführen**

**11. Run / Stop/Restart backend ausführen**

Micropython sollte sich jetzt melden.



```
Shell x Exception x
MicroPython v1.11-146-g154062d9c on 2019-07-12; ESP32 module with ESP32
Type "help()" for more information. [backend=ESP32]
>>>
```

### Erste Schritte

Jetzt wollen wir aber wissen, ob es funktioniert. Also schreiben wir unser erstes Python - Programm und führen es aus. Das Programm muss noch näher betrachtet werden. Das ist dann aber ein Thema für die nächste Lektion.

***blink.py***

```
# Ein einfaches Blink - Programm
import time
from machine import Pin

interne_led = Pin(25, Pin.OUT)

print("blink.py gestartet")
while True:
    interne_led.on()
    print("ein")
    time.sleep_ms(500) # 500 ms
    interne_led.off()
    print('aus') # auch diese Schreibweise ist möglich
    time.sleep(0.5) # 0.5 Sekunden
```

Dieses Programm läuft in einer Endlosschleife und muss abgebrochen werden! Der Abbruch kann mit **Stop/Restart backend** oder mit **ctrl d** durchgeführt werden.

[illegible]

## Lektion 4: Digitale Ein- und Ausgänge, Delay

Ein digitaler Pin muss erstellt werden. Dazu wird eine Variable angelegt, die dann dem Pin entspricht. Es handelt sich dabei nicht nur um eine Pinnummer, sondern um ein komplettes Objekt, welches alle Aspekte seines Pins kennt.

Um Pins zu verwenden, müssen wir die Klasse **Pin** aus dem Modul **machine** importieren.

```
from machine import Pin
```

### Digitale Ausgänge

#### **Initialisierung:**

```
led_pin = Pin(25, Pin.OUTPUT)
```

25 ist hier die GPIO - Nummer. Das entspricht oft nicht der Pinnummer an deinem Board.

#### **Bei der Initialisierung kann auch noch ein Startwert angegeben werden:**

```
led_pin = Pin(2, Pin.OUT, value = 1)
```

#### **Setzen des Pins (er gibt dann 3.3V aus):**

```
led_pin.on() oder led_pin.value(1)
```

#### **Zurücksetzen des Pins (er gibt dann 0V aus):**

```
led_pin.off() oder led_pin.value(0)
```

#### **Abfragen des Pins:**

```
status = led_pin.value()
```

Ist der Pin eingeschaltet, wird 1 zurückgegeben, sonst ist der Wert 0.

#### **Zustand wechseln (blinken):**

```
led_pin.value(not led_pin.value())
```

### Digitale Eingänge

#### **Initialisierung:**

```
button_pin = Pin(2, Pin.IN)
```

#### **Wie beim Arduino kann man auch hier einen internen Pull-Up - Widerstand zuschalten:**

```
button_pin = Pin(2, Pin.IN, Pin.PULL_UP)
```

#### **Der ESP32 stellt auch einen Pull-Down Widerstand zur Verfügung:**

```
button_pin = Pin(2, Pin.IN, Pin.PULL_DOWN)
```

Es scheint aber so, dass diese Option standardmässig aktiviert ist, wenn nichts angegeben wird. Darauf sollte man sich aber nicht verlassen. Falls kein externer Widerstand verbaut wird, immer PULL\_UP oder PULL\_DOWN angeben!

#### **Ohne Widerstand kann ebenfalls gearbeitet werden:**

```
button_pin = Pin(2, Pin.IN, Pin.PULL_HOLD)
```

```
button_pin = Pin(2, Pin.IN, Pin.OPEN_DRAIN)
```



## sleep (delay) und ticks (millis)

Im Blink - Beispiel der vorangegangenen Lektion haben wir einen Delay kennengelernt. Dieser ist im Modul **time** enthalten und muss zuerst importiert werden.

```
import time
```

Wir behandeln hier nur die Delay-Funktionen von **time**. **time** kann noch wesentlich mehr, das soll aber ein Thema für eine spätere Lektion sein.

### **Verzögerung in Sekunden:**

```
time.sleep(2.5)    # 2.5 Sekunden Pause
```

### **Verzögerung in Millisekunden:**

```
time.sleep_ms(2500) # 2500 Millisekunden Pause
```

### **Verzögerung in Mikrosekunden:**

```
time.sleep_us(2500) # 25 Mikrosekunden Pause
```

Diese Funktionen arbeiten wie delay() im Arduino. Sie sind also auch blockierend.

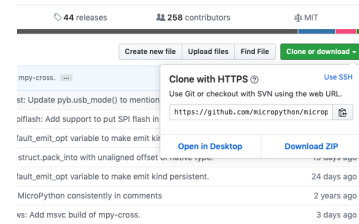
### **Es stehen auch Funktionen zur Verfügung, die wie millis() im Arduino arbeiten:**

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

## Lektion 5: Das Display

Micropython bringt standardmässig bereits eine Vielzahl von Bibliotheken mit. Diese Module können mit **import** dem Programm verfügbar gemacht werden.

Zusätzlich gibt es aber auch viele externe Module, die zusätzlich geladen werden können. Ein ganzes Bündel davon findest du auf Github ( <https://github.com/micropython/micropython> ). Lade einfach das ganze Zip - File herunter und entpacke es auf deinem Computer. Heute interessieren wir uns für den OLED - Treiber. Du findest ihn unter **drivers / display** als Datei **ssd1306.py**.



### Initialisierung

Zuerst muss das Modul **ssd1306.py** auf das Board kopiert werden! Die Angaben gelten für das Heltec - Board. Bei anderer Hardware müssen möglicherweise die Pinnummern für **scl** und **sda** angepasst werden. Dasselbe gilt für **pin16**, bei einigen Boards kann auch auf ihn verzichtet werden.

```
from machine import Pin, I2C
from ssd1306 import SSD1306_I2C

i2c = I2C(scl=Pin(15), sda=Pin(4))
pin16 = Pin(16, Pin.OUT)
pin16.on()
oled = SSD1306_I2C(128, 64, i2c)
```

Damit ist das Display bereit und kann über die Variable **oled**  $\epsilon$   $y=0$

Zuerst wird eine Schnittstelle benötigt:

```
i2c = I2C(scl=Pin(15), sda=Pin(4))
```

Diese Schnittstelle muss dem Konstruktor der Display - Klasse **SSD1306\_I2C** (with, height, i2c, addr=0x3C,

Die beiden hintersten Parameter werden nur übergeben, wenn



### Direkte Befehle

Nur wenige Befehle wirken sich direkt auf das Display aus. Normalerweise wird der Inhalt zuerst in einen internen Speicher geschrieben und erst mit dem Befehl **show()** angezeigt.

#### **oled.poweroff()**

Schaltet die Anzeige aus. Das Modul bleibt aber weiterhin aktiv. Es können Daten zum Display gesendet werden, diese werden aber erst sichtbar, wenn **poweron()** aufgerufen wird.

#### **oled.poweron()**

Die Anzeige wird wieder sichtbar.

### ***oled.contrast(contrast)***

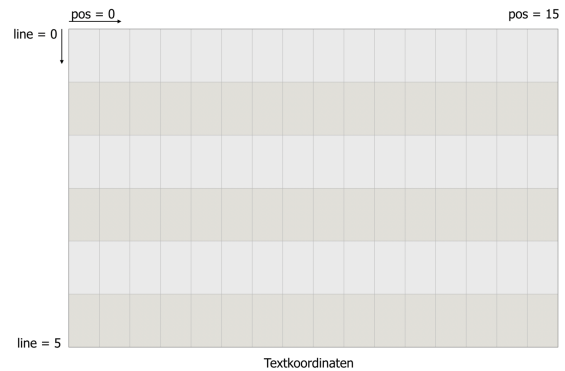
Es kann ein Kontrast zwischen 0 und 255 eingestellt werden mit 255.

### ***oled.invert(invert)***

Der Inhalt kann invertiert dargestellt werden. Der Parameter 0 gilt als **False**, jeder andere Wert wird als **True** interpretiert.

### ***oled.show()***

Der aktuelle Displayinhalt wird angezeigt.



## **Grafikbefehle**

Es handelt sich hier grundsätzlich um ein Grafikdisplay mit den zwei Farben schwarz und hell. Diese werden als 0 und 1 übergeben. Die Auswirkungen der Befehle werden erst mit **show()** sichtbar. Normalerweise werden diese mit einem Parameter namens **c** übergeben.

Die x - Koordinaten liegen zwischen 0 und 127, die y - Koordinaten zwischen 0 und 63.

Es werden hier nicht alle Befehle aufgelistet. Es sind alle Befehle der Klasse FrameBuffer verfügbar. Eine vollständige Referenz ist unter <https://docs.micropython.org/en/latest/library/framebuf.html> zu finden.

### ***oled.fill(c)***

Der ganze Bildschirm wird mit der angegebenen Farbe gefüllt. **fill(0)** kann daher gut zum Löschen des Bildschirms verwendet werden.

### ***oled.pixel(x, y, c)***

Setzen oder Löschen eines einzelnen Pixels. Bei unserem Display löscht die Farbe 0 den Pixel, jeder andere Wert setzt ihn.

### ***oled.pixel(x, y)***

Gibt die Farbe des Pixels zurück (0 oder 1).

### ***oled.hline(x, y, w, c)***

Zeichnet eine horizontale Gerade mit der Länge **w**

### ***oled.vline(x, y, h, c)***

Zeichnet eine vertikale Gerade mit der Höhe **h**.

### ***oled.line(x1, y1, x2, y2, c)***

Zeichnet eine beliebige Gerade von x1/y1 zu x2/y2.

### ***oled.rect(x, y, w, h, c)***

Zeichnet einen rechteckigen Rahmen mit Breite **w** und Höhe **h**.

### ***oled.fill\_rect(x, y, w, h, c)***

Zeichnet ein ausgefülltes Rechteck mit Breite **w** und Höhe **h**.

## **Textbefehle**

Es gibt nur einen einzigen Text - Befehl. Dieser ist in der Lage einen Text an einer beliebigen Stelle auszugeben. Der Text überlagert den bestehenden Inhalt, es wird also der darunter liegende Text nicht gelöscht.

### ***oled.text(s, x, y, c)***

Schreibt den Text in **s** an die Grafikkoordinaten **x** und **y**.

## Ausgabe an Zeichenposition

Dazu gibt es keinen passenden Befehl.

Es wird daher folgende Funktion vorgeschlagen:

```
def text_line(text, line, pos = 0):  
    x = 10 * pos;  
    y = (line) * 11  
    oled.text(text,x,y)
```

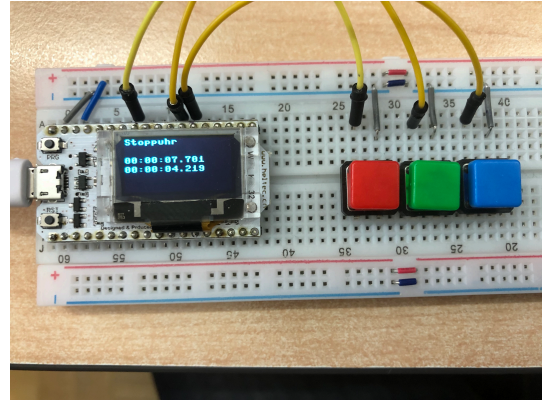
**line** ist die Zeilennummer (0 .. 5) und **pos** die horizontale Zeichenposition (0 .. 15). Wenn **pos** nicht mitgegeben wird, beginnt der Text am Zeilenanfang.

## Lektion 6: Die Stoppuhr

Heute bauen wir eine kleine Stoppuhr.  
Sie wird über drei Tasten gesteuert:

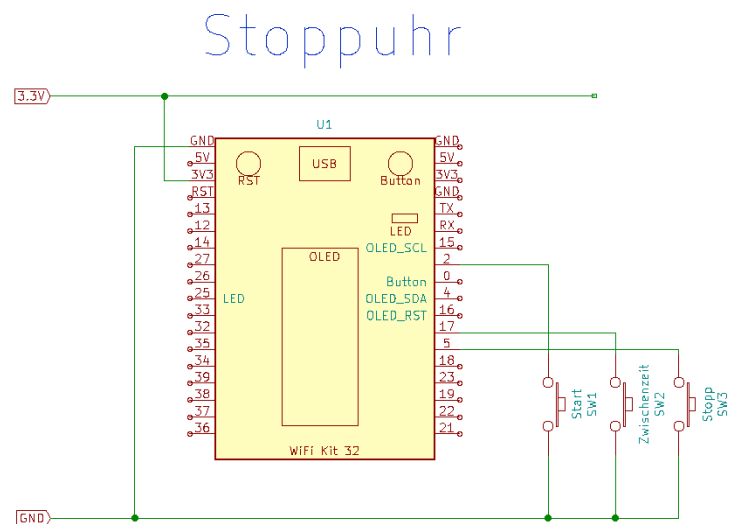
- Start
- Zwischenzeit
- Stopp

Wir müssen eine sinnvolle Verdrahtung erstellen und die Stoppuhr dann programmieren. Dazu benötigen wir das Wissen aus den vorherigen Lektionen, sowie weitere Informationen, die wir in dieser Lektion erarbeiten.



