

RainbowWarrior

Eine Einführung in die
Microcontrollerprogrammierung mit
MicroPython und dem ESP32 Development
Board



Christina Rost

Hochschule für Gestaltung Schwäbisch Gmünd
Internet der Dinge - Gestaltung vernetzter Systeme
in Kooperation mit der Hochschule für Technik und Wirtschaft Aalen

embedded systems - technische Informatik. 3. Semester.
bei Prof. Dr.-Ing. Jürgen Schüle, HS AA

Benötigte Materialien	4
Getting Started	5
Das ESP32 Development Board	5
Eine kurze Einführung ins Terminal	7
Terminalbefehle: Aktuelles Arbeitsverzeichnis	8
Terminalbefehle: Ordnerinhalt aufliste	8
Terminalbefehle: Verzeichnis wechseln	8
Terminalbefehle: Dateien kopieren	9
Abhängigkeiten	10
FreeRTOS	11
Terminalbefehle: USB-Port ermitteln	11
freeRTOS installieren	13
freeRTOS und das MicroPython Framework herunterladen	13
Compiler	16
Interpreter	17
Programmiersprachen	18
Bauteile verbinden	20
LED Strip an ESP anschließen	20
Eine Verbindung zum ESP32 herstellen -rshell	20
Micropython	21
Micropython: Module	21
Micropython: Neopixel	22
Micropython: Das machine-Modul	22
Micropython Neopixel: LEDs leuchten lassen	24
Micropython Neopixel: Farben	24
E: Datentypen in Python	28
Animationen Teil 1: Blinken	29
Schleifen	29
for-Schleife	30
While-Schleife	32
Operatoren und Ausdrücke	34
Funktionen	35
Funktionen mit optionalen Parametern	37
Aufgabe - Blinken in zwei Farben	38
Ein erstes kleines Skript	39
Skripten : Der Editor	39
Skripte auf dem ESP ausführen	43
Aufgabe: Schleifen erstellen	43
Dateisystem auf dem ESP	44
Animationen Teil 2: running dots	46
Kommentare in (Micro)Python	48
Micropython Animationen: running dots mit Schweiß - white level	50
Der Touch Sensor	51
Animationen steuern	51
Bedingte Anweisungen / Verzweigungen	53
Animation mit dem Touch Sensor starten	53
E: Iterator	56
E: Lokale und Globale Variablen in Python	57
E: Multithreading	59
Implementierung von _thread	60
Der Photowiderstand	62
Helligkeit der LEDs steuern	62
Analog Digital Converter	63
Photowiderstand mit ESP verbinden	64
Das Breadboard	65
LDR testen	65
ESP als WebServer	66
WLAN auf dem ESP	66
AP-Mode	66
STA-Mode	68
mDNS Service	69

Das WebServer-Modul	70
Aufbau einer HTML-Seite	71
HTML - Formulare	72
HTML Forulare - Kommunikation mit dem Server	73
Die HTML-Seite	74
HTML-Form	75
Eingaben auf dem Server verarbeiten	77
Thread-Messages vom Server in main empfangen und verarbeiten	78
Zwei ähnliche Routinen in einer Funktion zusammenfassen	80
Der Piezo-Speaker	81
PWM - Pulsweitenmodulation	81
Das PWM-Modul	81
Songs mit dem Piezo spielen - das RTTTL-Modul	83
Songs mit PWM abspielen	84
Dropdownmenü für Songs in die Webseite einfügen	86
Verarbeitung der Daten auf dem Webserver	87
Verarbeitung der Daten in main	88
RainbowWarrior als Wecker nutzen	90
Real Time Clock - das RTC Modul	90
Zeitabhängige Funktinen - das utime-Modul	91
Systemzeit auf der Webseite anzeigen	92
E: String Formaterung	92
Systemzeit über die Webseite einstellen	97
Fehler in der Eingabe auffangen	99
Das Timer-Modul	100
Timer Callback	102
Alarm stoppen	103
Terminalbefehle MacOS	105
Terminalbefehle Windows	106
Terminalbefehle: Linux	107

Benötigte Materialien

Für einfache Pixel-Animationen:

- ESP32 Development Board
- LED Strip SK6812 RGBW, ca 170 mm lang (oder nach Belieben)
- Breadboard
- Jumperwires

Für das gesamte Projekt zusätzlich:

Neben der gezeigten Variante gibt es sehr viele verschiedene Möglichkeiten, LED auf Plexiglas in Szene zu setzen, siehe im Kapitel "Plexiglas". Hierbei darf man auch gerne kreativ sein! Jedoch sollte bei abweichendem Design im Vorfeld überlegt werden, wie groß das Objekt insgesamt, wie die Plexiglasplatte und wie lang der Strip sein sollte. Mehr dazu im Kapitel "Prototyp bauen".

- Plexiglasplatte - 170 x 170 x 10 mm (oder nach Belieben)
- Holzsockel, passend zur Platte. Hier ca 50 x 70 x 190 mm
- Metallknopf als Touch-Sensor (gerne auch Ersatzknöpfe von Hosen, schöne Nägel o.ä.)
- Piezo Speaker (passiv)
- Photoresistor
- Widerstand, 1kΩ (braun – schwarz – rot – gold)
- Litzen in 3 Farben (Schwarz, rot, gelb oder andere)

Desweiteren werden folgende Werkzeuge benötigt:

- Lötkolben, Lötzinn, Lötsauglitze
- Holzsäge, Kreissäge oder die Möglichkeit, Holz zu verarbeiten
- Bohrmaschine incl. Bohrer in 5mm, ggf. auch 35mm
- Schleifpapier (am besten ein gröberes und ein recht feines)
- Heißklebepistole
- ggf. Öl, Lasur, oder Lack um das Holz zu versiegeln
- Computer inklusive USB Kabel.

Die Anleitung bezieht sich auf Rechner mit installiertem Linux Mint System. Soweit es mir möglich ist, werde ich versuchen, auch die nötigsten Schritte für Windows und MacOS bereitzustellen. Für Vollständigkeit kann ich leider keine Garantie geben, jedoch hilft hier oft eine kurze Internetrecherche zur betreffenden Fehlermeldung.

Getting Started

Das ESP32 Development Board

Beim ESP32 Development Board handelt es sich um einen Mikrocontroller, auch gerne Ein-Platinen-Computer oder Minicomputer genannt.

Neben dem 240 MHz DualCore Prozessor ist der ESP32 noch mit 520 KiB SRAM ausgestattet (SRAM = Static Random Access Memory - also ein Arbeitsspeicher, der Werte und Dateien zwischenspeichern kann solange er mit Strom versorgt wird - wie bei einem Computer. Kib (Kibibyte, 1 KiB = 1024 Byte) und 4 MiB flash-Speicher (= flash EEPROM, ein Speicherbereich, der nicht verloren geht, sobald kein Strom da ist)

Zum Vergleich: Heute haben die meisten PCs (teilweise auch viele Smartphones) mindestens einen Dual-Core Prozessor mit ca 2.5 GHz, das wäre also mindestens 100 Mal schneller als der ESP32.

Außerdem verfügen ebenso die meisten PCs über mindestens zwei, eher noch vier GB an Arbeitsspeicher, sprich vier bis achtmal so viel.

Dies sollte allerdings nicht abschrecken, da bei 240 MHz immerhin durchschnittlich circa 240,000,000 Rechenoperationen pro Sekunde durchgeführt werden. Und dies ist eine beachtliche Menge - absolut ausreichend für einen Microcontroller, mit dem man lediglich "kleine" Aufgaben im Vergleich zum PC erledigen möchte.

Neben der Rechenleistung und dem verfügbaren Speicher bringt der ESP noch 34 GPIO Pins (General Purpose Input/Output Pins) mit sich, das sind also 34 sogenannte Pins (kleine Metallstifte), an denen man über ein angeschlossenes Jumperwire (ein Kabel mit passendem Anschluss für die Pins) zum Beispiel Werte eines Feuchtigkeits- oder Temperatursensors auslesen, Werte an einen elektrische Motor oder - wie in unserem Fall - an einen LED-Strip senden kann.

Außerdem sind bereits einige Sensoren integriert: Der Touch-Sensor beispielsweise, der auf Berührung reagiert, oder der Hall-Sensor, der Schwankungen im Erdmagnetfeld misst. Außerdem ist bereits ein Temperatursensor integriert.

Zusätzlich kann dieses Board nicht nur eine Funkverbindung per WLAN bereitstellen oder sich per WLAN mit einem bestehenden (Heim-) Netzwerk verbinden, sondern ebenfalls eine Verbindung per Bluetooth herstellen.

Auf dem ESP32 läuft ein Mini-Betriebssystem namens freeRTOS (free Real Time Operating System) im Hintergrund. Dadurch hat man zum Beispiel die Möglichkeit zeitgesteuerte Aufgaben zu erstellen.

Die Einsatzmöglichkeiten sind sehr vielfältig. So kann man mit dem ESP32

zum Beispiel die Wetterdaten oder einfache Daten im Haus zur Luftfeuchtigkeit, Luftdruck und Temperatur messen und protokollieren. Ebenso kann man damit elektrische Motoren steuern - per WLAN oder Bluetooth.

Man kann die integrierten Touch-Sensoren - oder einfach Buttons, die angeschlossen wurden - nutzen, um etwas ein- oder auszuschalten. Oder man kann sich - wie hier geschehen - LED Strips kaufen und für diese Animationen programmieren. Einen Webserver erstellen, der eine Webseite zur Steuerung der Animationen bereitstellt. Zusätzlich kann man den integrierten Touch-Sensor verwenden, um diese Animationen zu starten. Man kann einen Piezo-Speaker nutzen, um Töne, oder gleich ganze Melodien zu spielen und einen Photowiderstand, um die Helligkeit der LEDs an die Umgebung anzupassen.

Programmieren kann man den ESP32 auf drei verschiedene Arten, bzw. drei verschiedenen Frameworks programmieren:

Arduino-IDE: Arduino ist - nach dem Raspberry Pi - bestimmt der bekannteste Ein-Platinen-Computer. Daher wird dies wohl auch die geläufigste Sprache bzw das geläufigste Framework sein. Arduino basiert auf C++/C. Man kann, nachdem man einige Einstellungen in der Entwicklungsumgebung für Arduino vorgenommen hat - einfach mit der selben Sprache und der selben Programmierumgebung (IDE = Integrated Development Environment), die man für den Arduino nutzt, auch den ESP32 programmieren.

MicroPython: MicroPython - eine abgespeckte Version von Python) zeichnet sich dadurch aus, dass sie von Anfängern gerne als leicht lernbar eingestuft wird. Allerdings ist hier die Einrichtung, die man im Vorfeld vornehmen muss, etwas komplexer.

ESP-IDF: Das ESP-IoT-Development Framework basiert ebenso auf C++ und ist das offizielle Framework, das vom Hersteller der ESP entwickelt wurde. Dies erfordert - sowohl für die Einrichtung als auch für den Upload und die Ausführung des Codes einige Grundkenntnisse.

Eine kurze Einführung ins Terminal

Mit dem Terminal hat man die Möglichkeit, über Befehle in geschriebener Form den kompletten PC steuern. Hier kann man Dateien anlegen, suchen und packen sowie System-, Netzwerk- oder Hardware-Befehle ausführen oder Benutzer verwalten.

Es sieht zwar sehr simpel aus, ist aber umso mächtiger im Vergleich zu den Optionen, die auf der grafischen Oberfläche zu Verfügung stehen. So können manche Aufgaben oft schneller und sauberer erledigt werden.

Manche Anwendungen stellen auch separate Kommandozeilen-Dienstprogramme zur Verfügung, welche Funktionen bieten, die über die grafische Nutzeroberfläche des Programms nicht zu erreichen sind.

Terminal öffnen:

- Unter Linux: Super - bzw. Starttaste drücken und "Terminal" eingeben. Alternativ kann mit STRG+ALT+T per Tastatur das Terminal geöffnet werden
- Windows: Windows-Taste drücken, "cmd" eingeben, mit Enter bestätigen
- MacOS: CMD Leertaste drücken um Spotlight zu öffnen, "Terminal" eingeben.

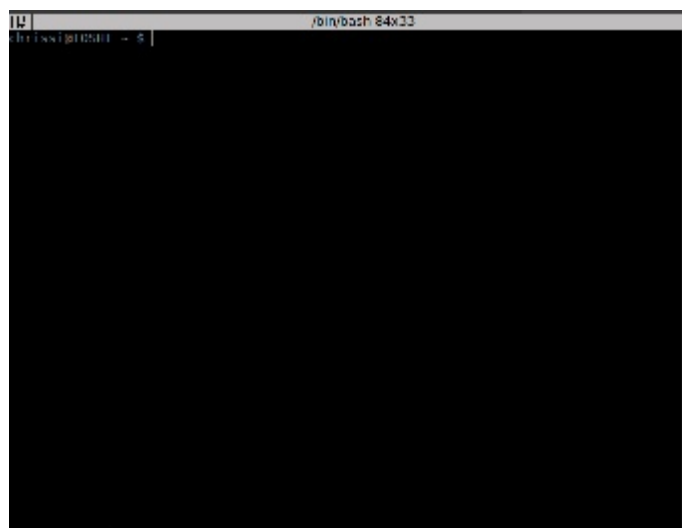
Die hier vorgestellten Befehle beziehen sich hauptsächlich auf Linux. Viele der Befehle sind auch auf MacOS zu finden. Einige auch auf Windows - manche davon aber leicht abgewandelt.

Eine kurze Auflistung der wichtigsten Befehle pro Betriebssystem befindet sich im Anhang!

Auf den ersten Blick sieht das Terminal überhaupt nicht spektakulär aus.

Jede Zeile beginnt mit dem Nutzernamen, gefolgt von dem Computernamen (getrennt durch ein @) Anschließend wird der Ordnername angezeigt, in dem man sich aktuell befindet. Hier sieht man nur ein unscheinbares „~“, welches aussagt, dass man sich gerade im Home-Verzeichnis befindet.

Abgeschlossen wird die Zeile mit einem „\$“, das zum einen signalisiert, dass man gerade als User (nicht etwas als Admin oder Superuser) im Terminal angemeldet ist, zum anderen auch zeigt, dass auf eine Nutzereingabe gewartet wird und nicht etwa ein Befehl gerade ausgeführt wird.



Aktuelles Arbeitsverzeichnis:

pwd (=print working directory)

Falls je nicht sicher sein sollte, ob man sich im richtigen Ordner befindet, kann man einfach **pwd** eingeben, was nichts weiter bedeutet als „gib aktuelles Arbeitsverzeichnis aus“

Ordnerinhalt auflisten

ls (= list)

Mit dem Befehl **ls** gibt man den Inhalt eines Ordners aus. Nur **ls**, ohne Optionen, listet den Inhalt des aktuellen Arbeitsverzeichnisses, während man mit

```
$ ls /home/$USER/Bilder
```

- unabhängig vom derzeitigen Arbeitsverzeichnis - zum Beispiel den Inhalt des Ordners "Bilder" im home-Verzeichnis des aktuellen Benutzers (= \$USER) ausgeben.

Verzeichnis wechseln

cd (=change directory)

Um in ein Unterverzeichnis zu wechseln, gibst du einfach den Befehl **cd** ein, gefolgt durch den Namen des Unterverzeichnisses. Leerzeichen zwischen diesen beiden nicht vergessen!

```
$ cd Bilder
```

wechselt also in einen Unterordner namens "Bilder", der sich im aktuellen Arbeitsverzeichnis befinden muss.

Tipp für Schreibfaule: Man kann auch einfach die ersten Buchstaben des Verzeichnisses eingeben. Mit einem Klick auf die TAB Taste erledigt die Autovervollständigung den Rest der Schreibarbeit – es sei denn es gibt mehrere Dateien oder Verzeichnisse mit dieser Zeichenfolge am Anfang, oder es befindet sich ein Schreibfehler darin.

Möchte man in ein ganz anderes Verzeichnis wechseln, gibt man einfach **cd** ein, gefolgt vom kompletten Verzeichnispfad ein. Dieses Mal muss allerdings ein **/** vor den Pfad, um zu signalisieren, dass es sich nicht um ein Unterverzeichnis handelt

```
$ cd /tmp
```

wechselt nun in das temporäre Verzeichnis namens "tmp", das sich im Root-Verzeichnis (Unix-Äquivalent zu Windows' "C:\"-Laufwerk).

Um wieder ins übergeordnete Verzeichnis zu wechseln, einfach **cd ..** eingeben.

```
$ cd ..
```


Dateien kopieren

cp (=copy)

Um eine Datei zu kopieren, benutzt man den Befehl **cp** (= copy)
Hierzu wird mit **cd** in den Ordner gewechselt, in dem sich die Datei befindet.

Anschließend muss eingegeben werden:

```
$ cp dateiname.html /home/$USER/htmlFiles
```

Dieser Befehl besteht aus drei Teilen, getrennt durch ein Leerzeichen:

1. der Befehl **cp**, gibt an, dass eine Datei kopiert werden soll.

2. die Datei, die kopiert werden soll (hier als Beispiel:

"dateiname.html")

3. der komplette Pfad zum Zielordner (/home/\$USER/htmlFiles).

Wer allerdings Ordner anstelle von Dateien kopieren möchte, muss noch die Option **-r**, was für Rekursiv steht, hinter den Befehl **cd** gehängt werden. So wird der Ordner inklusive allen darin enthaltenen Dateien und Ordnern kopiert.

Das ganze kann dann so aussehen:

```
$ cp -r htmlFiles /home/chrissi
```

Achtung! Ein wichtiger Unterschied zwischen Terminal und der grafischen Benutzeroberfläche: Das Terminal meckert nur, wenn etwas nicht stimmt!

Es ist also völlig normal, dass nach Eingabe des Befehls und Bestätigung mit der Enter-Taste einfach wieder Name/Computername + Arbeitsverzeichnis angezeigt werden! Das signalisiert, dass alles geklappt hat.

Zudem gibt es **keine Sicherheitsabfrage** und **keinen Papierkorb!** gelöscht ist gelöscht. Verschieben ist verschieben. Solange du Lese- und Schreibrechte in den betreffenden Ordnern und den Dateien hast, wird es keine Sicherheitswarnung geben. Daher ist hier **große Vorsicht** geboten!

Dies waren erst einmal die wichtigsten Befehle, damit wir erst einmal im Terminal zurecht finden. Weitere werden an gegebener Stelle folgen und nochmals erklärt.

Abhängigkeiten

Bevor es mit der Installation des Betriebssystems und des Micropython Frameworks losgehen kann ist die Installation von einigen kleinen (Software-)Paketen notwendig, um mit (Micro)Python arbeiten zu können und damit der PC in der Lage ist, eine Verbindung zum ESP aufzubauen um z.B. Dateien zu übertragen.

Außerdem müssen wir das Framework herunterladen.

Unter Linux wird hierzu einfach das Terminal geöffnet und folgende Befehle nacheinander eingeben. Nach jeder Zeile muss mit Enter bestätigt werden. Achtung: der erste Befehl geht über zwei Zeilen!:

```
$ sudo apt-get install git wget make libncurses-dev flex bison gperf python-serial python-pip rsync  
$ sudo pip install --upgrade pip  
$ sudo pip install esptool --upgrade
```

Unter Mac OS sind folgende Befehle nötig (ohne Gewähr! Falls etwas fehlen sollte → Google fragen):

```
$ sudo easy_install pip rsync  
$ sudo pip install pyserial
```

freeRTOS

FreeRTOS (free Realtime Operating System) ist nichts weiter als ein Echtzeitbetriebssystem, zugeschnitten für Mikrocontroller. Es basiert auf einer Mikrokernarchitektur und wurde auf verschiedene Mikrocontroller portiert.

Nur, was ist ein Kernel?

Ein Kernel, oder auch Betriebssystemkern ist - wie der Name schon sagt, der Kern, das Herzstück eines jeden Betriebssystems. Hier wird die Prozess- und Datenorganisation festgelegt, außerdem hat er direkten Zugriff auf die Hardware. Das bedeutet, der Kernel steuert, wann welcher Prozess den Prozessor belegen darf, welchen Speicherbereich dieser benutzen darf, wie die Hardware mit den Prozessen kommuniziert und so weiter.

Falls zusätzlich Lesebedarf besteht, hier einige zusätzliche Informationen:

<https://de.wikipedia.org/wiki/FreeRTOS>

<https://www.searchdatacenter.de/definition/Kernel>

USB Port ermitteln

Bevor mit der Einrichtung des Framework begonnen werden kann, ist es wichtig zu wissen, auf welchem USB Port der ESP angesprochen wird. Jedes USB-Gerät wird vom PC an einen speziellen Port geknüpft, mit dem es vom System angesprochen wird.

Unter Linux, sehr wahrscheinlich auch unter MacOS, ist dies für den ESP32 meist "ttyUSB0" oder "ttyUSB1".

Sobald der Controller eingesteckt ist, kannst du dies auch einfach prüfen:

```
$ ls /dev/ttyUSB*
```

ls bedeutet **list**, sprich es listet nun alle angeschlossenen Geräte (gespeichert im Ordner /dev), die mit "ttyUSB" anfangen. Der * signalisiert, dass nach dieser Zeichenfolge noch andere Zeichen folgen können.

Falls mit dem Befehl ls keine Ausgabe erfolgen sollte, kann man auch die **Kernelmeldungen auslesen**. Hierfür nutzt man - leider wieder nur unter Linux und MacOS - den Befehl

dmesg (=display messages)

```

/bin/bash 80x24
[33421.898098] cdc_ncm 1-1.6:1.6 wwp0s26ul0i6: renamed from wwan0
[33422.391944] usb 2-1.6: new full-speed USB device number 8 using ehci-pci
[33422.501425] usb 2-1.6: New USB device found, idVendor=0cf3, idProduct=3005
[33422.501434] usb 2-1.6: New USB device strings: Mfr=0, Product=0, SerialNumber
=0
[33423.443951] cdc_wdm 1-1.6:1.5: wdm int callback - 0 bytes
[33423.445985] cdc_wdm 1-1.6:1.8: wdm int callback - 0 bytes
[33423.760701] wlp4s0: authenticate with 5c:49:79:da:5c:35
[33423.781652] wlp4s0: send auth to 5c:49:79:da:5c:35 (try 1/3)
[33423.784742] wlp4s0: authenticated
[33423.787884] wlp4s0: associate with 5c:49:79:da:5c:35 (try 1/3)
[33423.793640] wlp4s0: RX AssocResp from 5c:49:79:da:5c:35 (capab=0x431 status=0
aid=3)
[33423.793840] wlp4s0: associated
[33423.794151] ath: EEPROM regdomain: 0x0114
[33423.794152] ath: EEPROM indicates we should expect a country code
[33423.794153] ath: doing EEPROM country->regdmn map search
[33423.794154] ath: country maps to regdmn code: 0x37
[33423.794155] ath: Country alpha2 being used: DE
[33423.794155] ath: Regpair used: 0x37
[33423.794157] ath: regdomain 0x0114 dynamically updated by country IE
[33423.800988] IPv6: ADDRCONF(NETDEV_CHANGE): wlp4s0: link becomes ready
[33494.695520] input: 88:69:C2:CD:92:4C as /devices/virtual/input/input20
chrissi@TOSH1 ~ $

```

Um die Arbeit vom Kernel zu demonstrieren, kann man den Befehl einmal ausführen bevor die Hardware eingesteckt wurde,

```

[33423.800988] IPv6: ADDRCONF(NETDEV_CHANGE): wlp4s0: link becomes ready
[33494.695520] input: 88:69:C2:CD:92:4C as /devices/virtual/input/input20
[33744.871253] toshiba acpi: Unknown key e00
[34130.011877] input: 88:69:C2:CD:92:4C as /devices/virtual/input/input21
[34140.342806] Bluetooth: hci0: last event is not cmd complete (0x0f)
[35463.627158] usb 3-1: new full-speed USB device number 2 using xhci_hcd
[35463.825010] usb 3-1: New USB device found, idVendor=10c4, idProduct=ea60
[35463.825018] usb 3-1: New USB device strings: Mfr=1, Product=2, SerialNumber=3
[35463.825023] usb 3-1: Product: CP2102 USB to UART Bridge Controller
[35463.825028] usb 3-1: Manufacturer: Silicon Labs
[35463.825032] usb 3-1: SerialNumber: 0001
[35464.897575] usbcore: registered new interface driver usbserial_generic
[35464.897584] usbserial: USB Serial support registered for generic
[35464.900039] usbcore: registered new interface driver cp210x
[35464.900102] usbserial: USB Serial support registered for cp210x
[35464.900155] cp210x 3-1:1.0: cp210x converter detected
[35464.913310] usb 3-1: cp210x converter now attached to ttyUSB0
chrissi@TOSH1 ~ $

```

und einmal, nachdem der Controller eingesteckt wurde. Der markierte Bereich zeigt alle Meldungen bezüglich dem zuletzt eingesteckten USB-Geräts. Zum Beispiel idVendor (= Hersteller-

ID) und idProduct, verwendeter Treiber (Product: CP2102 USB to UART Bridge Controller).

Für uns ist lediglich die letzte Zeile interessant. Hier steht "**cp210x converter now attached to ttyUSB0**", was bedeutet, dass unser Gerät, das tatsächlich mit dem cp210x-Treiber läuft, am Port **ttyUSB0** angeschlossen wurde.

Für Windows gibt es leider kein Terminal-Äquivalent zu dmesg. Jedoch kann man hier im Hardwaremanager (einfach in die Suche eingeben und mit Enter bestätigen) unter "USB-Controller" oder "Anschlüsse (COM & LTP)" bei eingestecktem Gerät sehen, unter welchem Port das Gerät zu erreichen ist. Unter Windows sind USB-Geräte an den sogenannten "COM-Port" gebunden.

freeRTOS und das MicroPython Framework herunterladen

Um das MicroPython Framework herunterzuladen gibt es zwei Möglichkeiten:

Entweder man nutzt das Terminal und lädt sich via Git (ein Tool zur Versionskontrolle <https://git-scm.com/book/de/v1/Los-geht%E2%80%99s-Git-Grundlagen>, <https://rogerdudler.github.io/git-guide/index.de.html>) eine Kopie davon auf deinen PC (Git erstellt ein Unterverzeichnis im aktuellen Arbeitsverzeichnis und kopiert die Inhalte in diesen):

```
$ git clone https://github.com/loboris/MicroPython_ESP32_psRAM_LoBo.git
```

Oder aber man besucht die Seite

https://github.com/loboris/MicroPython_ESP32_psRAM_LoBo

klickt auf den Button „clone or download“ und lädt sich manuell die Zip-Datei herunter.

Diese musst allerdings zuerst in ein geeignetes Verzeichnis entpackt werden.

freeRTOS installieren

Wichtig! Die folgenden Informationen beziehen sich gleichermaßen auf Linux wie auf MacOS.

Für Windows-Nutzer sind weitere, teils andere Schritte nötig, welche unter folgendem Link abrufbar sind:

https://github.com/loboris/MicroPython_ESP32_psRAM_LoBo/wiki/winsetup.

Um das Framework zu installieren hat man wieder zwei Möglichkeiten: Über das Terminal wird mittels `cd` in das zuvor heruntergeladene Verzeichnis "MicroPythonESP32_psRAM", dann ins Unterverzeichnis „MicroPython_BUILD“ gewechselt:

```
$ cd MicroPythonESP32_psRAM_LoBo/MicroPython_BUILD
```

Alternativ kann dieser Ordner auch im Filebrowser (Finder oder Dateiexplorer) geöffnet werden. Unter Linux und MacOS kann das aktuelle Verzeichnis mittels des Kontextmenüs, das man mit einem Klick auf die rechte Maustaste erreichen kann, im Terminal geöffnet werden. Klick auf die rechte Maustaste öffnet werden.

Anschließend geht es an die Kernel-Settings.

Dies hört sich nun aber dramatischer an als es tatsächlich ist!

Eigentlich ist in diesem Schritt nicht allzu viel zu erledigen, jedoch müssen wir kurz die `menuconfig` aufrufen, um eine Datei namens "sdkconfig" zu erstellen.

```
$ ./BUILD.sh menuconfig
```

Mit diesem Befehl öffnet sich die menuconfig, in der man nun verschiedene Einstellungen bezüglich Hardware (.z.B. Bluetooth aktivieren / deaktivieren), Betriebssystem (Nutzung des zweiten Prozessor-Kerns gestatten oder nicht) und dem genutzten Framework (wichtige Module einbinden, unwichtige ausschließen) vornehmen kann.

