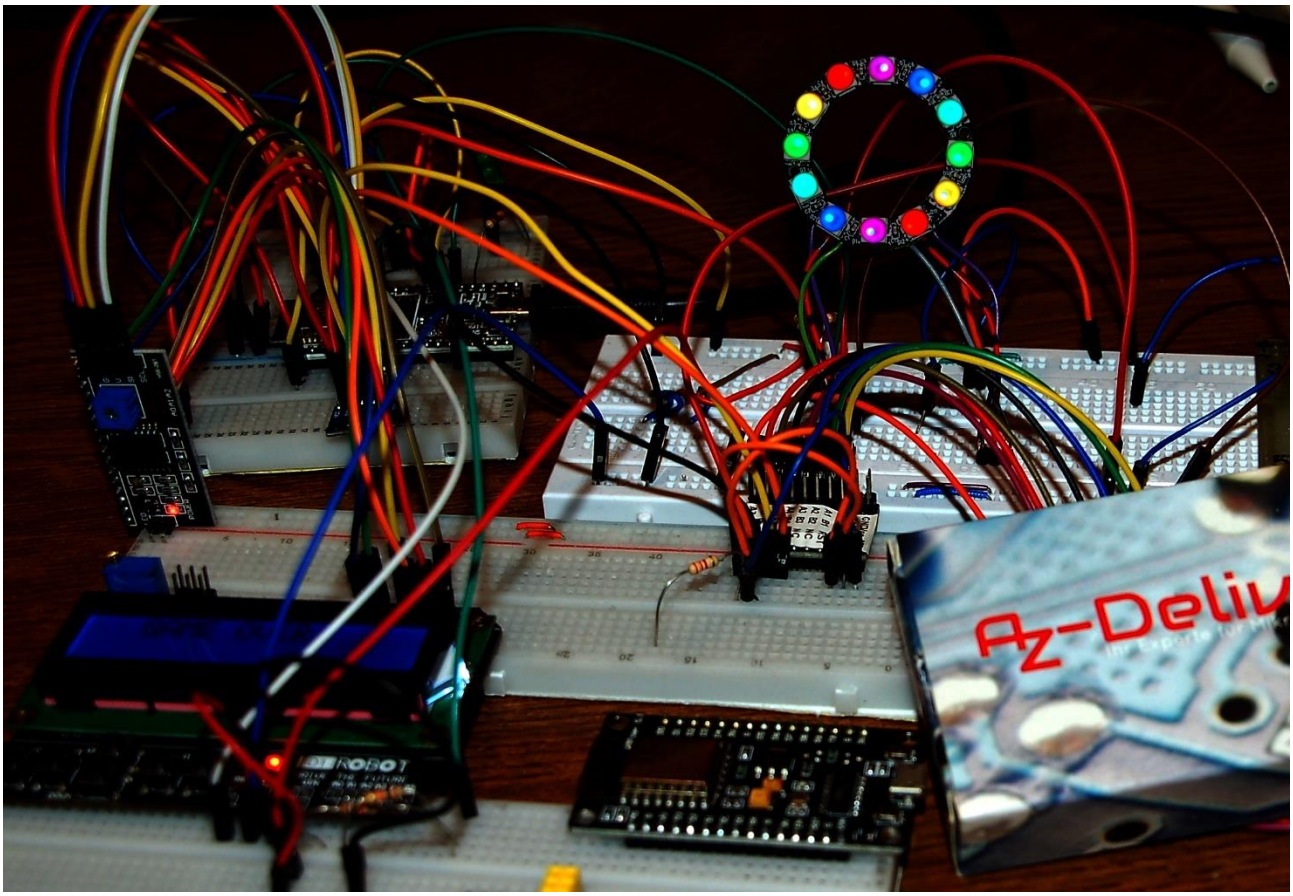


Diesen Beitrag gibt es auch als:
PDF in deutsch

This episode is also available as:
PDF in english

In diesem Beitrag stelle ein Spiel vor, bei dem es um das Erraten von Farben geht. Ziel ist es, die vier Farben mit möglichst wenig Durchgängen zu erraten. Da ist Strategie gefragt. Der Hardwareeinsatz ähnelt sehr stark dem in den Blogs zur Anwendung von GPS, SMS und Telefonie mit dem ESP32. Jedoch fehlt dieses Mal das Thema Funkverbindungen völlig. Auch Sensoren werden keine gebraucht. Seien Sie gespannt. Damit willkommen bei der Vorstellung des Spiels

Ring Master 1 –
Spiele mit dem ESP32 in MicroPython

Das LCD-Keypad aus anderen Blogfolgen kommt wieder zum Einsatz, die Tasten sowie das Display. Die RST-Taste erfüllt erneut die Funktion einer Notbremse. Alternativ zum LCD-Keypad werde ich in der zweiten Folge ein OLED-Display vorstellen, weil bei der Auswertung der Spielergebnisse dann ein paar potentielle Textzeilen mehr nicht schaden. Natürlich bedarf es dann auch einer anderen Tastatur.

Die Steuerung des Spiels passiert in dieser Folge über die fünf Tasten des LDC-Keypads. UP und DOWN blättern in der Liste der sechs LED-Farben, während LEFT und RIGHT eine der vier LED-Positionen ansteuern. Die Taste RST, wird bei mir während der Entwicklung gerne als Notbremse genutzt.

Notbremse benutzen heißt, punktgenau das laufende Programm beenden, ohne sofortigen Neustart. Der Name "Reset-Taste" ist daher eher irreführend und trifft nur dann zu, wenn man das Keypad-Shield im Dunstkreis des Arduino verwendet, wo es nativ hingehört.

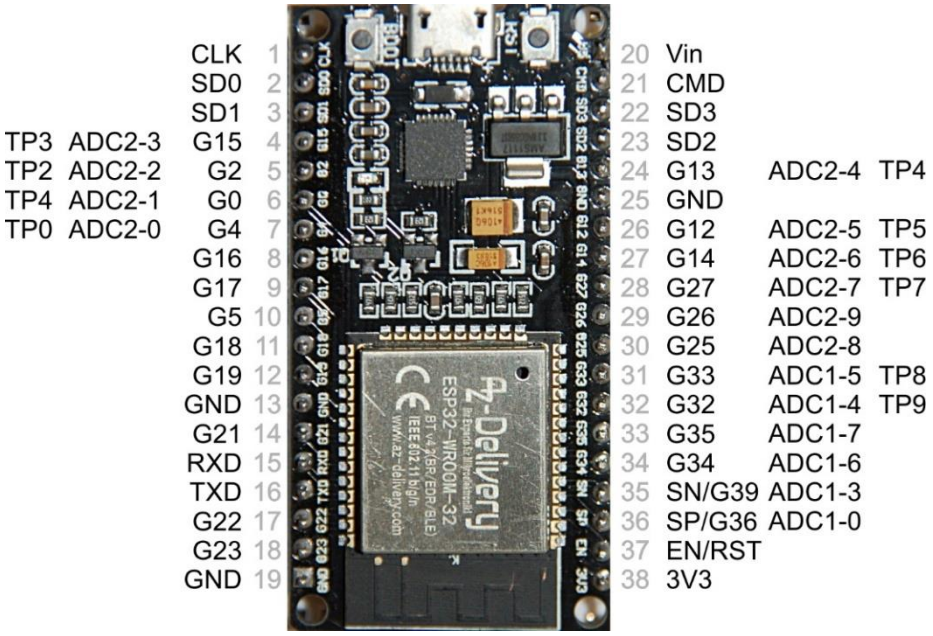
Alle, bis zum Abbruch erstellten Objekte, Variableninhalte und Funktionsdefinitionen, bleiben für den manuellen Zugriff über REPL, die MicroPython-Kommandozeile, erhalten. Auf diese Weise lassen sich zum Beispiel Funktionen und Programmteile testen, ohne vorher einen ganzen Rattenschwanz an Imports und Declarationen etc. jedes Mal neu eingeben zu müssen. Dass über die REPL-Kommandozeile solche Tests einfach durchgeführt werden können, ist ein entscheidender Vorteil der MicroPython-Umgebung.

Hardware

Für "Ring Master" wird also ein MicroPython-Programm erstellt. Das heißt wir brauchen einen MicroPython-fähigen Controller. Die Wahl fiel auf einen ESP32, denn es soll kein großer Bildschirm wie beim Raspi, sondern nur ein LCD angesteuert werden. Der ESP8266-12F scheidet wegen zu wenig RAM-Speicher aus, ihm fehlen gut 1200 Bytes. Als Tastatur begnügen wir uns vorerst, wie schon erwähnt, mit den fünf Tasten des LCD-Keypads. Das Display wird über einen I2C-Parallel-Adapter bedient, der gleichzeitig die Pegelanpassung von 3,3V am ESP32 zu 5V am LCD-Keypad erledigt. Für den Neopixelring gibt es in der MicroPython-Firmware ein bereits eingebautes Modul, das die Programmierung kinderleicht macht. Zur Funktion des Rings folgen weiter unten einige Anmerkungen. Seine Stromaufnahme liegt bei ca. 20mA.

