Python

Erste Schritte

Einführung in Python

Python ist eine beliebte textbasierte Programmiersprache, die sich hervorragend für Anfänger eignet, weil sie so knapp und verständlich ist. Außerdem bietet sich Python auch für Programmiererinnen und Programmierer an, weil sich diese Programmiersprache nicht nur für die Web- und Softwareentwicklung, sondern auch für wissenschaftliche Anwendungen eignet, beispielsweise für die Datenanalyse und maschinelles Lernen.

Im Abschnitt **Erste Schritte** werden die Grundlagen erläutert, wie sich Python mit LEGO® Education SPIKE™ Prime nutzen lässt. Der Abschnitt enthält Kapitel, in denen du:

Einführung in Python

1. Lernst, wie du den *Code-Editor* in der LEGO® Education SPIKE™ App benutzt, um Python-Programme zu schreiben.

Hello, World!

2. Eine Mitteilung auf der Lichtmatrix des SPIKE Prime Hubs schreibst.

Kommentare in Python

3. Lernst, wie dir Kommentare helfen, Programm-Entwürfe und fertige Programme zu beschreiben.

Steuerung von Motoren

4. *Asynchrone Funktionen* definierst und startest, um Motoren zu steuern.

Variablen

5. Zwei Motoren mit *lokalen* und *globalen* Variablen steuerst.

Die Macht des Zufalls

6. Möglichkeiten entdeckst, witzige Zufallsprogramme zu schreiben, die das Licht am Hub steuern.

Sensorsteuerung

7. Einen Motor mithilfe des Kraftsensors steuerst. Dann erfährst du, wie du die Konsole zum *Debuggen* deines Programms benutzt.

Sensorbedingungen

8. *Logische Ausdrücke* verwendest, um auf verschiedene Bedingungen zu reagieren. Anschließend lernst du, verschiedene Teile deines Programms zusammen auszuführen, um auf mehrere Bedingungen zu reagieren.

Nächste Schritte

9. Vorschläge für zusätzliche Informationsquellen erhältst, um mehr über die Nutzung von Python mit SPIKE Prime zu erfahren.

Python-Syntax

Beim Erlernen einer textbasierten Programmiersprache musst du als Erstes ihre *Syntax* verstehen. Diese Sprachsyntax gibt die Regeln fürs Schreiben von *Anweisungen* (Programmzeilen) vor und definiert, wie *Programmblöcke* kenntlich gemacht werden, die aus mehreren Anweisungen bestehen.

In Python beginnt jede Anweisung mit einer Ebene der *Einrückung* und endet mit einem *Zeilenumbruch*. Als Einrückung wird die Anzahl der Leerzeichen vor einer Anweisung bezeichnet. Programmzeilen mit derselben Anzahl von Leerzeichen besitzen dieselbe *Einrückungsebene* und gehören zum selben Programmblock. Die SPIKE App verwendet 4 Leerzeichen für jede Einrückungsebene.

Du schreibst dein Programm im *Code-Editor*. Dort sind Funktionen vorhanden, die dich dabei unterstützen, das Programm richtig zu schreiben. Wenn du zum Beispiel einen Programmblock anfängst, beispielsweise eine *Funktion* oder eine `if`-Anweisung, dann rückt der Editor die nächste Zeile um 4 Leerzeichen ein. Außerdem wird jede Zeile nummeriert, damit du dich in deinem Programm besser zurechtfindest.

Die *Syntaxhervorhebung* im Code-Editor zeigt *Kommentare*, *Schlüsselwörter*, Text und Zahlen in unterschiedlichen Farben an, damit das Programm leichter zu lesen ist. Im nachstehenden Programm ist der Kommentar in der ersten Zeile grün dargestellt, die Schlüsselwörter `print`, `if` und `True` dagegen blau, der Text `'LEGO'` violett und die Zahl `123` orange.

```
# Das ist ein Kommentar.
print('LEGO')
if True:
    print(123)
```

Die Anweisungen oben sind ein *Beispielprogramm*, das dir in allen Kapiteln des Abschnitts **Erste Schritte** immer wieder begegnen wird. Bei jedem Beispiel befindet sich in der oberen rechten Ecke ein Kopieren-Symbol:

Klicke auf dieses Symbol, um das gesamte Beispielprogramm zu kopieren. Klicke dann mit der rechten Maustaste auf den Code-Editor oder halte den Code-Editor lange gedrückt und wähle anschließend die Option „Einfügen“ aus dem Menü, um das Programm einzufügen. Auf einem Windows-Rechner kannst du auch `CTRL+V` (STRG+V) drücken bzw. `Command+V` auf einem Mac.

