

Inhaltsverzeichnis

1: Einleitung, erste Schritte, Sprachstruktur, interaktive Konsole	3
Einleitung	3
Erste Schritte	3
Die Sprachstruktur	4
REPL, die interaktive Konsole	4
2. Kommentare, Blöcke und Operatoren	5
Kommentare	5
Blöcke	5
Operatoren	6
3. Textformatierung	7
Einfache Textausgabe mit Platzhalter	7
Ganze Zahlen	8
Zahlen mit Nachkommastellen	9
Benannte Platzhalter	10
4. Einfache Datentypen	11
Ganze Zahlen (integer)	11
Dezimalzahlen (float)	11
Zeichen(chr)	11
Zeichenketten (String)	11
Boolscher Datentyp	12
Komplexe Zahlen (complex)	12
5-8. Mengentypen (Collections)	13
Set, Tuple, List und Dictionary	13
Eigenschaften	13
Erzeugen	14
Grundlegende Abfragen	15
Auf Untermengen zugreifen	16
Vergleichen	17
Bearbeiten	17
9. Kontrollstrukturen (Bedingungen und Schleifen)	18
Bedingungen: if .. elif .. else	18
while - Schleifen	19
for - Schleifen	20
10. Funktionen	21
Einfache Funktion	21
Mehrere Werte zurückgeben	21
Beliebige Anzahl Argumente	21
Standardwerte und benannte Argumente	22
Mehrere benannte Argumente	22
11. Globale und lokale Variablen	23
Globale Variablen in Funktionen	23
Erzeugen von Variablen	24
Globale Variablen als Argumente	25
12. Micropython, ESP32 und das Multitasking	26
Einfache Threads	26
Threads mit Parameter	27
Wann ist ein Thread beendet?	28
Eine Thread - Liste	29

1: Einleitung, erste Schritte, Sprachstruktur, interaktive Konsole

Einleitung

Dieser Kurs ist als Ergänzung zum Kurs 'Micropython mit ESP32' gedacht. Er gibt eine systematische Einführung in die Sprache Micropython.

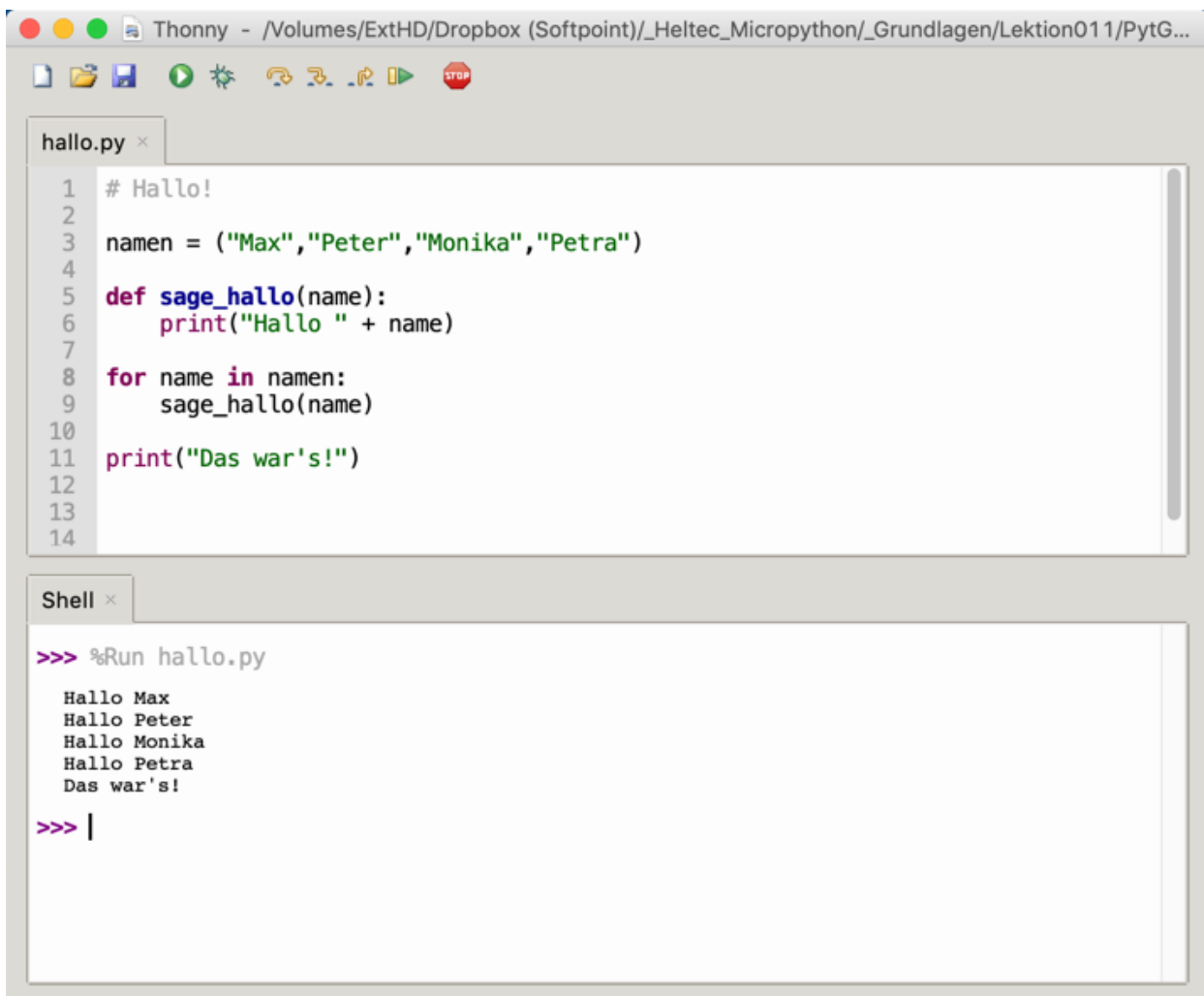
Ich nehme an, dass du die ersten Schritte mit dem ESP32 bereits unternommen hast. Damit sollten Thonny und das Heltec - Board bereits eingerichtet sein. Für diesen Kurs kannst du auch ein beliebiges anderes ESP32 - Board verwenden, falls du in der Lage bist, dieses einzurichten.

Vor Beginn jeder Lektion solltest du das Begleitmaterial herunterladen. Du findest den Link dazu in der Youtube Videobeschreibung oder im entsprechenden Eintrag im Forum.

Darin findest du alle Programmbeispiele (nur die Programme, nicht das was wir direkt in der Konsole eintippen) und die aktuellste Version dieses Dokumentes.

Erste Schritte

Das erste Programm ist sehr einfach. Erzeuge in Thonny ein neues Dokument und speichere es unter dem Namen `hallo.py` auf deiner Festplatte ab. Du kannst es aber auch direkt aus dem Begleitmaterial laden. Achte beim Abschreiben auf Gross- und Kleinschreibung. Das Einrücken nach dem `:` macht Thonny selbstständig. Du solltest das so lassen, da sonst das Programm nicht mehr läuft.