1	ESP32 NodeMCU Module WLAN WiFi Development Board mit CP2102 oder ähnlich
1	LCD1602 Display Keypad Shield HD44780 1602 Modul mit 2x16 Zeichen
1	I2C IIC Adapter serielle Schnittstelle für LCD Display 1602 und 2004
4	Widerstand 10kΩ
1	Battery Expansion Shield 18650 V3 inkl. USB Kabel
1	Li-Akku Typ 18650
1	LED Ring 5V RGB WS2812B 12-Bit 37mm oder ähnlich

Die Schaltung für "Ring Master 1" wird zum großen Teil aus der [fünften Folge des GPS-Blogs](http://www.grzesina.de/az/gps/teil2/gps_mcp_teil2_ger.pdf)

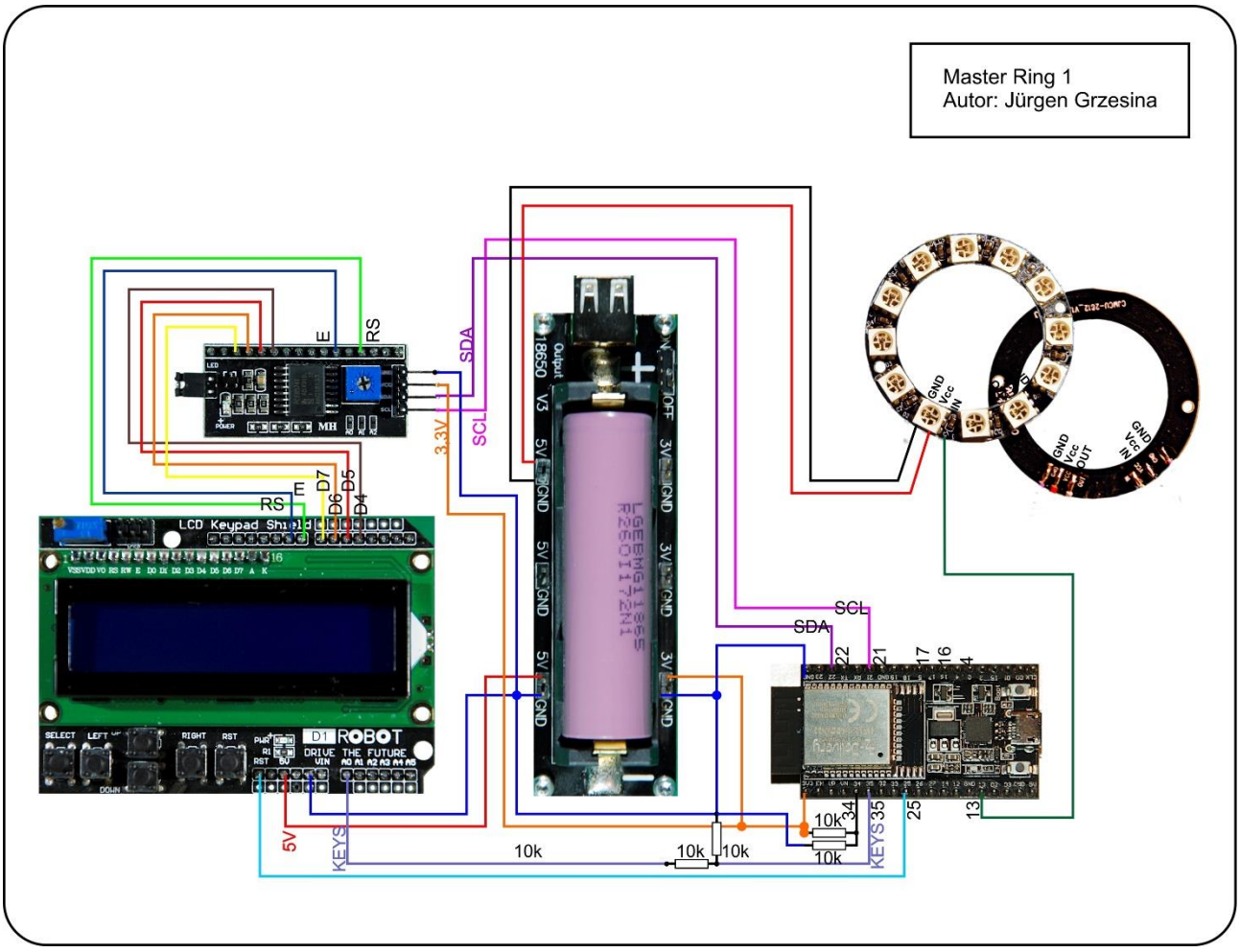


übernommen. Falls Sie statt des Batteriehalters und des Li-Akkus ein 5V-Netzteil verwenden wollen, müssen Sie die 5V an den Pin 20, Vin, des ESP32 legen. Der 3,3V-Pin des ESP32 versorgt dann den I2C-Parallelwandler mit.

*

Die Versorgung aus einem 4,5V-Block aus Alkalizellen wäre zwar für das Controllerboard ausreichend, aber das Display gibt sich damit nicht zufrieden. Für Versorgungsspannungen über 5V muss ein extra 5V-Regler verwendet werden, denn der Neopixelring darf nicht mehr als 5,3V abbekommen. Zum Experimentieren eignen sich alte PC-Netzteile sehr gut, weil sie neben 5V auch 3,3V und 12V zur Verfügung stellen.

Die folgende Abbildung zeigt das Schaltschema. Ein besser lesbares Exemplar in DIN A4 können Sie als PDF-Datei downloaden.



Auf dem LED-Ring sind 12 Neopixel-LEDs vom Typ WS2812B verbaut. Die Spannungsversorgung erfolgt parallel. Die Datenleitung führt seriell von einer LED-Einheit zur nächsten und stellt eine besondere Art von Bus dar. Jede Einheit enthält eine RGB-LED und einen Controller, der auf die erste ankommende 24-Bit-Folge der Farbinformation reagiert. Die Signale werden von einem Microcontroller wie dem ESP32 erzeugt. Je Neopixel-Einheit werden 24 Bit generiert (jeweils 8 für grün, rot und blau). Die Dauer für ein Bit ist 1,25µs +/-0,150µs, die Übertragungsfrequenz beträgt somit ca. 800kHz. Für eine 1 liegt die Leitung 0,8µs auf HIGH und 0,45µs auf LOW, eine 0 wird durch 0,4µs HIGH und 0,85µs LOW codiert. Die ersten ankommenden 24 Bits verarbeitet jede WS2812B-Einheit selbst, alle nun folgenden werden verstärkt und an die nächste Einheit weitergereicht. Die Signalfolge vom Microcontroller wird also von LED zu LED um 24 Bit kürzer. Anders als bei einem üblichen Datenbus erhalten die WS2812B-Einheiten die Daten aber nicht gleichzeitig, sondern zeitversetzt um jeweils 24Bit mal 1,25µs/Bit = 30µs.

Ein Framebuffer im RAM des ESP32 speichert die Farbwerte (3 x 256 = 16,7 Mio.) zwischen, und der Befehl NeoPixel.write() schickt die Informationen über den "Bus", der an einem GPIO-Ausgang hängt (bei uns GPIO13). Mehrere Ringe kann man genau so wie einzelne LEDs cascadieren, indem man den Eingang des nächsten Rings mit dem Ausgang des Vorgängers verbindet. Die Anschlüsse erfolgen rückwärtig, am besten mittels dünner Litzen. Um die Augen zu schonen, verwende ich als Helligkeitsstufe maximal 32. Die Gesamtstromaufnahme des Rings beläuft sich dadurch im Mittel auf weniger als 20mA. Die Komponenten für die Mischfarben ermittelt man am einfachsten experimentell über REPL. Die Helligkeit der einzelnen Teil-LEDs ist recht unterschiedlich. Die Farbcodes in den Tupels werden also selten den gleichen Wert haben.

```
>>> from neopixel import NeoPixel
>>> neoPin=Pin(13)
>>> neoCnt=12
>>> np=NeoPixel(neoPin,neoCnt)
>>> np[0]=(32,16,0)
>>> np.write()
```

Zum Abgleich werden die beiden letzten Befehle mit anderem RGB-Code wiederholt. Die hier angegebenen Werte erzeugen "gelb".

Bei voller Leuchtkraft saugen die LED-Einheiten 50mA pro Stück, was eine gute Konstantspannungsquelle und eine Kühlung des Rings erforderlich macht.