SPIKE Prime Module

Um den SPIKE Prime Hub und die Sensoren und Motoren zu steuern, benötigst du die SPIKE Prime *Module*. Module werden verwendet, um zusammengehörige Programmzeilen zu organisieren. Zu jeder SPIKE Prime Komponente gibt es ein Modul. Das Modul `motor` enthält zum Beispiel die Anweisungen zum Steuern der Motoren. Um die Funktionalität eines Moduls zu nutzen, musst du das Modul zunächst *importieren* – und zwar mithilfe der Anweisung `import`:

```
import motor
```

Importiere die Module, die du einmal benötigst, gleich am Anfang deines Python-Programms. Näheres zu den Modulen und ihrer Funktionalität findest du im Abschnitt **SPIKE Prime Module** dieses Benutzerhandbuchs.

MicroPython

Der SPIKE Prime Hub ist ein kleiner Computer, der als Mikrocontroller bezeichnet wird und der über eine begrenzte Speicher- und Rechenleistung verfügt. Da die Vollversion der Programmiersprache Python zu viel Speicher belegen würde, wird auf dem Hub *MicroPython* ausgeführt. Diese stark optimierte Python-Version kann auf Mikrocontrollern ausgeführt werden. Auch die Module zur Steuerung des SPIKE Prime Hubs sowie der Sensoren und Motoren wurden stark optimiert, indem optimierte *Datentypen* verwendet werden.

Wie du gesehen hast, zeigt der Code-Editor Text und Zahlen in unterschiedlichen Farben an, was daran liegt, dass es sich um verschiedene Datentypen handelt. Python unterscheidet darüber hinaus zwischen ganzen

Zahlen und Dezimalzahlen. Ganze Zahlen werden auch als *Ganzzahlen* vom Typ `int` bezeichnet, der in MicroPython optimiert ist. Dezimalzahlen nutzen den nicht optimierten Typ `float`. Deshalb vermeiden SPIKE Prime Module diesen Datentyp. Für dich bedeutet das, dass du nur Ganzzahlen benutzen kannst oder Dezimalzahlen mithilfe unterschiedlicher *Einheiten* beschreibst. Für eine halbe Sekunde solltest du anstelle von 0,5 Sekunden lieber 500 Millisekunden verwenden.

Aufgabe

Kannst du einen Teil des Beispielprogramms in diesem Kapitel kopieren und in den Code-Editor einfügen?

Hello, World!

Beim Erlernen einer neuen Programmiersprache ist es mittlerweile Tradition, ein Begrüßungsprogramm mit dem Text „Hello, World!“ zu schreiben. Du wirst also die Lichtmatrix des SPIKE Prime Hubs den Text „Hello, World“ anzeigen lassen. Vergewissere dich zunächst, dass dein SPIKE Prime Hub eingeschaltet und mit der SPIKE App verbunden ist. Befolge dann diese vier Schritte:

1. Vergewissere dich, dass der Code-Editor leer ist, indem du vorhandene Programmzeilen löschst.
2. Klicke auf das Kopieren-Symbol in der oberen rechten Ecke im Beispiel unten, um das Programm zu kopieren.
3. Klicke mit der rechten Maustaste auf den Code-Editor oder halte den Code-Editor lange gedrückt und wähle dann die Option Einfügen aus dem Menü, um das Programm einzufügen.
4. Drücke die Play-Taste, um das Programm auszuführen.

```
from hub import light_matrix

light_matrix.write('Hello, World!')
```

Jetzt siehst du den Text „Hello, World!“ über die Lichtmatrix laufen.

Schauen wir uns jetzt das Programm Zeile für Zeile an.

Die erste Zeile importiert das Modul `light_matrix` (Lichtmatrix) aus dem Modul `hub`, das die Lichtmatrix des Hubs steuert. Nach dem Importieren eines Moduls kannst du dessen unterschiedliche Funktionen nutzen.

Die letzte Zeile *ruft* die *Funktion* `write()` aus dem Modul `light_matrix` auf, um den Text „Hello, World!“ auf der Lichtmatrix zu schreiben.