The screenshot shows the Thonny IDE interface. The top window, titled 'hallo.py', contains the following Python code:

```

1  # Hallo!
2
3  namen = ("Max", "Peter", "Monika", "Petra")
4
5  def sage_hallo(name):
6      print("Hallo " + name)
7
8  for name in namen:
9      sage_hallo(name)
10
11 print("Das war's!")
12
13
14


```

The bottom window, titled 'Shell', shows the execution of the script:

```

>>> %Run hallo.py
Hallo Max
Hallo Peter
Hallo Monika
Hallo Petra
Das war's!
>>> |

```

Es ist nicht notwendig, dass du das Programm vor der Ausführung auf dein Board speicherst. Ausführen kannst du das Programm durch Drücken auf den grünen Pfeil. 

Die Sprachstruktur

Dieses Beispiel gibt und die Gelegenheit einen ersten Blick auf die Sprachstruktur von Micropython zu werfen. Im Wesentlichen handelt es sich um Python 3. Da Mikrocontroller aber wesentlich weniger Speicher und Leistung besitzen als Desktopcomputer, wurden einige Anpassungen vorgenommen. Detaillierte Informationen findest du auf der [Webseite von Micropython](#).

Selbstverständlich unterstützt auch Micropython **Kommentare**. Diese werden hier mit # eingeleitet.

```
namen = ("Max", "Peter", "Monika", "Petra")
```

So werden Variablen angelegt. Sie entstehen durch Zuweisung eines Wertes. Dadurch erhalten sie auch gleich ihren Typ.

Als Nächsten finden wir eine Funktionsdefinition. Hier fällt der : und die Einrückung danach auf. Mit : starten wir einen Block (in C wäre das {). Der Block bleibt solange bestehen, wie wir die Einrückung beibehalten.

```
def sage_hallo(name):  
    print("Hallo " + name)
```

Ausserhalb des Blocks geht es mit

```
for name in namen:  
    sage_hallo(name)
```

weiter. Auch hier haben wir einen Block.

Das Ganze beenden wir mit dem Schlusssatz.

```
print("Das war's!")
```


In Python wird normalerweise nur ein Befehl pro Zeile geschrieben. Der Befehl muss auch nicht mit ; abgeschlossen werden.

REPL, die interaktive Konsole

REPL steht für read-evaluate-print loop. Diese Konsole kann Befehle entgegennehmen, ausführen und Ergebnisse ausgeben.

Python ist eine interpretierte Sprache. Wir können jeden Befehl direkt eingeben und ausführen lassen. Es ist nicht notwendig, die Befehle zuerst in Maschinensprache zu übersetzen.

Hier einige Beispiele:



```
Shell <
>>> 4 + 5
9
>>> print(4 * 5)
20
>>> s = "25 / 3"
>>> print(s)
25 / 3
>>> print(eval(s))
8.333333
>>> |
```

2. Kommentare, Blöcke und Operatoren

Kommentare

In jedem Programm sind erläuternde Kommentare wichtig. Nur so weiss man nach Jahren noch, was man sich damals dabei gedacht hat.

Es gibt einfache Kommentare, die immer mit # eingeleitet werden. Sie beginnen mit # und enden mit dem Zeilenende.

```
a = 5  # Die Variable a wird auf 5 gesetzt
```

Mehrzeilige Kommentare werden mit """ oder ''' eingeleitet. Sie werden mit derselben Zeichenfolge beendet.

```
"""
erste Kommentarzeile
zweite Kommentarzeile
"""
```

oder

```
'''
erste Kommentarzeile
zweite Kommentarzeile
'''
```

Blöcke

In der Sprache C werden Blöcke bekanntlich in { } eingeschlossen. Python verwendet hier ein etwas anderes Konzept. Einrückungen, die in C nur der Übersichtlichkeit dienen, sind in Python Bestandteil der Syntax und helfen, Blöcke zu definieren.

```
def addiere(a,b):    # Mit dem : wird der Block eingeleitet
    c = a + b        # Nach dem : wird eingerückt
    print(c)         # Diese Einrückung bleibt für alle Befehle
                    # innerhalb des Blockes erhalten

addiere(3,5)         # Keine Einrückung mehr! Damit gehört dieser Befehl
                    # nicht mehr zum Block
```

In Bedingungen und Schleifen werden Blöcke auf identische Art gebildet.

Operatoren

Arithmetische Operatoren

+, -	Addition Subtraktion	$c = a + b$ $c = a - b$
*, /	Multiplikation Division	$c = a * b$ $c = a / b$
//	Ganzzahlige Division	$11 // 3 ==> 3$
%	Modulo (Rest der Division)	$11 \% 3 ==> 2$
**	Exponent	$2 ** 3 ==> 8$

Logische Operatoren

and, or, not Boolesche Operatoren

Bitoperatoren

&	Bitweise AND - Verknüpfung
	Bitweise OR - Verknüpfung
^	Bitweise XOR - Verknüpfung
~	Bitweises NOT
<<	Bits nach links verschieben
>>	Bits nach rechts verschieben

Vergleiche

==	ist gleich	$3 == 5$ # das ergibt False
!=	ist ungleich	$3 != 5$ # das ergibt True
>	grösser	$3 > 5$ # das ergibt False
<	kleiner	$3 < 5$ # das ergibt True
<=	kleiner oder gleich	
>=	grösser oder gleich	
in	ist enthalten in	"Max" in ("Max", "Peter")
not in	ist nicht enthalten in	"Fritz" in ("Monika", "Petra")

3. Textformatierung

In vielen Projekten werden Texte ausgegeben. Das ist einfach, solange sie nicht formatiert sein müssen. In der Realität benötigen wir aber oft zum Beispiel rechtsbündig angeordnete Zahlen, Texte mit fester Länge oder Zahlen mit Vornullen oder einer bestimmten Anzahl Stellen nach dem Koma. Wir erwarten aber auch, dass das Ganze einfach zu programmieren ist.

Python liefert uns hier viele komfortable Lösungen. Die Wichtigsten schauen wir uns jetzt an.

Einfache Textausgabe mit Platzhalter

```
print("Das ist ein String")
```

Das ist die einfachste Art der Textausgabe. Das muss ich wohl nicht mehr näher erläutern.

Wenn ich in einen Text einen anderen Text oder eine Zahl einbetten möchte, brauche ich einen Platzhalter. Dieser wird durch ein geschweiftes Klammernpaar {} gebildet.

