

Lösungsblatt

Diese Fassung enthält die Lösungen.

Wie die Aufgaben aufgebaut sind

Standard ist das Ziel, fang immer dort an. **Einstieg** führt hin, falls du nicht reinkommst.

Experte geht weiter, wenn du fertig bist. Nicht jede Aufgabe hat alle drei Stufen.

Zum Nachschlagen liegen **Kärtchen** in der Tischmitte – nimm dir, was du brauchst, und leg es zurück, wenn du fertig bist.

Alles auf diesem Blatt läuft **auf dem Board**. Unten rechts in Thonny muss "MicroPython (ESP32)" stehen – nicht der Python-Interpreter deines Rechners.

Aufgabe 1 – Was in den Klammern steht

Einstieg

Jede Funktion bekommt etwas und gibt etwas zurück. Trag beides ein und prüf es danach in der Kommandozeile.

Aufruf	Parameter	Rückgabewert
<code>round(21.4736, 1)</code>		
<code>len("Schulgarten")</code>		
<code>max(12.4, 21.8, 18.0)</code>		
<code>abs(-3.2)</code>		

Lösung

Aufruf	Parameter	Rückgabewert
<code>round(21.4736, 1)</code>	21.4736 und 1	21.5
<code>len("Schulgarten")</code>	"Schulgarten"	11
<code>max(12.4, 21.8, 18.0)</code>	12.4, 21.8, 18.0	21.8
<code>abs(-3.2)</code>	-3.2	3.2

`round` bekommt zwei Parameter, die anderen einen – `max` verträgt beliebig viele. Wie viele es sind, hängt an der Funktion.

Tipp die vier Zeilen nacheinander in die Kommandozeile:

```
>>> ergebnis = round(21.4736, 1)
>>> ergebnis
>>> ergebnis = print("Hallo")
>>> ergebnis
```

Was steht am Ende in ergebnis? Schreib in einem Satz auf, worin sich round und print unterscheiden.

Lösung

Zuerst steht 21.5 in ergebnis, am Ende None.

round **gibt einen Wert zurück** – er lässt sich in einer Variable auffangen und weiterverwenden.
print **gibt etwas aus**: Auf dem Bildschirm steht Hallo, zurück kommt aber nichts. None ist genau das – kein Wert.

Häufige Fehlvorstellung: "print gibt mir die Zahl". Nein – es zeigt sie nur. Wer damit weiterrechnen will, braucht eine Funktion mit Rückgabewert.

Standard

Öffne mittelwert.py – die Woche von gestern, ausgerechnet in einer einzigen Zeile:

```
# Die Woche aus dem Schulgarten von gestern - jetzt in einer einzigen Zeile.

messwerte = [12.4, 15.1, 18.9, 21.8, 19.3, 14.7, 16.2]

print("Im Mittel:", round(sum(messwerte) / len(messwerte), 1), "Grad")
```

Drei Funktionen stecken ineinander. Gelesen wird von innen nach außen: Der Rückgabewert der inneren ist der Parameter der nächsten. Trag ein, was Schritt für Schritt zurückkommt.

	Aufruf	Rückgabewert
1	sum(messwerte)	
2	len(messwerte)	
3	sum(messwerte) / len(messwerte)	
4	round(..., 1)	

Lösung

	Aufruf	Rückgabewert
1	sum(messwerte)	118.4
2	len(messwerte)	7
3	118.4 / 7	16.914285714285715
4	round(16.914285714285715, 1)	16.9

Schritt 3 zeigt, wozu das round überhaupt da ist: Ohne es stünden fünfzehn Nachkommastellen auf dem Bildschirm.

Experte

Schreib eine zweite Zeile, die die **Spanne** ausgibt – den Abstand zwischen dem wärmsten und dem kältesten Wert, auf eine Nachkommastelle gerundet. Benutz dafür einen einzigen verschachtelten Aufruf.