Definieren einer Funktion

Im vorherigen Beispiel hast du die Funktion `write()` benutzt. Eine Funktion ist ein Programmblock, der eine Aufgabe ausführt, wenn du ihn aufrufst. Du definierst eine Funktion mit dem Schlüsselwort `def`, gefolgt von dem Funktionsnamen, runden Klammern und einem Doppelpunkt. Der *Hauptteil* der Funktion ist eingerückt und enthält das gesamte Programm, das ausgeführt wird, wenn du diese Funktion aufrufst. Du rufst eine Funktion auf, indem du den Funktionsnamen schreibst und hinter den Funktionsnamen runde Klammern setzt. Vergewissere dich, dass der Funktionsaufruf *nicht eingerückt* ist, damit er nicht als Teil des Funktionstextes gelesen wird.

Das folgende Beispiel definiert die Funktion `hello()`, die „Hello World!“ auf der Lichtmatrix schreibt und die Funktion einmal aufruft. Versuche, das Beispielprogramm auszuführen. Denk daran, zunächst sämtliche vorhandenen Programmzeilen aus dem Code-Editor zu löschen, bevor du dein Programm kopierst, einfügst und ausführst.

```
from hub import light_matrix

def hello():
    light_matrix.write('Hello, World!')

hello()
```

Hinzufügen eines Parameters

Im Beispiel oben hat die Funktion `hello()` keine *Parameter*. Deshalb schreibt sie jedes Mal, wenn du sie aufrufst, den Text „Hello World“ auf der Lichtmatrix. Füge einen Parameternamen in die runden Klammern der Funktionsdefinition ein, um die Funktion dynamischer zu gestalten. Der Programmblock im Hauptteil der Funktion kann diesen Parameter verwenden, um basierend auf dessen Wert etwas anderes zu machen. Ein Beispiel hierfür ist die Funktion `write()`, die einen erforderlichen Parameter enthält: den Text, der auf der Lichtmatrix geschrieben werden soll.

Im Beispiel unten wird der Parameter `name` zur Funktion `hello()` hinzugefügt, die daraufhin auf der Lichtmatrix `'Hello, ' + name + '!'` schreibt. Beachte den `+`-*Operator*, der dich Textsegmente zusammenfügen lässt. Text wird auch als *Zeichenfolge* oder Typ `str` bezeichnet. Diese Zeichenfolge (bzw. dieser

String) wird in einfache (') oder doppelte (") gerade Anführungszeichen gesetzt. Eine Zeichenfolge muss von denselben Anführungszeichen umschlossen werden. Die aktualisierte Funktion `hello()` hat einen erforderlichen Parameter vom Typ `str`, damit du die Zeichenfolge `'World'` als *Argument weitergeben* kannst, wenn du die Funktion aufrufst, damit sie „Hello, World!“ auf der Lichtmatrix schreibt.

Versuche, das Beispielprogramm auszuführen. Vergiss nicht, zunächst sämtliche vorhandenen Programmzeilen aus dem Code-Editor zu löschen, bevor du dein neues Programm kopierst, einfügst und ausführst.

```
from hub import light_matrix

def hello(name):
    light_matrix.write('Hello, ' + name + '!')

hello('World')
```

Jetzt siehst du den Text „Hello, World!“ erneut über die Lichtmatrix laufen.

Aufgabe

Kannst du das Programm so verändern, dass der Hub anstelle der Welt *dich* begrüßt?

Kommentare in Python

Die Nutzung von Programmen ist einfacher, wenn du weißt, was das Programm machen soll. Du kannst die gewünschten Aktionen in Alltagssprache beschreiben, indem du Kommentare hinzufügst. Kommentare sind kein Teil des Programms, das auf dem Hub ausgeführt wird. Deshalb haben sie auch keinen Einfluss auf dessen Funktionalität.

Das Zeichen `#` kennzeichnet den Anfang eines Kommentars. Üblicherweise wird ein Kommentar vor das Programm gesetzt, das dieser Kommentar beschreibt. Kurze Kommentare kannst du jedoch auch hinter eine Programmanweisung setzen.

```
# Das ist ein Kommentar.
from hub import light_matrix
# Das ist ein weiterer Kommentar.
```

Mitunter verhält sich ein Teil deines Programms anders, als du es dir vorstellst. In solchen Fällen bietet es sich an, Teile deines Programms *auszukommentieren*, indem du das Zeichen `#` an den Anfang der Zeile setzt.