Jeder String kann durch den Format - Befehl erweitert werden. Dabei wird im Text an allen Stellen, in denen etwas eingefügt werden soll, ein {} geschrieben. Im Format - Befehl werden dann einfach die Werte in der richtigen Reihenfolge angegeben.

```
rechnung = "3 * 5"
print("Das Resultat von {} ist {}".format(rechnung, 3*5))
```

Das Resultat von 3 * 5 ist 15

Was ist jetzt aber, wenn ich {} in meinem Text wirklich schreiben möchte? Hier zwei Möglichkeiten:

```
print("In der Sprache {} werden Blöcke in {{{}} eingeschlossen.".format("C"))
```

In der Sprache C werden Blöcke in {} eingeschlossen.

```
text = "In der Sprache {} werden Blöcke in {} eingeschlossen."
print(text.format("C", "{{}}"))
print(text.format("Pascal", "begin end"))
```

***In der Sprache C werden Blöcke in {} eingeschlossen.
In der Sprache Pascal werden Blöcke in begin end eingeschlossen.***

Die Feldlänge kann ebenfalls angegeben werden. Ausserdem können wir den Text links- oder rechtsbündig oder eingemittet ausgeben. Dazu verwenden wir : als Einleitung der Formatierung.

```
text = "Für {:10} brauchen wir viel Platz."
print(text.format("HALLO"))
```

```
text = "Für {:>10} brauchen wir viel Platz."
print(text.format("HALLO"))
```

```
text = "Für {:>10} brauchen wir viel Platz."
print(text.format("HALLO"))
```

```
text = "Für {:^10} brauchen wir viel Platz."
print(text.format("HALLO"))
```

***Für HALLO brauchen wir viel Platz.
Für HALLO brauchen wir viel Platz.
Für HALLO brauchen wir viel Platz.
Für HALLO brauchen wir viel Platz.***

Ganze Zahlen

Ich habe hier eine Liste von ganzen Zahlen und möchte diese mit einem nachfolgenden Text ausgeben.

```
zahlen = [2,235,15,8,-12]

ausgabe = "Die Zahl ist {}, was kommt danach?"
for zahl in zahlen:
    print(ausgabe.format(zahl))
```

```
Die Zahl ist 2, was kommt danach?
Die Zahl ist 235, was kommt danach?
Die Zahl ist 15, was kommt danach?
Die Zahl ist 8, was kommt danach?
Die Zahl ist -12, was kommt danach?
```

Oft wird aber gewünscht, dass der nachfolgende Text schön untereinander dargestellt wird. Dazu müssen wir für die Zahlenausgabe eine fixe Länge definieren.

Der Platzhalter wird dazu mit `:` erweitert, danach geben wir die **Länge** an und setzen den Datentyp mit dem Buchstaben **d** auf eine ganze Zahl.

```
ausgabe = "Die Zahl ist {:4d}, was kommt danach?"
for zahl in zahlen:
    print(ausgabe.format(zahl))
```

```
Die Zahl ist    2, was kommt danach?
Die Zahl ist   235, was kommt danach?
Die Zahl ist    15, was kommt danach?
Die Zahl ist     8, was kommt danach?
Die Zahl ist   -12, was kommt danach?
```

Zahlen werden automatisch rechtsbündig ausgegeben. Das kann aber ebenfalls gesteuert werden.

```
Linksbündig:  {:<4d}
Rechtsbündig: {:>4d}
Zentriert:    {:^4d}
```

Weitere Optionen:

```
Vorzeichen immer ausgeben:  {:+4d}
Vornullen und Vorzeichen:   {:+04d}
Vorzeichen untereinander:   {: =+4d}
Vorzeichen untereinander und Vornullen: { :0=+4d}
```


Zahlen mit Nachkommastellen

Zahlen mit Nachkommastellen sind sehr rechenaufwändig und daher in Micropython nur minimal unterstützt. Es gibt Einschränkungen gegenüber Standardpython bezüglich der Formatierungsmöglichkeiten und der Genauigkeit.

Auch hier arbeiten wir mit einer Liste von Zahlen.

```
zahlen = [2.5, 235.25, 15.3, 8.735, -12.37]

ausgabe = "Die Zahl ist {}, was kommt danach?"
for zahl in zahlen:
    print(ausgabe.format(zahl))
```

```
Die Zahl ist 2.5, was kommt danach?
Die Zahl ist 235.25, was kommt danach?
Die Zahl ist 15.3, was kommt danach?
Die Zahl ist 8.735, was kommt danach?
Die Zahl ist -12.37, was kommt danach?
```

Wir können die Anzahl Nachkommastellen und die Gesamtlänge festlegen.

```
ausgabe = "Die Zahl ist {:.4f}, was kommt danach?"
for zahl in zahlen:
    print(ausgabe.format(zahl))
```

```
Die Zahl ist 2.5000, was kommt danach?
Die Zahl ist 235.2500, was kommt danach?
Die Zahl ist 15.3000, was kommt danach?
Die Zahl ist 8.7350, was kommt danach?
Die Zahl ist -12.3700, was kommt danach?
```

Man sollte nie mehr Stellen angeben, als tatsächlich benötigt werden. Dies kann zu Ungenauigkeiten führen!

```
ausgabe = "Die Zahl ist {:12.8f}, was kommt danach?"
for zahl in zahlen:
    print(ausgabe.format(zahl))
```

```
Die Zahl ist 2.50000000, was kommt danach?
Die Zahl ist 235.24999619, was kommt danach?
Die Zahl ist 15.30000114, was kommt danach?
Die Zahl ist 8.73499966, was kommt danach?
Die Zahl ist -12.36999989, was kommt danach?
```

Die weiteren Optionen entsprechen den ganzen Zahlen:

```
Linksbündig:  {:<8.4f}
Rechtsbündig: {:>8.4f}
Zentriert:     {:^8.4f}
```

```
Vorzeichen immer ausgeben:  {:+8.4f}
Vornullen und Vorzeichen:   {:+08.4f}
Vorzeichen untereinander:   {: =+8.4f}
Vorzeichen untereinander und Vornullen: {:0=+8.4f}
```

Benannte Platzhalter

Die Werte müssen nicht unbedingt in der Reihenfolge eingegeben werden, in der sie im Text stehen. In diesem Fall müssen sie mit Namen versehen werden.

```
text = "Das Resultat von {Rechnung} ist {Resultat}"  
print(text.format(Resultat = 3*5, Rechnung="3 x 5"))
```

Das Resultat von 3 x 5 ist 15

Selbstverständlich sind auch hier Formatierungen mit **:** erlaubt.

```
text = "Das Resultat von {Rechnung} ist {Resultat:4.2f}"  
print(text.format(Resultat = 6 / 4, Rechnung="6 / 4"))
```

Das Resultat von 6 / 4 ist 1.50

4. Einfache Datentypen

Obwohl man in Python das meiste, das mit Datentypen zu tun hat, automatisch läuft, muss man manchmal doch selbst eingreifen. Es ist gut, wenn man die Datentypen kennt und diese ineinander umwandeln kann.