Lösung

```
print("Spanne:", round(max(messwerte) - min(messwerte), 1), "Grad")
```

Ergibt 9.4. `max` und `min` geben je einen Wert zurück, dazwischen wird gerechnet, und `round` bekommt das Ergebnis als Parameter.

S031

Aufgabe 2 – Funktionen von anderen

Einstieg

Manche Funktionen sind einfach da. Probier die drei Zeilen in der Kommandozeile aus:

Aufruf	Läuft es?	Was kommt heraus
<code>len("Messung")</code>		
<code>round(3.7)</code>		
<code>sqrt(144)</code>		

Lösung

Aufruf	Läuft es?	Was kommt heraus
<code>len("Messung")</code>	ja	7
<code>round(3.7)</code>	ja	4
<code>sqrt(144)</code>	nein	ein <code>NameError</code>

`len` und `round` sind **built-in functions** und somit eingebaut und immer da. `sqrt` ist keine: Python kennt den Namen nicht, obwohl es die Funktion gibt.

Standard

a)

Es gibt sie doch – sie steckt nur in einer Bibliothek und muss geholt werden:

```
>>> import math
>>> math.sqrt(144)
```

Probier damit aus, was zurückkommt.

Aufruf	Rückgabewert
<code>math.sqrt(144)</code>	
<code>math.floor(3.7)</code>	
<code>math.ceil(3.2)</code>	

Lösung

Aufruf	Rückgabewert
--------	--------------

<code>math.sqrt(144)</code>	12.0
<code>math.floor(3.7)</code>	3
<code>math.ceil(3.2)</code>	4

`floor` rundet immer ab, `ceil` immer auf – anders als `round`, das zur nächstgelegenen Zahl geht. Drei Funktionen für drei verschiedene Fragen.

`math` gehört zur **Standardbibliothek**: Sie liegt schon auf dem Board, der `import` genügt. `time` von gestern ist dieselbe Sorte, ihr habt also längst eine Bibliothek benutzt.

Nach jedem Start ist die Kommandozeile leer

Wenn du ein Programm startest, fängt das Board von vorn an – und weiß nichts mehr von dem, was du vorher getippt hast. `math.sqrt(144)` endet dann im `NameError`, obwohl es eben noch lief.

Kein Fehler von dir: Der `import` gilt nur, bis das Board neu startet. Dann eben noch einmal.

b)

Standard

Die dritte Sorte importierst du gleich genauso. Sie liegt aber nicht auf dem Board. Du musst sie zuerst hinbringen:

1. Öffne loramint.de/workshop und lade `bme280.py` herunter.
2. Öffne die Datei in Thonny.
3. **Datei** → **Speichern unter...** → **MicroPython device**, Name unverändert `bme280.py`.

Jetzt liegt sie dort, wo das Board sie sucht:

```
>>> import bme280
```

Sieh in Thonny unter **Dateien** nach, was auf dem Board liegt. Tipp danach absichtlich falsch, etwa `import bme281`, und schreib auf, wie sich die beiden Fehlermeldungen unterscheiden: die von `sqrt(144)` oben und die von `import bme281`.

Lösung

`import bme280` läuft, `import bme281` endet mit einem `ImportError`.

`NameError` heißt: Diesen *Namen* kenne ich nicht – die Funktion ist nicht eingebaut, du hast die Bibliothek nicht geholt.

`ImportError` heißt: Diese *Bibliothek* gibt es hier nicht – die Datei liegt nicht auf dem Board.

`bme280` ist eine **externe Bibliothek**. In der Dateiliste steht sie als Datei; `math` und `time` stehen dort nicht, weil sie fest zur Firmware gehören.

Experte

Woher weißt du, was eine Bibliothek überhaupt kann? Frag sie:

```
>>> import math
>>> dir(math)
```

Probier dasselbe mit `bme280`: Ein Name darin sieht aus wie der Sensor – schreib ihn auf, du brauchst ihn in Aufgabe 4.

Lösung

`dir(math)` listet auf, was in `math` steckt – unter anderem `sqrt`, `floor` und `ceil`.