Hier navigiert man ausschließlich mit den Cursor-Tasten. Oben/Unten um die verschiedenen Menüpunkte auszuwählen, Rechts/Links um in der Menüliste unten zwischen **"select"** (=Menüpunkt auswählen / ändern), **"exit"** (=zurück zum übergeordneten Menüpunkt oder - falls keiner mehr darüber liegt - Programm beenden), **"help"** (Info zur Verwendung oder zum Wertebereich der Variablen einblenden), **"save"** (aktuelle Konfiguration (unter einem bestimmten Namen) abspeichern) oder **"load"** (=gespeicherte Konfigurationsdatei laden) auszuwählen. Sich hier etwas umzusehen schadet nicht, allerdings sollte man sich von den vielen Begriffen nicht allzusehr verwirren lassen, da nur einige wenige Änderungen wirklich notwendig sind.

Wenn also „select“ und der Menüpunkt "MicroPython" markiert sind und auf Enter gedrückt wird, gelangt man ins Untermenü von MicroPython. Wenn man nun wieder zurück - also eine Ebene höher - wechseln möchte, navigiert man mit den Pfeilen einmal nach rechts, sodass "exit" markiert wird und bestätigt mit Enter.

Wichtige Änderungen in der menuconfig:

USB Port prüfen:

Serial flasher config → default serial port

Hier sollte geprüft werden, ob der zuvor ermittelte USB Port (ttyUSB0) als default serial port gesetzt ist.

Webserver-Funktion aktivieren:

Micropython → SystemSettings → Enable Web server

Mit dem Cursor auf diesen Menüpunkt navigieren, auf „y“ für „yes“ drücken, um den Webserver zu aktivieren. Ein aktiviertes Modul wird mit einem "*" gekennzeichnet. „n“ für „no“ deaktiviert Module, die ggf. versehentlich aktiviert wurden oder einfach nicht gebraucht werden.

Telnet- und FTP-Server-Funktion mit "n" deaktivieren, falls diese aktiviert wurden:

Micropython → SystemSettings → Enable Telnet server
Micropython → SystemSettings → Enable Ftp server

Außerdem müssen wir dafür sorgen, dass wir den vollen Speicherbereich nutzen können:

MicroPython → System settings → MicroPython heap size (KB)

Hier sollte in der Klammer vor "MicroPython Heap Size" der Wert 96 stehen. Falls dies nicht der Fall ist, kann mit Enter bestätigt werden um diesen Wert anzupassen.

Das Webserver-Modul benötigt das Modul namens „**Websockets**“, das ebenso aktiviert werden sollte:

Micropython → Modules → Websockets

Also: Mit dem Cursor einmal nach rechts, sodass „Exit“ markiert wird, mit Enter bestätigen, einmal nach unten um auf Modules zu gelangen und ebenso mit Enter bestätigen. Hier einmal „**use Websockets**“ mit **y** auswählen, falls dieses noch nicht aktiviert war.

Zuletzt in der unteren Menüleiste mit Pfeil-Rechts Taste zu "**save**" wechseln und mit Enter bestätigen. Es erscheint noch eine Abfrage, unter welchem Dateinamen die sdkconfig gespeichert werden soll, hier bitte auch nur mit Enter bestätigen. Anschließend 3x **exit** wählen um die menuconfig zu beenden.

Falls hier etwas bei der Ausführung schief gehen sollte oder Unsicherheiten bestehen, kann die Datei „sdkconfig“ ebenso aus dem online-Repository zu diesem Projekt auf Github geladen werden. Den Link zum Repository gibt es allerdings erst im Kapitel "Prototypenbau". Falls die sdkconfig vom Repository genutzt wird, muss diese in der menuconfig zuerst mit "load" geladen werden, bevor die menuconfig wieder gespeichert und beendet werden kann

Jetzt geht's an die binary!

```
$ .BUILD.sh clean  
$ ./BUILD.sh
```

... eingeben ... warten ... erledigt!

Neugierig, was hier passiert?

Die Binary-File ist eine ausführbare Binärdatei, die mit einem Build-Prozess erstellt wird. Ein Build-Prozess ist der Vorgang in der Softwareentwicklung, bei dem eine Anwendung, bestehend aus Binärcode (binary) aus dem Quellcode erzeugt wird. Diesen Prozess nennt man auch "kompilieren". Kompiliert wird mit einem sogenannten Compiler.

Compiler

Compiler sind spezielle Übersetzer, die Programmcode aus problemorientierten Programmiersprachen (auch Hochsprachen genannt) in ausführbaren Maschinencode (also die Sprache, die die Prozessoren verstehen) übersetzen.

Mit einem Blick in die Ordnerstruktur des Micropython Frameworks findet man zum Beispiel im Ordner 'MicroPython_ESP32_psRAM_LoBoNeu/MicroPython_BUILD/components/micropython/esp32' einige Dateien mit der Endung ***.c** oder ***.h**. Dies sind Module, geschrieben in der Programmiersprache C - und die zugehörigen Header-Files.

Diese sind für uns Menschen noch einigermaßen lesbar. Der Prozessor, der dies allerdings ausführen muss, kann noch gar nichts damit anfangen. Dieser "versteht" nämlich nur den sogenannten Maschinencode, bestehend aus einer Folge von Bytes, die sowohl Befehle als auch Daten repräsentieren.

Da dieser Code für den Menschen kaum lesbar ist, werden in der Assemblersprache zum Beispiel (eine Sprache, die der Maschiensprache noch am nächsten kommt) die Befehle durch besser verständliche Abkürzungen, sogenannte Mnemonics, dargestellt.

Der Compiler scannt also während des kompiliervorgangs den Code und übersetzt ihn - so ressourcenschonend es geht - in Maschinensprache. Die dadurch entstandene binary lässt sich dann vom Computer ausführen.

Nachfolgend ein kleines Beispiel, wie es für den Prozessor aussehen kann, wenn man der Variablen "a" einen Wert zuweist oder wenn man zwei Werte, die in Variablen gespeichert sind, miteinander addiert. Wichtig zu wissen ist hierbei, dass die Variablen in sogenannten Registern gespeichert werden. Dies sind Speicherbereiche innerhalb des Prozessors, die direkt mit dem Rechenkern verbunden sind.

Für neugierige hier einige Links mit detaillierteren Informationen:

<https://de.wikipedia.org/wiki/Compiler>

<https://de.wikipedia.org/wiki/Maschinensprache>

[https://de.wikipedia.org/wiki/Register_\(Computer\)](https://de.wikipedia.org/wiki/Register_(Computer))

Maschinencode (Hexadezimal)	Assemblercode	C-Code	Erklärung
C7 45 FC 02	mov DWORD PTR [rbp-4], 2	int a = 2;	Setze Variable a, die durch Register RBP adressiert wird, auf den Wert 2.
8B 45 F8	mov eax, DWORD PTR [rbp-8]	int c = a + b;	Setze Register EAX auf den Wert von Variable b. Setze Register EDX auf den Wert von Variable a. Addiere den Wert von EDX zum Wert von EAX. Setze Variable c, die durch RBP adressiert wird, auf den Wert von EAX.
8B 55 FC	mov edx, DWORD PTR [rbp-4]		
01 D0	add eax, edx		
89 45 F4	mov DWORD PTR [rbp-12], eax		

Interpreter

Im Gegensatz zu einem Compiler, der vor Ausführung des Programms den vorhandenen Code in Maschinensprache übersetzt, macht dies ein sogenannter Interpreter erst während der Laufzeit.

Sprich es wird eine Zeile nach der anderen zuerst eingelesen und dann in Maschinensprache übersetzt. Nach diesem Schritt erst können die Befehle dem Prozessor zur Ausführung weitergegeben werden.

Ein Vorteil hiervon ist, dass diese Programme auf verschiedenen Systemen mit verschiedenen Prozessorarchitekturen ausführbar sind. Leider muss man aber - bedingt durch die Übersetzung während der Laufzeit, mit Performanceeinbußen rechnen. Das bedeutet, dass das Programm allgemein langsamer läuft als selbes Programm, das in einer kompilierten Sprache geschrieben wird.

Dies wird meist bei der Ausführung von rechenintensiven Anwendungen deutlich spürbar.

Programmiersprachen

In einer Programmiersprache werden alle aufeinanderfolgenden Arbeitsschritte beschrieben, die ein Computer ausführen soll.

https://wiki.ruby-portal.de/Scriptsprachen_vs.html

Frage: nötig, das auch noch zu erklären? falls nein kann man löschen

Framework "flashen"

Genug des Ausflugs in die Theorie. nun wird es Zeit, den ESP auszupacken und anzuschließen!

Also. Raus aus der Verpackung, ran ans USB-Kabel und im Computer einstecken!

Jedes USB-Gerät wird vom PC an einen speziellen Port geknüpft, mit dem es vom System angesprochen wird.

Unter Linux ist dies meist ttyUSB0 oder ttyUSB1.

Sobald der Controller eingesteckt ist, kannst du dies auch einfach prüfen:

```
$ ls dev/ttyUSB*
```

ls bedeutet list, sprich es listet nun alle angeschlossenen Geräte, die mit ttyUSB anfangen. Der * signalisiert, dass nach dieser Zeichenfolge noch andere Zeichen folgen

Bevor wir die Binary drauf spielen können, löschen wir vorsichtshalber alles:

```
$ esptool.py --port /dev/ttyUSB0 erase_flash
```

oder:

```
$ ./BUILD.sh erase
```

Und spielen dann die binary auf den Controller:

```
$ ./BUILD.sh flash
```

Bauteile verbinden

Neugierig? Erster Test??

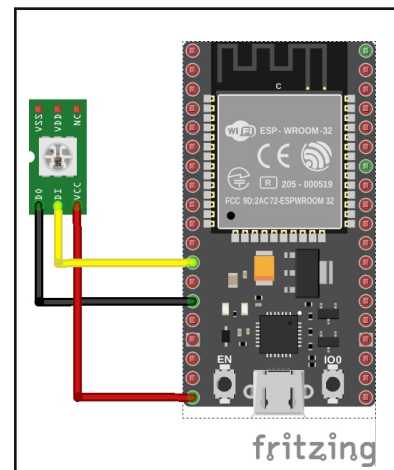
LED Strip an ESP32 anschließen

Also.

Du hast den LED-Strip, dieser hat an jedem Ende drei Anschlüsse, einen für 5V (rot), einen Data-Pin (meistens gelb) und einen für Ground, oder GND (schwarz oder weiß).

Schließe hier das rote und das schwarze (oder weiße) Kabel an eine externe Stromquelle (falls es nur wenige LED sind, kannst du auch den 5V Ausgang und GND vom ESP nutzen. Allerdings kann dies bei Überspannung deinen USB-Port am Rechner zumindest vorübergehend außer Gefecht setzen). Wichtig: um den Stromkreis zu schließen muss der GND der externen Stromquelle mit dem GND des ESP verbunden sein. Anschließend verbindest Du den Data-Pin des LED-Strips (DIN oder DI) mit Pin 14 am ESP.

Der Controller wird - falls noch nicht geschehen - per USB-Kabel mit dem Computer verbunden.



Eine Verbindung zum ESP herstellen - rshell und REPL

Hierfür verwenden wir das Tool "**rshell**" (= remote shell), das es uns ermöglicht, eine Verbindung zum Controller herzustellen um Dateien und Ordner hochzuladen, auszulesen, zu verschieben oder zu löschen, oder REPL zu starten.

Rshell benötigt einige Angaben, wie zum Beispiel die Puffergröße (--buffer-size=30) und den Port (-p /dev/ttyUSB0), wobei hier der vorher ermittelte Port (/dev/ttyUSB0) anzugeben ist.

Beendet werden kann rshell mit der Tastenkombination STRG+C

repl (= Read-eval-print loop) wiederum ist dazu da, um per Kommandozeile Befehle auf dem Board auszuführen, kleine Funktionen zu erstellen und diese, oder andere vorhandene Skripte auszuführen.

```
$ rshell --buffer-size=30 -p /dev/ttyUSB0 -a -e nano
$ repl
```

```
/bin/bash
chrissi@TOSHIBA ~ $ rshell --buffer-size=30 -p /dev/ttyUSB0 -a -e nano
Connecting to /dev/ttyUSB0 ...
Welcome to rshell. Use Control-D to exit.
/home/chrissi> repl
Entering REPL. Use Control-X to exit.
>
MicroPython ESP32 LoBo v3.2.21 - 2018-08-27 on ESP32 board with ESP32
Type "help()" for more information.
>>>
>>> █
```

Ab hier verändert sich die Darstellung der Zeilen!
Im Python-REPL werden die Zeilenanfänge mit den Zeichen ">>>" signalisiert. So weiß man sofort, dass man sich

nicht in der systemeigenen Shell, sondern in der Python-Shell befindet, hier also alle Befehle in Python geschrieben werden.
Beendet wird REPL mit der Tastenkombination STRG+X

Micropython

MicroPython ist - wie bereits erwähnt - eine etwas "abgespeckte" Version von Python (Version 3). Daher kann man sich auf die Beschreibungen, die sich auf Python beziehen, nur teilweise verlassen. Jedoch bieten sie gute Beschreibungen in Sachen Aufbau, Verwendung der einzelnen Module und Funktionen sowie der gängigen Programmier-Routinen, weshalb öfter auf Seiten verwiesen wird, die eigentlich Tutorials / Anleitungen für Python 3 bereitstellen:

Python4Kids, sowie deren Unterseite "Denken wie ein Informatiker" bietet sich speziell für jüngere Generationen an:

python4kids.net
<http://python4kids.net/how2think/index.htm>

Auch auf der deutschen Seite python-kurs.eu findet sich ein sehr schön beschriebenes und durch simple Grafiken unterstütztes Tutorial für Python 3:

https://www.python-kurs.eu/python3_kurs.php

Module

Modulare Programmierung ist eine Software-Design-Technik, die auf dem allgemeinen Prinzip des modularen Designs beruht. Unter modularem Design versteht man, dass man ein komplexes System in kleinere selbständige Einheiten oder Komponenten zerlegt. Diese Komponenten bezeichnet man üblicherweise als Module. Ein Modul kann unabhängig vom Gesamtsystem erzeugt und separat getestet werden. In den meisten Fällen kann man ein Modul auch in anderen Systemen verwenden.

https://www.python-kurs.eu/python3_modularisierung.php

Um ein Modul nutzen zu können muss es mit dem "import"-Befehl eingebunden werden. Mehr hierzu später.

Das machine-Modul

Um in Micropython unter anderem die Pins anzusteuern zu können benötigt man das Modul "machine". Eigentlich handelt es sich bei "machine" um ein Modul, das sogar weitere Module, beziehungsweise sogenannte "Klassen" beinhaltet.

<https://www.python-kurs.eu/klassen.php>

https://github.com/loboris/MicroPython_ESP32_psRAM_LoBo/wiki/machine

mit dem machine-Modul kann man zum Beispiel die Prozessorgeschwindigkeit in MHz auslesen oder einstellen (**machine.freq([speed in MHz])**) oder den Controller neu starten (**machine.reset()**).

Außerdem beinhaltet es das für uns wichtige Modul, bzw. die Klasse "Neopixel", mit dem wir den Strip konfigurieren und ansteuern, also zum leuchten bringen können.

Neopixel

Bevor wir den Strip leuchten lassen können, müssen wir also zuerst das Modul namens machine einbinden und damit den Pin definieren, an den wir den LED-Strip angeschlossen haben. Anschließend muss ein Objekt von "Neopixel" erzeugt werden. Im Fachjargon nennt man das "eine Instanz einer Klasse" erstellen. Aber da das Prinzip der Klassen für Anfänger oft schwer nachvollziehbar ist, lassen wir dieses Detail einmal aus. Gerne darf aber bei Interesse nachgelesen werden. Das deutschsprachige Python-Tutorial bietet dazu eine einfach geschriebene und detaillierte Erklärung: https://www.python-kurs.eu/python3_klassen.php.

mit dem "**import**" - Befehl sind wir in der Lage, Module einzubinden.

Auch wichtig für den Umgang mit der Neopixel-Library ist die Modulbeschreibung des Entwicklers:

https://github.com/loboris/MicroPython_ESP32_psRAM_LoBo/wiki/neopixel

machine.Pin([pin_number], [input / output]) definiert den Pin, den wir in [pin_number] angeben, sowie ob hier Signale empfangen (input) oder gesendet (output) werden. **machine.Pin.OUT** gibt zum Beispiel an, dass hier Signale vom Controller aus (an den LED-Strip) gesendet werden. Gib nun folgende Befehle nacheinander in REPL ein und bestätige sie jeweils mit Enter:


```
>>> import machine
>>> led_pin = machine.Pin(14, machine.Pin.OUT)
```

Um ein Objekt "Neopixel" zu erstellen, müssen ebenso einige Angaben gemacht werden. So muss der Pin angegeben werden, an dem der Strip angeschlossen ist, sowie die Anzahl der Pixel auf den Strip (in meinem Fall sind das 14 Pixel). Auch der verwendete LED-Typ sollte angegeben werden, sofern du auch den Strip vom Typ "SK6812". Wenn man dies auslässt ist als Standard-Typ TYPE_RGB angegeben, also LEDs mit jeweils drei Farben: rot, blau grün. Der SK6812 besteht sogenannten "RGBW" LED, sprich ein Pixel besteht aus vier Farben. Neben RGB auch weiß, so sind die LEDs auch einzeln dimmbar. So setzt sich dieser Schritt nun zusammen:

machine.Neopixel(pin, num_pixels[, type])

Den Pin haben wir ja bereits im letzten Schritt in einer Variablen namens "led_pin" gespeichert. Es bietet sich an, auch die Pixelanzahl (hier: 14) in einer Variablen zu speichern. Warum dies sinnvoll ist, wirst du in einem späteren Kapitel erfahren.

Gib nun folgende Befehle ein, um die Anzahl der Pixel in der Variablen "num_pixels" zu speichern und eine Instanz der Klasse Neopixel in der Variablen "strip" zu speichern:

```
>>> num_pixels = 14
>>> led_pin = machine.Pin(14, machine.Pin.OUT)
>>> strip = machine.Neopixel(led_pin, num_pixels,
machine.Neopixel.TYPE_RGBW)
```

Selbstverständlich kann man dies auch auf zwei Zeilen, bzw. zwei Befehle komprimieren, allerdings wirst du gleich sehen, dass dies vergleichsweise sehr unübersichtlich ist:

```
>>> strip = machine.Neopixel(machine.Pin(14, machine.Pin.OUT), 14,
machine.Neopixel.TYPE_RGBW)
```

LEDs leuchten lassen

np.set(pos, color [, white, num, update]) ist dafür zuständig, die Pixels leuchten zu lassen. Auch hier sind zwei Angaben zwingend erforderlich, alle anderen (in der eckigen Klammer) optional.

- **pos** bezeichnet den Index des Pixels, dessen Farbwert gesetzt werden soll.
- **color** gibt - wie der Name schon sagt - an, in welcher Farbe der Pixel leuchten soll. Hier kann man entweder vordefinierte Farben verwenden (wie im folgenden Beispiel), oder Farbwerte als Hexadezimalwert (wird später noch erklärt)
- **white** (optional) gibt - bei RGBW-Strips - an, wie hell die Farbe weiß in diesem Pixel leuchten soll. Wird hier nichts angegeben, wird als Standardwert 0 gesetzt. Hier sind Werte von 0-255 möglich. Je höher, desto mehr weiß wird in die Farbe gemischt.
- **num** (optional) definiert, wieviele Pixel in diesem Schritt gesetzt werden sollen, ausgehend vom Pixel, der in pos angegeben wurde. Standardwert ist hier 1, also wird standardgemäß nur der eine Pixel gesteuert, der in num angegeben wurde.
- **update** (optional) bestimmt, ob der oder die Pixel sofort in der jeweiligen Farbe angezeigt werden soll, bzw sollen (was auch Standardwert ist), oder ob mit der Anzeige gewartet werden soll. Falls abgewartet werden sollte, ist update=False anzugeben.

Um dann allerdings die Pixel in der vorher gesetzten Farbe leuchten zu lassen, ist zusätzlich noch der Befehl `strip.show()` nötig.

```
>>> strip.set(0, strip.NAVY)
```

lässt also den ersten Pixel des Strips in der Farbe "Navy" leuchten, während

```
>>> strip.set(0, strip.NAVY, num=num_pixels)
```

alle Pixel in der Farbe "Navy" leuchten lässt

Farben

Wie bereits erwähnt kann man die Farbe aus vordefinierten Farben oder als Hexadezimalwert angeben.

Um herauszufinden, welche Farben verfügbar sind, gibt es einen kleinen Trick:

Gib einfach "strip." in REPL ein und drücke einmal auf TAB.

So erscheint eine Auflistung aller vordefinierten Werte und sogar auch der vorhandenen Funktionen. Die vordefinierten Werte sind konstante Werte und werden in Großbuchstaben angezeigt. So sind folgende Farbwerte bereits definiert: **BLACK, WHITE, RED, LIME, BLUE, YELLOW, CYAN, MAGENTA, SILVER, GRAY, MAROON, OLIVE, GREEN, PURPLE, TEAL, NAVY.**

```

>>>
>>> strip.
__class__      BLACK      BLUE      CYAN
GRAY           GREEN     HSBtoRGB  HSBtoRGBint
LIME           MAGENTA   MAROON    NAVY
OLIVE          PURPLE    RED       RGBtoHSB
SILVER         TEAL      TYPE_RGB  TYPE_RGBW
WHITE          YELLOW    brightness
color_order    deinit   get       info
rainbow        set      setHSB    setHSBint
setWhite       show     timings
>>> strip.

```

Alternativ hat man die Möglichkeit, die Farbe anhand des Hexadezimalwerts anzugeben.

RGB-Farben werden definiert durch einen Wert für rot (=r), einen für grün (=g) und einen für blau (=b). Dieser Wert darf zwischen 0 und 255 liegen. Für die Farbe rot muss also für r=255, sprich höchster Wert, für g=0, und b=0 gesetzt werden. Anders geschrieben: (255, 0, 0). Dies ist der RGB-Wert als Tupel dargestellt.

Der Hexadezimalwert wäre für rot "#FF0000", bzw. 0xFF0000, wobei die Zeichen "#" oder "0x" nur vorangestellt werden, um zu zeigen, dass es sich hierbei um einen Hexadezimalwert handelt. Da "#FF0000" mit der '#' als initiales Zeichen, eine Zeichenkette darstellt, 0xff0000 hingegen einen (hexadezimalen) Zahlenwert und sich mit Zahlen besser rechnen lässt, verwenden wir hier die Darstellung 0xff0000.

Wandeln wir die Dezimalzahl "255" in eine Hexadezimalzahl erhalten wir "FF". Der Hexadezimalwert einer Farbe ist also nichts weiter, als die jeweiligen Farbwerte für r, g und b, einzeln konvertiert in eine Hexadezimalzahl und anschließend aneinandergereiht.

Die Farbe "aquamarin" hat zum Beispiel den RGB-Wert (127,255,212). Konvertieren wir die einzelnen Werte in Hexadezimalwerte erhalten wir 0x7FFFD4 (also 127 = 0x7F, 255 = 0xFF, 212 = 0xD4. Konvertieren wir wiederum den kompletten Hexadezimalwert in einen Dezimalwert erhalten wir 8388564. Beide Werte sind hier als Farbangabe zulässig.

Die Konvertierung von Dezimal- in Hexadezimalwerte übernimmt Python für uns. Hierfür gibt es Funktionen, die bestimmte Werte in Werte eines anderen Typs (wie zum Beispiel Dezimalzahl in Hexadezimalzahl: hex(decimal_value) und andersrum: int(hex_value) konvertiert. Dies kann gleich in REPL getestet werden:

```

>>> int(0xff0000)
16711680
>>> int(0xff)
255
>>> hex(128)
'0x80'
>>>

```

Möchten wir jedoch ein RGB-Tupel, also hier (127, 255, 212) als Angabe verwenden, müssen wir dieses erst in den entsprechenden Hexadezimalwert konvertieren.

Wollen wir die Farbe Schwarz, was im additiven Farbsystem, also dem Farbsystem unseres sichtbaren Lichts soviel wie "kein Licht" bedeutet, müssen wir einfach alle Farben auf 0 setzen. Dies geht indem man für color den Wert "0x00" angibt. also 0.

Datentypen in Python: Primitive Datentypen

Bisher haben wir nur ganz allgemein über Variable gesprochen. Variablen werden genutzt, um veränderbare Objekte zu speichern. Objekte können zum Beispiel ganz einfache Zahlenwerte oder etwa Zeichen(ketten) sein. Oder einfach nur den Wert 'True' oder 'False' annehmen:

Boolean ist der einfachste, auch kleinste Datentyp. Dieser kann lediglich zwei verschiedene Werte, nennen wir es lieber Zustände, annehmen: True oder False. Da dies ebenso mit 1 oder 0 ausgedrückt werden kann, genügt diesem Zustand ein Speicherplatz von genau einem Byte (was ebenso nur entweder auf 1 oder auf 0 gesetzt werden kann) Hierbei handelt es sich um einen logischen Datentyp.

Integer bezeichnen ganze Zahlenwerte. Diese können für eine 32-bit integer entweder nur positive Werte (unsigned integer) von 0 bis 4.294.967.295 annehmen, oder sowohl positive als auch negative Werte (signed integer) von $-2.147.483.648$ bis $2.147.483.647$

Float oder **double** bezeichnet Gleitkommawerte. Je nach Größe dieser Variablen (32 oder 64 Bit) kann sie Werte im Bereich $1,5 \cdot 10^{-45}$ bis $3,4 \cdot 10^{38}$ (float, 32 Bit) oder im Bereich $5,0 \cdot 10^{-324}$ bis $1,7 \cdot 10^{308}$ (double, 64 Bit) annehmen.

Char, oder character (= Zeichen) bezeichnet ein einzelnes Zeichen, das sowohl Buchstaben als auch Zahlen, sowie Sonderzeichen beinhalten kann. Codiert werden diese allerdings als Ganzzahl. Dies mag im ersten Moment seltsam klingen, allerdings nur, bis man sich anschaut, wie diese Codierung funktioniert. Dies wird durch eine ASCII-Tabelle klar, die man zum Beispiel bei Wikipedia sehen kann:

<https://upload.wikimedia.org/wikipedia/commons/2/26/Ascii-codes-table.png>

Wie man sehen kann gibt es für jeden Buchstaben, ob klein oder groß, jede Zahl und jedes Sonderzeichen auf der Tastatur einen zugehörigen numerischen Wert. Anhand diesem ist man sogar in der Lage, einfache Textverschlüsselungen zu bauen - wirklich sicher sind diese Verschlüsselungen allerdings nicht!

String bezeichnet eine Zeichenkette, also eine Aneinanderkettung von Zeichen - oder Sequenz von einzelnen Zeichen. Meist ist dies ein einfacher Text. Jedes einzelne Zeichen kann über seinen Index angesprochen werden. So ist es auch möglich, neben ganzen Worten nach dem Vorkommen von Buchstaben oder Buchstabensequenzen zu suchen, den String anhand dieser zu teilen oder aber einzelne Buchstaben(sequenzen) aus dem String zu entfernen.

Mit den Strings haben wir allerdings bereits das Feld der primitiven Datentypen verlassen.

Sequentielle Datentypen in Python

Ein sequentielle Datentypen bezeichnet man Datentypen, die eine Folge an gleichen oder verschiedenen Elementen beinhaltet.

List kann man als eine Art Stapelspeicher bezeichnen, auf den man weitere Elemente ablegen oder wieder herausnehmen kann. Diese wird gekennzeichnet durch eckige Klammern []. Man kann mittels dem Index auf die einzelnen Elemente der Liste zugreifen und diese ggf. auch verändern.

Tuple, oder auch Tupel sind einer Liste sehr ähnlich. Man kann ebenso mittels dem Index auf die einzelnen Elemente zugreifen. Jedoch werden Tupel in runde Klammern () gepackt und sind im Gegensatz zu Listen nicht veränderbar.

Dictionaries enthalten zusätzlich zu den Werten auch sogenannte Bezeichner, über den man das Objekt im Dictionary aufrufen kann:
`capital_dict = {"Baden-Wuerttemberg":"Stuttgart",
"Bayern":"Muenchen", ... }`

Ein Dictionary ist allerdings nicht mehr als sequentieller Datentyp einzuordnen. Hier kann man Schlüssel auf Objekte "abbilden", daher gehört ein Dictionary unter die Kategorie "Mapping".

Während in anderen Sprachen der Typ der Variable explizit definiert werden muss und auch nicht während der Laufzeit verändert werden kann, geschieht das in Python mit der Zuweisung. Zum Beispiel ist die Variable `a = 10` automatisch eine Integer-Zahl. Wenn man später aber `a` mit `a = 2.3` überschreibt, ist `a` vom typ float.

Man kann nicht nur einzelne Zahlenwerte oder Zeichen(ketten) in Variablen speichern. Man hat ebenso die Möglichkeit, eine Liste, ein Tupel oder einen Dictionary in eine Variable speichern.

Nicht nur das. Man kann ebenso eine Liste aus Listen anlegen. Oder aber ganze Funktionen in Variablen speichern!
Diese Eigenschaft machen wir uns in einem späteren Kapitel zunutze.