### Die Verdrahtung

Für die 3 Taster benötigen wir drei freie Eingabepins. Das führt dann zu diesem Schema:



### Der Ablauf

Oft ist es sinnvoll, einen ungefähren Ablauf des Programmes festzulegen.

- Zuerst fragen wir die Tasten ab
- Falls die Stoppuhr noch nicht aktiv ist, reagieren wir nur auf die Start-Taste. Wird sie gedrückt, starten wir die Uhr und merken uns den Startzeitpunkt.
- Falls die Stoppuhr aktiv ist, berechnen wir die aktuell vergangene Zeit. Wenn die Zwischenzeittaste gedrückt wird, speichern wir diese. Die Stopp-Taste hält die Uhr an.
- Am Schluss zeigen wir den aktuellen Stand an.

Wir müssen also den aktiv-Zustand und die Zeiten speichern. Dazu legen wir vier globale Variablen an:

```
start_zeit = 0
total_zeit = 0
zwischen_zeit = 0
aktiv = False
```

## Initialisierung der Komponenten

Um es einfach zu halten, betrachten wir momentan nur die zwei Tasten **Start** und **Stopp**.

```
start_taste = Pin(2, Pin.IN, Pin.PULL_UP)
stopp_taste = Pin(5, Pin.IN, Pin.PULL_UP)
```

Wir verwenden bei den Eingängen die internen PULL\_UP - Widerstände.

Den Code für das Display können wir aus einer früheren Lektion übernehmen.

## Eine Funktion zum Abfragen der Tasten

Wir definieren die Funktion **tasten\_abfragen()**:

Eine Funktion wird immer mit dem Schlüsselwort **def** eingeleitet. Es gibt keinen Unterschied zwischen Funktionen, die etwas zurückgeben und solchen, die nur einige Befehle ausführen.

Der Funktionskopf lautet also

```
def tasten_abfragen():
```

- **def** leitet die Funktionsdefinition ein
- **tasten\_abfragen** ist der Name der Funktion
- Zwischen den beiden Klammern **()** könnte man Werte an die Funktion übergeben. In unserem Fall haben wir keine Parameter, deshalb bleiben nur die beiden Klammern übrig.
- Der Doppelpunkt **:** signalisiert, dass jetzt der Code der Funktion beginnt.

Danach folgen die Befehle, die von der Funktion ausgeführt werden sollen.

Alle Befehle, die zur Funktion gehören, sind um 4 Zeichen eingerückt.

```
    start_gedrueckt = start_taste.value()
    stopp_gedrueckt = stopp_taste.value()
    return start_gedrueckt
```

Mit **return** kann ein Wert zurückgegeben werden. Was machen wir aber, wenn wir beide Werte zurückgeben sollen? Das ist in Python ganz einfach! Wir geben einfach beide Werte zurück.

```
    return start_gedrueckt, stopp_gedrueckt
```

Wenn wir das ausprobieren, werden wir feststellen, dass es nicht wie gewünscht funktioniert. Die Taste wird als gedrückt zurückgegeben, wenn wir sie nicht drücken und umgekehrt. Das liegt an der Verdrahtung und am PULL\_UP. Wir müssen also den Zustand umkehren. Das erfolgt mit dem Schlüsselwort **not**.

```
    return not start_gedrueckt, not stopp_gedrueckt
```

Es ist nicht notwendig, die Zustände in eigenen Variablen zu speichern. So funktioniert auch die verkürzte Version:

```
def tasten_abfragen():
    return not start_taste.value(), not stopp_taste.value()
```

Unten im Hauptprogramm nehmen wir beide Rückgabewerte gleichzeitig in Empfang und speichern sie in den Variablen **start** und **stopp**:

```
start, stopp = tasten_abfragen()
```

## Was machen wir, wenn Start gedrückt wird?

Auf die Start - Taste reagieren wir nur, wenn die Uhr nicht aktiv ist. Das müssen wir testen:

if not aktiv:

Auch hier haben wir wieder diesen Doppelpunkt und die anschliessende Einrückung. Damit definieren wir, welcher Code ausgeführt werden soll, wenn die Bedingung erfüllt ist. Darunter testen wir noch auf die gedrückte Starttaste. Nur dann wird die Uhr gestartet.

```
if not aktiv:
    if start: # Start ist gedrückt
        start_zeit = time.ticks_ms()
        aktiv = True
```

## Was machen wir, wenn die Uhr aktiv ist und wie reagieren wir auf Stopp?

```
if aktiv:
    total_zeit = time.ticks_ms() - start_zeit
    if stopp: # Uhr anhalten
        aktiv = False
```

Wir testen zuerst auf **aktiv**. Nur dann speichern wir die aktuell vergangene Zeit in **total\_zeit**. Diese wird berechnet aus der aktuellen Zeit minus der Startzeit.

Dann wird noch auf die Stopptaste geprüft. Wenn sie gedrückt ist, halten wir die Uhr an.

## Anzeige auf dem Display und die format() - Anweisung

Wir rufen einfach **anzeigen()** auf.

Leider haben wir da ein kleines Problem. **anzeigen()** gibt es noch gar nicht. Wir müssen diese Funktion also schreiben und dies ist eine anspruchsvolle Arbeit.

Der Anfang ist ganz einfach. Wir löschen den Bildschirm, schreiben die Texte und zeigen alles am Schluss mit **.show()** an.

```
def anzeigen():
    oled.fill(0) # Bildschirm löschen
    text_line("Stoppuhr",0)
    oled.show() # Alles anzeigen
```

Das Wichtigste fehlt hier aber. Wir wollen ja die vergangene Zeit anzeigen. Diese steht in der Variablen **total\_zeit**. Es wäre nicht sehr hilfreich, wenn wir einfach diese Zeit in Millisekunden anzeigen würden. Wir müssen sie also in Stunden, Minuten, Sekunden und Tausendstel aufteilen.

Dazu brauchen wir die numerischen Operatoren (sie wurden bereits im Kurs [Micropython Grundlagen](#) besprochen).

```
millis = zeit % 1000
sekunden = (zeit // 1000) % 60
minuten = (zeit // 1000 // 60) % 60
stunden = (zeit // 1000 // 60 // 60) % 24
```

% ist der Modulo-Operator, er gibt uns den Rest einer ganzzahligen Division zurück

// ist der Operator für eine ganzzahlige Division

Jetzt müssen wir diese Werte nur noch in einen Text verwandeln.

Da wir später diese Umwandlung von **total\_zeit** zu unserem Ausgabertext sowohl für die Gesamtzeit als auch für die Zwischenzeit brauchen, lagern wir sie in eine Funktion aus.

Diese Funktion soll uns einen String (ein Text in Form einer Zeichenkette) zurückgeben. HH:MM:SS.TTT soll das Format sein.

Dazu gibt es die Format-Funktion. Wenn ich schreibe

```
"{:}:{:}:{:}.".format(stunden, minuten, sekunden, millis)
```

werden die Platzhalter {} mit den entsprechenden Werten gefüllt. Jetzt können aber diese Werte verschiedene Anzahl Stellen aufweisen. Das ist unschön, lässt sich aber mit zusätzlichen Format - Anweisungen verbessern.

```
def zeit_text(zeit):  
    millis = zeit % 1000  
    sekunden = (zeit // 1000) % 60  
    minuten = (zeit // 1000 // 60) % 60  
    stunden = (zeit // 1000 // 60 // 60) % 24  
    return "{:02d}:{:02d}:{:02d}.{:03d}".format(stunden,minuten,sekunden,millis)
```

Unsere Funktion **anzeigen()** sieht dann so aus:

```
def anzeigen():  
    oled.fill(0) # Bildschirm löschen  
    text_line("Stoppuhr",0)  
    total_text = zeit_text(total_zeit)  
    text_line(total_text,2)  
    oled.show()
```

## Deine Übungsaufgabe

Jetzt bist du dran. Das Projekt erfüllt ja noch nicht alle Kriterien, da wir noch keine Zwischenzeit haben. Das zu programmieren, ist deine Aufgabe. Ich wünsche dir viel Spass.

Eine mögliche Lösung werde ich in der nächsten Lektion zeigen.

Unsere Stoppuhr hat noch diverse kleine Schönheitsfehler, der die genaue Zeitmessung erschwert. Hast du eine Idee, welche das sind? Auch das werden wir in der nächsten Lektion behandeln.

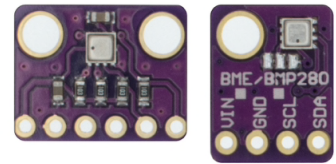


## Lektion 7: Eine mögliche Lösung

Bitte schau dir das Video an und experimentiere mit den in den Begleitunterlagen vorhandenen Dateien.

### Die Stoppuhr soll automatisch ausgeführt werden. Wie geht das?

Wenn das Board neu gestartet wird, wird automatisch zuerst `boot.py` und dann `main.py` ausgeführt. `boot.py` ist zwar auf dem Board vorhanden, enthält aber nur auskommentierte Zeilen. `main.py` musst du selber auf das Board kopieren.



Du kannst dein Stoppuhrprogramm mit dem Menüpunkt **Device / Upload current script as main script** auf das Board kopieren. Das gilt aber nur für Thonny bis zur Version 3.1.2. Ab der Version 3.2.1 gibt es diesen Punkt nicht mehr. Du kannst aber unter **File / save as..** das Programm als `main.py` auf das Board speichern.

## Lektion 8: Druck, Temperatur und Luftfeuchtigkeit (BME280)

### Überblick

Beim BME280 handelt es sich um einen Sensor von Bosch, der die gewünschten Werte liefern kann. Es gibt zwei Versionen von Breakout-Boards. Die eine ist für 5V ausgelegt, die andere für 3.3V.

Die 5V - Variante darf nur mit 3.3V betrieben werden, da sonst der ESP32 Schaden nehmen könnte.

Hier die wichtigsten Daten:

Schnittstellen: I2C und SPI  
Betriebsspannung Sensor: 1.71 - 3.6V

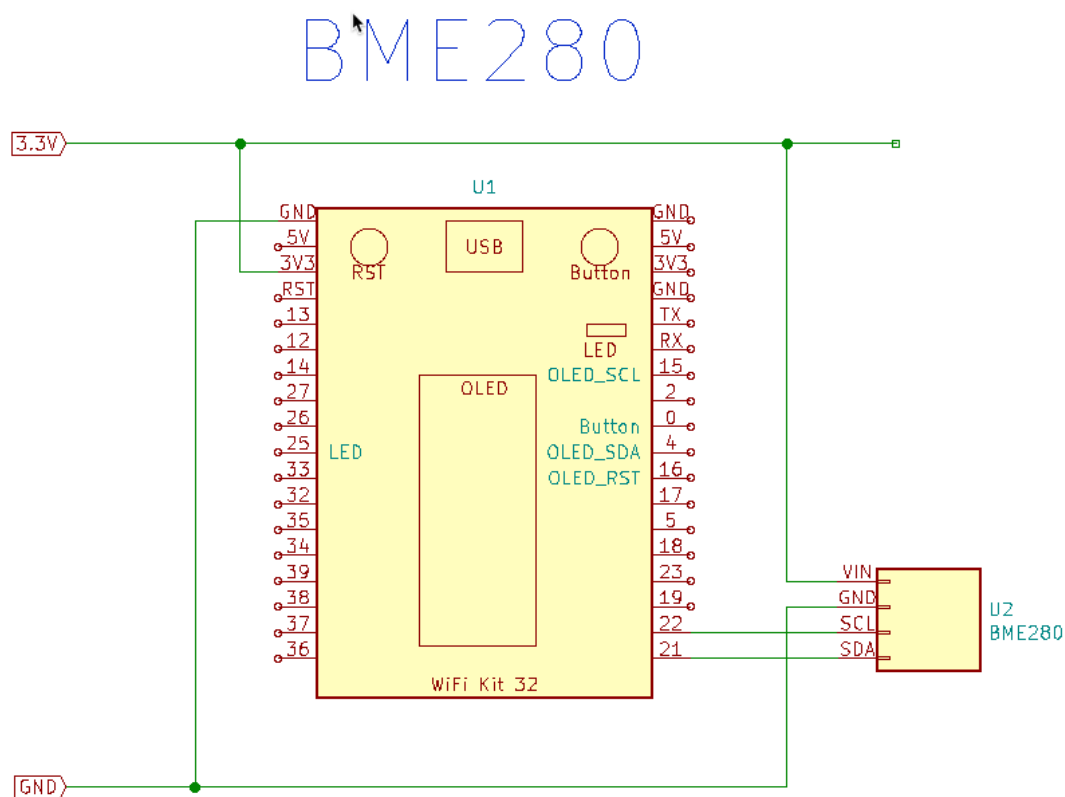
Genauigkeit:

Feuchte: +/- 3%  
Druck: +/- 1.5 hPa  
Temperatur: +/- 1 °C

**Webseite:** [https://www.bosch-sensortec.com/bst/products/all\\_products/bme280](https://www.bosch-sensortec.com/bst/products/all_products/bme280)

**Datenblatt:** <https://ae-bst.resource.bosch.com/media/tech/media/datasheets/BST-BME280-DS002.pdf>

### Der Versuchsaufbau



### Die Software

Bosch stellt eine C - Bibliothek zur Verfügung, von der eine gute Umsetzung in Micropython existiert.

