

RainbowWarrior

Bauanleitung



Benötigte Materialien	3
Eine kurze Einführung ins Terminal	4
Getting Started	4
Terminalbefehle	5
Abhängigkeiten	7
FreeRTOS	8
USB Port ermitteln	8
Compiler	12
Interpreter	13
Bauteile verbinden	15
LED Strip an ESP anschließen	15
Auf den ESP zugreifen	15
Micropython	16
Neopixel	16
Farben	18
Animationen Teil 1: Blinken	20
Schleifen	20
For-Schleife	20
While-Schleife	22
Funktionen	23
Funktionen mit optionalen Parametern	25
Links	26
Aufgabe - Blinken in zwei Farben	26
Ein erstes kleines Skript	27
Skripte auf dem ESP ausführen	30
Aufgabe: Schleifen erstellen	30
Dateisystem auf dem ESP	31
Animationen Teil 2: running dots	33
Kommentare in (Micro)Python	35
##	36
Der Touch Sensor	38
Animationen steuern	38
Animation mit dem Touch Sensor starten	40
Bedingte Anweisungen / Verzweigungen	41
E: Datentypen in Python	44
E: Iterator	46
E: Lokale und Globale Variablen in Python	47
E: Multithreading	49
Implementierung von _thread	51
Fotowiderstand	54
Helligkeit der LEDs steuern	54
Analog Digital Converter	55
Photowiderstand mit ESP verbinden	56
Das Breadboard	57
LDR testen	58
Zeitgesteuerte Ausführung von code - noch nicht geschrieben!!	59
ESP als WebServer	60
WLAN auf dem ESP	60
AP-Mode	60
STA-Mode	62
mDNS Server	63
Das WebServer-Modul	64
Terminalbefehle: Linux	66
Terminalbefehle Windows	67
Terminalbefehle MacOS	68

getting the framework

Seite 2

pt (repl and local)

opython (let there be light)

oPython (make it blink)

n)

colors

i (t... ..)

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.

- 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, 1k Ω (braun – schwarz – rot – gold)
- Litzen in 3 Farben (Schwarz, rot, gelb oder andere)

Desweiteren werden folgende Werkzeuge benötigt:

- Lötkolben, Lötzinn, Löttauglitze
- 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

Eine kurze Einführung ins Terminal

Mit dem Terminal könnt ihr über Shellbefehle euren kompletten PC steuern. Hier könnt ihr 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 auf der grafischen Oberfläche. So können viele Aufgaben schneller und sauberer erledigt werden.

Manche Anwendungen bieten darüber hinaus auch bestimmte Kommandozeilen-Dienstprogramme, welche Funktionen bieten, die über das normale Programm nicht zu erreichen sind.

Terminal öffnen:

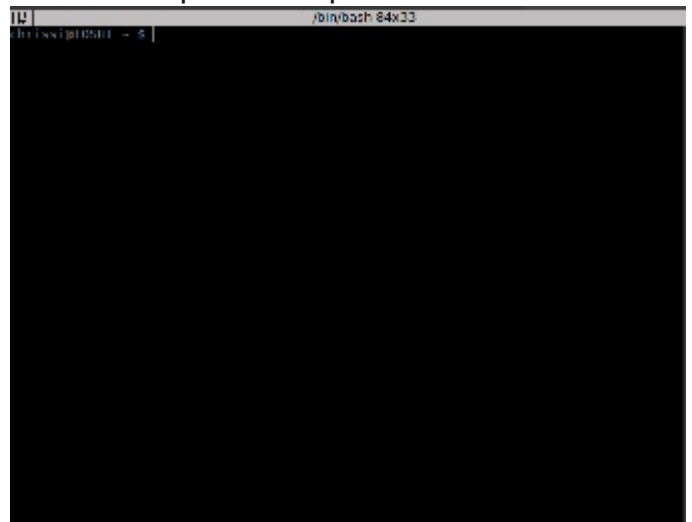
- Unter Linux: Super - bzw. Starttaste drücken und "Terminal" eingeben. Alternativ kann mit STRG+ALT+T per Tastatur das Terminal geöffnet werden
- Windows: Windows-Taste drücken, "cmd" eingeben, mit Enter bestätigen
- MacOS: CMD Leertaste drücken um Spotlight zu öffnen, "Terminal" eingeben.

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 signalisiert, dass man gerade als User, nicht etwas als Admin, im Terminal angemeldet ist.



Aktuelles Arbeitsverzeichnis:

pwd (=print working directory)

Falls du dir je nicht sicher sein solltest, ob du Dich im richtigen Ordner befindest, kannst du 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, mit

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

kann man, 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!

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

Möchtest Du in ein ganz anderes Verzeichnis wechseln, gibst du 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

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

Dateien kopieren

cp (=copy)

Um eine Datei zu kopieren, benutzt man den Befehl **cp** (= copy)
Hierzu wechselst Du mit **cd** in den Ordner, in dem sich die Datei befindet.
Anschließend gibst du ein:

```
$ 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).

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

Das ganze kann dann so aussehen:

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

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/Computernamen + 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 sind noch einige Pakete von Nöten, damit wir mit Python arbeiten können, der PC auch eine Verbindung zum Board aufbauen kann und so weiter. Außerdem müssen wir das Framework herunterladen.

Unter Linux hierzu einfach das Terminal öffnen und folgende Befehle nacheinander eingeben und jeweils mit Enter bestätigen (Achtung: der erste Befehl geht über zwei Zeilen!):