Diese Zeilen werden dann zu Kommentaren und werden nicht mehr als Teil deines Programms ausgeführt. Das Auskommentieren von Programmteilen kann dir das *Debuggen* erleichtern. Mit Debuggen ist das Finden und Beheben von Fehlern gemeint. Mehrere Programmzeilen lassen sich rasch auskommentieren, indem du sie auswählst und dann unter Windows `CTRL+/` (STRG+7) drückst – oder auf `Command+/` auf einem Mac. Um mehrere Kommentare wieder in Programmzeilen zu verwandeln, musst du diese Kommentare auswählen und dieselbe Tastenkombination drücken.

```
# Die nächste Zeile ist auskommentiert:
# light_matrix.write('Hello, World!')
```

Du kannst auch Kommentare benutzen, um dein Programm zu beschreiben, bevor du das funktionierende Programm schreibst. Das wird als *Pseudocode* bezeichnet und kann dir helfen, in Alltagssprache zu beschreiben, was dein Programm machen soll. Im folgenden Beispiel werden Kommentare als Pseudocode für eine Animation mit blinzelnden Augen auf der Lichtmatrix verwendet.

```
# Lass auf der Lichtmatrix ein fröhliches Gesicht mit Augen
leuchten.
# Warte ein wenig.
# Lass auf der Lichtmatrix ein lächelndes Gesicht ohne Augen
leuchten.
# Warte ein wenig.
# Lass das erste Bild erneut auf der Lichtmatrix leuchten.
```

Blinzelnde-Augen-Programm

Dieses Programm zeigt ein Gesicht mit blinzelnden Augen auf der Lichtmatrix des Hubs. Kopiere den nachstehenden Code und füge ihn in den Code-Editor ein. Führe das Programm dann aus. Wie immer solltest du zunächst jegliche im Code-Editor vorhandenen Programme löschen, bevor du das neue Programm einfügst.

Wenn du dieses Programm ausführst, wirst du nach einer Sekunde das Smiley-Gesicht blinzeln sehen. Das Programm ruft die Funktion `show_image()` aus dem Modul `hub.light_matrix` auf, um ein Bild auf der Lichtmatrix anzuzeigen. Das Programm verwendet die Funktion `sleep_ms()` aus dem Modul `time`, um zwischen den verschiedenen Bildern eine Verzögerung von einigen Millisekunden einzufügen. In dem Programm beschreibt jeder Kommentar, was die nächste Programmzeile machen soll.

```
import time
```

```
from hub import light_matrix

# Lass auf der Lichtmatrix ein fröhliches Gesicht leuchten.
light_matrix.show_image(light_matrix.IMAGE_HAPPY)

# Warte eine Sekunde.
time.sleep_ms(1000)

# Lass auf der Lichtmatrix ein lächelndes Gesicht leuchten.
light_matrix.show_image(light_matrix.IMAGE_SMILE)

# Warte 0,2 Sekunden.
time.sleep_ms(200)

# Lass auf der Lichtmatrix ein fröhliches Gesicht leuchten.
light_matrix.show_image(light_matrix.IMAGE_HAPPY)
```

„WET“ oder „DRY“?

Es mag verlockend erscheinen, jede Programmzeile zu kommentieren, was jedoch dazu führt, dass du alles doppelt schreibst. Diese Vorgehensweise wird auch als *Write Everything Twice* (WET) bezeichnet. Bei selbsterklärenden Programmen sind diese *WET*-Kommentare für die Leserinnen und Leser jedoch wenig hilfreich. Halte dich lieber ans *DRY*-Prinzip (*Don’t Repeat Yourself*) und wiederhole dich nicht.

Im nachstehenden Beispiel sind die Programmzeilen, die die Augen blinzeln lassen, in die neue Funktion `blink()` eingebettet. Das Programm ruft die Funktion dann dreimal auf, um die Augen dreimal blinzeln zu lassen. Beachte bitte, dass in diesem Fall die Kommentare nur die wesentlichen Teile des Programms beschreiben, damit die Leser leichter nachvollziehen können, was das Programm machen soll.

```
import time

from hub import light_matrix

# Diese Funktion lässt die Augen blinzeln.
def blink():
    light_matrix.show_image(light_matrix.IMAGE_HAPPY)
    time.sleep_ms(1000)
    light_matrix.show_image(light_matrix.IMAGE_SMILE)
    time.sleep_ms(200)
    light_matrix.show_image(light_matrix.IMAGE_HAPPY)

# Lass die Augen drei Mal blinzeln.
blink()
blink()
```

