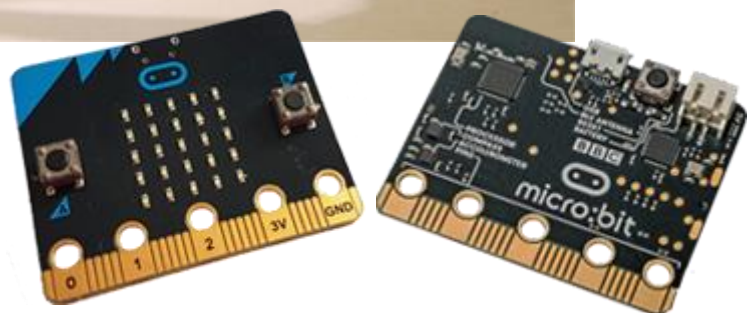




## Teil 2: ROBOTIK

Jarka Arnold  
Aegidius Plüss



## INHALT

### ROBOTIK MIT MICRO:BIT

Was ist ein micro:bit? .....	3
Loslegen .....	4
LED-Display .....	7
Beschleunigungssensor .....	10
Buttons .....	13
Sound .....	16
Magnetfeldsensor, Kompass .....	19
Bluetooth-Kommunikation .....	23
Fahrende Roboter .....	26
Alarmanlagen .....	32
Datenerfassung .....	34
7-Segment Digitalanzeige .....	37

### ARBEITSBLÄTTER:

"Fang das Ei" mit micro:bit .....	41
MOVEmini .....	43

### ANHANG:

Dokumentation micro:bit .....	45
Über die Autoren .....	52
Links .....	53

**Dieses Werk ist urheberrechtlich nicht geschützt und darf für den persönlichen Gebrauch und den Einsatz im Unterricht beliebig vervielfältigt werden. Texte und Programme dürfen ohne Hinweis auf ihren Ursprung für nicht kommerzielle Zwecke weiter verwendet werden.**



To the extent possible under law, TJGroup has waived all copyright and related or neighboring rights to TigerJython4Kids.

Version 1.0, Oktober 2017

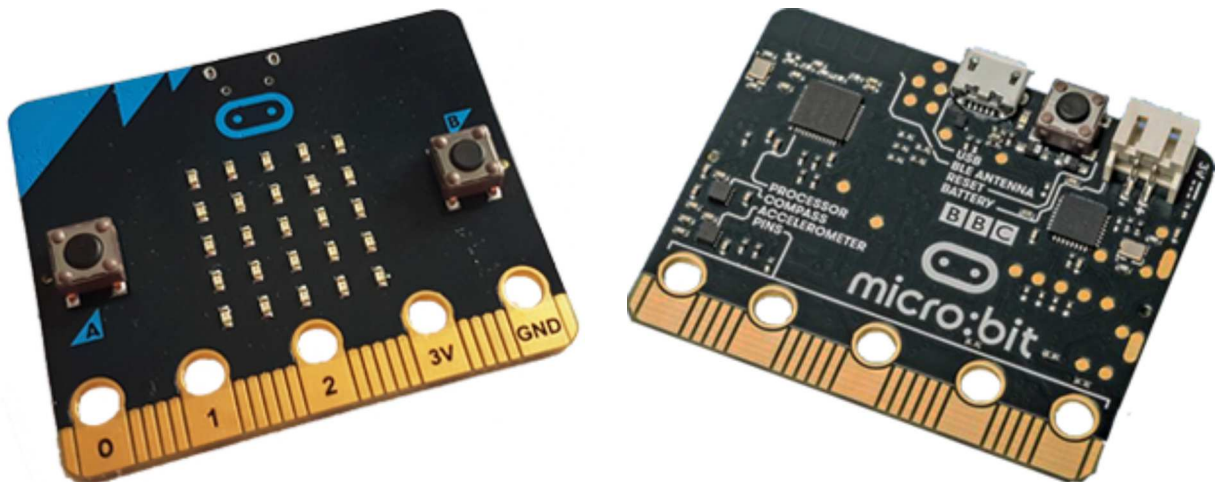
Autoren: Jarka Arnold, Aegidius Plüss  
Kontakt: [help@tigerjython.com](mailto:help@tigerjython.com)

# MICRO:BIT

---

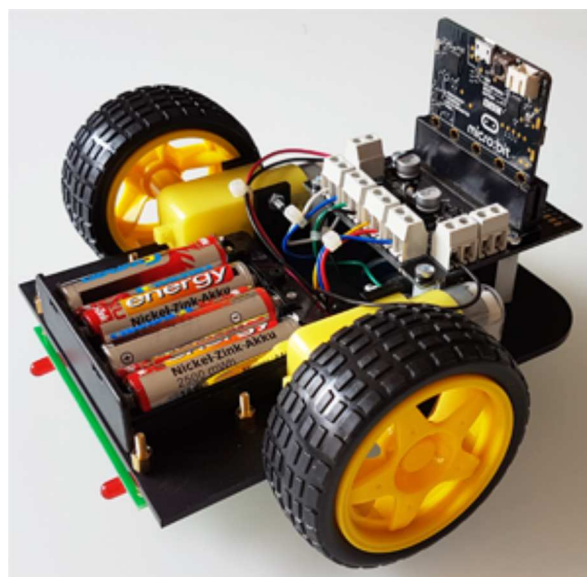
## ■ WAS IST EIN MICRO:BIT ?

Der BBC micro:bit ist ein programmierbarer Computer im Kreditkartenformat. Er besteht aus einer 4 x 5 cm grossen Platine mit einem 32-Bit Microcontroller, Flash Speicher, 25 roten Leds, mit welchen du einfachen Bilder und Nachrichten anzeigen kannst, zwei Buttons und einer USB-Schnittstelle.



Der micro:bit ist sogar in der Lage, mit dem Beschleunigungssensor seine Neigung und Bewegungen zu erkennen, verfügt über einen eingebauten Kompass, einen Temperaturfühler und kann über Bluetooth mit anderen micro:bits kommunizieren. Weitere Aktoren und Sensoren. kannst du mit Krokodilklemmen über gut zugängliche Pins anschliessen. Mit ein wenig Bastelarbeit kannst du den micro:bit sogar zu einem fahrenden Roboter ausbauen.

Die Programmentwicklung mit Python ist einfach und wird dir sicher Spass machen. Wir gehen davon aus, dass du die grundlegenden Programmstrukturen bereits kennst und dass du mit der TigerJython-Entwicklungsumgebung umgehen kannst.



# 1. LOSLEGEN

---

## ■ DU LERNST HIER...

wie du mit Python ein Programm entwickeln und auf dem micro:bit ausführen kannst.

## ■ PROGRAMMENTWICKLUNG

Ein Programm, das auf dem micro:bit laufen soll, kannst du mit TigerJython auf irgendeinem Computer entwickeln. Nachdem du das Programm im Editor geschrieben hast, lädst du es über ein USB-Kabel auf den micro:bit hinunter und es wird dort mit **MicroPython** (einer reduzierten Python-Version) ausgeführt. Dazu muss auf dem micro:bit eine **Firmware** installiert sein.

## ■ USB-VERBINDUNG

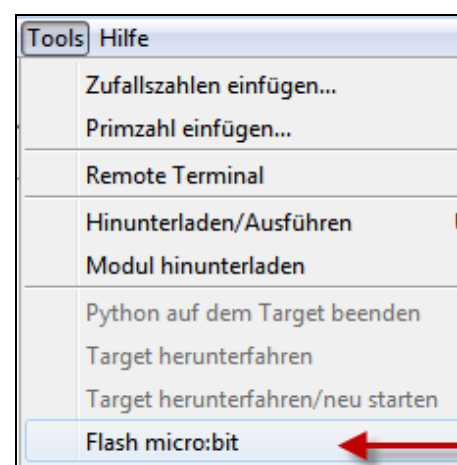
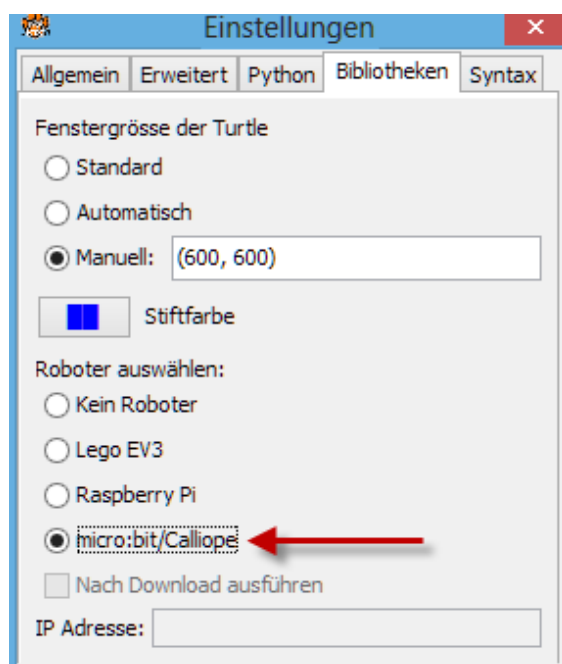
Schliesse den micro:bit über ein USB-Kabel am Computer an. Du siehst ein zusätzliches externes USB-Gerät (unter Windows mit einem Laufwerk-Buchstaben).

Für Windows früherer Versionen als Windows 10 musst du für die serielle Kommunikation einen Treiber installieren. Die Installationsdatei kannst du hier herunterladen: [mbed.zip](http://mbed.zip). Packe die Datei aus und führe sie per Mausklick aus. Dazu brauchst du allerdings Administrator-Rechte.



## ■ FIRMWARE INSTALLIEREN

Vor der ersten Verwendung musst du die Firmware auf den micro:bit hinunterladen. Mit TigerJython ist dies sehr einfach: Wähle unter *Einstellungen/Bibliotheken* die Option *micro:bit/Calliope*. Diese Einstellung bleibt gespeichert und ist auch für das Hinunterladen der Programme erforderlich. Wähle im Menü unter *Tools* die Option **Flash micro:bit**.



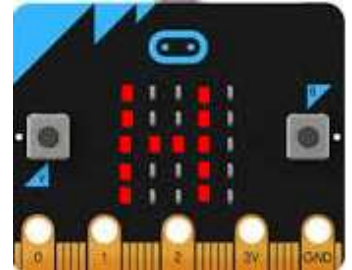
Die gelbe LED neben der USB-Buchse beginnt zu blinken. Sobald der Download fertig ist, erscheint im Ausgabefenster die Meldung: "Successfully transferred file to micro:bit".

## ■ MUSTERBEISPIEL

Zum Einstieg schreibst du nur einen kurzen Text auf dem Display aus. Tippe im Editor das unten stehende Programm ein oder klicke auf *Programmcode markieren* und füge es mit Ctrl+C und Ctrl+V ein.

```
from microbit import *  
  
display.scroll("HELLO PYTHON!")
```

**Programmcode markieren** (Ctrl+C kopieren, Ctrl+V einfügen)



Kontrolliere, ob der micro:bit am Computer angeschlossen ist und klicke auf die Schaltfläche *Hinunterladen/Ausführen*.

Neben dem TigerJython-Fenster erscheint ein zweites Terminal-Fenster. Hier werden Mitteilungen und Fehlermeldungen angezeigt.

Wenn HALLO PYTHON als Scrolltext angezeigt wird, funktioniert dein micro:bit einwandfrei.

Der Programmcode ist leicht zu verstehen: In der ersten Programmzeile wird das Modul *microbit* importiert und die Objekte und Funktionen aus dieser Bibliothek bereit gestellt, beispielsweise das Objekt *display*, mit dem du auf die LEDs zugreifst. Mit dem Befehl *display.scroll("HELLO PYTHON!")* kannst du kurze Mitteilungen als Lauftext anzeigen. Beachte, dass der Text in einfachen oder doppelten Anführungszeichen stehen muss!

## ■ MERKE DIR...

Du schreibst ein Programm für den micro im TigerJython-Editor. Um das Programm auf dem micro:bit auszuführen, klickst du nicht auf die grüne Schaltfläche, sondern auf die Schaltfläche *Hinunterladen/Ausführen*.

Auf dem micro:bit ist jeweils das zuletzt heruntergeladene Programm gespeichert. Die Programmausführung startet automatisch beim nächsten Anschluss einer Stromversorgung. Sollte einmal das gespeicherte Programm wegen eines Programmierfehlers den micro:bit blockieren, so musst du den micro:bit neu flashen.

## ■ ZUM SELBST LÖSEN

1. Schreibe verschiedene Meldungen als Scrolltext aus (auch kleine Buchstaben sind erlaubt).
2. Schreibe eine Mitteilung aus, die sich endlos wiederholt. Verwende dazu in *scroll()* den zusätzlichen Parameter *loop = True*.

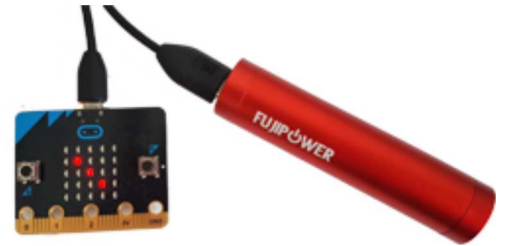
Das Programm läuft endlos. Du kannst es allerdings im Terminalfenster mit Ctrl+C abbrechen (und mit Ctrl+D wieder starten).

## ■ ZUSATZBEMERKUNGEN

Nachdem das Programm zu Ende gelaufen ist, wird der Kommandozeilen-Prompt des Python-Interpreters `>>>` ausgeschrieben und du kannst hier auch Python-Befehle interaktiv eintippen und ausführen. Mit der Tastenkombination `Ctrl+D` kannst du ein heruntergeladenes Programm erneut starten.

Falls du einmal ein länger laufendes Programm abbrechen willst, so kannst du im Terminal `Ctrl+C` eintippen, es wird aber auch abgebrochen, wenn du ein neues Programm hinunterlädst.

Dein Programm bleibt so lange auf dem micro:bit gespeichert, bis du es mit einem neuen Programm überschreibst (oder der micro:bit geflasht wird). Du kannst also den micro:bit beim Computer ausstecken und an eine andere Spannungsquelle anschliessen, beispielsweise an ein USB-Ladegerät oder eine PowerBank. Sofort startet das zuletzt gespeicherte Programm wieder.



Um ein Programm mehrmals auszuführen, kannst du auch den Reset-Button klicken, der sich neben der USB-Buchse befindet, statt die Spannungsversorgung zu unterbrechen.



## 2. LED-DISPLAY

### ■ DU LERNST HIER...

wie man mit dem LED-Display Textmeldungen und einfache Bilder anzeigen kann.

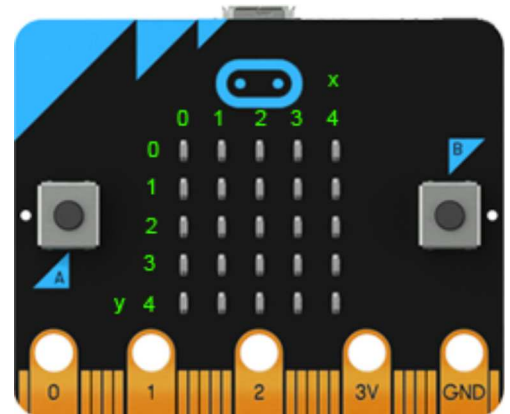
### ■ 25 LEDs

Die 25 LEDs des Displays sind in einer 5x5 Matrix angeordnet und über ihre x- und y-Position einzeln ansprechbar. Mit

***display.set\_pixel(x, y, n)***

leuchtet die LED an der Position x, y mit der Helligkeit n, wobei n eine Zahl von 0 bis 9 ist. Für n = 9 ist die LED am hellsten, für n = 0 ist sie ausgeschaltet.

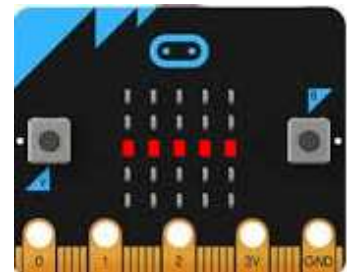
Du kannst alle LEDs mit ***display.clear()*** ausschalten.



### ■ MUSTERBEISPIELE

#### LEDs ein- und ausschalten

Dein Programm soll die LEDs in der mittleren Reihe der Reihe nach ein- und ausschalten. Der Befehl *sleep(400)* hält das Programm 400 Millisekunden an. Dadurch bleibt die gerade leuchtende LED während dieser Zeit eingeschaltet.



```
from microbit import *

for x in range(5):
    display.set_pixel(x, 2, 9)
    sleep(400)
    display.set_pixel(x, 2, 0)
```

Damit du mit Zufallszahlen programmieren lernst, soll in diesem Beispiel wiederholt eine zufällig ausgewählte LEDs ein- und ausgeschaltet werden. In der Funktion *randomLed()* werden zuerst alle LEDs gelöscht, und nachher zwei Zahlen zwischen 0 und 4 gewählt und die LED mit diesen Koordinaten mit maximaler Helligkeit eingeschaltet.



Im Hauptprogramm wird die Funktion *randomLed()* in einer Endlosschleife alle 200 Millisekunden aufgerufen. Das Programm läuft also so lange, bis du ein neues Programm hinunterlädst, den Resetknopf (Button neben dem USB-Anschluss) drückst oder die Stromversorgung unterbrichst.

```

from microbit import *
from random import randint

def randomLed():
    display.clear()
    x = randint(0, 4)
    y = randint(0, 4)
    display.set_pixel(x, y, 9)

while True:
    randomLed()
    sleep(200)

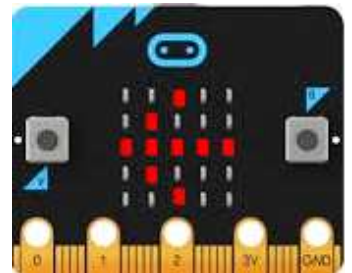
```

[Programmcode markieren](#) (Ctrl+C kopieren, Ctrl+V einfügen)

## Images anzeigen

Im Modul microbit sind mehrere 5x5 Pixelbilder gespeichert, die du auf dem Display anzeigen kannst. Die Bildnamen siehst du hier oder in der Dokumentation.