**Von Bosch (C):** [https://github.com/BoschSensortec/BME280\\_driver](https://github.com/BoschSensortec/BME280_driver)

**Micropython:** [https://github.com/triplepoint/micropython\\_bme280\\_i2c](https://github.com/triplepoint/micropython_bme280_i2c)

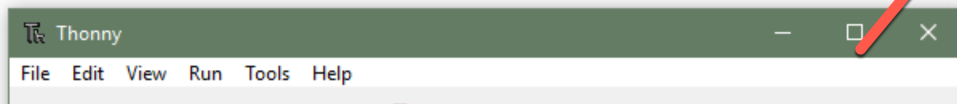
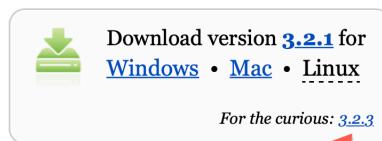
Die Datei **bme280\_i2c.py** enthält die Python-Bibliothek und muss auf das Board kopiert werden.

Die Quelltexte der ersten Messungen sind in den Begleitunterlagen zu finden.

Die Werte von Temperatur und Luftfeuchtigkeit entsprechen etwa den erwarteten Werten. Der Luftdruck erscheint im Moment als sehr ungenau. Das liegt aber daran, dass der Sensor den absoluten Luftdruck misst. Angegeben wird aber normalerweise ein auf Meereshöhe normierter Wert. Diese liegt höher als der gemessene Wert. Wir werden uns in der nächsten Lektion genauer damit befassen.

# Thonny

Python IDE for beginners



ReleasesTags

Latest release

v3.2.3  
f3ff7c5  
Verified

## Version 3.2.3

aivarannamaa released this 7 days ago

3.2.3 together with short lived 3.2.2 are bug-fix releases.

Changes in 3.2.2 and 3.2.3:

- NEW: ESP plug-in has been merged into main Thonny package

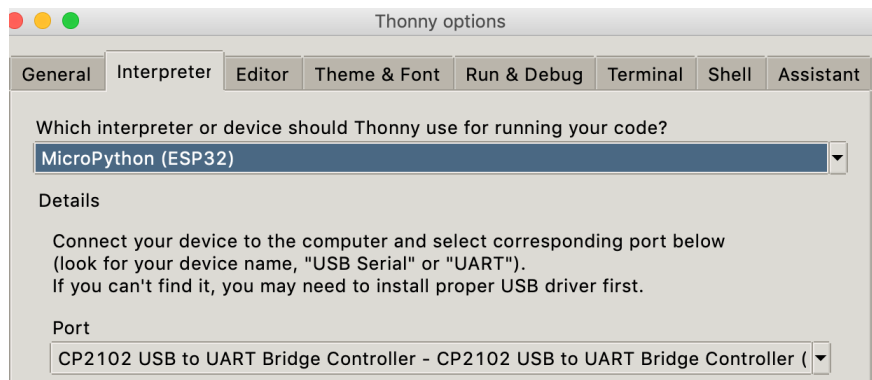
<a href="#">thonny-3.2.3.dmg</a>	18.7 MB
<a href="#">thonny-3.2.3.exe</a>	14.1 MB

## Lektion 9: Die Thonny-Version 3.2.3

### Installation von Thonny

Auf der Webseite von Thonny ( <https://thonny.org> ) findet man den Download - Link. Mit der Version 3.2.3 wurden einige Neuerungen eingeführt, die wir hier besprechen möchten.

Falls bereits eine ältere Version vorhanden war, bleiben bei der Installation alle Plugins und die Einstellungen erhalten. Das Plugin esptool liegt inzwischen In Version 2.8 vor und sollte allenfalls aktualisiert werden. Das Plugin thonny-esp wird nicht mehr benötigt und kann deinstalliert werden.



In den Einstellungen sollte überprüft werden, ob der richtige Interpreter und der korrekte Treiber ausgewählt ist.

## Aktualisieren der Micropython - Version auf dem Board

Bei dieser Gelegenheit kann auch die Micropython - Version auf dem Board aktualisiert werden.

Es gibt drei Möglichkeiten:

1. Die aktuellste ESP-IDF v3.x

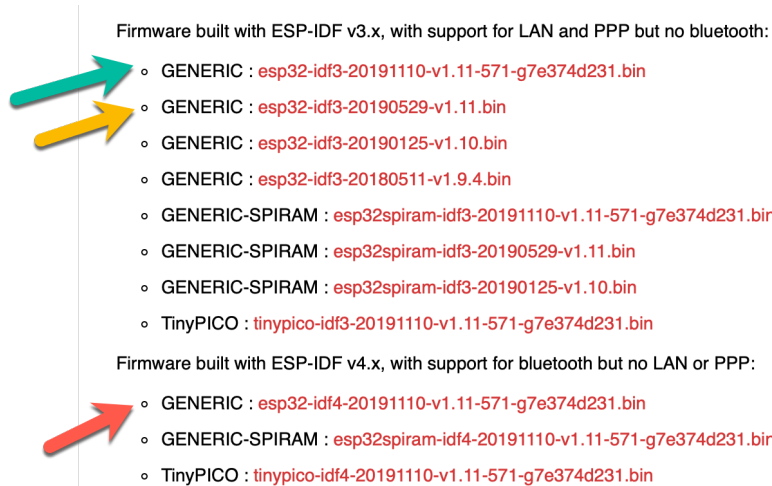
Diese Version unterstützt noch kein Bluetooth, enthält aber alle Teile, die wir benötigen. Es handelt sich um einen aktuellen Build, der beinahe täglich erneuert wird.

2. Die stabile ESP-IDF v3.x

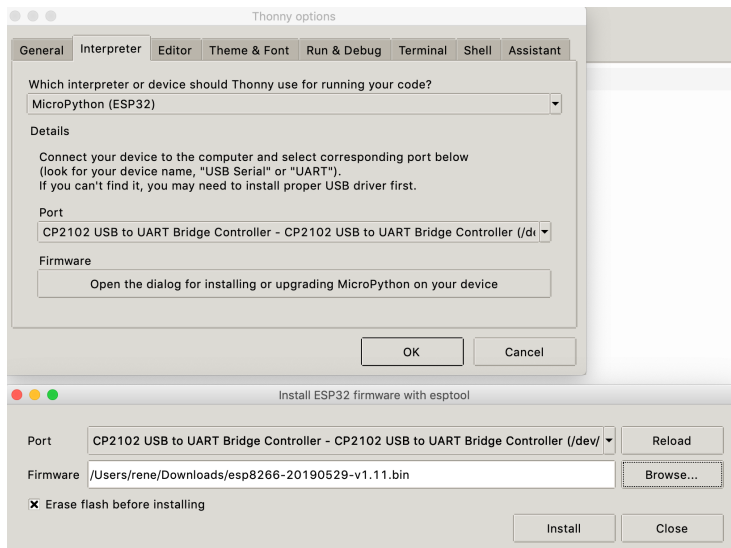
Diese Version ist ebenso geeignet. Sie stammt vom 29.05.2019 und gilt als stabile Version 1.11

3. Die aktuellste ESP-IDF v4.x

Ich habe diese Version installiert, da sie Experimente mit Bluetooth erlaubt. Das wird allerdings im Kurs momentan nicht benutzt.



Die Installation erfolgt über Tools / Options / Interpreter



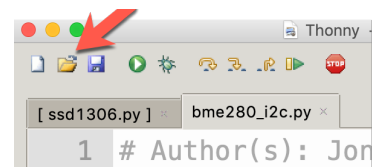
Die Option **Erase flash before installing** sollte aktiviert sein.

## Neue Arbeitsweise mit Thonny 3.2.3

Das Laden, Speichern und Kopieren von Dateien wird über Funktionen des Menüs File durchgeführt. Dabei hat man immer die Wahl zwischen der lokalen Festplatte und dem ESP32 - Board. Dabei kann es vorkommen, dass der Text **This Computer** nicht sichtbar ist. Die Auswahl funktioniert aber trotzdem. **Alternativ können die meisten Aufgaben auch über den File - Browser vorgenommen werden.**

### Laden einer Datei

Die Funktion wird über **File / Open** oder das Laden - Symbol aufgerufen. Der Inhalt der geladenen Datei erscheint dann im Edit - Fenster. Der Dateiname erscheint im Titel des Tabs. Wenn er von eckigen Klammern umschlossen ist, wurde die Datei vom Board geladen, sonst vom PC.

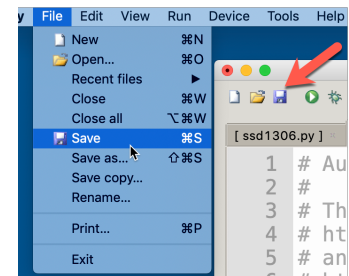


### Speichern einer Datei

Die Funktion wird über **File / Save** oder das Speichern - Symbol aufgerufen. Dabei wird der Inhalt des aktuellen Edit-Fensters ohne Nachfrage in die ursprüngliche Datei kopiert.

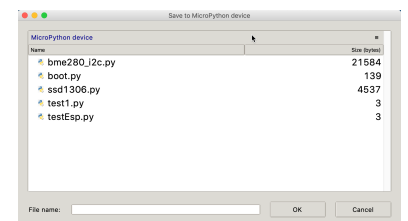
### Speichern einer neuen Datei

Neue Dateien, die noch nie gespeichert wurden, sollten mit **File / Save as...** gespeichert werden.



### Kopieren einer Datei von der lokalen Festplatte auf das Board

Das File wird mit **File / Load** von der lokalen Festplatte geladen und dann mit **Save copy...** auf das Board kopiert. Leider bietet das Programm den ursprünglichen Dateinamen nicht als Vorschlag an. Er muss deshalb eingegeben werden.



### Umbenennen einer Datei

Das Umbenennen erfolgt mit **Rename...**

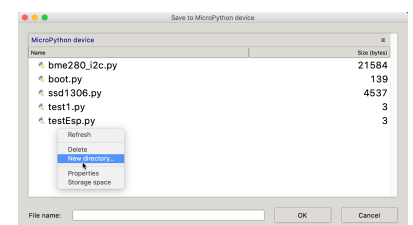
Der Befehl bezieht sich immer auf die aktuelle Datei im Editor - Fenster. Sie muss also dort geöffnet sein.

### Verzeichnis anlegen

Dazu gibt es keinen direkten Befehl. Es gibt aber die Möglichkeit in der Dateiauswahl von Thonny mit der **rechten Maustaste** einen **New directory...** - Befehl aufzurufen.

### Löschen einer Datei oder eines Verzeichnisses

Mit der **rechten Maustaste** erhält man auch die Möglichkeit den **Delete** - Befehl auszuführen. Er kann Dateien und Verzeichnisse löschen.