Mehr Informationen zu den bisher gezeigten Datentypen kann man auf folgenden Links abrufen:

https://www.python-kurs.eu/python3_variablen.php

https://www.python-kurs.eu/python3_sequentielle_datentypen.php

https://www.python-kurs.eu/python3_listen.php

https://www.python-kurs.eu/python3_dictionaries.php

Animationen Teil 1: Blinken

Bevor die LEDs allerdings blinken können, muss man sich Gedanken darüber machen, was hier eigentlich genau geschehen soll:

Die LEDs werden alle zur gleichen Zeit eingeschalten. Sie leuchten eine Weile, anschließend werden sie ausgeschalten. Nach einer Weile gehen sie wieder an, und so weiter.

Wenn die LEDs also zweimal blinken sollen, könnte ich einfach schreiben:

mach alle an – warte – mach alle aus – warte – mach alle an – warte –
mach alle aus.

Nur wäre diese Herangehensweise auf Dauer sehr umständlich. Wollten wir die LEDs auf diese Weise 1000x blinken lassen, hätten wir mittels copy + paste einiges zu tun! Außerdem würde das Programm viel zu groß und unübersichtlich werden!

Viel besser wäre es doch wenn wir sagen könnten:

mach bitte 5x:
mach alle an – warte – mach alle aus – warte.

Schleifen

Aus diesem Grund gibt es Schleifen.

Mit Schleifen kann man zum Beispiel festlegen, dass bestimmte Codezeilen genau 5, 10, 1000 Mal ausgeführt werden sollen (**for-Schleife**). Außerdem kann man auch sagen, dass ein Code-Block, wie bei uns das Blinken, nur ausgeführt werden soll, während es draußen dunkel ist (**while-Schleife**).

Zum Beispiel: während es draußen dunkel ist: blinke, checke ob's immernoch dunkel ist. wenn dunkel: nochmal das ganze! Wenn hell, abbrechen.

Auch könnte man mit einer while-Schleife sagen: mache das einfach immer wieder! Eine sogenannte Endlos-Schleife.

Grundlegend braucht eine Schleife für jeden Schleifendurchlauf eine Bedingung, deren Wahrheitsgehalt geprüft werden muss. Trifft die Bedingung zu, wird von dieser Bedingung der Boolesche Wert True zurück gegeben, was bedeutet, die Schleifenbedingung trifft zu und sie wird ein weiteres mal ausgeführt. Trifft die Bedingung nicht zu, wird der Wert False zurück gegeben. Die Schleifenbedingung trifft nicht zu und die Ausführung des Codes geht in der ersten Zeile nach der Schleife weiter.

For-Schleife

https://www.python-kurs.eu/python3_for-schleife.php

Um zu sagen: führe diesen Codeblock 5x aus schreibt man in Python:

```
>>> for i in range(5):
```

range(5) ist eine Funktion, die nacheinander alle ganzen Zahlen von 0 bis 4(!!!) ausgibt. Genauer sollte man eigentlich schreiben: range(0, 5).

Warum 4 und nicht 5? 5 gibt hier das obere Limit der Zahlenfolge, sowie die Anzahl der Zahlen an. Also bedeutet range(5) (bzw range(0,5)), dass eine Zahlenreihe, bestehend aus 5 ganzen Zahlen, beginnend bei 0, endend bei < 5 - also 4, zurückgegeben wird.

Bedeutet wiederum: dieser Codeblock wird 5x ausgeführt. Für jedes i von 0 bis < 5.

Aber - was befindet sich in diesem Codeblock - was außerhalb?

Dies wird in Python durch Einrückungen definiert.

Sprich alles, was hier 5x ausgeführt werden soll, muss nun im Vergleich Zeilenanfang der for-Schleife einmal mit Tab eingerückt sein.

range(5) ist eine Funktion, die nacheinander alle ganzen Zahlen von 0 bis 4(!!!) ausgibt. Genauer sollte man eigentlich schreiben: range(0, 5).

Warum 4 und nicht 5? 5 gibt hier das obere Limit der Zahlenfolge, sowie die Anzahl der Zahlen an. Also bedeutet range(5) (bzw range(0,5)), dass eine Zahlenreihe, bestehend aus 5 ganzen Zahlen, beginnend bei 0, endend bei < 5 - also 4, zurückgegeben wird.

Bedeutet wiederum: dieser Codeblock wird 5x ausgeführt.

Ein mal für jedes i von 0 bis < 5.

Aber - was befindet sich in diesem Codeblock, der nun fünf mal ausgeführt werden soll - und was außerhalb?

Dies wird in Python durch Einrückungen definiert.

```
>>>
>>>
>>> for i in range(5):
...     □
```

Sprich alles, was hier 5x ausgeführt werden soll, muss nun im Vergleich Zeilenanfang der for-Schleife einmal mit TAB eingerückt werden.

REPL erledigt das für uns, sobald man die Zeile "for i in range(5):" mit einem Doppelpunkt beendet und einmal Enter gedrückt hat. Der Cursor ist nun eingerückt, zudem hat sich wieder der Zeilenanfang verändert. Hier stehen nun "...":

Auch wenn wir mit Enter in die nächste Zeile wechseln, bleibt dieser noch auf selber Position.

Um die For-Schleife mit der Range-Funktion mal zu demonstrieren, sagen wir:

für jedes i von 0 bis <5 :
gib einmal den Wert von i aus.

Um Werte (oder Texte) im Terminal auszugeben nutzt man die Funktion `print([variable / Text (in Anführungszeichen)])`

Zweimaliges Drücken der Enter-Taste lässt den Cursor wieder an den Anfang der Zeile stellen, sprich, falls hier noch etwas gemacht werden sollte, dann wird das erst ausgeführt, sobald die Schleife verlassen, also 5x ausgeführt wurde.

```
>>>
>>> for i in range(5):
...     print(i)
...
...
...
...
...
... 
```

Da wir nach der Schleife nichts weiter ausführen wollen, können wir nochmals mit Enter bestätigen, sodass die Schleife ausgeführt werden

```
>>>
>>> for i in range(5):
...     print(i)
...
...
...
0
1
2
3
4
>>>
```

kann.

Dies ging nun allerdings sehr sehr schnell! Wenn wir in diesem Tempo unsere LEDs blinken lassen würden, würden wir das Blinken sehr wahrscheinlich überhaupt nicht mehr wirklich wahrnehmen!

Um diesen Prozess zu verlangsamen, müssen wir sogenannte Delays einbauen.

Dies erledigt man am einfachsten mit dem Modul "**utime**", bzw. dessen Funktion **sleep_ms()**

utime.sleep_ms(200) sorgt dafür, dass die nächste Codezeile (oder das nächste Mal, dass die Schleifenbedingung geprüft wird) erst 200 ms später stattfindet. Wenn nun folgende Zeilen ins Terminal eingegeben und - wie eben schonmal - 3x mit Enter bestätigt werden ...

```
>>> import utime
>>> for i in range(5):
...     print(i)
...     utime.sleep_ms(200)
```

... erscheinen nun die Ergebnisse der einzelnen Schleifendurchläufe in zeitlichen Abständen auf dem Terminal. Selbes Verhalten wollen wir für

unsere blinkenden LEDs!

While-Schleife

https://www.python-kurs.eu/python3_schleifen.php

Die einfachste, wahrscheinlich auch bekannteste While-Schleife ist die Endlosschleife. Einmal gestartet soll sie endlos den beinhalteten Codeblock ausführen. Hierzu sind gerade mal zwei Worte notwendig:

while True:

Wollen wir also unsere LEDs endlos blinken lassen müsste die Anweisungsfolge lauten:

 make für immer:
 schalte ein – warte – schalte aus – warte.

Dies würde dann folgendermaßen aussehen:

```
>>> import utime
>>> while True:
...     strip.set(0, strip.TEAL, num=num_pixels)
...     utime.sleep_ms(200)
...     strip.set(0, 0x00, num=num_pixels)
...     utime.sleep_ms(200)
```

Wir können allerdings ebenso mit einer while-Schleife ausdrücken, dass der Codeblock nur 5x ausgeführt werden soll.

Nehmen wir an, eine Variable *i* hat vor Schleifenaufruf den Wert 0. Dann kann man sagen 'while (i<5)', also während der Wert von *i* kleiner ist als 5, wird die Schleife ausgeführt. Dazu muss aber auch die Variable *i* innerhalb des nachfolgenden Codeblocks um 1 erhöht werden, ansonsten kann man sich auch so eine Endlosschleife generieren!

```
>>> while (i < 5):
...     i += 1
... 
```

Die Bedingung einer while-Schleife darf auch von Funktionen abhängen! Nehmen wir einmal an, wir haben bereits eine Funktion geschrieben, die prüft, ob es draußen dunkel ist. Wenn es dunkel ist, gibt sie den Wert "True" zurück, ansonsten "False". Nennen wir sie mal "is_dark()".

Schleifen funktionieren so, dass sie prüfen, ob die Bedingung nach dem Codewort "while" wahr ist. Wenn dem so ist, darf die Schleife ausgeführt, bzw. fortgeführt werden, ansonsten wird die Ausführung des Codes nach der Schleife fortgesetzt.

```
>>> while is_dark():
```

... wäre also genauso möglich.

Die Funktion "is_dark()" werden wir jetzt allerdings noch nicht implementieren.

Jedoch ist es an der Zeit, einen Blick auf Funktionen zu werfen.

Diese kann man nun hervorragend einsetzen, um etwas mit den Werten in blink() zu spielen und so ein Gefühl für die ideale Zeit in Millisekunden zu kriegen, die die LEDs leuchten (oder eben nicht leuchten).

Selbstverständlich kann man auch ohne Funktionen ein wenig mit den Werten beim Blink-Algorithmus spielen. Allerdings würde das sehr viel unnötige Schreiarbeit bedeuten und den Sinn (oder einer der Sinne) der Programmierung verfehlen:

Wiederholt vorhandene Routinen sollten stets zusammengefasst werden!

Operatoren und Ausdrücke

Unter einem Ausdruck versteht man in der Programmierung eine Kombination aus Variablen, Konstanten, Operatoren und Rückgabewerten von Funktionen. Die Auswertung eines Ausdrucks ergibt einen Wert, der meistens einer Variablen zugewiesen wird. In Python werden Ausdrücke unter Verwendung der gebräuchlichen mathematischen Notationen und Symbolen für Operatoren geschrieben.

Hier werden nur die für uns wichtigsten Operatoren genannt. Der arithmetische Operator ist in dieser Auflistung der einzige Operator, der (fast) immer einen numerischen Wert (int oder float) zurückgibt, während ein Logischer- oder ein Vergleichsoperator den Wahrheitsgehalt prüft und lediglich True oder False zurück gibt.

Arithmetische Operatoren

Operator	Aktion	Beispiel
+	Addition	$11 = b + c$
-	Subtraktion	$1 = c - b$
*	Multiplikation	$30 = c * b$
/	Division	$5 = c / b$
//	Truncation Division / Ganzzahldivision Das Ergebnis der Division ist der ganzzahlige Anteil der Division. Falls jedoch mindestens einer der Operanden ein Floatwert ist, so wird der ganzzahlige Anteil der Division als Float ausgegeben.	<pre>>>> 10 // 3 3 >>> >>> 10.0 // 3 3.0</pre>
%	Modulo - Division mit Rest (nur Ganzzahlen)	$1 = b \% c$ ($5 * 1 = 5$, $6 - 5 = \text{Rest } 1$)
=	Zuweisung von Werten	

Vergleichsoperatoren

>	Größer als	
<	Kleiner als	
>=	Größer gleich als	
<=	Kleiner gleich als	
==	Gleich	
!=	Nicht gleich, ungleich	
in	"Element von"	<code>1 in [3, 2, 1]</code>

Logische Operatoren

or	Boolsches Oder	(a or b) and c
and	Boolsches Und	
not	Boolsches Nicht	

Funktionen

Zu diesem Zweck gibt es Funktionen.

Bevor man eine Funktion implementieren möchte, muss man sich darüber im Klaren sein:

1. Welche Anweisungen in dieser Funktion gemacht werden - Funktionsname daher möglichst beschreibend wählen
2. ob oder welche Werte dazu nötig sind, die man ggf. verändern kann
3. ob die Funktion irgend etwas zurückgeben muss.
wie True/False bei `is_dark()`

Funktionen werden mit dem Schlüsselwort "**def**" (=define) eingeleitet. Der Aufbau sieht folgendermaßen aus:

```
def funktions_name(parameterliste):  
    """ funktion_name(parameterlist)  
  
    an example function to show structure of functions.  
  
    parameterlist: (type of parameter: Text or number) short description  
    what parameter(s) is / are needed  
    returns: (type of returned value) if a value is returned, describe  
it    shortly  
  
    """  
    Anweisung(en)  
    (return)
```

funktios_name: Wie in Punkt 1 beschrieben sollte der Funktionsname stets möglichst treffend beschreiben, was mit dieser Funktion bezweckt werden soll, bzw. was der Code für Auswirkungen hat.

Die **Parameterliste** kann aus beliebig vielen Parametern bestehen, die benötigt werden, um die in der Funktion stehenden Anweisungen

auszuführen.

Bei dem in grün dargestellten Text handelt es sich um einen "docstring". Diese werden genutzt, um Funktionen und Module zu beschreiben. Mehr dazu im Kapitel "Kommentare in Python"

Das **return** Statement ermöglicht es einem, Ergebnisse von Berechnungen, oder eben Werte eines ausgelesenen Sensors, wieder zurück zu geben.

Diesen Wert kann man also mit dem Funktionsaufruf in eine Variable speichern.

Weitere Infos findet man wie immer im Netz. Zum Beispiel auf:

<http://python4kids.net/how2think/kap05.htm>

https://www.python-kurs.eu/python3_funktionen.php

https://www.python-kurs.eu/python3_parameter.php

Wollten wir jetzt, als sehr einfaches Beispiel, eine Funktion schreiben, in der zwei Werte (meist Parameter genannt) miteinander addiert werden sollen, würden wir wie folgt vorgehen:

Zu 1. Wir wollen eine Funktion, in der zwei Zahlen miteinander addiert werden.

Nennen wir sie also einfach "addieren".

Zu 2. Zwei Werte sind dazu nötig. Der erste und der zweite Summand.

Zu 3. Die Summe der Addition soll zurückgegeben werden.

```
def addieren(a, b):  
    e = a + b  
    return e  
  
ergebnis = addieren(a, b)
```

Übertragen auf die Blink-Funktion würde es also folgendermaßen aussehen:

Zu 1. Wir wollen den LED-Strip blinken lassen. Daher nennen wir die Funktion

blink()

Zu 2. Wir möchten die Farbe und die Zeiten (an + aus) verändern.

Brauchen

dazu also drei Werte.

Zu 3. Da wir lediglich eine Ausgabe an einen Pin machen, brauchen wir keinen Rückgabewert

In der bisherigen Blink-Funktion wurden festen Werten für die Farbe und die Zeiten verwendet. Diese müssen nun durch Variablen ersetzt werden, deren Wert erst bei Aufruf der Funktion bestimmt, bzw übergeben wird.

Auch bei Variablen gilt stets darauf zu achten, treffende Namen zu wählen, um die Lesbarkeit zu gewährleisten und - ganz wichtig - um auch nach

einigen Wochen oder Monaten noch nachvollziehen zu können, was hier genau geschieht.

Für die Farbe bietet es sich daher an, die Variable "color" zu nennen.

Für die Zeit, die die LEDs leuchten sollen, "time_on"

Für die Zeit, die die LEDs nicht leuchten sollen, "time_off"

```
>>> def blink(color, time_on, time_off):  
...     strip.set(0, color, num=num_pixels  
...     utime.sleep_ms(time_on)  
...     strip.set(0, 0x00, num=num_pixels)  
...     utime.sleep_ms(time_off)
```

Nun ist es ganz einfach möglich, mit den Farbwerten und Blink-Intervallszeiten herumzuspielen.

Nachdem die Funktion in REPL eingetragen und fehlerfrei gespeichert wurde, kann man sie mit ...

```
>>> blink(strip.RED, 200, 10
```

... aufrufen.

Gerne kann anstatt des Standardwerts für color auch mit Farbwerten experimentiert werden. Im Netz findet sich unter dem englischen Suchbegriff "hex color picker" (= hexadezimaler Farbwähler) eine einfache Möglichkeit, die gewünschte Farbe als Hexadezimalwert zu erhalten. Keine Sorge, falls hier öfter Fehler passieren. Dies passiert immer wieder und soll nur beim Lernen unterstützen!

Funktionen mit optionalen Parametern

Sicherlich ist bereits aufgefallen, dass bei der Funktion strip.set(), im Falle, dass alle LEDs gleichzeitig geschaltet werden sollen, die Anzahl der zu schaltenden LEDs als Parameter "num= .." übergeben wird.

Hierbei handelt es sich um einen optionalen Parameter, auch Default-Parameter genannt.

Dies ist sehr praktisch, da wir hierfür einen Standardwert (Default-Wert) nutzen können, der bei Bedarf verändert werden kann.

Wenn im Vorfeld bereits ansprechende Zeit-Werte durch Versuche ermittelt wurden, kannst man diese ebenso als optionale Parameter angeben:

```
>>> def blink(color, time_on = 200, time_off = 100):  
...     strip.set(0, color, num=num_pixels  
...     utime.sleep_ms(time_on)  
...     strip.set(0, 0x00, num=num_pixels)  
...     utime.sleep_ms(time_off)
```

Zu beachten ist hier lediglich, dass die Default-Parameter grundsätzlich zuletzt genannt werden, da alle Variablen davor, hier lediglich "color", sogenannte **Positionsparameter** sind, sprich anhand ihrer Position identifiziert werden.

Die ursprüngliche Fassung der Blink-Funktion bestand nur aus Positionsparametern. color, time_on, time_off wurden beim Funktionsaufruf in dieser Reihenfolge eingegeben und auch so zugewiesen.

Da wir nun time_on und time_off als optionale Parameter definiert haben, dürfen diese genau aus diesem Grund nicht am Anfang stehen. Ansonsten würde die Gefahr bestehen, dass die Werte falsch zugeordnet werden!

Würden wir nun alle drei Werte nennen, müssten wir die Schlüsselwörter (time_on / time_off) nicht zwingend mit angeben.

Falls wir aber nur den Wert für time_off ändern wollten, den für time_on aber auf default lassen würden, können wir time_on also einfach auslassen und die Funktion mit ...

```
>>> blink(0x3F03F, time_off=50)
```

... aufrufen.

Aufgabe: Blinken in zwei Farben

Schaffst du es auch, den Strip anstatt ein- und auszuschalten zwischen zwei Farben wechseln zu lassen?

Oder abwechselnd in zwei Farben blinken zu lassen?

Ein erstes kleines Skript

REPL ist zwar durchaus ein nützliches Tool, jedoch kann es sehr mühsam werden, da man vieles einach öfter tippen muss als dies notwendig wäre.

Da unsere Funktionen nun nach und nach immer komplexer werden und es nun vorteilhaft ist, Codepassagen einfach überarbeiten zu können, steigen wir nun auf einen Texteditor um.

Editor

Hierzu würde für den Anfang ein normalen Texteditor nutzen, der standardgemäß auf jedem Betriebssystem vorinstalliert ist. Allerdings ist es vorteilhaft, einen Editor zu nutzen, der auch fürs coden geacht wurde. Sehr beliebt ist zum Beispiel Atom.io, der kostenlos für alle Systeme verfügbar ist. Dieser kann auf

<https://atom.io/>

heruntergeladen werden.

Falls noch keinen Ablageort für Skripte oder ganze Projekte vorhanden sein sollte, empfiehlt es sich, gleich einen anzulegen.

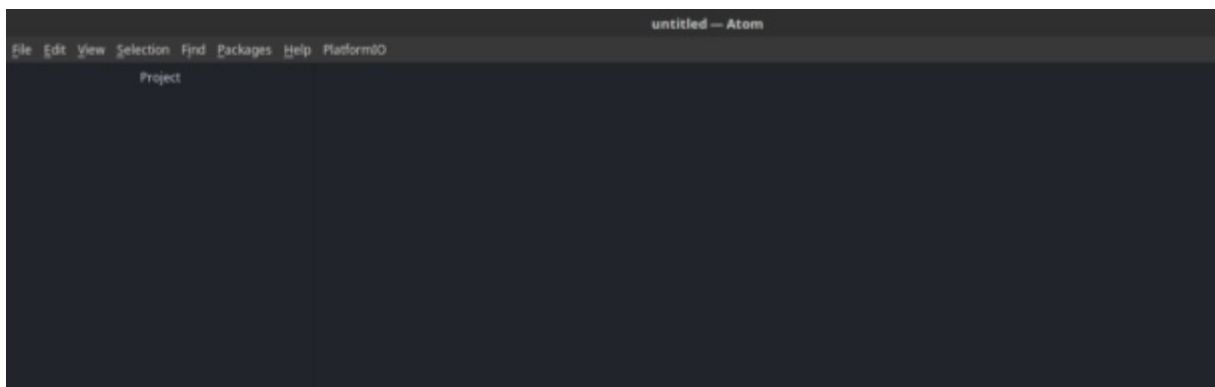
Ich habe zum Beispiel im meinem Home-Verzeichnis ein Verzeichnis namens "coding" in dem ich meine Skripte (als einzelne Files) und Projekte (in Ordnern) - sortiert nach Programmiersprache - ablege.

Atom besteht im Wesentlichen aus zwei Bereichen:

Der linke Bereich, "Project" genannt, ist der Bereich für die Projektordner. Der rechten Teil bietet Platz für den eigentlichen Editor, wobei man hier auch mehrere Dateien parallel öffnen kann. Diese sind dann sowohl auf eigenen Reitern als auch nebeneinander, in mehreren Fenstern, anzusehen.

Um einen Projektordner zu erstellen, bzw. auszuwählen, klickst du einfach mit der rechten Maustaste in das "Project"-Feld und gehst auf "Add Project Folder".

Falls noch kein eigener Ordner für dieses Projekt besteht, sollte nun einer erstellt und in Atom als Projektordner eingefügt werden. Der Name ist frei wählbar, sollte jedoch ebenso beschreibend für das gesamte Projekt sein.



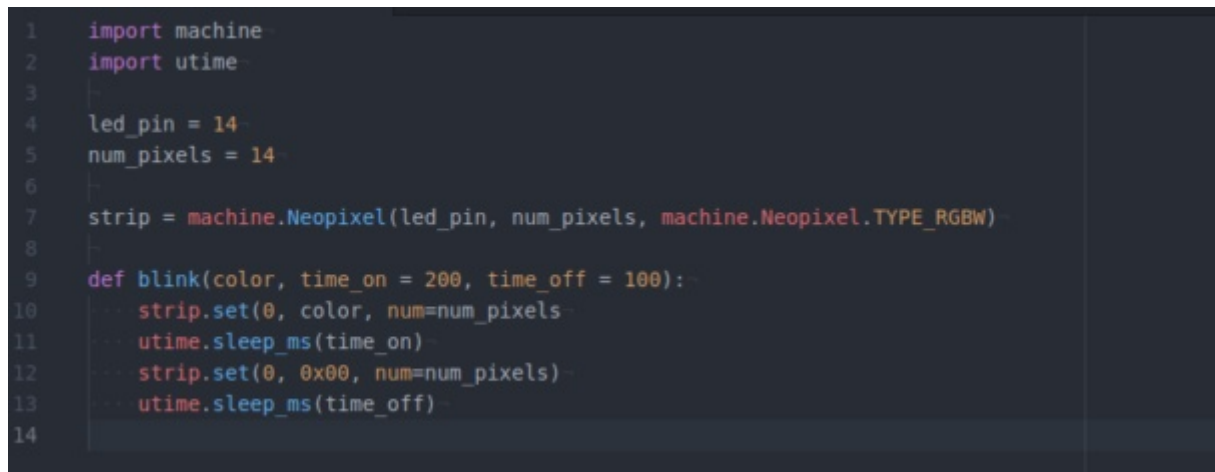
Möchte man innerhalb des neu erstellten Projektordners eine neue Datei anlegen, kann man dies, indem man mit der rechten Maustaste das Kontextmenü des Projektordners öffnet und "new File" wählt. Alternativ kann man bei markiertem Projektordner die Taste A drücken.

In beiden Fällen öffnet sich ein Eingabefenster, in dem der Name der Datei - inklusive Dateierweiterung (animations.py) eingegeben werden muss.

Im rechten Bereich befindet sich dann der eigentliche Texteditor, hier wird der Inhalt der geöffneten (Projekt-)Dateien (in allen gängigen Sprachen) angezeigt.

Außerdem wird hier nun, nach und nach, beginnend mit dem import-Statement, alles bisher gemachte übertragen.

Sehr schnell erkennt man hier die Vorteile:



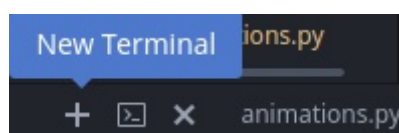
```
1  import machine
2  import utime
3
4  led_pin = 14
5  num_pixels = 14
6
7  strip = machine.Neopixel(led_pin, num_pixels, machine.Neopixel.TYPE_RGBW)
8
9  def blink(color, time_on = 200, time_off = 100):
10     strip.set(0, color, num=num_pixels)
11     utime.sleep_ms(time_on)
12     strip.set(0, 0x00, num=num_pixels)
13     utime.sleep_ms(time_off)
14
```

Durch farbige Hervorhebungen wirkt der Text auf Anhieb viel strukturierter.

So werden zum Beispiel alle Schlüsselwörter (import, def, ...) lila gekennzeichnet. Alle Funktionen, Konstruktoren sind cyan, sowie Werte orange gekennzeichnet sind.

Auch die Zeilennummern am linken Rand werden noch von Vorteil sein, wenn es zu den ersten Syntaxfehlern (oder manchmal einfach Tippfehlern) kommt.

Nachdem nun alles in den Editor übertragen und gespeichert wurde, müssen wir aber doch wieder das Terminal nutzen.



Atom.io beinhaltet ebenso ein Terminal, das praktischerweise beim Start sofort den Projektordner als Arbeitsverzeichnis öffnet. Diesen erreicht man über das kleine '+' - Zeichen am linken unteren Rand.

Von hier aus können wir direkt via rshell eine Verbindung zum ESP herstellen:

```
$ rshell --buffer-size=30 -p /dev/ttyUSB0 -a -e nano
```

bevor wir aber REPL starten, kopieren wir das erstellte Skript auf den ESP:

```
$ cp animations.py /flash
```

Anschließend müssen wir den ESP neu starten. Hierzu einfach den Reset-Button (rst) auf dem Controller drücken.

Nun können wir den REPL-Mode starten. Also wieder einfach "repl" im Terminal eingeben und mit Enter bestätigen.

Um nun die Funktion blink zu starten, müssen wir das Skript zuerst importieren.

```
>>> import animations
```

Wie wie unschwer zu verkennen, funktioniert dies auf die selbe Weise wie auch vorhandene Module eingefügt werden. Dies bedeutet, dass mit diesen wenigen Codezeilen bereits ein Mini-Modul erstellt wurde! Es besteht zwar momentan nur aus einer Funktion, aber die wird nicht sehr lange alleine bleiben!

Oooops! Schon hat sich ein kleiner 'Fehlerteufel' eingeschlichen. Wer findet den Fehler?

```
>>> import animationen
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "animationen.py", line 11
SyntaxError: invalid syntax
>>> □
```

Aber so wird gleich klar, wozu die Zeilenanzeige im Editor gut ist! Bei dieser Fehlermeldung sind für uns erst einmal die letzten beiden Zeilen relevant. In der letzten steht die Art des Fehlers, der gefunden wurde - also ein Syntax-Fehler. Die Zeile darüber sagt mir, welche Datei betroffen ist und in welcher Zeile es zu diesem Fehler gekommen ist - also Zeile 11.

utime.sleep_ms(time_on) in Zeile 11 sieht allerdings richtig aus. Kein Syntax-Fehler zu finden! In diesem Fall sollte man sich auch die darüberliegende Zeile genauer ansehen.

strip.set(0, color, num=num_pixels in Zeile 10 - hier stimmt definitiv etwas nicht! Eine Klammer wurde geöffnet, aber nicht wieder geschlossen! So hat der Interpreter verstanden, dass die darauf folgende Zeile - Zeile 11 - also noch Bestandteil dessen ist, was in die Klammer gehört. Da dies aber so keinen Sinn macht, gab es nen Syntax-Fehler. Und da dieser erst in Zeile 11 als Fehler registriert wurde, wurde auch Zeile 11 angegeben!