Mit dem Befehl **display.show()** kannst du diese Bilder anzeigen. In diesem Beispiel werden nacheinander Pfeile angezeigt, die nach Norden, Osten, Süden und Westen zeigen.



Jedes Bild wird 1000 Millisekunden angezeigt. Mit einer for-Schleife kannst du den Vorgang dreimal wiederholen.

```

from microbit import *

for i in range(3):
    display.show(Image.ARROW_N)
    sleep(1000)
    display.show(Image.ARROW_E)
    sleep(1000)
    display.show(Image.ARROW_S)
    sleep(1000)
    display.show(Image.ARROW_W)
    sleep(1000)

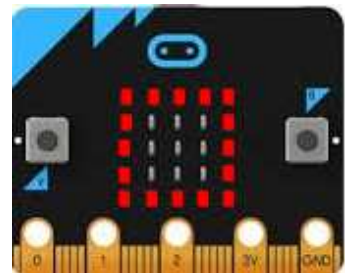
```

[Programmcode markieren](#) (Ctrl+C kopieren, Ctrl+V einfügen)

## Eigene Images erstellen

Du kannst auch eigene Images erstellen, indem du als Parameter von Image einen String definierst, der aus 5 Blöcken mit je 5 Zahlen 0 bis 9 besteht. Die Blöcke entsprechen den einzelnen Zeilen. Die Zahlen legen die Helligkeit im Bereich 0 (dunkel) bis 9 (ganz hell) fest.

Das folgende Programm zeigt ein Quadrat mit den Randpixeln in voller Helligkeit.



```

from microbit import *

img = Image('99999:90009:90009:90009:99999:')
display.show(img)

```



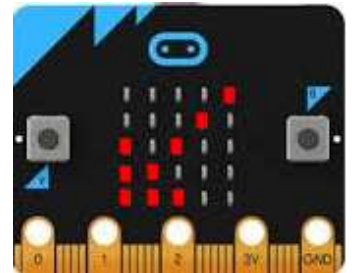
## ■ MERKE DIR...

Zur Ansteuerung des LED-Displays verwendest du das Objekt *display* mit den Funktionen *set\_pixel()*, *show()*, *scroll()* und *clear()*. Du orientierst dich am besten im TigerJython-.Menü unter *Hilfe* | *APLU-Dokumentation* | *micro:bit* | *MicroPython API* über die erlaubten Parameter.

## ■ ZUM SELBST LÖSEN

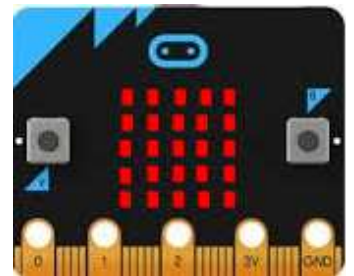
1. Schreibe ein Programm, welches nacheinander vier diagonale Pfeile darstellt:

Image.ARROW\_NW  
Image.ARROW\_NE  
Image.ARROW\_SW  
Image.ARROW\_SE.



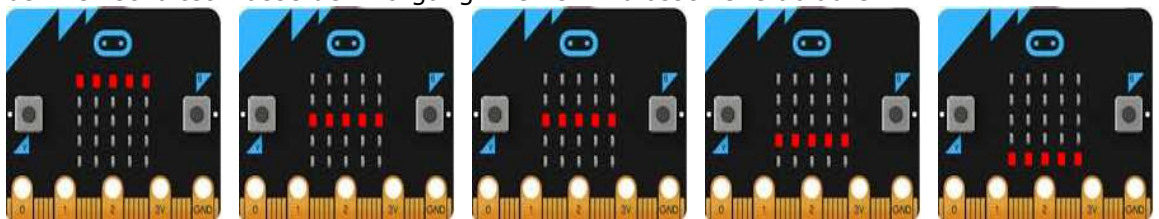
2. Mit einer doppelten for-Schleife

```
for y in range(5):  
    for x in range(5):  
        display.set_pixel(x, y, 9)  
        sleep(400)
```



kannst du alle LEDs der Reihe nach einschalten. Die LEDs sollen danach alle gleichzeitig mit dem Befehl *clear()* ausgeschaltet werden und zwar 1000 Millisekunden nachdem die letzte LED eingeschaltet wurde.

3. Schreibe ein Programm, welches zuerst die oberste LED-Reihe, dann die zweite, dritte usw. einschaltet. Lasse den Vorgang in einer Endlosschleife ablaufen.



### 3. BESCHLEUNIGUNGSSENSOR

---

#### ■ DU LERNST HIER...

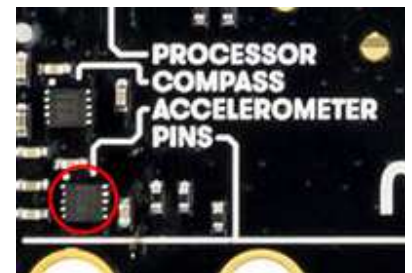
wie du mit dem Beschleunigungssensor Lageänderungen und Bewegungen des micro:bit erfassen kannst.

#### ■ SENSORWERTE

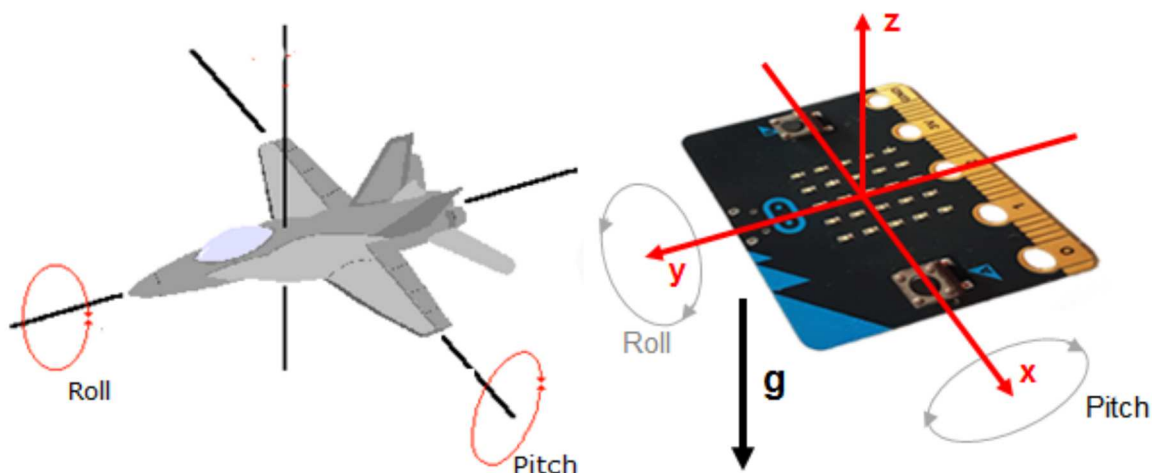
Der Beschleunigungssensor des micro:bit misst sowohl die konstante Erdbeschleunigung von rund  $10 \text{ m/s}^2$ , die vertikal nach unten zeigt und Beschleunigungen, die durch Bewegungen zustande kommen.

Der Beschleunigungssensor ist auf dem Board gut sichtbar. Ähnliche Sensoren sind auch in den meisten Smartphones eingebaut.

Der Sensor kann die Beschleunigung (inkl. Erdbeschleunigung) in x-, y, und z-Richtung messen. Aus diesen Komponenten kann die Vorwärtsneigung (Pitch) und Seitwärtsneigung (Roll) berechnet werden.



Es ist ähnlich wie bei der Lageanzeige von Flugzeugen. Wenn man das Board nach vorne oder nach hinten neigt, ändert der Pitch-Winkel, bei der Seitwärtsneigung ändert der Roll-Winkel.



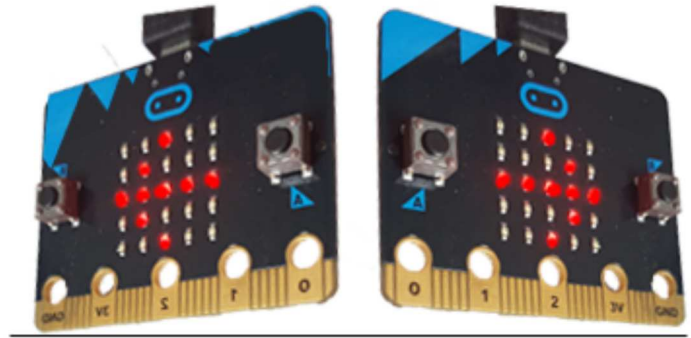
Im Programm verwendest du das Accelerometer-Objekt und rufst `accelerometer.get_x()`, `accelerometer.get_y()` oder `accelerometer.get_z()` auf, die Werte im Bereich von ungefähr -2000 bis 2000 liefern, die den Beschleunigungen  $-20 \text{ m/s}^2$  und  $20 \text{ m/s}^2$  entsprechen. Mit `accelerometer.get_values()` erhältst du alle drei Werte in einem Tupel zurück.

#### ■ MUSTERBEISPIELE

##### Sensorwerte abfragen

Starte das unten stehende Programm, kippe den micro:bit ausgehend von der horizontalen Lage nach links bzw. nach rechts unten und beobachte dabei die Sensorwerte, die im Terminalfenster ausgeschrieben werden.

Im Programm fragst du einer Endlosschleife die Beschleunigung in x-Richtung alle 100 Millisekunden ab und schreibst sie mit `print(acc)` im Terminalfenster aus. Zudem zeigst du auf dem Display bei einem positiven Wert einen Links- und bei einem negativen Wert einen Rechtspfeil.



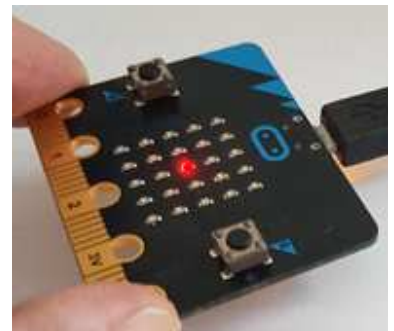
```
from microbit import *

while True:
    acc = accelerometer.get_x()
    print (acc)
    if acc > 0:
        display.show(Image.ARROW_E)
    else:
        display.show(Image.ARROW_W)
    sleep(100)
```

## Wasserwaage-Libelle



Der micro:bit soll wie eine Wasserwaage-Libelle funktionieren. Dabei verwendest du die Tatsache, dass die x- und y-Komponenten der Beschleunigungen ungefähr 0 sind, wenn sich das Board in waagrechter Lage befindet.



Es wird nur ein Pixel angezeigt, das sich je nach Neigung nach rechts, links, oben oder unten verschiebt. Ziel ist es, die Libelle so auszurichten, dass die mittlere LED leuchtet.

Im Programm verwendest du vier Bedingungen, um den x- und y-Wert des Pixels zu bestimmen und löschst dann das Pixel bevor du das neue anzündest.

```
from microbit import *

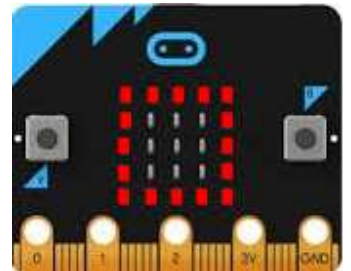
x = 1
y = 1
while True:
    accX = accelerometer.get_x()
    accY = accelerometer.get_y()
    if accX > 100 and x < 4:
        x += 1
    elif accX < -100 and x > 0:
        x -= 1
    elif accY > 100 and y < 4:
        y += 1
    elif accY < -100 and y > 0:
        y -= 1
    display.clear()
    display.set_pixel(x, y, 9)
    sleep(100)
```

## ■ MERKE DIR...

Mit dem Beschleunigungssensor kannst du die Bewegungen und die Lage des micro:bits erfassen. Meist werden die Sensorwerte in einer Messschleife regelmässig alle 10 - 100 Millisekunden abgefragt. Man nennt dies auch *Pollen der Sensorwerte*.

## ■ ZUM SELBST LÖSEN

1. Ergänze das erste Beispiel so, dass auch die Drehung des micro:bits in der y-Richtung gemessen und mit den passenden Pfeilen `Image.ARROW_N`, bzw. `Image.ARROW_S` angezeigt wird.
2. Durch Kippen des micro:bits sollen alle Pixel am Rand einzeln eingeschaltet werden.



3. Du willst das "Schütteln" des micro:bits messen. Dazu holst du mit `a = accelerometer.get_values()` die Beschleunigungskomponenten, betrachtest aber nur den Betrag `b` der x- und y-Komponenten, der sich wie folgt berechnet

$$b = \sqrt{a[0] * a[0] + a[1] * a[1]}$$

- a) Schreibe im Terminalfenster den Betrag in einer Endlosschleife 50x pro Sekunde aus.
- b) Führe nach Start des Programm die Messungen nur während 1 Sekunde aus und schreibe das Maximum von `b` aus.
- c) Schreibe das Maximum als Integer auf dem Display aus (als endlosen Scrolltext).  
Anleitung: Um aus `b` einen Integer zu machen, verwendest du `n = int(b)`. Für den Display musst du daraus mit `s = str(n)` einen String (Text) machen.
- d) Mache daraus ein "Schüttelspiel", bei dem derjenige gewinnt, der am stärksten schüttelt. Die Messung soll aber erst beim Erscheinen des Zeichens `>` nach einer zufälligen Zeit von 3-6 Sekunden beginnen und man darf vorher nicht bereits schütteln.

## 4. BUTTONS

---

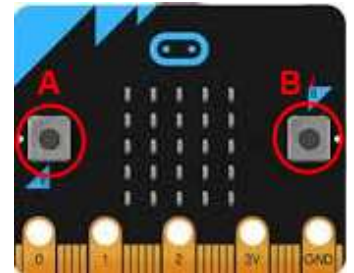
### ■ DU LERNST HIER...

wie man die beiden Tastenschalter (Buttons) verwendet, um interaktive Programme zu entwickeln.

### ■ AUF DRÜCKEN EINES BUTTONS REAGIEREN

Der micro:bit verfügt über zwei programmierbare Buttons. Der linke wird mit der Variablen *button\_a*, der rechte mit *button\_b* angesprochen.

Die Funktion ***button\_a.is\_pressed()*** gibt *True* zurück, wenn der linke Button im Moment des Aufrufs gedrückt ist (analog *button\_b.is\_pressed()*).



Dein Programm wartet in einer Endlosschleife, bis du den Button A drückst. Dann beginnt die mittlere LED mit einer Periode von 2s zu blinken, und zwar so lange du den Button gedrückt hältst.

Zur besseren Strukturierung definierst du eine Funktion *blink(x, y)*, welche die LED an der Position x, y ein- und wieder ausschaltet.



```
from microbit import *

def blink(x, y):
    display.set_pixel(x, y, 9)
    sleep(1000)
    display.set_pixel(x, y, 0)
    sleep(1000)

while True:
    if button_a.is_pressed():
        blink(2, 2)
        sleep(10)
```

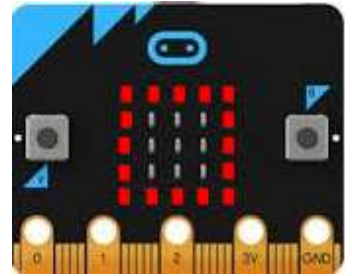
**Programcode markieren** (Ctrl+C kopieren, Ctrl+V einfügen)

Das überflüssig erscheinende *sleep(10)* ist wichtig, damit du nicht unnötig viele Rechnerressourcen verschwendest, wenn das Programm nichts anderes machen muss, als zu überprüfen, ob der Button gedrückt ist. In der Fachsprache sagt man auch, dass der Zustand des Buttons in der Endlosschleife "gepollt" wird.

## ■ AUF KLICKEN EINES BUTTONS REAGIEREN

So wie du es von den Mausklicks kennst, möchtest du hier mit einem Button-Klick ein laufendes Programm unterbrechen und eine andere Aktion ausführen. In einer endlosen while-Schleife lässt du die mittlere LED mit einer Periode von 2 s blinken. Ein Klick auf den Button A soll das Blinken unterbrechen und ein Quadrat anzeigen.

Leider funktioniert das folgende Programm mit `button_a.is_pressed()` nicht richtig.



```
from microbit import *

def blink(x, y):
    display.set_pixel(x, y, 9)
    sleep(1000)
    display.set_pixel(x, y, 0)
    sleep(1000)

while True:
    if button_a.is_pressed():
        display.show(Image.SQUARE)
        sleep(1000)
        display.clear()
    blink(2, 2)
    sleep(10)
```

[Programmcode markieren](#) (Ctrl+C kopieren, Ctrl+V einfügen)

Da das Programm die meiste Zeit in der blink-Funktion verbringt, kann es nicht merken, wenn du während dieser Zeit den Button drückst und wieder loslässt.

Um einen Klicks trotzdem zu erfassen, musst du die Funktion **`button_a.was_pressed()`** verwenden. Diese liefert True, wenn du irgend einmal den Button geklickt hast. Der Klick wird als **Event** aufgefasst, der vom System auch dann registriert wird, wenn dein Programm gerade etwas anderes macht.

Das so korrigierte Programm reagiert nun wunschgemäß auf Klicks.

```
from microbit import *

def blink(x, y):
    display.set_pixel(x, y, 9)
    sleep(1000)
    display.set_pixel(x, y, 0)
    sleep(1000)

while True:
    if button_a.was_pressed():
        display.show(Image.SQUARE)
        sleep(1000)
        display.clear()
    blink(2, 2)
    sleep(10)
```

[Programmcode markieren](#) (Ctrl+C kopieren, Ctrl+V einfügen)



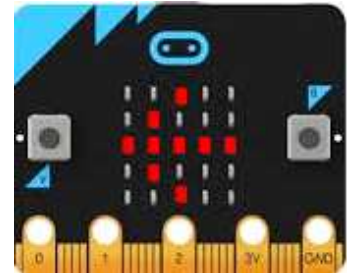
## ■ MERKE DIR...