## Der File - Browser

Insbesondere das Kopieren von Dateien zwischen Festplatte und Board lässt sich bequem mit dem File-Browser erledigen. Er ist erreichbar über View / Files.

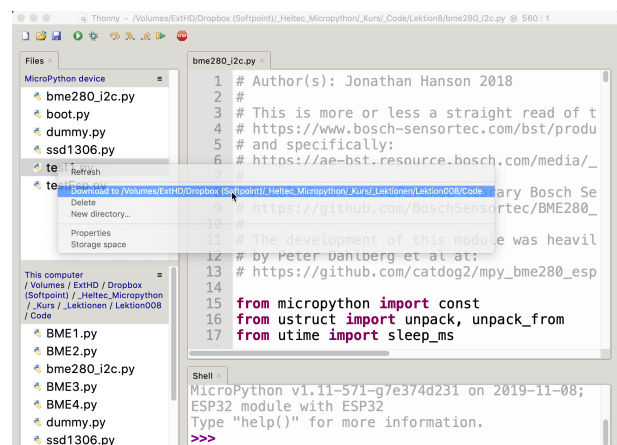
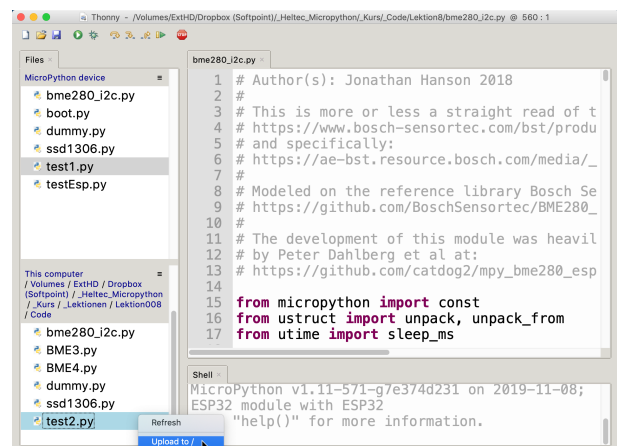
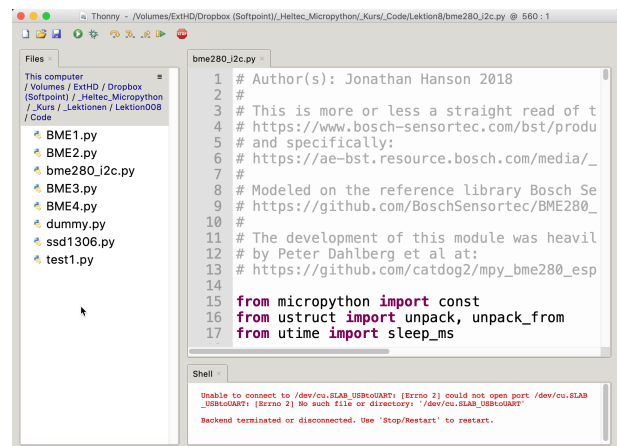
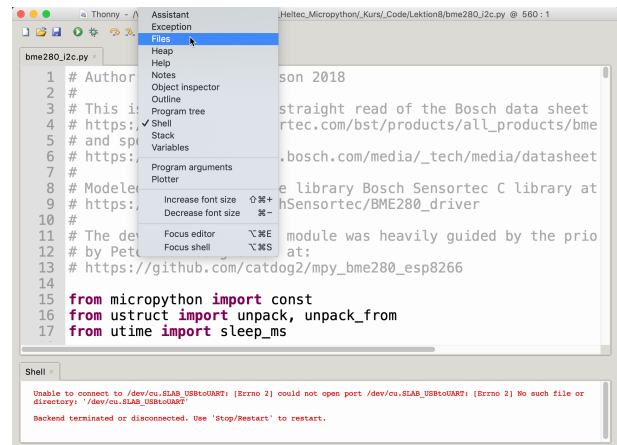
Falls du dann nur deine lokale Festplatte siehst, ist dein Board nicht verbunden.

Sobald das Device verbunden ist, erscheint das zweite Fenster mit den Dateien auf dem Board.

Mit der **rechten Maustaste** kannst du die Funktion **Upload to** / auslösen. Diese kopiert die Datei unter demselben Namen auf das Board.

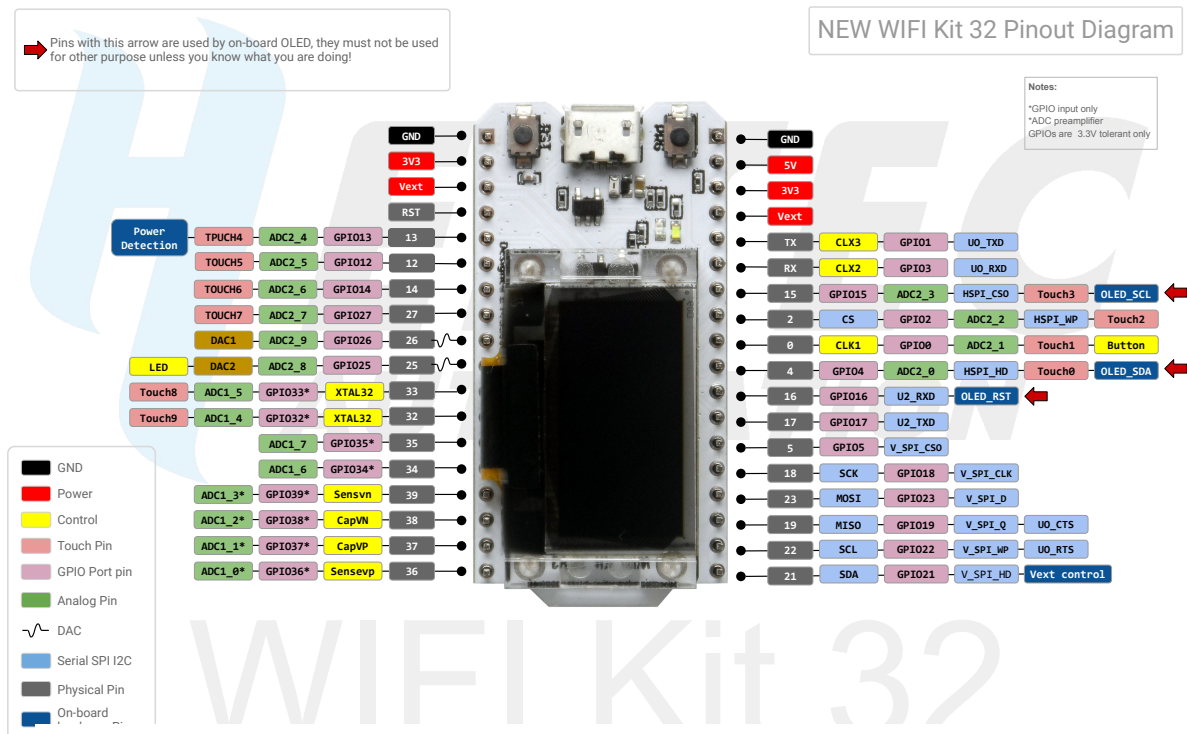
Der umgekehrte Weg ist ebenfalls möglich. Die Funktion **Download to** erlaubt eine Datei vom Board auf deine Festplatte zu kopieren.

Zusätzlich stehen im Kontextmenu auch die Funktionen **Delete** und **New directory...** zur Verfügung.

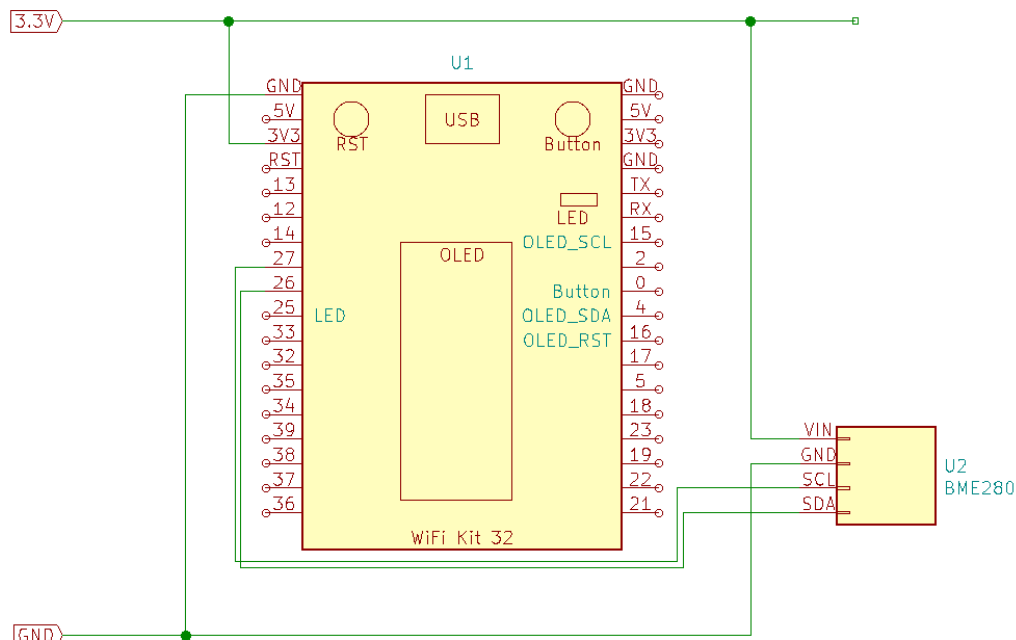


## Lektion 10: Neue Heltec-Version

Es gibt eine neue Version des Heltec - Boards. Damit unser Programm auf beiden Boards läuft, passen wir unsere Schaltung an.



BME280





Das führt zu zwei Programmänderungen:

**statt**

```
i2c_bme = I2C(scl=Pin(22), sda=Pin(21), freq=10000)
...
i2c_oled = I2C(scl=Pin(15), sda=Pin(4), freq=10000)
```

**schreiben wir**

```
i2c_bme = I2C(scl=Pin(27), sda=Pin(26), freq=10000)
...
i2c_oled = I2C(scl=Pin(15, Pin.OUT, Pin.PULL_UP), sda=Pin(4, Pin.OUT,
Pin.PULL_UP))
```

## Lektion 11: Nochmals BME280

Wir verwenden also den BME280 auf den Ports 26 und 27. So funktioniert die Schaltung sowohl mit der neuen Boardversion, wie auch der alten.

Wir haben auch festgestellt, dass beide BME - Versionen (5V und 3.3V) einsetzbar sind. Beide müssen mit 3.3V betrieben werden.

In einer früheren Lektion haben wir festgestellt, dass unsere Luftdruckwerte überhaupt nicht mit denen im Wetterbericht übereinstimmen. Das liegt daran, dass wir den tatsächlichen Luftdruck messen, der von Wetterstationen angegebene Luftdruck aber auf Meereshöhe umgerechnet wurde. Der Luftdruck hängt von der Höhe, der Temperatur und in geringen Masse auch von der Luftfeuchtigkeit ab. Um vergleichbare Werte zu erhalten, wird er normiert. Dies kann über die barometrische Höhenformel erfolgen. Der Blick auf die Formel in Wikipedia zeigt uns aber sofort, dass das viel zu kompliziert ist.

So verwenden wir eine viel einfachere Lösung. Wir arbeiten mit einem leicht zu ermittelnden Korrekturfaktor.

Das ist ganz einfach:

- Wir suchen im Internet nach einer offiziellen Wetterstation in der Nähe
- Das Wetter sollte ruhig sein. So können wir annehmen, dass der Luftdruck in unserem Umkreis mehr oder weniger konstant ist.
- Den Druck bei der offiziellen Station dividieren wird durch unseren gemessenen Druck. Das gibt dann den gewünschten Faktor.

Dieser Faktor sollte mehr oder weniger konstant bleiben, da er hauptsächlich von unserem Standort abhängt. Er ist aber auch temperaturabhängig, so dass es vermutlich einen Unterschied zwischen Sommer und Winter gibt. Ob dadurch weitere Korrekturen notwendig sind, wird sich noch zeigen.

## Lektion 12: WLAN - Verbindung

Ein grosser Vorteil des ESP32 ist, dass er sich einfach in das heimische WLAN einbinden lässt.