In `dir(bme280)` steht `BME280`. Groß geschrieben, weil es keine gewöhnliche Funktion ist, sondern der Bauplan für den Sensor. Benutzt wird er wie eine Funktion: `bme280.BME280(...)`.

`dir` ist die Antwort auf "Was kann das hier eigentlich?" – eine Frage, die man sonst nur googeln kann.

`dir(math)` allein ergibt einen `NameError`, wenn seit dem letzten Programmstart kein `import math` mehr getippt wurde. Deshalb steht er hier noch einmal darüber.

S032

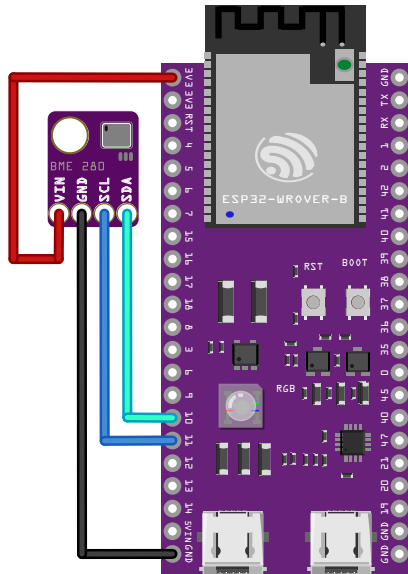
Aufgabe 3 – Den Sensor anschließen

Vier Drähte, mehr ist es nicht. Zwei für den Strom, zwei für die Daten. Die Beschriftung steht auf beiden Platinen – du musst nichts auswendig wissen, nur genau lesen.

Erst stecken, dann einschalten

Zieh das USB-Kabel ab, bevor du steckst, und steck es erst wieder ein, wenn alle vier Drähte sitzen. Nie umgekehrt.

Und: **niemals** 5V. Der Sensor arbeitet am ESP32 mit 3V3.



Der fertige Aufbau

Dasselbe noch einmal als Tabelle: Nimm die Variante, mit der du besser zurechtkommst.

	am Sensor	ans Board	Kabel
1	VIN	3V3	rot
2	GND	GND	schwarz
3	SDA	GPIO 10	türkis
4	SCL	GPIO 11	blau

Je nach Platine steht auf dem Sensor VIN, VCC oder 3V3 – gemeint ist immer derselbe Anschluss.

Die Probe. Steck das USB-Kabel wieder ein und starte `anschluss-pruefen.py`:

```
# Die Probe nach dem Stecken: Meldet sich der Sensor auf dem Bus?

from machine import Pin, I2C

i2c = I2C(0, sda=Pin(10), scl=Pin(11))

print(i2c.scan())
```

An den beiden Datenleitungen können mehrere Geräte gleichzeitig hängen, damit man sie auseinanderhalten kann, hat jedes eine eigene Nummer, seine **Adresse**. `i2c.scan()` einmal in die Leitung hinein und gibt als Liste zurück, wer sich gemeldet hat.

Was steht auf dem Bildschirm? Schreib es auf. Was bedeutet es, wenn dort nur `[]` steht.

Lösung

Bei richtigem Aufbau steht dort [118]. Die 118 ist die **Adresse** des Sensors auf dem Bus, in Hexadezimal 0x76.

`i2c.scan()` ruft gewissermaßen in die Leitung hinein und sammelt ein, wer sich meldet. Genau eine Zahl heißt: genau ein Gerät antwortet.

Eine leere Liste [] heißt: niemand antwortet. In dieser Reihenfolge prüfen:

1. Sitzt jeder Draht fest – an beiden Enden?
2. Sind SDA und SCL vertauscht? Der häufigste Fehler.
3. Hängt der rote Draht wirklich an 3V3 und nicht an 5V?
4. Stehen im Programm dieselben Nummern wie in der Tabelle, also `sda=Pin(10)` und `scl=Pin(11)`?

Findest du den Fehler nach zwei Minuten nicht, heb die Hand. Fehlersuche an Kabeln ist heute nicht das Lernziel, der Sensor soll laufen.