Du kannst interaktive Programme entwickeln, die auf einen gedrückt gehaltenen Button oder auf einen Button-Klick reagieren. Mit der Funktion ***is\_pressed()*** muss der Button im Moment des Funktionsaufrufs gedrückt sein, damit sie True zurückgibt, mit der Funktion ***was\_pressed()*** wird True zurückgegeben, wenn seit dem dem Start des Programms oder seit dem letzten Aufruf der Button irgendwann mal geklickt wurde.

## ■ ZUM SELBST LÖSEN

1. Programmiere die Buttons wie folgt:

Wenn du den Button A drückst, so erscheint ein Pfeil, der nach links zeigt (ARROW\_W), wenn du den rechten Button drückst, so erscheint ein Pfeil nach rechts (ARROW\_E).



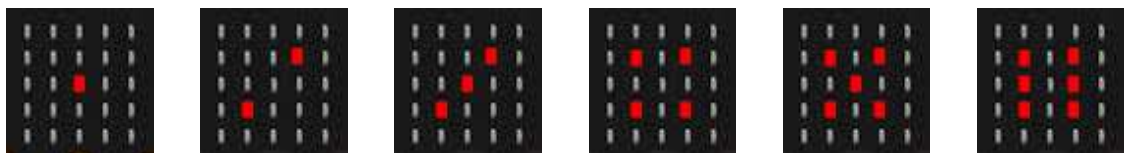
2. Programmiere einen Klickzähler: Bei jedem Klick auf den Button A wird die Anzahl  $n$ , die zu Beginn 0 ist, um eins vergrößert und zur Kontrolle im Terminalfenster ausgeschrieben. Beim Klick auf den Button B soll die Totalzahl als Lauftext auf dem Display erscheinen.

Bemerkung Der Befehl *scroll()* kann nur Texte anzeigen. Mit *str(n)* kannst du eine Zahl in einen Text umwandeln.

3. Du simulierst mit dem micro:bit das Werfen eines Würfels. Bei jedem Klicken auf den Button A wird auf dem Display das Bild einer zufälligen Würfelseite dargestellt.

Anmerkung:

Am einfachsten erstellst du die Bilder als Images z. B. die Vier mit  
`img = Image('00000:09090:00000:09090:00000:')`



## 5. SOUND

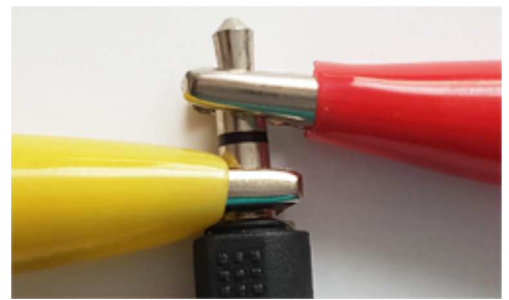
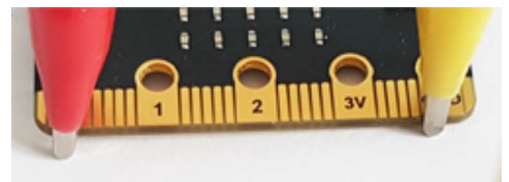
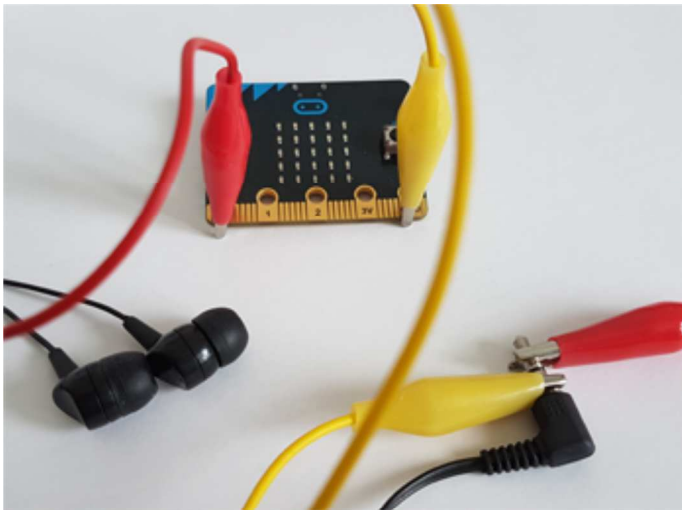
---

### ■ DU LERNST HIER...

wie man an den micro:bit einen Kopfhörer oder Lautsprecher anschliesst, um Töne, Tonfolgen und kurze Melodien abzuspielen.

### ■ SOUNDANLAGE AUFBAUEN

Um mit dem micro:bit Sounds abzuspielen, brauchst du einen Kopfhörer und 2 Kabel mit Krokodilklemmen. Mit dem ersten Kabel (hier gelb) verbindest du den Pin GND mit dem hinteren Teil des Kopfhörersteckers. Mit dem zweiten Kabel (hier rot) verbindest du den Pin0 mit dem vorderen Teil des Kopfhörersteckers. Damit du den Sound in den beiden Hörmuscheln hörst, muss du versuchen, die Krokodilklemme so zu plazieren, dass sie die beiden vorderen Teile des Steckers berührt.



### ■ MUSTERBEISPIELE

#### Eine Tonfolge abspielen

Mit dem Befehl `pitch(f, 500)` wird ein Ton mit der Frequenz  $f$  während 500 Millisekunden abgespielt. Um mehrere Töne nacheinander abzuspielen, gibst du die zugehörigen Frequenzen in einer Liste an und durchläufst diese mit einer for-Schleife.

Eine Tabelle mit Tönen und ihren Frequenzen findest du im Overlay-Fenster:

```
from music import *

song = [262, 294, 330, 349, 392, 392, 392, 0, 440, 440, 440, 440, 392]
for f in song:
    pitch(f, 500)
```

#### Eingebaute Melodien abspielen

Einige Melodien sind bereits eingebaut. Ihre Namen findest du hier oder in der Dokumentation. Du kannst das Abspielen der Melodien mit der Verwendung der Buttons kombinieren. Wenn du den Button A drückst, wird die Melodie JUMP\_UP abgespielt, beim Drücken des Buttons B die Melodie JUMP\_DOWN.

```

from microbit import *
from music import *

while True:
    if button_a.was_pressed():
        play(JUMP_UP)
    if button_b.was_pressed():
        play(JUMP_DOWN)
    sleep(10)

```

Was passiert, wenn du beide Button miteinander gedrückt hältst?

## Melodien in musikalischer Notation abspielen

Die Notation hält sich an folgende Regeln:

- Die zwölf Halbtöne einer Oktave haben mit Kreuz geschriebene Bezeichnungen C, C#, D, D#, E, F, F#, G, G#, A, A#, H (auch mit Kleinbuchstaben). Für die Pause verwendet man die Note "R"
- Will man die Oktave wechseln, so schreibt man eine Oktavezahl hinter die Note, also C2 für die zweite Oktave (zu Beginn ist man in der 4. Oktave). Diese Oktave gilt dann für alle folgenden Noten, bis wieder eine Oktavezahl angegeben wird
- Hinter der Note kann ein Doppelpunkt mit einer Angabe der Dauer (in Ticks) stehen. Alle folgenden Noten werden dann mit dieser Dauer abgespielt, bis wieder eine neue Angabe der Dauer folgt.

Um eine Melodie zu komponieren, schreibst du die Noten in eine Liste, beispielsweise im folgenden Programm für einen Dur- und nachfolgenden Moll-Akkord.

```

from music import *

melody = ['e:4', 'f#', 'g#', 'r:2', 'e', 'f#', 'g:4']
play(melody)

```

## Ein akustisches Lagemessgerät

Messgeräte mit akustischen Ausgaben sind sehr beliebt. Du erzeugst hier ein Tonsignal, dessen Höhe von der Seitwärtsneigung des micro:bit abhängt. Ausgehend von einer Anfangsfrequenz  $f_0 = 1000$ , berechnest du die steigenden bzw. fallenden Tonfrequenzen für Halbtöne der wohltemperierten Stimmung und spielst jeden Ton 100 Millisekunden lang ab.

Mit dem Button B kannst du das Tonsignal ausschalten.

```

from music import *
from microbit import *

def beep(n):
    freq = int(f0 * r**n)
    pitch(int(r**n * f0), 100)
    sleep(50)

f0 = 1000
r = 2**(1/12)
while not button_b.was_pressed():
    n = int(accelerometer.get_x() / 100)
    beep(n)
    sleep(10)

```

## ■ MERKE DIR...

Den Kopfhörer musst du an die Ausgänge, die mit GND und P0 bezeichnet sind, anschliessen. Einen Ton mit der Frequenz  $f$  spielst du mit dem Befehl `pitch(f, time)` ab. Mit `play(song)` kannst du ganze Melodien, entweder als Tonfolge oder in musikalischer Notation abspielen.

## ■ ZUM SELBST LÖSEN

1. a) Schreibe ein Programm, welches die folgende Tonfolge abspielt:

`song = [262, 294, 330, 262, 262, 294, 330, 262, 330, 349, 392, 330, 349, 392]`

Du kannst selbstverständlich auch eigene Tonfolgen komponieren.

- b) Du möchtest die Abspielgeschwindigkeit interaktiv ändern:

- Wenn du den Button A klickst, soll der Song doppelt so schnell abgespielt werden
- Wenn du den Button B klickst, erfolgt das Abspielen langsamer.

Beachte, dass im Befehl `pitch(f, time)` der Parameter `time` eine ganze Zahl sein muss.

2. Du kannst eine Melodie wiederholt abspielen lassen, indem du `play()` mit zusätzlichen Parametern verwendest, beispielsweise für die Melodie BIRTHDAY

`play(BIRTHDAY, wait = False, loop = True)`

Wegen `wait = False`, erfolgt das Abspielen im Hintergrund und das Programm läuft weiter.

Schreibe ein Programm, dass das Geburtstagslied solange endlos abspielt, bis man den Button A klickt. (Anmerkung: Du kannst eine Soundausgabe mit dem Befehl `stop()` beenden.)

3. Schreibe die folgende Melodie in der musikalischen Notation und spiele sie ab.

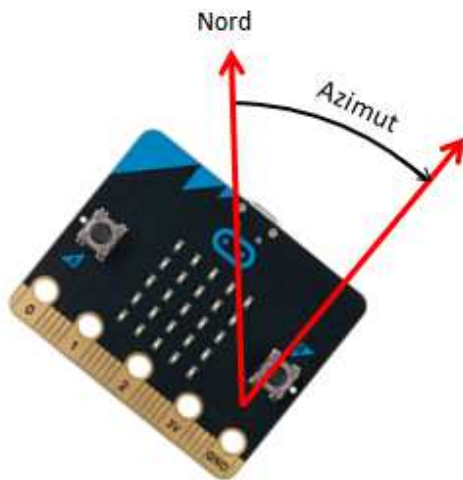


## 6. MAGNETFELDSSENSOR, KOMPASS

### ■ DU LERNST HIER...

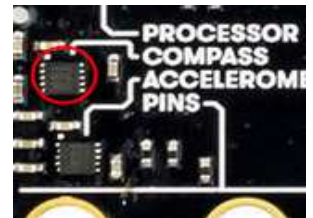
wie du mit dem micro:bit das Magnetfeld messen und die Himmelsrichtung (das Azimut) bestimmen kannst.

### ■ SENSOR KALIBRIEREN UND WERTE ANZEIGEN



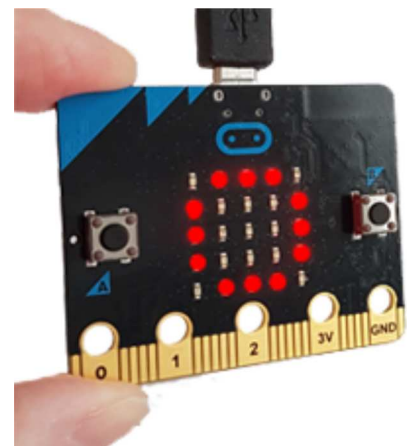
Du kannst den Magnetfeldsensor dazu benutzen, wie mit einem Kompass die Lage des micro:bits in Bezug auf die Nordrichtung zu bestimmen. Dazu verwendest du den Befehl `compass.heading()`, der bei horizontaler Lage des Boards das Azimut in Grad (positiv gegen Osten) zurückgibt.

Der Sensor ist auf deinem micro:bit gut sichtbar und ist mit COMPASS beschriftet.



Bevor du mit der Messung beginnen kannst, musst du den Sensor mit dem Befehl `compass.calibrate()` kalibrieren. Nach dem Aufruf blockiert das Programm und es erscheint auf dem Display die Aufforderung, eine Kreisfigur zu erstellen (*Draw a circle*). Kippe das Board in alle Richtungen, bis alle äusseren LEDs einen leuchtenden Kreis bilden.

Sogleich wird auf dem Display das Image HAPPY angezeigt und das Programm läuft weiter. In der Endlosschleife schreibst du das Azimut als ganze Zahl zwischen 0 und 360 aus. Wie üblich kannst du mit Ctrl+C die Ausgabe im Terminalfenster abbrechen.



```
from microbit import *

compass.calibrate()
while True:
    print(compass.heading())
    sleep(300)
```

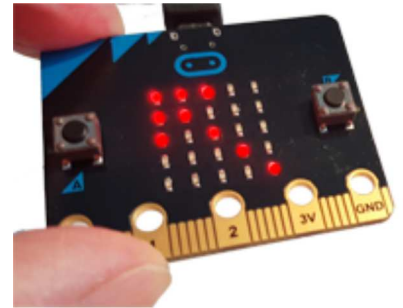
**Programmcode markieren** (Ctrl+C kopieren)

## ■ MUSTERBEISPIELE

### Einen Kompass erstellen

Wie bei einem realen Kompass soll auf dem Display ein Pfeil erscheinen, der ungefähr nach Norden zeigt.

Da du nur Pfeile in 45-Grad-Richtungen darstellen kannst, teilst du das Azimut in 8 Bereiche ein und ordnest jedem Bereich denjenigen Pfeil zu, der am besten passt. Wenn der Sensor noch nicht kalibriert ist, musst du zuerst die Kalibrierung durchführen.



Eigentlich müsstest du bei den Bereichsgrenzen auch auf die Gleichheit testen. Da der Sensor aber nur ganzzahlige Werte zurückgibt, kannst du darauf verzichten.

```
from microbit import *

if not compass.is_calibrated():
    print("Perform calibration please!")
    compass.calibrate()

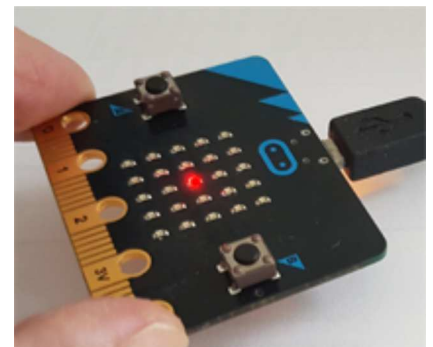
while True:
    h = compass.heading()
    print(h)
    if h > 22.5 and h < 67.5:
        display.show(Image.ARROW_NW)
    elif h > 67.5 and h < 112.5:
        display.show(Image.ARROW_W)
    elif h > 112.5 and h < 157.5:
        display.show(Image.ARROW_SW)
    elif h > 157.5 and h < 202.5:
        display.show(Image.ARROW_S)
    elif h > 202.5 and h < 247.5:
        display.show(Image.ARROW_SE)
    elif h > 247.5 and h < 292.5:
        display.show(Image.ARROW_E)
    elif h > 292.5 and h < 337.5:
        display.show(Image.ARROW_NE)
    else:
        display.show(Image.ARROW_N)
    sleep(10)
```

[Programmcode markieren](#) (Ctrl+C kopieren)

### Minen suchen

Lege einen kleinen Magneten, wie du ihn für Memoboards finden kannst, unter einen Kartondeckel. Den Magneten kannst du wie eine Mine auffassen, die man mit dem micro:bit als Minensucher auffinden muss. In deinem Programm leuchtet die mittlere LED umso heller, je näher das Board zur Mine kommt.

Du verwendest den Befehl `compass.get_values()`, der die Stärke des Magnetfeldes als ein Tupel mit den x-, y- und z-Komponenten zurückgibt.



Dann berechnest du den Betrag aus diesen drei Komponenten und skalierst ihn so, dass du einen Helligkeitswert zwischen 0 und 9 erhältst.



```

from microbit import *
import math

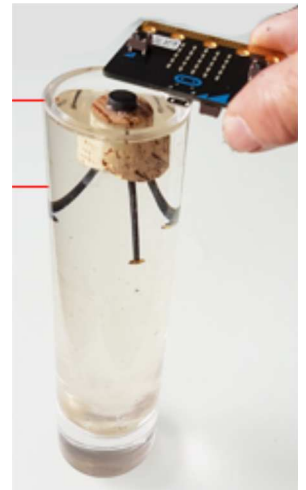
while True:
    v = compass.get_values()
    b = math.sqrt(v[0] * v[0] + v[1] * v[1] + v[2] * v[2])
    z = min(9, int(b / 500000)) # brightness
    display.set_pixel(2, 2, z)
    sleep(10)

```

### Wasserstand messen

Deine Testanlage besteht aus einer Blumenvase und einem Magneten, der auf einem schwimmenden Korkzapfen befestigt ist. Mit drei Nägeln kannst du den Zapfen so stabilisieren, dass sich der Magnet bei änderndem Wasserstand immer in der gleichen vertikalen Linie bewegt.

Aus der Messung des Betrages des Magnetfeldes erhältst du eine Information über die Höhe der Wassersäule. Die willst sie in 3 Wasserstandsbereiche einteilen.



Am besten stellst du dir vor, dass sich das Gefäß in drei "Zuständen" befindet, die du mit der Variablen *state* erfasst, welche die Werte "high", "ok" und "low" annehmen kann. Aus dem skalierten Wert des Magnetfelds triffst du den Entscheid, ob sich der Zustand ändert.