```
$ sudo apt-get install git wget make libncurses-dev flex bison  
gperf python 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
```

Um das MicroPython Framework auf das Board zu kriegen, hast Du zwei Möglichkeiten.
Entweder Du nutzt das Terminal und holst Dir eine Kopie davon auf deinen PC:

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

Oder aber Du gehst auf die Seite
„https://github.com/loboris/MicroPython_ESP32_psRAM_LoBo“,
klickst auf den Button „clone or download“ und lädst dir manuell die Zip-Datei herunter.
Diese musst du allerdings erst noch entpacken.

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 wir mit dem Framework loslegen können, ist es wichtig zu wissen, auf welchem USB Port der Controller 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, 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 weder nur unter Linux und MacOS - den Befehl

dmesg (=display messages)

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

```
[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: 0x8114
[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 0x8114 dynamically updated by country IE
[33423.808988] IPv6: ADDRCONF(NETDEV_CHANGE): wlp4s0: link becomes ready
[33494.695520] input: B8:69:C2:CD:92:4C as /devices/virtual/input/input20
chrissi@T05H1 ~ $
```



```

[33423.000988] IPv6: ADDRCONF(NETDEV_CHANGE): wlan0: link becomes ready
[33494.695520] input: 08:69:C2:CD:92:4C as /devices/virtual/input/input20
[33744.071253] toshiba acpi: Unknown key e00
[34130.011877] input: 08:69: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@TOSHIBA ~ $

```

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 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.

Betriebssystem / Framework installieren

Wichtig! Die folgenden Informationen funktionieren auf Linux gleichermaßen 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 kannst du entweder mit deinem FileBrowser zuerst in das heruntergeladene Verzeichnis, dann ins Unterverzeichnis „MicroPython_BUILD“ wechseln, von dort aus mittels Rechtsklick das Terminal öffnen.

Oder, falls du auf im Terminal geblieben bist, mit

```
$ cd MicroPythonESP32_psRAM_LoBo/MicroPython_BUILD
```

ins entsprechende Unterverzeichnis wechseln. Anschließend geht's an die Kernel-Settings.

Eigentlich ist in diesem Schritt nicht viel zu tun, jedoch müssen wir kurz die menuconfig aufrufen, um die sdkconfig zu erstellen.

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

Mit diesem Befehl öffnest du die menuconfig.

Hier navigiert man ausschließlich mit den Cursor-Tasten. Oben/Unten um die verschiedenen Menüpunkte auszuwählen, Rechts/Links um in der Menüliste unten auszuwählen.

Gerne kannst Du dich etwas umschauen und dir Gedanken machen, aber, auch ich weiß bei vielem hier noch nicht, was es zu bedeuten hat, bzw. welche Auswirkungen das hat.

Mit „select“ kannst du das Untermenü, bzw. in die Einstellungen des ausgewählten Menüpunktes gelangen. Wenn du einmal nach rechts drückst, sodass „exit“ markiert ist, anschließend auf Enter, gelangst Du wieder ins übergeordnete Menü.

Wir müssen hier allerdings die Webserver-Funktion aktivieren.

Dazu wechselst Du in **Micropython** → **SystemSettings** und navigierst zum Menüpunkt „**Enable WebServer**“. Wenn dieser noch nicht mit einem Stern markiert ist, einmal auf „y“ für „yes“ drücken um diesen zu aktivieren. Falls Du versehentlich etwas aktiviert haben solltest, das Du nicht wolltest, kannst Du dieses mit „n“ für „no“ wieder deaktivieren. Zum Beispiel kann man hier auf den Telnet- sowie den FTP-Server verzichten. Es sei denn, Du spielst mit dem Gedanken, einen Fileserver zu erstellen oder etwa Kommandos über das Internet, sprich via WLAN an den Controller zu schicken. Dann könnte Telnet interessant werden.

Das Webserver-Modul benötigt das Modul namens „**Websockets**“. Dieses befindet sich unter **Micropython → Modules**.

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.

Außerdem müssen wir dafür sorgen, dass wir den vollen Speicherbereich nutzen können. Dies kann man unter **MicroPython → System settings → MicroPython heap size (KB)** einstellen. Hier sollte der Wert **96** stehen.

Anschließend in der unteren Menüleiste mit Pfeil-Rechts Taste zu **save** wechseln und mit Enter bestätigen. Du wirst noch einmal gefragt, 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 je etwas bei der Ausführung schief gehen sollte oder Du Dir nicht sicher sein solltest, kannst Du ebenso die Datei „sdkconfig“ aus dem Ordner „instruction“ nehmen und in „MicroPython_ESP32_psRAM_LoBoNeu/MicroPython_BUILD“ einfügen.

Anschließend ebenso die Menuconfig öffnen, mit dem Cursor nach rechts auf **load**, mit Enter bestätigen und die Datei „sdkconfig“ auswählen. Nun – wie oben beschrieben – speichern und beenden.

Jetzt geht's an die binary!

Hört sich aber schlimmer an als es ist :)

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

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

Du möchtest wissen, was hier passiert?

Ein Build ist der Vorgang in der Softwareentwicklung, bei dem eine Anwendung, bestehend aus Binärcode (binary) aus dem Quellcode erzeugt wird. Diesen Prozess nennt man "kompilieren". Kompiliert wird mit einem sogenannten Compiler.

Compiler

Compiler sind spezielle Übersetzer, die Programmcode aus problemorientierten Programmiersprachen, sogenannten Hochsprachen, in

ausführbaren Maschinencode, also die Sprache, die die Prozessoren verstehen, übersetzen.

Wenn du dich einmal in der Ordnerstruktur des Micropython Frameworks umsiehst, findest du 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 schwer lesbar ist, werden in der Assemblersprache (die Sprache, die der Maschinsprache 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, die Variablen werden in sogenannten Registern gespeichert. Dies sind Speicherbereiche innerhalb des Prozessors, die direkt mit dem Rechenkern verbunden sind.

Falls du mehr darüber erfahren möchtest, hier einige Links:

<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 8B 55 FC 01 D0 89 45 F4	mov eax, DWORD PTR [rbp-8] mov edx, DWORD PTR [rbp-4] add eax, edx mov DWORD PTR [rbp-12], eax	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.

Interpreter

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

Vorteil hiervon ist, dass ein zum Beispiel in Python geschriebenes Programm auf verschiedenen Systemen mit verschiedenen Prozessorarchitekturen ausführbar ist. Leider muss man aber im Gegensatz zum Compiler mit Performanceeinbußen rechnen.

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 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

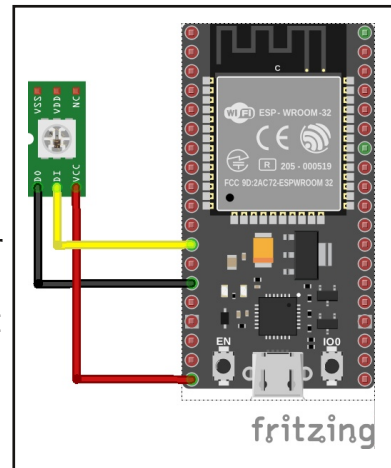
Neugierig? Ersten Test??

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.



Auf den ESP zugreifen

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 löschen, und um 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. Bitte hier den richtigen Port für dein System verwenden!

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
/bin/bash 80x24
chrissi@TOSHI ~ $ 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 sind.

Micropython

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.

Modul: machine

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

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.

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
>>> 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:

```
>>> import machine
>>> strip = machine.Neopixel(machine.Pin(14, machine.Pin.OUT),
11, 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** gibt an, welcher Pixel genau 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 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    clear
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 angegeben 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 r=255, sprich höchster Wert, g=0, b=0. Anders geschrieben: (255, 0, 0).

Dies ist der sogenannte 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.

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, konvertiert in eine Hexadezimalzahl, aneinandergereiht.

Die Farbe "aquamarin" hat zum Beispiel folgenden 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.

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.

Animationen Teil 1: Blinken

Bevor wir die LEDs blinken lassen können, müssen wir uns überlegen, was hier eigentlich genau geschieht:

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

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

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

Nur. Das wäre ja ganz schön doof. Wollten wir die LEDs auf diese Weise 1000x blinken lassen, hätten wir mittels copy + paste einiges zu tun! Außerdem würde das Programm unübersichtlich werden!

Viel besser wäre es doch wenn wir sagen könnten:

mach bitte 5x:
mach an – warte – mach 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 obs 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.

For-Schleife

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

Seite 20

< 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.

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 "...":

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

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)])`

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

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 lassen wollen, können wir nochmals mit Enter bestätigen, sodass die Schleife ausgeführt werden kann.

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

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

Um diesen Prozess zu verlangsamen, müssen wir sogenannte delays einbauen. Dies macht man 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. Gib nun folgende Zeilen ins Terminal und bestätige - wie eben schonmal - 3x mit Enter.

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

Wie du siehst erscheinen nun die Ergebnisse der einzelnen Schleifendurchläufe in zeitlichen Abständen auf dem Terminal. Selbes Verhalten wollen wir für unsere blinkenden LEDs!

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 "...":

While-Schleife

Die einfachste 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 – pause – schalte aus – pause.

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 wird es Zeit, einen Blick auf Funktionen zu werfen. Selbstverständlich kannst du auch ohne Funktionen ein wenig mit den Werten beim Blink-Algorithmus spielen.

Allerdings wird dich sehr schnell die viele Schreibarbeit nerven. Dabei möchtest du doch nur die Farbe verändern, oder auch die Zeiten, wie lange die LEDs ein oder aus sein sollte.