S035

S036

Aufgabe 4 – Der erste echte Messwert

Einstieg

Öffne `erste-messung.py`. Du musst nichts daran ändern, nur starten.

```
# Die erste echte Messung. Dieses Programm bekommt ihr fertig.

from machine import Pin, I2C
import bme280

# SDA an GPIO 10, SCL an GPIO 11 - so, wie ihr eben gesteckt habt.
i2c = I2C(0, sda=Pin(10), scl=Pin(11))

sensor_bme = bme280.BME280(i2c=i2c)

print(sensor_bme.values)
```

Leg danach die Hand um den Sensor, warte kurz und starte noch einmal. Trag beide Male ein, was auf dem Bildschirm steht.

Temperatur	Luftdruck	Luftfeuchte
ohne Hand		
mit Hand		

Welcher der drei Werte reagiert am schnellsten? Warum?

Lösung

Ausgegeben wird ein Dreierpack, etwa "21.53C", "1013.25hPa", "48.20%". Die genauen Zahlen hängen vom Raum ab – wichtig ist die Bewegung: Mit der Hand steigt die Temperatur um ein bis zwei Grad, die Feuchte deutlich stärker; der Luftdruck bleibt praktisch gleich.

Am schnellsten reagiert die **Luftfeuchte**. Der Luftdruck ändert sich im Zimmer überhaupt nicht – er hängt am Wetter, nicht an der Hand.

Der eigentliche Punkt dieser Aufgabe: Diese Zahl steht in keinem Programm. Sie kommt aus der Luft in diesem Raum.

Standard

Lässt sich mit den Werten bereits rechnen? Probier es aus:

```
>>> werte = sensor_bme.values
>>> werte[0]
>>> werte[0] + 1
```

Was für ein Datentyp steckt in `werte[0]`, und warum scheitert die Rechnung? Den Datentyp verrät dir die Funktion `type`:

```
>>> type(werte[0])
```

Lösung

`type(werte[0])` antwortet mit `<class 'str'>`: eine Zeichenkette, die zufällig wie eine Zahl aussieht – "21.530". Das `c` steht mit darin.

`werte[0] + 1` endet im `TypeError`: Zu einer Zeichenkette lässt sich keine Zahl addieren. `values` ist zum **Ansehen** gemacht, nicht zum Rechnen.

Dafür gibt es in der Bibliothek eine zweite Funktion. Sie gibt drei Zahlen in dieser Reihenfolge: Temperatur, Luftdruck, Luftfeuchte zurück.

messung.py

```
temperatur, druck, feuchte = sensor_bme.read_compensated_data()

print("Temperatur:", temperatur)
print("Luftdruck: ", druck)
print("Feuchte:  ", feuchte)
```

Schreib das Programm so um, dass es sauber liest: alle drei Werte auf eine Nachkommastelle gerundet, mit Einheit dahinter und der Luftdruck in **hPa** statt in Pa. Ein Pascal (Pa) ist der hundertste Teil eines Hektopascal (hPa).

Lösung

```
temperatur, druck, feuchte = sensor_bme.read_compensated_data()

print("Temperatur:", round(temperatur, 1), "Grad")
print("Luftdruck:", round(druck / 100, 1), "hPa")
print("Feuchte:", round(feuchte, 1), "Prozent")
```

`read_compensated_data` gibt **Zahlen** zurück – `round` nimmt sie an, `values` hätte hier einen `TypeError` ergeben.

Der Luftdruck kommt in **Pascal** aus dem Sensor: 101325.0. Erst geteilt durch 100 steht dort die Zahl, die man aus dem Wetterbericht kennt.

Ein Aufruf, drei Rückgabewerte, drei Variablen – die Reihenfolge entscheidet, welcher Wert wo landet. Wer `feuchte` und `druck` vertauscht, bekommt keine Fehlermeldung, sondern falsche Namen.

Eine Messung ist keine Messreihe. Bau eine geeignete Schleife von gestern darum, sodass das Programm alle zwei Sekunden neu misst und ausgibt.