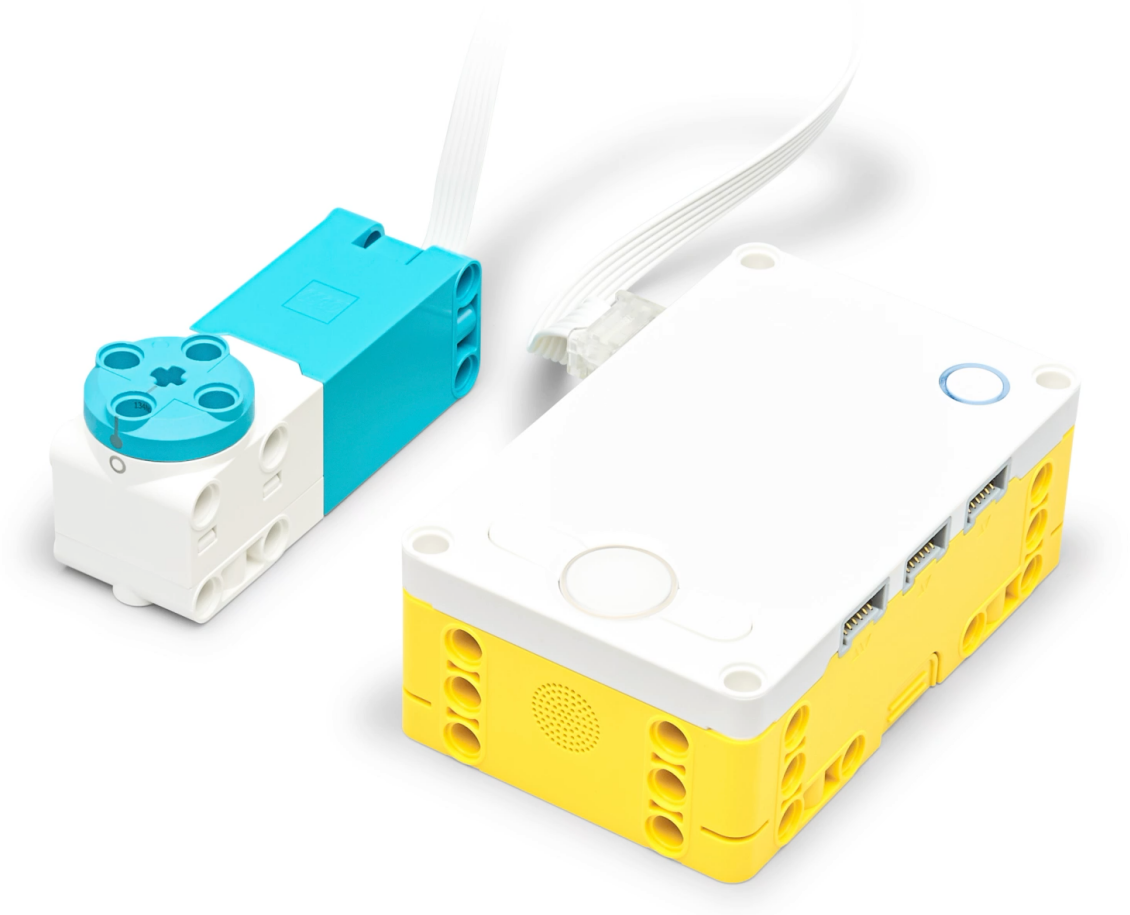
```
blink()
```

Aufgabe

Kannst du das Programm so ändern, dass die Augen bei jedem Blinzeln länger offen bleiben?

Steuerung von Motoren

Du bist jetzt bereit, Motoren anzuschließen und zu benutzen. Schließe einen Motor an Port A (Ein-/Ausgang A) an und probiere das nachstehende Programm aus.



```
import motor
from hub import port

# Lass einen an Port A (Ein-/Ausgang A) angeschlossenen Motor um
360 Grad mit 720 Grad pro Sekunde laufen.
motor.run_for_degrees(port.A, 360, 720)
```

Jetzt solltest du sehen, dass der Motor eine 360-Grad-Drehung (eine volle Umdrehung) mit einer Geschwindigkeit von 720 Grad (2 Umdrehungen) pro Sekunde vollführt.

Jetzt werden wir das Programm Zeile für Zeile betrachten.

Die erste Zeile importiert das Modul `motor`, das die Motoren steuert.

Die zweite Zeile importiert `port` vom Modul `hub`, das den Wert für jeden Port (Ein-/Ausgang) enthält. Du kannst `port.A` für Port A, `port.B` für Port B usw. schreiben, um die gewünschten Ports (Ein-/Ausgänge) anzugeben.

Die letzte Zeile ruft die Funktion `run_for_degrees()` mit drei *Argumenten* auf:

1. Mithilfe des Port-Werts gibt der erste Parameter an, welcher Motor laufen soll.
2. Der zweite Parameter gibt an, um wie viel Grad sich der Motor drehen soll.
3. Der dritte Parameter gibt an, mit welcher Geschwindigkeit der Motor laufen soll, und zwar in Grad pro Sekunde.