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):
```

Anweisung(en) (return)

das **return** Statement ermöglicht es einem, Ergebnisse von Berechnungen, oder eben Werte eines ausgelesenen Sensors, wieder zurück zu geben. Diesen Wert kann man also beim Funktionsaufruf in eine Variable speichern. 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.

Zu 3. das Ergebnis der Addition soll zurückgegeben werden.

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

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

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

Zu 2. Wir möchten die Farbe und die Zeiten verändern. Brauchen dazu also drei Werte.

Zu 3. Da wir lediglich eine Ausgabe an einen Pin machen, brauchen wir also keinen Wert zurückgeben

In unserer bisherigen Blink-Funktion haben wir mit festen Werten für die Farbe und die Zeiten gearbeitet. Diese müssen wir nun durch Variablen ersetzen, dessen 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, nehmen wir "time_on"

Für die Zeit, die die LEDs nicht leuchten sollen, nehmen wir "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 kannst du ganz einfach mit Farbe und Blink-Intervallszeiten herumspielen.

Gebe dazu einfach folgendes im Terminal ein und ersetze die Werte nach Belieben:

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

Gerne kannst du anstatt den vordefinierten Werten auch selbst mit Farbwerten herumspielen. Im Netz findet sich unter dem englischen Suchbegriff "hex color picker" (= hexadezimaler Farbwähler) eine einfache Möglichkeit, die gewünschte Farbe als Hexadezimalwert zu erhalten.

Funktionen mit optionalen Parametern

Sicherlich hat es dich bereits verwundert, dass bei der Funktion `strip.set()`, im Falle, dass alle LEDs gleichzeitig geschaltet werden sollen, die Anzahl 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 du nun schon etwas mit den Zeiten der Blink-Funktion herumgespielt hast und für dich optimale Zeit-Werte herausgefunden hast, kannst du 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

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

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

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

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

Jetzt kommt der entscheidende Vorteil der MicroPython implementierung zum Einsatz!

Während man bei Arduino-Umgebung oft mühsam durch Recherchen erst herausfinden muss, welche Funktionen in einem Modul zur Verfügung stehen, und erst nach dem Upload auf den Controller herausfindet, ob das so überhaupt funktioniert, kann man das hier mittels REPL (Read Eval Print Loop) ganz schnell.

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 Beispiel hierfür findest du im Anhang.

Ein erstes kleines Skript

Sicherlich hast auch du dich beim Funktionen schreiben im Terminal bereits einige Male vertippt und musstest alles neu eingeben?
Dies ist auf Dauer sehr mühsam.

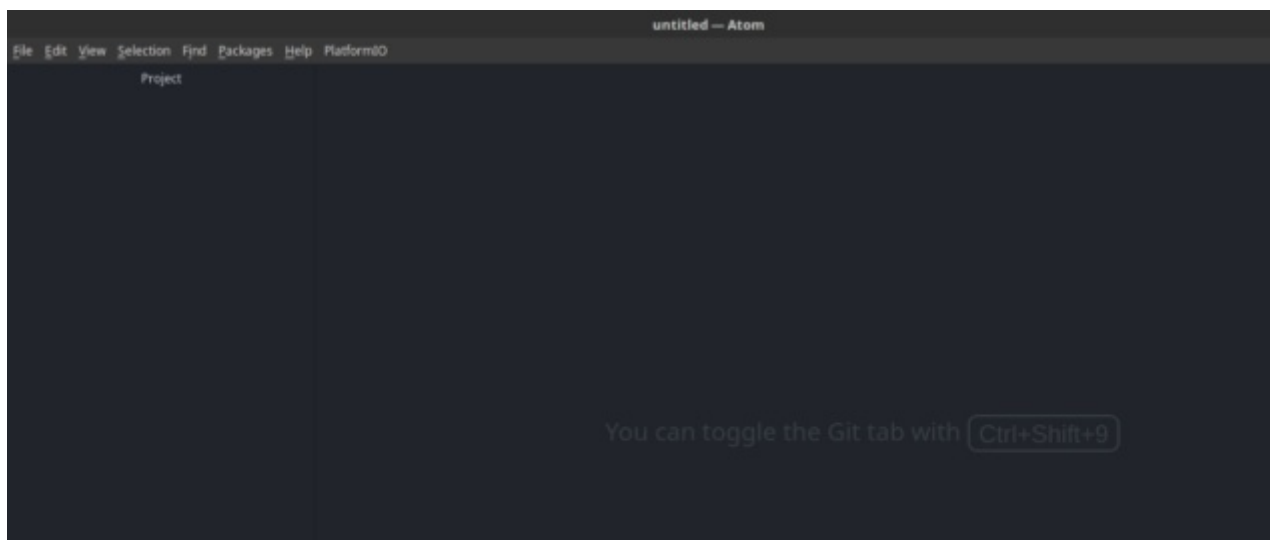
Da unsere Funktionen nun nach und nach immer komplexer werden und es nun vorteilhaft ist, Codepassagen einfach überarbeiten zu können, steigen wir nun auf einen Texteditor um.

Editor

Hierzu könntest du für den Anfang den normalen Texteditor nutzen, der auf jedem System vorinstalliert ist. Allerdings ist es sehr von Vorteil, einen Editor zu nutzen, der auch fürs coden gedacht ist. Ich nutze für meine MicroPython Projekte den Editor "Atom.io", der kostenlos für alle Systeme verfügbar ist. Dieser kann auf <https://atom.io/> heruntergeladen werden.

Falls du noch keinen Ablageort für deine Skripte haben solltest, empfiehlt es sich, gleich einen anzulegen.
Ich habe zum Beispiel in meinem Home-Verzeichnis ein Verzeichnis namens "coding" in dem ich meine Skripte - sortiert nach Programmiersprache - ablege.

Atom besteht im Wesentlichen aus zwei Bereichen:
Im linken Bereich, "Project" genannt, hast du die Möglichkeit, Projektordner hinzuzufügen. Im rechten Teil werden dann die Textdateien angezeigt.
Um einen Projektordner zu erstellen, bzw. auszuwählen, klickst du einfach mit der rechten Maustaste in das "Project"-Feld und gehst auf "Add Project Folder".
Am besten ist es, du legst im selben Schritt innerhalb des Ordners für deine coding-Projekte einen eigenen Ordner für das Projekt an, das hier entsteht.
Nennen wir es "rainbowWarrior"



Wenn du nun auf den neu erstellten Projektordner wieder mit der rechten Taste klickst, kannst du mit "new File" eine neue Datei anlegen. Alternativ geht das - bei markiertem Projektordner - mit der Taste A.
Die neue Datei muss gleich benannt werden (Dateiendung nicht vergessen!):
Nennen wir sie "animations.py"

Im rechten Bereich befindet sich dann der eigentliche Texteditor, hier werden deine geöffneten Dateien angezeigt. Und hier werden wir nun nach und nach, beginnend mit dem import-Statement, alles bisher gemachte übertragen.