Oft kommt es vor, dass selbst dies nicht hilft, um die Fehlerquelle zu finden. Dann lohnt sich ein Blick auf die Zeilen der Terminalausgabe, die sich über der letzten beiden befinden. Hier wird dann der Backtrace aufgelistet. Sprich jede Funktion, jeder übergebene Parameter, der mit

diesem Fehler zusammenhängen könnte, wird gelistet. So kann man Schritt für Schritt verfolgen, was passiert ist und somit ermitteln, was zu diesem Fehler geführt hat. Debugging nennt man diesen Prozess, und die hier gezeigte Methode ist relativ simpel - allerdings auch nur geeignet, um kleine, überschaubare Programme zu debuggen.

Ein kleiner Scherz am Rande: Debugging is like being the detective in a crime movie where you are also the murderer.

Es kann sehr nervenaufreibend sein, und jede(r) kann immer wieder an den Punkt kommen, an dem sie / er alles in Zweifel stellt. Dies ist also relativ normal! Und: Es geht vorbei! Irgendwann findet man den - oft zu banalen - Fehler und prompt geht es weiter!

Also. Im Editor Klammer einfügen. Skript speichern. Im Terminal mit STRG +X REPL beenden. Skript erneut kopieren und REPL wieder starten. ESP neu starten.

Tipp für "faule": Man kann diese Befehle auch auf eine Zeile komprimieren! In Linux ist die Systemsprache C. Sprich Linux wurde komplett in C geschrieben. In fast allen Programmiersprachen (außer Python) werden die Zeilenenden mit einem ";" versehen, um dem Compiler oder dem Interpreter zu signalisieren, dass der Befehl hier nun zuende ist. Selbes kann man sich auch im Terminal zunutze machen, indem der Befehl zum kopieren und der zum REPL starten auf eine Zeile - getrennt durch einen ';' - geschrieben wird:

```
$ cp animations.py; repl
```

Auch wenn in Python keine Semikolons nötig sind um die Zeile (oder den Befehl) zu beenden, funktioniert selbes auch im REPL-Mode! Da wir wieder den Controller neu starten müssen, können wir - anstatt den Button zu drücken - hier einfach eingeben:

```
>>> import machine; machine.reset()
```

Nun sollte das Importieren problemlos funktionieren. Übrigens: Wenn der ESP nicht gerade außer Betrieb (= ohne Strom) war, muss man eventuell das machine-Modul nicht erneut einbinden.

Dies lässt sich testen, indem man "ma" in REPL eingibt und anschließend die TAB-Taste drückt. Sollte das Modul bereits verfügbar sein, wird die Autovervollständigung den Rest des Wortes "machine" einfügen.

Skript auf dem ESP ausführen

Eine Funktion in einem Modul wird aufgerufen, indem man zuerst den Modulnamen schreibt, anschließend, durch einen Punkt getrennt, der Funktionsnamen inkl. Parameter in der Klammer:

```
>>> import animations  
>>> animations.blink(0x00ff00)
```

Nun haben wir zwar eine Blink-Funktion, diese lässt die LED allerdings wieder nur 1x blinken!

Aufgabe: Schleifen erstellen

Erstelle nun anhand der vorherigen Beispiele eine Blink-Funktion, die

1. 10 x blinkt (einmal mit for- und einmal mit while-Schleife)
2. für immer blinken wird.
3. (Special) erweitere die Funktion (Parameterliste und Funktionsrumpf) so, dass beim Funktionsaufruf definiert werden kann, wie oft die Schleife durchlaufen werden soll.

Dateisystem auf dem ESP

Schauen wir uns nun mal die vorhandenen Dateien auf dem ESP an. Dazu müssen wir zuerst REPL mit STRG+X wieder verlassen um anschließend mit **ls /flash** den Inhalt des flash-Speichers auf dem ESP32 ausgeben zu lassen:

```
>>> ls /flash
```

gibt uns nun den aktuellen Inhalt des Flash-Speichers auf dem ESP aus.

Neben unserer "animations.py" befindet sich nur eine Datei auf dem Controller: "boot.py".

Dies ist die erste Datei, die beim Boot-Vorgang (oder einfach beim Starten) des ESP ausgeführt wird. Hier sollte lediglich Code stehen, der finale Einstellungen vornimmt, um den Boot-Prozess abzuschließen. Sehr viel passiert hier nicht - und wir sollten die Datei auch zum jetzigen Zeitpunkt nicht verändern, jedoch darf gerne mal einen Blick darauf geworfen werden. Damit die Datei auch im Projektordner landet und nicht unter Umständen irgendwo im lokalen Dateisystem "verloren" geht, muss hier als zweites Argument (Ziel) der vollständige Pfad zum Projektordner eingegeben werden:

```
>>> cp /flash/boot.py [/path/to/your/projectDirectory]/[projectDir]
>>> cp /flash/boot.py /home/chrisi/coding/microPy/rainbowWarrior
```

kopiert die Datei vom Controller in dein lokales Filesystem, sodass diese mit jedem beliebigen Texteditor gelesen werden kann.

Hier wird das Modul "**sys**" (Abkürzung für System) eingebunden, und der Liste sys.path (eine Variable des Moduls sys) ein Pfad zu einem Ordner namens 'lib' auf dem Flash-Speicher als Text zugewiesen.

```
This file is executed on every boot (including wake boot from deepsleep)
import sys
sys.path[1] = '/flash/lib'
```

Um herauszufinden, was hier genau geschieht, lohnt es sich, einen Blick in die Dokumentation der Micropython-Implementierung zu werfen:

<https://docs.micropython.org/en/latest/pyboard/library/sys.html>

sys.path: A mutable list of directories to search for imported modules.

Bedeutet, dass hier eine veränderbare Liste an Ordnern in der Variablen gespeichert wird, in denen nach eingefügten Modulen gesucht wird. Sprich falls wir je weitere (externe) Module benötigen, die nicht in der Micropython-Implementierung vorhanden sind, müssen diese im Ordner "flash/lib" gespeichert werden, andererseits werden sie wahrscheinlich nicht gefunden.

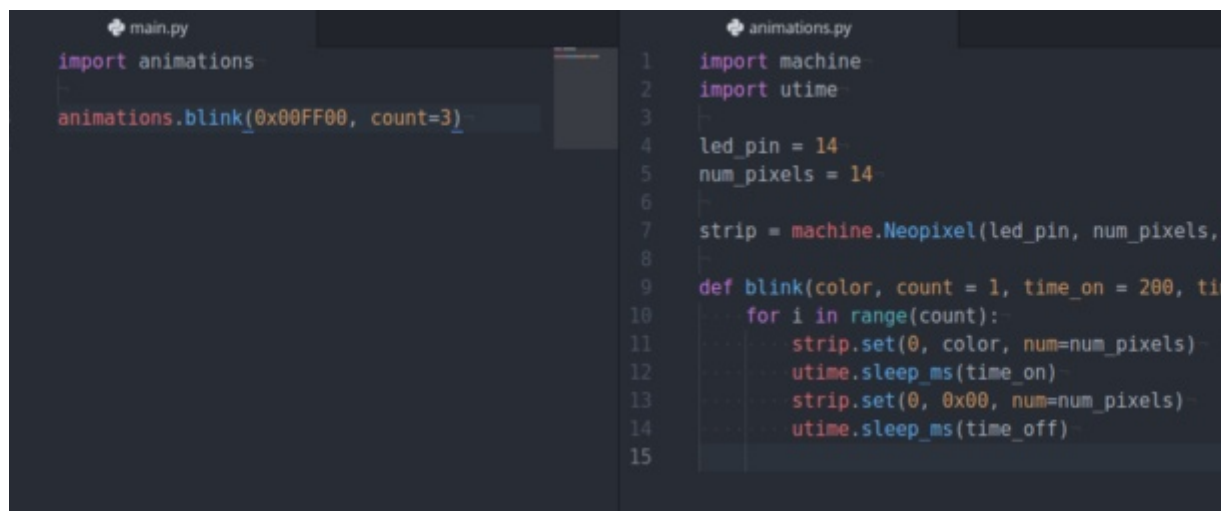
Nachdem "**boot.py**" ausgeführt wurde, wird standardgemäß nach einer File namens "**main.py**" gesucht. Falls diese vorhanden ist, wird sie ausgeführt.

Nur, was macht main.py?

Dies kommt sehr darauf an, wofür der ESP eingesetzt werden soll! So können hier weitere Module eingebunden und ausgeführt werden, wie unsere Blink-Funktion.

Daher können wir nun alles in main.py übertragen, was wir nach dem letzten Neustart in REPL eingegeben haben, sodass die Funktion "blink" automatisch nach dem Boot-Prozess startet.

Im Editor erstellen wir also eine neue Datei namens "main.py", die im Projektordner gespeichert wird und fügen die zuletzt in REPL ausgeführten



```
main.py
import animations
animations.blink(0xFF0000, count=3)

animations.py
1 import machine
2 import utime
3
4 led_pin = 14
5 num_pixels = 14
6
7 strip = machine.Neopixel(led_pin, num_pixels,
8
9 def blink(color, count = 1, time_on = 200, ti
10     for i in range(count):
11         strip.set(0, color, num=num_pixels)
12         utime.sleep_ms(time_on)
13         strip.set(0, 0x000, num=num_pixels)
14         utime.sleep_ms(time_off)
15
```

Zeilen ein.

In Atom können nicht nur mehrere Dateien gleichzeitig geöffnet sein (Zugriff über die Reiter), es gibt auch einen sogenannten Split-Mode, mit dem wir zwei Dateien nebeneinander setzen können. Dazu klicken wir auf den Reiter "animations.py", halten die Taste gedrückt und ziehen sie in den rechten Bereich des Fensters. So kann man beide Dateien nebeneinander betrachten, was gerade beim Schreiben von Funktionsaufrufen praktisch sein kann.

In der Zwischenzeit wurden die gestellten Aufgaben sicherlich bereits erledigt? Für den weiteren Verlauf des Projekts ist es gut, wenn wir der Blink-Funktion mitteilen können, wie oft sie blinken kann. Dazu ist ein weiterer optionaler Parameter notwendig.

Vorsicht bei der Namensgebung für diesen Parameter! Hier würde sich "range" als treffender Name anbieten. Jedoch kann dies zu Konflikten führen. Ebenso darf man keine Schlüsselwörter als Variablennamen einsetzen. Daher nennen wir diese Variable lieber "count"

Animationen Teil 2: running dots

Eine Möglichkeit, mehr Bewegung in die Animation einzubringen, ist es, wenn zum Beispiel nur eine LED zur gleichen Zeit - jedoch nach und nach um eine Position verschoben - leuchtet.

Nur - wie erreichen wir das?

```
schalte LED an Pos 0 ein - warte - schalte LED an Pos 0 aus
schalte LED an Pos 1 ein - warte - schalte LED an Pos 1 aus
.....
schalte LED an letzter Pos ein - warte - schalte LED an letzter Pos aus
```

Hierfür wird also wieder eine for-schleife benötigt, wobei die obere Grenze der range-Funktion gleich der Anzahl der LED (also num_led) ist. Anders geschrieben lautet dieser Pseudocode also:

```
für jedes LED in der range (0, num_led):
schalte LED ein - warte - schalte LED aus
```

Diese Funktion wird einfach in animations.py eingefügt

```
def running_dot(color = 0x00FFFF, time_on=50):
    while True:
        for i in range(num_pixels):
            strip.set(i +1, color)
            utime.sleep_ms(time_on)
            strip.set(i +1, 0x00)
            utime.sleep_ms(1)
```

Nun soll wieder erreicht werden, dass die Farbe und die Zeit, die die LED leuchten sollen, bestimmt werden können, daher werden color und time_on als optionale Parameter angegeben. color = 0x00FFFF ergibt die Farbe cyan, mit time_on = 50 ms läuft die Animation relativ zügig durch. Hier ist am Ende der Schleife ein kurzes Delay von min 1ms notwendig, da es sonst zu fehlerhaften Zuweisungen auf dem Strip kommt. Würden wir dieses Delay auslassen, würden die LEDs in allen möglichen Farben kreuz und quer leuchten!

Hier muss ich auf eine kleine "Besonderheit" in dieser MicroPython Implementierung hinweisen, die leider nicht den gängigen Programmierkonventionen entspricht!

Normalerweise fängt der Informatiker bei 0 an zu zählen. Von 0 bis 9 hat man genau 10 Ziffern. Und zwar genau die 10 Ziffern, die notwendig sind, um alle Zahlen im 10er Zahlensystem darzustellen.

Dies bedeutet, dass - normalerweise - das Element, das sich an erster Stelle befindet, mit dem Index 0 angesprochen wird. Ebenso wird das Element, das sich an Stelle n befindet, mit dem Index n-1 angesprochen.

Auf den Strip übertragen würde dies bedeuten, die LED an 1. Stelle würde mit pos = 0 angesprochen werden, die LED an der letzten Stelle mit pos =

(num_pixels -1)

Dies ist hier aber nicht der Fall! Die erste LED kann man sowohl mit pos = 0 als auch mit pos = 1 ansprechen. Dagegen wird mit pos = num_pixels-1 die vorletzte LED leuchten, nicht etwa die letzte. Die letzte LED wird mit num_pixels angesprochen.

Daher müssen wir in strip.set() als Index grundsätzlich i+1 übergeben!

Gestartet wird die Animation wieder in "main.py". Da alle Parameter optional sind, muss man auch beim Funktionsaufruf nicht zwingend Parameter übergeben!

```
animations.running_dots()
```

... führt die Animation mit Default-Werten aus.

Kommentare in (Micro)Python

Nun haben wir eine Situation, in der wir ein Stück Code geschrieben haben, das wir zeitweise "ausblenden" wollen. Eine Möglichkeit wäre, diese einfach zu entfernen. Sobald es aber um mehr als nur eine Zeile geht, läuft man in Gefahr, dass dieser Codeblock in der Zwischenablage "vergessen" und somit schnell gelöscht oder überschrieben werden kann. Eine andere Möglichkeit bieten Kommentare.

Üblicherweise werden Kommentare eingesetzt um zu beschreiben, was im nachfolgenden Codeblock geschieht. Aber man kann diese während der Entwicklung ebenso einsetzen, um Codezeilen vorübergehend vom Interpreter 'überlesen' zu lassen.

Aus Performancegründen sollten diese Art Kommentare allerdings gelöscht werden, sobald sie nicht mehr gebraucht werden. Spätestens aber wenn das Programm 'fertig' ist.

Man unterscheidet zwischen zwei verschiedenen Arten von Kommentaren: dem Zeilenkommentar und dem Blockkommentar.

Zeilenkommentare

Zeilenkommentare gehen - wie der Name schon sagt - nur über eine Zeile und werden gekennzeichnet durch eine # zu Beginn der Zeile / des Kommentars.

Man kann Zeilenkommentare auf die komplette Zeile anwenden, oder eben nur auf einen Teil der Zeile. Alles, was sich hinter der # befindet, wird also vom Interpreter übersprungen und nicht ausgeführt.

```
# Dies ist ein Zeilenkommentar.  
  
i = x % 5 # hier kann man kurz beschreiben, was passiert
```

Zeilenkommentare werden nicht nur genutzt, um Codezeilen vom Interpreter "überlesen" zu lassen, sie werden hauptsächlich eingesetzt, um Codefragmente, auch Routinen genannt, zu beschreiben. Dies dient dazu, den Code lesbarer zu machen. Nicht nur für andere, auch für einen selbst. Nach einiger Zeit weiß man sonst oft nicht mehr, was im entsprechenden Teil geschieht.

Blockkommentare / Docstrings

Blockkommentare können über mehrere Zeilen gehen. Dazu müssen zu Beginn und am Ende des Blockkommentars drei Anführungszeichen in Folge stehen.

```
""" Dies ist ein Blockkommentar.  
Dieser darf auch mehrzeilig sein und wird auch häufig eingesetzt, um  
Module, bzw. Funktionen zu beschreiben.  
"""
```

Docstrings - oder auch Documentation-Strings werden meist für die Dokumentation genutzt. Hier wird möglichst prägnant beschrieben, was in der Funktion geschieht. Diese Beschreibung sollte so ausfallen, dass andere, die den Code dazu nicht kennen, diesen nachbauen können.

Es gibt zusätzliche Tools, die aus diesen Docstrings eine Dokumentation des Projekts erstellen, die im besten Fall sogar von Menschen verstanden werden, die nicht programmieren können. Außerdem kann man in der Python Shell via `my_function.__doc__` oder mit `print(my_function.__doc__)` diese Beschreibung anzeigen lassen

Damit aus den Docstrings eine Dokumentation erstellt werden kann, müssen sie ein gewisses Format haben:

Zuerst kommt eine kurze Beschreibung des Moduls / der Funktion. Anschließend sollten die zu übergebenden Parameter und ggf. Return-Value sowie deren Datentyp beschrieben werden.

```
""" running_dot(color = 0x00FFFF, time_on = 50)

Neopixel-Animation, every single LED will light up in the color specified
by color for a time, specified in time_on. after time has passed, LED is
turned off and next LED will light up.

color: color (hex) of the LEDs
time_on: time (ms) the LED will light up
"""
```

running dots mit Schweif - white level

Wollen wir nun erreichen, dass nicht nur eine, sondern 3 aneinandergereihte LEDs leuchten, diese aber mit jeder LED an Farbintensität abnimmt, können wir die vierte Farbe der SK2812 LEDs nutzen: Weiß! Diesen Farbwert steuert man in `strip.set()` über den Parameter 'white'. Dieser kann Werte zwischen 0 und 255 annehmen.

Die zweite LED, die also etwas weniger farbintensiv sein soll, können wir ganz einfach in der Schleife mit `pos=i-1` ansprechen.

```
strip.set(i-1, color, white=100)
```

... würde also die LED ansprechen, die sich direkt hinter der aktuell leuchtenden befindet.

```
strip.set(i-2, color, white = 150)
```

... würde dann die LED ansprechen, die sich zwei Positionen dahinter befindet!

Aber: im Fall `i=1` wäre `i-2 = -1`. Ein negativer Index kann aber zu einer Fehlermeldung führen (Index Error - List Index out of range).

Daher müssen wir garantieren, dass der Wert für `pos` in `strip.set` nie kleiner als 1 und auch nie größer als `num_pixels` ist.

Um zu garantieren, dass der Wert für `i-1`, `i-2` nie kleiner als 0 und nie größer oder gleich als `num_pixels` ist, können wir den Modulo-Operator, also Rest-Division einsetzen.

```
pos1 = (i - 1) % num_pixels  
pos2 = (i - 2) % num_pixels
```

Da bei der Modulodivision - genau wie bei der normalen Division - die Regel "Punkt vor Strich" gilt, muss der Ausdruck `(i - 1)` zwingend in Klammern.

Somit haben wir garantiert, dass `pos1` immer kleiner bleibt als `num_pixels`, da zum Beispiel `6 % 5 = 1` (1 Rest 1).

Aber `5 % 5 = 1` Rest 0. Wenn wir Pixel 0 in cyan leuchten lassen, im nächsten Schritt aber Pixel 1, dann würde jeweils die selbe LED leuchten. Dafür würde aber die letzte LED (an Stelle `num_pixels`) ausbleiben.

Um den "Fehler" in der MicroPython Implementierung wieder aufzufangen, muss bei `strip.set` für `pos` wieder `i+1` (`, pos1+1, pos2+1, ..`) übergeben werden:

```
strip.set(i+1, color, white=white_level, update = False)  
strip.set(pos1+1, color, white=white_level, update = False)  
strip.set(pos2+1, color, white=white_level, update = False)
```

Animationen steuern

Nun haben wir einige Animationen gebastelt, zwischen denen wir im Idealfall auch wechseln wollen.

Natürlich könnten wir jetzt sagen, wir lassen diese einfach nacheinander abspielen, sodass zum Beispiel Animation Nr. 1 für einige Zeit läuft, anschließend Animation Nr. 2, usw.

Besser wäre es doch, wenn wir selbst bestimmen könnten, wann die nächste Animation starten soll.

Der Touch Sensor

Unser ESP32 hat praktischerweise einige Sensoren an Board, wie zum Beispiel den Touch Sensor (oder Berührungssensor).
Genauer gesagt haben wir insgesamt 10 Touch Sensoren. Diese befinden sich an den Pins 0, 2, 4, 13, 12, 14, 15, 27, 33 und 32.

Beim Touch-Sensor handelt es sich um einen Kapazitive Näherungssensoren.

Die Funktion dieses Sensors beruht auf der Änderung des elektrischen Feldes in der Umgebung vor seiner Sensorelektrode (aktive Zone). Diese aktive Zone ist bei dem verbauten Sensor allerdings sehr klein.

Im Prinzip handelt es sich hierbei um eine Art Kondensator. Da sich die Kapazität eines Kondensators mit dem Abstand seiner Elektroden verändert, kann diese messbare Größe zur Erkennung von Berührung eingesetzt werden.

Wir wollen uns momentan nur auf den Touch Sensor konzentrieren und lassen den Teil mit den Animationen vorerst aus.

Alle Arten von Pins am ESP werden über das machine-Modul eingebunden. Wollen wir also den Touch Sensor an Pin 27 nutzen, schreiben wir:

```
import machine

# touch related values
touchPin = 27
touch = machine.TouchPad(machine.Pin(touchPin))
```

Somit haben wir eine Instanz des Touchpads in der Variable touch gespeichert.

Wenn wir nun den Wert des Touch Pins auslesen wollen, schreiben wir:

```
touch.read()
```

Da es hin und wieder beim Auslesen zu Fehlern (ValueError) kommen kann und diese dazu führen, dass die Ausführung des kompletten Programms anhält (was nur durch Neustart behoben werden kann), müssen wir diesen

Fehler abfangen.

Dieser Fehler tauchte bei meinen Tests anfangs überhaupt nicht auf, später dafür immer wieder. Falls dieser Fehler zu häufig auftreten sollte, empfiehlt es sich, einen anderen Pin als Touch-Sensor zu nutzen, weshalb ich die Pin-Nummer auch in eine separate Variable gespeichert habe. Dies wird, sobald das Programm eine gewisse Größe erreicht hat, zur Übersicht und besseren Wartbarkeit beitragen.

Um den Fehler abzufangen müssen wir dem Programm sagen, dass es versuchen soll, diesen Pin auszulesen. Falls dies nicht möglich sein sollte, kann (muss aber nicht) eine Ausgabe im Terminal geschehen.

Dies regelt man mit einer Ausnahmebehandlung (exception handling).

Der Code, der das Risiko für eine Ausnahme beherbergt, wird in ein **try**-Block eingebettet. Abgefangen werden diese Ausnahmen im **except**-Block.

Wie bereits erwähnt wird die Ausführung des Programms linear vorgenommen. Sprich es wird Zeile für Zeile ausgeführt. Wenn eine Funktion (auch aus anderen Modulen) aufgerufen wird, geht die Ausführung des Programms - wieder Zeile für Zeile - in der aufgerufenen Funktion weiter, bis diese komplett ausgeführt wurde. Anschließend geht die Ausführung weiter in der Zeile nach der aufgerufenen Funktion. Da wir den Wert des Sensors mehr als nur einmal auslesen wollen um die Werte zu vergleichen, müssen wir den Codeblock wieder in eine Endlosschleife packen. Es reicht, den Wert 5 mal pro Sekunde auszulesen, also nehmen wir einen sleep-Wert von 200 ms.

```
import machine, utime

# touch related values
touch_pin = 27
touch = machine.TouchPad(machine.Pin(touch_pin))

while True:
    # try to read touch pin
    try:
        touch_val = touch.read()

    except ValueError:
        print("ValueError while reading touch_pin")
    print(touch_val)
    utime.sleep_ms(200)
```

Bevor wir aber den Touch Sensor testen können, müssen wir noch ein Kabel an den Touch Pin (IO27) stecken. Nehmt dazu am besten ein JumperWire (male - female), also eins, das man mit der einen Seite über den Pin stülpen kann und bei dem dann auf der anderen Seite eine Spitze herausragt.

Main.py Speichern, Datei auf den ESP kopieren, Repl starten und Controller neustarten. Schon sehen wir im Terminal, welche Werte der Sensor hat. Ohne Berührung bewegt sich der Wert zwischen 400 und 500, mit Berührung grob zwischen 30 und 150. Diese Werte können allerdings je nach Umgebungsverhältnissen variieren

Setzen wir also den Schwellwert, ab dem der Sensor als "berührt" gelten soll, auf 180. Falls es im weiteren Verlauf zu Schwierigkeiten kommen sollte, wenn zum Beispiel eine Berührung registriert wurde obwohl keine stattgefunden hat, oder keine Reaktion erfolgen sollte obwohl der Sensor berührt wurde, sollte man diesen Schritt wiederholen.

Animationen mit dem Touch Sensor starten

Um Animationen mit dem Touch Sensor zu starten, müssen wir also jedes mal, wenn der Schwellenwert unterschritten wurde, die nächste Animation starten.

Um einen Codeblock nur auszuführen, wenn eine bestimmte Bedingung zutrifft, nutzt man bedingte Anweisungen, auch Verzweigungen genannt.

So können wir bestimmen:

Wenn der Sensorwert kleiner ist als Schwellwert:
starte die nächste Animation.

Bedingte Anweisungen / Verzweigungen

In der Programmierung versteht man unter einer bedingten Anweisung (oder Verzweigung) Codeteile, die nur unter bestimmten Bedingungen ausgeführt werden. Liegt diese Bedingung nicht vor, werden diese nicht ausgeführt. Alternativ kann (aber muss nicht) statt dessen ein anderer Codeteil ausgeführt werden.

Anders ausgedrückt: Eine Verzweigung legt fest, welcher von zwei (oder auch mehr) Programmteilen (Alternativen) in Abhängigkeit von einer (oder mehreren) Bedingungen ausgeführt wird.

So könnte man ganz einfach sagen:

Wenn dies: mache das.

Dazu könnte man noch definieren:

Wenn dies nicht: mache was anderes.

Oder wenn man mehrere Bedingungen in Frage kommen könnten:

Wenn dies: mache das

Wenn dies nicht sondern das: mache jenes

wenn dies und das nicht: mache was anderes.

Hierzu nutzt man die if-Anweisung.

Nehmen wir an, wir haben den derzeitigen Wochentag in der Variable "weekday" gespeichert. Wollen wir zum Beispiel erreichen, dass nur am Montag eine Anzeige im Terminal erscheint, die einen an diesen (meist ungeliebten) Tag erinnert schreiben wir:

```
if weekday == "monday":
    print("today is monday. ")
```

der Operator "==" ist ein sogenannte Gleichheitsoperator. Dieser prüft, ob die beiden Werte (davor und danach) gleich sind. Ist diese Bedingung erfüllt, wird True zurück geliefert, die Bedingung wird somit wahr und der nachfolgende Block darf ausgeführt werden.

Wollen wir, dass die Ausgabe nur erfolgt, wenn der Wochentag nicht der Montag ist, so können wir auch das Gegenteil mit einem Gleichheitsoperator prüfen. Also ob eine Bedingung nicht stimmt. Hierzu nutzt man "!=" (=ungleich):

```
if weekday != "monday":
    print("today's not monday")
```

Falls wir nur den Samstag oder Sonntag als Tag ausgeben, die restlichen Tage unter der Woche aber nur, dass kein Wochenende ist:

```
if weekday == "saturday":
    print("today is saturday")
elif weekday == "sunday":
    print("today is sunday")
else:
    print("no weekend today")
```

elif (Abkürzung von else if) wird nur ausgeführt, wenn die erste Bedingung (if) nicht zutrifft.

Else wird nur ausgeführt, wenn sowohl if als auch elif nicht zutreffen.

Man kann beliebig oft verzweigen.

In unseren Code eingefügt sieht das nun so aus:

```
touch_threshold = 180

while True:
    # read touch pin
    try:
        touch_val = touch.read()
    except ValueError:
        # if ValueError occurs print on terminal and blink 3 times in
red
        print("TouchPad Error")
        animations.blink(color=0xff0000, count=3, time_on=50,
time_off=50)

        # check if touch was registered
        if touch_val < touch_threshold:
            # print output and start function handleAnimations() without
            # argument to start next animation
            animations.next()
            utime.sleep_ms(200)
```

Jetzt haben wir aber mit `animations.next()` eine Funktion namens `next` erfunden, die es in `animations.py` aber noch gar nicht gibt! Diese müssen wir noch in die File `animations.py` einbauen.

Wie der Name schon sagt, soll diese Funktion einfach die nächste Animation aufrufen.

Am einfachsten erreicht man dies, wenn man alle Funktionen in eine Liste oder in einen Dictionary - schreibt, bei der man ganz einfach sagen kann: starte die Nächste!

Für den jetzigen Verwendungszweck würde eine Liste ausreichen. Aber da wir das ganze später noch ausweiten wollen, werden wir hier einen Dictionary verwenden. Spätestens wenn wir die Webseite erstellen, um bestimmte Animationen auszuwählen, werden wir nicht um einen Dictionary kommen.

Zur Erinnerung:

Ein Dictionary zeichnet sich dadurch aus, dass zu jedem Wert ein Schlüsselwort gehört, über den man diesen Wert auch abrufen kann.