Mehrere Motoren

Schließe jetzt einen zweiten Motor an Port B (Ein-/Ausgang B) an, probiere das nachstehende Programm aus.

```
import motor
from hub import port

# Lass zwei an Port A und Port B angeschlossene Motoren um 360
Grad mit 720 Grad pro Sekunde laufen.
# Die Motoren laufen gleichzeitig.
motor.run_for_degrees(port.A, 360, 720)
motor.run_for_degrees(port.B, 360, 720)
```

Du wirst feststellen, dass sich beide Motoren um 360 Grad (eine Umdrehung) mit 720 Grad pro Sekunde drehen und dabei gleichzeitig anlaufen und wieder anhalten. Da sich die beiden Motoranweisungen in unterschiedlichen Zeilen befinden, hätte man erwarten können, dass sie beide nacheinander laufen. Sie laufen jedoch gleichzeitig, weil die Funktion `run_for_degrees()` eine *await*-Funktion ist. Das heißt, du *kannst* darauf warten, bis sie fertig ist, musst es aber nicht. Standardgemäß macht das Programm sofort mit der nächsten Programmzeile weiter, während die await-Anweisungen im Hintergrund ausgeführt werden, bis sie fertig sind. Dadurch ist es möglich, mehrere Befehle gleichzeitig auszuführen.

Run Loop, Async, and Await

Um await-Anweisungen zu nutzen und so flexibel auszulegen, dass Befehle entweder gleichzeitig oder nacheinander ausgeführt werden, musst du dein Programm in einer *asynchronen Funktion * mithilfe einer *run-Schleife* ausführen. Das Modul `runloop` steuert die run-Schleife auf dem Hub und lässt dich asynchrone Funktionen mit seiner Funktion `run()` ausführen. Eine asynchrone Funktion, die auch als *Coroutine* bezeichnet wird, ist ein „Awaitable“, bei dem das Schlüsselwort `async` vor die Funktionsdefinition gesetzt wird. Üblicherweise wird die Coroutine angegeben, die dein Hauptprogramm `main()` enthält. Die nachstehenden Anweisungen zeigen die allgemeine Struktur eines Programms, das eine run-Schleife verwendet.

```
import runloop

async def main():
    # Schreib hier dein Programm.

runloop.run(main())
```

Im Text einer Coroutine kannst du das Schlüsselwort `await` verwenden, bevor du einen await-Befehl aufrufst. Dadurch wird die Coroutine angehalten, bis der Befehl vollständig ausgeführt wurde. Ohne das Schlüsselwort macht das Programm sofort mit der nächsten Programmzeile in der Coroutine weiter. Du kannst auch weiterhin regulären (nicht abwartbaren) Code in der Coroutine verwenden. Dadurch wird jedoch immer das gesamte Programm angehalten oder *blockiert*, bis der Befehl vollständig ausgeführt wurde.

Das nachstehende Programm definiert die Coroutine `main()`, die das Schlüsselwort `await` vor den beiden Funktionsaufrufen `run_for_degrees()` verwendet. Es verwendet die Funktion `run()` aus dem Modul `runloop`, um die Coroutine `main()` in der letzten Programmzeile auszuführen.

```
import motor
import runloop
from hub import port

async def main():
    # Lass zwei an Port A und Port B angeschlossene Motoren um 360
Grad mit 720 Grad pro Sekunde laufen.
    # Die Motoren laufen nacheinander.
    await motor.run_for_degrees(port.A, 360, 720)
    await motor.run_for_degrees(port.B, 360, 720)

runloop.run(main())
```

Probier das Beispielprogramm aus. Du solltest feststellen, dass sich beide Motoren nacheinander um 360 Grad (eine Umdrehung) mit 720 Grad pro Sekunde drehen.

Aufgabe

Kannst du das Programm so ändern, dass beide Motoren nochmal gleichzeitig laufen?

Variablen

Manchmal ertappst du dich dabei, dass du immer wieder dieselbe Zahl schreibst. Im vorigen Kapitel ließen die Befehle die Motoren zum Beispiel jedes Mal mit derselben Gradzahl und derselben Geschwindigkeit laufen. Durch Nutzung von Variablen ist es in solchen Fällen einfacher, mehrere Befehle zu ändern.

Man erstellt eine Variable, indem man den Namen der Variable schreibt, gefolgt von einem einzelnen =-Zeichen und dem Ausgangswert der Variable. Wenn du den Wert einer vorhandenen Variable ändern möchtest, verwendest du exakt dasselbe Format, um ihr einen neuen Wert *zuzuweisen*.