```
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
```

Sehr schnell erkennt man hier die Vorteile:

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 Zeilenzahlen 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 ins Terminal switchen.

Hier wechseln wir mit cd zu dem Ordner, in dem unser Skript gespeichert wurde (in meinem Fall chrissi/coding/microPy/RainbowWarrior) und stellen eine Verbindung zum ESP her.

Auch ich muss mir den Befehl hierfür immer wieder rauskramen. Copy & Paste geht hier einfach schneller :)

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

bevor wir aber REPL nutzen, 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 du sicherlich erkannt hast, funktioniert das auf die selbe Weise wie vorhandene Module eingefügt werden. Das bedeutet ja - du hast im Prinzip schon dein eigenes Mini-Modul geschrieben! Besteht zwar momentan nur aus einer Funktion, aber die wird nicht lange alleine bleiben!

Oooops! Wem ist's aufgefallen?

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

Da hat sich der Fehlerteufel eingeschlichen! Aber so können wir gleich mal sehen, wozu die Zeilenanzeige im Editor gut ist! Bei dieser Fehlermeldung sind für uns erst einmal die letzten beiden Zeilen interessant. 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 einem Fehler gekommen ist - also Zeile 11.

`utime.sleep_ms(time_on)` sieht allerdings richtig aus. Ich kann hier keinen Syntax-Fehler erkennen! In dem Fall sollte man sich auch die darüberliegende Zeile genauer ansehen.

`strip.set(0, color, num=num_pixels` - 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.

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

Tipp für "faule": Du kannst diese Befehle auch auf eine Zeile komprimieren! In Linux ist die Systemsprache eigentlich 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 können wir uns im

Terminal zunutze machen, indem wir den Befehl zum kopieren und REPL starten auf eine Zeile bringen und mit einem ";" trennen:

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

Auch wenn in Python keine Semikolons nötig sind um die Zeile (oder den Befehl) zu beenden, funktioniert selber Trick 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.

Skript auf dem ESP ausführen

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

```
animationen.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 ist es am einfachsten, wenn wir REPL wieder verlassen und uns den Inhalt mittels `ls` ausgeben lassen.

Um REPL zu verlassen, muss nun wieder STRG+X gedrückt werden.

```
ls /flash
```

gibt uns nun den aktuellen Inhalt des ESP aus.

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

Dies ist die erste Datei die beim Start 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 nicht verändern, jedoch können wir gerne mal einen Blick hinein werfen.

```
cp /flash/boot.py /home/chrisi/coding/microPy
```

kopiert die Datei vom Controller in dein lokales Filesystem, sodass du sie mit jedem beliebigen Editor öffnen kannst.

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

Hier wird das Modul "sys" (Abkürzung für System) eingebunden, und die Funktion `sys.path()` aufgerufen.

Um herauszufinden, was dies bewirkt, 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 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 nicht gefunden.

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

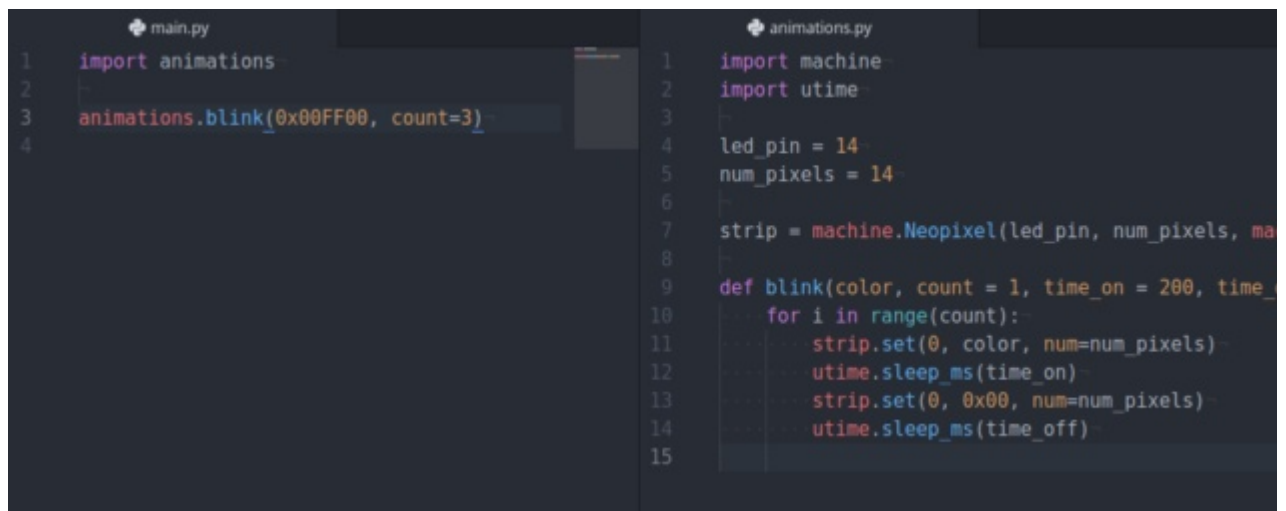
Nur, was macht die main.py?

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

In unserem Fall können wir eigentlich alles übertragen, was wir zuletzt nach dem 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" in unserem Projektordner. und fügen die zuletzt in REPL ausgeführten 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 hast du sicherlich bereits die Aufgaben 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.



```
main.py
1 import animations
2
3 animations.blink(0x00FF00, count=3)
4

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, ma
8
9 def blink(color, count = 1, time_on = 200, time_
10     for i in range(count):
11         strip.set(0, color, num=num_pixels)
12         utime.sleep_ms(time_on)
13         strip.set(0, 0x00, num=num_pixels)
14         utime.sleep_ms(time_off)
15
```

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 man zum Beispiel nur eine LED zur gleichen Zeit - jedoch nach und nach um eine Position verschoben - leuchten lässt.

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.

Kommt dir bekannt vor?

Hierfür brauchen wir eine for-schleife, wobei die obere Grenze der range-Funktion gleich der Anzahl der LED (also num_led) ist.