Die Software

Verwendete Software:
Fürs Flashen und die Programmierung des ESP32:
Thonny oder
µPyCraft

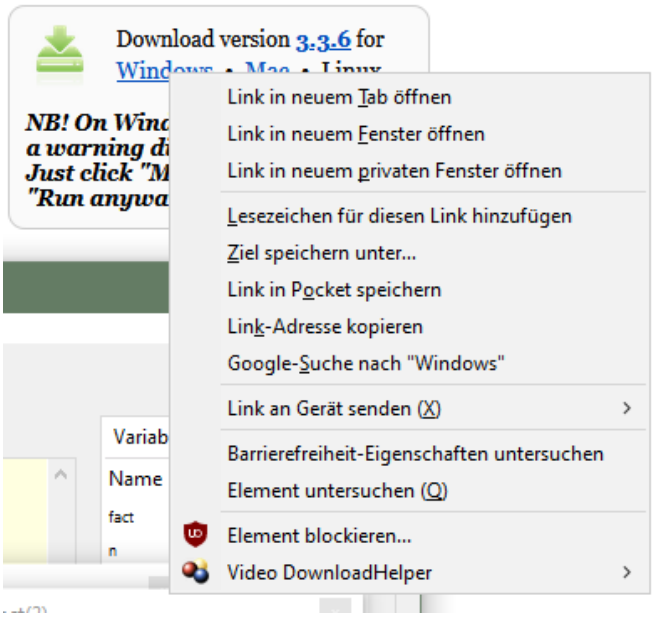
Verwendete Firmware:
MicropythonFirmware
Bitte eine Stable-Version aussuchen

MicroPython-Module und Programme
LCD-Standard-Modul
HD44780U-I2C-Erweiterung zum LCD-Modul
keypad.py Modul für Tastenfeld-Unterstützung
mcp.py Modul für Porterweiterungsbaustein MCP23017
ringmaster1.py Hauptprogramm

Die Entwicklungsumgebung – Beispiel: Thonny

Thonny ist unter MicroPython das Gegenstück zur Arduino-IDE. In Thonny sind ein Programmeditor und ein Terminal sowie weitere interessante Entwicklungstools in einer Oberfläche vereint. So haben sie das Arbeitsverzeichnis auf dem PC, das Dateisystem auf dem ESP32, Ihre Programme im Editor, die Terminalconsole und zum Beispiel den Object inspector in einem Fenster übersichtlich im Zugriff.

Die Ressource zu Thonny ist die Datei thonny-3.3.x.exe , dessen neuste Version direkt von der Produktseite heruntergeladen werden kann. Dort kann man sich auch einen ersten Überblick über die Eigenschaften des Programms verschaffen.

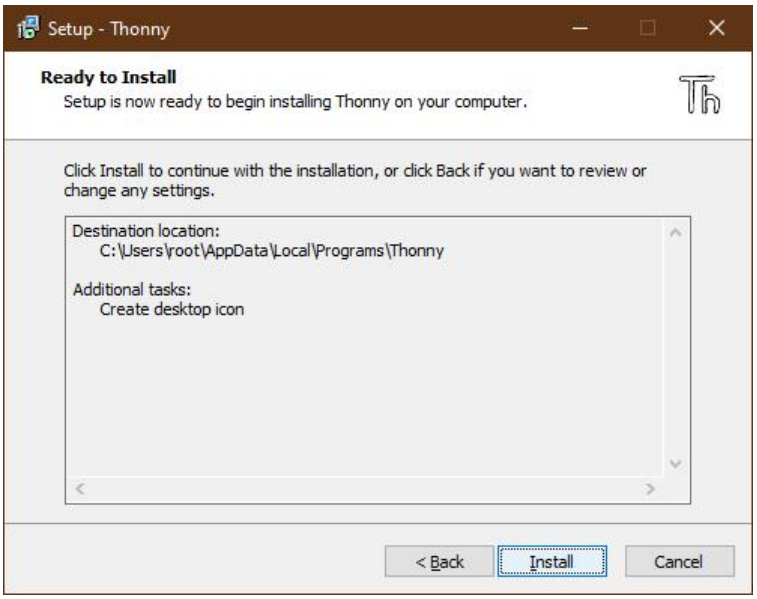


Mit Rechtsklick auf **Windows** und **Ziel speichern unter** laden Sie die Datei in ein beliebiges Verzeichnis Ihrer Wahl herunter. Alternativ können Sie auch diesem Direktlink folgen.
Im Bundle von **Thonny** sind neben der IDE selbst auch **Python 3.7** für Windows und **esptool.py** enthalten. Python 3.7 (oder höher) ist die Grundlage für Thonny und esptool.py. Beide Programme sind in Python geschrieben und benötigen daher die Python-Laufzeitumgebung. **esptool.py** dient unter

anderem auch in der Arduino-IDE als Werkzeug, um Software auf den ESP32 (und andere Controller) zu transferieren.

Starten Sie jetzt die Installation von Thonny durch Doppelklick auf ihre heruntergeladene Datei, wenn Sie die Software nur für sich selbst nutzen möchten. Wenn Thonny & Co. auf Ihrem Rechner allen Usern zur Verfügung stehen soll, müssen Sie die exe-Datei als Administrator ausführen. In diesem Fall klicken Sie rechts auf den Dateieintrag im Explorer und wählen **Als Administrator ausführen**.

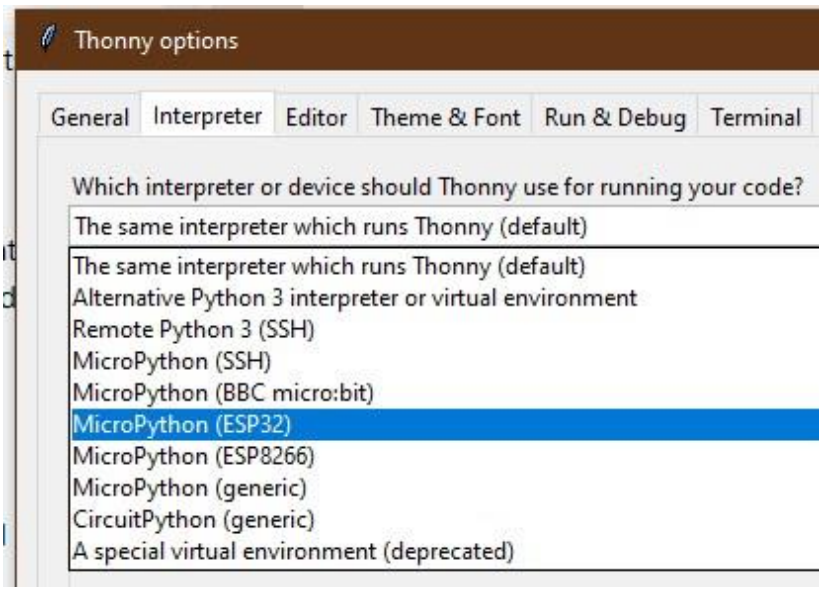
Sehr wahrscheinlich meldet sich der Windows Defender (oder Ihre Antivirensoftware). Klicken Sie auf **weitere Informationen** und im folgenden Fenster auf **Trotzdem ausführen**. Folgen Sie jetzt einfach der Benutzerführung mit **Next**.



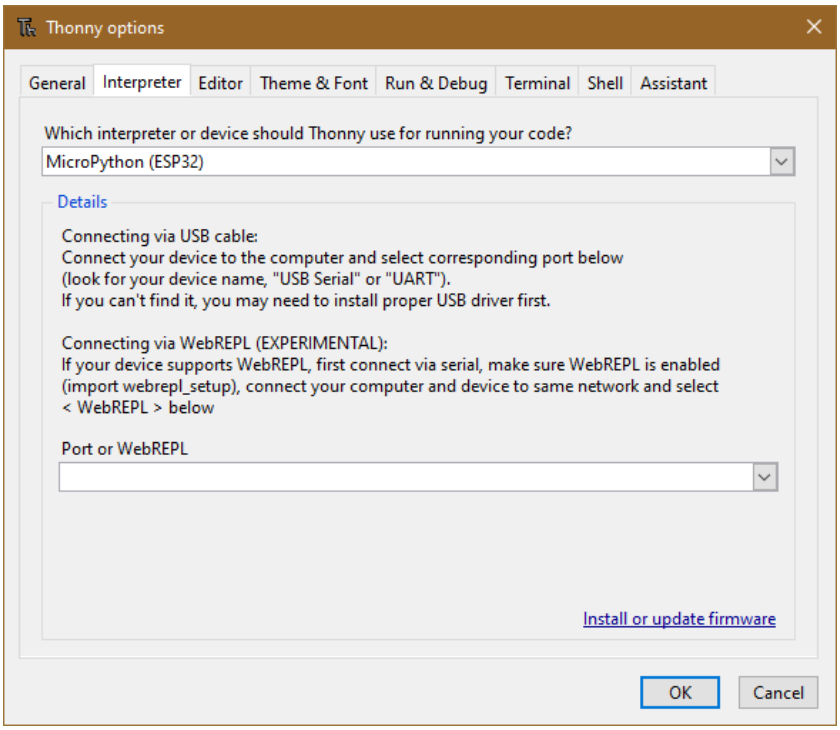
Mit Klick auf **Install** startet der Installationsprozess.