Schließe die Motoren an die Ports (Ein-/Ausgänge) A und B an und probiere das nachstehende Programm aus.

```
import motor
import runloop
from hub import port

async def main():
    # Erstelle eine Variable `velocity` (Geschwindigkeit) mit
einem Wert von 720.
    velocity = 720

    # Lass zwei an Port A und Port B angeschlossene Motoren um 360
Grad drehen.
    # Verwende den Wert der Variablen `velocity` (Geschwindigkeit)
für die Motorgeschwindigkeit.
    await motor.run_for_degrees(port.A, 360, velocity)
    await motor.run_for_degrees(port.B, 360, velocity)

runloop.run(main())
```

Genau wie im vorherigen Kapitel wirst du feststellen, dass beide Motoren nacheinander eine Drehung um 360 Grad (eine Umdrehung) mit 720 Grad

pro Sekunde vollführen. Bei dem Beispiel hier wird eine Variable `velocity` (Geschwindigkeit) erstellt und in den Aufrufen der Funktion `run_for_degrees()` verwendet. Da wir eine Variable verwendet haben, lässt sich die Motorgeschwindigkeit leicht in allen Motorbefehlen ändern. Versuche, den Wert der Variable `velocity` (Geschwindigkeit) zu ändern, und führe das Programm erneut aus.

Variablenbereich

Man sollte unbedingt verstehen, dass es darauf ankommt, *wo* eine Variable erstellt wird. Wenn man eine Variable innerhalb einer Funktion erstellt, ist sie nur für diese Funktion verfügbar. Dann bezeichnet man sie als *lokale* Variable. Wenn du eine Variable für mehrere verschiedene Funktionen in deinem Programm verwenden möchtest, musst du diese Variable außerhalb der Funktionen erstellen, beispielsweise unter deinen `import`-Anweisungen. Dann bezeichnet man sie als *globale* Variable.

```
import motor
import runloop
from hub import port

# Erstelle eine globale Variable `velocity` (Geschwindigkeit) mit
dem Wert 720.
velocity = 720

async def main():
    # Erstelle eine lokale Variable `degrees` (Grad) mit dem Wert
360.
    degrees = 360

    # Lass zwei an Port A und Port B angeschlossene Motoren
laufen.
    # Verwende den Wert der Variablen `degrees` (Grad) für die
Gradzahl.
    # Verwende den Wert der Variablen `velocity` (Geschwindigkeit)
für die Motorgeschwindigkeit.
    await motor.run_for_degrees(port.A, degrees, velocity)
    await motor.run_for_degrees(port.B, degrees, velocity)

runloop.run(main())
```

Du wirst erneut feststellen, dass beide Motoren nacheinander eine Drehung um 360 Grad (eine Umdrehung) mit 720 Grad pro Sekunde vollführen. Dieses Mal hat die Variable `velocity` (Geschwindigkeit) einen globalen *Bereich*, und eine neue Variable `degrees` (Grad) hat einen lokalen Bereich. Man kann die Variable `velocity` (Geschwindigkeit) sowohl innerhalb als auch außerhalb

der Funktion `main()` verwenden. Die lokale Variable `degrees` (Grad) kann man jedoch nur innerhalb der Funktion `main()` verwenden, wo sie definiert wird.

Es kann verlockend sein, all deine Variablen ganz oben in deinem Programm zu definieren, damit sie im globalen Bereich gültig sind, denn dann kannst du sie praktischerweise in deinem ganzen Programm verwenden. Das hat jedoch auch zur Folge, dass der Wert dieser Variablen an *jeder Stelle * deines Programms geändert werden kann, was unerwünschte „Nebenwirkungen“ nach sich ziehen kann. Du solltest deshalb für deine Variablen einen *engen Bereich* definieren, damit nur die Teile deines Programms Zugriff auf die Variablen haben, die diese Werte verwenden und ändern müssen.

Variablen in Schleifen

In Python ist es am einfachsten, Programmzeilen mehrmals zu wiederholen, indem man eine `for`-Schleife mit der integrierten Funktion `range()` verwendet. Um zum Beispiel etwas vier Mal zu wiederholen, schreibt man `for i in range(4):` gefolgt von den Programmzeilen, die vier Mal ausgeführt werden sollen. `range(4)` kann man sich auch als *Tupel* `(0, 1, 2, 3)` vorstellen. Tupel und *Listen* wie `[1, 2, 3]` sind *Iterablen*. Die `for`-Schleife nimmt eine Iterable und *und durchläuft* deren Werte, bis sie das Ende erreicht.