Falls du keinen Sound zur Verfügung hast, kannst du auch eine andere Alarmanzeige realisieren.

```

from microbit import *
import math
import music

state = "ok"
while True:
    v = compass.get_values()
    b = math.sqrt(v[0] * v[0] + v[1] * v[1] + v[2] * v[2])
    z = int(b / 100000)
    if z >= 15 and state != "high":
        state = "high"
        print("->high")
    elif z >= 8 and z < 15 and state != "ok":
        state = "ok"
        print("->ok")
    elif z < 8 and state != "low":
        state = "low"
        print("->low")
    if state == "high":
        music.pitch(2000, 100)
    elif state == "low":
        music.pitch(500, 100)
    sleep(10)

```

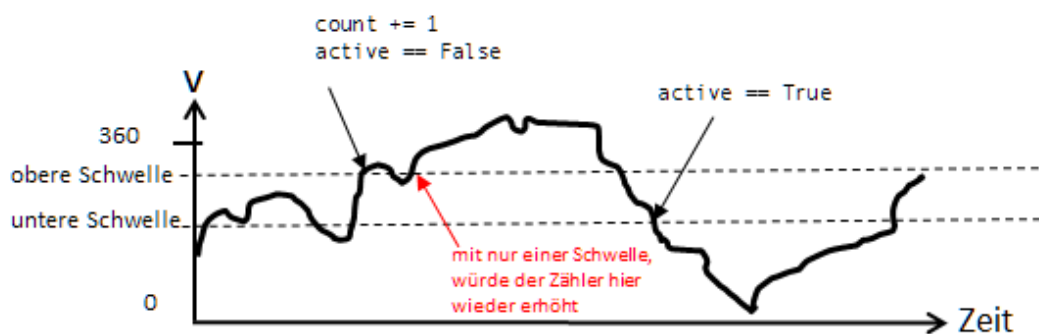
## ■ MERKE DIR...

Der Magnetfeldsensor kann im Zusammenhang mit Magneten oder zur Bestimmung der Himmelsrichtung als Kompasssensor verwendet werden. In dieser Anwendung muss er vor der ersten Verwendung kalibriert werden.

## ■ ZUM SELBST LÖSEN

1. Baue einen "Pirouettenzähler": Jedes Mal wenn du dich mit dem micro:bit in der Hand einmal um deine eigenen Achse gedreht hast, wird eine weitere LED auf dem Display eingeschaltet.

Anleitung: Wenn sich der micro:bit dreht, liefert `compass.heading()` einen unregelmässigen Signalverlauf.



Zum Zählen der Umläufe genügt ein einziger Schwellenpegel nicht, sondern es müssen zwei Pegel definiert werden. Beim Überschreiten des oberen Pegels wird der Zähler erhöht und gleichzeitig aber das Zählen inaktiv gemacht. Erst beim Unterschreiten des unteren Pegels wird das Zählen wieder aktiviert.

2. Berührungslose Umdrehungszähler (z.B. bei Rädern) verwenden oft einen Magneten, der bei jeder Umdrehung einmal vor einem Magnetfeldsensor vorbei läuft. Befestige ein Magnetchen (z.B. von einem Memoboards oder einem Magnetschnäpper) an einem Velorad und schreibe ein Programm, dass im Terminalfenster die Anzahl Umdrehungen ausschreibt. Wie lässt sich damit ein Kilometerzähler bauen?

## 7. BLUETOOTH-KOMMUNIKATION

---

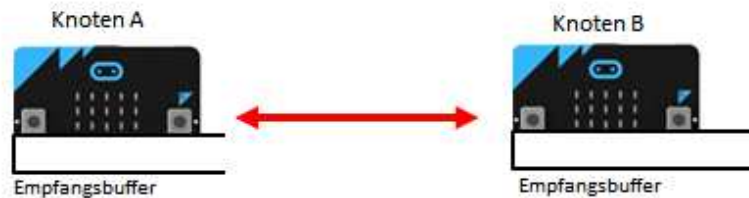
### ■ DU LERNST HIER...

wie zwei Computer über Bluetooth miteinander Informationen austauschen.

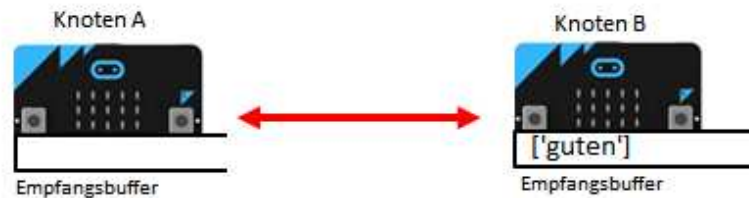
### ■ DRAHTLOSE KOMMUNIKATION

Bei der Kommunikation zwischen zwei micro:bits (auch Knoten genannt) ist jeder der Knoten gleichzeitig ein Sender und ein Empfänger. Zuerst schalten beide Knoten mit `radio.on()` das Sende-/Empfangsgerät ein. Dabei wird eine drahtlose Bluetooth-Verbindung erstellt. Nachfolgend kann jeder der Knoten mit `radio.send(msg)` eine Message (als String) an den anderen Knoten senden, die dort zuerst in einen Empfangsbuffer wie in einer Warteschlange gespeichert wird. Mit `radio.receive()` holst du die "älteste" Message, die dann im Buffer gelöscht wird.

- ▷ Knoten A und Knoten B haben `radio.on()` aufgerufen:



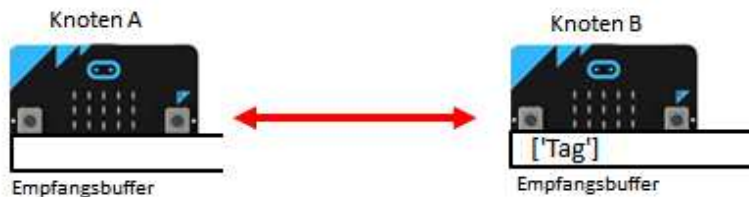
- ▷ Knoten A hat `send('guten')` aufgerufen:



- ▷ Knoten A hat `send('Tag')` aufgerufen:

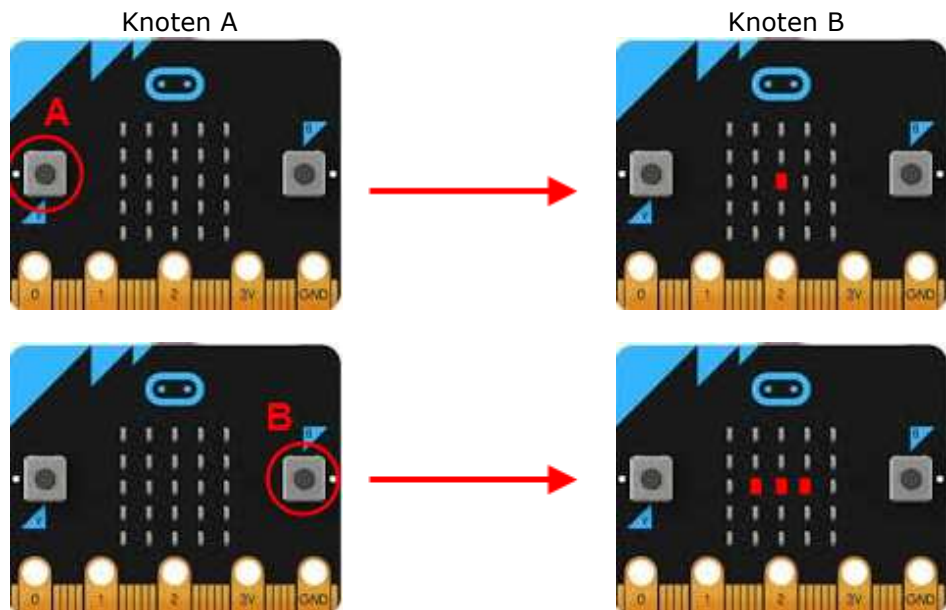


- ▷ Knoten B hat `msg = receive()` aufgerufen. `msg` enthält 'guten':



## ■ MUSTERBEISPIELE

Du erstellst mit einer anderen Person eine drahtlose Kommunikation, um Informationen mit Punkten und Strichen (wie im Morsecode) auszutauschen. Wenn man auf dem einen micro:bit den Button A klickt, so leuchtet beim anderen die mittlere LED des Displays während ½ s auf (wie ein Punkt), klickt man auf Button B so leuchten drei LEDs auf (wie ein Strich).



Der zentrale Teil des Programms besteht aus einer Endlosschleife, in der zuerst das Senden und dann das Empfangen geregelt wird. Wenn A geklickt wird, überträgst du die Message "p", (kurz für "point"), wenn B geklickt wird, überträgst du "d" (kurz für "dash").

Nachfolgend holst du mit `receive()` den ältesten Eintrag in der Warteschlange. (Falls diese leer ist, kriegst du den speziellen Wert `None`.) Je nachdem, ob du "p" oder "d" erhalten hast, führst du die entsprechende Aktion aus.

```
import radio
from microbit import *

point = Image('00000:00000:00900:00000:00000')
dash = Image('00000:00000:09990:00000:00000')

def draw(img):
    display.show(img)
    sleep(500)
    display.clear()
    sleep(500)

radio.on()
while True:
    if button_a.was_pressed():
        radio.send("p")
    elif button_b.was_pressed():
        radio.send("d")
    rec = radio.receive()
    if rec != None:
        if rec == "p":
            draw(point)
        elif rec == "d":
            draw(dash)
    sleep(10)
```

Echte Morsestationen arbeiten fast immer mit kurzen und langen Tönen. Dazu musst du einen Kopfhörer (oder wie im Kapitel *Sound* gezeigt einen aktiven Lautsprecher) anschliessen. Dein Programm soll jetzt beim Empfänger den Ton einschalten, solange der Button A gedrückt ist. Sobald der Button gedrückt wird, sendest du dem Empfänger die Message "d" (für "down"), wenn du den Button loslässt, so sendest du "u" (für "up"). Dazu musst du eine boolesche Variable *down* einführen, wo du dir merkst, ob der Button gedrückt oder losgelassen ist.

Orientiere dich über den Morsecode und versuche einige kurzen Meldungen (z.B. "SOS") auszutauschen.

```
import radio
from microbit import *
import music

radio.on()
down = False
while True:
    if button_a.is_pressed() and not down:
        down = True
        radio.send("d")
    elif not button_a.is_pressed() and down:
        down = False
        radio.send("u")
    rec = radio.receive()
    if rec != None:
        if rec == "d":
            music.pitch(500, 1000000, wait = False)
        elif rec == "u":
            music.stop()
    sleep(10)
```

[Programmcode markieren](#) (Ctrl+C kopieren)

## ■ MERKE DIR...

dass der Sender mit *send(msg)* eine Message sendet, die dann im Empfangsbuffer der Reihe nach gespeichert wird, bis der Empfänger die "älteste" mit *msg = receive()* im Buffer abholt.

## ■ ZUM SELBST LÖSEN

1. Modifiziere das Musterprogramm so, dass beim Drücken des Buttons A laufend Punkte und beim Drücken von B Striche (3x so lang wie Punkte) gesendet werden.
2. a) Schreibe ein Programm, das die aktuelle Neigung (in einer Richtung) des micro:bit einem Empfängerknoten mitteilt, der den Wert im Terminal ausschreibt.  
b) Schreibe den Wert beim Empfänger auf dem Display aus (skaliert auf eine Zahl zwischen 0 und 9)

## 8. FAHRENDE ROBOTER

---

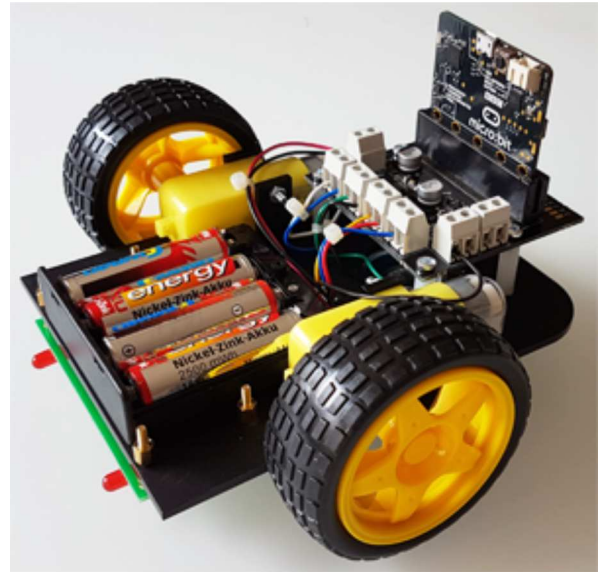
### ■ DU LERNST HIER...

wie du einen Roboter programmierst, damit dieser bestimmte Tätigkeiten selbständig ausführt.

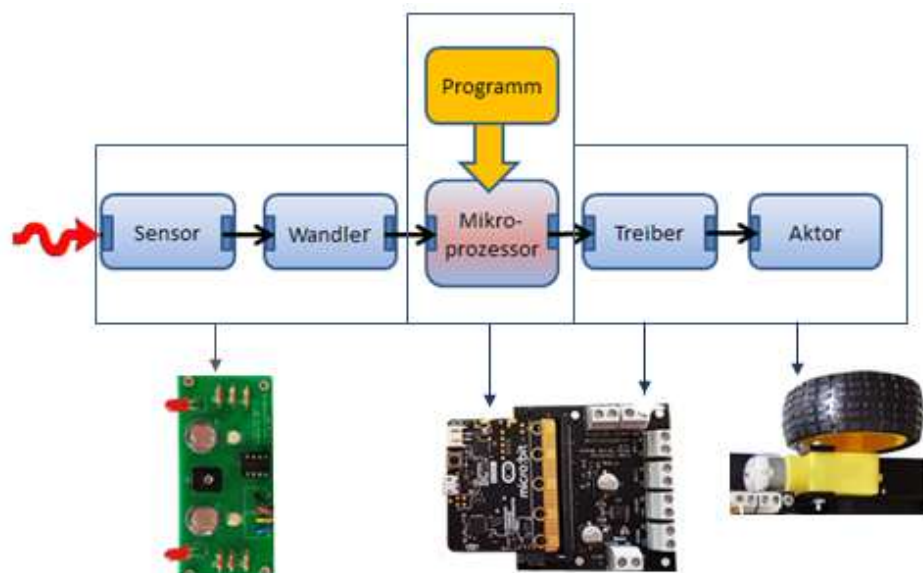
### ■ ROBOTER "BUGGY"

Mit wenig Aufwand kannst du einen fahrenden Roboter zusammenbauen, der Motoren und Lichtsensoren hat. Eine Bauanleitung findest [hier](#).

Die wichtigste Komponente des Buggy-Roboters ist der micro:bit. Dieser übernimmt die Funktion eines Gehirns, das auf Grund der Sensordaten, z.B. Hell-Dunkel-Werte von Lichtsensoren, Aktoren (Motoren, Display) betätigt.



Das folgende Blockbild für eine automatisierte Maschine ist darum sehr allgemein gültig und du erkennst diese Komponenten leicht bei deinem Buggy-Roboter.



Da der micro:bit in der Maschine integriert ist, nennt man sie auch ein "eingebettetes System" (embedded system).

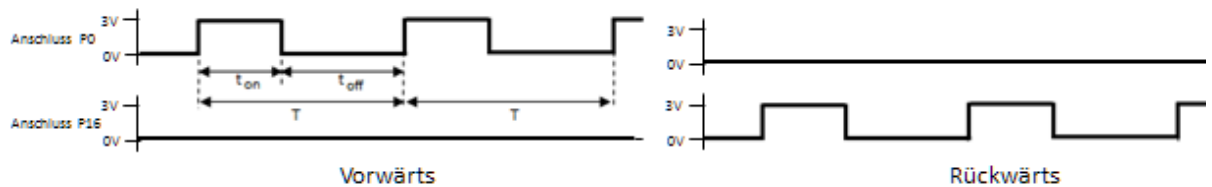


## ■ MUSTERBEISPIELE

Bevor du ein anspruchsvolleres Problem anpackst, solltest du immer einige Tests durchführen, um das richtige Funktionieren der Komponenten einzelnen zu überprüfen und den Code zur Ansteuerung der Hardware kennen zu lernen.

### 1. Tests mit dem linken und rechten Motor

Die Motoren besitzen zwei Anschlüsse. Wenn du sie mit einer Spannung von 3 V versorgst,, so drehen sie in der einen oder anderen Richtung, je nachdem, wo der positive und negative Pol ist. Um die Geschwindigkeit der Motoren zu regeln, werden sie nur in bestimmtem Intervallen mit 3 V versorgt, in der restlichen Zeit ist die Spannung 0 V. Das Ansteuerungssignal hat also für den linken Motor mit den Anschlüssen P0 und P16 folgenden zeitlichen Verlauf (der rechte Motor hat die Anschlüsse P12 und P8):



Solche Signale werden als Puls-Width-Modulation (PWM) bezeichnet und du erzeugst ein PWM-Signal beispielsweise am Pin0 mit dem Befehl: `pin0.write_analog(duty)`, wo *duty* eine Zahl zwischen 0 und 1023 ist, welche die Einschaltdauer  $t_{on}$  festlegt. Mit folgendem Programm dreht der linke Motor zuerst 2000 Millisekunden vorwärts und dann 2000 Millisekunden rückwärts. Damit die Motoren mit den Batterien versorgt werden, musst du den kleinen Schalter auf der unteren Seite des Buggys auf *on* stellen. Mit diesem Schalter kannst du die Motoren auch jederzeit ausschalten.

```
from microbit import *

pin0.write_analog(200)
sleep(2000)
pin0.write_analog(0)

pin16.write_analog(100)
sleep(2000)
pin16.write_analog(0)
```

Führe den gleichen Test auch mit dem rechten Motor durch, indem du die Pins 12 und 8 verwendest.

### 2. Tests mit den Lichtsensoren

Die Lichtsensoren geben eine kleinere oder grössere Spannung ab, je nachdem, ob sich der Roboter auf einer hellen oder dunklen Unterlage befindet.