Lösung

```
import time

while True:
    temperatur, druck, feuchte = sensor_bme.read_compensated_data()
    print("Temperatur:", round(temperatur, 1), "Grad")
    time.sleep(2)
```

`import` und `i2c` gehören **vor** die Schleife: Der Sensor wird einmal angelegt, nicht bei jeder Messung neu.

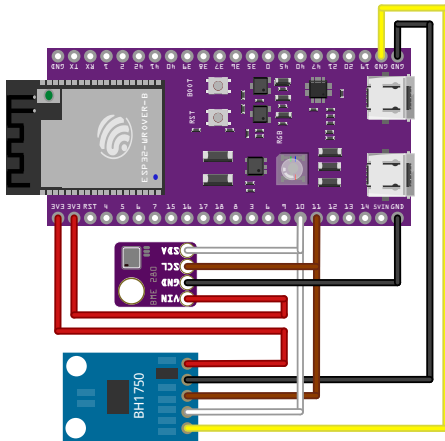
Häufiger Fehler: das Einrücken. Was wiederholt werden soll, steht eingerückt unter `while`; `time.sleep(2)` gehört dazu, sonst rennt die Ausgabe.

Experte

In der Projektbox liegt ein zweiter Sensor, der **BH1750**. Er misst Helligkeit in **Lux**.

a) Den zweiten Sensor dazustecken

Er kommt an dieselben zwei Datenleitungen wie der erste, der BME280 bleibt angeschlossen. Genau dafür haben die Geräte Adressen. Zieh vorher das USB-Kabel ab.



Beide Sensoren an denselben Leitungen

am BH1750	ans Board	Kabelfarbe
VCC	3V3	rot
GND	GND	schwarz
SDA	GPIO 10	weiß
SCL	GPIO 11	braun
ADDR	GND	gelb

SDA und SCL landen in denselben Reihen wie beim BME280. Die beiden Sensoren teilen sich die Leitungen.

Steck das USB-Kabel wieder ein und prüf nach, wer sich jetzt meldet:

```
>>> i2c.scan()
```

Lösung

Jetzt stehen dort **zwei** Zahlen: [35, 118], also 0x23 für den BH1750 und 0x76 für den BME280.

Das ist die Antwort auf die Frage aus Aufgabe 3, wozu Adressen gut sind. Zwei Geräte hängen an denselben zwei Drähten und kommen sich trotzdem nicht in die Quere, weil jedes auf seine eigene Nummer hört.

Der gelbe Draht von ADDR nach GND legt fest, welche der beiden möglichen Adressen der BH1750 benutzt. Ohne ihn wäre es die andere, und die Bibliothek würde ihn nicht finden.

b) Der neue Sensor allein

Hol dir `bh1750.py` von loramint.de/workshop und speicher sie aufs Board, genau wie in Aufgabe 2. Dann starte dieses Programm:

```
# Der zweite Sensor allein - dieses Programm bekommt ihr fertig.

from machine import Pin, I2C
import bh1750
import time

i2c = I2C(0, sda=Pin(10), scl=Pin(11))

sensor_lux = bh1750.BH1750(i2c)

while True:
    lux = sensor_lux.luminance(bh1750.BH1750.ONCE_HIRES_1)
    print("Helligkeit:", round(lux, 1), "Lux")
    time.sleep(2)
```

Halt die Hand darüber, dann wieder weg. Leuchte danach mit einer Taschenlampe darauf,

notfalls mit der deines Handys. Trag ein, was du bekommst.

Helligkeit in Lux

im Raum

Hand darüber

Taschenlampe

Lösung

Grobe Größenordnungen: ein paar Lux mit der Hand darüber, einige hundert bei Zimmerlicht, mehrere tausend, wenn eine Taschenlampe aus kurzem Abstand daraufhält. Der Wert reagiert sofort, viel deutlicher, als sich die Temperatur je bewegt hat.

Die genauen Zahlen hängen vom Raum und von der Lampe ab. Es geht um die Größenordnung, nicht um den Messwert.