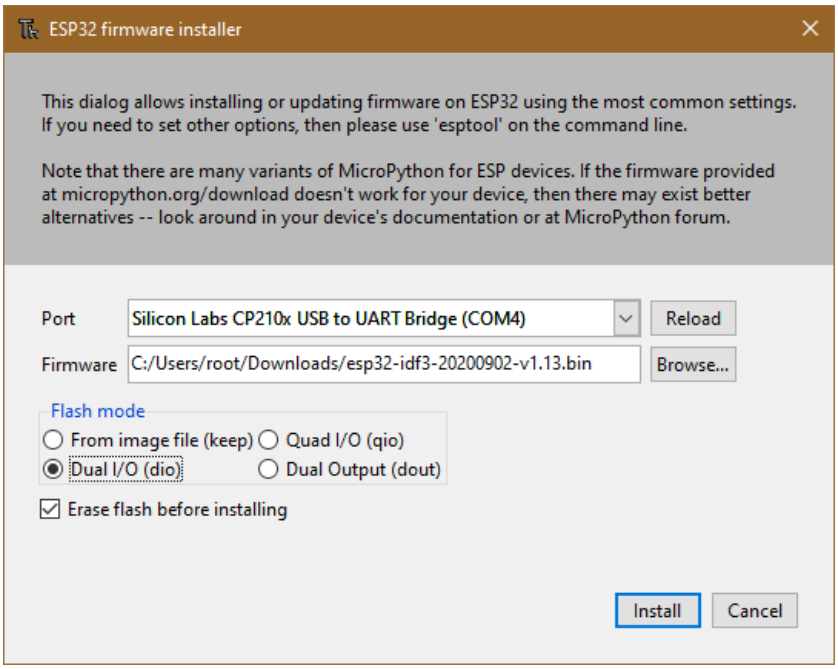
Beim ersten Start geben Sie die Sprache an, dann wird das Editorfenster zusammen mit dem Terminalbereich angezeigt.

Stellen Sie als erste Aktion den verwendeten Controllertyp ein. Mit **Run – Select Interpreter …** landen Sie in den Optionen. Für dieses Projekt stellen Sie bitte Micropython (ESP32) ein.



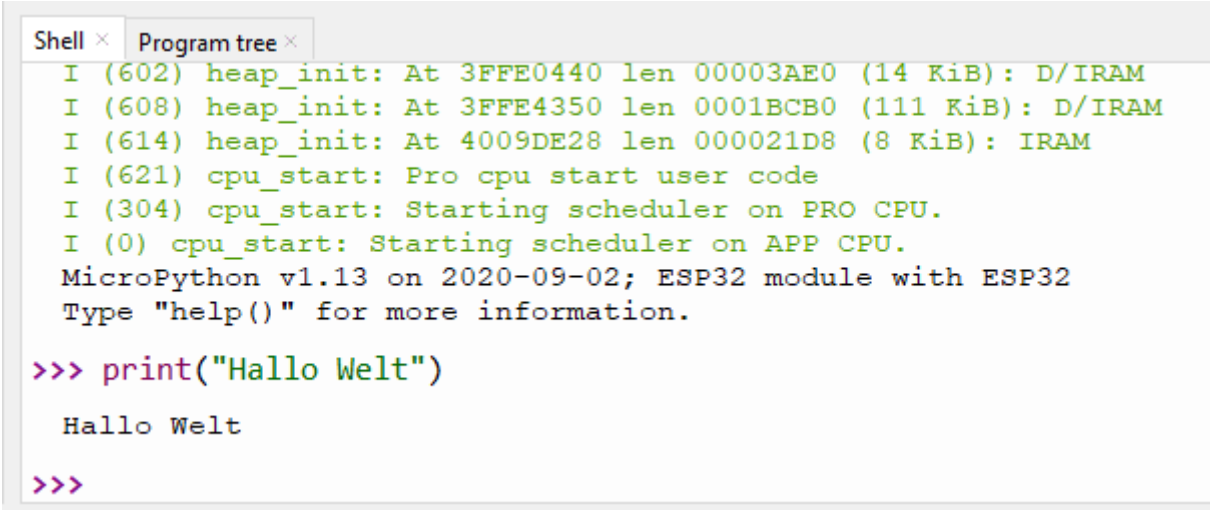
Laden Sie jetzt die Firmware Micropython für den ESP32 von der Auswahlseite herunter und speichern Sie diese Datei in einem Verzeichnis Ihrer Wahl. Die bin-Datei muss als erstes auf den ESP32 transferiert werden. Das geschieht auch mit Thonny. Rufen Sie wieder mit **Run – Select Interpreter … Thonny Options** auf. Rechts unten klicken Sie auf **Install or update Firmware**.





Wählen Sie den seriellen Port zum ESP32 und die heruntergeladene Firmwaredatei aus. Mit **Install** starten Sie den Prozess. Nach kurzer Zeit befindet sich die MicroPython-Firmware auf dem Controller und Sie können die ersten Befehle über REPL, die MicroPython-Kommandozeile, an den Controller senden. Geben Sie im Terminalfenster zum Beispiel folgenden Befehl ein.

```
print("Hallo Welt")
```



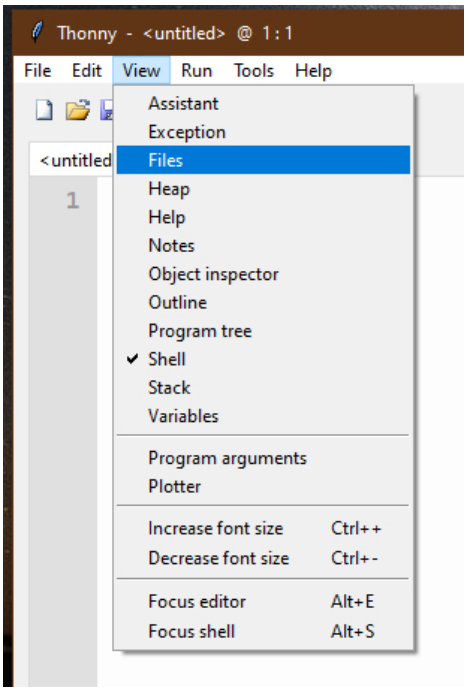
Anders als in der Arduino-IDE können Sie einzelne Befehle an den ESP32 senden und er wird, so es MicroPython-Anweisungen sind, brav antworten. Senden Sie dagegen einen für den MicroPython-Interpreter unverständlichen Text, wird er sie mit einer Fehlermeldung darauf aufmerksam machen.

>>> print"hallo nochmal"

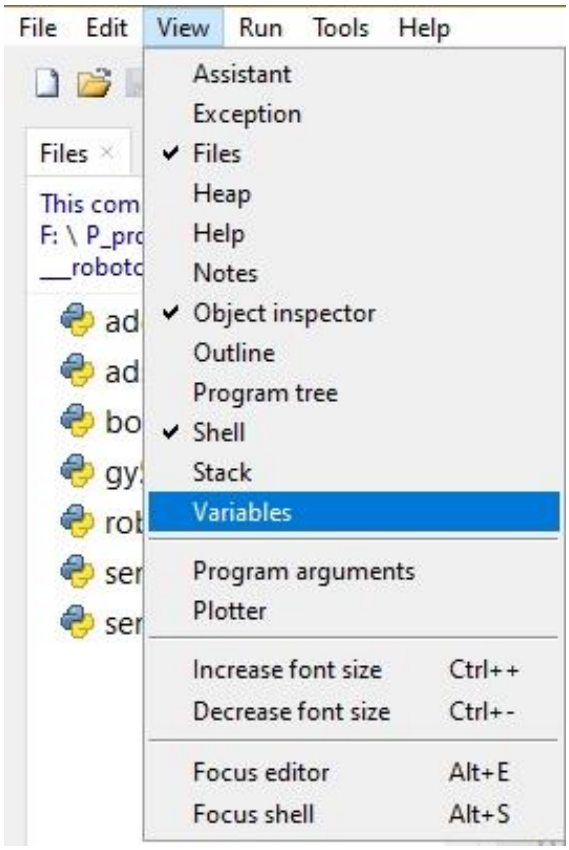
SyntaxError: invalid syntax
Traceback (most recent call last):
  File "<stdin>", line 1
SyntaxError: invalid syntax

Zum Arbeiten fehlt jetzt aber noch die Übersicht über den Workspace und das Device Directory. Der Workspace ist ein Verzeichnis auf dem PC, in dem sich alle, der für ein Projekt wichtigen, Dateien befinden. In Thonny ist sein Name **This Computer**. Das Device Directory ist dazu das Gegenstück auf dem ESP32. In Thonny heißt es **MicroPython device.** Sie bringen es folgendermaßen zur Anzeige.

Klicken Sie auf **View** und dann auf **Files**



Jetzt werden beide Bereiche, oben der Workspace und unten das Device Directory, angezeigt. Weitere Tools blenden Sie über das Menü **View** ein.