### Verbindung aufbauen

Alle notwendigen Funktionen werden vom Modul **network** bereitgestellt. Also muss dieses importiert werden.

```
from machine import Pin, I2C
import time
from ssd1306 import SSD1306_I2C # Für Display
import network
```

Wir brauchen die Zugangsdaten für unser WLAN und müssen das Interface-Objekt für den Zugriff erstellen.

```
wlan_ssid      = "ssid"
wlan_passwort  = "geheim"

wlan = network.WLAN(network.STA_IF)
```

Jetzt kann das Interface aktiviert und die Verbindung aufgebaut werden.

```
wlan.active(True);
wlan.connect(wlan_ssid,wlan_passwort)
```

Das benötigt etwas Zeit und darum müssen wir laufend abfragen, ob die Verbindung bereit ist. Erst dann können wir die Netzwerkkonfiguration abfragen.

```
while not wlan.isconnected():
    pass
print("WLAN: ", wlan.ifconfig())
```

Das funktioniert so weit, hat aber noch einige Schwächen. Wenn die Verbindung einmal steht, bleibt sie bestehen, auch wenn wir einen neuen Connect mit anderen Zugangsdaten versuchen. Wir müssen das Interface deaktivieren und nachher erneut starten. Das braucht Zeit und darum ist noch ein kleiner Delay notwendig.

```
wlan.active(False);
wlan.active(True);
time.sleep(0.5)
wlan.connect(wlan_ssid,wlan_passwort)
while not wlan.isconnected():
    pass
```

Wenn die Verbindung aus irgend einem Grund nicht aufgebaut werden kann, dann sollte das Programm nicht einfach hängen bleiben. Darum muss eine Timeout - Funktion eingebaut werden. Das ist eine gute Gelegenheit, den Verbindungsaufbau in eine Funktion zu packen.

```
def wlan_connect():
    start = time.ticks_ms();
    wlan.active(False)
    wlan.active(True);
    time.sleep(0.5)
    wlan.connect(wlan_ssid,wlan_passwort)
    while not wlan.isconnected() and start + 5000 > time.ticks_ms():
        pass
```

## Die Zeit abfragen

Jetzt soll die aktuelle Zeit ausgegeben werden. Der ESP32 hat ja eine Real Time Clock (RTC) eingebaut, die mit Datum und Zeit umgehen kann. Diese verliert ihre Zeit aber immer, wenn die Stromversorgung wegfällt. Also müssen wir beim Start die Zeit aus dem Internet beziehen und damit die Uhr initialisieren.

Für die Abfrage der Zeit aus dem Internet steht uns das Modul **ntptime** zur Verfügung. Die interne Uhr des ESP32 lässt sich über das normale **time** - Modul bedienen.

Mit **ntptime.settime()** wird die interne Uhr gestellt und mit **time.localtime()** wieder abgefragt.

```
def setze_zeit():
    ntptime.settime()

def zeige_zeit():
    print(time.localtime())

# time.localtime: YYYY, MM, DD, HH, MM, SS, day of week (0=Montag), day of
year)
```

Die Zeit wird als Tupel zurückgegeben.

```
(2019, 12, 16, 9, 52, 22, 0, 350)
(YYYY, MM, DD, HH, MM, SS, day of week (0=Montag), day of year)
```

Darauf kann wie bei einem Array zugegriffen werden.

`localtime[0]` gibt das Jahr zurück und `localtime[2]` den Tag.

Obwohl die Funktion `localtime()` heisst, gibt sie nicht die lokale Zeit zurück. Wir müssen also die Anpassung für unsere Zeitzone selbst machen. Dazu gibt es zwei Funktionen:

```
sek = time.mktime(time.localtime()) # Anzahl Sekunden seit dem 1.1.2000
zeit = time.localtime(sek + 3600) # Korrigiert die Zeit um 1 Stunde
```

In **zeit** haben wir dann wieder das obige Tupel, aber um eine Stunde korrigiert.

```
(2019, 12, 16, 10, 52, 22, 0, 350)
```

Als Übungsaufgabe kann das jetzt sauber formatiert auf das Oled - Display ausgegeben werden.

## Lektion 13: Modularisierung und eine Lösung der Übungsaufgabe

### Die Struktur unseres Programmes

Mit der Grösse eines Programms wächst auch seine Unübersichtlichkeit. Ein Mittel dagegen ist die Aufteilung in Module. Das bringt einige Vorteile:

- Die einzelnen Module sind überschaubar und können einzeln getestet und dokumentiert werden.
- Diese Module können in anderen Programmen wiederverwendet werden.
- Sie können bei Bedarf einem Programm hinzugefügt werden, sind aber auch sehr einfach wieder zu entfernen.

Das Hauptprogramm und die Module kommunizieren nur über exakt definierte Schnittstellen. Diese sollten möglichst nicht mehr verändert werden. Falls notwendig werden sie erweitert, sollten aber kompatibel zu den Vorgängerversionen bleiben.

***Wir werden das nicht so strikt umsetzen, da wir im Moment die Dinge etwas einfach halten möchten. In diesem Zusammenhang wäre der Einsatz von Klassen vernünftig. Damit werden wir uns aber erst später befassen.***

Wir werden diese Module als einzelne Python-Dateien (.py) realisieren. ***Diese müssen auf das Board geladen werden.***

Als Erstes überlegen wir uns, welche Funktionen wir in unserem Programm schon haben. Was kann in ein Modul ausgelagert werden?

#### **Das Hauptprogramm**

Hier wird die eigentliche Funktionalität bestimmt. Auch das könnte später aufgeteilt werden, momentan ist es aber noch sehr einfach und überschaubar.

#### **Vertrauliche Informationen (geheim.py)**

Alle Daten, die vertraulich behandelt werden müssen, sollten in eine eigene Datei ausgelagert werden.

#### **Verbindung mit dem WLAN (netzwerk.py)**

Hier wird die Verbindung mit dem WLAN zur Verfügung gestellt.

#### **Ausgabe auf das OLED - Display (display.py)**

Alles was notwendig ist, um Ausgaben auf das OLED-Display zu machen.

#### **Lesen der BME-Sensordaten (bme\_sensor.py)**

Die Sensordaten werden hier gelesen und zu Verfügung gestellt. In dieser Lektion wird dieses Modul noch nicht erstellt, da wir es für die Zeitanzeige nicht benötigen.

## geheim.py

Hier werden alle vertraulichen Informationen abgelegt. Momentan sind das nur die Zugangsdaten zum WLAN.

```
# WLAN
wlan_ssid      = "ssid"
wlan_passwort  = "geheim"
```

## netzwerk.py

Wir stellen hier das Objekt **wlan** und die Funktion **wlan\_connect()** zur Verfügung. Dieses Modul benötigt das Modul **geheim**.

```
import network
import time
from geheim import *

# WLAN
wlan = network.WLAN(network.STA_IF)
def wlan_connect():
    ...
    wlan.connect(wlan_ssid,wlan_passwort)
    ...
```

geheim.py wird in der Form **from geheim import \*** importiert. Das hat den Vorteil, dass wir **wlan\_ssid** und **wlan\_passwort** direkt verwenden können. Bei einem normalen Import müssten wir jeweils den Modulnamen voranstellen (**geheim.wlan\_ssid**).

## display.py

Dieses Modul stellt gegen Aussen das Objekt **oled** und die Funktion **text\_line()** zur Verfügung. Es wären noch weitere Elemente sichtbar, diese sollten aber nicht ausserhalb des Moduls verwendet werden.

```
...
oled = SSD1306_I2C(128,64,i2c_oled)

# Hilfsfunktion für Display
def text_line(text, line, pos = 0):
    x = 10 * pos;
    y = (line-1) * 11
    oled.fill_rect(x,y,128-10*pos,11,0)
    oled.text(text,x,y)
```

Die Funktion **text\_line()** wurde noch etwas erweitert. Bisher blieb der vorherige Inhalt der Zeile immer stehen, so dass es oft notwendig war, das Display vollständig zu löschen.

Die Zeile **oled.fill\_rect(x,y,128-10\*pos,11,0)** erstellt ein Rechteck um die ganze Zeile und löscht den Inhalt. Danach kann die Zeile neu geschrieben werden.

## Das Hauptprogramm (Zeitanzeige)

Mit Hilfe dieser Module lässt sich unsere Übungsaufgabe einfach lösen. Das vollständige Programm findest du in der Datei **zeitanzeige4.py**. Denke daran, dass die Dateien **geheim.py**, **netzwerk.py** und **display.py** auf dem Board sein müssen. Das gilt ebenso für die Datei **ssd1306.py**.

Zuerst müssen wir alles importieren, das im Hauptprogramm angesprochen wird.

```
import time
import ntptime
from netzwerk import *
import display as disp
```

Hier sehen wir drei Import-Varianten.

**import time** importiert alle Elemente aus **time**. Sie müssen über den Modulnamen angesprochen (**time**.) werden.

**from netzwerk import \*** importiert alle Elemente aus **netzwerk**. Sie können ohne den Modulnamen angesprochen werden.

**import display as disp** importiert alle Elemente aus **display**. Sie müssen über den modifizierten Modulnamen **disp**. angesprochen werden.

Die Ausgabe auf das kleine Display muss auf Grund des beschränkten Platzes wohl überlegt werden. Ausserdem wird jetzt die Zeit nicht nur ein Mal ausgegeben, sondern laufend. Zwischen zwei Ausgaben wird eine Pause von 0.1 Sekunden eingelegt.

```
if wlan.isconnected():
    disp.oled.fill(0) # Bildschirm löschen
    conf = wlan.ifconfig() # (ip, subnet, gateway, dns)
    disp.text_line(wlan.config('ssid'),1);
    disp.text_line(conf[0],2);
    disp.oled.show();
    setze_zeit()
    while True:
        zeige_zeit()
        time.sleep(0.1)
```

Eine besondere Herausforderung ist die Funktion **zeige\_zeit()**. **zeit** ist ein Tupel, aus dem die einzelnen Elemente wie aus einem Array ausgelesen werden können. Diese Werte dann werden mit Hilfe des **.format()** - Befehls auf den Bildschirm geschrieben.

## Lektion 14: Zugriff auf Webservice

Da unsere Wetterstation auch Wetterprognosen anzeigen soll, müssen wir uns die notwendigen Informationen über das Internet besorgen.

<https://openweathermap.org>

OpenWeatherMap stellt solche Informationen zur Verfügung. Wir können direkt auf der Webseite suchen, haben aber auch die Möglichkeit die Informationen über ein Programm abzurufen. Dazu stellt uns OpenWeatherMap eine Schnittstelle zur Verfügung. Solche Schnittstellen sind auch unter der Bezeichnung API (Application Programming Interface) bekannt. So ist es auch hier:

<https://openweathermap.org/api>

Einige Informationen sind kostenlos, andere benötigen ein Monatsabo. Wir konzentrieren uns auf die kostenlosen Angebote.

Folgende Möglichkeiten stehen uns zur Verfügung (Stand Jan. 2020):

- Aktuelles Wetter
- Vorhersage über 5 Tage in 3 Stunden Intervallen
- Sehr einfache Wetterkarten
- UV - Index
- Wetterwarnungen
- Max. 60 Abfragen pro Minute
- 95% Verfügbarkeit
- Datenupdateintervall kleiner 2 Stunden für Wetterdaten, kleiner 3 Stunden für Wetterkarten
- Lizenz für Daten und Datenbank: [ODbL](#)
- Lizenz für Karten, APIs usw.: [CC BY-SA 4.0](#)

<https://openweathermap.org/price>

Das sollte für unsere Zwecke genügen.

### Erstellen eines Accounts und eines API - Keys

Du musst einen Account eröffnen und brauchst dazu einen Usernamen, ein Passwort und eine gültige EMail-Adresse. Ausserdem solltest du mindestens 16 Jahre alt sein.

Nach dem Login findest du im Bereich **API keys** bereits einen solchen Key. Den kannst du verwenden. Irgendwo steht, dass ein eine Weile dauern kann, bis der Key wirklich funktioniert. Bei mir war das aber sehr schnell der Fall. Es ist möglich, weitere Keys zu erstellen. Das wird aber in unserem Fall kaum notwendig sein.

### Erste Abfrage im Browser

Wir nutzen diesen Key um eine Abfrage im Browser durchzuführen. Vorher solltest du aber überprüfen, ob deine Ortschaft dem System bekannt ist. Das kannst du direkt auf der Webseite machen.