Wenn sich der Buggy auf einer dunklen Unterlage befindet, leuchten zusätzlich die beiden roten LEDs auf, auf der hellen sind sie ausgeschaltet. Der linke Sensor ist an Pin1, der rechte an Pin2 angeschlossen. Dein Programm "pollt" die Spannung an diesen Anschlüssen und schreibt den Messwert im Terminal aus.

```
from microbit import *

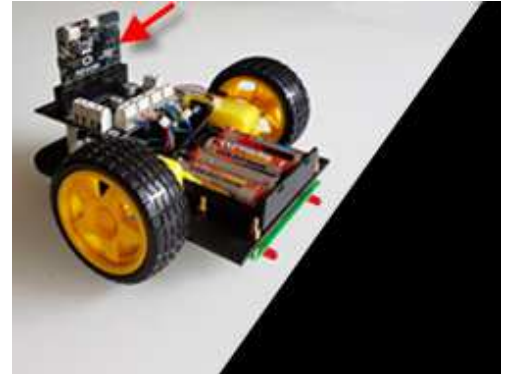
while True:
    left = pin1.read_analog()
    right = pin2.read_analog()
    print("L: " + str(left) + " - R: " + str(right))
    sleep(500)
```

Mit einem kleinen Schraubenzieher kannst du am Potentiometer drehen und den Schaltpegel zwischen hell und dunkel verändern. Stelle ihn so ein, dass dein Programm feststellt, wenn sich unter dem Buggy ein heller bzw. ein dunkler Untergrund befindet.



### 3. Roboter steuern

Im nächsten Programm lernst du einige Steuerungsmöglichkeiten kennen. Du startest die Bewegung mit Klick auf den Button A. Dies ist vor allem praktisch, wenn du den Roboter am Boden fahren lässt. Damit der Roboter vorwärts fährt, musst du beide Motoren vorwärts rotieren lassen. Schaltest du den rechten Motor ab, dreht der Buggy nach rechts, schaltest den rechten Motor wieder ein, fährt er wieder vorwärts. Auf einer dunklen Unterlage soll der Roboter anhalten.



```
from microbit import *

while not button_a.was_pressed():
    sleep(10)
pin0.write_analog(150)
pin12.write_analog(150)
sleep(2000)
pin12.write_analog(0)
sleep(2000)
pin12.write_analog(150)

while True:
    left = pin1.read_analog()
    print(left)
    if left > 100:
        pin0.write_analog(0)
        pin12.write_analog(0)
    sleep(10)
```

**Programmcode markieren** (Ctrl+C kopieren, Ctrl+V einfügen)

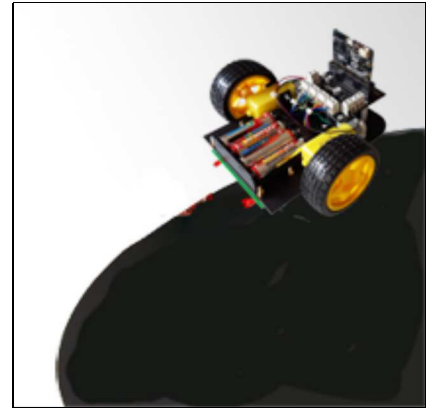
### 4. Befehle aus dem Modul *mbutils* verwenden

Noch einfacher kannst du den Buggy steuern, wenn du die Befehle aus dem Modul *mbutils* verwendest (siehe auch [Dokumentation](#)):

buggy_forward()	buggy_leftArc(r)
buggy_backward()	buggy_rightArc(r)
buggy_right()	buggy_setSpeed(speed)
buggy_left()	isDark(ldr)
buggy_stop()	

Für eine langsame Richtungsänderung sind die Befehle *buggy\_rightArc(r)* und *buggy\_leftArc(r)* geeignet, wobei du für r Zahlen zwischen 0 und 1 wählen kannst. (r ist der Reduktionsfaktor, um den das eine Rad langsamer dreht.)

Der Buggy soll auf dem Rand der schwarzen Fläche fahren. Die Strategie ist einfach: Du musst geradeaus fahren, wenn der linke Sensor dunkel und der rechte Sensor hell sieht. Du fährst einen Rechtsbogen, wenn beide Sensoren dunkel sehen und einen Linksbogen, wenn beide Sensoren hell sehen.



```
from microbit import *
from mbutils import *

display.show(Image.YES)
while not button_a.was_pressed():
    sleep(10)
buggy_setSpeed(15)
buggy_forward()

while not button_b.was_pressed():
    if isDark(ldrL) and not isDark(ldrR):
        buggy_forward()
    elif isDark(ldrL) and isDark(ldrR):
        buggy_rightArc(0.6)
    elif not isDark(ldrL) and not isDark(ldrR):
        buggy_leftArc(0.6)
    sleep(10)
buggy_stop()
display.show(Image.NO)
```

[Programmcode markieren](#) (Ctrl+C kopieren, Ctrl+V einfügen)

## 5. Buggy mit einem zweiten micro:bit fernsteuern

Du willst den Buggy via Bluetooth fernsteuern. Dazu verwendest du als Fernsteuerung einen zweiten micro:bit. Wenn du den linken Button A drückst, wird die Message "LEFT" gesendet und der Buggy fährt nach links. Drückst du den rechten Button, so wird "RIGHT" gesendet und er biegt nach rechts, drückst du beide Buttons gleichzeitig, wird "FORWARD" gesendet und der Buggy fährt gerade aus. Wenn du die Button löslässt, wird "STOP" gesendet und der Buggy hält an.



Das Programm für den Buggy:

```
from microbit import *
from mbutils import *
import radio

radio.on()
display.show(Image.YES)
buggy_setSpeed(20)
while True:
    rec = radio.receive()
    if rec == "FORWARD":
```

```

        buggy_forward()
    elif rec == "LEFT":
        buggy_leftArc(0.6)
    elif rec == "RIGHT":
        buggy_rightArc(0.6)
    elif rec == "STOP":
        buggy_stop()
    sleep(10)

```

[Programmcode markieren](#) (Ctrl+C kopieren, Ctrl+V einfügen)

Das Programm für die Steuerung:

```

import radio
from microbit import *

radio.on()
display.show(Image.YES)
state = "STOP"
oldState = ""
while True:
    if button_a.is_pressed() and button_b.is_pressed():
        state = "FORWARD"
    elif button_a.is_pressed():
        state = "LEFT"
    elif button_b.is_pressed():
        state = "RIGHT"
    else:
        state = "STOP"
    if oldState != state:
        radio.send(state)
        oldState = state
    sleep(10)

```

[Programmcode markieren](#) (Ctrl+C kopieren, Ctrl+V einfügen)

Das Programm für die Fernsteuerung ist ein typisches Beispiel für die Zustandsprogrammierung. Da der die Buttons in einer endlosen while-Schleife alle 10 Millisekunden überprüft werden, muss du darauf achten, dass nur bei einer Zustandsänderung (*oldState != state*) eine Message gesendet wird. Sonst wird alle 10 ms ein überflüssiger Befehl gesendet und damit das System unnötig belastet .

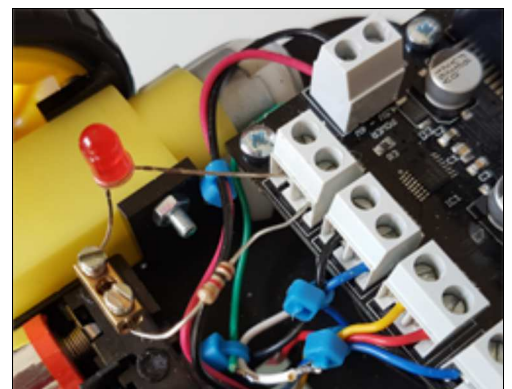
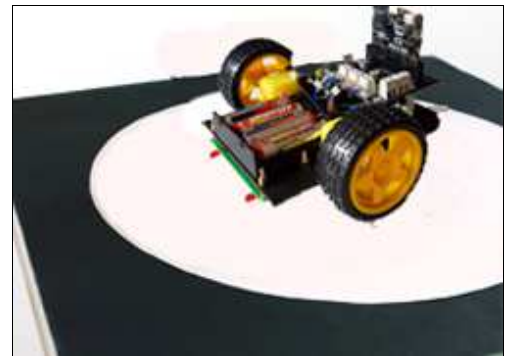
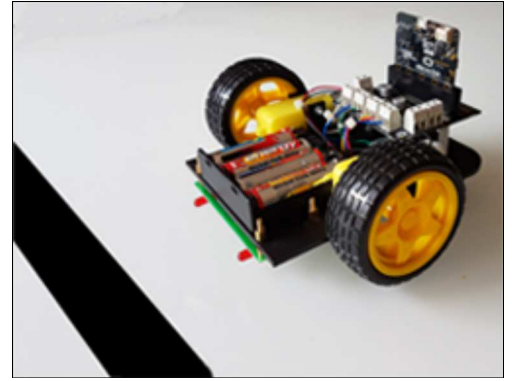
## ■ MERKE DIR...

Um ein komplexes System in Betrieb zu setzen, muss man einzelne Komponenten einem Test unterziehen und erst dann alle miteinander betreiben. Ein Motor bleibt so lange in einem Zustand, bis du ihn in einen anderen Zustand versetzt.

Für die Steuerung kannst du auch Befehle aus dem Modul *mbutils* verwenden.

## ■ ZUM SELBST LÖSEN

1. Ein heller Untergrund wechselt wie bei einer Barriere auf einen dunklen Untergrund. Der Buggy fährt geradeaus auf die Barriere zu und hält dann 3 s an. Nachher fährt er 3 s zurück und hält an.
  - a) Löse die Aufgabe, indem du die Motoren und Lichtsensoren direkt über die einzelnen Pin-Ausgänge ansprichst.
  - b) Löse die Aufgabe mit den Befehlen aus dem Modul *mbutils*.
2. Du erstellst auf einem grossen Blatt Papier einen weiss-schwarzen Kreisring. Der Buggy soll endlos (bis du einen Button klickst) ausgehend von der Mitte zum Ring, dann wieder zurück fahren, etwas drehen und wieder vorwärts fahren.
3. Erstelle auf einem grossen Blatt Papier eine geschlossene Bahn (am besten mit schwarzem, selbstklebendem Filz, den du auf Rollen im Hobbymarkt findest). Schreibe ein Programm so, dass sich der Buggy längs der Bahn bewegt, bis du einen der Buttons klickst.
4. Du willst den Buggy via Bluetooth fernsteuern. Dazu verwendest du als Fernsteuerung einen zweiten micro:bit und die Fahrrichtungen werden durch dessen Lage bestimmt: Bei horizontaler Lage soll der Buggy anhalten. Kippst du die Fernsteuerung nach vorne, fährt der Buggy vorwärts, kippst du sie nach hinten, fährt er rückwärts, kippst du sie nach links, so fährt er nach links und kippst du sie nach rechts, so fährt er nach rechts.
5. PWM-Signale werden auch zum Dimmen von Beleuchtungen (LED-Lampen) verwendet. Schliesse an Stelle eines Motors eine kleine LED mit einem Vorwiderstand von 1 kOhm an und schreibe ein Programm, das die Helligkeit der LED endlos hoch und wieder runter regelt. Vergiss den in Serie geschalteten Vorwiderstand nicht, sonst wird die LED zerstört.



## 9. ALARMANLAGEN

---

### ■ DU LERNST HIER...

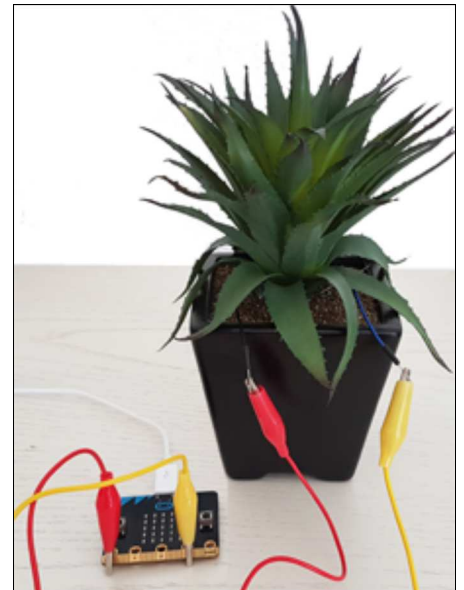
wie du den micro:bit einsetzen kannst, um Systeme zu überwachen und gegebenenfalls Alarm auszulösen.

### ■ WASSERSTAND ÜBERWACHEN

Deine kleine Pflanze musst du selten giessen, aber ganz vergessen darfst du es nicht. Mit deinem micro:bit kannst du den Wasserstand kontrollieren und wenn er zu niedrig ist, Alarm auslösen.

Du verwendest dabei die Eigenschaft von Wasser, Strom zu leiten. Wenn du also von der 3V Spannungsversorgung einen Stromkreis zum Pin0 erstellst, der durch das Wasser führt, so wird am Pin0 eine höhere Spannung zu messen sein, als wenn der Stromkreis offen ist. Du brauchst für den Aufbau nur zwei Kabel mit Krokodilklemmen und zwei Draht- oder sonstige Metallstücke, die du als Sonden in den Blumentopf steckst.

Das eine Kabel verbindest du mit Pin0, das andere mit 3V.



Für die Programmentwicklung kannst du einen beliebigen Wasserbehälter verwenden.

Mit dem Befehl `pin0.read_analog()`

misst du die Spannung an P0 und schreibst den Werte ins Terminalfenster. Wie du siehst, ist der Unterschied der Messwerte gross, je nachdem ob die Sonden im Wasser sind oder nicht.

```
from microbit import *

while True:
    v = pin0.read_analog()
    print(v)
    sleep(500)
```

[Programmcode markieren](#) (Ctrl+C kopieren, Ctrl+V einfügen)

Ergänze deine Anlage mit einer optischen Alarmanzeige auf dem LED-Display.

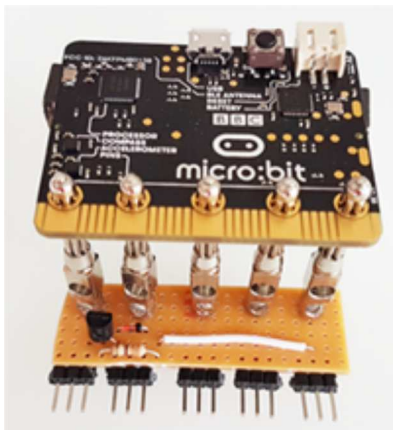
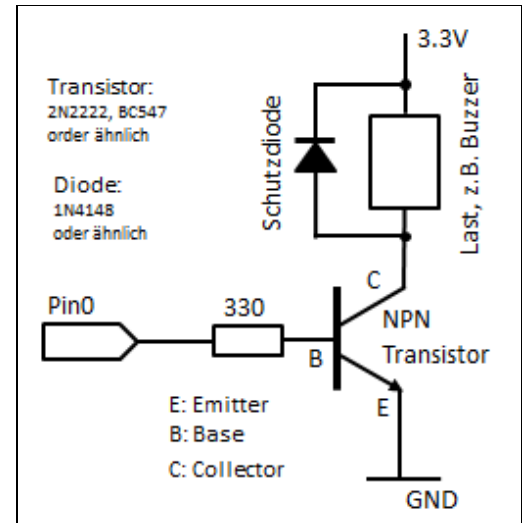
Wenn die einen akustischen Alarm auslösen willst, so eignet sich dazu ein Buzzer. Diesen kannst du aber nicht direkt mit dem micro:bit ansteuern, da er zuviel Strom benötigt. Am einfachsten ist es, wenn du einen kleinen Transistorverstärker baust. Diesen kannst du auch verwenden, um mit dem micro:bit andere Geräte ein- und auszuschalten, beispielsweise ein Relais oder eine Weiche deiner Modelleisenbahn.



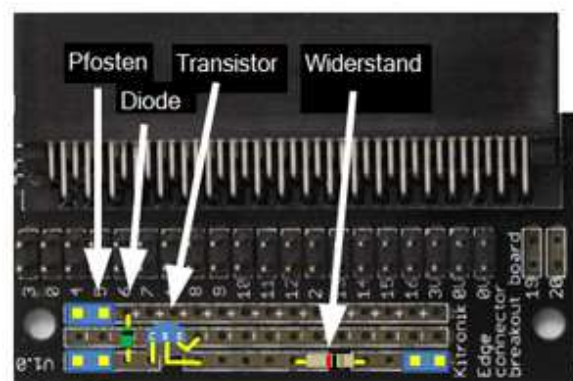
## ■ EINFACHER TRANSISTORVERSTÄRKER

Du benötigst dazu einen Transistor vom Typ 2N2222 oder ähnlich (z.B. BC 547), eine Diode vom Typ 1N4148 oder ähnlich und einen Widerstand von 330 Ohm. Zudem musst du noch etwas Aufbaumaterial haben, beispielsweise eine kleine Leiterplatte. Für den akustischen Alarm musst du einen Piezobuzzer einbauen, der bei einer Spannung von 3V pfeift (also keinen Lautsprecher).

Du lötest die Komponenten gemäß folgendem Schema zusammen:



Aufbau mit einer Lochstreifenplatte



Aufbau mit einem Anschlusstecker  
(Bezugsquelle: Kitronik, Code: 5601B)

## ■ MERKE DIR...

Einfache Alarmanlagen kannst du leicht mit dem micro:bit als eingebettetes System aufbauen, wenn du deine handwerklichen Fähigkeiten mit etwas Elektronik- und Programmierkenntnissen kombinierst.

## ■ ZUM SELBST LÖSEN

1. Baue eine Alarmanlage auf, welche überwacht, ob eine Tür geöffnet wird. Du kannst dabei einen Sicherheitsdraht, den Beschleunigungs- oder den Magnetfeldsensor verwenden.
2. Schliesse an den Transistorverstärker einen gewöhnlichen kleinen Lautsprecher an (ohne Verstärker) und verwende diesen zur akustischen Alarmierung.



## 10. DATENERFASSUNG

---

### ■ DU LERNST HIER...

wie du den micro:bit als Datenerfassungsgerät einsetzen kannst, das Messwerte aufnimmt und speichert (Datenlogger) und wie du nachher die Daten auf einen PC transferierst.