Das sind die wichtigsten Typen:

```
ganze_zahl = 10
dezimalzahl = 5.75
komplexe_zahl = 3.5 + 2.5j
text = "2.5"
ja_nein = True
```

Ganze Zahlen (integer)

Wenn keine Nachkommastellen vorhanden sind, wird ein Integer - Datentyp angelegt.

```
ganze_zahl = 10
```

Es lassen sich aber auch andere Typen in ganze Zahlen verwandeln. Dazu dient der **int()** - Befehl. Man nennt diesen Vorgang auch **type cast**.

```
ganze_zahl = int(5.75) ==> speichert 5
ganze_zahl = int("5") ==> speichert 5; wenn der String keiner ganzen
                             Zahl entspricht, gibt es eine Fehlermeldung
ganze_zahl = int(True) ==> speichert 1
```

Dezimalzahlen (float)

Falls Nachkommastellen vorhanden sind, wird eine Float - Variable angelegt.

```
dezimalzahl = 5.75
```

Auch hier besteht die Möglichkeit eines Type - Casts.

```
dezimalzahl = float(5) ==> speichert 5.0
dezimalzahl = float("3.5") ==> speichert 3.5; wenn der String keiner
                                Zahl entspricht, gibt es eine
                                Fehlermeldung
dezimalzahl = float(True) ==> speichert 1.0
```

Zeichen(chr)

Ein Teil der Werte zwischen 0 und 255 können auch als Zeichen ausgegeben werden. Diese Umwandlung funktioniert beidseitig.

```
zeichen = 'A'
code = 65

print(chr(code)) ==> 'A'
print(ord(zeichen)) ==> 65
```

Zeichenketten (String)

Texte werden in Form von Strings gespeichert.

```
text = "2.5"
```

Andere Datentypen lassen sich in Strings umwandeln. Dazu wird der **str()** - Befehl verwendet. Das genaue Format lässt sich aber nicht festlegen, daher sollte der Formatbefehl aus der letzten Lektion verwendet werden.

```
text = str(5)           ==> "5"
text = str(3.5)         ==> "3.5"
text = str(3.5 + 2.5j)   ==> "(3.5 + 2.5j)"
text = str(True)        ==> "True"
```

Boolscher Datentyp

Dieser Datentyp kann nur die Werte True oder False erhalten.

```
ja_nein = True
```

Nullwerte werden bei der Umwandlung als **False** betrachtet, alle anderen als **True**.

Komplexe Zahlen (complex)

Darauf möchte ich nicht im Detail eingehen, da dieser Zahlentyp nicht zur Basismathematik gehört.

```
komplexe_zahl = 3.5 + 2.5j
```

Diese Zahlen bestehen aus einem Real- (3.5) und einem Imaginärteil (2.5).

Man kann sie auch als Vektoren betrachten, dann ist 3.5 die X-Achse und 2.5 die y - Achse.

Oder als Schenkel eines rechtwinkligen Dreiecks. Darum gibt $\text{abs}(3 + 4j)$ den Wert 5, was einer Anwendung des Satzes von Pythagoras entspricht.

Komplexe Zahlen sind in der Wechselstromtechnik ganz nützlich.

Das kannst du auch schnell wieder vergessen. Im Zusammenhang mit dem ESP32 wirst du es kaum brauchen.

Hier trotzdem noch eine Umwandlung:

```
komplexe_zahl = complex(3.5) ==> es wird 3.5 + 0j gespeichert
```

5-8. Mengentypen (Collections)

Hier werden verschiedene Werte in einer Variablen abgelegt. Anders als bei den meisten anderen Programmiersprachen, können diese Werte verschiedene Typen aufweisen.

Es werden folgende Type besprochen:

Set, Tuple, List und Dictionary

Diese Typen haben viele Gemeinsamkeiten. Nicht jeder dieser Typen unterstützt aber alle Befehle.

Eigenschaften

Set	<ul style="list-style-type: none"> - Ein Set ist eine Sammlung von Werten, von denen jeder nur einmal vorkommt. - Im Set gibt es keine Reihenfolge. Es gibt keine Möglichkeit die Werte in einer bestimmten Reihenfolge abzurufen. - Bestehende Werte können zwar gelöscht, aber nicht verändert werden. - Es können neue Werte hinzugefügt werden.
Tuple	<ul style="list-style-type: none"> - Ein Tuple ist eine Sammlung von Elementen, die mehrfach vorkommen können. - Alle Elemente sind indexiert und haben somit eine bestimmte Reihenfolge. - Das Tuple ist konstant. Nach der Erzeugung können keine Werte geändert, hinzugefügt oder gelöscht werden.
List	<ul style="list-style-type: none"> - Eine List ist eine Menge von Werten verschiedener Datentypen. - Werte können mehrfach vorkommen. - Die Werte sind geordnet und indexiert. - Die List kann verändert werden.
Dictionary	<ul style="list-style-type: none"> - Menge von Werten verschiedener Datentypen - Werte können mehrfach vorkommen - Die Werte sind indexiert - Die Elemente sind nicht geordnet - Das Dictionary kann verändert werden Jeder Eintrag besteht aus einem Wertepaar, bestehend aus einem Schlüssel und einem Wert (key / value). Der Schlüssel wird als Index verwendet. - Der Schlüssel muss eindeutig sein und kann aus einem String, einer Zahl oder einem Tupel bestehen. - Als Wert kann jeder beliebige Datentyp eingesetzt werden.

Erzeugen

Direkte Zuweisung	<pre> set1 = {"Hallo",3,1.25,"Welt"} tuple1 = ("Hallo",3,1.25,"Welt",3,"Welt") list1 = ["Hallo",3,1.25,"Welt",3,"Welt"] dict1 = { "land": "Schweiz", "ort": "Bern", "einwohner": 140000 } </pre>
Leer oder aus anderen Mengen	<pre> set1 = set() set1 = set(range(1,5)) set1 = set(tuple1) set1 = set(list1) tuple1 = tuple() # nicht sinnvoll!!! tuple1 = tuple(set1) tuple1 = tuple(range(1,5)) tuple1 = tuple(list1) list1 = list() list2 = [] list1 = list(set1) list1 = list(tuple1) list1 = list(range(1,5)) dict1 = dict() dict2 = dict(dict1) dict1 = dict.fromkeys(("land","ort")) dict1 = dict.fromkeys(("land","ort"),"default") </pre>
Kopieren oder zusammenfügen	<pre> set2 = set1.copy() set3 = set1.union(set2) tuple2 = tuple1 tuple3 = tuple1 + tuple2 list2 = list1.copy() list2 = list(list1) list3 = list1 + list2 dict2 = dict1.copy() dict2.update(dict2) dict2.update({"kanton":"BE","ort":"Bern"}) </pre>

Grundlegende Abfragen

Existiert ein bestimmter Wert?	<pre>if "Hallo" in set1: if "Hallo" in tuple1: if "Hallo" in list1: if "ort" in dict1: # Frage nach Key</pre>
Alle Elemente auflisten	<pre>for element in set1: for element in tuple1: for element in list1: for x in dict1: # Schlüssel for x in dict1.keys(): # Schlüssel for x in dict1.values(): # Werte for k,v in dict1.items(): # Schlüssel und Wert</pre>
Anzahl Elemente	<pre>len(set1) len(tuple1) len(list1) len(dict1)</pre>