<https://openweathermap.org/city>

Im Feld **Your city name** gibst du den Namen der Ortschaft ein. Um möglichst präzise zu sein, gibst du Ort und Land ein (z. Bsp. Berlin, DE). Möglicherweise werden mehrere Einträge gefunden. Dann musst du selbst herausfinden, welcher Eintrag für dich korrekt ist. Diese Seite wählst du dann an. In der URL findest du die City-ID.

<https://openweathermap.org/city/2950159> ist eine der Stationen in Berlin.



Unter <https://openweathermap.org/api> findest du Links zu den API - Docs. Heute schauen wir uns das aktuelle Wetter an ( <https://openweathermap.org/current> ).

Um immer sicher die richtige Station zu erreichen, arbeiten wir mit der City ID.

<https://api.openweathermap.org/data/2.5/weather?id=2950159&appid=b6907d289e10d714a6e88b30761fae22>

Im letzten Parameter musst du **DEINE** App-ID eingeben sonst erhältst du eine Fehlermeldung. Die Antwort sieht dann etwas so aus:

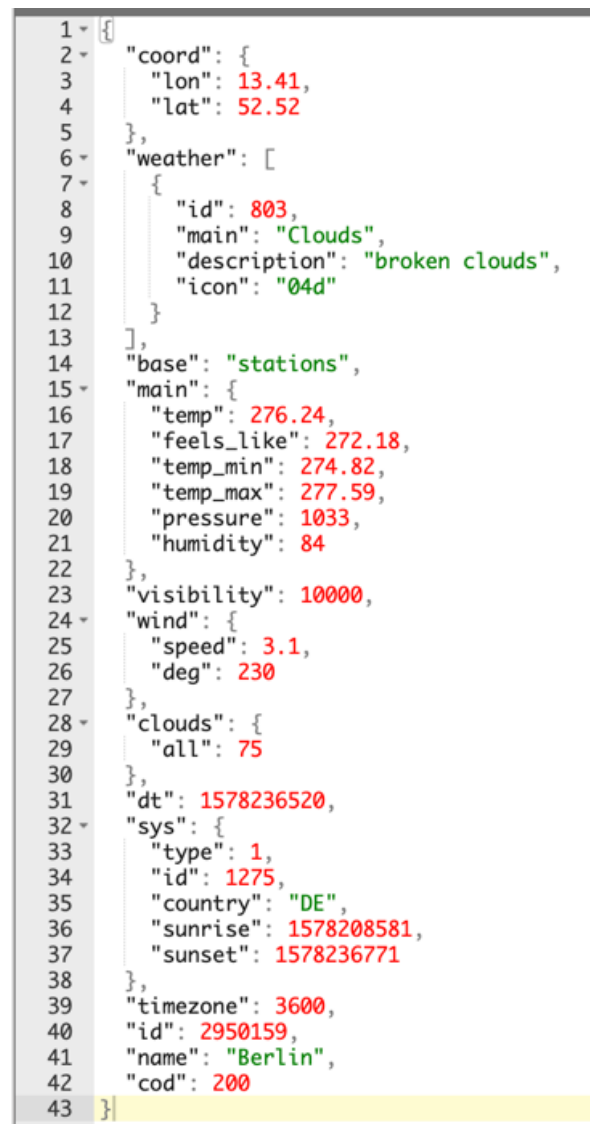
```
{ "coord": { "lon": 13.41, "lat": 52.52 }, "weather": [ { "id": 803, "main": "Clouds", "description": "broken clouds", "icon": "04d" } ], "base": "stations", "main": { "temp": 276.24, "feels_like": 272.18, "temp_min": 274.82, "temp_max": 277.59, "pressure": 1033, "humidity": 84 }, "visibility": 10000, "wind": { "speed": 3.1, "deg": 230 }, "clouds": { "all": 75 }, "dt": 1578236520, "sys": { "type": 1, "id": 1275, "country": "DE", "sunrise": 1578208581, "sunset": 1578236771 }, "timezone": 3600, "id": 2950159, "name": "Berlin", "cod": 200 }
```

Die Aufgabe unserer Wetterstation wird es sein, diese Informationen für uns verständlich anzuzeigen. Dieses Datenformat nennt man JSON (JavaScript Object Notation). Dieses Format lässt sich in Python recht einfach entschlüsseln.

Im Internet findest du auch Viewer, die es erlauben die Informationen etwas strukturierter anzuschauen ( <https://codebeautify.org/jsonviewer> ).

Hier stellst du schnell fest, dass die Beschreibung 'broken clouds' nicht übersetzt ist und die Temperatur mit 276.24 Grad nicht gerade den Erwartungen entspricht.

Durch eine kleine Änderung des Aufrufs kann das verbessert werden.



```
1 {
2   "coord": {
3     "lon": 13.41,
4     "lat": 52.52
5   },
6   "weather": [
7     {
8       "id": 803,
9       "main": "Clouds",
10      "description": "broken clouds",
11      "icon": "04d"
12    }
13  ],
14  "base": "stations",
15  "main": {
16    "temp": 276.24,
17    "feels_like": 272.18,
18    "temp_min": 274.82,
19    "temp_max": 277.59,
20    "pressure": 1033,
21    "humidity": 84
22  },
23  "visibility": 10000,
24  "wind": {
25    "speed": 3.1,
26    "deg": 230
27  },
28  "clouds": {
29    "all": 75
30  },
31  "dt": 1578236520,
32  "sys": {
33    "type": 1,
34    "id": 1275,
35    "country": "DE",
36    "sunrise": 1578208581,
37    "sunset": 1578236771
38  },
39  "timezone": 3600,
40  "id": 2950159,
41  "name": "Berlin",
42  "cod": 200
43 }
```

<https://api.openweathermap.org/data/2.5/weather?id=2950159&lang=de&units=metric&appid=b6907d289e10d714a6e88b30761fae22>

Dadurch erhältst du 'Überwiegend bewölkt' und 2.8 Grad.

## Abruf der Daten mit Python

Wir müssen also einen GET-Request durchführen. Den dazu notwendigen Code können wir mit einer kleinen Änderung direkt dem Abschnitt 5.2. der Micropython - Dokumentation entnehmen ( [https://docs.micropython.org/en/latest/esp8266/tutorial/network\\_tcp.html](https://docs.micropython.org/en/latest/esp8266/tutorial/network_tcp.html) ).

```
import socket

def http_get(url):
    _, _, host, path = url.split('/', 3)
    addr = socket.getaddrinfo(host, 80)[0][-1]
    s = socket.socket()
    s.connect(addr)
    s.send(bytes('GET /%s HTTP/1.0\r\nHost: %s\r\n\r\n' % (path, host), 'utf8'))
    response = ""
    while True:
        data = s.recv(100)
        if data:
            response += str(data, 'utf8')
        else:
            break
    s.close()
    return response

def lade_wetter():
    request_string = "https://api.openweathermap.org/data/2.5/weather?id={}&lang=de&units=metric&APPID={}".format(owm_ortID, owm_id)
    return http_get(request_string)
```

Die Ort-ID und der API-Key werden im Modul **geheim** gespeichert.

**Schau dir unbedingt das Video dazu an.**

Jetzt bekommen wir aber nicht nur die gewünschten Daten zurück, sondern auch den ganzen HTTP-Header. Diesen müssen wir zuerst entfernen.

```
lines = response.split("\r\n")
json_string = lines[-1]
```

Jetzt haben wir zwar unseren JSON-String, der ist aber nur ein Text und noch keine verarbeitbare Datenstruktur. Dazu können wir das Modul JSON verwenden.

```
import json
wetterdaten = json.loads(json_string)
```

Jetzt haben wir ein Dictionary, das mit normalen Python - Befehlen ausgewertet werden kann. Nähere Informationen über den Datenaufbau findest du unter <https://openweathermap.org/current>

## Auslesen der Temperatur aus den Wetterdaten

Aus den JSON-Daten sehen wir, dass die Temperatur in **main** eingebunden ist. Wir extrahieren also zuerst **main** aus den Wetterdaten.

```
wetterdaten_main = wetterdaten["main"]
```

Aus dieser Datenmenge entnehmen wir die Temperatur und schreiben sie auf die Konsole.

```
print("Temperatur: {} Grad".format(wetterdaten_main["temp"]))
```

## Übungsaufgabe

Schreibe folgende Informationen auf das OLED-Display:

**Ort, Land**

**Temperatur**

**Luftdruck**

## Lektion 15: Eine bessere Darstellung

### Die Lösung der Übungsaufgabe *wetter6.py, netzwerk.py*

Zuerst müssen wir feststellen, wo die Daten übermittelt werden.

```
{
  "timezone": 3600,
  "cod": 200,
  "dt": 1579644362,
  "base": "stations",
  "weather": [
    {
      "id": 800,
      "icon": "01n",
      "main": "Clear",
      "description": "Klarer Himmel"
    }
  ],
  "sys": {
    "country": "CH",
    "sunrise": 1579590293,
    "sunset": 1579622953,
    "id": 6941,
    "type": 1
  },
  "name": "Regensdorf",
  "clouds": {
    "all": 0
  },
  "coord": {
    "lon": 8.47,
    "lat": 47.43
  },
  "visibility": 6000,
  "wind": {
    "speed": 0.5
  },
  "id": 2659083,
  "main": {
    "feels_like": -5.41,
    "pressure": 1036,
    "temp_min": -4,
    "humidity": 92,
    "temp_max": -0.56,
    "temp": -2.59
  }
}
```

Die Temperatur haben wir ja schon früher gesehen. Wir finden sie im Abschnitt **main** mit dem Schlüsselwort **temp**.

```
wetterdaten_main = wetterdaten["main"]
temp = wetterdaten_main["temp"]
```

Den Luftdruck finden wir ebenfalls im Abschnitt **main**:

```
druck = wetterdaten_main["pressure"]
```

Das Land finden wir im Abschnitt **sys**:

```
wetterdaten_sys = wetterdaten["sys"]
country = wetterdaten_sys["country"]
```

Der Ortsname ist nicht einem Abschnitt untergeordnet. Wir können ihn direkt auslesen:

```
ort = wetterdaten["name"]
```

Das muss jetzt nur noch auf dem Display dargestellt werden (siehe Video).

## Ein verbessertes Netzwerkmodul

**wetter7.py, netwerk15.py**

In unserem Hauptprogramm benötigen wir noch viel Wissen, wie die Internet - Anbindung funktioniert. Aus diesem Grund bauen wir netwerk.py aus und erstellen eine Klasse, die das Aufbauen der Verbindung und das Absetzen eines Get - Requests für uns erledigen kann. Ich gehe hier nicht auf Details ein, der Zweck dieses Moduls ist es, dir einfache Mittel im Umgang mit WLAN und Internet zur Verfügung zu stellen.

Wie das mit den Klassen genau funktioniert, wird Thema einer späteren Lektion in **Micropython Grundlagen** sein.

**netwerk15.py** (Version 1.5) stellt die Klasse **WiFi** und automatisch auch eine Instanz mit dem Namen **wifi** zur Verfügung. Für dich ist nur wichtig, dass du mit **from netwerk15 import \*** das Objekt **wifi** verfügbar machen kannst. Alle Funktionen werden mit **wifi.** aufgerufen.

netwerk1.py Vers. 1.5, Klasse WiFi

Die Instanz wifi wird automatisch zur Verfügung gestellt und muss nicht erzeugt werden.

Es dürfen keine weiteren Instanzen von WiFi erzeugt werden.

Funktion	Funktion	Beschreibung
connect(ssid, passwort, timeout = 10000)	<b>ssid</b> und <b>passwort</b> : Zugangsdaten <b>timeout</b> : maximale Wartezeit in Sekunden	Verbindet mit dem WLAN und gibt True oder False zurück.
isconnected()		True, wenn Verbindung besteht.
get_wlan()		Gibt das wlan - Objekt aus network zurück. Das wird normalerweise nicht benötigt.
get_ip()		Gibt die IP-Adresse zurück.
get_ssid()		Gibt die SSID zurück.
http_get(url)	<b>url</b> : auf diese url wird der Get-Request gemacht	Führt einen Get-Request aus und gibt das Resultat zurück.

Beispiel:

```
if wifi.connect(wlan_ssid,wlan_passwort):
    print("Verbindung hergestellt")
else:
    print("Verbindung konnte nicht hergestellt werden.")
```

## Der Webserver

### *WebTest1.py, netzwerk15.py*

Der ESP32 kann auch als Webserver arbeiten. Auf einer Webseite lassen sich wesentlich mehr Informationen darstellen als auf dem kleinen OLED.