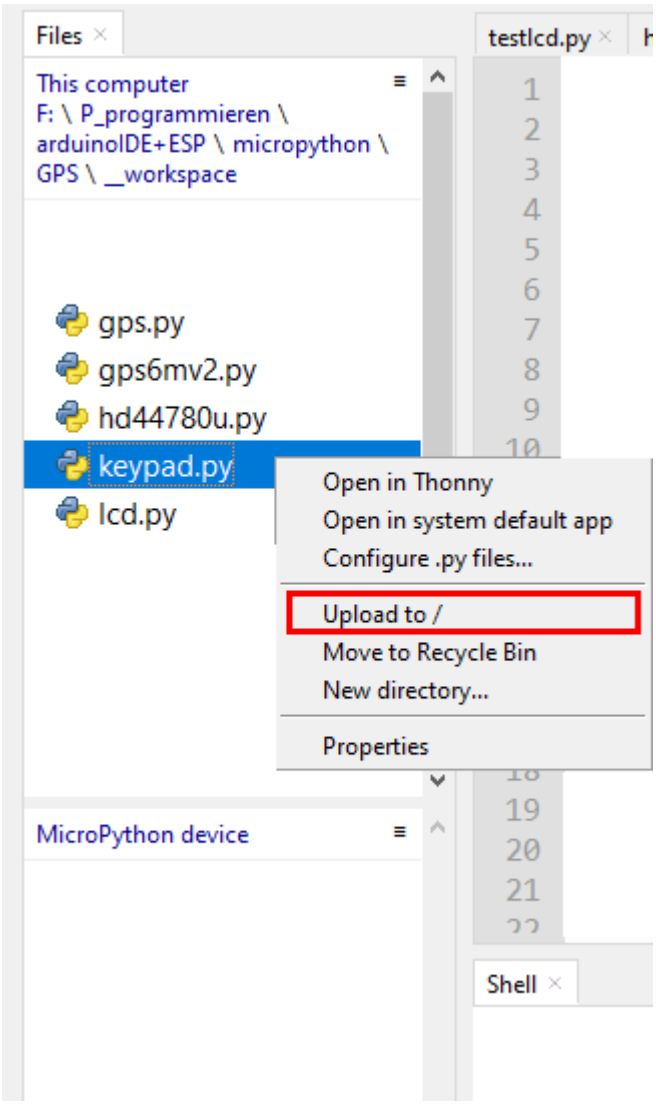
Unsere Programme geben wir im Editorbereich ein. Für ein neues Programm öffnen Sie ein Editorfenster durch Klick auf die Schaltfläche **New** oder durch Tastenfolge **Strg+N**.

In der Arduino-IDE werden Libraries bei jeder Übersetzung des Programms neu übersetzt und in den Programmtext eingebunden. In MicroPython müssen Sie fertige Module, sie

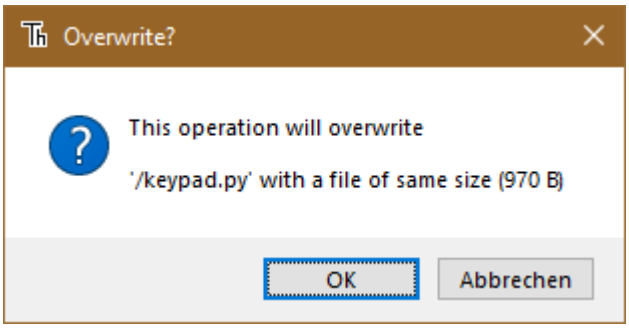
entsprechen den Libraries der Arduino-IDE, nur einmal am Beginn in den Flash des ESP32 hochladen. Ich zeige das an einem Beispiel.

Legen Sie im Explorer in einem beliebigen Verzeichnis einen Projektordner auf Ihrem Rechner an. In diesem Verzeichnis erzeugen Sie einen Ordner mit dem Namen workspace. Alle weiteren Aktionen starten in diesem Verzeichnis und alle Programme und Programmteile werden dort wohnen.

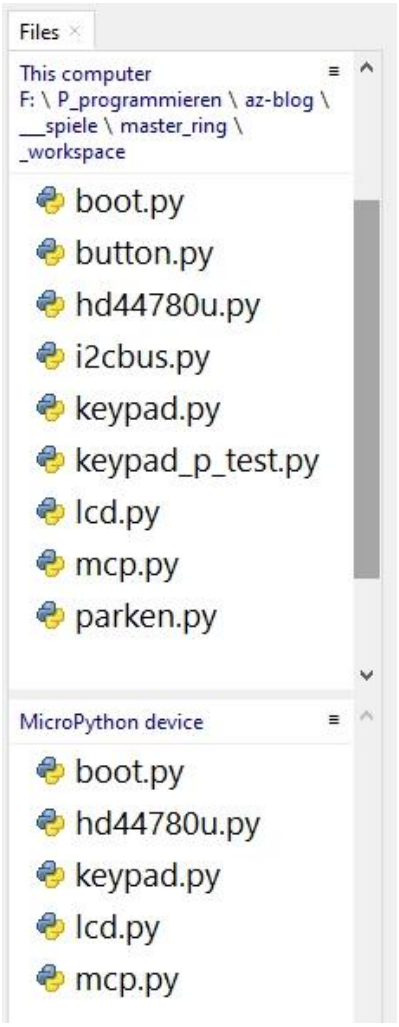
Im Projekt wird die Klasse KEYPAD benötigt. Der Text dazu steht in der Datei keypad.py. Laden Sie am besten gleich alle Module in Ihren workspace. Starten Sie jetzt, falls noch nicht geschehen, Thonny und navigieren Sie im Fenster "This Computer" zu Ihrem Arbeitsverzeichnis. Im workspace sollten jetzt die heruntergeladenen Dateien erscheinen. Ein Rechtsklick öffnet das Kontextmenü, und mit Klick auf **Upload to /** wird der Vorgang gestartet.



Haben Sie an einem Modul etwas geändert, muss dieses, aber auch nur dieses, erneut hochgeladen werden. Die Sicherheitsabfrage zum Überschreiben beantworten Sie dann mit **OK**.



Nach dem Hochladen der ersten 4 Module sieht das dann so aus. Die Datei boot.py im Device Directory wird beim Flashen der Firmware automatisch angelegt. In diese Datei werden wir am Schluss, wenn alles getestet ist, den Inhalt unseres Programms kopieren. Danach wird der ESP32 bei jedem Start das Programm autonom ausführen. Eine Verbindung zum PC ist dann nicht mehr nötig.



Tricks und Infos zu MicroPython

MicroPython ist eine Interpretersprache. Der Hauptunterschied zur Arduino-IDE, wo Sie stets und ausschließlich ganze Programme flashen, ist der, dass Sie die MicroPython-Firmware nur einmal zu Beginn auf den ESP32 flashen müssen, bevor der Controller MicroPython-Anweisungen versteht. Sie können dazu Thonny, µPyCraft oder esptool.py benutzen. Für Thonny habe ich den Vorgang oben beschrieben.

Sobald die Firmware geflasht ist, können Sie sich zwanglos mit Ihrem Controller im Zwiegespräch unterhalten, einzelne Befehle testen und sofort die Antwort sehen, ohne vorher ein ganzes Programm compilieren und übertragen zu müssen. Genau das stört mich nämlich an der Arduino-IDE. Man spart einfach enorm Zeit, wenn man einfache Tests der Syntax und der Hardware bis hin zum Ausprobieren und Verfeinern von Funktionen und ganzen Programmteilen, über die Kommandozeile vorab prüfen kann, bevor man ein Programm daraus strickt. Zu diesem Zweck erstelle ich auch gerne immer wieder kleine Testprogramme. Als eine Art Macro fassen sie wiederkehrende Befehle zusammen. Aus solchen Programmfragmenten entwickeln sich dann mitunter ganze Anwendungen. Soll das Programm autonom mit dem Einschalten des Controllers starten, kopieren Sie den Programmtext in eine neu angelegte Blankodatei. Speichern Sie diese Datei unter boot.py im Workspace ab und laden Sie sie zum ESP32/ESP8266-01 hoch. Beim nächsten Reset oder Einschalten startet das Programm automatisch.

Manuell gestartet werden Programme aus dem aktuellen Editorfenster in der Thonny-IDE über die Taste F5. Das geht schneller als der Mausklick auf den Startbutton oder über das Menü **Run**. Lediglich die im Programm verwendeten Module müssen sich im Flash des ESP32 befinden.

Sollten Sie den Controller später wieder zusammen mit der Arduino-IDE verwenden wollen, flashen Sie das Programm einfach in gewohnter Weise. Allerdings hat der ESP32/ESP8266 dann vergessen, dass er jemals MicroPython gesprochen hat. Umgekehrt kann jeder Espressif-Chip, der ein compiliertes Programm aus der Arduino-IDE oder die AT-Firmware oder LUA enthält, problemlos mit der MicroPython-Firmware versehen werden. Der Vorgang ist immer wie oben beschrieben.

Bevor wir mit der Programmierung des Spiels loslegen, mache ich Sie noch kurz mit ein paar wichtigen Strukturen und Befehlen von MicroPython bekannt, die ich im Programm verwendet habe.

Datenfelder

Neben den einfachen Datentypen wie Ganz- und Fließkommazahlen, Strings (aka Zeichenketten) und den boolschen Werten True und False sowie dem exotischen None treffen wir im Programm auf Listen, Dicts (Dictionaries aka Hasharrays) und Tupel. Listen ersetzen die Arrays, die Sie vielleicht aus der Arduino-IDE kennen. Beide Typen gehören zu den iterierbaren Objekten. Das bedeutet, man kann sie mit Hilfe von Schleifen bearbeiten. Das tun wir an verschiedenen Stellen im Programm.

Durch das sogenannte Slicing können Teilmengen von Listen und Strings extrahiert werden. Zu diesem Zweck wird der ":" verwendet. Einzelne Elemente einer Liste werden durch ihren Index adressiert. Das ist die Platznummer in der Liste oder der Schlüsselbegriff in einem Dict. Die folgenden Strukturen kommen in unserem Programm vor. Die folgende Anweisung definiert eine Liste. Anders als in der Arduino-IDE können in einer Liste oder eine Dict Daten verschiedenen Typs vereinigt werden.