Auf Untermengen zugreifen

Wert eines Elementes auslesen	<pre>wert = tuple1[2] # 3. Element: Der unterste Index is 0 wert = list1[2] # 3. Element: Der unterste Index is 0 wert = dict1["ort"] # Schlüssel "ort" muss existieren wert = dict1.get("ort") # nicht vorhanden: None wert = dict1.get("ort","def") # nicht vorhanden: "def"</pre>
Wieviele Male ist der Wert vorhanden?	<pre>tuple1.count("Hallo") list1.count("Hallo")</pre>
Der niedrigsten Index eines Wertes abfragen	<pre>tuple1.index("Hallo") # Der Wert muss existieren list1.index("Hallo") # Der Wert muss existieren</pre>
Untermengen	<pre>tuple2 = tuple1[startindex:endindex] list2 = tuple1[startindex:endindex] startindex zeigt immer auf das erste Element endindex zeigt immer auf das Element nach dem letzten Element</pre> <p>Beispiel:</p> <pre>tuple1 = ("Hallo",1,"Welt",2,5,"Hallo",3,2) tuple1[2,5] erzeugt ("Welt",5) list1 = ["Hallo",1,"Welt",2,5,"Hallo",3,2] list1[2,5] erzeugt ["Welt",5]</pre> <p>Bei Listen wird keine eigene Liste erstellt, sondern nur ein Verweis auf den Unterbereich innerhalb der ursprünglichen Liste.</p> <p>Eine eigene Liste wird mit</p> <pre>list3 = list1[2,5].copy() list3 = list(list1[2,5]) erzeugt.</pre>
Zählung von hinten	<p>Als startindex und endindex können auch negative Werte angegeben werden. -1 bezeichnet das letzte Element.</p> <pre>tuple1 = ("Hallo",1,"Welt",2,5,"Hallo",3,2) tuple2 = tuple1[-5:-2] das erzeugt (2,5,"Hallo") list1 = ["Hallo",1,"Welt",2,5,"Hallo",3,2] list2 = list1[-5:-2] das erzeugt [2,5,"Hallo"]</pre>
Anfang und Ende	<p>startindex und endindex können auch weggelassen werden. Der Doppelpunkt bleibt aber auf jeden Fall erhalten. Durch einen fehlenden Startwert erhalten wir alle Werte ab Anfang, ein fehlender Endwert gibt alle Werte bis zum Ende aus.</p> <pre>tuple1 = ("Hallo",1,"Welt",2,5,"Hallo",3,2) tuple2 = tuple1[:-2] # ("Hallo",1,"Welt",2,5,"Hallo") tuple2 = tuple1[-5:] # (2,5,"Hallo",3,2) list1 = ["Hallo",1,"Welt",2,5,"Hallo",3,2] list2 = list1[:-2] # ["Hallo",1,"Welt",2,5,"Hallo"] list2 = list1[-5:] # [2,5,"Hallo",3,2]</pre>

Vergleichen

Welche Werte sind im ersten Set enthalten, nicht aber im zweiten Set?	<code>set1.difference(set2)</code>
Welche Werte sind nur in einem der beiden Sets enthalten?	<code>set1.symmetric_difference(set2)</code>
Welche Werte sind in beiden Sets enthalten?	<code>set1.intersection(set2)</code>
Ist Set1 eine Teilmenge von Set2?	<code>set1.issubset(set2)</code>
Enthält Set1 alle Werte von Set2?	<code>set1.issuperset(set2)</code>
Haben die beiden Sets keine gemeinsamen Elemente?	<code>set1.isdisjoint(set2)</code>

Bearbeiten

Die Variable entfernen	<code>del set1</code> <code>del tuple1</code> <code>del list1</code> <code>del dict1</code>
Alle Elemente der Menge löschen	<code>set1.clear()</code> <code>list1.clear()</code> <code>dict1.clear()</code>
Einzelne Werte löschen	<code>set1.remove("Hallo")</code> # Der Wert muss existieren <code>set1.discard("Hallo")</code> # Der Wert muss nicht existieren <code>list1.remove("Hallo")</code> # Der Wert muss existieren <code>list1.pop()</code> # Löscht das letzte Element <code>list1.pop(2)</code> # Löscht das Element mit Index 2 <code>list1.pop("ort")</code> # Schlüssel "ort" muss existieren <code>list1.popitem()</code> # letztes hinzugefügtes Element entfernen
Ein einzelnes Element hinzufügen	<code>set1.add("Hallo")</code> <code>list1.append("Hi")</code> # Fügt "Hi" am Ende hinzu <code>list1.insert(2,"neu")</code> # Fügt "neu" in Indexposition 2 ein <code>dict2["kanton"] = "BS"</code>
Eine andere Menge hinzufügen	<code>set1.update(set2)</code> <code>list1.append(list2)</code> <code>list1.extend(list2)</code> <code>dicts2.update(dict1)</code>
Resultat des Vergleichs von zwei Sets im ersten Set speichern	<code>set1.difference_update(set2)</code> <code>set1.symmetric_difference_update(set2)</code> <code>set1.intersection_update(set2)</code>
Sortierung	<code>list1.reverse()</code> # Reihenfolge wird umgekehrt <code>list1.sort()</code> # Nur bei gleichen Typen!

9. Kontrollstrukturen (Bedingungen und Schleifen)

Bedingungen: if .. elif .. else

Das Programm soll abhängig von Bedingungen Entscheidungen treffen können. Dazu wird eine Bedingung abgefragt und dann je nach Ergebnis eine bestimmte Operation ausgeführt.

```
x = 4
print("Die Zahl {}".format(x))
if x < 5:
    print(" ist kleiner als 5")
```

In diesem Fall ist die Bedingung erfüllt und der Text **ist kleiner als 5** ausgegeben.

Was ist aber, wenn die Bedingung nicht erfüllt ist? Dann soll auch das ausgegeben werden.

```
x = 6
print("Die Zahl {}".format(x))
if x < 5:
    print(" ist kleiner als 5")
else:
    print(" ist grösser als 5")
```

Dazu dient das Schlüsselwort **else**. Also immer wenn die Bedingung nicht erfüllt ist, werden die Aktionen des else - Blocks ausgegeben werden.

Was ist aber, wenn x exakt 5 ist? Dann erhalten wir eine Falschaussage (ist grösser als 5). Wir brauchen also einen zusätzlichen Test.

```
x = 5
print("Die Zahl {}".format(x))
if x < 5:
    print(" ist kleiner als 5")
elif x == 5:
    print(" ist exakt 5")
else:
    print(" ist grösser als 5")
```

elif (entspricht else if) leitet diesen Test ein. Er wird nur durchgeführt, wenn alle vorangegangenen Tests negativ waren. Es können beliebig viele elif - Tests eingefügt werden.

Einfache if - Abfragen können auch in einer Kurzschreibweise auf einer Zeile geschrieben werden.

```
if x > 5: print(" ist grösser als 5")
```

Die Bedingungen können auch mit logischen Operatoren verknüpft werden.

```
if x >= 5 and x <= 10 : print(" liegt zwischen 5 und 10")
```

while - Schleifen

Ein Programm ist in der Lage, dieselbe Arbeit unermüdlich beliebig viele Male zu erledigen. Dazu benötigen wir Schleifen. Sie führen immer dieselben Befehle aus, so lange eine bestimmte Bedingung erfüllt ist.