### ■ VERWENDUNG DES ANALOGEN EINGANGS

Um eine physikalische Grösse zu erfassen und digital weiter zu bearbeiten, benötigst du einen Sensor, der den Messwert in eine Spannung umwandelt. Diese kann im Gegensatz zu einem digitalen Signal grundsätzlich beliebige Werte annehmen, deshalb spricht man von einem analogen Signal. Dieses legst du an einen Eingang des micro:bits, der es mit einem Analog-Digital-Wandler (ADC) in eine digitale Zahl im Bereich 0 bis 1023 umwandelt. Verwendest du P0 als Eingang, so liefert dir der Befehl `pin0.read_analog()` den Messwert.

### ■ MUSTERBEISPIELE

Für deine ersten Versuche willst du untersuchen, wie gut deine Finger elektrisch leiten. Du berührst dazu mit dem Zeigefinger den Anschluss 3V und mit dem Mittelfinger den Anschluss P0.

Mit deinem Programm machst du alle 100 ms eine Messung und schreibst die aktuelle Zeit (in 1/10s) und den Messwert aus.



```
from microbit import *

t = 0
while True:
    v = pin0.read_analog()
    print("t = %4.1f v = %d" % (t, v))
    t += 0.1
    sleep(100)
```

**Programmcode markieren** (Ctrl+C kopieren, Ctrl+V einfügen)

Beim Ausschreiben verwendest du eine elegante **Formatierungsangabe**, damit die Werte auch gleich automatisch auf die gewünschte Stellenzahl gerundet werden. Dazu brauchst du im String Platzhalter, beispielsweise für eine Dezimalzahl `%4.1f`, was sagt, dass du ein Feld der Länge 4 mit einer Dezimalzahl (float) ausgeben willst, die auf 1 Kommastelle gerundet ist. `%d` ist ein Platzhalter für eine Ganzzahl. Die Werte selbst folgen dann nach dem String in einer Klammer, die mit `%` eingeleitet wird.

Damit du mit dem micro:bit eine Messserie "*draussen im Feld*" ohne Verwendung eines PCs aufnehmen kannst, schreibst du die Messwerte in eine Datei. Dazu musst du einfach mit `open('data.log')` eine Textdatei öffnen und mit `f.write()` Zeile um Zeile in die Datei schreiben. Die Felder `t` und `v` trennst du hier zweckmässigerweise mit einem Strichpunkt und die Feldbezeichner kannst du weglassen. Dabei darfst du aber das **Zeilenendzeichen** `\n` **nicht vergessen**. Durch Klick auf den Button A startest du die Datenaufnahme, die 10 s dauert.

```

from microbit import *

display.show(Image.SQUARE_SMALL)
while not button_a.was_pressed():
    sleep(10)
display.show(Image.SQUARE)

T = 10
with open('data.log', 'w') as f:
    t = 0
    while t < T:
        v = pin0.read_analog()
        f.write("%4.1f; %d\n" % (t, v))
        t += 0.1
        sleep(100)
display.show(Image.NO)

```

**Programmcode markieren** (Ctrl+C kopieren, Ctrl+V einfügen)

Kommst du vom Ausseneinsatz zurück, so willst du die Daten natürlich auf einen PC transferieren, um sie dort weiter zu bearbeiten. Du kannst dazu das Modul *mbm* von TigerJython verwenden, welches mit der Funktion *extract()* die Datei in das Verzeichnis, indem sich das Programm befindet, kopiert. Da es sich ja um ein Python-Programm handelt, das auf dem PC ausgeführt wird, musst du den grünen Run-Button und nicht etwa den schwarzen Download/Execute-Button klicken.

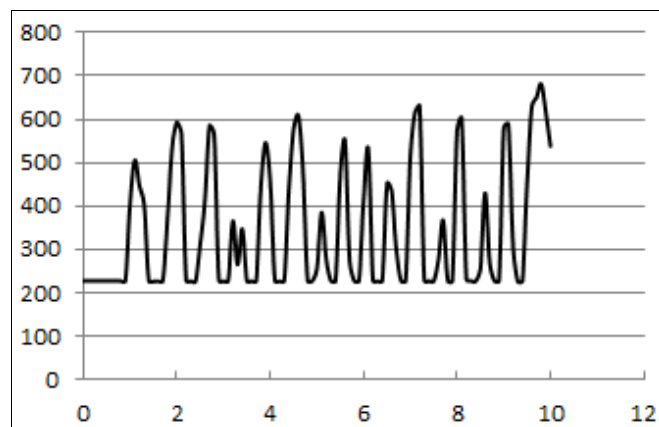
```

from mbm import *

extract('data.log')

```

Verwende irgendeinen Editor, um *data.log* zu öffnen. Du hast sicher Freude, dass alle draussen im Feld aufgenommenen Messwerte auf dem PC angekommen sind. Mit einer Tabellenkalkulation oder einem anderen Grafiktool (z.B. dem GPanel von TigerJython) kannst du sie grafisch darstellen.

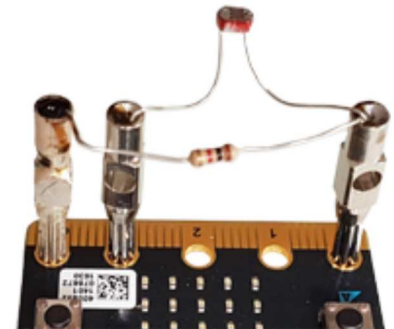


## ■ MERKE DIR...

Ein Datenlogger konvertiert die physikalischen Messwerte in eine digitale Zahl und schreibt diese, meist versehen mit einem Zeitstempel (timestamp), in eine Datei.

## ■ ZUM SELBST LÖSEN

1. Schreibe die Messwerte des Beschleunigungssensors in eine Datei und stelle den Verlauf grafisch dar. Nimm beispielsweise der Betrag der Beschleunigung während 2 Sekunden alle 20 Millisekunden auf. Damit kannst du sogar den freien Fall untersuchen.
2. Du willst die Helligkeit erfassen. Beschaffe dir dazu im Elektronikhandel einen Lichtwiderstand (Light Dependent Resistor, LDR), den du zusammen mit einem Widerstand von 1 kOhm in folgender Schaltung am micro:bit anschließst. Nimm beispielsweise den Helligkeitsverlauf während 10 s alle 0.1 s auf und stelle ihn grafisch dar.



3. Du kannst im Terminalfenster kontrollieren, ob das Schreiben in die Datei gelungen ist. Dazu tippst du beim Python-Prompt ein:  

```
>>> from os import *
```

  
Jetzt stehen dir die Funktionen des Moduls os zur Verfügung. Mit  

```
>>> listdir()
```

  
werden alle Dateien ausgeschrieben und mit  

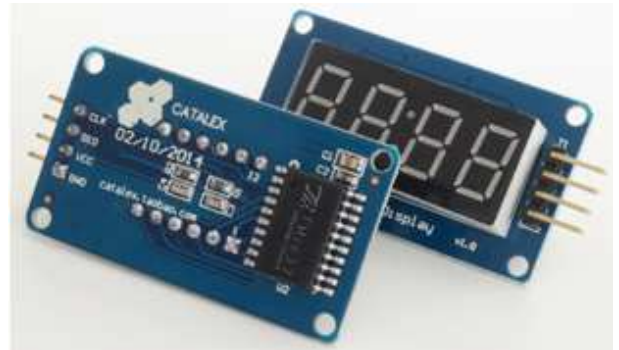
```
>>> size('data.log')
```

  
siehst du die Grösse der Log-Datei.

## ANHANG: 7-SEGMENT DIGITALANZEIGE

### BESCHAFFUNG UND ANSCHLUSS

Die Anzeigeeinheit verwendet den Treiberbaustein TM1637 und eine 4-stellige Siebensegmentanzeige mit einem Doppelpunkt in der Mitte. Zur Ansteuerung sind neben der Stromversorgung mit 3V und GND lediglich 2 Leitungen mit den Bezeichnungen *Clock* (CLK) und *Data* (DIO) nötig.



Es wird ein spezielles Protokoll verwendet (ähnlich, aber nicht gleich wie I<sup>2</sup>C). Es gibt viele Bezugsquellen (Arduino/Raspberry Pi-Lieferanten, Grove, eBay, der Preis schwankt zwischen \$1 und \$10).

Die 4 Leitungen werden mit Kabeln am micro:bit angeschlossen. Für CLK und DIO können irgendwelche Ports P0, P1, P2, P8, P12, P16 verwendet werden, üblich sind aber P0 für CLK und P1 für DIO.

Auf einer Siebensegmentanzeige kann nur ein beschränkter Zeichensatz dargestellt werden.



Zur Ansteuerung der Anzeige wird eine Pythonmodul *mb7seg* verwendet, das den Programmierer von der trickreichen Kommunikation mit der Anzeige isoliert und ihm einige einfache Darstellungsfunktionen zur Verfügung stellt (siehe API Dokumentation). Du kannst *mb7seg.zip* von [hier](#) herunterladen. Es enthält *mb7seg.py* und eine Minimalversion *mb7segmin.py*, die du dann verwenden kannst, wenn du wegen der Programmgrösse den Fehler "memory allocation failed" kriegst. Du musst das eine oder andere Modul dann im Menü von TigerJython unter *Tools* | *Modul* herunterladen auf den micro:bit kopieren.

### TYPISCHE BEISPIELE

#### 1. Von 0 bis 9999 hochzählen

Zuerst wird mit *d = FourDigit(pin0, pin1)* ein Displayobjekt (eine Instanz) erzeugt. Dabei werden die Anschlussports für CLK und DIO angegeben. (Lässt du die Parameter weg, so wird pin0 für CLK und pin1 für DIO angenommen.) Die Methode *show(text)* kann auch direkt Integers anzeigen.

```
from mb7seg import FourDigit
from microbit import *
```

```
d = FourDigit(pin0, pin1)

for n in range(10000):
    d.show(n)
```

**Programmcode markieren** (Ctrl+C kopieren, Ctrl+V einfügen)

Du erkennst, dass für die Anzeige einer einzelnen Zahl ungefähr 30 ms benötigt wird. Mit einer Formatangabe, kannst du die Zahlen rechtsbündig ausschreiben: `d.show("%4d" %n)`.

## 2. Die Beschleunigung anzeigen

Die Digitalanzeige wird vielfach eingesetzt, um Messwerte eines Sensors anzuzeigen, beispielweise die x-Komponente der Beschleunigung. Dazu pollst du in einer Endlosschleife den Sensorwert und schreibst in aus. Mit einem `sleep()` stellst du den Messzyklus ein, musst aber beachten, dass auch die anderen Aufrufe im Schleifenkörper Prozessorzeit benötigen.

```
from mb7seg import FourDigit
from microbit import *

d = FourDigit()
while True:
    acc = accelerometer.get_x()
    d.show(acc)
    sleep(100)
```

**Programmcode markieren** (Ctrl+C kopieren, Ctrl+V einfügen)

Hier wird der Displayinstanz ohne Parameterwerte erstellt und dabei vorausgesetzt, dass sich die Anschlüsse CLK und DIO bei P0 und P1 befinden.

Für Dezimalzahlen wird zuerst mit einer Formatangabe eine vierziffrige Zahl mit 2 Dezimalstellen und Vornullen angefordert (in der Längenangabe wird der Dezimalpunkt mitgezählt). Dann muss der Dezimalpunkt entfernt werden, da dieser ja mit dem Doppelpunkt simuliert wird. Im Programm wird die Beschleunigung in  $\text{m/s}^2$  ausgeschrieben.

```
from mb7seg import FourDigit
from microbit import *

d = FourDigit()
d.setColon(True)
while True:
    acc = accelerometer.get_x() / 100
    v = "%05.2f" %acc
    v1 = v.replace(".", "")
    d.show(v1)
    sleep(100)
```

**Programmcode markieren** (Ctrl+C kopieren, Ctrl+V einfügen)

## 3. Eine Eingabeaufforderung anzeigen

Nach dem Start bleibt das Programm in einer Wiederholschleife hängen, bis der Benutzer den Button A klickt. Dabei wird der Prompt-Text in Laufschrift ausgeschrieben. Es wäre dabei ungünstig, die blockierende Methode `scroll()` zu verwenden, da der Benutzer nach dem Klicken des Buttons warten müsste, bis der ganze Text fertig angezeigt ist. Vielmehr ist es angebracht, mit `toLeft()` den Text zu Scrollen, damit du immer wieder prüfen kannst, ob der Buttonklick erfolgt ist. Um aus den Wiederholschleifen "auszubrechen", verwendest du zweimal `break`.

```

from mb7seg import FourDigit
from microbit import *

d = FourDigit()
d.show("Press A to start")
while True:
    while d.toLeft() > 0 and not button_a.is_pressed():
        sleep(300)
    if button_a.is_pressed():
        break
    d.toStart()
    sleep(500)
d.show("Go")

```

**Programmcode markieren** (Ctrl+C kopieren, Ctrl+V einfügen)

Hier darfst du nicht `button_a.was_pressed()` verwenden, da du damit nicht zweimal hintereinander `True` erhältst, wenn du nur einmal klickst.

#### 4. Die Uhrzeit anzeigen

Mit einem billigen Clock-Module (Real Time Clock, RTC), kannst du aus deinem micro:bit eine präzise laufende digitale Uhr machen. Das Modul verwendet ein Board mit dem DS3231 Chip. Es gibt viele Bezugsquellen (Arduino/Raspberry Pi-Lieferanten, Grove, eBay, der Preis schwankt zwischen \$1 und \$10). Das Modul verwendet das I<sup>2</sup>C-Protokoll mit den 4 Anschlüssen GND, VCC, SCL und SDA.

Du schließt es am micro:bit wie folgt an:

GND ⇒ GND

VCC ⇒ 3V

SCL ⇒ P19

SDA ⇒ P20

Nach dem Einsetzen der Batterie musst du zuerst Datum und Uhrzeit mit einem Programm einstellen. Das Modul arbeitet im BCD-Format (Binary Code Decimal). Du kannst dich im Internet orientieren, was man darunter versteht.



```

from microbit import *

s = 0 # Sekunden
m = 33 # Minuten
h = 15 # Stunden
w = 2 # Wochentag (Sonntag = 1)
dd = 3 # Tag
mm = 11 # Monat
yy = 2018 # Jahr

def dec2bcd(dec):
    tens, units = divmod(dec, 10)
    return (tens << 4) + units

addr = 0x68
t = bytes([s, m, h, w, dd, mm, yy - 2000])
for i in range(0,7):
    i2c.write(addr, bytes([i, dec2bcd(t[i])]))
print("Datetime set to %d-%d-%d %d:%d:%d" % (dd, mm, yy, h, m, s))

```

**Programmcode markieren** (Ctrl+C kopieren, Ctrl+V einfügen)



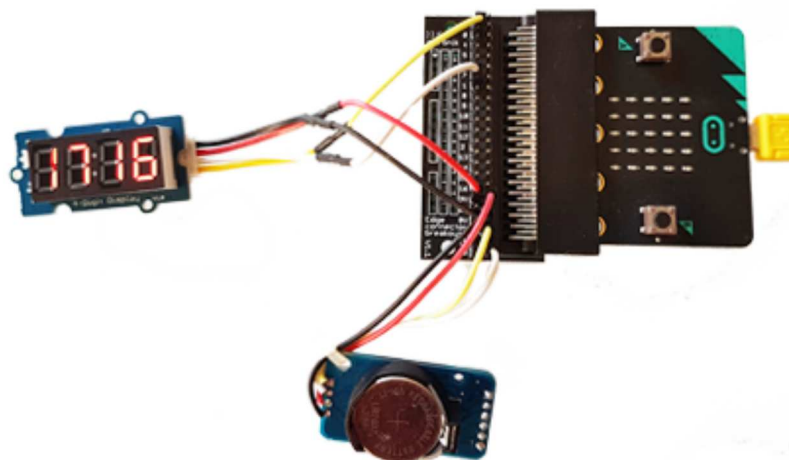
Dein Uhrenprogramm sendet mit `i2c.write()` dem RTC-Modul den Befehl, die Datumszeit zurück zu geben und liest sie mit `buf = i2c.read()` wiederum im BCD-Format in einen Buffer. Nach der Umwandlung ins Dezimalformat stellst du Stunden und Minuten auf der Siebensegmentanzeige dar und lässt noch ungefähr alle Sekunden den Doppelpunkt blinken.

```
from mb7seg import FourDigit
from microbit import *

def bcd2dec(bcd):
    return ((bcd & 0xf0) >> 4) * 10 + (bcd & 0x0f)

d = FourDigit()
showColon = False
while True:
    addr = 0x68
    i2c.write(addr, b'\x00')
    buf = i2c.read(addr, 7)
    mm = bcd2dec(buf[1])
    hh = bcd2dec(buf[2])
    d.show("%02d%02d" % (hh, mm)) # show leading zeros
    if showColon:
        d.setColon(False)
        showColon = False
    else:
        d.setColon(True)
        showColon = True
    sleep(1000)
```

**Programmcode markieren** (Ctrl+C kopieren, Ctrl+V einfügen)



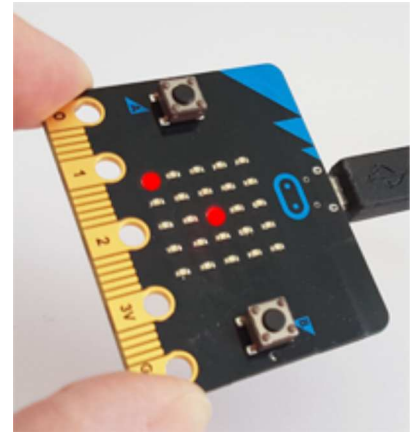
# ARBEITSBLATT: "FANG DAS EI" MIT MICRO:BIT

## ■ SPIELBESCHREIBUNG

Auf dem 5x5 LED-Display des micro:bit bewegt sich ein Pixel auf einer zufälligen Spalte von oben nach unten, analog einem Ei, das von einem Tisch hinunter fällt.