Zwei Zeilen sind anders als beim BME280. `bh1750.BH1750(i2c)` bekommt den Bus *ohne* `i2c=`, einfach als ersten Parameter. `bme280.BME280(i2c=i2c)` will ihn benannt. Zwei Bibliotheken, zwei Schreibweisen; nachsehen muss man es jedes Mal.

Und `luminance` **braucht** einen Parameter: wie genau gemessen werden soll.

`sensor_lux.luminance()` allein endet in einem `TypeError`. Das ist der Parameterbegriff aus Aufgabe 1 von der anderen Seite. Manche Funktionen kommen ohne nicht aus.

c) Beide zusammen

Jetzt gehört beides in ein Programm: alle zwei Sekunden erst den einen, dann den anderen Sensor auslesen und **Temperatur und Helligkeit gemeinsam** ausgeben. Das Gerüst steht schon, fünf Stellen fehlen. Alles, was du dafür brauchst, steht in b) und in der Standardstufe.

beide.py

```
from machine import Pin, I2C
import bme280
import _____ # 1
import time

i2c = I2C(0, sda=Pin(10), scl=Pin(11))

sensor_bme = bme280.BME280(i2c=i2c)
sensor_lux = _____ # 2

while True:
    temperatur, druck, feuchte = sensor_bme.read_compensated_data()
    lux = _____ # 3

    print("Temperatur:", round(temperatur, 1), "Grad")
    print("Helligkeit:", _____, "Lux") # 4

    _____ # 5
```

Lücke	Was dort hingehört
-------	--------------------

1	
2	
3	
4	
5	

Ein i2c genügt

Die Leitung wird einmal angelegt und an *beide* Sensoren weitergegeben. Zwei `import`-Zeilen, zwei Sensor-Variablen, aber nur ein `I2C(...)`. In Lücke 2 gehört also kein zweites.

Lösung

Lücke	Was dort hingehört
-------	--------------------

1	<code>bh1750</code>
2	<code>bh1750.BH1750(i2c)</code>
3	<code>sensor_lux.luminance(bh1750.BH1750.ONCE_HIRES_1)</code>
4	<code>round(lux, 1)</code>
5	<code>time.sleep(2)</code>

Zu 2: ohne `i2c=` und ohne ein zweites `I2C(...)`. Es ist dieselbe Leitung, sie wird nur weitergereicht.

Zu 3: Der Modus ist Pflicht. `sensor_lux.luminance()` endet im `TypeError`.

Zu 4: Ohne `round` stehen dort fünfzehn Nachkommastellen, dasselbe Argument wie in Aufgabe 1.

Zu 5: `time.sleep(2)` steht *in* der Schleife und gleich weit eingerückt wie die `print`-Zeilen. Eine Stufe weiter, und es gehört zu gar nichts mehr; eine Stufe weniger, und die Schleife ist zu Ende.

Die Reihenfolge der beiden Messungen ist egal, sie stören sich nicht. Damit hast du zwei Sensoren, zwei fremde Bibliotheken und eine Schleife in einem Programm, alles, was das Projekt braucht.

S033

Aufgabe 5 – Messwerte mit Namen

Einstieg

Eine Messung von gestern, in einer Liste:

```
>>> messung = [21.5, 48.2, 1013.0]
>>> messung[1]
```

Was kommt zurück? Und was bedeutet die Zahl? Frag die Person neben dir, ob sie es ohne Nachdenken sagen kann.

Lösung

Zurück kommt 48.2. Es ist die Luftfeuchte, aber das steht nirgends. Man muss die Reihenfolge kennen, in der die Liste geschrieben wurde.

Eine Liste **ordnet** Werte. Sie **benennt** sie nicht.

Standard

Dafür gibt es das **Dictionary**. Statt einer Nummer bekommt jeder Wert einen Namen, den **Schlüssel**. Geschrieben wird das mit geschweiften Klammern, darin Paare aus Schlüssel und Wert:

```
>>> messung = {"temperatur": 21.5, "feuchte": 48.2, "druck": 1013.0}
>>> messung["feuchte"]
```

Gelesen wird wie bei der Liste mit eckigen Klammern, nur steht darin jetzt der Name statt der Nummer. Leg das Dictionary an und probier dann diese vier Eingaben nacheinander aus.