Die while - Schleife testet zuerst, ob die Bedingung noch erfüllt ist. Wenn ja, wird der Code ausgeführt, sonst wird mit dem Code nach der Schleife weiter gearbeitet.

```
x = 0
while x < 6:
    x += 1
    y = 2 * x
    print(x, y)
```

x wird bei jedem Durchgang hinaufgezählt. Sobald der Wert 6 erreicht, wird die Schleife beendet.

Es gibt noch weitere Möglichkeiten die Schleife abubrechen.

```
x = 0
while x < 6:
    x += 1
    y = 2 * x
    print(x, y)
    if (x == 3): break
```

Im Innern der Schleife kann ein zusätzlicher Test stattfinden und mit **break** ein sofortiges Ende der Schleife veranlasst werden.

Es ist auch möglich, die Abarbeitung der Befehle im Innern der Schleife abubrechen und sofort mit der nächsten Iteration weiterzumachen.

```
x = 0
while x < 6:
    x += 1
    if (x == 3): continue
    y = 2 * x
    print(x, y)
```

Wenn x den Wert 3 aufweist, wird y nicht mehr berechnet. Die Schleife fährt dann direkt mit dem nächsten x weiter. Dies wird durch **continue** veranlasst.

for - Schleifen

for - Schleifen werden immer dann eingesetzt, wenn eine bestimmte Menge an Werten abgearbeitet werden müssen. Diese Menge muss bereits vor dem Start der Schleife festgelegt sein.

Dazu eignen sich speziell Mengentypen wie Tuple, Set oder List. Oft wird die Menge auch mit range() generiert. Es können aber auch einzelne Buchstaben aus einem String abgearbeitet werden.

```
menge = (1,3,2,5,3)
for x in menge:
    print(x)
```

Auch hier kann mit **break** oder **continue** gearbeitet werden:

```
menge = [1,3,2,5,3]
for x in menge:
    print(x)
    if x == 3: break
```

Ein Spezialfall ist **range()**. Dieser Befehl erstellt eine Menge an aufeinanderfolgenden Werte.

```
menge = range(2,10)
for x in menge:
    print(x)
```

Dabei wird eine Liste mit Werten von 2 bis 9 erstellt. Die untere Angabe ist immer inklusive, die obere Angabe exklusive.

Es können auch Werte ausgelassen werden. Dazu kann die Schrittgrösse angegeben werden.

```
menge = range(2,10,2)
```

In **menge** sind dann die Werte 2,4,6,8 enthalten.

Die Zählrichtung kann auch umgekehrt werden. Dazu wird ein negativer Schritt angegeben.

```
menge = range(10,2,-2)
```

In **menge** sind dann die Werte 10,8,6,4 enthalten.

10. Funktionen

Einfache Funktion

```
def flaeche(a, b):
    return a * b    # Rechteck mit den Seiten a und b

a = 3
b = 5

resultat = flaeche(a,b) # Fläche des Rechtecks
print(resultat)
```

Die Funktion nimmt zwei Argumente entgegen und gibt einen Wert zurück. In **flaeche(a,b)** werden die Argumente als **Positionsargumente** (positional arguments) übergeben. Die Reihenfolge legt also fest, in welche Variable der Wert eingetragen wird.

Mit **return** wird das Resultat zurückgegeben. Auch in Python kann eine Funktion **nichts** zurückgeben. Wir verzichten dann einfach auf **return**.

Mehrere Werte zurückgeben

```
def flaeche(a, b):
    rechtecksflaeche = a * b    # Rechteck mit den Seiten a und b
    dreiecksflaeche = a * b / 2 # Dreieck mit Grundlinie a und Höhe b
    return rechtecksflaeche, dreiecksflaeche

a = 3
b = 5

resultat = flaeche(a,b) # Gibt Tupel zurück
print(resultat)
print(resultat[1])

rechteck, dreieck = flaeche(a, b) # Direkte Zuweisung
```

Eine Funktion kann auch mehrere Werte als **Tupel** zurückgeben. Diese Werte können dann direkt mehreren Variablen zugewiesen werden.

Beliebige Anzahl Argumente

Es kann auch mit Argumentslisten gearbeitet werden.

```
def laenge(a, b, *args):
    print()
    print(args)
    total = a+b
    for i in args:
        total += i
    return total

print(laenge(2,3))
print(laenge(2,3,4,10))
```

Im Tupel **args** befindet sich eine Liste der zusätzlichen Werte. Es handelt sich hier immer um Positionsargumente. Der Begriff **args** ist nicht zwingend. Massgebend ist der * vor dem Namen.

Standardwerte und benannte Argumente

Argumente können auch mit Namen übergeben werden. Man nennt das named arguments oder keyword arguments. Wenn man dem Argument einen Standardwert gibt, kann auf die Übergabe eines Wertes verzichtet werden.

```
import math

def flaeche(r=None, d=None):
    if r:
        return r**2 * math.pi
    elif d:
        return d**2 * math.pi / 4
    else:
        return None

print(flaeche(3))      # Positionsargument
print(flaeche(r=3))    # Benanntes Argument
print(flaeche(d=3))
```

Mehrere benannte Argumente

Auch benannte Argumente können als Liste entgegengenommen werden.

```
import math

def flaeche(**kwargs):
    print()
    print(kwargs)
    if "r" in kwargs:
        return kwargs["r"]**2 * math.pi
    elif "d" in kwargs:
        return kwargs["d"]**2 * math.pi / 4
    elif "g" in kwargs and "h" in kwargs:
        return kwargs["g"] * kwargs["h"] / 2
    elif "a" in kwargs:
        a = kwargs["a"]
        if "b" in kwargs:
            b = kwargs["b"]
        else:
            b = a
        return a * b
    else:
        return None

# print(flaeche(5)) # Kreis mit Radius r geht nicht!
print(flaeche(d=2)) # Kreis mit Durchmesser d
print(flaeche(r=2)) # Kreis mit Radius r

print(flaeche(g=5, h=2)) # Dreieck mit Grundlinie c und Höhe h
print(flaeche(a=5))      # Quadrat mit Seitenlänge a
print(flaeche(a=5, b=2)) # Rechteck mit Seiten a und b
```


Im Dictionary **kwargs** befindet sich eine Liste der zusätzlichen Werte. Es handelt sich hier immer um benannte Argumente. Der Begriff **kwargs** ist nicht zwingend. Massgebend ist ****** vor dem Namen.

11. Globale und lokale Variablen

Globale Variablen in Funktionen

In Python gibt es keine Deklaration von Variablen. Jede Zuweisung erzeugt eine neue Variable. Deshalb sind in Python einige Spezialitäten zu beachten.