Liste=[34,"Test",3.1415,[4,5,6]]
Indizierung: Einzelne Elemente ansprechen
Liste[2] liefert 3.1415, denn die Indizierung beginnt bei 0, nicht bei 1

Slicing: Einen Teilbereich der Liste herausschneiden
Liste[1:3] liefert ["Test", 3.1415], denn die 3 begrenzt den Bereich zwar, bleibt jedoch stets außen vor.
Liste[:2] liefert [34, "Test"], denn der Index 0 muss nicht angegeben werden.
Liste[2 :] liefert [3.1415, [4,5,6]] also alles vom 2. Element bis zum letzten. Listenelemente wie [4,5,6] dürfen auch beliebige Strukturen sein.
Liste[2 : -1] liefert [3.1415], vom 2. Element bis zum letzten ausschließlich
Liste[2:-2] =[] vom 2. Element bis ausschließlich zum vorletzten. Da bleibt nichts übrig.
Liste[-2] = 3.1415, eben das vorletzte Element

palette ist ein Dictionary, dessen Schlüssel Strings sind. Es wird mit geschweiften Klammern geschrieben. Die Werte folgen nach dem ":", hier sind es sogenannte Tupel. Die werden mit runden Klammern geschrieben. Jedes Tupel besteht hier aus den drei Farbinformationen für rot, grün und blau. Kommata trennen die einzelnen Elemente des Tupels. **palette**["yellow"] hat etwa den Wert (32,16,0). Man kann Dicts auch einfach ausdrucken. Allerdings ist nicht garantiert, dass die Elemente in der Reihenfolge ihrer Definition erscheinen. Bei Hashlisten muss man damit leben.

```
>>> palette= { # (r,g,b)
   "red":(32,0,0),
   "green":(0,16,0),
   "blue":(0,0,16),
   "yellow":(32,16,0),
   "magenta":(16,0,8),
   "cyan":(0,16,8),
   "white":(12,12,12),
   "black":(0,0,0)
   }
```

```
>>> palette
{'magenta': (16, 0, 8), 'yellow': (32, 16, 0), 'cyan': (0, 16, 8), 'blue': (0, 0, 16), 'white': (12, 12, 12), 'black': (0, 0, 0), 'red': (32, 0, 0), 'green': (0, 16, 0)}
```

color ist eine normale, indizierte Liste. Sie wird mit eckigen Klammern definiert. Listen beginnen mit dem Index 0. **color**[3] referenziert also zum Beispiel den vierten Eintrag, den String "yellow".

```
color=[
   "red",
   "green",
   "blue",
   "yellow",
   "magenta",
   "cyan",
   "white",
   "black",
```

]

Die Liste color weist den Farbcodes eine eindeutige Platznummer zu. Dadurch ist es möglich, die Farbcodes in palette in Zählschleifen (for-Schleifen) durch Zahlen als Indexwerte gezielt und reproduzierbar anzusprechen. Das machen wir an verschiedenen Stellen im Programm.

Ähnlich wie bei der Zuweisung von einfachen Variablen verhält sich MicroPython auch bei der Zuweisung von Listen.

```
a=7
b=a
ListeA=[1,2,3,4]
ListeB=ListeA
```

b und ListeB sind keine neuen Variablen, sondern nur neue Namen für das gleiche Objekt. Sie können das mit Hilfe von REPL überprüfen.

```
>>> a=7
>>> b=a
>>> a is b
True
>>> id(a)
15
>>> id(b)
15
>>> ListeA=[1,2,3]
>>> ListeB=ListeA
>>> ListeB is ListeA
True
```

Auch ein Blick in den Objekt inspector bestätigt das anhand der Speicheradresse von ListeA und ListeB.

Das müssen wir beachten, wenn wir tatsächlich eine Kopie eines Objekts erzeugen wollen. Bei einfachen Variablen ist das kein großer Act. Sobald a oder b ein neuer Wert zugewiesen wird, trennen sich die Wege der beiden. Aber bei Listen stellen wir einen bösen Seiteneffekt fest.

```
>>> ListeA=[1,2,3]
>>> ListeB=ListeA
>>> ListeA[2] = 5
>>> ListeB
[1, 2, 5]
>>> ListeA is ListeB
True
```

ListeA ist also immer noch identisch mit ListeB. Die beiden Namen verweisen auf dieselbe Speicherstelle. Wird ein Listenelement unter dem Namen ListeA verändert, dann wirkt sich das auch auf den Namen ListeB aus, denn beide zeigen auf dieselbe Speicherstelle, den Beginn der Liste [1,2,5]. Um wirklich eine eigenständige Kopie der ListeA zu erzeugen, müssen alle Elemente einzeln kopiert werden. Das geht außer durch eine for-Schleife

noch viel einfacher durch folgende Schreibweise, die auch im Programm verwendet wird. Solche Feinheiten erscheinen oft unerklärlich und werden häufig auch gern übersehen.

```
>>> ListeB=ListeA[:]
>>> ListeA is ListeB
False
```

colors=len(color) bestimmt die Anzahl von Einträgen in der Liste color.

Mit einem Trick, genannt Comprehension, kann man eine Liste oder ein Dict sogar dazu bringen, sich selbst zu definieren.

```
neoCnt=12
…
…
…
kringel=[7 for i in range(neoCnt)]
```

Unser Neopixelring hat 12 LEDs. Die Liste **kringel** soll als Schattenvariable Farbinformationen für jede LED enthalten und mit 7, also schwarz (= aus), vorbelegt werden. Weil sich die Anzahl an LEDs ändern kann, legen wird die Liste nicht mit konstanter Länge fest. range(neoCnt) umfasst die ganzen Zahlen von 0 bis 11, denn MicroPython schließt die Obergrenze von Bereichen stets aus, wie wir wissen. Die Comprehension

```
7 for i in range(neoCnt)
```

erzeugt also eine Liste mit 12-mal der 7 als Elemente. Ändert man neoCnt auf 36, dann erhält man ohne Programmänderung für **kringel** eine Liste mit 36 Siebenern .

Um die gegenwärtige Belegung von **kringel** darzustellen wird die folgende for-Scheife in der Funktion **lightKringel**() benutzt.

```
for i in range(neoCnt):
    np[i]=palette[color[kringel[i]]]
```

Als Laufindex durchläuft i die Werte von 0 bis 12. kringel[i] enthält den Farbindex 0 bis 5. damit wird der Farbstring aus der Liste color ermittelt, welcher schließlich als Schlüssel aus dem Dict palette das entsprechende Tupel holt. Dieses wird schließlich an die Neopixelliste np[] mit gleichem Index übergeben.

Eine zentrale Operation sollte vielleicht noch kurz besprochen werden, bevor es endgültig ans Programmieren geht. An mehreren Stellen im Programm geht es beim Erhöhen und Verringern von Indizes um die Einhaltung des Wertebereichs. Die Farbnummern dürfen sich nur im Bereich zwischen 0 und 5 bewegen, die LED-Positionen zwischen 0 und 3. Damit nach cyan sofort rot kommt und nicht weiß oder nach black ein Index-Fehler auftritt, weil halt keine Farbe mit der Nummer 8 existiert, muss bei jedem Erhöhen oder Verringern des Index eine Bereichsprüfung stattfinden. Statt umständlich mit if und else habe ich die elegantere Methode der Ringaddition gewählt. Die ganz normale Addition/Subtraktion gepaart mit der Modulo-Division, die statt dem Quotientenwert bei einer Ganzzahl-Division (Operator //) den Rest der Operation liefert. Ihr Operator ist das "%"-Zeichen.

– Wie, Sie meinen, das ist zu kompliziert? – Ja natürlich, ich verstehe, sie ziehen Beispiele der allgemeinen Darstellung vor, kein Problem, also dann:

13 % 6 = 1, weil 13 // 6 = 2 und somit Rest = 1 gilt.
5 % 6 = 5, weil 5 // 6 = 0 und Rest = 5
(3+1) % 6 = 4, weil 4 // 6 = 0 und Rest = 4
aber:
(5+1) % 6 = 0, weil 6 // 6 = 1 und Rest = 0
Damit bleiben wir bei der Addition innerhalb der Gruppe der Zahlen von 0 bis 5 und nach 5 kommt 0 beim Zählen.
Das funktioniert auch mit anderen Summanden als 1
(5+4) % 6 = 3, weil … aber das können Sie jetzt sicher schon selbst begründen.