Ein passendes Modul finden wir auf Github ( <https://github.com/jczic/MicroWebSrv> ). Ich habe mit Absicht ein sehr einfaches Modul gewählt, da wir dieses besser durchschauen können. Wir arbeiten hier nur mit der Datei **microWebSrv.py**. Diese Datei wird auf das Board kopiert.

Für unsere erste Webseite brauchen wir 3 Teile.

#### Start des Webservices:

```
srv = MicroWebSrv(webPath='www/')
srv.MaxWebSocketRecvLen = 256
srv.WebSocketThreaded = True
srv.Start(True)
```

Der Webserver läuft jetzt im Hintergrund.

#### Erzeugen einer Webseite:

```
@MicroWebSrv.route('/test')
def httpHandler(httpClient, httpResponse) :
    httpResponse.WriteResponseOk( headers = ({ "Cache-Control": "no-cache" }),
                                     contentType = "text/html",
                                     contentCharset = "UTF-8",
                                     content = html("Hallo"))
```

Diese Seite kann zum Beispiel unter <http://192.168.1.48/test> abgerufen werden. Die IP-Adresse wird bei der Verbindung zum WLAN zugewiesen. Als content muss eine vollständige HTML - Seite übergeben werden.

#### Die HTML - Seite:

```
def html(content) :
    return """\
<!DOCTYPE html>
<html lang=en>
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="refresh" content="60">
    <title>Webtest</title>
  </head>
  <body>
    <h1>ESP32 Wetterstation</h1>
    <h3>
      {}
    </h3>
  </body>
</html>
""".format(content)
```

Der auszugebende Text wird hier in einen <h3> - Tag eingeschlossen. Zur Erinnerung: """ leitet einen mehrzeiligen String ein, der mit weiteren """ abgeschlossen werden. Der Parameter in format() wird an der Position der beiden Klammer {} eingesetzt.

## Mehrere Webseiten

**WebTest2.py, netzwerk15.py, webseiten01.py**

Wir erstellen für die weiteren Experimente drei Webseiten: die **Homepage** ("/"), die **Wetterseite** ("/wetter") und eine **Log** - Seite ("/log"), die uns bei der Fehlersuche hilft.

Wir erstellen dafür ein eigenes Modul (**webseiten**). Da dieses vermutlich noch oft geändert wird, bezeichnen wir es mit der Version 0.1 und nennen die Datei **webseiten01.py**.

Gestartet wird mit **startHTTP()**. Dabei werden die drei oben genannten Webseiten zur Verfügung gestellt. Der Inhalt wird mit Hilfe von Instanzen der Klasse **BigScreen** übermittelt. Die drei Instanzen **hp** (Homepage), **wetter** (Wetterseite) und **log** (Log - Seite) werden automatisch erzeugt.

### Die Klasse BigScreen

Funktion	Funktion	Beschreibung
BigScreen(titel, refresh)	<b>titel:</b> Titel der Seite <b>refresh:</b> nach soviel Sekunden wird die Webseite automatisch neu geladen	Erzeugen der Instanz. Dies geschieht automatisch.
set_refresh(refresh)	<b>refresh:</b> nach soviel Sekunden wird die Webseite automatisch neu geladen	Es wird eine neue Refresh - Zeit gesetzt. Diese tritt erst nach dem nächsten Seitenaufruf in Kraft.
clear()		Der Text der Webseite wird gelöscht. Es wird nur noch der Titel dargestellt.
add(line)	<b>line:</b> Textzeile, die hinzugefügt werden soll.	Hinzufügen einer Textzeile. add() fügt eine Leerzeile hinzu.
get_content()		Gibt alle hinzugefügten Textzeilen in einem String zurück. Der Zeilenumbruch wird mit   markiert.
print():		Gibt alle Textzeilen auf die Konsole aus.
getHTML():		Gibt den HTML-Code für die ganze Webseite zurück. Dazu wird die Hilfsfunktion <b>_html()</b> verwendet.

## Die Funktion \_html()

```
def _html (titel, content, refresh) :
    return """\
    <!DOCTYPE html>
    <html lang=de>
    <head>
        <meta charset="UTF-8" />
        <meta http-equiv="refresh" content="{refresh}">
        <title>{titel}</title>
    </head>
    <body>
        <h1>{content}</h1>
        <h3>
            {refresh}
        </h3>
    </body>
    </html>
    """.format(refresh, titel, titel, content)
```

Der Seite wird ein Titel (Parameter **titel**) mitgegeben. Dieser erscheint dann oben auf der Seite. Der eigentliche Inhalt wird im Parameter **content** übergeben und auf der Seite dargestellt. Die Seite soll sich nach einer gewissen Zeit selbstständig neu laden. Dieser Zeitabstand wird im Parameter **refresh** in Sekunden angegeben.

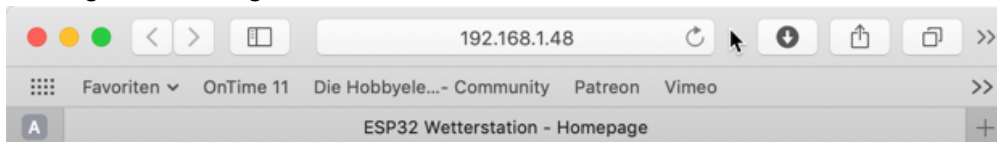
## Darstellung des Wetters

**WebWetter1.py, netzwerk15.py, webseiten01.py**

Bevor wir weitere Daten darstellen, versuchen wir die bisherigen Daten mit Hilfe der neuen Module im Browser darzustellen.

### Die Homepage

Sie zeigt uns die Möglichkeiten auf.



## ESP32 Wetterstation - Homepage

**Die ESP32 - Wetterstation kennt momentan folgende Funktionen:**

**Das aktuelle Wetter:** <http://192.168.1.48/wetter>

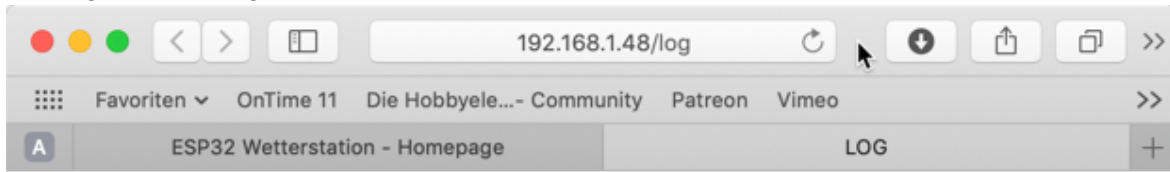
**Ein LOG als Debug-Hilfe beim Programmieren:** <http://192.168.1.48/log>

Das wird so realisiert:

```
def homepage() :
    ip = wifi.get_ip()
    hp.add("Die ESP32 - Wetterstation kennt momentan folgende Funktionen:")
    hp.add()
    hp.add('Das aktuelle Wetter: <a href="http://{ip}/wetter">http://{ip}/wetter</a>'.format(ip, ip))
    hp.add('Ein LOG als Debug-Hilfe beim Programmieren: <a href="http://{ip}/log">http://{ip}/log</a>'.format(ip, ip))
```

## Die Log - Seite

Sie zeigt uns die aufgerufenen Funktionen Schritte.



# LOG

**Mit WLAN verbinden...**

**Mit spWLAN verbunden, IP-Adresse 192.168.1.48**

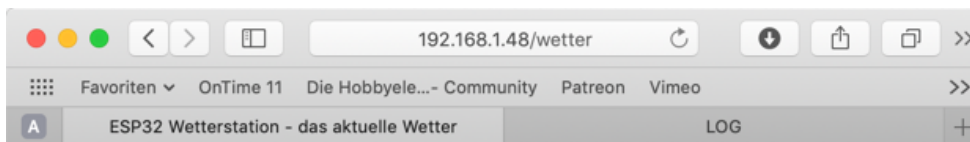
**Wetter geladen**

**Wetterdaten anzeigen**

Die Einträge werden so erstellt:

```
disp.text_line("verbinden...",2)
disp.oled.show();
log.add("Mit WLAN verbinden...");
```

## Die Wetterseite



# ESP32 Wetterstation - das aktuelle Wetter

**Regensdorf, CH**

**Temp. 3.2 Grad**

**Druck 1016 hPa**

Auch diese ist einfach zu programmieren:

```
def zeige_wetter(wetterdaten):
    log.add("Wetterdaten anzeigen");
    wetterdaten_main = wetterdaten["main"]
    wetterdaten_sys = wetterdaten["sys"]
    ...
    disp.text_line("Druck {} hPa".format(wetterdaten_main["pressure"]),6)
    disp.oled.show()
    wetter.add("{} , {}".format(wetterdaten["name"],wetterdaten_sys["country"]))
    wetter.add("Temp. {} Grad".format(wetterdaten_main["temp"]))
    wetter.add("Druck {} hPa".format(wetterdaten_main["pressure"]))
```



## Lektion 16: Automatische Aktualisierung und Zeitanzeige

### Anzeige der Zeit im Log

*WebWetter2.py, netzwerk20.py, webseiten02.py*

Die Ermittlung der aktuellen Zeit haben wir bereits in Lektion 12 besprochen. Das bauen wir noch etwas aus und stellen das in einem Modul zur Verfügung.

Da die Ermittlung der aktuellen Zeit eine Verbindung ins Internet erfordert, bringen wir die Funktionen im **Netzwerkmodul** unter.

Dieses wird um folgende Funktionen erweitert (**netzwerk20.py**):

Funktion	Funktion	Beschreibung
set_timezone(zone)	<b>zone:</b> Zeitverschiebung in Sekunden	Die Zeitverschiebung ist beim Start auf 3600 Sekunden festgelegt. Das kann hier geändert werden.
get_seconds()	Aktuelle Zeit in Sekunden (UTC)	Die aktuelle UTC-Zeit wird über das Internet gelesen.
get_local_seconds()	Aktuelle Lokalzeit in Sekunden	Die aktuelle UTC-Zeit wird über das Internet gelesen und mit Hilfe der Zeitzone in lokale Zeit umgewandelt.
get_time_values(seconds)	<b>seconds</b> ist die Zeit in Sekunden. Diese wird in die Einzelwerte umgerechnet.	Es werden YYYY, MM, DD, HH, MM, SS, day of week (0=Montag), day of year zurückgegeben.
date_text(seconds)	<b>seconds</b> ist die Zeit in Sekunden.	Gibt tt.mm.yyyy zurück.
time_text(seconds)	<b>seconds</b> ist die Zeit in Sekunden.	Gibt hh:mm:ss zurück.
date_time_text(seconds)	<b>seconds</b> ist die Zeit in Sekunden.	Gibt tt.mm.yyyy hh:mm:ss zurück.

Um im Log die Zeit automatisch auszugeben, wird im Modul **webseiten** noch eine Funktion zu der Klasse **BigScreen** hinzugefügt (**webseiten02.py**):

Funktion	Funktion	Beschreibung
add_log(line)	<b>line:</b> Text ohne Zeitangabe	Die exakte Zeitangabe wird automatisch dem Text vorangestellt. Ohne Text fügt add_log() eine Leerzeile ohne Zeitangabe hinzu.

Im Beispielprogramm **WebWetter2.py** werden alle **log,add()** - Aufrufe in **log.add\_log()** umgewandelt. Es darf nicht vergessen werden, die neuen Module **netzwerk20** und **webseiten02** auf das Board zu kopieren.

## Wetterdaten automatisch aktualisieren

**WebWetter3.py, netzwerk20.py, webseiten02.py**

Die Webseite hat bereits einen Refresh definiert. Wir müssen aber noch die Daten in regelmässigen Abständen von OpenWeatherMap abholen. Das sollte nicht zu häufig geschehen, daher machen wir das nur alle 2 Minuten.

```
while True:
    wetterdaten = lade_wetter()
    zeige_wetter(wetterdaten)
    time.sleep(120)
```

Der Zeitpunkt dieser Abholung wird in die Ausgabe geschrieben.

```
wetter.add("Gelesen am {}".format(wifi.date_time_text(wifi.get_local_seconds())))
```

Das ist gleich ein kleines Anwendungsbeispiel für die neuen Zeit - Funktionen im Netzwerk-Modul.