Eingabe	Was kommt zurück
---------	------------------

<code>messung["druck"]</code>	
-------------------------------	--

<code>messung["temperatur"]</code>	<code>= 22.1</code>
------------------------------------	---------------------

<code>messung</code>	
----------------------	--

<code>messung["helligkeit"]</code>	
------------------------------------	--

Lösung

Eingabe	Was kommt zurück
<code>messung["druck"]</code>	1013.0
<code>messung["temperatur"] = 22.1</code>	nichts, der Wert wird ersetzt
<code>messung</code>	das ganze Dictionary, jetzt mit 22.1
<code>messung["helligkeit"]</code>	ein <code>KeyError</code>

Lesen und Ändern sehen gleich aus. Steht der Ausdruck rechts vom Gleichheitszeichen, wird gelesen; steht er links, wird geschrieben. Genau wie bei den Variablen von Tag 1.

`KeyError` heißt: Diesen Schlüssel gibt es nicht. Im Fehlertext steht der Name, den du gesucht hast, und damit meistens auch der Tippfehler.

Zweiter Stolperstein: `messung[temperatur]` ohne Anführungszeichen sucht nach einer *Variablen* dieses Namens und endet im `NameError`. Ein Schlüssel ist Text und gehört deshalb in Anführungszeichen.

Jetzt mit echten Werten. Nimm dein Programm aus Aufgabe 4 und schreib die drei Messwerte in ein Dictionary, bevor du sie ausgibst.

Lösung

```
temperatur, druck, feuchte = sensor_bme.read_compensated_data()

messung = {"temperatur": round(temperatur, 1),
           "druck": round(druck / 100, 1),
           "feuchte": round(feuchte, 1)}

print(messung)
```

Gerundet wird schon beim Anlegen, nicht erst bei der Ausgabe. Dann steht im Dictionary genau der Wert, den man später weitergeben will.

`print(messung)` gibt alle drei Werte auf einmal aus, samt Namen.

Experte

Die Schleife von gestern läuft auch über ein Dictionary. Sie bekommt dabei nicht die Werte, sondern die Schlüssel. Gib damit jeden Messwert beschriftet aus, ohne die Namen selbst zu tippen.

Lösung

```
for name in messung:
    print(name, messung[name])
```

Ausgegeben wird

```
temperatur 21.5
druck 1013.0
feuchte 48.2
```

In `name` steht nacheinander jeder Schlüssel. Den passenden Wert holt `messung[name]`. In den eckigen Klammern steht diesmal eine Variable, deshalb keine Anführungszeichen.

Der Vorteil: Kommt ein vierter Messwert dazu, gibt die Schleife ihn ohne Änderung mit aus.

Wozu das Ganze: An Tag 4 wird jeder Messwert einzeln verschickt, und zu jedem muss dabeistehen, welcher er ist. Das Dictionary hält Name und Wert zusammen, und diese Schleife läuft dann über beides. Bei einer Liste müsstest du die Namen daneben im Kopf behalten.

Anhang • Funktionen, Bibliotheken und Dictionaries

Alles von den Kärtchen noch einmal, zum Mitnehmen und Nachschlagen. Auswendig lernen musst du davon nichts.

Funktionen aufrufen

```
gerundet = round(21.4736, 1)
```

- **Name** — round sagt, welche Funktion gemeint ist.
- **Parameter** — was in den Klammern steht, bekommt sie zum Arbeiten. Wie viele es sind, hängt an der Funktion.
- **Rückgabewert** — was herauskommt. Er lässt sich in einer Variablen auffangen und weiterverwenden.

Ausgeben ist nicht Zurückgeben. print zeigt etwas an und gibt nichts zurück; ergebnis = print("Hallo") setzt ergebnis auf None — auf keinen Wert.