In der File `animations.py` erstellen wir also zuerst einen Dictionary, mit dem wir arbeiten können. In diesem werden alle vorhandenen Animationen, zusammen mit einem Schlüsselwort - also den Namen der Animation als String - gespeichert. Wichtig ist, dass diese Variable erst nach den Funktionen, die eingesetzt werden sollen, deklariert wird. Beim Import der File `animations.py` werden - von oben nach unten - alle Variablen und Funktionen angelegt. Würden wir den Dictionary zu Beginn der Datei anlegen, würde der Interpreter die Funktionen noch gar nicht kennen und somit einen Fehler melden. Daher wird die Variable und die zugehörige `next()`-Funktion am Ende positioniert.

```
# saves all animations in a dictionary
animation_dict = ["blink" : blink, "running_dot" : running_dot,
"crossing_dot" : crossing_dots, "rainbowCycle" : rainbow_cycle ]
```

Wie man sehen kann, werden hier nur die Bezeichner der Funktionen gespeichert. Würden wir `"blink()"` anstatt `"blink"` schreiben, würde das zur sofortigen Ausführung der Funktion führen, was wir aber nicht wollen.

Anschließend überlegen wir uns, was genau in der Funktion `next()` geschehen soll.

Innerhalb dieser Funktion möchten wir, dass, wenn bisher keine Funktion aufgerufen wurde, die erste in der Liste gestartet wird. Falls aber im Vorfeld bereits eine gestartet wurde, soll die jeweils nächste gestartet werden.

Eine einfache, aber komplizierte Lösung wäre, dass wir zum Beispiel sagen, dass `blink()` die Animation Nr 0 ist, `running_dot()` Animation Nr 1 (...),

während ein Zähler die Touch-Eingaben hochzählt.

Einfacher geht es mit einem Iterator

Iterator

Ein Iterator ist einfach ein Zeiger, der auf das Element einer Liste (oder ähnlichem) zeigt, das gerade an der Reihe ist. Er funktioniert ähnlich wie eine for-Schleife.

```
for i in range(3):  
    print(i)
```

könnte man auch folgendermaßen schreiben:

```
numbers = [0,1,2]  
num_iterator = iter(numbers)  
  
while True:  
    try:  
        num = num_iterator.__next__()  
    except StopIteration:  
        break  
    print(num)
```

Im ersten Moment scheint dies zwar komplizierter, aber oftmals - wie in unserem Fall - definitiv die bessere Wahl.

Zu beachten ist hier auch der try / except - Bereich. Würden wir diesen auslassen, würde es zu einer Fehlermeldung (StopIteration) und folglich zu einer Unterbrechung des Programms kommen, sobald wir das Ende des Dictionarys erreicht, bzw. überschritten haben.

Wir möchten aber nicht, dass die Iteration abbricht, nachdem wir die letzte Animation erreicht haben. Wir wollen, dass, wenn das letzte Element erreicht wurde, die Iteration wieder von vorn beginnt.

Also behelfen wir uns eines kleinen Tricks:

Wir legen einfach eine neue Iteration an!

Dies führt uns aber zu einem weiteren Thema, das hier behandelt werden muss: Lokale und globale Variablen.

Lokale und globale Variablen in Python

Jede Variable, die man in Python in einer Funktion definiert, ist automatisch eine lokale Variable. Sprich sie existiert nur innerhalb dieser Funktion. Man kann zwar innerhalb der Funktion auf eine Variable, die außerhalb der Funktion definiert wurde, zugreifen, diese auch verändern. Jedoch hat diese Veränderung keinen Einfluss auf die Variable außerhalb.

Sprich wenn eine Variable x außerhalb der Funktion den Wert 10 hat, wir diesen innerhalb der Funktion durch 2 teilen, hat die Variable außerhalb der Funktion immernoch den Wert 10, während die Variable x in der Funktion den Wert 5 hat.

Weitere Infos sowie einige Beispiele sind unter anderem im Python-Forum zu finden:

https://www.python-kurs.eu/python3_global_lokal.php

Die Variable `animation_iterator` wird außerhalb der Funktion `next()` deklariert (=angelegt). Würden wir sie erst beim Aufruf der Funktion deklarieren, würde das bedeuten, dass, bei jedem Aufruf die Iteration bei 0 beginnen würde.

Da wir sie aber - um nach Ende der Iteration wieder bei 0 zu beginnen - im `except`-Block neu anlegen und somit verändern müssen, würde dies zu einer Fehlermeldung führen (Variable referenziert bevor sie deklariert wurde), da wir die Variable zuerst verwenden und anschließend verändern. Gerne darf dieser Effekt ausprobiert werden!

Daher müssen wir in der Funktion angeben, dass es sich hier um eine globale Variable handelt.

Anders verhält es sich bei `animation_dict`. Diese wird in der Funktion lediglich aufgerufen, nicht aber verändert. Daher muss diese nicht global sein.

```
# saves all animations in a dictionary
animation_dict = ["blink" : blink, "running_dot" : running_dot,
                  "crossing_dots" : crossing_dots, "rainbowCycle" : rainbow_cycle ]
animation_iterator = iter(animation_dict)

def next():
    global animation_iterator
    try:
        running_function = animation_iterator.__next__()
        running_function()
    except StopIteration:
        animation_iterator = iter(animation_dict)
    next()
```

`running_function = animation_iterator.__next__()` bewirkt, dass die nächste Funktion in die Variable `running_function` gespeichert wird. Mit **`running_function()`** wird diese dann ausgeführt.

Damit nun aber bei Erreichen des except-Blocks aus Sicht des Nutzers nicht einfach 'nichts' passiert, können wir - aus der Funktion heraus - selbe noch einmal aufrufen.

Wenn eine Funktion sich selbst aufruft nennt man diese eine rekursive Funktion. Da diese im Projekt nur eine untergeordnete Rolle spielt, verweise ich auf Quellen im Internet, um weitere Informationen zu erhalten:

https://www.python-kurs.eu/rekursive_funktionen.php
<http://python4kids.net/how2think/kap04.htm>

Fertig für den Test!

Also, Datei speichern, zum Terminal wechseln. Datei auf den ESP kopieren, Repl starten, ESP neu starten.

Nun sollte es möglich sein, die erste Animation mit dem Touch Sensor zu starten.

Bei erneuter Berührung sollte die laufende Animation stoppen und eine neue starten.

Allerdings wird dies so nicht funktionieren!
Der Sensor reagiert nicht mehr auf Eingaben!

Dader Code linear abgearbeitet wird, wird die Ausführung von main.py unterbrochen sobald die Animation gestartet wurde. Die Animation läuft ebenso in einer Endlosschleife. Das bedeutet, es ist momentan nicht möglich, den Sensor auszulesen und diese Animation von außen abubrechen.

Natürlich könnten wir jetzt einfach sagen, wir begrenzen die Ausführung der Animationen, sodass die Schleife nur noch 20 Mal ausgeführt werden kann. Aber was ist, wenn wir die Animation früher wechseln möchten?

Um mehrere Prozesse quasi Parallel laufen lassen zu können, behilft man sich mit dem Prinzip des Multithreading.

Multithreading

Multithreading wird wahrscheinlich den wenigsten ein Begriff sein. Dafür dürfte Multitasking bekannt sein!

Jeder PC beherrscht Multitasking. So werden mehrere Programme quasi gleichzeitig ausgeführt.

Quasi gleichzeitig bedeutet, sie laufen nicht wirklich gleichzeitig. Es bedeutet lediglich, dass das Betriebssystem dafür sorgt, dass jedem Programm eine gewisse CPU-Zeit zugesprochen wird. Sprich ein Zeitfenster, indem es die CPU (also den Prozessor) beanspruchen darf. In Wirklichkeit laufen diese Prozesse gestückelt nacheinander - aber in solchen engen Zeitfenstern, dass es der User nicht wahrnimmt.

Multithreading verhält sich ähnlich, nur dass hier nicht zwei Programme gemeint sind, die parallel laufen, sondern ein Programm in verschiedene Threads, also verschiedene Aufgabenbereiche gegliedert wird, die parallel abzuarbeiten sind.

Bisher läuft unser Programm in einem Thread, dem main-Thread. Weitere Threads können wir in unserem Code hinzufügen.

Hierfür nutzt man das Micropython Modul `_thread`.

Im Python-Forum wird nochmal das Verhalten vom Threads erklärt:

<https://www.python-kurs.eu/threads.php>

Allerdings können wir diese Beispiele nicht anwenden, da Micropython lediglich eine Abgespeckte Version von Python ist. Die Modul-Beschreibung zu `_thread` inklusive Beispielen findet man hier:

https://github.com/loboris/MicroPython_ESP32_psRAM_LoBo/wiki/thread

Um eine Funktion in einem Thread zu starten, nutzt man

```
thread = _thread.start_new_thread(th_name, th_func, args [, kwargs])
```

th_name ist der Name des Threads, der als String (also in doppelten Anführungszeichen) anzugeben ist.

th_func ist die Funktion, die gestartet werden soll. Hier soll man nur den Bezeichner angeben, ohne Klammer und Argumente mit **args** werden dann, falls vorhanden, die Argumente als Tupel übergeben. Falls keine Argumente übergeben werden, muss man eine leere Klammer einfügen. Falls nur ein Argument übergeben wird, muss vor der schließenden Klammer noch ein Komma stehen. Dies symbolisiert ein Tupel mit nur einem Wert. Zum Beispiel (arg1,)

kwargs sind optional. Hier werden die optionalen Parameter mit ihren Schlüsselworten übergeben. Diese müssen als Dictionary eingetragen werden. Bsp: {"arg1": value1, "arg2": value2}.

Die Funktion gibt beim Aufruf die ID des Threads zurück, die dann in einer Variablen (hier: `thread`) gespeichert werden sollte.

Diese ID wird benötigt, um die Ausführung des Threads auszusetzen oder ihn wieder zu stoppen.

mit **`_thread.suspend(thread_id)`** und **`_thread.resume(thread_id)`** wird die Ausführung des Threads unterbrochen und anschließend wieder fortgesetzt.

Dies funktioniert allerdings nur, wenn in der auszuführenden Funktion die Unterbrechung mit **`_thread.allowsuspend(True)`** erlaubt wurde.

Möchte man den Thread stoppen, kann man **`_thread.stop(thread_id)`** nutzen. Dadurch wird dem Thread allerdings nur eine Benachrichtigung geschickt! Damit diese auch verarbeitet werden kann, muss in dem Thread, der abgebrochen werden soll, die Funktion **`notification = _thread.getnotification()`** ausgeführt werden. Sobald in `notification` der selbe Wert steht, der auch `_thread.EXIT` hat (`EXIT` ist keine normale Variable sondern eine Konstante. Wenn man hier einen Namen anstatt eines Werts nutzt trägt das zur besseren Lesbarkeit bei)

Alternativ kann man auch **`_thread.wait(timeout)`** nutzen.

Dies ist sehr praktisch, da man diese Funktion anstelle von `utime.sleep_ms(timeout)` einsetzen kann. Damit wird erreicht, dass während des Timeouts trotzdem Benachrichtigungen erhalten werden können und auch währenddessen ein Abbruch möglich ist.

Durch das Schlüsselwort `return` wird die derzeitig laufende Schleife (egal ob endlich oder endlos) wieder verlassen wird.

Implementierung von `_thread`

Wir haben nun schon einige Animationen geschrieben. Bevor wir nun in jeder einzelnen prüfen, ob eine Benachrichtigung erhalten wurde, packen wir das ganze lieber in eine Funktion.

Es gehört zu einem guten Programmierstil, dass kein gleicher Codeblock mehrmals vorkommt. In dieser Funktion wollen wir den `timeout` laufen lassen (der `utime.sleep_ms` ersetzen soll). Also muss der Funktion mitgeteilt werden, wie lange dieser `timeout` sein soll.

Dann wollen wir prüfen, ob eine Benachrichtigung vorliegt. Wenn die Benachrichtigung `_thread.EXIT` ist, wollen wir zurückgeben, dass die Abbruchbedingung gegeben ist. Also brauchen wir ein `return`-Statement.

Dies können wir so erreichen:

```
def waitForExitNotification(timeout):
    notification = _thread.wait(timeout)

    if notification == _thread.EXIT:
        return True
    else:
```



```
return False
```

Um diese Funktion in den Animationen aufzurufen und den Return-Wert zu verarbeiten können wir `utime_sleep_ms(timeout)` durch folgendes ersetzen:

```
exit = waitForExitNotification(timeout)

if exit:
    return
```

Dies fügen wir jetzt anstelle von `utime.sleep_ms` in jede Animation ein, die in einer Endlosschleife läuft. Nicht vergessen: Den Wert von `timeout` angeben!

Jetzt haben wir unsere Funktionen so angepasst, dass sie mit den Benachrichtigungen umgehen können. Nun müssen wir noch dafür sorgen, dass der Thread gestartet und gestoppt werden kann.

Um nicht für jede einzelne Animation `_thread.start` anlegen zu müssen, können wir sagen, dass `animations.next()` einfach im Thread gestartet werden soll. der Aufruf der Animation geschieht dann innerhalb von `next()`.

Also ersetzen wir in `main.py` `animations.next()` durch

```
animation_thread = _thread.start_new_thread("animation", animations.next,
() )
```

Jetzt haben wir den Start der Animation geregelt, aber noch nicht den Abbruch!

Wenn wir aber bereits eine Animation am laufen haben und wir diese zuerst abbrechen müssen, müssen wir eine Eigenschaft der Variablen `animation_thread` prüfen bevor sie deklariert wurde!

Variablendeklarationen sollten wenn möglich immer zu Beginn einer Funktion oder eines Moduls geschehen - es sei denn, es werden dazu Angaben benötigt, die erst später im Code vorkommen (wie im Fall des `animation_dict`).

Also werden wir dies in `main.py` auch zu Beginn - am besten nach den Variablen für `touch` - einsetzen und den Wert 100 zuweisen.

`_thread.stop(animation_thread)` wird eine Zeile über der Zeile, in der der Thread gestartet wird, eingesetzt.

Hier nochmal die komplette `main.py`:

```
import machine, utime, _thread
import animations

touch_pin = 27
touch = machine.TouchPad(machine.Pin(touch_pin))
touch_threshold = 180
```

```

animation_thread = 100

while True:
    try:
        touch_val = touch.read()
    except ValueError:
        print("ValueError while reading touch_pin")

    if touch_val < touch_threshold:
        print("touched!!")
        _thread.notify(animation_thread, _thread.EXIT)
        utime.sleep_ms(50)
        animation_thread = _thread.start_new_thread
        ("animation", animations.next, () )
        utime.sleep_ms(200)

```

Helligkeit der LEDs steuern

Bestimmt ist bereits aufgefallen, wie hell die LEDs leuchten können. Diese Helligkeit kann man aber verändern!

Relativ einfach kann man mit `Neopixel.brightness(brightness_value)` bestimmen, wie hell die LEDs leuchten sollen. `brightness_value` kann hier einen Wert zwischen 0 und 255 annehmen.

Gerne kann dies wieder in Repl ausprobiert werden! Allerdings empfehle ich, dies bei unterschiedlichen Lichtverhältnissen zu testen, um ein Gefühl dafür zu bekommen, wann welcher Wert passt.

```

>>> import machine, animations
>>> animations.running_dot()
>>> animations.strip.brightness(1)

```

Der Photowiderstand

Selbstverständlich kann man diese Werte manuell definieren. Aber viel besser wäre es doch, wenn dies automatisch, abhängig von der Umgebungshelligkeit, geschehen würde - so wie es auch bei Smartphones passiert!

Hierzu nutzt man einen Photowiderstand.

Ein Photowiderstand ist ein Halbleiter, dessen Widerstand abhängig von der Umgebungshelligkeit ist. Er wird auch LDR (=Light Dependent Resistor) genannt.

Er wird oft als Beleuchtungsstärkemesser, Flammenwächter, Dämmerungsschalter und als Sensor in Lichtschranken verwendet.

Um den Photowiderstand am ESP einzusetzen muss man wissen, dass der ESP grundsätzlich eigentlich nur digitale Ein- und Ausgänge hat. In der Regel können diese nur Binärcode ausgeben und entgegennehmen.

Analog Digital Converter - das ADC Modul

Allerdings besitzt der ESP auch einen Analog-Digital-Wandler, auch ADC (=Analog Digital Converter) genannt.

Den ADC findet man - wie alles, was mit Pins zu tun hat - im machine Modul:

https://github.com/loboris/MicroPython_ESP32_psRAM_LoBo/wiki/adc

Die Modulbeschreibung sieht relativ komplex aus für Menschen, die mit Elektrotechnik nicht viel zu tun haben.

Aber wir setzen hier nur die nötigsten Bausteine ein, um die Umgebungshelligkeit zu messen.

Zu beachten ist allerdings, dass es zwei Arten von ADC gibt. ADC1 und ADC2. Jeweils auf verschiedenen Pins. ADC2 kann man allerdings nur nutzen, wenn WLAN nicht aktiv ist. Da wir die WLAN Funktion später noch brauchen werden, nutzen wir ADC1.

Diesen kann man auf den Pins 32 - 38 nutzen.

Eine Instanz des ADC bilden wir mit **ldr = machine.LDR(pin[, unit])**, wobei unit standardgemäß 1 (für ADC1) ist. Daher muss dies nicht angegeben werden.

ldr_val = ldr.read() liest die Spannung (in mV), die am Sensor anliegt und speichert sie in ldr_val.

Aus der Modulbeschreibung (machine.ADC.vref) geht hervor, dass die Spannung einen Wert zwischen 0 und 1100 mV annehmen kann, wobei 0 als sehr dunkel und 1100 als sehr hell zu interpretieren ist.

Diesen Wert können wir also nicht einfach 1:1 nutzen, um die Helligkeit der LED zu setzen, da diese nur Werte zwischen 0 und 255 akzeptiert. Für den Anfang reicht es daher, den Wert des Sensors auf den Helligkeitswert zu "mappen".

Dies ist ganz einfache Mathematik:

$$\text{brightness_val} = \text{ldr_val} / 1100 * 255.$$

Würde der LDR nun einen Wert von 700 haben, würde als Ergebnis 162,272727273 herauskommen. Das Problem dabei: Neopixel.brightness kann nur mit ganzen Zahlen, sogenannten Integer umgehen!

Dies ist aber leicht behoben. **int(brightness_val)** sorgt dafür, dass aus der Gleitkommazahl 162,272727273 die Ganzzahl 162 wird.

Hierbei wird allerdings nicht gerundet! der Wert wird einfach nach dem Komma abgetrennt, sodass wir eine ganze Zahl erhalten. Da wir keine sensiblen Berechnungen durchführen ist dies vernachlässigbar. Um Gleitkommazahlen zu runden, gibt es Funktionen im Modul math.

brightness_val können wir nun einfach einer Funktion in animations.py übergeben, die dann die Helligkeit des Strips setzt.

Die Funktion, um die Helligkeit in animations.py zu setzen sieht ganz einfach aus:

```
def set_brightness(brightness_val):  
    strip.brightness(brightness_val)
```

Aufgerufen wird sie in main.py - am besten in der Endlos-Schleife:

```
while True:  
    ldr_val = ldr.read()  
    brightness_val = int(ldr_val / 1100 * 255)  
    animations.set_brightness(brightness_val)
```

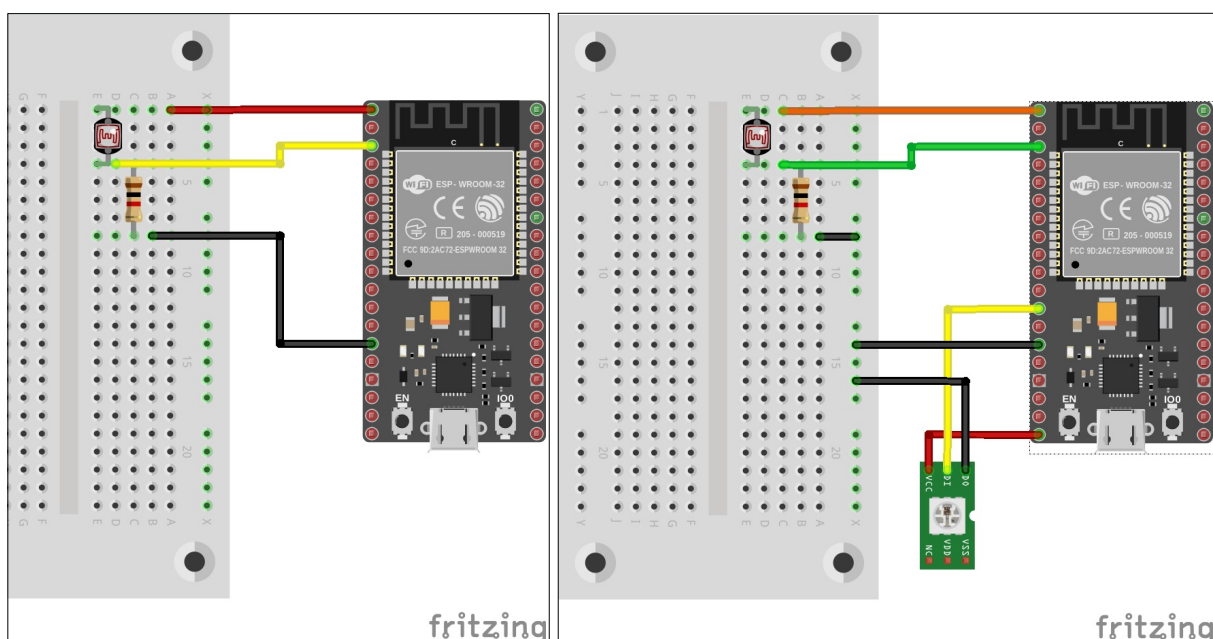
Diese vier Zeilen lassen sich auch auf zwei komprimieren!

```
while True:  
    animation.set_brightness(int(ldr.read() / 1100 * 255)t)
```

Photowiderstand mit ESP verbinden

Um den Photowiderstand ordnungsgemäß auszulesen, müssen wir einen sogenannten Spannungsteiler bauen. Hierzu benötigen wir neben dem Photowiderstand und einer Leitung, um die Werte an den ESP zu übertragen eine Spannungsquelle mit 3.3V und einen passenden Widerstand (1 kΩ). Eine Seite des LDR verbinden wir also mit dem 3.3V Ausgang am ESP, die andere Seite wird mit dem Widerstand verbunden. Diesen wiederum verbinden wir mit GND am ESP um den Stromkreis zu schließen. Um die Werte auszulesen, müssen wir nun einen der ADC Pins (hier Pin 36) zwischen LDR und Widerstand mit dem Stromkreis verbinden.

Mit einem kleinen Breadboard kann man die Schaltung ganz einfach umsetzen:



Das Breadboard

Zum besseren Verständnis: Bild 1 beinhaltet die LDR-Schaltung ohne angeschlossenen LED Strip, Bild 2 mit angeschlossenem LED-Strip.

Um die Schaltung besser zu verstehen, muss man wissen, wie ein Breadboard aufgebaut ist:

Ein Breadboard besteht aus sehr vielen kleinen Steckplätzen, die auf eine bestimmte Weise miteinander verbunden sind:

Auf der linken und rechten Seite befinden sich die Steckplätze, die am Besten für VCC oder GND eingesetzt werden. Diese Steckplätze sind jeweils senkrecht miteinander verbunden. So kann ich, wenn ich für zwei oder mehr Elemente GND brauche, den GND vom ESP einfach mit dieser Reihe verbinden. Genauso verfähre ich mit GND der die LED sowie mit GND des LDR. So kann ich einen GND-Anschluss am ESP für mehrere Bauteile verwenden.

Im mittleren Teil sind jeweils fünf Steckplätze auf der rechten und auf der linken Seite waagerecht miteinander verbunden. So kann ich - wie hier - eine Seite des LDR, eine Seite des Widerstands und ein Kabel zum Auslesen der Werte in eine Reihe stecken, um sie miteinander zu verbinden.

LDR testen

Nachdem der LDR ordnungsgemäß angeschlossen wurde und die Dateien auf den ESP kopiert wurden, darf nun getestet werden.

Dazu kann man den LDR einfach mit der Hand bedecken, oder mit einer Leuchtquelle hellerem Licht aussetzen.

Wenn alles richtig gemacht wurde, sollten die LEDs heller leuchten, je heller die Umgebung ist.

Der ESP als WebServer

Ein Webserver ist ein System, das Webseiten bereitstellt. Um aber auf diese Webseiten zugreifen zu können, müssen sich der Server und der Client, der die Seite aufrufen möchte, im selben Netzwerk (oder im Internet) befinden.

Wir wollen den Webserver nutzen, um über die Webseite direkt unsere Animationen auszuwählen.

Um dies zu ermöglichen, muss man zuerst dafür sorgen, dass man sich mit seinem Rechner oder Smartphone mit dem ESP verbinden kann. Hierfür ist das network-Modul zuständig.

WLAN auf dem ESP

Das network-Modul bietet neben der Möglichkeit, eine WLAN Verbindung herzustellen unter anderem die Möglichkeit, einen FTP-Server (zur Bereitstellung von Datei(systemen)), einen Telnet-Server (um Kommandos über das Terminal zu senden), einen MQTT-Client (zum Senden und Empfangen von Nachrichten - Facebook Messenger nutzt zum Beispiel das MQTT-Protokoll) oder einen mDNS-Service (zur Namensauflösung - sprich um eine URL in eine IP-Adresse - und andersrum - zu konvertieren). Wir beschränken uns aber vorerst auf die WLAN-Connectivity.

https://github.com/loboris/MicroPython_ESP32_psRAM_LoBo/wiki/network

Um den Code übersichtlich zu halten, wird an dieser Stelle empfohlen, eine neue Datei anzulegen. Ich nenne sie mal wifi.py Selbstverständlich beginnen wir auch hier damit, die erforderlichen Module einzubinden:

```
import network, utime
```

Man kann den ESP als Station (STA) nutzen, was bedeutet, dass der ESP sich lediglich mit einem anderen AccessPoint (einem Router zum Beispiel) verbinden kann, als AccessPoint (AP), wodurch sich andere Geräte (wie unser Computer oder Mobiltelefon) mit dem ESP verbinden können, oder aber als Kombination aus beidem, was bedeutet, er kann sich mit einem Router verbinden und gleichzeitig als AccessPoint dienen.

AP-Mode

Um also den ESP als WebServer nutzen zu können, müssen wir eine Instanz von WLAN im AP-Mode nutzen:

```
ap = network.WLAN(network.AP_IF)
```

Dadurch wurde das WLAN-Interface allerdings noch nicht gestartet! Dies geschieht erst durch

```
ap.active(True)
```

Zudem müssen wir noch eine SSID (Service Set Identifier = der Name des WLAN-Netzwerks) und ein zugehöriges Passwort definieren. Hierbei handelt es sich um Konstante, die groß geschrieben werden sollten:

```
AP_SSID = "esp32"  
AP_PASS = "HuchEinPw"
```

Diese Einstellungen muss man dem ESP noch mitteilen:

```
ap.config(essid=AP_SSID, password=AP_PASS)
```

Anschließend sollte man die Konfiguration (wie zum Beispiel die IP-Adresse, unter der der ESP erreichbar ist) mit `print(ap.ifconfig())` im Terminal ausgeben. Dies geht aber nur, sobald der AccessPoint gestartet wurde, was eventuell länger dauern könnte wie die Zeit, die zwischen diesen beiden Befehlen vergeht. `WLAN.isconnected(False)` wartet im AP-Mode nicht etwa darauf, bis sich ein Client verbunden hat. Dieses "False" bewirkt, dass die Funktion nur True zurückgibt, wenn der AP-Mode gestartet wurde.

```
timeout = 50  
while not ap.isconnected(True):  
    utime.sleep_ms(100)  
    timeout -= 1  
    if timeout == 0:  
        break  
print("\n===== AP started =====\n")  
print(ap.ifconfig())
```

Bei "\n" handelt es sich um eine sogenannte Escape-Sequenz. Diese wird in Strings dazu genutzt, den "Cursor" um eine Zeile nach unten zu schieben.

Weitere Infos hierzu:

<https://de.wikipedia.org/wiki/Escape-Sequenz>

Um diesen Code nach dem Start ausführen zu lassen, genügt es, die Datei - analog zu `animatons.py` - in `main.py` mittels `import` einzubinden.

Nachdem nun beide Dateien gespeichert und auf dem ESP aktualisiert wurden kann man die Funktionalität prüfen, indem man die verfügbaren WLAN-Netzwerke prüft. Befindet sich hier ein Netzwerk namens "esp32" und kann man sich mit dem angegebenen Passwort damit verbinden, ist dieser Schritt auch geschafft!