Wir brauchen nun keine weitere Datei mehr anzulegen. Diese Funktion können wir einfach in animations.py einfügen!

```
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)
```

Dieses mal wollen wir wieder die Möglichkeit haben, die Farbe und die Zeit, die die LED leuchten soll, zu bestimmen, daher werden color und time_on als optionale Parameter angegeben. color = 0x00FFFF ergibt die Farbe cyan, mit time_on = 50 ms läuft das ganze 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 "Kleinigkeit" in dieser MicroPython Implementierung hinweisen, die 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 - alles, was sich an erster Stelle befindet, mit dem Index 0 angesprochen wird. Ebenso wird alles, was 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 0 angesprochen werden, die LED an der letzten Stelle mit (num_pixels -1)

Dies ist hier aber leider anders umgesetzt! Die erste LED kann man sowohl mit 0 als auch mit 1 ansprechen, mit `num_pixels-1` würde aber die vorletzte leuchten!

Daher müssen wir in `strip.set()` als Index `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

Du möchtest nun, dass lediglich die Animation `running_dots()` ausgeführt wird, nicht aber die Blink-Animation? Dann kannst du selbstverständlich die entsprechende Zeile in `main.py` löschen - oder einfach dafür sorgen, dass diese Zeile "überlesen" wird.

Dies kann man durch Kommentare erreichen.

Üblicherweise werden Kommentare eingesetzt um zu beschreiben, was im nachfolgenden Codeblock geschieht. Aber man kann diese während der Entwicklung ebenso einsetzen, um Codezeilen vorübergehend nicht ausgeführt werden.

Aus Performancegründen sollten diese Kommentare allerdings gelöscht werden, sobald das Programm fertig ist.

Man unterscheidet zwischen zwei verschiedenen Arten: 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
```

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. """
```

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, müssen wir das white-level einsetzen (steuerbar mit dem optionalen Parameter "white" - in strip.set() beschrieben).

Die zweite LED, die also etwas weniger farbintensiv ist, 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. LEDs, die mit negativen Werten angesprochen werden, gibt es nicht! Auch würde dies beim Ausführungsversuch zu Fehlermeldungen führen (Index Error - List Index out of range)!

Daher müssen wir garantieren, dass der Wert nie kleiner als 1 und auch nie größer als num_pixels ist.

Würde der Index des Strips von 0 bis num_pixels-1 gehen, könnte man dies ganz einfach mit dem Modulo-Operator (einfache Restdivision, die als Ergebnis den Restwert zurückgibt) lösen. Da dies aber nicht der Fall ist, können wir so lediglich ausschließen, dass der Wert für i nie größer wird als num_pixels +1 :

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

Da bei der Modulodivison - 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).

Also müssen wir jedes Mal, wenn der Wert für pos1 = 0 ist, diesen auf num_pixels setzen.

Hierfür nutzt man in der Programmierung sogenannte Verzweigungen.

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

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

Genauer gesagt sind diese Touch-Sensoren sogenannte Kapazitive Näherungssensoren.

Die Funktion dieses Sensors beruht auf der Änderung des elektrischen Feldes in der Umgebung vor seiner Sensorelektrode (aktive Zone).

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

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 konfigurieren, 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 Aushandhabehandlung (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_pin = 27
touch = machine.TouchPad(machine.Pin(touch_pin))

while True:
    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 130.

Setzen wir also den Schwellwert, ab dem der Sensor als "berührt" gelten soll, auf 150. Falls es im weiteren Verlauf zu Schwierigkeiten kommen sollte, wenn der Code eine Berührung registriert hat obwohl keine stattgefunden hat, oder keine Reaktion erfolgen sollte obwohl der Sensor berührt wurde, kann man diesen Schritt einfach wiederholen.

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

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 600, mit Berührung grob zwischen 30 und 140.

Animationen mit dem Touch Sensor starten

Um Animationen mit dem Touch Sensor zu starten, müssen wir also jedes mal, wenn der Schwellenwert unterschritten wird, 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 Gegenteilige 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 = 150

while True:
    try:
        touch_val = touch.read()
    except ValueError:
        print("TouchPad Error")

    if touch_val < touch_threshold:
        animations.next()
    utime.sleep_ms(200)
```

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

Diese müssen wir noch in die File `animations.py` basteln.

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 - besser noch in einen Dictionary - schreibt, bei der man ganz einfach sagen kann: starte die Nächste!

Datentypen in Python Teil 1

Bisher haben wir nur ganz allgemein über Variable gesprochen.

Variablen werden genutzt, um veränderbare Objekte zu speichern. Man nicht nur einzelne Zahlenwerte, die zum Beispiel als ganze Zahl (Integer) oder Gleitkommazahl (float oder double) vorliegen können, oder sogenannte Strings (einfache Zeichenfolgen, sprich Text - welcher innerhalb doppelter Anführungszeichen stehen muss) in Variablen speichern.

Man kann sogar Listen (also eine Liste an Werten oder Variablen), sogenannte Tupel (bestehend aus mehreren Werten, eingeklammert und durch Komma getrennt), Dictionarys (ähnlich den Listen, nur dass jedes Objekt noch einen Bezeichner enthält. Diese werden in geschweiften Klammern angegeben) und sogar ganze Funktionen in Variablen speichern.

Genauso kann man Funktionen in Listen eintragen. Diese Eigenschaft machen wir uns nun zunutze.

Listen werden erstellt, indem man einer Variable mehrere Werte zuweist, die in eckigen Klammern und durch Komma getrennt werden:

```
my_list = [list_item1, list_item2, list_item3, ... ]
```

Analog verfährt man mit den **Tupeln**:

```
my_tupel = (item_1, item_2, item_3, ... )
```

Dictionaries haben zusätzlich zum Objekt einen Bezeichner, über den man das Objekt in der Liste aufrufen kann:

```
capital_dict = {"Baden-Wuerttemberg":"Stuttgart", "Bayern":"Muenchen", ... }
```

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_listen.php

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

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.

In der File animations.py erstellen wir also zuerst einen Dictionary, mit der wir arbeiten können. In dieser Liste werden alle vorhandenen Animationen gespeichert. Wichtig ist, dass diese Variable erst nach den ganzen Funktionen deklariert wird. Beim Import der File animations.py werden - von oben nach unten - alle Variablen und Funktionen angelegt. Würden wir diese 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 ans Ende der File gestellt.