Spannend wird es beim Subtrahieren.
(4-1) % 6 = 3, klar!
aber was machen wir hiermit?
(0-1) % 6 = ? REPL sagt hier 5. Aber warum?
Eigentlich passiert Folgendes. Um negative Zahlen zu vermeiden, wird zunächst der Modul 6 addiert. Das könnte ich auch mehrmals tun. Modulo 6 sind die Werte 0, 6, 12, … also gleichwertig, sie alle haben den gleichen 6-er-Rest, nämlich 0. Jetzt kann man 1 subtrahieren, ohne in den negativen Bereich zu gelangen.
(0-1) % 6 = (0+6-1) % 6 = 5, weil 5 // 6 = 0 und Rest = 5.
In dieser Form steht es auch im Programm an verschiedenen Stellen. Wenn Sie den Teil der Zahlengerade ausschneiden und zu einem Kringel biegen, haben Sie die grafische Veranschaulichung des Ganzen. Ach ja, bei unserem LED-Ring kommt ja auch nach der LED Nummer 11 die LED 0, Modulo 12.

Damit das Programm ringmaster1.py ausgeführt werden kann, müssen alle oben aufgelisteten Module in den Flashspeicher des ESP32 hochgeladen werden. Das sind die Dateien hd44780u.py, i2cbus.py, lcd.py und mcp.py. Wenn das erledigt ist, können wir das Programm ringmaster1.py im Editorfenster mit F5 starten – sofern die Hardware zusammengebaut und der ESP32 am PC angeschlossen ist.

Hier das Listing des Programms.

```
# ringmaster1.py
# Author: Juergen Grzesina
# Revision: 1.1
#    Score-Bug geloest
#    Anzeige verdeckter Farben korrigiert
#    Abbruch durch Taste A eingebaut
#    kleinere Bugs beseitigt
# Stand: 04.06.2021
# *****************************************************
# Importgeschaeft
# *****************************************************
import os,sys       # System- und Dateianweisungen

import esp          # nervige Systemmeldungen aus
esp.osdebug(None)

import gc           # Platz fuer Variablen schaffen
```

```
gc.collect()
#
from machine import Pin, I2C
from neopixel import NeoPixel
from keypad import KEYPAD_LCD, KEYPAD
#from i2cbus import I2CBus
from time import sleep, time, ticks_ms
from lcd import LCD
from hd44780u import HD44780U, PCF8574U_I2C
#from button import BUTTON32,BUTTONS
#
# ***************** Objekte declarieren ******************
# Pins fuer parallelen Anschluss des 4x4-Pads

i2c=I2C(-1,scl=Pin(21),sda=Pin(22),freq=400000)
#ibus=I2CBus(i2c)

disp=LCD(i2c,adr=0x27,cols=16,lines=2) # LCDPad am I2C-Bus

keyHwadr=0x20 # HWADR des Portexpanders fuer das 4x4-Pad
#kp=KEYPAD_I2C(ibus,keyHwadr) #  Hardware Objekt am I2C-Bus
#cols=(15,5,18,19)
#rows=(13,12,14,27)
#kp=KEYPAD_P(rows,cols) #  HW-objekt mit Parallel-Anschluss
kp=KEYPAD_LCD(pin=35) # LCD-Keypad-Tastatur an ADC35
k=KEYPAD(kp,d=disp) # hardwareunabhaengige Methoden

rstNbr=25
#rst=BUTTON32(rstNbr,True,"RST")
ctrl=Pin(rstNbr,Pin.IN,Pin.PULL_UP)
#t=BUTTONS() # Methoden fuer Taster bereitstellen

neoPin=Pin(13)
neoCnt=12
np=NeoPixel(neoPin,neoCnt)

palette= { # (r,g,b)
    "red":(32,0,0),
    "green":(0,16,0),
    "blue":(0,0,16),
    "yellow":(32,16,0),
    "magenta":(16,0,8),
    "cyan":(0,16,8),
    "white":(12,12,12),
    "black":(0,0,0)
    }

color=[
    "red",
    "green",
    "blue",
    "yellow",
```

```
    "magenta",
    "cyan",
    "white",
    "black",
    ]
colors=len(color)
red=0; green=1; blue=2; yellow=3
magenta=4; cyan=5; white=6; black=7

kringel=[7 for i in range(neoCnt)]
ready=False

gameState=[7,7,7,7]
myState=gameState[:]
positions=len(gameState)
numberOfTrials=0

def lightKringel():
    for i in range(neoCnt):
        np[i]=palette[color[kringel[i]]]
    np.write()

def clearKringel():
    global kringel
    kringel=[7 for i in range(neoCnt)]
    for i in range(neoCnt):
        np[i]=(0,0,0)
    np.write()
    sleep(0.03)

def clearRing():
    for i in range(neoCnt):
        np[i]=(0,0,0)
    np.write()
    sleep(0.03)

def rainbowKringel(colList,cnt=3,delay=0.3):
    colors=len(colList)
    cols=colList[:]
    cols.extend([7 for i in range(colors,neoCnt)])
    colors=len(cols)
    global kringel
    global ready
    ready = False
    clearKringel()
    for m in range(colors):
        #print(m)
        for n in range(m+1):
            kringel[m-n]=cols[n]
            #print(m,n)
        #print(kringel)
        lightKringel()
```

```
        sleep(delay)
    for m in range (cnt-1):
        for k in range(neoCnt):
            h11=kringel[neoCnt-1]
            for n in range(neoCnt-1):
                kringel[(neoCnt-1)-n]=kringel[(neoCnt-1)-n-1]
            kringel[0]=h11
            #print(kringel)
            lightKringel()
            sleep(delay)
    ready=True

def dimKringel(delay=0.1,stufen=8,down=True):
    global ready
    ready=False
    for h in range(stufen+1):
        for i in range(neoCnt):
            r,g,b=palette[color[kringel[i]]]
            if down:
                col=(r*(stufen-h))//stufen
                rn=(col if h<stufen else 0)
                col=(g*(stufen-h))//stufen
                gn=(col if h<stufen else 0)
                col=(b*(stufen-h))//stufen
                bn=(col if h<stufen else 0)
            else:
                col=(r*(h))//stufen
                rn=(col if h<stufen else r)
                col=(g*(h))//stufen
                gn=(col if h<stufen else g)
                col=(b*(h))//stufen
                bn=(col if h<stufen else b)
            np[i]=(rn,gn,bn)
        np.write()
        sleep(delay)
    ready=True

def blinkKringel(on=0.3,off=0.7,cnt=1):
    c=cnt
    for i in range (c):
        lightKringel()
        sleep(on)
        clearRing()
        sleep(off)

def randomKringel():
    global kringel
    kringel=[int(i)%7 for i in os.urandom(neoCnt)]
    lightKringel()

def showStatus(stat):
    for i in range(len(stat)):
```

```
        kringel[i*3]=stat[i]
    lightKringel()

def initGame():
    clearKringel()
    edge=[int(i)%6 for i in os.urandom(4)]
    #print(edge)
    for i in range(positions):
        #np[i*3]=palette[color[edge[i]]]
        print(color[edge[i]],end="*")
        pass
    #np.write()
    print("")
    rainbowKringel([red,yellow,green,cyan,blue,magenta],\
                   cnt=2,delay=0.03)
    dimKringel(stufen=8)
    clearKringel()
    return edge # goes to gameState

def startGame():
    # Keyblock:
    # Taste  Funktion
    # *      Position back  (n+9)%12 (n+3)%4
    # #      Position next  (n+3)%12 (n+1)%4
    # A      Abbruch
    # D      OK, set myStatus
    state=[7,7,7,7]
    clearKringel()
    getColorStatus(state)
    return state

def compareToSolution(mystat):
    global kringel
    global numberOfTrials
    numberOfTrials+=1
    reply=True
    for i in range(positions):
        kringel[3*i+1]=7
        if mystat[i]==gameState[i]:
            kringel[3*i+1]=gameState[i]
            reply=reply & True
        elif mystat[i] in gameState:
            kringel[3*i+1]=6
            reply=False
        else:
            reply=False
    lightKringel()
    sleep(0.3)
    return reply

def getColorStatus(myStat,delay=0.3):
    ms=myStat
```

```
    showStatus(ms)
    i=0
    w=ms[i]
    np[neoCnt-1]=palette[color[white]]
    np.write()
    disp.clearAll()
    disp.writeAt("up=2, down=0  {}".format(numberOfTrials),0,0)
    while 1:
        disp.writeAt("Position {}".format(i),0,1)
        ch=k.asciiKey()
        if ch != "\xFF":
            if ch=="*":
                rp=(i*3+(neoCnt-1))%neoCnt
                np[rp]=palette[color[black]]
                i=(i+3)%positions # 1 Position zurueck
                rp=(i*3+(neoCnt-1))%neoCnt
                np[rp]=palette[color[white]]
                np.write()
                w=ms[i]
                sleep(delay)
            elif ch=="+":
                rp=(i*3+(neoCnt-1))%neoCnt
                np[rp]=palette[color[black]]
                i=(i+1)%positions # 1 Position vor
                rp=(i*3+(neoCnt-1))%neoCnt
                np[rp]=palette[color[white]]
                np.write()
                w=ms[i]
                sleep(delay)
            elif ch=="\x0d":
                rp=(i*3+(neoCnt-1))%neoCnt
                np[rp]=palette[color[black]]
                np.write()
                disp.clearAll()
                return ms
            elif ch=="2":
                w=ms[i]
                w=(w+1)%(colors-2) # mod (colors-2) Addition
                np[i*3]=palette[color[w]]
                ms[i]=w
                np.write()
                sleep(delay)
            elif ch=="0":
                w=ms[i]
                w=(w+colors-3)%(colors-2) # mod (colors-2) Subtr.
                np[i*3]=palette[color[w]]
                ms[i]=w
                np.write()
                sleep(delay)
            elif ch=="\x08":
                print("Game Over")
                disp.clearAll()
```

```
                disp.writeAt("  GAME OVER",0,0)
                clearKringel()
                sleep(delay)
                sys.exit()
        if ctrl.value()==0:
            print("Game Over")
            disp.clearAll()
            disp.writeAt("  GAME OVER",0,0)
            sys.exit()

def play(mystat):
    ms=mystat
    showStatus(ms)
    if compareToSolution(ms):
        disp.clearAll()
        return
    while 1:
        ms=getColorStatus(ms)
        showStatus(ms)
        vergleich=compareToSolution(ms)
        if vergleich:
            disp.writeAt("TRIALS: {}".format(numberOfTrials),6,0)
            sleep(1)
            disp.clearAll()
            return
        sleep(0.5)

# ********************************************************
# ******************** Hauptschleife *********************
# ********************************************************
disp.clearAll()
disp.writeAt("RINGMASTER 1",0,0)
disp.writeAt("WELCOME",0,1)
sleep(3)
totalScore=0
games=0
while True:
    numberOfTrials=0
    gameState=initGame()
    clearKringel()
    disp.writeAt("Start now",0,1)
    sleep(1)
    myState=startGame() #[1,0,3,1]
    play(myState)
    totalScore=totalScore+numberOfTrials
    games+=1
    disp.clearAll()
    disp.writeAt("Rounds {}".format(games),0,0)
    disp.writeAt("Total score {}".format(totalScore),0,1)
    sleep(1)
    taste=k.waitForKey(0,ascii=True)
    print("Taste",taste)
```

```
    if taste=="+":
        print("Game Over")
        disp.clearAll()
        disp.writeAt("  GAME OVER",0,0)
        sys.exit()
    disp.clearAll()
```