So lange auf lokaler Ebene nur gelesen wird, entstehen keine Probleme. Das System verhält sich so, wie man es von anderen Programmiersprachen kennt.



```

a = "Die Zahl lautet"
b = 5

def ausgabe1():
    print("*** Ausgabe 1 ***")
    print(a,b)
    print()

print("Globale Werte: a = {}, b = {}".format(a,b))
ausgabe1()
print("Globale Werte: a = {}, b = {}".format(a,b))
    
```


a und b werden als globale Variablen erzeugt.

Die globalen Variablen a und b werden ausgegeben.

Hier werden immer die globalen Variablen a und b ausgegeben.

Der Hobbyelektroniker - Micropython Grundlagen
Globale und lokale Variablen
<https://micropython.org>

Sobald aber innerhalb auf lokaler Ebene (zum Beispiel innerhalb einer Funktion) eine Zuweisung erfolgt, wird die Variable neu angelegt und verliert die Verbindung zur ursprünglichen globalen Variablen. Die neu angelegte Variable ist jetzt lokal zur Funktion und wird wieder aufgelöst, sobald die Funktion verlassen wird.



```

a = "Die Zahl lautet"
b = 5

def ausgabe2():
    print("*** Ausgabe 2 ***")
    a = "Eine andere Zahl:"
    b = 7
    print(a,b)
    print()

ausgabe2()
print("Globale Werte: a = {}, b = {}".format(a,b))
    
```

a und b werden als globale Variablen erzeugt.


Es werden neue lokale Variablen a und b angelegt. Diese sind unabhängig von den globalen Variablen a und b.

Die neuen lokalen Variablen a und b werden ausgegeben.

Die globalen Variablen a und b wurden nicht verändert.

Der Hobbyelektroniker - Micropython Grundlagen
Globale und lokale Variablen
<https://micropython.org>

Es ist aber trotzdem möglich, eine globale Variable innerhalb einer Funktion zu ändern. Dazu muss vorher angegeben werden, dass die Variable global ist. Das geschieht mit dem Schlüsselwort **global**.



```

a = "Die Zahl lautet"
b = 5

def ausgabe2():
    print("*** Ausgabe 2 ***")
    global a,b
    a = "Eine andere Zahl:"
    b = 7
    print(a,b)
    print()

ausgabe2()
print("Globale Werte: a = {}, b = {}".format(a,b))
    
```


a und b werden als globale Variablen erzeugt.

Mit global wird gesagt, dass wir die globalen Variablen verwenden möchten. Diese werden dann verändert.

Die geänderten globalen Variablen a und b werden ausgegeben.

Die globalen Variablen a und b wurden verändert.

Der Hobbyelektroniker - Micropython Grundlagen
Globale und lokale Variablen




<https://micropython.org>

Mit dieser Angabe ist es sogar möglich, globale Variablen innerhalb einer Funktion zu erzeugen.

Erzeugen von Variablen

Python ist eine interpretierte Sprache. Daher erfolgt die Erzeugung von Variablen immer in der Reihenfolge der Abarbeitung. Im Gegensatz dazu werden bei compilierenden Sprachen die globalen Variablen bereits beim Compilieren erzeugt. Der Parser erwartet dabei, dass die Variable dem Compiler vor der ersten Benutzung bereits bekannt ist.



```

def ausgabe1():
    print("*** Ausgabe 1 ***")
    print(a,b,c)
    print()


a = "Die Zahl lautet"
b = 5
c = [1,3,5]

ausgabe1()
print("Globale Werte: a = {}, b = {}, c = {}".format(a,b,c))
    
```

Die globalen Variablen a, b und c können hier ausgegeben werden, obwohl sie erst weiter unten erzeugt werden.

Die globalen Variablen a, b und c werden hier erzeugt. Da dieser Programmteil vor dem Aufruf von ausgabe1() durchlaufen wird, stehen die Variablen in der Funktion dann zur Verfügung.

Der Hobbyelektroniker - Micropython Grundlagen
Globale und lokale Variablen



<https://micropython.org>

Globale Variablen als Argumente

Variablen können einer Funktion als Argumente übergeben werden. Dabei werden Referenzen übergeben. Falls einer dieser Referenzen ein neuer Wert zugewiesen wird, entsteht eine neue Variable. Der Zugriff auf das übergebene Argument geht dadurch verloren.

Das gilt nicht für Elemente innerhalb einer änderbaren Liste oder einer Klasseninstanz. Diese Elemente werden innerhalb der übergebenen Variablen geändert.

```
def ausgabe3(a,b,c):
    print("*** Ausgabe 3 ***")
    a = "Eine andere Zahl"
    b = 7
    c[1] = 2
    c = [1,7,4,6,5]
    c[2] = 12
    print(a,b,c)
    print()
```

c wurde als Argument übergeben und ist eine Referenz auf die globale Liste c. Darum wird diese verändert.

Eine neue lokale Liste c wird angelegt.

Hier wird die neue lokale Liste verändert.



12. Micropython, ESP32 und das Multitasking

Einfache Threads

Im Gegensatz zu Python 3.x kennt Micropython nur das Modul ***_thread***. Es ist in der offiziellen Dokumentation nicht im Detail enthalten und wird dort als ***highly experimental*** bezeichnet. Eine Dokumentation zu ***_thread*** ist aber auf der pycom - Webseite zu finden: https://docs.pycom.io/firmwareapi/micropython/_thread/ . Diese funktioniert auch mit unserem ESP32 - Board.

Ein Beispiel:

```
# Importe
import _thread
import time
from machine import Pin

# Pins initialisieren
rot = Pin(33, Pin.OUT)
gruen = Pin(25, Pin.OUT)
gelb = Pin(32, Pin.OUT)

# Funktionen, die als Thread ausgeführt werden sollen.
def blink_rot():
    for i in range(10):
        rot.on()
        time.sleep(1)
        rot.off()
        time.sleep(1)

def blink_gelb():
    for i in range(10):
        gelb.on()
        time.sleep(0.5)
        gelb.off()
        time.sleep(0.5)

def blink_gruen():
    for i in range(10):
        gruen.on()
        time.sleep(0.25)
        gruen.off()
        time.sleep(0.25)

# Threads werden mit start_new_thread() gestartet.
_thread.start_new_thread(blink_rot, ())
_thread.start_new_thread(blink_gelb, ())
_thread.start_new_thread(blink_gruen, ())

print("Ich bin noch nicht fertig! Ich beende nur das Hauptprogramm.")
```

Jetzt blinken alle LEDs gleichzeitig. Das Hauptprogramm wird zwar sofort beendet, die Threads laufen aber weiter, bis sie abgeschlossen sind.