STA-Mode

Sobald man länger mit dem Projekt arbeitet, kann es Umständlich werden, den ESP lediglich im AP-Mode zu betreiben. Die Endgeräte registrieren, dass zwar WLAN verbunden ist, jedoch keine Internetverbindung besteht. Das kann dazu führen, dass sich die Geräte automatisch wieder mit dem heimischen (oder schulischen) Netzwerk verbinden. Oder aber denkt nicht daran, dass man mit dem ESP verbunden ist und wundert sich, warum man keine anderen Webseiten aufrufen kann. Beim Debuggen (Fehler finden und entfernen) und Testen des Webserverns kann dies sehr zeitraubend sein.

Somit erstellen wir eine weitere Instanz, dieses mal im STA-Mode, und aktivieren diese:

```
sta = network.WLAN(network.STA_IF)
sta.active(True)
```

Auch hier müssen wir SSID und Passwort angeben - allerdings vom heimischen Router:

```
STA_SSID = "DeineSSID"
STA_PASS = "DeinPw"
```

Um Fehlfunktionen zu vermeiden, die auftreten können, sobald man nicht in Reichweite des Heimnetzes ist, sollte man vorher prüfen, ob eine SSID mit dem gegebenen Namen vorhanden und nur dann versuchen, eine Verbindung herzustellen. Auch hier sollte man dem ESP etwas Zeit geben, bis er sich verbunden hat:

```
networks = sta.scan()
for nets in networks:
    if nets[0] == bytes(STA_SSID, 'utf-8'):
        sta.connect(STA_SSID, STA_PASS)

        timeout = 50
        while not sta.isconnected():
            utime.sleep_ms(100)
            timeout -= 1
            if timeout <= 0:
                break

        if sta.isconnected():
            print("\n===== STA CONNECTED =====\n")
            print(sta.ifconfig())
if not sta.isconnected():
    print("\n===== STA NOT CONNECTED =====\n")
    sta.active(False)
```

sta.scan() scannt alle verfügbaren WLAN-Netzwerke und speichert sie in `networks`.

`networks` beinhaltet dann eine Liste mit Tupel. Jeweils an erster Stelle [0] dieser Tupel ist die SSID, allerdings in bytes, zu finden.

Da `STA_SSID` aber ein String ist, muss man diesen ebenso in bytes

konvertieren, um beide Angaben miteinander vergleichen zu können.

Durch diesen Vergleich wird gewährleistet, dass ein Verbindungsversuch nur stattfindet, wenn die angegebene SSID auch Verfügbar ist.

mDNS Service

Nachdem eine Netzwerkverbindung im jeweiligen Modus erfolgreich hergestellt wurde, werden jeweils die IP Adressen ausgegeben, unter denen unsere Website später erreichbar sein wird.

Leider können sich nur wenige Menschen solche Zahlenfolgen dauerhaft merken!

Aus diesem Grund gibt es DNS-Server (Domain Name System).

Ein DNS-Server wird auch kurz "Telefonbuch des Internets" genannt. Ähnlich wie man in einem Telefonverzeichnis nach einem Namen sucht, um die Telefonnummer heraus zu bekommen, schaut man im DNS nach einem Computernamen, um die dazugehörige IP-Adresse zu bekommen. Die IP-Adresse wird benötigt, um eine Verbindung zu einem Server aufbauen zu können, über den nur der Computername bekannt ist.

mDNS erfüllt einen ähnlichen Zweck- nur im lokalen Netzwerk. Somit ist die Seite unter [Domain-Name].local zu erreichen.

https://de.wikipedia.org/wiki/Zeroconf#Multicast_DNS

https://github.com/loboris/MicroPython_ESP32_psRAM_LoBo/wiki/mdns

Folgendermaßen erreichen wir, dass unsere Website unter der Adresse "rainbowwarrior.local" aufzurufen ist:

```
hostname = "RainbowWarrior"
try:
    mdns = network.mDNS()
    mdns.start(hostname, "MicroPython with mDNS")
    mdns.addService('_http', '_tcp', 80, "MicroPython", {"board":
"ESP32", "service": "mPy Web server"})
    print("webpage available at {}.local".format(hostname))
except:
    print("mDNS not started")
```

Leider funktioniert dies nicht mit Android-Mobiltelefonen. Hier muss man weiterhin die IP-Adresse im Browser angeben. Diese werden allerdings bei erfolgreichem Aufbau der Verbindung(en) im Terminal ausgegeben.

Das microWebSrv-Modul

Ein Webserver erfüllt die Funktion, Webseiten (im Internet) bereitzustellen. Selbstverständlich können wir auf dem ESP nicht den vollen Funktionsumfang eines richtigen Servers erwarten. Aber die Grundfunktionalität ist gewährleistet.

Das microWebSrv-Modul stellt HTML-Seiten bereit, die - entweder als String vorliegen - oder als html-File in /flash/www auf dem Microcontroller gespeichert sind.

Eine Modulbeschreibung inklusive Beispielen ist hier zu finden:
https://github.com/loboris/MicroPython_ESP32_psRAM_LoBo/wiki/microWebSrv

Auch hier empfiehlt es sich, den Webserver-Code in eine eigene Datei zu packen. Nennen wir sie webSrv.py.

Neben dem MicroWebSrv beinhaltet microWebSrv auch MicroWebSockets, was wir allerdings nicht benötigen. Daher genügt es, den import auf MicroWebSrv zu beschränken:

```
from microWebSrv import MicroWebSrv
```

Um eine Instanz des MicroWebSrv zu erstellen und diesen zu starten schreiben wir:

```
srv = MicroWebSrv(webPath = 'www')  
srv.Start(threaded = True, stackSize= 8192)
```

webPath gibt an, in welchem Ordner die HTML-Dateien zu finden sind.
threaded = True ist zwingend notwendig, da wir parallel in main.py weiterhin die Touch Sensor-Eingaben sowie auch die Anfragen vom WebServer verarbeiten müssen.

Um die gespeicherten Seiten über die URL aufrufen zu können, müssen wir Route-Handler definieren. So wird gewährleistet, dass wir unter rainbowwarrior.local unsere Startseite aufrufen können, bzw unter rainbowwarrior.local/[irgend-ne-andere-seite] andere Seiten, die wir bereitstellen wollen. Hier ein einfaches Beispiel für einen Route-Handler, der bei einem Zugriff (bzw. einer Anfrage) auf rainbowwarrior.local die Seite index.html (im Ordner www) zurück gibt:

```
@MicroWebSrv.route('/')  
def _httpHandler(httpClient, httpResponse) :  
    httpResponse.WriteResponseFile(filepath = 'www/index.html',  
    contentType= "text/html", headers = None)
```

Die Route-Handler sollten bekannt sein bevor die WebServer-Instanz erstellt wird, weshalb man diese bitte oberhalb davon in die Datei einträgt.

Eine hübsche Webseite habe ich bereits vorbereitet. Diese muss nun nach und nach mit den bereits erstellten Animationen gefüllt werden.

Die Seite wurde mithilfe von Bootstrap und CSS aufgehübscht. Für eine fehlerfreie Darstellung ist es also notwendig, dass zwei Dateien im Ordner '/flash/www' vorhanden sind:

<https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css>
https://github.com/crxcv/Leuchteding/blob/master/pyboard_code/www/style.css

Falls ein einfacher Download nicht möglich sein sollte, können die Inhalte auch mit copy + paste in Files mit den Namen "bootstrap.min.css" und "style.css" übertragen werden.

Diese werden auf der HTML-Seite (index.html) direkt im <head>-Tag eingebunden:

```
<head>
  <title>RainbowWarriorSettings</title>
  <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
  <link rel="stylesheet" href="bootstrap.min.css" >
  <link rel="stylesheet" href="style.css">
</head>
```

Hier einige Basics zu HTML.

Kommunikation zwischen HTML-Seite und dem Server:
<https://wiki.selfhtml.org/wiki/HTTP/Anfragemethoden>

Formulare:
<https://wiki.selfhtml.org/wiki/HTML/Formulare/form>

Aufbau einer HTML-Seite

HTML ist eine „Strukturierungssprache“, mit der dem Browser „gesagt“ wird, wie die Inhalte strukturiert sind: welche Bereiche (Buchstaben, Wörter, Sätze) z.B. Überschriften, was Absätze, was Aufzählungen, was Tabellen usw. sind.

HTML ist keine Programmiersprache – man beschreibt, welche logische Struktur ein Inhalt hat (nicht mehr, nicht weniger).

Jeder einzelne Bereich wird mit einem Tag definiert. Der Beginn des Bereichs (am Beispiel Absatz - (engl. paragraph)) wird mit einem öffnenden Tag <p>, das Ende des Bereichs mit einem schließenden Tag </p> gekennzeichnet.

Die verschiedenen Bereiche

1. DOCTYPE

```
<! DOCTYPE html>
```

Im Bereich DOCTYPE wird dem Internet-Browser mitgeteilt, was er an Befehlen erwarten kann und an welchem Standard man sich bei der Erstellung der Seite gehalten hat.

2. HEAD

```
<head>  
  <title>Titel der Webseite</title>  
  <link rel="stylesheet" href="style.css">  
</head>
```

Im Bereich HEAD stecken die Metainformationen über die Seite, also beispielsweise der Titel der Seite, der im Browserfensterkopf angezeigt wird sowie Links zur CSS-File.

CSS (Cascading Style Sheet) ist für das Aussehen der der Webseite verantwortlich

3. BODY

```
<body>  
  ...  
</body>
```

Im Bereich BODY steckt der eigentliche Inhalt der Seite und die HTML-TAGs.

Mehr Informationen hierzu inklusive einem Tutorial findet man unter anderem hier:

<https://www.html-seminar.de/einsteiger.htm>

<https://www.html-seminar.de/html-grundlagen.htm>

<https://www.html-seminar.de/html-seitenaufbau.htm>

<https://www.html-seminar.de/css-lernen.htm>

HTML - Formulare

Formulare eignen sich hervorragend um Nutzereingaben in Form von Text (Bsp. Name, Nachrichten ...) oder einem Auswahlmenü an den Server zu senden.

<https://www.html-seminar.de/formulare.htm>

<https://wiki.selfhtml.org/wiki/HTML/Formulare>

HTTP - Kommunikation mit dem Server

Bei HTTP handelt es sich um eine Art Sprache, in der ein Webserver und ein Browser miteinander kommunizieren. Diese Kommunikation erfolgt nach einem festgelegten Schema, das man auch "Protokoll" nennt. HTTP ist also ein Protokoll, daher auch der Name "HyperText Transport Protocol".

Request und Response

Beim Aufruf einer Seite sendet der Browser zuerst einen "Request" (Anfrage, Bsp: ich möchte die Seite sehen), die vom Server mit einem "Response" (Antwort, Bsp: ich zeige Dir die Seite) beantwortet wird.

GET und POST

Jeder Request wird durch die Angabe der Methode eingeleitet. Methoden bestimmen die Aktion der Anforderung. Die aktuelle HTTP-Spezifikation sieht acht Methoden vor: OPTIONS, GET, HEAD, POST, PUT, DELETE, TRACE und CONNECT. Wir beschränken uns jetzt aber auf die Methoden GET und POST.

Die mit Abstand wichtigste Methode ist die Methode **GET**. Hiermit wird - mittels einem Request - zum Beispiel ein Dokument (die aufzurufende HTML-Seite) angefordert. Als Antwort auf diesen Request erhält der Client (unser Browser) die HTML-Seite als Response

Die **POST**-Methode übermittelt in erster Linie Formulareingaben an den Webserver.

Werden durch den Request lediglich andere Daten als Antwort empfangen, so ist die GET-Methode die richtige Wahl.

Werden durch den Request Daten auf dem Server verändert, ist die POST-Methode die richtige Wahl.

Werden Daten für Logins, insbesondere Passwörter übermittelt, dann ist nur POST die einzig richtige Wahl.

https://de.wikipedia.org/wiki/Hypertext_Transfer_Protocol

<https://wiki.selfhtml.org/wiki/HTTP/Einsteiger-Tutorial>

<https://wiki.selfhtml.org/wiki/HTTP/Anfragemethoden>

Die HTML-Seite

```
<!DOCTYPE html>
<html>

  <head>
    <title>RainbowWarriorSettings</title>
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
    <link rel="stylesheet" href="bootstrap.min.css" >
    <link rel="stylesheet" href="style.css">
  </head>

  <body class="h-100">
    <div class="ground"></div>

    <div class="sky">
      <div class="cloud variant-1"></div>
      <div class="cloud variant-2"></div>
      <div class="cloud variant-3"></div>
      <div class="cloud variant-4"></div>
      <div class="cloud variant-5"></div>
    </div>

    <div class="rainbow-preloader">
      <div class="rainbow-stripe"></div>
      <div class="rainbow-stripe"></div>
      <div class="rainbow-stripe"></div>
      <div class="rainbow-stripe"></div>
      <div class="rainbow-stripe"></div>
      <div class="shadow"></div>
      <div class="shadow"></div>
    </div>

    <div class="container text-white align-items-center d-flex flex-column h-100 justify-
content-end pb-md-5">
      <h1 id="id1">RainbowWarrior</h1>
      <h2 id="id3"></h2>

      <div class="mt-3">
        <form class="form-group" method="POST" action="/" role="form">
          <h3 class="mb-3" id="id1">LED Pattern ausw&auml;hlen:</h3>
          <div class="input-group">
            <select class="custom-select" id="inputGroupSelect04" name="light">
              <option selected>Animation ausw&auml;hlen ...</option>

              </select>
            <div class="input-group-append">
              <button class="btn btn-dark" type="submit" >ausw&auml;hlen</button>
            </div>
          </div>
        </form>
      </div>
    </div>
  </body>
</html>
```

Das meiste des hier gezeigten HTML-Codes ist lediglich für die Darstellung der Objekte und die Animationen im Hintergrund zuständig.

Für uns sind lediglich folgende Zeilen wichtig, in denen ein Dropdownmenü zur Auswahl der Animationen erstellt wird:

```
<form class="form-group" method="POST" action="/" role="form">
  <h3 class="mb-3" id="id1">LED Pattern auswahl:</h3>
  <div class="input-group">
    <select class="custom-select" id="inputGroupSelect04" name="light">
      <option selected>Animation auswahl ...</option>
      <option value="myValue">MyName</option>

    </select>
    <div class="input-group-append">
      <button class="btn btn-dark" type="submit">auswahl</button>
    </div>
  </div>
</form>
```

Eine HTML-Form erfüllt den Zweck, User-Eingaben entgegenzunehmen und an den Server zur Weiterverarbeitung zu senden. Dazu muss mit "method" angegeben werden, mit welcher Methode (hier: Post) gearbeitet wird und mit "action", welcher Route-Handler verwendet werden soll. Die Form beginnt mit dem öffnenden HTML-Tag:

```
<form class="form-group" method="POST" action="/">
```

und endet mit dem schließenden Tag:

```
</form>
```

Direkt unter dem öffnenden form-Tag befindet sich das h3-Tag. Hier wird eine Überschrift angegeben. Dies ist allerdings optional. Die Formatierung von h3 wird in der Datei style.css festgelegt.

```
<h3 class="mb-3" id="id1">LED Pattern auswahl:</h3>
```

h3 steht für "header 3" - also Überschrift Nr. 3. Wobei mit der Numerierung die Rangfolge angegeben wird. h1 ist in der Regel sehr groß. Die Größe ändert sich mit der Rangfolge.

Um die Daten an den Server zu senden, wird ein Button verwendet:

```
<div class="input-group-append">
  <button class="btn btn-dark" type="submit" >auswahl</button>
</div>
```

Welche Werte an den Server gesendet werden, werden durch die Optionen **name** und **value** definiert. Diese werden dann als Dictionary { name:value} an den Server übergeben

```
<select class="custom-select" id="inputGroupSelect04" name="light">
  <option selected>Animation ausw&uuml;hlen ...</option>
  <option value="animationName">MyName</option>
</select>
```

So wird ein Dropdown-Menü erstellt. Im öffnenden select-Tag wird der name (light) des zu übergebenden Werts bestimmt.

Innerhalb des select-Tags werden die options angegeben, also die Werte, die dann mittels dem Dropdown-Menüs ausgewählt werden können. Hier wird auch der zu übergebende value definiert.

Nun müssen wir nur noch unsere Animationen einfügen.

Für jede unserer Animationen müssen wir nun zwischen <option selected> und </select> einen eigenen option value eintragen:

```
<option value="running_dot">Running Dot</option>
```

value="running_dot" ist der Wert, der nach dem Absenden vom Server weiter verarbeitet wird. Ihn werden wir nutzen, um die richtige Animation zu starten. Hier ist es wichtig, dass der Eintrag in value derselbe ist, den wir im Dictionary animation_dict in animations.py angegeben haben! andernfalls werden wir die Animation nicht starten können.

Running Dot - was vor dem schließenden </option>-Tag steht ist lediglich der Text, der im Dropdown Menü angezeigt wird..

Die "Hieroglyphen" **ä** werden zur Darstellung von Umlauten benötigt, da nicht jeder Browser Umlaute in Form von 'ä', 'ö', 'ü' darstellen kann.

Hier nutzt man stattdessen **ä** (für ä), **ö** (für ö) und **ü** (für ü).

Eingaben auf dem Server verarbeiten

Nun wissen wir, dass für die erste Darstellung der Seite die GET-Methode, für das Absenden der Formulardaten die POST-Methode zum Einsatz kommen.

Das bedeutet wir brauchen zwei Rout-Handler-Methoden in webSrv.py.

Der Route-Handler für GET muss also vor der Definition folgende Zeile enthalten:

```
@MicroWebSrv.route('/')
```

Der Route-Handler für POST wird mit folgender Zeile eingeleitet:

```
@MicroWebSrv.route('/', 'POST')
```

für jeden Request brauchen wir eine eigene Funktion, die entweder nur die Seite als Antwort sendet, oder die Eingabedaten weiter verarbeitet bevor die Seite als Antwort gesendet wird. Leider können wir - obwohl beide Funktionen teilweise selben Inhalt haben, diese beiden nicht kombinieren.

Für den GET-Request reicht es also aus, die Seite als Antwort zu senden:

```
@MicroWebSrv.route('/')
def _routeHandlerGet(httpClient, httpResponse):
    httpResponse.WriteResponseFile(filepath = 'www/index.html',
                                   contentType= "text/html",
                                   headers = None)
```

Beim POST-Request muss der Server zusätzlich die eingehenden Daten abgerufen werden.

Ein Blick in die Modulbeschreibung verrät uns, wie wir an die FormData herankommen können:

```
@MicroWebSrv.route('/', 'POST')
def _routeHandlerGet(httpClient, httpResponse):
    formData = httpClient.ReadRequestPostedFormData()

    httpResponse.WriteResponseFile(filepath = 'www/index.html',
                                   contentType= "text/html",
                                   headers = None)
```

Nun müssen wir diese empfangene Daten an main.py senden.

Wie bereits erwähnt, läuft der WebServer in einem eigenen Thread. So wie auch der MainThread ein eigener Thread ist. Auch unsere Animationen laufen in einem eigenen Thread.

Wir könnten jetzt - wie bei `animations.set_brightness` auch sagen: wir schreiben eine Funktion namens `get_values` in `webSrv.py`, die wir von `main.py` aus aufrufen.

Dies würde uns allerdings immense Probleme bereiten, wenn main.py auf die Variable zugreift, während sie zur gleichen Zeit von webSrv.py geändert wird (und dies kommt häufiger vor als man denkt!)

Daher ist hier der Einsatz von Thread-Messages die sicherere Wahl. Sobald neue Daten vorliegen, kriegt main.py eine Benachrichtigung und holt sie dann erst ab:

```
if "light" in formData:  
    _thread.sendmsg(_thread.getReplID(), "light:  
{}".format(formData["light"]))
```

if "light" in formData prüft, ob die Zeichenfolge "light" in formData zu finden ist, um fehlerhafte Verarbeitung auszuschließen.

_thread.sendmsg(thread_id, message) sendet eine Nachricht als Zeichenfolge an den Thread, der mit thread_id angegeben wurde.

_thread.getReplID gibt die ID des MainThreads zurück. Achtung! Import des _thread-Moduls nicht vergessen!

"light:{}".format(value) konvertiert den Wert in value in einen String und setzt ihn in den Platzhalter {} ein.

Der Doppelpunkt hinter "light" wird später bei der Verarbeitung in main.py noch wichtig sein! Ebenso ist wichtig, dass sich weder vor noch nach dem Doppelpunkt Leerzeichen befinden!

Thread-Messages vom Server in main.py empfangen und verarbeiten

Um im MainThread überhaupt Messages empfangen zu können, müssen wir dies in main.py zuerst explizit mit **_thread.ReplAcceptMsg(True)** erlauben!

msg = _thread.getmsg() speichert - wenn vorhanden - empfangene Nachrichten in msg.

msg enthält dann ein Tupel (message_type, sender_id, message), wobei **message_type** 0 (=none), 1 (=integer) oder 2 (=String) sein kann.

sender_id ist die Thread-ID des Senders (wobei hier nur der Server infrage kommt)

message ist die Nachricht, die wir vom Server aus geschickt haben, also "light:formData["light"].

Um an den Wert von formData["light"] zu kommen, müssen wir also msg[2] an der Stelle ":" splitten.

values = msg[2].split(":") erledigt das für uns, sodass values dann aus ("light", formData["light"]) besteht.

Doch sollten wir - bevor wir mit dem Wert in formData["light"] die Animation starten noch prüfen, ob wir mit dem Wert wirklich die Animationen steuern. Jedoch ist hier vorsicht geboten: viel zu oft kommt es aus Gewohnheit vor, dass sich vor oder nach dem Doppelpunkt Leerzeichen

einschleichen. Diese kann man aber einfach abtrennen.
values[0].strip() entfernt Leerzeichen, die zu Beginn oder am Ende des Strings vorkommen.

Die Stringverarbeitung in Python ist ein sehr komplexes Thema, weshalb ich mich hier auf die Beschreibung der notwendigsten Funktionen beschränke. Weitere Infos kann man zum Beispiel hier finden:
<https://docs.python.org/2/library/string.html>

Auch hier sollten wir vor dem Start einer neuen Animation prüfen, ob bereits eine Animation im Thread läuft. Außerdem ist es zwingend notwendig, vor Start des neuen Threads dem WebServer einige Zeit (2000 ms) zu geben, dass dieser die Response-Seite dem Client (also unserem Computer) zur Verfügung stellen kann, andernfalls führt dies zu kritischen Fehlern. Das bedeutet, das System stürzt ab und startet neu - die Animation kann also nicht gestartet werden

Zusammengefasst sieht das dann so aus:

```
_thread.ReplAcceptMsg(True)
wait_before_start_thread = 2000

# ...

while True:
    # ...
    msg = _thread.getmsg()
    if msg[0] == 2:      #true if msg is a String
        values = msg[2].split(":")
        if values[0].strip() == "light":
            if _thread.status(animation_thread) !=
_thread.TERMINATED:
                _thread.notify(animation_thread, _thread.EXIT)
                utime.sleep_ms(wait_before_start_thread)
                _thread.start_new_thread("animation", animations.start,
(values[1].strip(), ))
```

Animationen mit einem String starten

Nun haben wir wieder mit animations.start(String) wieder eine Funktion in animatons angedeutet, die noch nicht vorhanden ist. Also wechseln wir in animations.py und erstellen unter der Funktion next() die Funktion start(animation_name).

In dieser Funktion soll die Animation gestartet werden, die als String in animation_name übergeben wurde.

Aus diesem Grund haben wir, um unsere Animationen zu speichern, einen Dict angelegt anstelle einer Liste.

Nun können wir einfach mit dem Schlüsselwort, gespeichert in animation_name, die Animation starten:

```
def start(animation_name):
    animation_dict[animation_name]()
```

Zwei ähnliche Routinen in einer Funktion zusammenfassen

Nun haben wir zwei Routinen in main.py, die im Prinzip ähnliches tun:

Wenn eine Animation im Thread läuft wird diese gestoppt, nach einem kurzen Delay dann eine neue im Thread gestartet. Dies geschieht entweder mit animations.next oder mit animations.start.

Diese sollten wir in einer Funktion - handleAnimations - zusammenfassen.

Nachdem der animation_thread gestoppt wurde, müssen wir zunächst prüfen, ob der Funktion ein Animations-name übergeben wurde (falls die Animation auf der Website ausgewählt wurde) oder nicht (falls sie durch den Touch-Sensor gewechselt wurde).

In Abhängigkeit dessen müssen wir nun den Wert von sleep_ms wählen und die Animation mit der richtigen Funktion in animations.py (next oder start) starten.

```
def handleAnimations(animation_name=None):  
    _thread.notify(animation_thread, _thread.EXIT)  
    if animation_name == None:  
        utime.sleep_ms(50)  
        animation_thread = _thread.start_new_thread("animation",  
            animations.next, ())  
    else:  
        print("starting animation {}".format(animation_name))  
        utime.sleep_ms(wait_before_start_thread)  
        animation_thread = _thread.start_new_thread("animation",  
            animations.start, (animation_name,))
```

Nun können wir entsprechende Zeilen in der Touch Sensor-Verarbeitung ersetzen durch:

```
handleAnimations()
```

in der Verarbeitung der Server-Messages durch

```
handleAnimations(values[1])
```

Der Piezo-Speaker

Der Piezo-Speaker ist - einfach ausgedrückt - ein kleiner Lautsprecher, der im Prinzip durch elektrische Signale (deren Frequenz ähnlich aufgebaut ist wie unsere hörbaren Schallwellen) seine Membran zum Schwingen bringt, wodurch ein akustisches Signal entsteht. Er verwandelt Strom in Ton - wie jeder andere Lautsprecher.

https://de.wikipedia.org/wiki/Ferroelektrischer_Lautsprecher

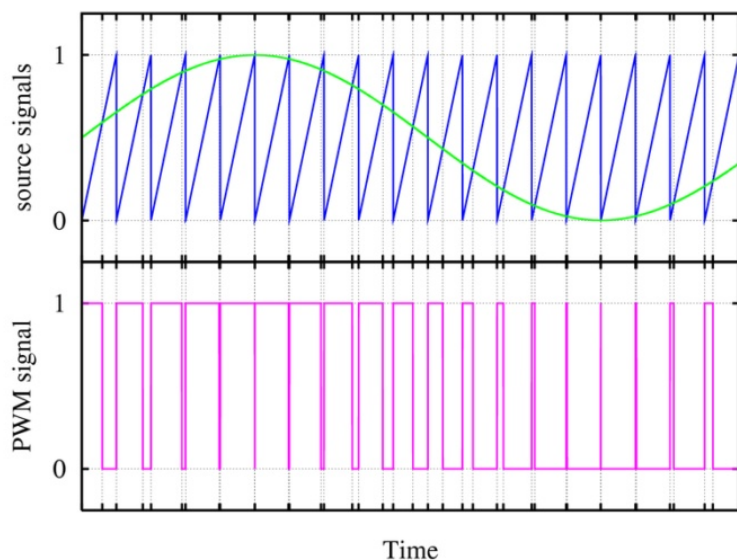
PWM - Pulsweitenmodulation (Pulse-Width-Modulation)

Ein PWM ist im Prinzip ein Digital Analog Converter, macht also genau das Gegenteil eines ADCs, den wir für den LDR eingesetzt haben.

Leider finde ich keinen Weg, die Arbeitsweise eines PWM in einfachen Worten zu beschreiben, jedoch veranschaulicht diese Grafik aus Wikipedia (<https://de.wikipedia.org/wiki/Pulsweitenmodulation>) die Übersetzung von einem digitalen Signal (das ja wie bekannt nur zwischen "ein" und "aus", bzw. 1 und 0 unterscheiden kann) eine analoge Sinuswelle werden kann - oder eben anders herum.

Wobei man beim Verwandeln eines digitalen in ein analoges Signal nicht mehr wirklich von "Welle" sprechen kann, da diese "Welle" doch irgendwie eckig wird

Auf dem Bild kann man oben die Sinuswelle, unten das Digitalsignal erkennen. Das Sägezahnformige Signal, das über der Sinuswelle liegt, kann man als "Übersetzer" betrachten. So gibt der x-Abstand zwischen den Schnittpunkten zwischen Sinuswelle und Sägezahnsignal vor, wie lange das Digitalsignal auf "ein" stehen muss.