Das Hauptprogramm fällt durch die Verlagerung der Teilarbeiten auf die diversen Funktionen mit seinen 28 Zeile sehr überschaubar aus. Nach der Begrüßung erzeugt initGame()ein neues 4-Tupel an Farben, die zu erraten sind. Alle Spielfarben marschieren ein und tanzen 3 Reigen. startGame() fordert zum ersten Tanz auf, will sagen zur ersten Farbwahl. Die weiße LED kennzeichnet die Eingabeposition. Das ist stets die nächste LED im Uhrzeigersinn daneben. Mit den Tasten UP und DOWN blättern Sie die Farbskala durch, mit LEFT und RIGHT steuern Sie die nächste LED-Position OST, NORD, WEST, SÜD oder umgekehrt an. Die Auswahl wird mit SELECT übernommen.

Mit der Funktion play() treten Sie in die heiße Spielphase ein. Nach der Überprüfung der ersten Farbwahl, die wohl in den meisten Fällen keinen sofortigen Volltreffer melden wird, werden wir zu einer weiteren Auswahl aufgefordert. Stellt die Überprüfung die Übereinstimmung der Farbfolge des **gameState** mit **myState** fest, haben wir die Farben alle richtig geortet - Volltreffer. Jede richtig erratene Farbe wird durch das Einschalten des gleichen Farbtons auf der im Uhrzeigersinn folgenden LED angezeigt. Ist die von uns gewählte Farbe in der Lösung enthalten, aber in einer anderen Himmelsrichtung zu finden, dann wird uns das durch die Farbe weiß mitgeteilt. Mit jedem SELECT wird die Anzahl der Versuche um 1 erhöht. Dieser Wert wird in der rechten oberen Ecke des Displays angezeigt.

Nach der Feststellung der Übereinstimmung für alle Positionen kehrt das Programm aus der Funktion play() zurück. Der Inhalt der globalen Variable numberOfTrials, die Anzahl an Versuchen, wird zu totalScore addiert. Dieser Wert und die Anzahl an Spielrunden erscheinen in der Anzeige. Nach dem Drücken einer (fast) beliebigen Taste startet eine neue Spielrunde. Die Taste RIGHT beendet das Programm an dieser Stelle.

Noch ein paar Anmerkungen zur Tastenabfrage. Das Modul keypad.py kann die Tasten des LCD-Keypad abfragen, aber genauso gut auch Matrixtastenfelder. Zu einem Vertreter der letzten Kategorie kommen wir in der nächsten Folge. Damit verschiedene Hardwarekomponenten in gleicher Weise abgefragt werden können, enthält das Modul mehrere Klassen. Die Klasse KEYPAD_LCD ist für das LCD-Keypad zuständig. Sie enthält, wie die anderen Klassen auch, eine Methode key(), die an die Hardware angepasst ist und eine Tastennummer zurückgibt. Diese Tastennummer kann nachfolgend in ein ASCII-Zeichen übersetzt werden. Dafür ist der String asciiCode zuständig. Der String enthält so viele Zeichencodes, dass jeder rohen Tastennummer ein ASCII-Zeichen entspricht.

asciiCode="+20*\x0d"

Die Tasten RIGHT, UP, DOWN, LEFT, SELECT entsprechen den rohen Tastencodes 0, 1, 2, 3, 4. Durch die ASCII-Tabelle ergibt sich folgende Zuordnung.

RIGHT	UP	DOWN	LEFT	SELECT
+	2	0	*	ENTER= Linefeed

Sie wundern sich, wie ich auf diese verrückte Festlegung gekommen bin? Gut, ich will es Ihnen verraten. Es hat mit dem nächsten Blogbeitrag zu tun. Dort verwende ich ein OLED-Display, das natürlich keine Tasten besitzt. Also musste ich für die erweiterten Spieloptionen neben der Anzeige auch einen 4x4 Tastaturblock einsetzen. Ja, und auf dieser Matrixtastatur liegt "nach oben" auf der 2, "nach unten" auf der 0 usw.

Durch diesen Trick muss durch die Tastaturumstellung am Programm nur an einer Stelle etwas geändert werden und zwar am Anfang.

```
from keypad import KEYPAD_LCD, KEYPAD
#from keypad import KEYPAD_I2C, KEYPAD
…
…
keyHwadr=0x20 # HWADR des Portexpanders fuer das 4x4-Pad
#kp=KEYPAD_I2C(ibus,keyHwadr) #  Hardware Objekt am I2C-Bus
#cols=(15,5,18,19)
#rows=(13,12,14,27)
#kp=KEYPAD_P(rows,cols) #  HW-objekt mit Parallel-Anschluss
kp=KEYPAD_LCD(pin=35) # LCD-Keypad-Tastatur an ADC35
k=KEYPAD(kp,d=disp) # hardwareunabhaengige Methoden
```

Die Kommentierung der fettdargestellten Zeilen muss umgedreht werden, das ist alles. Auch die Klassen KEYPAD_I2C und KEYPAD_P haben nämlich eine Methode key(), welche eine Tastennummer zurückgibt und eine eigene ASCII-Tabelle für die Übersetzung in ASCII-Zeichen.

Die darübergelegte Klasse KEYPAD stellt, unabhängig von der Hardware, Methoden für das Handling von Tastaturen bereit: waitForKey(), asciiKey() und padInput()

waitForKey nimmt die beiden optionalen Parameter timeout und ascii. timeout ist die Wartezeit in Sekunden bis die Methode entweder -1 oder "\xFF" zurückgibt, falls keine Taste gedrückt wird. Wird rechtzeitig eine Taste gedrückt, dann wird deren rohe Tastennummer oder, falls ascii=True übergeben wurde, das entsprechende ASCII-Zeichen zurückgegeben. timeout=0 wartet ewig auf die Tastenbetätigung.

Die Methode padInput() erwartet an der Position xp, yp auf dem Display die Eingabe einer Zeichenkette, die durch D abgeschlossen wird. Durch geeignete Programmierung und weitere ASCII-Tabellen ließe sich sogar das gesamte Alphabet über das 4x4-Tastenfeld erfassen.

Eine ähnliche Struktur weisen die Module hd44780u.py und lcd.py auf. Im ersten Modul sind Klassen zur Bedienung der Hardware und grundlegenden Ausgabe enthalten. Die

Klasse LCD bietet Methoden an, die in gleicher Funktion auch in der Klasse OLED enthalten sind. Durch Änderung des Imports und der Initialisierung kann jederzeit ein LCD gegen ein OLED-Display ausgetauscht werden. Genau das machen wir in der nächsten Episode Ring Master 2.

Ach, ich vergaß die Erwähnung einer bestimmten Programmzeile. In der Funktion **initGame**() gibt es eine for-Schleife, die einzig und allein zum Schummeln dient.

```
    for i in range(positions):
        #np[i*3]=palette[color[edge[i]]]
        #print(color[edge[i]],end="*")
        pass
    #np.write()
```

Während der Testphase verraten die auskommentierten Zeilen den geheimen Zahlencode für die Farbenvorlage. Danach sollten sie wieder auskommentiert werden, sonst hat der Spaß ein Loch.

Bis dann, viel Vergnügen beim Basteln, Programmieren und Spielen!

PDF in deutsch

PDF in english