Threads mit Parameter

Jede Funktion kann als Thread gestartet werden. Es ist auch möglich, Parameter zu übergeben. Die Argumente können beim Erzeugen des Threads mitgegeben werden.

```
_thread.start_new_thread(blink,(rot, 1))
```

Sie werden von der Funktion als normale Argumente entgegengenommen.

```
def blink(led, pause):
```

Ein Beispiel:

```
# Importe
import _thread
import time
from machine import Pin

# Pins initialisieren
rot = Pin(33, Pin.OUT)
gruen = Pin(25, Pin.OUT)
gelb = Pin(32, Pin.OUT)

# Funktionen, die als Thread ausgeführt werden sollen.
# Wir übergeben welche LED blinken soll und wie gross die Pause sein muss.
# Darum müssen wir nur noch eine Funktion schreiben.
def blink(led, pause):
    for i in range(10):
        led.on()
        time.sleep(pause)
        led.off()
        time.sleep(pause)

# Threads werden mit start_new_thread() gestartet.
_thread.start_new_thread(blink,(rot, 1))
_thread.start_new_thread(blink,(gelb, 0.5))
_thread.start_new_thread(blink,(gruen, 0.25))

print("Ich bin noch nicht fertig! Ich beende nur das Hauptprogramm.")
```

Bitte auch die Dokumentation zu **thread** beachten: [thread](#)

Wann ist ein Thread beendet?

Wir können mit **`_thread`** nicht feststellen, ob ein Thread bereits gestartet oder schon beendet ist. Also müssen wir das selbst verwalten. Glücklicherweise haben wir Zugriff auf unsere Thread-Funktion. Wenn die Funktion aufgerufen wird, ist der Thread gestartet. Wenn die Funktion beendet wird, wird auch der Thread aufgelöst. Damit wir diese Information nutzen können, geben wir dem Thread einen Namen.

Ein Beispiel:

```
# Importe
import _thread
import time
from machine import Pin

# Pins initialisieren
rot = Pin(33, Pin.OUT)
gruen = Pin(25, Pin.OUT)
gelb = Pin(32, Pin.OUT)

# Diese Funktion soll für Threads verwendet werden.
# Damit wir den Thread später besser identifizieren
# können, geben wir ihm einen Namen
def blink(name, led, pause):
    # Hier startet der Thread
    for i in range(10):
        led.on()
        time.sleep(pause)
        led.off()
        time.sleep(pause)
    # Hier wird der Thread beendet
```

Wenn der Thread aktiv wird, wird die Funktion gestartet. Wenn die Funktion abgearbeitet ist und verlassen wird, wird der Thread beendet und aus dem Speicher entfernt..

```
# Threads werden mit start_new_thread() gestartet.
_thread.start_new_thread(blink, ("ROT", rot, 1))
_thread.start_new_thread(blink, ("GELB", gelb, 0.5))
_thread.start_new_thread(blink, ("GRUEN", gruen, 0.25))

print("Hauptprogramm beendet! Threads noch nicht fertig.")
```

Bitte auch die Dokumentation zu **`_thread`** beachten: [_thread](#)

Eine Thread - Liste

Ausserhalb der Funktion haben wir keinen Zugriff auf den Thread. Aus diesem Grund müssen wir selbst eine Liste führen. Wir tragen den Namen des Threads beim Start in die Liste ein. Wenn der Thread beendet wird, löschen wir den Namen wieder aus der Liste.

Diese Liste können wir dann während der Abarbeitung laufend anzeigen.

Ein Beispiel:

```
# Importe
import _thread
import time
from machine import Pin

# Pins initialisieren
rot = Pin(33, Pin.OUT)
gruen = Pin(25, Pin.OUT)
gelb = Pin(32, Pin.OUT)

# Diese Liste enthält die Namen aller aktiven Threads
thread_list = set()

# Diese Funktion soll für Threads verwendet werden.
# Damit wir den Thread später besser identifizieren
# können, geben wir ihm einen Namen
def blink(name, led, pause):
    # Hier startet der Thread
    thread_list.add(name)
    for i in range(10):
        led.on()
        time.sleep(pause)
        led.off()
        time.sleep(pause)
    # Hier wird der Thread beendet
    thread_list.discard(name)
```

Wenn der Thread aktiv wird, wird die Funktion gestartet. Wenn die Funktion abgearbeitet ist und verlassen wird, wird der Thread beendet und aus dem Speicher entfernt.

```
# Threads werden mit start_new_thread() gestartet.
_thread.start_new_thread(blink, ("ROT", rot, 1))
_thread.start_new_thread(blink, ("GELB", gelb, 0.5))
_thread.start_new_thread(blink, ("GRUEN", gruen, 0.25))

print("Hauptprogramm beendet! Threads noch nicht fertig.")

# warten bis der erste Thread gestartet ist
while not thread_list: pass

# aktive Threads ausgeben, bis keiner mehr vorhanden ist
while thread_list:
    print(thread_list)
    time.sleep(0.05)

print("Jetzt bin ich wirklich fertig!")
```

Bitte auch die Dokumentation zu [_thread](#) beachten: [_thread](#)

Zugriffskonflikte vermeiden

Wenn mehrere Threads global angelegte Variablen oder Objekte verändern, kann das zu Konflikten führen. Aus diesem Grund stellt uns Micropython Semaphore zur Kontrolle der Zugriffe zur Verfügung. Diese werden wie globale Variablen angelegt.

```
lock = _thread.allocate_lock()
```

Nur wer im 'Besitz' dieses Semaphores ist, darf die Veränderung der Liste vornehmen. Alle Anderen müssen warten. Mit

```
lock.acquire()
```

kann ich das Semaphore anfordern. Der Aufruf stoppt, bis das Semaphor für mich frei ist. Wenn ich meine Arbeit abgeschlossen habe, kann ich das Semaphor für den Nächsten freigeben.

```
lock.release()
```

Ein Beispiel:

```
# Importe
import _thread
import time
from machine import Pin

# Pins initialisieren
rot = Pin(33, Pin.OUT)
gruen = Pin(25, Pin.OUT)
gelb = Pin(32, Pin.OUT)

# Diese Liste enthält die Namen aller aktiven Threads
thread_list = set()

# Diese Funktion soll für Threads verwendet werden.
# Damit wir den Thread später besser identifizieren
# können, geben wir ihm einen Namen
def blink(name, led, pause):
    # Hier startet der Thread
    lock.acquire()
    thread_list.add(name)
    lock.release()
    for i in range(10):
        led.on()
        time.sleep(pause)
        led.off()
        time.sleep(pause)
    # Hier wird der Thread beendet
    lock.acquire()
    thread_list.discard(name)
    lock.release()
```

Wenn der Thread aktiv wird, wird die Funktion gestartet. Wenn die Funktion abgearbeitet ist und verlassen wird, wird der Thread beendet und aus dem Speicher entfernt..

```
# Threads werden mit start_new_thread() gestartet.
_thread.start_new_thread(blink, ("ROT", rot, 1))
_thread.start_new_thread(blink, ("GELB", gelb, 0.5))
_thread.start_new_thread(blink, ("GRUEN", gruen, 0.25))

print("Hauptprogramm beendet! Threads noch nicht fertig.")

# warten bis der erste Thread gestartet ist
while not thread_list: pass
```

```
# aktive Threads ausgeben, bis keiner mehr vorhanden ist
while thread_list:
    print(thread_list)
    time.sleep(0.05)

print("Jetzt bin ich wirklich fertig!")
```

Bitte auch die Dokumentation zu **thread** beachten: [thread](#)