```
animation_dict = ["blink" : blink, "running_dot" : running_dot,  
"crossing_dot" : crossing_dots, "v_shape" : v_shape ]
```

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 wird, 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 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.

```
animation_dict = ["blink" : blink, "running_dot" : running_dot,
"crossing_dots" : crossing_dots, "v_shape" : v_shape ]
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)
```

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

Fertig für den Test!

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

Nun müsstest du in der Lage sein, die Blink-Animation mit dem Touch Sensor zu starten.

Bei erneuter Berührung sollte `running_dot` starten.

Nur. Wenn man jetzt die nächste Animation starten möchte steht man vor einem Problem:

Der Sensor reagiert nicht mehr auf Eingaben!

Das liegt ganz einfach daran, dass der Code ja linear abgearbeitet wird.

Sprich während der Ausführung der Main-Funktion wird im Endlos-Loop die Animation gestartet.

`Running_dot` ist allerdings eine Funktion, die ebenso in einer Endlos-Schleife steckt. Somit ist es nicht mehr möglich, in der Main den Sensor auszulesen!

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 solch engen Zeitfenstern, dass es der User nicht wahrnimmt.

Multithreading verhält sich ähnlich, nur dass hier nicht zwei Programme, sondern ein Programm in verschiedene Threads, also verschiedene Aufgabenbereiche gegliedert wird, die parallel abuarbeiten 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 diesem die Funktion **notification = _thread.getnotification()** ausgeführt werden. Sobald in Notification der selbe Wert steht, der auch **_thread.EXIT** hat (EXIT ist keine 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 **break** 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 etwas eigenes 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 diese zuerst abbrechen müssen, müssen wir die variable `animation_thread` einsetzen 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 0 zuweisen und in der Zeile bevor der Thread gestartet wird `_thread.stop(animation_thread)` einsetzen.

Hier nochmal die komplette `main.py`:

```
import machine, utime, _thread
import animations

touch_pin = 27
touch = machine.TouchPad(machine.Pin(touch_pin))
touch_threshold = 150

animation_thread = 0

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 Fotowiderstand

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.

Analog Digital Converter - das ADC Modul

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.

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.

Dieser ist auf den Pins 32 - 38.

32, 33, 34, 35, 36, 37, 38, 38

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)
```

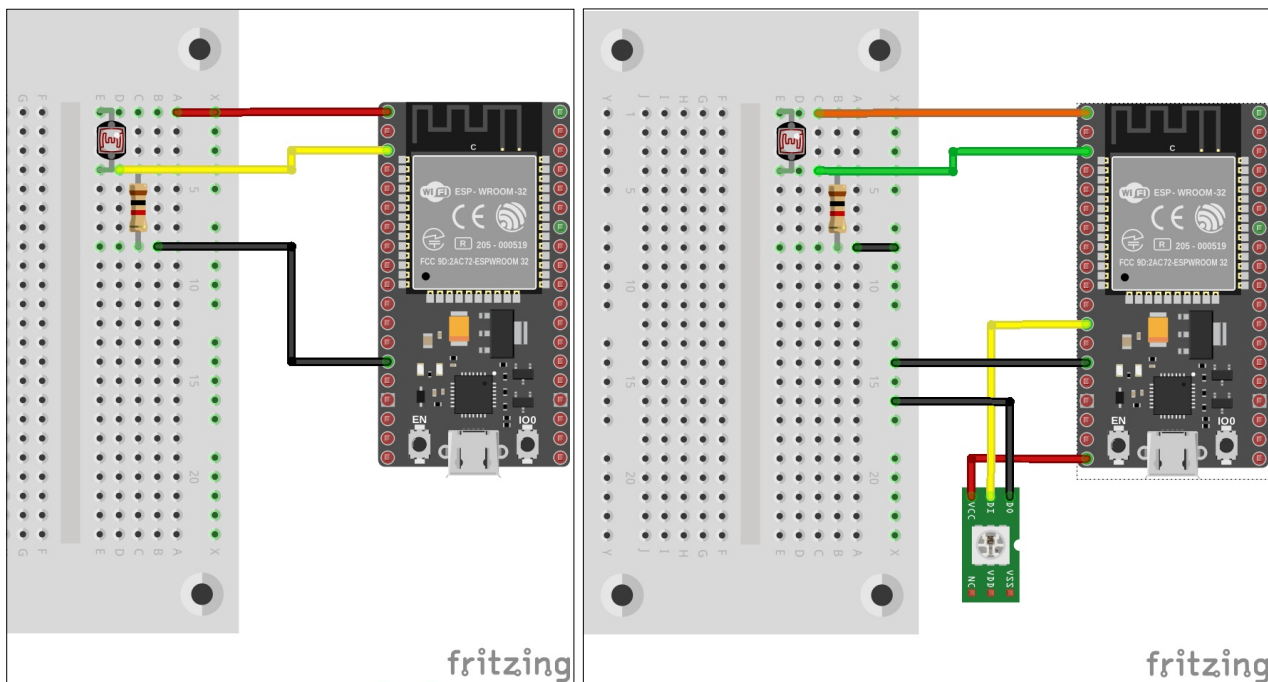
Diese drei Zeilen lassen sich auch auf eine komprimieren!