Das PWM-Modul

https://github.com/loboris/MicroPython_ESP32_psRAM_LoBo/wiki/pwm

```
pwm = machine.PWM(pin [, freq=f] [, duty=d] [, timer=tm])
```

erstellt eine Instanz des PWM, wobei

pin die Pin-Nummer ist, an der .. anliegt. Der Pin kann entweder als

machine.Pin, oder als Integer-Wert übergeben werden.

freq (optional) bezeichnet die Frequenz in Hz, auf der der Piezo beim Start schwingen soll (weshalb es ratsam ist, den Piezo erst zu initialisieren, wenn er auch was abspielen soll). Default: 5 kHz

duty (optional) ist der Arbeitszyklus in %. Default: 50%

timer (optional) gibt an, ob ein Zeitmesser aktiviert werden soll (default: 0)

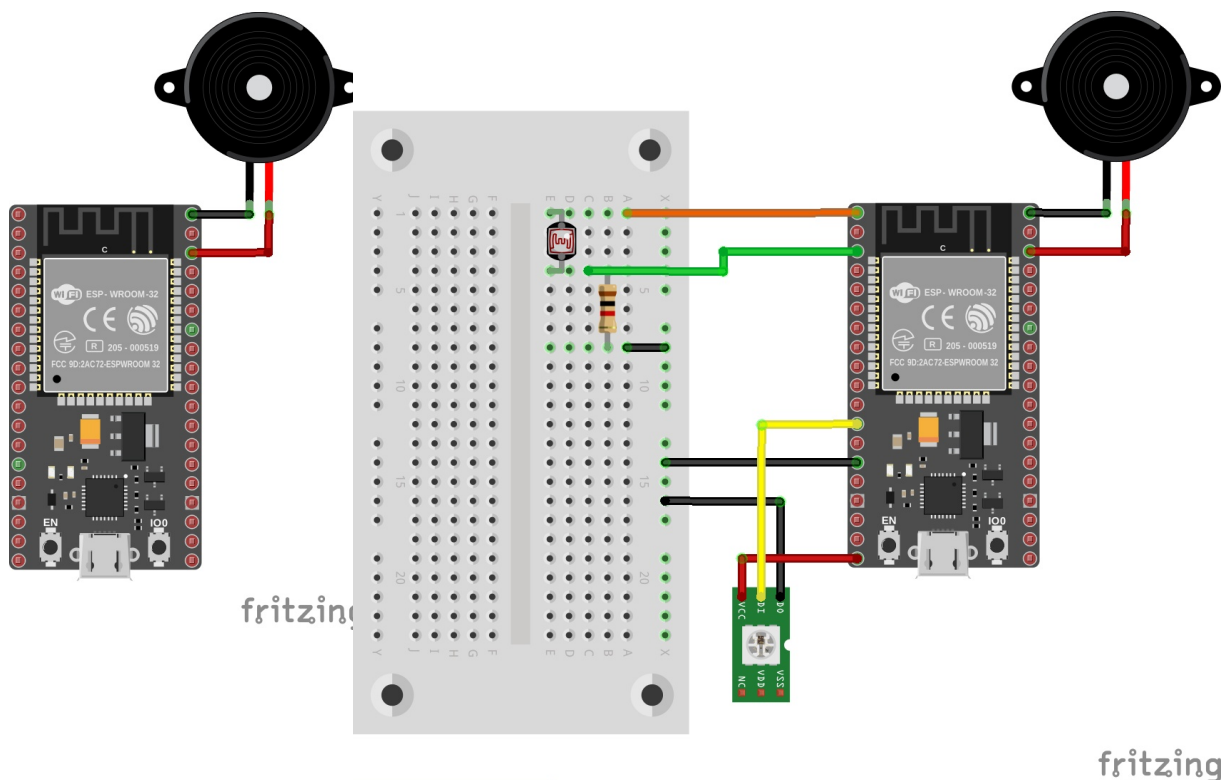
pwm.freq([freq]) setzt die Frequenz in Hz. Dies hat Auswirkungen auf alle Kanäle, die den selben Timer nutzen. Wenn kein Argument übergeben wird, gibt diese Funktion die Frequenz zurück, die auf diesem Kanal gespielt wird.

pwm.deinit() deinitialisiert den pwm-Pin, Ausgabe auf diesem Pin wird gestoppt.

pwm.init([freq=f] [, duty=d] [, timer=tm]) reinitialisiert pwm.

Piezo-Speaker mit dem ESP verbinden

Um den Piezo-Speaker mit dem ESP zu verbinden, muss man lediglich Vin mit Pin 22 verbinden, sowie GND mit einem GND-Pin am ESP (alternativ mit der GND-Reihe auf dem Breadboard)



Das RTTTL-Modul

RTTTL steht für Ring Tone Text Transfer Language und ist ein verbreitetes Format für Klingeltöne auf Mobiltelefonen. Hier ein Beispiel für den Song "One More Time":

```
'OneMoreT:d=16,o=5,b=125:4e,4e,4e,4e,4e,4e,8p,4d#.,4e,4e,4e,4e,4e,4e,8p,4d#.,4e,4e,4e,4e,4e,4e,8p,4d#.,4f#,4f#,4f#,4f#,4f#,4f#,8f#,4d#.,4e,4e,4e,4e,4e,4e,8p,4d#.,4e,4e,4e,4e,4e,4e,8p,4d#.,1f#,2f#'
```

Der Klingelton muss drei Teile beinhalten, um vom Klingeltonprogramm (oder hier des RTTTL-Moduls) erkannt zu werden:

1. Name des Klingeltons (hier: 'OneMoreT')
2. Standardvorgaben (d=16,o=5,b=125), wobei "d=" die Standardnotenlänge ist. "4" bedeutet, dass jede Note ohne Längenangabe eine Viertelnote ist. "8" würde eine Achtelnote bedeuten usw. "o=" die Standardoktave. Es gibt vier Oktaven im RTTTL-Format. "b=" steht für die Abspielgeschwindigkeit in Schlägen pro Minute.
3. Die Tonangaben. Sie sind durch Komma voneinander getrennt und beinhalten:
 1. (optional) Tonlänge
 2. Notenangabe (c, d e, f, g, a, b, oder p = Pause). Eine Raute hinter der Tonangabe erhöht um einen Halbton.
 3. (optional) Oktave

Das RTTTL-Modul für MicroPython kann hier heruntergeladen werden, wobei wir lediglich Datei "rtttl.py" brauchen werden :
<https://github.com/dhylands/upy-rtttl>

Dieser sollte am besten im Projektordner, wo auch alle bisher erstellten *.py-Files liegen, gespeichert werden.

Wie wir beim Betrachten des Dateisystems festgestellt haben, ist der Speicherort für weitere Module auf dem ESP der Ordner "lib". Also müssen wir einen Ordner namens "lib" erstellen und die Datei rtttl.py in diesen kopieren.

```
mkdir /flash/lib  
cp rtttl.py /flash/lib
```

Weitere Beispielsongs findet man hier:

<http://www.picaxe.com/RTTTL-Ringtones-for-Tune-Command/>

<http://mines.lumpyumpy.com/Electronics/Computers/Software/Cpp/MFC/RingTones.RTTTL>

Songs mit dem PWM abspielen

Um die Songs zu speichern und abzuspielen, erstellen wir eine neue Datei namens songs.py.

Wir müssen nicht immer das komplette machine-Modul einbinden, wenn wir lediglich Pin und PWM daraus nutzen werden. Ebenso benötigen wir aus rtttl lediglich die Klasse RTTTL.

Außerdem wird noch das `_thread`-Modul importiert, da auch das Abspielen der Songs in einem Thread laufen wird. Außerdem können wir gleich den `piezo_pin` (22) definieren und PWM kurz initialisieren, um ihn sofort wieder zu deinitialisieren. Dies geschieht aus dem Grund, dass unsere PWM-Instanz auch von beiden Funktionen aus verändert werden können. Zudem benötigen wir noch eine Variable, die auf True gesetzt wird, sobald eine Benachrichtigung zum Abbruch des Threads kommt:

```
from machine import Pin, PWM
from rtttl import RTTTL
import _thread

piezo_pin = 22
piezo = PWM(piezo_pin)
piezo.deinit()
abort_playback = False
```

Unsere Songs speichern wir in einer Liste namens SONGS. Wie bereits bekannt müssen Elemente einer Liste durch Komma getrennt werden. Aufgrund der Länge der verwendeten Strings und der Tatsache, dass sie viel zu lang sind, um auf dieser Seite einzellig dargestellt zu werden, kann dies gerne übersehen werden:

```
SONGS = [
    'OneMoreT:d=16,o=5,b=125:4e,4e,4e,4e,4e,4e,8p,4d#.,4e,4e,4e,4e,4e,4e,8p,4d#.,4e,4e,4e,4e,4e,4e,8p,4d#.,4f#,4f#,4f#,4f#,4f#,4f#,8f#,4d#.,4e,4e,4e,4e,4e,4e,8p,4d#.,4e,4e,4e,4e,4e,4e,8p,4d#.,1f#,2f#',
    'MarioTitle:d=4,o=5,b=125:8d7,8d7,8d7,8d6,8d7,8d7,8d7,8d6,2d#7,8d7,p,32p,d6,8b6,8b6,8b6,8d6,8b6,8b6,8b6,8d6,8b6,8b6,8b6,16b6,16c7,b6,8a6,8d6,8a6,8a6,8a6,8d6,8a6,8a6,8a6,8d6,8a6,8a6,8a6,16a6,16b6,a6,8g6,8d6,8b6,8b6,8b6,8d6,8b6,8b6,8b6,8d6,8b6,8b6,8b6,16a6,16b6,c7,e7,8d7,8d7,8d7,8d6,8c7,8c7,8c7,8f#6,2g6',
    'Tetris:d=4,o=5,b=160:e6,8b,8c6,8d6,16e6,16d6,8c6,8b,a8a,c6,e6,8d6,8c6,b,8b,8c6,d6,e6,c6,a,2a,8p,d6,8f6,a6,8g6,8f6,e6,8e6,8c6,e6,8d6,8c6,b,8b,8c6,d6,e6,c6,a,a'
]
```

Wir werden insgesamt drei Funktionen benötigen, wovon nur eine vom MainThread aus aufgerufen wird.

Die vom MainThread aufzurufende Funktion wird aus SONGS den angeforderten Song wählen und in einer separaten Variable speichern. Anschließend wird mit dieser Variable als Argument eine Instanz von RTTTL erstellt. Nun wird für jede Note, die in diesem Song vorhanden ist, die nächste Funktion `play_tone` aufgerufen, die die Note dann abspielt. In

dieser Schleife sollte auch geprüft werden, ob der Thread abgebrochen werden soll:

```
def find_song(name):
    """ find_song(name)
    searches in SONGS list for song name and plays
    its tones with the play_tone function
    name: name of the song to search for
    """
    global abort_playback
    abort_playback = False

    for song in SONGS:
        song_name = song.split(":")[0]
        if song_name == name:
            tune = RTTTL(song)

            for freq, msec in tune.notes():
                play_tone(freq, msec)
                if abort_playback:
                    return
            piezo.deinit()
```

Zwei Ausdrücke, die wir - zumindest so - noch nicht besprochen haben:

`song_name = song.split(":")[0]` - splittet den String an der Stelle, an der der Doppelpunkt vorkommt. Dies funktioniert natürlich ebenso, wenn mehr als ein Doppelpunkt im String vorhanden ist. [0] beinhaltet dann den Namen, [1] die Standardvorgaben und [2] die Tonangaben.

`for freq, msec in tune.notes()` deutet darauf hin, dass von der Funktion `tune.notes()` zwei Objekte zurückgegeben werden: `freq` für die Frequenz und `msec` für die Dauer in ms.

somit kann man mit dieser for-Schleife zwei Werte parallel verarbeiten. `abort_playback` ist - wie der Name schon sagt, eine Variable, in der festgehalten wird, ob die Wiedergabe abgebrochen werden soll oder nicht. Dies wird in `play_tone` geprüft

`play_tone(freq, ms)` muss vor der Funktion `find_song()` platziert werden. Hier wird für jede einzelne Note ein Ton mittels PWM erzeugt. Anstelle von `utime.sleep_ms(ms)` können wir - analog zu `animations.py` - `waitForExitNotification(ms)` einsetzen, die im darauffolgenden Schritt erstellt wird:

```
def play_tone(freq, msec):
    """play_tone(freq, msec)
    plays a single tone on piezo buzzer
    freq: frequency of the tone
    msec: duration in millis
    """
    global piezo
    global abort_playback
    print('freq = {:.1f} msec = {:.1f}'.format(freq, msec))
    if freq > 0:
        piezo.freq(int(freq))
        piezo.duty(50)
```

```

ntf0 =waitForExitNotification(int(msec *0.9))
piezo.duty(0)
ntf1 = waitForExitNotification(int(msec * 0.1))
if (ntf0 or ntf1):
    abort_playback = True
    piezo.deinit()

```

Hier wird der Wert von ms gesplittet: zuerst wird gewartet, bis 90% des ms-Wertes vergangen sind, um dann duty auf 0 zu setzen. Anschließend wird die letzten 10% von ms abgewartet. Um nicht versehentlich eine Benachrichtigung zu überschreiben, werden diese in zwei verschiedene Variablen gespeichert.

ntf0 or ntf1 ist eine logische Operation. Sie gibt genau dann True zurück, wenn mindestens eine der beiden Operatoren (ntf0 oder ntf1) True ist. Somit wird auch keine Abbruchbedingung übersehen.

Zuletzt benötigen wir noch eine Funktion, die die Abbruchbedingung prüft:

```

def waitForExitNotification(timeout):
    """ waitForExitNotification(timeout)
    uses _thread.wait(timeout) to sleep for amount of ms saved in
    timeout,
    checks notification for _thread.EXIT notification. Returns True if
    _thread.EXIT notification is recieved, else returns False

    timeout: (ms) time in ms used for _thread.wait()
    return: (boolean) True if _thread.EXIT is recieved, else False.
    """
    ntf = _thread.wait(timeout)

    if ntf == _thread.EXIT:
        return True
    return False

```

return False muss hier nicht zwingend in einem else-Block stehen. Das return-Statement im if-Block bewirkt, dass diese Funktion mit dieser Zeile verlassen wird. Wenn also die Bedingung (ntf == _thread.EXIT) zutrifft, wird return False gar nicht mehr ausgeführt.

Dropdown Menü für Songs in die Webseite einfügen

Um den Song auf der Webseite auswählen zu können verfahren wir analog wie mit den LED Animationen.

Um uns arbeit zu sparen, können wir den gesamten Bereich innerhalb der form-Tags kopieren und unterhalb des kopierten Bereichs wieder einfügen.

Um etwas mehr Abstand zwischen den beiden Menüs zu schaffen, sollte man `</br>`-Tags einfügen. Diese bewirken einen Zeilenumsprung. Hier darf gerne getestet werden, was optisch akzeptabel ist.

h3 sollten wir nun selbstverständlich anpassen: Song auswählen
passt hier besser!

als nächstes muss im **select**-Tag der **name** angepasst werden:
name="song"

Nun müssen wir noch unsere Songs ins Dropdown Menü einfügen.

Hier sollten wir - wie auch bei den LED Animationen - darauf achten, dass unter **value** der Name steht, der in der SONGS-Liste in songs.py zu finden ist. Dies spart uns unnötige Abfragen. Der Name, der auf der Seite gezeigt werden soll, ist dafür immernoch frei wählbar.

Hier nun der neu eingefügte Part in index.html:

```
</br>
</br>
<h3 class="mb-3" id="id1">Song ausw hlen:</h3>
  <div class="input-group">
    <select class="custom-select" id="inputGroupSelect04" name="sound">
      <option selected>Song ausw hlen...</option>
      <option value="OneMoreT">One More Time</option>
      <option value="MarioTitle">Super Mario Titelsong</option>
      <option value="Tetris">Tetris Theme</option>
    </select>
    <div class="input-group-append">
      <button class="btn btn-outline-secondary" type="button">Button</
button>
    </div>
  </div>
```

 brigens: man muss die HTML-Seite nicht zwingend zuerst auf den ESP kopieren um sie anschauen zu k nnen. Da sie sich gemeinsam mit ihren zugeh rigen CSS-Seiten im Ordner www befindet, kann man diese einfach mit einem Doppelklick im Browser  ffnen.

Verarbeitung der Daten auf dem Webserver

Auch hier kann man analog zu den Animationen verfahren:

in webSrv.py sollte nun nicht nur nach "light" gepr ft werden, sondern auch nach "song".

Oder aber man verzichtet auf diesen Schritt und  berl sst diese Unterscheidung dem MainThread!

Dazu kann die Zeile:

```
if "light" in formData:
```

ersetzt werden durch:

```
if formData:
```

Das bedeutet, dass die Thread-Message nur gesendet wird, wenn Daten empfangen wurden.

Thread-Message sollte entweder aus einem String, oder einem Integer-Wert bestehen. Ein Dictionary ist allerdings weder das eine, noch das andere!

Dafür kann man einen Dictionary sehr einfach mit `str(dict)` in einen String verwandeln! und einen String (sofern er den Aufbau eines Dictionarys hat) ebenso einfach mit `eval(str)` wieder in einen Dictionary!

```
if formData:
    message = str(formData)
    _thread.sendmsg(_thread.getReplID(), message)
```

konvertiert nun den vom webServer empfangenen Dictionary in einen String und sendet diesen an den MainThread.

In `main.py` lassen wir uns nun ausgeben, was der Server uns geschickt hat:

```
msg = _thread.getmsg()
if msg[1] == 2:
    print(msg[2])
```

```
>>> {'light': 'crossing_dots', 'sound': 'Song+auswählen+...'}
```

Oh!

Dies bedeutet, dass nun - unabhängig davon, welcher Button von beiden gedrückt wurde, beide Werte übergeben werden!

Dies liegt wohl daran, dass sich beide innerhalb einer Form befinden!

Man kann nun entweder eine separate Form für jedes Menü erstellen - oder aber wir versehen die Buttons mit `name` und `value`, wodurch der angeklickte Button diese Werte ebensovorgibt. Anhand diesen wir entscheiden, wie mit diesen Daten weiter verfahren wird.

Für beide Buttons setzen wir also **name="button"** ein. so können wir später einfach den Wert prüfen, der an der Stelle "button" ist.

Als **value** setzen wir **"light"** bei den Animationen und **"sound"** bei den songs. Dann sieht der Output so aus:

```
>>> {'light': 'crossing_dots', 'sound': 'Song+auswählen+...', 'button': 'light'}
```

Verarbeitung der Daten in main

Nun haben wir uns eine Routine gebaut, die sehr Variabel mit allen Möglichen Inputs umgehen kann. Es fehlt lediglich noch die Verarbeitung in `main.py`.

`_thread.getmsg()[2]` liefert uns einen Dictionary als String, den wir nun wieder in einen Dictionary umwandeln müssen. Anschließend müssen wir prüfen, ob der Button mit dem value "light" gedrückt wurde. dann können wir `handleAnimationThread` mit dem Wert, der in "light" steht starten:

```
# read threadMessage from microWebSrv
msg = _thread.getmsg()
# check if message is String
if msg[0] == 2:
    # convert String in dictionary
    values_dict = eval(msg[2])
    # call function according to value of "button"
    if values_dict["button"] == "light:
        handleAnimationThread(values_dict["light"])
```

Analog verfahren wir mit "sound":

```
elif values_dict["button"] == "sound":
    handleSoundThread(values_dict["sound"])
```

Ohne die Funktion handleSoundThread(song) zu definieren, können wir allerdings nicht fortfahren:

```
sound_thread = 100
def handleSoundThread(song):
    """ stops running sound thread and starts a new thread with value
    saved
    in 'song'

    song: (String) name of the song to start
    """
    global sound_thread
    _thread.notify(sound_thread, _thread.EXIT)

    sound_thread = _thread.start_new_thread("sound", songs.find_song,
    (song, ))
```

Fertig für den Test!

Wenn alles fehlerfrei eingefügt und auf den ESP kopiert wurde, kann man nun nach einem Neustart den Song von der Webseite aus abspielen lassen.

Leider aber stößt der ESP an seine Grenzen, wenn man rechenintensive Animationen zeitgleich mit einem Song laufen lässt. In diesem Fall ist es möglich, dass Song und Animation merkbar langsamer abgespielt werden.

RainbowWarrior als Wecker nutzen

Nun haben wir gelernt, wie man den LED Strip ansteuert sowie mit dem Piezo Speaker Songs abspielt. Selbstverständlich lassen sich diese beiden Elemente auch kombinieren um sie als Wecker einzusetzen.

FreeRTOS ist ein Real Time Operating System. Das bedeutet, man kann mit dem ESP auch zeitgesteuerte Operationen ausführen. Die Klasse `machine.RTC` stellt uns eine Real Time Clock zur Verfügung, mit der wir Datum- und Uhrzeit einstellen können. Sowie uns das Modul `utime` Zeit-abhängige Funktionen zur Verfügung stellt

Real Time Clock - das RTC Modul

Mit dem RTC-Modul können wir die Systemzeit des ESP einstellen. Zudem haben wir die Möglichkeit - sofern wir im Station-Mode mit dem Internet verbunden sind - die Uhrzeit im Internet zu synchronisieren (so, wie es auch unsere Mobiltelefone machen).

Außerdem können wir den ESP in den sogenannten "deepsleep"-Modus versetzen. Das bedeutet, dass viele Funktionen heruntergefahren werden und somit weniger Strom verbraucht wird. Selbstverständlich stellt es auch Möglichkeiten zur Verfügung, den ESP wieder aus dem Tiefschlaf zu wecken!

Die Modulbeschreibung ist hier zu finden:

https://github.com/loboris/MicroPython_ESP32_psRAM_LoBo/wiki/rtc

Die wichtigsten Funktionen, um den ESP als Wecker zu nutzen, sind:

`rtc = machine.RTC()` erstellt eine Instanz der RTC

`rtc.init(date)` lässt uns die Systemzeit einstellen, wobei für `date` ein Tupel erwartet wird: (year, month, day [,hour [,minute [, second]]])

`rtc.now()` gibt uns die aktuelle Zeit als Tupel zurück: (year, month, day, hour, minute, second)

Um die Zeit im Internet zu synchronisieren sollte vorab eine Prüfung stattfinden, ob der ESP im Station-Mode verbunden ist.

`rtc.ntp_sync(server [,update_period] [,tz])` wobei als Server "de.pool.ntp.org" genutzt werden kann.

Zeitabhängige Funktionen - das utime-Modul

Das utime-Modul stellt ebenso Funktionen zur Verfügung, mit denen man die aktuelle Zeit ausgeben kann. Außerdem Zeitintervalle messen und Delays ausführen.

<https://docs.micropython.org/en/latest/pyboard/library/utime.html>

utime.localtime([secs]) konvertiert die Zeit, angegeben in Sekunden (secs) in ein Tupel aus 8 Elementen (year, month, mday, hour, minute, second, weekday, yearday). Wenn kein Argument übergeben wird, wird die aktuelle Systemzeit genutzt.

year - als Jahrtausendangabe (z.B. 2014).

month (1-12)

mday (1-31)

hour (0-23)

minute (0-59)

second (0-59)

weekday (0-6 für Mo-So)

yearday (1-366)

utime.mktime(time_tuple) gibt - Gegenteilig zu localtime - für ein vollständiges Tupel aus 8 Elementen (year, month, mday, hour, minute, second, weekday, yearday) die Anzahl der Sekunden zwischen dem 01.01.1970 und der dateTIme, die im Tupel übergeben wurde, zurück.

utime.sleep_ms(milliseconds) führt ein Delay für die Angegebene Zeit in Millisekunden aus

utime.sleep_us(microseconds) führt ein Delay für die Angegebene Zeit in Mikrosekunden aus

utime.ticks_ms() gibt eine steigende Zahl an Millisekunden zurück - der Referenzpunkt ist allerdings willkürlich gesetzt. Diese Funktion wird zum Beispiel eingesetzt, um ein Delay zu erstellen.

utime.ticks_add(ticks, delta) führt eine Addition mit Ticks aus, wobei ticks vorher mit ticks_ms(), ticks_us oder ticks_cpu - in Modulbeschreibung zu finden) erstellt werden muss. delta kann eine positive oder negative Integer sein)

utime.ticks_diff(ticks1, ticks2) gibt die Zeitdifferenz zwischen zwei ticks-Werten zurück.

Normale Addition bzw Subtraktion von ticks-Werten sollten nicht ausgeführt werden, da dies zu inkorrekten Ergebnissen führen kann.

utime.time() gibt die Anzahl der Sekunden zurück, die zwischen dem 01.01.1970 und der eingestellten Systemzeit vergangen sind.

Systemzeit auf der Webseite anzeigen

Um lediglich die aktuelle Zeit anzuzeigen reicht uns das `utime`-Modul. Nachdem dieses in `webSrv.py` eingebunden wurde, kann die aktuelle Systemzeit als Tupel in der Variable `datetime` gespeichert werden:

```
import utime  
  
datetime = utime.localtime()
```

Damit bei jedem Aufruf der Webseite die aktuelle Systemzeit genutzt wird, muss `datetime = utime.localtime()` innerhalb der Route-Handler Funktionen stehen.

Zur Erinnerung, das Tupel besteht aus 8 Elementen: (year, month, mday, hour, minute, second, weekday, yearday)
Wollen wir nun aus diesem Tupel z.B. eine Ausgabe am Terminal machen, die nur die Uhrzeit (Stunden und Minuten) getrennt durch einen Doppelpunkt - anzeigt, müssen wir uns zuerst etwas mit dem Thema String Formatierung beschäftigen:

String Formatierung

Bisher haben wir die `print`-Ausgabe nur genutzt, um entweder Integer-Variablen oder Texte auszugeben.

Man kann diese beiden Ausgaben aber auch kombinieren! Hier ein Beispiel für die Ausgabe mit Text und Variable anhand der `Print`-Ausgabe. Zuerst werden wir das Alter des Nutzers in `Repl` einlesen, anschließend einen Text inklusive dem eingegebenen Alter ausgeben.

```
>>> age = input("Alter?  ")
```

Dadurch erfolgt in `Repl` die Ausgabe 'Alter? ', bei der man dann das Alter (z.B. 22) eingeben kann. Dieses wird nach der Bestätigung mit `Enter` in der Variable `age` gespeichert:

```
Alter? 22
```

Nun wollen wir das Alter in einen Text einfügen und ausgeben:

```
>>> print("Alter: {} Jahre".format(age))  
Alter ist 22 Jahre
```

`String.format(var)` konvertiert dabei die Variable `var` in einen String und fügt sie in den Platzhalter `{}` ein.

Wollen wir nun wissen, um welchen Datentyp es sich bei "age" handelt, schreiben wir:

```
>>> type(age)  
<class 'str'>
```


Was bedeutet, dass es sich um einen String handelt.

Um mit dieser Eingabe berechnungen ausführen zu können, bräuchten wir allerdings einen Integer-Wert. mit `int(age)` lässt sich der String in `age` in einen Integer-Wert konvertieren. Dies geht ebenso direkt mit der Eingabe:

```
>>> age = int( input("Alter? "))
Alter? 22
>>> type(age)
<class 'int'
```

Arbeiten mit mehreren Platzhaltern

Nehmen wir an, wir haben Geschwister im Alter von 12, 14 und 18 Jahren (Alter gespeichert in `age1`, `age2`, `age3`) und wollen diese in einen Text einfügen, gehen wir folgendermaßen vor:

```
>>> print("Meine Geschwister sind {}, {} und {} Jahre alt".format(age1,
age2, age3))
Meine Geschwister sind 12, 14 und 18 Jahre alt
```

Wobei in die Platzhalter der Reihe nach die Werte in `format()` eingetragen werden.

Allerdings können wir auch die Reihenfolge der Ausgabe anpassen, ohne die Reihenfolge in `format()` zu verändern:

```
>>> print("Meine grosse Schwester ist {2} Jahre alt, mein kleiner Bruder
{0} und meine kleine Schwester {1}".format(age1, age2, age3))
Meine grosse Schwester ist 18 Jahre alt, mein kleiner Bruder 12 und meine
kleine Schwester 14
```

Nehmen wir nun an, wir haben eine Variable `time`, in der die Zeit als Tupel (`h`, `m`) gespeichert ist. Derzeit ist es kurz nach 9. Die Ausgabe der Uhrzeit in einem String würde dann folgendermaßen aussehen:

```
>>> print("Zeit: {}:{} Uhr".format(time[0], time[1]))
Zeit: 9:6 Uhr
```

Leider sieht dies schrecklich aus, da wir gewohnt sind, dass die Uhrzeit in dieser Darstellung immer zweistellig für Stunde und Minute ist. Diese Darstellung erreichen wir folgendermaßen:

```
>>> print("Uhrzeit: {0:02d}:{1:02d} Uhr".format(time[0], time[1]))
Uhrzeit: 09:06 Uhr
```

Wobei `02` bedeutet, dass die Ausgabe mindestens zweistellig sein muss, bei kleineren Werten soll eine `0` vorangestellt werden. Das `d` bedeutet, dass es sich bei diesem Wert um eine Dezimalzahl handelt. Hier ist die Position (`0:` oder `1:`) zwingend anzugeben!