Auf der untersten Zeile kannst du durch Links- und Rechtsneigen des Boards einen Pixel hin und her bewegen, analog einem Korb, mit dem du das Ei auffangen möchtest, bevor es am Boden zerschlägt. Bist du mit dem Korbpixel auf der untersten Zeile dort, wo das Eipixel ankommt, so hast du es "gefangen" und kriegst einen Punkt gutgeschrieben. Das Spiel wird 10 x wiederholt und es geht darum, dass du möglichst viele Eier bzw. Punkte sammelst, also im Maximum 10 von 10.

Nach dem Prinzip von "Teile und Herrsche" teilst das komplexe Problem in mehrere Teilaufgaben ein.



## ■ TEILAUFGABE 1: AUF MITTELLINIE FALLENDEN EIPIXEL

Als erstes schreibst du ein Programm, dass bei jedem Aufruf der Funktion *moveEgg()* das Eipixel auf der fixen Spalte  $x = 2$  um eine Zeile nach unten bewegt. Du gehst von folgendem Programmgerüst aus, und fügt die fehlenden Anweisungen ein:

```
from microbit import *

def move():
    global y, yOld
    # erase old
    # draw new
    # update position
    yOld = y
    y = y + 1
    if y == 5:
        y = 0

# Initialization
x = 2
y = 0
yOld = 4

# Animation loop
while True:
    move()
    sleep(300) # animation speed
```

## ■ TEILAUFGABE 2: AUF ZUFÄLLIGER LINIE FALLENDEN EIPIXEL

Als nächstes erweiterst du das Programm so, dass die Falllinie zufällig gewählt wird. Verwende den Aufruf *random(a, b)* aus dem Modul *random*, der eine Zufallszahl zwischen  $a$  und  $b$  ( $a$  und  $b$  eingeschlossen) liefert. Führe dazu analog zu *yOld* auch ein *xOld* ein.

### ■ TEILAUFGABE 3: VERSCHIEBBARES KORBPIXEL

Schreibe im gleichen Stil die Funktion `moveBasket()`, welche das Korbpixel auf der untersten Zeile hin und her verschiebt, wenn du das Board links oder rechts neigst. `z` ist die aktuelle x-Koordinate des Korbpixels. Das Hauptprogramm besteht aus:

```
# Initialization
z = 0

# Animation loop
while True:
    moveBasket()
    sleep(300) # animation speed
```

### ■ TEILAUFGABE 4: ZUSAMMENFÜGEN VON EI UND KORB

Füge nun die Codeteile so zusammen, so dass sich sowohl das Eipixel wie der Korbpixel (allerdings noch unabhängig voneinander) bewegen. Das Hauptprogramm enthält die Schleife:

```
# animation loop
while True:
    moveEgg()
    moveBasket()
    sleep(300) # animation speed
```

Du kannst jetzt bereits etwas spielen, indem du versuchst, mit dem Korb am richtigen Ort das Ei zu fangen.

### ■ TEILAUFGABE 5: HERAUSFINDEN, OB DER KORB DAS EI FÄNGT

Das Programm soll nun von sich aus herausfinden, ob das Eipixel und das Korbpixel am gleichen Ort zusammentreffen (kollidieren). In diesem Fall erhöhst du die Spielpunktzahl um eins. Den Kollisionstest fügst du am besten im Hauptprogramm ein, wobei du beachten musst, dass die Position des aktuell angezeigten Eipixels mit `xOld`, `yOld` beschrieben wird (`x`, `y` ist die Position des nächsten Pixels). Bei einem Hit zeigst du das Bild `Image.YES` (ein Gutzeichen), bei Misserfolg zeigst du `Image.NO` (ein Kreuz).

```
if yOld == 4: # egg at bottom
    sleep(1000)
    if xOld == z: # hit
        display.show(Image.YES, clear = True)
    else: # miss
        display.show(Image.NO, clear = True)
```

### ■ SPIEL FERTIGSTELLEN UND PERSONALISIEREN

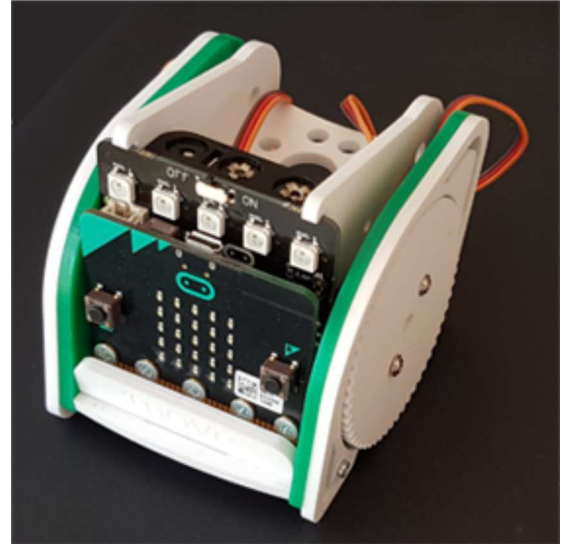
Das Schwierigste ist hinter dir. Du musst jetzt nur noch eine Zählvariable `hits` einfügen, die um eins erhöht wird, falls Ei und Korb zusammentreffen und dann noch dafür sorgen, dass das Spiel genau 10 mal wiederholt wird. Nachher zeigst du auf dem Display die Zahl der Hits an und beendest das Programm. Du kannst es ja durch Klicken des Reset-Buttons beliebig oft starten. Selbstverständlich ist es nun deiner Fantasie überlassen, dem Spiel durch einige Varianten und Verbesserungen einen persönlichen Touch zu geben. Viel Spass!

# ARBEITSBLATT: MOVEmini

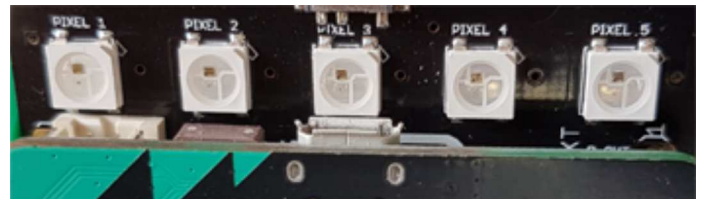
## ■ ZUSAMMENBAU UND FUNKTIONSWEISE

Den Bausatz des MOVEmini kannst du in kurzer Zeit zusammenbauen. Eine Bauanleitung findest du hier Schau dir auch das [YouTube-Video](#) an.

Wie du beim Zusammenbau siehst, erfolgt der Antrieb mit zwei kleinen Servomotoren, wie man sie oft im Modellbau einsetzt. Diese werden an der Zusatzplatine angeschlossen, die mit den Pins P0, P1, P2, 3V und GND des micro:bits verbunden ist. Der rechte Motor wird über P1 und der linke Motor über P2 angesteuert und auf der Zusatzplatine befindet sich ein Stromverstärker für die Servomotoren.



Oben an der Platine siehst du 5 Farb-LEDs, die man NeoPixels nennt. Diese werden über P0 angesteuert.



Orientiere dich im Internet, wie solche Farb-LEDs funktionieren und wo sie überall verbaut werden (z.B. in LED-Strips).

## ■ ERSTE INBETRIEBNAHME

### **NeoPixels**

Es ist immer spannend, ein Gerät das erste Mal in Betrieb zu setzen. Dabei führst du kleine Tests aus, um das richtige Funktionieren einzelner Komponenten zu überprüfen. Beginne mit den NeoPixels. Um diese mit einem Python-Programm anzusteuern, musst du zusätzlich zum Modul *microbit* noch das Modul *neopixel* importieren und du erzeugst dann eine Variable mit dem Namen *np* (der Name ist gleichgültig),

```
from neopixel import *  
np = NeoPixel(pin0, 5)
```

Die Parameterwerte legen fest, dass die Neopixels an P0 angeschlossen sind und dass du 5 Pixels verwendest. Nachher kannst du sehr einfach die RGB-Farbe jedes Pixels einzeln setzen, indem du wie bei einer Liste mit *np[i]* das Pixel auswählst (*i* = 0..4) und ihm in einem Tupel den RGB-Werte (3 Zahlen zwischen 0 und 255) zuweist, also beispielsweise um die grüne LED des Pixel mit *i* = 1 auf den Wert 10 zu stellen:

```
np[1] = (0, 10, 0)
```

Damit die LED mit dieser Farbe leuchtet, musst du nach der Zuweisung immer

```
np.show()
```

aufrufen. Schreibe das Programm und führe es aus.

Wenn du Lust hast, so kannst du schon jetzt oder auch später mit den Pixels noch ein wenig spielen, beispielsweise:

- Alle Pixels in irgendeiner Farbe blinken lassen
- Lauflicht erstellen, d.h. die Pixel werden der Reihe nach ein- und wieder ausgeschaltet
- Farbenspiel: die Pixels leuchten jede Sekunde mit anderen zufälligen Farben

**Beachte:** Wenn deine Pixels nicht leuchten, hast du wahrscheinlich `np.show()` vergessen!

### **Motoren**

Grundsätzlich steuert man Servomotoren mit einem PWM-Signal an. Orientiere dich im *Kapitel 8: Fahrende Roboter* oder im Internet, was man darunter versteht und was der *duty\_cycle* (eine Zahl zwischen 0 und 1 = 100% ist. Allerdings rotieren Servomotoren üblicherweise nicht ständig, sondern nur bis zu einer bestimmten Position, die vom *duty\_cycle* abhängt. Bei deinen Motoren ist das aber anders: der *duty\_cycle* bestimmt hier die Geschwindigkeit des Motors, besser gesagt seine Tourenzahl und ob er in der einen oder anderen Richtung dreht.

Ein PWM-Signal auf Pin1 (rechter Motor) erzeugst du mit

```
pin1.write_analog(duty_cycle)
```

wobei aber hier der *duty\_cycle* nicht in Prozent, sondern als Zahl zwischen 0 und 1023 angegeben wird. Deine Servomotoren laufen für Werte zwischen 73 und hinauf auf 93 immer schneller vorwärts und für Werte zwischen 73 und hinunter auf 63 immer schneller rückwärts.

Erstelle ein Programm, dass den rechten Motor je 1 Sekunde mit dem *duty\_cycle* 63 bis 73 laufen lässt. Am Schluss solltest du den Motor mit `write_analog(0)` stoppen. Schreibe den *duty\_cycle* im Terminalfenster aus.

Mache denselben Test mit dem linken Motor.

## ■ **VORWÄRTS FAHREN UND ABBIEGEN**

Um vorwärts zu fahren, müssen beide Räder gleich schnell in der gleichen Richtung drehen. Das ist nicht exakt realisierbar, aber so ungefähr. Um links zu drehen, stoppst du das linke Rad.

Es ist sinnvoll, ein paar Funktionen zu definieren, zum Beispiel:

[Programmcode markieren](#) (Ctrl+C kopieren, Ctrl+V einfügen)

Erstelle ein Programm so, dass der Roboter vier Mal hintereinander 3 Sekunden lang vorwärts fährt und dann 0.7 s links steuert. Am Ende soll er stoppen.

Ergänze die Funktionssammlung mit *backward()* und *right()* und führe einige weitere Testläufe durch.

## ■ **MIT BELEUCHTUNG UND FAHRRICHTUNGSANZEIGER FAHREN**

Jetzt kannst du deine Kenntnisse kombinieren und lustige Roboteranwendungen programmieren. Ein Vorschlag:

Der Roboter soll nach Programmstart endlos ungefähr ein Quadrat abfahren, bis man den Button A klickt. Dabei sollen bei der Vorwärtsfahrt die drei mittleren Pixels weiss und beim Drehen der entsprechende äusserste Pixel wie Fahrlicht blinker gelb leuchten.

- Verbessere das Programm so, dass der gelbe Fahrlichtblinker tatsächlich blinkt
- Verbessere es, dass das Programm möglichst rasch beendet wird, wenn du den Button klickst
- Verbessere das Programm, dass du es mit Klick auf den Button B immer wieder neu starten kannst
- Verbessere das Programm so, dass es sinnvolle Ausgaben auf dem 5x5 LED Display ausgibt
- Erweitere das Programm so, dass der Roboter Links- und Rechtsdrehungen macht.

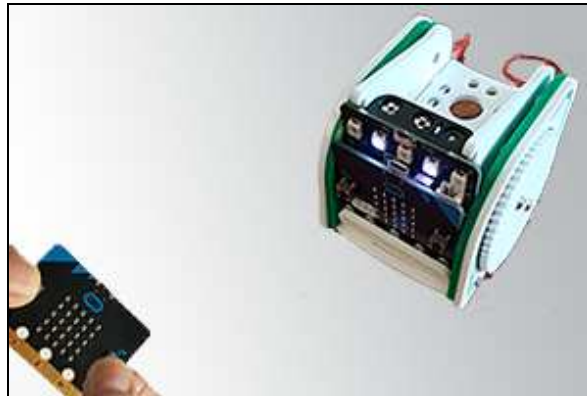


## ■ ROBOTER FERNSTEUERN

Mit einem zweiten micro:bit willst du den Roboter fernsteuern, und zwar wie folgt:

- Kein Button gedrückt → angehalten
- Linker Button gedrückt → links fahren
- Rechter Button gedrückt → rechts fahren
- beide Buttons gedrückt → geradeaus fahren

Orientiere dich im *Kapitel 7. [Bluetooth](#)*, wie du die Kommunikation zwischen der Fernsteuerung und dem Roboter programmierst.



# Dokumentation micro:bit

Modul import: from microbit import \*

## Direkte Funktionsaufrufe

Funktion	Aktion
panic(n)	blockiert das System und zeigt endlos ein "Trauriggesicht" gefolgt von n (für Entwickler)
reset()	startet das System neu (führt main.py aus)
sleep(dt)	hält das Programm während dt Millisekunden an
running_time()	gibt die Zeit in Millisekunden zurück, seit das Board eingeschaltet oder resetted wurde
temperature()	gibt die Temperatur in Grad Celsius zurück (als Float)

## Klasse MicroBitButton

button_a	Objektreferenz (Instanz) des Buttons A
button_b	Objektreferenz (Instanz) des Buttons B
is_pressed()	gibt True zurück, falls der Button beim Aufruf gedrückt ist; andernfalls False
was_pressed()	gibt True zurück, falls der Button seit dem letzten Aufruf (oder dem Start des Programms) gedrückt wurde. Ein erneuter Aufruf gibt False zurück, bis der Button wieder gedrückt wird
get_presses()	gibt die Anzahl Tastenbetätigungen seit dem letzten Aufruf (oder dem Start des Programms) zurück. Ein erneuter Aufruf gibt 0 zurück, bis die Taste wieder betätigt wird

### Beispiel:

```
if button_a.was_pressed():  
    mache_etwas
```

## Klasse MicroBitDisplay

display	Objektreferenz (Instanz)
set_pixel(x, y, value)	setzt die Intensität des Pixels an der Position x, y. value ist im Bereich 0..9
clear()	löscht alle Pixels
show(str)	schreibt str auf dem LED-Display aus. Enthält dieser mehrere Zeichen, so werden diese in Laufschrift angezeigt, die auf dem letzten Zeichen stehen bleibt
show(list_of_img, delay = 400, loop = False, wait = True, clear = False)	zeigt alle Images der Liste nacheinander an. Falls loop = True ist, wird die Anzeigesequenz endlos wiederholt. Für wait = True ist die Methode blockierend, andernfalls kehrt sie zurück und die Anzeige erfolge im Hintergrund. delay ist die Anzeigezeit pro Bild in Millisekunden (default: 400). Für clear = True wird die Anzeige nach dem letzten Bild gelöscht
show(img)	zeigt das img auf dem LED-Display. Ist img grösser als 5x5 pixels, so wird der Bereich x, y = 0..4 angezeigt. Ist img kleiner als 5x5 pixels, sind die fehlenden Pixels ausgeschaltet



scroll(str)	zeigt str als Laufschrift. Das letzte Zeichen verschwindet (blockierende Methode)
scroll(str, delay = 150, loop = False, wait = True, monospace = False)	zeigt str als Laufschrift. Falls loop = True ist, wird die Anzeigesequenz endlos wiederholt. Für wait = True ist die Methode blockierend, andernfalls kehrt sie zurück und die Anzeige erfolge im Hintergrund. delay ist die Anzeigzeit pro Spalte in Millisekunden (default: 150)

#### Beispiele:

```
display.show("A")
display.scroll("Hallo")
display.show([Image.HAPPY, Image.SAD])
```

#### Klasse MicroBitImage

Image(str)	erzeugt eine Objektreferenz (Instanz). str hat das Format "aaaaa:bbbb:cccc:dddd:eeee:", wo a eine Zahl im Bereich 0..9 ist, welche die Intensität des Pixels angibt. a sind die Werte für die erste Zeile, b für die zweite, usw.
Image()	erzeugt eine Objektreferenz (Instanz) mit 5x5 ausgeschalteten Pixels
Image(width, height)	erzeugt eine Objektreferenz (Instanz) mit der gegebenen Zahl horizontaler und vertikaler Pixel, die alle ausgeschaltet sind (value = 0)
set_pixel(x, y, value)	setzt die Intenistät des Pixels an der Position x, y. value ist im Bereich 0..9
shift_left(n)	gibt ein Image-Objekt zurück, das um n Spalten nach links verschoben ist
shift_right(n)	gibt ein Image-Objekt zurück, das um n Spalten nach rechts verschoben ist
shift_up(n)	gibt ein Image-Objekt zurück, das um n Zeilen nach oben verschoben ist
shift_down(n)	gibt ein Image-Objekt zurück, das um n Zeilen nach unten verschoben ist
copy()	gibt einen Klone des Image zurück
invert()	gibt ein Image-Objekt mit invertieren Pixels zurück (new_value = 9 - value)
fill(value)	gibt ein Image-Objekt zurück, bei dem alle Pixel den gegebenen Wert haben (value = 0..9)
dest.blit(img, x, y, w, h, xdest, ydest)	kopiert vom gegebenen img einen rechteckigen Bereich an der Position x, y mit Breite w und Höhe h in das Image dest an der Stelle xdest, ydest