```
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 basteln. Hierzu benötigen wir neben dem Photowiderstand eine Spannungsquelle mit 3.3V vom ESP 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 einfach 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 miteinander verbunden. So kann ich, wenn ich für zwei Elemente GND brauche, den GND einfach mit dieser Reihe verbinden. Genauso verfähre ich mit GND für die LED sowie mit GND für den 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 waagrecht 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.

ESP als WebServer

Ein Webserver ist ein System, das Webseiten bereitstellt. Um aber auf diese Webseiten zugreifen zu können, bedarf es einer Inter- oder Intranetverbindung in Form von Ethernet oder WLAN zu diesem System.

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-Verbindungen 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. 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 WLAN 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 Computernamen 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

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.

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 _httpHandlerPost(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://github.com/crxcv/Leuchteding/blob/master/pyboard_code/www/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.

```

<!DOCTYPE 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">
    <div class="sticky-top" id="id3">
      <a href = "/" class = "btn btn-dark btn-lg" role="button" disabled> Animationen </a>
      <a href = "/alarm" class = "btn btn-dark btn-lg" role="button" > Alarm </a>
      <a href = "/led" class = "btn btn-dark btn-lg" role="button"> Farbe</a>
    </div>
    <h1 id="id1">RainbowWarrior</h1>
    <h2 id="id3"></h2>
    <div class="mt-3">
      <form method="POST" action="/">
        <h3 class="mb-3" id="id1">LED Pattern ausw&auml;hlen:</h3>

        <div class="my-4">
          <input class="btn btn-dark" type="submit" name="li" value="Submit color"></b>
        </div>
      </form>
    </div>
  </div>
</body>
</html>

```

Mindestens 90% 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:

```
<form method="POST" action="/">
  <h3 class="mb-3" id="id1">LED Pattern auswahl</h3>

  <div class="my-4">
    <input class="btn btn-dark" type="submit" name="li" value="Submit color"></b>
  </div>
</form>
```

Eine HTML-Form erfüllt den Zweck, User-Eingaben entgegenzunehmen und vom Server weiterverarbeiten zu lassen. Dazu muss mit "method" angegeben werden, mit welcher Methode (Get oder Post) gearbeitet werden soll und mit "action", welcher Route-Handler verwendet werden soll. Die Form beginnt mit dem öffnenden HTML-Tag:

```
<form method="POST" action="/">
```

und endet mit dem schließenden Tag:

```
</form>
```

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

```
<div class="my-4">
  <input class="btn btn-dark" type="submit" name="li" value="Submit color"></b>
</div>
```

Nun müssen wir nur noch unsere Animationen einfügen.

Für jede unserer Animationen müssen wir nun ein eigenes div-Tag eintragen:

```
<div class="form-check">
  <label class="form-check-label"><input class="form-check-input" type="radio"
id="light1" name="light" value="running_dot">Running Dot</label>
</div>
```

Ich werde hier nur die relevanten Schlüsselwörter beschreiben:

type="radio" gibt an, dass es sich um einen Radio-Button handelt. Dies gewährleistet, dass man innerhalb dieser Form nur eines dieser Elemente aussuchen kann.

name="light" werden wir später nutzen, um zu prüfen, ob der von der Seite empfangene Wert wirklich dazu da ist, die Animationen zu steuern.

Seite 68

value="running_dot" ist der Wert, der 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. t

Running Dot - was vor dem schließenden `</label>`-Tag steht ist lediglich der Text, der auf der Webseite angezeigt wird.

Eingaben auf dem Server verarbeiten

Nun wissen wir, dass eine HTML-Post-Form zum Einsatz kam, die wir auf dem Server verarbeiten müssen.

Dies müssen wir dem Route-Handler mitteilen.

Also ersetzen wir die erste Zeile des Route-Handlers in webSrv.py mit:

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

Anschließend müssen wir überlegen, wie Route-Handler-Funktion modifiziert werden muss, um die Daten weiterzuverarbeiten.

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

```
formData = httpClient.ReadRequestPostedFormData()
```

Nun müssen wir unsere empfangenen 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 parallel von webSrv.py gerade 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!

Seite 70

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>

Zusammengefasst sieht das dann so aus:

```
_thread.ReplAcceptMsg(True)
def handleSrvMsg():
    msg = _thread.getmsg()
    if msg[0] == 2:          #true if msg is a String
        values = msg[2].split(":")
        if values[0].strip() == "light":
            animations.start(values[1].strip())
```

Die Funktion handleSrvMsg() sollte nun noch in der Endlosschleife in main.py eingefügt werden, sodass die Nachrichten regelmäßig abgerufen werden können.

Animationen mit einem String starten

Nun haben wir in `checkSrvMsg()` 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 einem for-Loop prüfen, welche Animation gestartet werden soll:

```
def next(animation_name):
    for ani in animation_dict:
        if ani[0] == animation_name:
            ani[1]()
```


Wichtigste Linux Befehle

VERZEICHNISSE, DATEIN:

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

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	Legt ein neues Verzeichnis an (auch mkdir Verzeichnis)
del Datei	Löscht die angegebene Datei
del Datei /s	Löscht zusätzlich zur angegebenen Datei auch alle Unterordner
rd Verzeichnis	Löscht das angegebene Verzeichnis (muss leer sein)
re Verzeichnis /s	Löscht das Verzeichnis (muss nicht leer sein)
copy Quelle Ziel 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

macOS: Wichtigste Terminal Befehle

Befehl	Bedeutung	Funktion
cd	change directory	Welchset 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