Verschachtelt liest man von innen nach außen: Bei round(sum(werte), 1) läuft erst sum, sein Rückgabewert wird zum Parameter von round.

Immer da, ohne import: print, round, len, sum, max, min, abs — die **built-in functions**.

Bibliotheken

```
import math  
  
print(math.sqrt(144))
```

Der import steht **ganz oben** im Programm. Vor dem Punkt steht, *woher* die Funktion kommt, nach dem Punkt, *welche* es ist.

Sorte	Beispiel	Woher
eingebaut	print, len	immer da, kein import
mitgeliefert	time, math, machine	Standardbibliothek, liegt auf dem Board
dazugelegt	bme280	externe Bibliothek, muss als Datei auf dem Board liegen

Was in einer Bibliothek steckt, verrät dir(math).

Nach jedem Programmstart beginnt das Board von vorn. Der import in der Kommandozeile gilt dann nicht mehr — einfach noch einmal tippen.

Den Sensor auslesen

Diese Zeilen sind immer gleich:

```
from machine import Pin, I2C  
import bme280  
  
i2c = I2C(0, sda=Pin(10), scl=Pin(11))  
sensor_bme = bme280.BME280(i2c=i2c)
```

Danach gibt es zwei Wege an die Messwerte:

- sensor_bme.values liefert **Text mit Einheit**, etwa "21.53C" — zum Ansehen. Rechnen geht damit nicht.

- `sensor_bme.read_compensated_data()`
liefert **drei Zahlen**, in der Reihenfolge Temperatur, Druck, Feuchte – zum Weiterrechnen.

```
temperatur, druck, feuchte = sensor_bme.read_compensated_data()
print("Temperatur:", round(temperatur, 1), "Grad")
print("Luftdruck:", round(druck / 100, 1), "hPa")
```

Der Druck kommt in **Pascal**. Geteilt durch 100 ergibt er die hPa aus dem Wetterbericht.

Dictionaries

```
messung = {"temperatur": 21.5, "feuchte": 48.2, "druck": 1013.0}
```

Geschweifte Klammern, darin Paare aus **Schlüssel** und **Wert**, getrennt durch einen Doppelpunkt. Der Schlüssel ist Text und steht in Anführungszeichen.

Was du willst	Wie es geht
einen Wert lesen	<code>messung["feuchte"]</code>
einen Wert ändern	<code>messung["feuchte"] = 51.0</code>
einen Wert hinzufügen	<code>messung["ort"] = "Raum 101"</code>
alles ausgeben	<code>print(messung)</code>
alles durchlaufen	<code>for name in messung:</code>

```
for name in messung:
    print(name, messung[name])
```

Die Schleife bekommt die **Schlüssel**, nicht die Werte. In den eckigen Klammern steht dann die Variable `name` – ohne Anführungszeichen, denn es ist keine Zeichenkette, sondern ein Name.

Wenn etwas schiefgeht

Meldung	Was sie bedeutet
<code>NameError</code>	Diesen <i>Namen</i> kenne ich nicht. Meist fehlt der <code>import</code> , oder das Board wurde seither neu gestartet. Auch: Anführungszeichen beim Schlüssel vergessen.
<code>ImportError</code>	Diese <i>Bibliothek</i> gibt es hier nicht. Die Datei liegt nicht auf dem Board – unter Dateien nachsehen.
<code>KeyError</code>	Diesen <i>Schlüssel</i> gibt es im Dictionary nicht. Der Name im Fehlertext ist der gesuchte.
<code>TypeError</code>	Die Sorten passen nicht zusammen. Meist: mit <code>sensor_bme.values</code> gerechnet – das ist Text, keine Zahl.
<code>OSError / []</code> bei <code>i2c.scan()</code>	Der Sensor meldet sich nicht. Drähte prüfen, SDA und SCL nicht vertauschen, 3V3 statt 5V.

Zwei Minuten suchen, dann Hand heben. Fehlersuche an Kabeln ist heute nicht das Lernziel.