

# Inhaltsverzeichnis

<b>1: Einleitung, erste Schritte, Sprachstruktur, interaktive Konsole</b>	<b>2</b>
Einleitung	2
Erste Schritte	2
Die Sprachstruktur	3
REPL, die interaktive Konsole	3
<b>2. Kommentare, Blöcke und Operatoren</b>	<b>4</b>
Kommentare	4
Blöcke	4
Operatoren	5
<b>3. Textformatierung</b>	<b>6</b>
Einfache Textausgabe mit Platzhalter	6
Ganze Zahlen	7
Zahlen mit Nachkommastellen	8
Benannte Platzhalter	9
<b>4. Einfache Datentypen</b>	<b>10</b>
Ganze Zahlen (integer)	10
Dezimalzahlen (float)	10
Zeichen(chr)	10
Zeichenketten (String)	11
Boolscher Datentyp	11
Komplexe Zahlen (complex)	11

# 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.

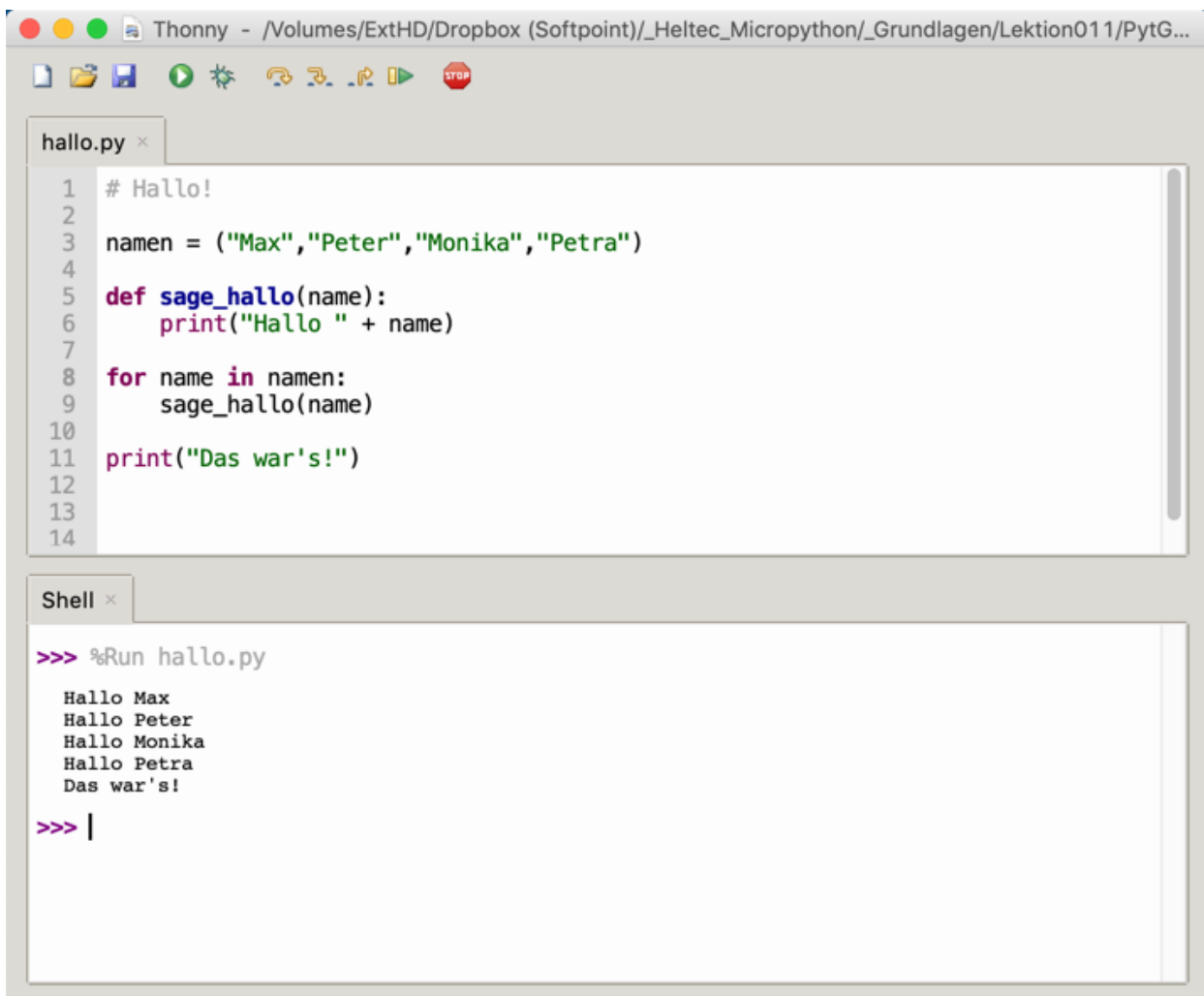
Ich nehme an, dass du die ersten Schritte mit dem ESP32 bereits unternommen hast. Damit sollten Thonny und das Heltec - Board bereits eingerichtet sein. Für diesen Kurs kannst du auch ein beliebiges anderes ESP32 - Board verwenden, falls du in der Lage bist, dieses einzurichten.

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 `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
```