## Wann wurden die Wetterdaten gemessen?

**WebWetter4.py, netzwerk20.py, webseiten02.py**

Die Messung der Wetterdaten erfolgt etwa alle 10 Minuten. Wir können das überprüfen, in dem wir das entsprechende Datenfeld auslesen. Gleichzeitig können wir auch noch die Zeitzone auslesen und für die Umrechnung verwenden.

```
timezone = wetterdaten["timezone"]
wifi.set_timezone(timezone)
serverzeit = wetterdaten["dt"] + timezone
zeit_string = wifi.date_time_text(serverzeit)
```

## Erweiterte Funktionen und Fehlertoleranz

**WebWetter5.py, netzwerk20.py, webseiten02.py, openweathermap10.py**

Beim Abruf der Wetterdaten können Fehler auftreten. Möglicherweise antwortet der Server nicht oder ein Datensatz wird nicht geliefert. In diesem Fall darf unser Programm nicht abstürzen. Ausserdem erfolgt die Messung bei **OpenWeathermap** nicht immer exakt nach der selben Zeit. Unser Ziel ist es, Aktualisierungen so rasch als möglich zu erhalten, ohne dass wir unnötige Abfragen machen müssen.

Deshalb habe ich diese Funktionalität in das neue Modul **openweathermap** gepackt und stelle es zur Verfügung. Wie es intern genau funktioniert, werden wir hier nicht besprechen. Wir konzentrieren uns auf die Verwendung des Moduls.

Selbstverständlich können wir im Forum über Details und Verbesserungsvorschläge diskutieren.

Ab sofort wird zu jeder Lektion eine aktuelle Referenz mit dem Titel **Die Module der ESP32 Wetterstation** in den Begleitunterlagen mitgeliefert. Sie beschreibt das Interface der von mir bereitgestellten Module.

In WebServer5.py wird dieses neue Modul verwendet.

Momentan verarbeiten wir nur das aktuelle Wetter, noch keine Wetterprognosen. Darum erstellen wir eine Instanz von WetterAktuell.

```
aktuell = WetterAktuell(owm_id, owm_ortID, "", "", 600)
```

Dazu geben wir die **AppID** und die **OrtsID** mit und legen den **Refresh** auf 600 Sekunden fest. Die Daten vom Server werden etwa alle 10 Minuten erneuert, so dass es keinen Sinn macht, diese häufiger abzurufen.

Das Objekt aktuell enthält jetzt die ganze Funktionalität des Abrufs der aktuellen Wetterdaten. unser Hauptprogramm vereinfacht sich dadurch stark.

Nach der Initialisierung des Objekts können mit seiner Hilfe die Daten abgerufen werden. Das erfolgt in der üblichen Schleife.

```
while True:
    if aktuell.refresh():
        zeige_wetter()
    time.sleep(60)
```

**aktuell.refresh()** hält selbst den bei der Initialisierung vorgegebenen Takt ein. Wenn allerdings nach den vorgegebenen 10 Minute keine neuen Daten bereitstehen, wird häufiger abgefragt, bis neue Daten geliefert werden. **aktuell.refresh()** gibt **True** zurück, wenn neue Daten empfangen wurden. In diesem Fall werden sie an die Webseite weitergegeben.

In **zeige\_wetter()** werden die zuletzt abgerufenen Daten dem **aktuell** - Objekt entnommen. Einige Daten stehen fertig aufbereitet zur Verfügung. So gibt zum Beispiel **aktuell.get\_messzeit\_text()** den Messzeitpunkt fertig formatiert zurück. Für andere Daten müssen die jeweiligen Topics angegeben werden.

```
aktuell.get_data("main", "pressure")
```

gibt zum Beispiel den Luftdruck zurück.

Eine Beschreibung der zur Verfügung stehenden Funktionen findest du in **Die Module der ESP32 Wetterstation**.

## Lektion 17: Speichern von Daten in einer Datei

Bei Verwendung des Logs können mit der Zeit sehr viele Daten anfallen. So ist es sinnvoll, diese nicht dauernd im Arbeitsspeicher zu halten. Aus diesem Grund wird im Modul **webseiten** eine Möglichkeit geschaffen, die Daten in einer Datei zu speichern.

Wir werden in Zukunft versuchen, das Modul **webseiten** kompatibel zu älteren Versionen unseres Programms zu halten. Alle Änderungen am Modul ergänzen die Funktionalität, bestehende Funktionen bleiben unangetastet. Man sagt auch, dass in der neuen Version keine 'breaking changes' enthalten sind. Aus diesem Grund verzichten wir in Zukunft auf die Angabe der Version im Dateinamen. Stattdessen erlauben wir eine Abfrage der Version über eine Funktion.

Die Dateifunktionen werden automatisch bei **log** verwendet. Die drei Instanzen von BigScreen werden jetzt so erstellt:

```
log = BigScreen("LOG",10,"log.txt")
hp = BigScreen("ESP32 Wetterstation - Homepage",300)
wetter = BigScreen("ESP32 Wetterstation - das aktuelle Wetter",60)
```

Das kann jederzeit geändert werden. Mit

```
log.set_filename()
```

kann jederzeit zur alten Methode zurückgekehrt werden.

Ebenso ist es möglich mit

```
wetter.set_filename("wetter.txt")
```

auf Dateispeicherung zu wechseln.

Hier noch etwas zu den Filefunktionen in Python. Wir besprechen hier nur die Funktionen, die wir momentan brauchen.

### Erzeugen einer Datei, schreiben in eine Datei

Sobald wir eine Datei zum Schreiben öffnen, wird sie automatisch angelegt, falls sie noch nicht existiert.

```
with open(self.filename,"a") as file:
    file.write("Ein Text" + "\n")
```

In diesem Beispiel öffnen wir die Datei um einen Text hinzuzufügen. **"a"** ist der Filemode, in diesem Fall **append**. Eine bestehende Datei würde geöffnet und der Text am Ende hinzugefügt.

```
with open(self.filename,"w") as file:
    file.write("Ein Text" + "\n")
```

Hier wird die Datei geöffnet oder neu erzeugt. Der bestehende Text wird aber gelöscht und durch den neuen Text ersetzt.

Mit **"\n"** fügen wir einen Zeilenumbruch hinzu.

Das Konstrukt mit **with .. as file** öffnet das File und weist es der Filevariablen **file** zu. Diese lebt nur lokal und wird automatisch nach Beenden des Blocks wieder abgebaut. Deshalb erfolgt das Schliessen der Datei automatisch und ein Aufruf von **close()** ist nicht notwendig.

## Lesen einer Datei

```
with open(self.filename, "r+") as file:
    return file.readlines()
```

Die Datei wird geöffnet und der Inhalt gelesen. **"r"** steht für **read** und erlaubt nur lesenden Zugriff auf die Datei. Mit **"r+"** erlauben wir nicht nur das Lesen, sondern auch das Schreiben. Das hat in unserem Fall den Vorteil, dass wir keine Fehlermeldung erhalten, wenn die Datei noch nicht existiert. Sie wird in diesem Fall automatisch angelegt.

Wir erhalten alle gespeicherten Zeilen zurück. In jeder Zeile bleibt das **"\n"** erhalten. Bei der Ausgabe der Zeilen müssen wir dieses Zeichen mit **rstrip()** wieder entfernen.

```
for line in self.read_lines_from_file():
    print(line.rstrip())
```

Bei der Ausgabe auf eine Webseite wird für den Zeilenumbruch jeweils **<br>** eingesetzt.

```
content = ""
for line in self.read_lines_from_file():
    content += line.rstrip() + "<br>"
return content
```

## Lektion 18: Das OneCall API

In dieser Lektion lernen wir ein neues API von OpenWeatherMap kennen. Es wurde neu eingeführt und wir werden es in den nachfolgenden Lektionen verwenden. Das neue API bezieht das aktuelle Wetter und die Wetterprognose mit einem Aufruf.

Wir erhalten dadurch eine grosse Datenmenge zurück und müssen aus diesem Grund einige Speicheroptimierungen vornehmen.

<https://openweathermap.org/api/one-call-api>

### Aufruf

<https://api.openweathermap.org/data/2.5/onecall?lat={lat}&lon={lon}&exclude={part}&appid={API KEY}>

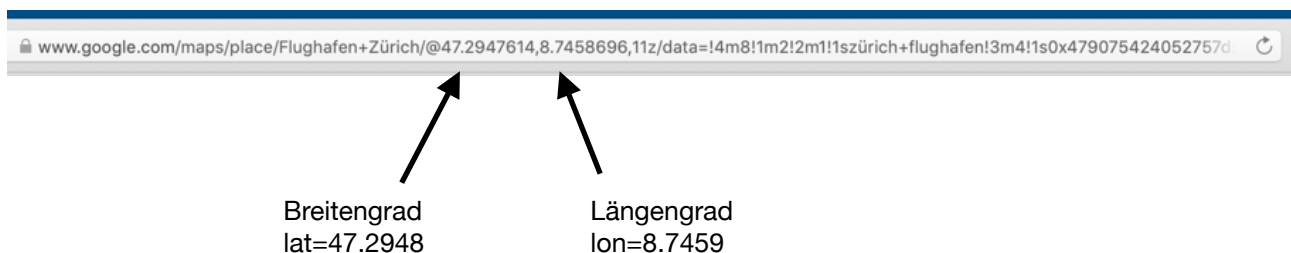
Es gibt zwei Unterschiede zum bisherigen Aufruf.

1. Wir verwenden das Schlüsselwort **oncall** anstelle von **weather**.
2. Wir können nicht mehr mit der Orts-ID oder dem Ortsnamen arbeiten. Stattdessen benötigen wir die Koordinaten in Form von Längen- und Breitengraden.

Falls SSL - Fehler auftreten, kann für den Moment der Aufruf auch mit http:// erfolgen.

### Woher bekommst du die Koordinaten?

Eine Möglichkeit ist Google Maps ( <https://maps.google.com> ). Dort kannst du einen beliebigen Punkt suchen und findest danach die Koordinaten in der url.



Diese Koordinaten hinterlegen wir ebenfalls in der Einstellungsdatei **geheim.py**.

```
# WLAN
wlan_ssid      = "ssid"
wlan_passwort  = "gemein"

# OpenWeatherMap
owm_id        = "1234567890abcd"
owm_ortID     = 2659083
owm_ort       = "Regensdorf"
owm_land      = "CH"
owm_lat       = 47.4315
owm_lon       = 8.4661
```

## Notwendige Programmänderungen

Nebst dem geänderten Aufruf müssen wir noch diverse Optimierungen vornehmen. So werden in dieser Lektion unsere Module **geheim**, **netzwerk**, **webseiten** und selbstverständlich **openweathermap** bearbeitet.

In **geheim** haben wir, wie oben gezeigt, nur die Koordinaten des Zielortes eingetragen.

In den anderen Modulen wurden teilweise substanzielle Änderungen vorgenommen.

### webseiten.py

Bisher haben wir immer alle in einer Datei abgelegten LOG-Daten dauerhaft gespeichert. Neu bleiben nur noch die Daten der aktuellen Session erhalten.

Dazu wird im Konstruktor der Klasse **BigScreen** `self.clear()` aufgerufen. `self.clear()` löscht den internen Buffer und falls ein Filename angegeben ist, auch den Inhalt der Datei.

### netzwerk20.py

Das Modul **urequests** erlaubt uns eine einfachere Durchführung des GET - Requests. Die Klasse **WiFi** wird daher um die Funktion **get\_request(url)** erweitert.

```
def get_request(self, url):  
    return requests.get(url)
```

### openweathermap20.py

Diese Datei ist aus `openweathermap11.py` entstanden. **Momentan ist hier alles experimentell und kann noch nicht normal verwendet werden.**

Die neue Klasse **WetterOneCall** stellt mit **get\_request\_string()** die url für den OneCall - Abruf bereit.

## Garbage Collection (Das Modul gc)

In **Webwetter8.py** wird etwas mit der Speicherverwaltung experimentiert.

Als erste, einfache Massnahme wird **esp** importiert und mit **esp.osdebug(None)** die Generierung von Debug - Meldungen ausgeschaltet.

Weitaus wichtiger ist die Verwendung des Moduls `gc`. Damit lässt sich der freie Speicherplatz ausgeben und eine garbage collection erzwingen.

```
import gc  
  
# Ausgabe des belegten und verfügbaren Speicherplatzes  
print("alloc: {}, free: {}".format(gc.mem_alloc(), gc.mem_free()))  
  
# Garbage collection erzwingen  
gc.collect()
```