#### Operationen:

image_new = image * n	gibt ein Image-Objekt zurück, bei dem alle Pixel-Intensitäten mit dem Faktor n multipliziert sind
image_new = image1 + image2	gibt ein Image-Objekt zurück, bei dem die Intensitäten der Pixel von image1 und image2 addiert wurden

#### Vordefinierte Objekte:

- Image.HEART
- Image.HEART\_SMALL
- Image.CONFUSED
- Image.ANGRY
- Image.FABULOUS
- Image.MEH

- Image.HAPPY
- Image.SMILE
- Image.SAD
- Image.TRIANGLE\_LEFT
- Image.CHESSBOARD
- Image.DIAMOND
- Image.DIAMOND\_SMALL
- Image.SQUARE
- Image.SQUARE\_SMALL
- Image.RABBIT
- Image.COW
- Image.MUSIC\_CROCHET
- Image.MUSIC\_QUAVER
- Image.ASLEEP
- Image.SURPRISED
- Image.SILLY
- Image.MUSIC\_QUAVERS
- Image.PITCHFORK
- Image.XMAS
- Image.PACMAN
- Image.TARGET
- Image.TSHIRT
- Image.ROLLERSKATE
- Image.DUCK
- Image.HOUSE
- Image.YES
- Image.NO
- Image.TRIANGLE
- Image.TORTOISE
- Image.BUTTERFLY
- Image.STICKFIGURE
- Image.GHOST
- Image.SWORD
- Image.GIRAFFE
- Image.SKULL
- Image.UMBRELLA
- Image.SNAKE
- Image.CLOCK12, Image.CLOCK11, Image.CLOCK10, Image.CLOCK9, Image.CLOCK8, Image.CLOCK7, Image.CLOCK6, Image.CLOCK5, Image.CLOCK4, Image.CLOCK3, Image.CLOCK2, Image.CLOCK1
- Image.ARROW\_N, Image.ARROW\_NE, Image.ARROW\_E, Image.ARROW\_SE, Image.ARROW\_S, Image.ARROW\_SW, Image.ARROW\_W, Image.ARROW\_NW
- Listen: Image.ALL\_CLOCKS, Image.ALL\_ARROWS

### Beimerkung:

Ein MicroBitImage Objekt (kurz ein "Image") ist eine Abstraktion eines realen Pixelbildes und wird erst sichtbar, wenn `display.show(img)` aufgerufen wird. Das Image kann eine beliebige Zahl horizontaler und vertikaler Pixels (w, h) haben, aber es werden mit `show(img)` nur die Pixels im Bereich `x = 0..4`, `y = 0..4` angezeigt. (Ist das Image kleiner, so sind die nicht definierten Pixels dunkel.) Für grössere Images kann `blit()` verwendet werden, um einen Teilbereich auszuschneiden. Beachte, dass ausser `set_pixel()` und `blit()` die Methoden das Image selbst nicht verändern, sondern ein neues Image zurückgeben. Um `img` zu verändern, muss es also neu zugewiesen werden.

### Beispiele:

```
img = Image(2, 2)
img = img.invert()
display.show(img)
```

### Klasse MicroBitTouchPin

pin0, pin1, pin2, pin8, pin12, pin16	Instanzen für allgemeines Digital-in/Digital-out
pin0, pin1, pin2	Instanzen für Analog-in/Analog-out (PWM)
pin3, pin4, pin6, pin7, pin9, pin10	Instanzen vordefiniert für LED display (display mode)
pin5, pin11	Instanzen vordefiniert für Button A, B (button mode)
pin13, pin14, pin15	Instanzen vordefiniert für SPI (spi mode)
pin19, pin20	Instanzen vordefiniert für I2C (i2c mode)
<code>read_digital()</code>	gibt True zurück, falls Pin auf logisch 1 (HIGH) ist; gibt False zurück, falls Pin auf logisch 0 (LOW) ist (Pulldown 10 kOhm)
<code>write_digital(v)</code>	falls <code>v = 1</code> , wird der Pin auf logisch 1 (HIGH) gesetzt; falls <code>v = 0</code> , wird der Pin auf logisch 0 (LOW) gesetzt (max. Strom: 5 mA)
<code>read_analog()</code>	gibt Wert des ADC im Bereich 0..1023 zurück (Eingangsimpedanz: 10 MOhm)
<code>write_analog(v)</code>	setzt den PWM Duty Cycle ( <code>v = 0..1023</code> entsprechend 0..100%) (max. Strom: 5 mA)
<code>set_analog_period(period)</code>	setzt die PWM-Periode in Millisekunden
<code>set_analog_period_microseconds(periode)</code>	setzt die PWM-Periode in Mikrosekunden (> 300)

### Klasse MicroBitAccelerometer

accelerometer	Objektreferenz (Instanz)
get_x(), get_y(), get_z()	gibt die Beschleunigung in x-, y- oder z-Richtung zurück (int, Bereich ca. -2047 bis +2048, entsprechend ungefähr -20 m/s <sup>2</sup> bis +20 m/s <sup>2</sup> , die Erdschleunigung von ungefähr 10 m/s <sup>2</sup> wird mitgezählt). x-Richtung: ButtonA-ButtonB; y-Richtung: Pin2-USB; z-Richtung: Normale zu Board
get_values()	gibt ein Tupel mit den Beschleunigungen in x-, y- oder z-Richtung zurück (Einheit wie oben)

### Klasse MicroBitCompass

compass	Objektreferenz (Instanz)
calibrate()	startet eine blockierende Kalibrierungsroutine, die für genaue Messungen nötig ist. Man muss den micro:bit in verschiedenen Richtungen schief stellen, so dass der blinkende Punkt die Randpixel erreicht und diese anzündet. Erst wenn eine Kreisfigur erstellt ist, fährt das Programm weiter
is_calibrated()	gibt True zurück, falls der Sensor kalibriert wurde
clear_calibration()	setzt den Sensor auf den nicht-kalibrierten Zustand zurück
heading()	gibt den aktuellen Winkel des micro:bit zur Nordrichtung (Grad, int)
get_x(), get_y(), get_z()	gibt den aktuellen Wert der x, y oder z-Komponente des Magnetfeldes an der Stelle des Sensors zurück (int, Mikrottesla, keine Kalibrierung nötig)
get_values()	gibt ein Tupel der x-, y- und z-Komponenten des Magnetfeldes an der Stelle des Sensors zurück (int, Mikrottesla, keine Kalibrierung nötig)

### Klasse NeoPixel

Modul import: from neopixel import \*

np = NeoPixel(pin, n)	erzeugt eine Neopixel Objekt (Instanz) mit n Neopixels, die an den gegebenen Pin angeschlossen sind. Jeder Pixel wird durch seine Position adressiert (beginnend bei 0) und seine Farbe wird durch eine Zuweisung eines RGB-Tupels bestimmt, z.B. np[2] = (0, 100, 255) setzt Pixel # 2 auf Rot = 0, Grün = 100, Blue = 255. show() muss aufgerufen werden, damit die Änderung sichtbar wird. (Strips mit WS2812 LEDs unterstützt.)
clear()	löscht alle Pixels
show()	zeigt die Pixels an. Muss bei jeder Änderung der Farbwerte aufgerufen werden, damit diese sichtbar ist

### Modul music

Modul import: from music import \*

set_tempo(bpm = 120)	setzt die Anzahl Beats pro Minute (default: 120)
pitch(frequency, len, pin = microbit.pin0, wait = True)	spielt einen Ton mit gegebener Frequenz in Hertz während der gegebenen Zeit in Millisekunden. pin definiert den Output-Pin am GPIO-Stecker (default: P0). Falls wait = True, ist die Funktion blockierend; sonst kehrt sie zurück, während der Ton weiter spielt (bis die Abspieldauer erreicht ist oder stop() aufgerufen wird)

<code>play(melody, pin = microbit.pin0, wait = True, loop = False)</code>	spielt eine Melodie mit dem aktuellen Tempo.). <code>pin</code> definiert den Output-Pin am GPIO-Stecker (default: P0). Falls <code>wait = True</code> , ist die Funktion blockierend; sonst kehrt sie zurück, während die Melodie weiter spielt (bis die Abspieldauer erreicht ist oder <code>stop()</code> aufgerufen wird). Falls <code>loop = True</code> , wird die Melodie endlos erneut abgespielt
<code>stop(pin = microbit.pin0)</code>	stoppt alle Sound-Ausgaben am gegebenen GPIO-Pin (default: P0)

**Bemerkungen:** Eine Melodie ist eine Liste mit Strings in folgendem Format: ["note:dauer", "note:dauer",...]

note in musikalischer Notation: c, d, e, f, g, a, h mit optionaler Octavezahl (default: 1): z.B.. c2, d2, ... und optionalem Versetzungszeichen (Halbtonkreuz): c#, d#,... oder c#2, d#2,...

dauer in Anzahl Ticks (optional, default: 1)

#### Vordefinierte Melodien:

- ADADADUM - Eröffnung von Beethoven's 5. Sinfonie in C Moll
- ENTERTAINER - Eröffnungsfragment von Scott Joplin's Ragtime Klassiker "The Entertainer"
- PRELUDE -Eröffnung des ersten Prelude in C Dur von J.S.Bach's 48 Preludien und Fugen
- ODE - "Ode an Joy" Thema aus Beethoven's 9. Sinfonie in D Moll
- NYAN - das Nyan Cat Thema
- RINGTONE - ein Klingelton
- FUNK - ein Geräusch für Geheimagenten
- BLUES - ein Boogie-Woogie Blues
- BIRTHDAY - "Happy Birthday to You..."
- WEDDING - der Chorus des Bräutigams aus Wagner's Oper "Lohengrin"
- FUNERAL - der "Trauerzug", auch bekannt als Frédéric Chopin's Klaviersonate No. 2 in B♭Moll
- PUNCHLINE - a lustiger Tonclip, nachdem ein Witz gemacht wurde
- PYTHON - John Philip Sousa's Marsch "Liberty Bell", ein Thema aus "Monty Python's Flying Circus"
- BADDY - Filmclip aus "The Baddy "
- CHASE - Filmclick aus einer Jagdszene
- BA\_DING - ein Signalton, der darauf hinweist, dass etwas geschehen ist
- WAWAWAWAA - ein trauriger Posaunenklang
- JUMP\_UP - für Spiele, um auf eine Aufwärtsbewegung hinzuweisen
- JUMP\_DOWN - für Spiele, um auf eine Abwärtsbewegung hinzuweisen
- POWER\_UP - ein Fanfarenklang, der darauf hinweist, dass etwas erreicht wurde
- POWER\_DOWN - ein trauriger Fanfarenklang, der darauf hinweist, dass etwas verloren gegangen ist

#### Modul radio:

Modul import: `from radio import *`

Computerkommunikation über Bluetooth

<code>on()</code>	schaltet die Bluetooth-Kommunikation ein. Verbindet mit einem micro:bit mit eingeschaltetem Bluetooth
<code>off()</code>	schaltet die Bluetooth-Kommunikation aus
<code>send(msg)</code>	sendet eine String-Message in den Messagebuffer des Empfängerknotts (First-In-First-Out, FIFO-Buffer)
<code>msg = receive()</code>	gibt die älteste Message (string) des Messagebuffers zurück und entfernt sie aus dem Buffer. Falls der Buffer leer ist, wird None zurückgegeben. Es wird vorausgesetzt, dass die Messages mit <code>send(msg)</code> gesendet wurden, damit sie sich in Strings umwandeln lassen [sonst wird eine ValueError Exception ("received packet is not a string") geworfen]
<code>send_bytes(msg_bytes)</code>	sendet eine Message als Bytes (Klasse <i>bytes</i> , e.g <i>b'\x01\x48'</i> ) in den Messagebuffer des Empfängerknotts (First-In-First-Out, FIFO-Buffer)
<code>receive_bytes()</code>	gibt die älteste Message (bytes) des Messagebuffers zurück und entfernt sie aus dem Buffer. Falls der Buffer leer ist, wird None zurückgegeben. Zum Senden muss <code>send_bytes(msg)</code> verwendet werden (und nicht <code>send(msg)</code> )

### Modul mbutils:

Modul import: from mbutils import \*

mot_rot(mot, speed)	lässt den Motor vorwärts (speed > 0) oder rückwärts (speed < 0) laufen. mot = motL oder motR, speed = 0..100
buggy_setSpeed(speed)	setzt Geschwindigkeit beider Motoren im Bereich 0..100, default: 40
buggy_forward(speed)	lässt Buggy vorwärts laufen, speed = 0..100
buggy_backward(speed)	lässt Buggy rückwärts laufen, speed = 0..100
buggy_stop()	stoppt Buggy
buggy_left()	dreht Buggy nach links (rechter Motor vorwärts, linker Motor rückwärts)
buggy_right()	dreht Buggy nach rechts (linker Motor vorwärts, rechter Motor rückwärts)
buggy_leftArc(reduce)	dreht Buggy auf Linkskreis; reduce = 0..1: Faktor, um den linker Motor langsamer läuft
buggy_rightArc(reduce)	dreht Buggy auf Rechtskreis; reduce = 0..1: Faktor, um den rechter Motor langsamer läuft
isDark(ldr)	True, falls Linesensor ldr dunkel ist (ldr = ldrL oder ldrR)

### Modul mb7seg

Modul import: from mb7seg import FourDigit

(\*) im Modul mb7segmin nicht enthalten

FourDigit()	erzeugt eine Displayinstanz mit Standardwerten: Clock-Anschluss: P0, Data-Anschluss: P1, Helligkeit: 4
FourDigit(clk = pin0, dio = pin1, lum = 4)	erzeugt eine Displayinstanz mit gegebenen Werten für den Clock-Anschluss, Data-Anschluss und die Helligkeit
show(text, pos = 0)	schreibt text mit ersten Zeichen an der gegebenen Position (0: Ziffer ganz links, 3: Ziffer ganz rechts). Der Text kann beliebig lang sein, aber es werden nur 4 Zeichen angezeigt. Der Text wird gespeichert und kann mit toLeft() und toRight() verschoben werden. Integer werden linksbündig dargestellt
setColon(True)	zeigt den Doppelpunkt beim Ausschreiben von Text
(*)erase()	löscht den Text und den Doppelpunkt
(*)toLeft()	verschiebt den Text um eine Stelle nach links und gibt die Anzahl Zeichen zurück, da auf der rechten Seite noch zur Verfügung stehen (0: wenn der ganze Text durchgelaufen ist)
(*)toRight()	verschiebt den Text um eine Stelle nach rechts und gibt die Anzahl Zeichen zurück, die auf der linken Seite noch zur Verfügung stehen (0: wenn der ganze Text durchgelaufen ist)
(*)toStart()	setzt den Text an die Start-Position
(*)scroll(text)	schreibt text mit dem ersten Zeichen an Position 0 aus und scrollt dann den Text nach links, bis alle Zeichen angezeigt wurden (blockierende Methode)
(*)setLuminosity(lum)	setzt die Helligkeit auf den gegebenen Wert (0..7)

## ■ ÜBER DIE AUTOREN

**Jarka Arnold** war als Dozentin an der Pädagogischen Hochschule Bern für die Informatikausbildung angehender Lehrkräfte für die Sekundarstufe 1 tätig. Sie hat dabei Informatikgrundkonzepte und das Programmieren mit Java, PHP und Python vermittelt. Ihre langjährige Erfahrung in der Aus- und Weiterbildung von Informatiklehrpersonen und viele Musterbeispiele sind in diesen Lehrgang eingeflossen. Sie ist zudem verantwortlich für den Webauftritt dieses Lehrgangs.

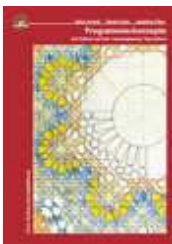
**Aegidius Plüss** war an der Universität Bern Professor für Informatik und deren Didaktik und hat in dieser Tätigkeit viele Informatiklehrkräfte aus- und weitergebildet, die heute aktiv an den Schulen tätig sind. Er gilt als Urgestein in der Informatikausbildung und hat eine grosse Erfahrung mit vielen Programmiersprachen und Computersystemen. Er ist für die textliche Formulierung dieses Lehrgangs und für die didaktischen Libraries in TigerJython verantwortlich.

## ■ KONTAKT

Die Entwicklergruppe von TigerJython4Kids ist dankbar für jede Art von Rückmeldungen, insbesondere für Fehlermeldungen und Richtigstellungen, Anregungen und Kritik. Wir bieten auch Hilfe und Beratung bei fachlichen oder didaktischen Fragen zu Python und den in TigerJython integrierten Libraries, sowie zur Robotik-Hardware.

Schreiben Sie ein Email an:

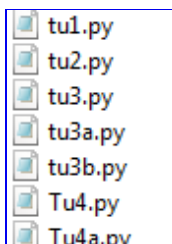
[help@tigerjython.com](mailto:help@tigerjython.com)



## **J. Arnold, T. Kohn, A. Plüss: Programmierkonzepte mit Python**

Umfangreiches Online-Lehrmittel, mit vielen Beispielen aus verschiedenen Gebieten, geeignet für den Einsatz in weiterführenden Schulen und zum Selbststudium.

[www.tigerjython.ch](http://www.tigerjython.ch)



## **Programmbeispiele**

Sourcecodes aller Programmbeispiele aus *TigerJython4Kids*.

<http://www.tigerjython4kids.ch/download/examples.zip>



## **Jarka Arnold: Turtlegrafik, Robotik und Spiele mit Python**

Online-Lernprogramm mit vielen lauffähigen Programmbeispielen und Aufgaben für den Einsatz im Unterricht auf der Sekundarstufe 1 und 2.

<http://www.jython.ch>



## **Tobias Kohn: Python**

Eine Einführung in die Computer-Programmierung.

Ein Skript im PDF-Format.

<http://jython.tobiaskohn.ch/PythonScript.pdf> (166 Seiten)



## **Jarka Arnold, Aegidius Plüss: Python exemplarisch**

Ergänzende Python-Tutorials: Visualisierung, Raspberry PI und Data Mining

<http://www.python-exemplarisch.ch>