Man kann bei der Ausgabe ebenso mit Schlüsselwortparametern arbeiten:

```
>>> print("die Hauptstadt von {province} ist {capital}.".format("province" :  
"Hessen", "capital" : "Wiesbaden"))  
die Hauptstadt von Hessen ist Wiesbaden.
```

Diese Angaben gelten nicht nur für die print-Ausgabe, sondern sind ebenso auch auf Strings anwendbar:

```
>>> month = "April"  
>>> text = "Mein Geburtsmonat ist {}"  
>>> text.format(month)  
>>> print(text)  
Mein Geburtsmonat ist April
```

Weitere Anwendungsbeispiele findet man im Python-Forum:
https://www.python-kurs.eu/python3_formatierte_ausgabe.php

Um also die Systemzeit auf der Webseite anzeigen zu können, müssen wir auf dieser erst einmal Platzhalter einfügen. Anschließend müssen wir dafür sorgen, dass uns die HTML-Datei als String in webSrv.py vorliegt, da wir format() nur auf einen String anwenden können. Zum Schluss müssen wir noch httpResponse unseres RouteHandlers anpassen.

Die Zeitausgabe auf der Webseite soll folgendermaßen formatiert werden:

22.09.2018 15:05 Uhr

Unser Tupel besteht aus year (0), month (1), mday (2), hour (3), minute (4), second (5), weekday (6), yearday (7)

Zur besseren Übersicht können wir nun die Zeit und das Datum in unserer gewohnten Reihenfolge in einen Dictionary in webSrv.py speichern:

```
datetime_dict = { "year" : datetime[0], "month" : datetime[1],  
                  "mday" : datetime[2], "hour": datetime[3], "minute" : datetime[4],  
                  "second" : datetime[5], "weekday" : datetime[6], "yearday" :  
datetime[7]}
```

Nun fügen wir in index.html die entsprechenden Platzhalter ein. Dies erreichen wir, indem wir einen neuen "paragrafen" mit <p> (text) </p> einfügen. Nur Text innerhalb eines Tags wird auch auf der Webseite dargestellt!

Noch vor dem ersten <form>-Tag fügen wir also zuerst einen öffnenden <p>-Tag, dann unsere Platzhalter inklusive Text, danach den schließenden Tag </p> ein:

```
<p>  
    {mday:02d}.{month:02d}.{year:04d} {hour:02d}:{minute:02d} Uhr  
</p>  
<form ...
```

bevor wir die korrekte Darstellung testen können, müssen wir in webSrv.html dafür sorgen, dass uns index.html als String vorliegt:

```
file = open("www/index.html")
htmlSite = file.read()
file.close()
```

Mit `file = open` wird die Seite, die mit dem Pfad (als String) übergeben wird, in `file` gespeichert. Dies geschieht zum Beispiel auch, wenn wir mit einem Texteditor eine Datei öffnen. Wie im Texteditor muss auch hier die Datei mit `file.close()` nach der Verarbeitung geschlossen werden. `file.read()` gibt den Inhalt der Seite als String zurück.

Nun sorgen wir dafür, dass beim Aufruf der Webseite die aktuelle Systemzeit in `datetime` gespeichert wird und erstellen daraus einen Dictionary. Dazu fügen wir in unsere RouteHandler-Funktion (`_httpHandlerPost`) folgende Zeilen ein. :

```
datetime = utime.localtime()
datetime_dict = { "year" : datetime[0], "month" : datetime[1], "mday" :
datetime[2], "hour": datetime[3], "minute" : datetime[4], "second" :
datetime[5], "weekday" : datetime[6], "yearday" : datetime[7]}
```

Nun können wir dafür sorgen, dass die Werte in die Platzhalter mit `String.format()` eingefügt werden können:

```
htmlSite.format(**datetime_dict)
```

Das doppelte Sternchen (`**`) vor `datetime_dict` sorgt dafür, dass `datetime_dict` in die Form `'year'= value, 'month'= value, ...` umgewandelt wird.

Nun müssen wir noch `httpResponse` anpassen. Also das, was bei einem Aufruf (=einer Anfrage) des Clients an den Server vom Server zurückgegeben wird.

Bisher wurde mit `httpResponse.writeResponseFile(.... path = "www/index.html")` die Datei `"www/index.html"` zurückgegeben. Dies müssen wir nun ändern, da wir keine Datei, sondern den String mit dem HTML-Code der Datei als Antwort senden wollen.

Hierfür nutzt man `writeResponseOk` und fügt anstatt `'path= ... '` `content = htmlFile` ein:

```
httpResponse.WriteResponseOk(headers = ({'Cache-Control': 'no-cache'}),
contentType = 'text/html', contentCharset = 'UTF-8', content =htmlFile)
```

Insgesamt sieht `webSrv.py` nun so aus:

```
from microWebSrv import MicroWebSrv
import _thread, utime
```

```

srv_run_in_thread = True

# open file "www/index.html" and save content as String in htmlSite and
# close file
file = open("www/index.html")
htmlSite = file.read()
file.close()

# route handler for index.html with method GET
@MicroWebSrv.route('/')
def _httpHandlerPost(httpClient, httpResponse) :
    # get system time and save it to datetime_dict
    datetime = utime.localtime()
    datetime_dict = { "year" : datetime[0], "month" : datetime[1],
                      "mday" : datetime[2], "hour": datetime[3],
                      "minute" : datetime[4], "second" : datetime[5],
                      "weekday" : datetime[6], "yearday" : datetime[7]}

    # insert datetime_dict values in placeholders in htmlSite
    htmlSite = htmlSite.format(**datetime_dict)
    # send htmlSite as response to client
    httpResponse.WriteResponseOk(headers = ({'Cache-Control': 'no-cache'}),
                                   contentType = 'text/html',
                                   contentCharset = 'UTF-8',
                                   content = htmlSite)

# route handler for index.html with method POST
@MicroWebSrv.route('/', 'POST')
def _httpHandlerPost(httpClient, httpResponse) :
    # read posted form data. If data (dictionary) received, convert to
    string
    # and send threadMessage to mainThread
    formData = httpClient.ReadRequestPostedFormData()
    print(formData)
    if formData:
        message = str(formData)
        _thread.sendmsg(_thread.getReplID(), message)

    # get system time and save it to datetime_dict
    datetime = utime.localtime()
    datetime_dict = { "year" : datetime[0], "month" : datetime[1],
                      "mday" : datetime[2], "hour": datetime[3],
                      "minute" : datetime[4], "second" : datetime[5],
                      "weekday" : datetime[6], "yearday" : datetime[7]}

    # insert datetime_dict values in placeholders in htmlSite
    htmlSite = htmlSite.format(**datetime_dict)
    # send htmlSite as response to client
    httpResponse.WriteResponseOk(headers = ({'Cache-Control': 'no-cache'}),
                                   contentType = 'text/html',
                                   contentCharset = 'UTF-8',
                                   content = htmlSite)

# create instance of MicroWebSrv and start server
srv = MicroWebSrv(webPath = 'www')
srv.Start(threaded = srv_run_in_thread, stackSize= 8192)

```

Nachdem alles auf dem ESP aktualisiert und neugestartet wurde, sollte nun auf der Webseite die Systemzeit angezeigt werden.

Systemzeit und Datum über die Webseite einstellen

Um die Systemzeit einstellen zu können, brauchen wir neben der Webform, über die die Zeit und das Datum eingegeben werden kann, in main.py eine Routine, die die Werte entgegennimmt und RTC initialisiert.

Also nehmen wir uns wieder die HTML-Seite. Die Webform platzieren wir direkt unter dem Abschnitt mit der Systemzeit. Hier sollen zwei Textfelder (für Datum und Uhrzeit) sowie ein Button erscheinen. Aus optischen Gründen sollen diese aber - ähnlich wie beim Dropdown-Menü und dem zugehörigen Button - in einer Reihe platziert werden. Daher ist bei class "form-inline" anzugeben.

```
<form class="form-inline" method="POST" action="/" role="form">
</form>
```

Die zwei Textfelder und der Button in der Form werden in eine <div>-Tag zusammengefasst

```
<div class="input-group">
</div>
```

Innherhalb dieses <div>-Tags fügen wir nun zuerst unsere Textfelder ein. Textfelder werden durch <input>-Tags erstellt. Zu jedem <input>-Tag gehört noch ein <label>, das vor <input> platziert werden soll. Dies ist wichtig, da manche Browser sonst Probleme mit der Darstellung haben. Wie auch beim Sound und den Animationen sind die wichtigsten Schlüsselwörter "name" und "value". Im erste <input>-Tag geben wir also bei name "date" ein, im zweiten "time". Als Value wird beim Absenden der Form die Nutzereingabe an der Stelle gesetzt. Optional können wir einen placeholder, also Platzhalter definieren. Hier können wir einfach die Platzhalter nutzen, die wir zur Anzeige der Systemzeit genutzt haben.

```
<label class="sr-only" for="inlineFormInputDate">Datum</label>
  <input type="text" name="date" class="form-control mb-2"
id="inlineFormInputDate" placeholder="{mday:02.02d}.{month:02.02d}.{year:
02.02d}" name="date">
  <label class="sr-only" for="inlineFormInputTime">Uhrzeit</label>
  <input type="text" name="time" class="form-control mb-2"
id="inlineFormInputTime" placeholder="{hour:02.02d}:{minute:02.02d}">
```

Fehlt nur noch der Button. Dieser muss zwischen den Textfeldern und dem schließenden <div>-Tag platziert und in einem eigenen <div>-Tag liegen. Auch hier sind die Schlüsselworte "name" und "value für die Weiterverarbeitung wichtig.

Wie auch bei Sound und Animationen werden wir zuerst prüfen, welcher Button gedrückt wird. Daher müssen wir bei name wieder "button" angeben, als value setzen wir "datetime"

```
<div class="input-group-append">
  <button type="submit" class="btn btn-dark mb-2" name="button"
```

```
value="datetime">Zeit einstellen</button>
</div>
```

Zusammengesetzt sieht das dann so aus:

```
<form class="form-inline" method="POST" action="/" role="form">
  <div class="input-group">
    <label class="sr-only" for="inlineFormInputDate">Datum</label>
    <input type="text" name="date" class="form-control mb-2"
id="inlineFormInputDate" placeholder="{mday:02.02d}.{month:02.02d}.{year:
02.02d}" name="date">
    <label class="sr-only" for="inlineFormInputTime">Uhrzeit</label>
    <input type="text" name="time" class="form-control mb-2"
id="inlineFormInputTime" placeholder="{hour:02.02d}:{minute:02.02d}">
    <div class="input-group-append">
      <button type="submit" class="btn btn-dark mb-2" name="button"
value="datetime">Zeit einstellen</button>
    </div>
  </div>
</form>
```

In main.py müssen wir nun zuerst eine Instanz von RTC erstellen und der Variable rtc zuweisen. Dies geschieht wieder am Anfang der File:

```
rtc = machne.RTC()
```

Anschließend gehen wir an die Stelle, an der die Server-Messages empfangen werden und setzen eine weitere Bedingung ein, die prüft, ob der Wert an der Stelle 'button' "datetime" entspricht. Wenn dies zutrifft, soll rtc initialisiert werden.

```
elif values_dict["button"] == "datetime":
    #Systemzeit initialisieren
```

RTC erwartet zur Initialisierung höchstens 6 Werte (year, month, day [,hour [,minute [, second]]]), wir haben aber in values_dict zwei Strings mit Datum (dd.mm.yyy) und Uhrzeit (hh:mm). Diese müssen vor der Initialisierung angepasst werden. Um die Endlosschleife übersichtlich zu halten, lagern wir dies in eine Funktion aus:

Innerhalb dieser Funktion müssen wir zuerst Datum und Uhrzeit in die einzelnen Werte splitten. Damit erhalten wir eine Liste mit den Werten, die allerdings immernoch als String vorliegen. Damit diese von RTC.init() verarbeitet werden können, müssen wir sie noch in Integer konvertieren.

```
def setSystemTime(date, time):
    """converts date and time (String) in integer values and sets system
time
    date: date as String (dd.mm.yy)
    time: time as String (hh:mm)
    """
    date_list = list(int(i) for i in date.split("."))
    date = tuple(date_list.reverse())
    time = tuple(int(i) for i in time.split(":"))
    rtc.init(date+time)
```

date_list = int(i) for i in date.split(".") kombiniert drei Schritte: Teilt den String an jeder Stelle, an der "." vorkommt und konvertiert jeden dieser Teile in einen Integer-Wert.

Da dieser Schritt aber einen Generator zurückgibt, wird diese Rückgabe mit list() in eine Liste konvertiert.

date = tuple(date_list.reverse()) kombiniert zwei Schritte:

date_list.reverse() kehrt die Reihenfolge der Elemente in der Liste um (von dd, mm, yyyy in yyyy, mm, dd) und wandelt die Liste in ein Tupel (reverse kann man leider nicht auf Tupel anwenden)

time = tuple(int(i) for i in time.split(":")) kombiniert ebenso drei Schritte: Teilt den String an jeder Stelle, an der ":" vorkommt und konvertiert jeden dieser Teile in einen Integer-Wert.

tuple() konvertiert diese Werte in ein Tupel.

rtc.init(date+time) kombiniert auch zwei Schritte: date+time bewirkt, dass die beiden Tupel date und time aneinandergehängt werden, die dann die RTC initialisieren.

Diese Funktion wird dann in der Endlosschleife folgendermaßen aufgerufen:

```
elif values_dict["button"] == "datetime":
    #initialize system time
    setSystemTime(values_dict["date"], values_dict["time"])
```

Fehler in der Eingabe auffangen

Oftmals kann es bei der Eingabe von Datum und Uhrzeit zu Fehlern kommen. So kann man aus Gewohnheit anstatt 2018 einfach nur 18 eingeben. Oder man verwendet für die Uhrzeit einen Punkt anstatt des Doppelpunkts.

In beiden Fällen würde die Initialisierung der Systemzeit fehlschlagen.

Um herauszufinden, ob wir split() mit einem Doppelpunkt oder einem Punkt durchführen müssen, können wir einfach den String nach dem gesuchten Zeichen (oder der gesuchten Zeichenkette) durchsuchen.

String.find(char(s) [, startPos = 0, endPos = len]) durchsucht den String nach der Zeichenkette, die mit charakter(s) angegeben wurde. startPos und endPos sind optional und bezeichnen die Position ab der, bzw bis zu der durchsucht werden muss.

Wenn die Suche erfolgreich war und die Zeichenkette gefunden wurde, gibt sie die Startposition zurück, ansonsten -1.

date.find(":") würde also bei einer korrekter Eingabe den Wert -1 zurückgeben, da wir zur Trennung zwischen Tag, Monat und Jahr in der Regel einen Punkt verwenden.

date.find(".") würde hingegen den Wert 2 zurückgeben, da die erste Position, an der der Punkt auftritt, die Position Nr 2 ist (nicht vergessen: Informatiker fangen bei 0 an zu zählen!)

Ebenso können wir prüfen, ob die eingegebene Ziffer für das Jahr lediglich zwei Stellen enthält (also 18 statt 2018) und für den Fall, dass lediglich zwei Ziffern vorhanden sind, 2000 dazu addieren.

Mit `len()` könnten wir die Anzahl der Ziffern für einen String erhalten. Dazu müssen wir für diese Prüfung wieder in einen String konvertieren.

Dies können wir verallgemeinern und in eine Funktion schreiben, die gleichzeitig auch die Konvertierung von String in Tupel aus Integern übernimmt und dann unter anderem in `setSystemTime` aufrufen:

```
def convertDateOrTimeToTuple(s):
    """ splits given String by '.' or ':'. If resulting List has len = 4
    it is interpreted as date and will be converted in (yyyy, mm, dd).
    if value for year has two digits, 2000 is added.

    s: string in format hh:mm or dd.mm.yy(yy)
    return: Tuple (hh, mm) or (yyyy, mm, dd)
    """
    if s.find(".") >= 0:
        s_list = list(int(i) for i in s.split("."))
    else:
        s_list = list(int(i) for i in s.split(":"))
    if len(s_list) == 4:
        s_list.reverse()
        if len(str(s_list[0])) == 2:
            s_list[0] += 2000

    return tuple(s_list)
```

Das Timer-Modul

Nun haben wir erreicht, dass auf dem ESP die aktuelle Systemzeit eingestellt werden kann.

Um den ESP aber als Wecker nutzen zu können, müssen wir zusätzlich einen Timer definieren. Hierzu gibt es das Timer-Modul:

https://github.com/loboris/MicroPython_ESP32_psRAM_LoBo/wiki/timer

Der Timer ist in folgenden Modi verfügbar:

- **ONESHOT:** Timer läuft für die vorgegebene Zeit und stoppt dann. Wenn die vorgegebene Zeit abgelaufen ist, stoppt der Timer und ruft eine Callback-Funktion auf, wenn diese definiert ist.
- **PERIODIC:** Timer läuft wiederholt bis er explizit gestoppt wird. Wenn die Zeit abgelaufen ist, wird die Callback-Funktion aufgerufen, sofern diese definiert ist.
- **CHRONO:** wird genutzt, um verstrichene Zeit in μ s zu messen. Er kann gestartet, gestoppt, pausiert und wieder gestartet werden.
- **EXTBASE:** Basis-Timer für verlängerte Timer. Hierfür kann nur Timer 0 verwendet werden.

tm = machine.Timer(timer_no) erstellt eine Timer-Instanz. `timer_no` ist die Nummer des verwendeten Timers.

für Hardware-Timer müssen wir hier 0-3 setzen

tm.init(period, mode, callback, dbgpin) initialisiert den Timer

- **period**: legt die Zeit in ms fest (Default: 10 ms)
- **mode**: definiert den Modus (Default: PERIODIC)
- **callback**: Die Callback-Funktion, die ausgeführt werden soll, wenn die Zeit abgelaufen ist (Default: None)
- **dbgpin**: Debug-Pin. Wenn dieser definiert ist, wird der Wert, der an diesem Pin anliegt gewechselt (wenn ein- dann aus und andersrum)

Das Timer-Modul als Wecker einsetzen

Um das Timer-Modul als Wecker einsetzen zu können müssen wir - neben den bekannten Schritten (Eingabemöglichkeit auf der Webseite erstellen sowie die Daten in main.py entgegennehmen) - auch die Zeit in ms berechnen, die als period in Timer.init() gesetzt wird. Zudem muss eine Callback-Funktion erstellt werden, die aufgerufen wird, sobald die Zeit abgelaufen ist.

Wir sind bei der Eingabe auf der Webseite faul und wollen lediglich die Zeit eingeben, zu der der Alarm losgehen soll. Also fügen wir in index.html unter der Form für die Systemzeit mit etwas Abstand noch eine Form zur Eingabe der Alarm-Zeit hinzu:

```
<form class="form-inline" method="POST" action="/" role="form">
  <div class="input-group">
    <label class="sr-only" for="inlineFormInputTime">Weckzeit</label>
    <input type="text" name="alarm" class="form-control mb-2"
      id="inlineFormInputTime" placeholder="{hour:02d}:{minute:02d}">
    <div class="input-group-append">
      <button type="submit" class="btn btn-dark mb-2" name="button"
        value="alarm">Weckzeit einstellen</button>
    </div>
  </div>
</form>
```

In main.py initialisieren wir zuerst Timer 1:

```
timer = machine.Timer(1)
```

und definieren dann eine Funktion setAlarmTime(time), in der wir zuerst die Zeit in ein Tupel konvertieren:

```
def setAlarmTime(time):
    time = convertDateOrTimeToTuple(time)
```

Anschließend brauchen wir die derzeitige Systemzeit aus Tupel und in Millisekunden, mit denen wir dann die Millisekunden zur Zeitpunkt des Alarms kriegen können:

```
curr_time_toup = utime.localtime()
curr_time_sec = utime.mktime(curr_time_toup)

alarm_time_sec = utime.mktime(curr_time_toup[0:3] + time +
curr_time_toup[5:])
```

curr_time_toup[0:3] bzw [5:] funktionieren ähnlich wie range. Bei [0:3] (oder [:3]) werden alle Elemente von Position 0 bis < 3 zurückgegeben, bei [5:] alle Elemente ab Position 5 (bis Ende).

Um period_sec zu berechnen, müssen wir nun curr_time_sec von alarm_time_sec abziehen.

Nehmen wir an, wir wollen jetzt (19.30 Uhr) den Wecker für morgen früh um 7.30 Uhr stellen, dann stehen wir vor einem kleinen Problem: Da wir für utime.mktime() alle Werte - bis auf die für Stunde und Minute - des heutigen Tags genutzt haben, wäre der Zeitpunkt in der Vergangenheit. Das bedeutet alarm_time_sec wäre kleiner als curr_time_sec und somit period_ms negativ!

Wenn dies der Fall ist, müssen wir die Anzahl der Sekunden pro Tag zu period_ms addieren!

```
period_sec = alarm_time_sec - curr_time_sec
if period_sec < 0:
    period_sec += seconds_per_day
```

Nun können wir den Timer initialisieren und die Callback-Funktion definieren. Aber da der Timer für period den Wert in Millisekunden erwartet, wir aber bisher mit Sekunden gearbeitet haben, müssen wir period = period_sec*1000 einsetzen:

```
timer.init(period = period_sec*1000, mode = timer.ONE_SHOT, callback=
timer_callback )
```

Timer Callback

Nun müssen wir definieren, was passieren soll, wenn der Timer abgelaufen ist.

Wir können den zuletzt aufgerufenen Song abspielen und eine Animation starten. Wie aber bereits bekannt kann es bei rechenintensiven Animationen dazu kommen, dass alles nur sehr langsam abgespielt wird. Daher würde ich - wenn überhaupt eine Animation laufen soll - die Animation blink empfehlen und diese mit langen Delays ausstatten. Um den zuletzt gewählten Song abzuspielen, muss dieser erst in einer Variablen gespeichert werden. Damit es keine Probleme gibt, falls seit dem letzten Start kein Song gewählt wurde, empfiehlt es sich, einen initialen Song am Anfang von main.py zu bestimmen.

Die Callback-Funktion des Timers braucht als Argument den Timer, für den diese Callback-Funktion gilt.

```
def _timer_callback(timer):
    """ Timer callback function to start playing alarm_song
    and start "blink" animation
    timer: according timer
    """
    global is_alarm_running
```

```
handleSoundThread(alarm_song)
handleAnimations("blink")
```

Wenn nun also der Timer abgelaufen ist, wird eine Melodie sowie die Animation "blink" abgespielt

Alarm stoppen

Nun benötigen wir nur noch eine Routine die es uns ermöglicht, den Alarm wieder zu stoppen.

Zum Anhalten der beiden Threads können wir ebenso den Touch Sensor einsetzen! Allerdings müssen wir gewährleisten, dass dieser auch nur zum Stop der Animationen fungiert, wenn auch ein Alarm läuft!

Dies erreichen wir durch eine Variable. `is_alarm_running` wird zu Beginn des Programms auf `False` gesetzt.

Wenn der Alarm losgeht wird sie `True`, sodass der Touch Sensor zum Anhalten der Animationen fungiert (und nicht etwa zum Wechsel). Wenn die Animationen angehalten wurden, muss man sie wieder auf `False` setzen, sodass Touch wieder dazu dienen kann, zwischen den Animationen zu wechseln.

Zu Beginn der File `main.py` setzen wir nun also ein:

```
is_alarm_running = False
```

Diese wird dann auf `True` gesetzt, sobald der Timer abgelaufen ist. Dies geschieht also in `_timer_callback`:

```
is_alarm_running = True
```

Anschließend bearbeiten wir die Routine, in der die Touch-Eingabe verarbeitet wird. Hier wird zuerst geprüft, ob `is_alarm_running` wahr ist und stoppt dann die Animationen. Falls nicht, soll die nächste Animation gestartet werden:

```
# if touch is registered check if alarm is running
if touch_val < touch_threshold:
    if is_alarm_running:
        handleSoundThread("")
        handleAnimations("off")
        is_alarm_running = False
    else:
        # print output and start function handleAnimations() without
        # argument to start next animation
        print("touched!! value: {}".format(touch_val))
        handleAnimations()
```


macOS: Wichtigste Terminal Befehle

Befehl	Bedeutung	Funktion
cd	change directory	Welchselt in das angegebene Verzeichnis
ls	list	Auflistung von Verzeichnissen und Inhalten
cp	copy	Kopiert Dateien und Verzeichnisse
mv	move	Verschiebt Dateien und Verzeichnisse
rm	remove	Löscht Dateien oder Verzeichnisse
mkdir	make directory	Verzeichnis/Ordner erstellen
rmdir	remove directory	Verzeichnis/Ordner löschen
open		Öffnet die angegebene Datei
sudo	substitute user do	Führt den Befehl als Superuser (root) aus
pbcopy	pasteboard copy	Kopiert die Inhalte in die Zwischenablage
pbpaste	pasteboard past	Fügt die Inhalte aus der Zwischenablage
ein		
kill		Beendet den angegebenen Prozess
killall		Beendet alle Prozesse, die den
angegebenen Befehl ausführen		
chmod	change mode	Zugriffsrechte von Dateien und Ordner
ändern		
zip		Verpackt Dateien und Verzeichnisse
unzip		Entpackt Dateien und Verzeichnisse
clear		Leert das aktuelle Terminal-Fenster
screencapture		Erstellt ein Screenshot des aktuellen
Bildschirms		
find		Suche nach Datei
mdfind		Spotlight-Suche
ps		Listet alle aktuell aktiven Prozesse auf
top		Listet eine detaillierte Prozessliste auf
history		Listet die zuletzt benutzten Befehle auf
reboot		Das System neustarten
shutdown		Das System herunterfahren

Windows: Wichtigste Terminalbefehle:

Befehl	Was macht das
dir	Listet den Inhalt des aktuell ausgewählten Verzeichnisses auf
dir /p	Zeigt den Inhalt seitenweise an
dir /w	Lässt die ausführlichen Informationen weg
dir /s	Listet zusätzlich die Unterverzeichnisse mit auf
cd	Wechstelt das Verzeichnis
cd..	Wechselt ins übergeordnete Verzeichnis
cd\	Wechselt ins Root-Verzeichnis
md Verzeichnisname Verzeichnis)	Legt ein neues Verzeichnis an (auch mkdir)
del Datei	Löscht die angegebene Datei
del Datei /s	Löscht zusätzlich zur angegebenen Datei auch alle Unterordner
rd Verzeichnis sein)	Löscht das angegebene Verzeichnis (muss leer sein)
re Verzeichnis /s	Löscht das Verzeichnis (muss nicht leer sein)
copy Quelle Ziel Datei an	Kopiert von Quelle nach Ziel und legt eine neue Datei an
move Quelle Ziel	Verschiebt von Quelle nach Ziel
rename NameAlt NmeNeu	Benennt eine Datei um
attrib	Ändert die Windows-Dateiattribute
attrib +R	Schaltet Schreibschutz an
attrib -R	Schaltet Schreibschutz aus
attrib s	Datei ist eine Systemdatei
attrib +H	Datei ist versteckt
attrib -H	Datei ist sichtbar
attrib A	Datei ist geändert/ archiviert
type	Zeigt den Inhalt einer Textdatei an

Wichtigste Linux Befehle

VERZEICHNISSE, DATEIEN:

cd	Wechselt in ein beliebiges Verzeichnis	cd /media/disk
cd ..	Wechselt ein Verzeichnis zurück	(/media/disk -> /
media)		
cd /	Wechselt in das tiefste Verzeichnis	cd /
cd -	Wechselt in das zuletzt besuchte Verzeichnis	cd -
cp	Kopiert eine Datei in angegebenes Verzeichnis	cp /tmp/test.txt /
media/disk		
mv	Verschiebt eine Datei und löscht die Quelldatei	mv /tmp/bla.txt /
media/disk		
mv	Benennt auch Dateien um	mv /tmp/x1.txt /tmp/x3.txt
rm	Löscht eine Datei	rm /tmp/bla.txt
rm -rf	Löscht alles in dem Verzeichnis	rm -rf /tmp/
mkdir	Erstellt ein Verzeichnis	mkdir /media/disk/bla
rmdir	Löscht ein Verzeichnis	rmdir /media/disk/bla
ls	Zeigt alle Dateien in einem Ordner an	ls /home/ubuntu
ls -l	Zeigt eine ausführliche Liste, mit ausführlichen Rechten an	ls -l /
home/ubuntu		
ls -la	Zeigt auch versteckte Dateien an	ls -la /home/ubuntu
pwd	Zeigt den Pfad zum aktuellen Verzeichnis	pwd
cat	Zeigt Inhalt einer Textdatei an	cat /home/test.txt
more	Zeigt Inhalt einer Datei seitenweise an	more test.txt
touch	Erstellt eine leere Datei in einem beliebigen Ordner	touch /ubuntu/
123.txt		

SYSTEM:

top	Gibt eine Übersicht über alle laufenden Prozesse und Systemauslastung	
free	Zeigt an wie stark der RAM ausgenutzt wird	
uptime	Dieser Befehl zeigt an wie lange das System schon online ist	
uname	Zeigt mit der Option -a einige Systeminformationen an z.B. die Kernelversion	uname -a
shutdown -h now	Führt den Computer runter	
whoami	Zeigt den eingeloggten Benutzer ein	whoami