Wenn eine `for`-Schleife ein Tupel oder eine Liste *iteriert*, ändert sie den Wert der lokalen Variable bei jeder Iteration. Bisher hast du Variablen erstellt und ihnen mithilfe des `=`-Zeichens einen Wert zugewiesen. In einer `for`-Schleife wird der Name der lokalen Variable hinter dem Schlüsselwort `for` definiert, in diesem Fall `i`. Jedes Mal, wenn die Schleife ausgeführt wird, ändert sich die lokale Variable `i`. Bei `0` wird die Schleife zum ersten Mal ausgeführt und bei `3` das letzte Mal, gemäß den Werten in `(0, 1, 2, 3)`.

Im nächsten Beispiel wird eine `for`-Schleife verwendet, um die globale Variable `velocity` (Geschwindigkeit) vier Mal zu ändern, um den an Port A (Ein-/Ausgang A) angeschlossenen Motor jedes Mal mit einer anderen Geschwindigkeit laufen zu lassen. Um das Ändern der globalen Variable `velocity` (Geschwindigkeit) im *lokalen Kontext* der Funktion `main()` zu ermöglichen, muss man das Schlüsselwort `global` vor `velocity` (Geschwindigkeit) am Anfang des Funktionstextes setzen.

```
import motor
import runloop
```

```
from hub import port

# Erstelle eine globale Variable `velocity` (Geschwindigkeit) mit
dem Wert 450.
velocity = 450

async def main():
    # Verwende das Schlüsselwort `global`, um hier die `velocity`
(Geschwindigkeit) ändern zu können.
    global velocity

    # Erstelle eine lokale Variable `degrees` (Grad) mit dem Wert
360.
    degrees = 360

    # Die `for`-Schleife erstellt eine lokale Variable `i` und
wiederholt sich 4 Mal.
    # Die Werte der Variablen `i` sind 0, 1, 2 und 3.
    for i in range(4):
        # Ändere die globale Variable `velocity`
(Geschwindigkeit), indem du jedes Mal `i`*90 hinzufügst.
        # Die Werte der Variablen `velocity` (Geschwindigkeit)
sind 450, 540, 720 und 990.
        velocity = velocity + i*90
        await motor.run_for_degrees(port.A, degrees, velocity)

    # Der Wert der Variablen `velocity` (Geschwindigkeit)
außerhalb der `for`-Schleife ist 990.
    await motor.run_for_degrees(port.B, degrees, velocity)

runloop.run(main())
```

Führe das Beispielprogramm aus. Du wirst feststellen, dass sich der an Port A (Ein-/Ausgang A) angeschlossene Motor vier Mal um 360 Grad dreht, und zwar jedes Mal mit einer höheren Geschwindigkeit. Beim letzten Mal dreht sich der an Port B (Ein-/Ausgang B) angeschlossene Motor einmal um 360 Grad mit 990 Grad pro Sekunde.

Aufgabe

Kannst du das Programm so ändern, dass der an Port A (Ein-/Ausgang A) angeschlossene Motor jedes Mal unterschiedlich viele Grad läuft?

Die Macht des Zufalls

Manchmal ist ein unvorhersehbares Programm das beste Programm. Wenn man nicht weiß, was ein Programm als Nächstes macht, wirkt es viel

lebendiger. Um diesen Effekt zu erzielen, musst du Zufälligkeit ins Programm einbauen.

Das nachstehende Programm lässt das Licht des Ein-/Aus-Schalters am Power SPIKE Prime Hub in zehn verschiedenen Farben leuchten, wobei der Wechsel zwischen den einzelnen Farben mit einer zufälligen Verzögerung erfolgt.

```
import random
import time
from hub import light

for color in range(11):
    # Lass das Licht in der aktuellen Farbe leuchten.
    light.color(light.POWER, color)

    # Lass das Licht 0,5 bis 1,5 Sekunden lang eingeschaltet.
    sleep_time = random.randint(500, 1500)
    time.sleep_ms(sleep_time)
```

Jede Farbe wird durch eine andere Zahl dargestellt. Die `for`-Schleife durchläuft `range(11)` und weist den Wert der Variable `color` zu. Dieser Wert wird `0` (schwarz) sein und das Licht ausschalten, wenn die Schleife das erste Mal durchläuft, und nach dem letzten Durchlauf wird der Wert `10` (weiß) zugewiesen. Dir wird auffallen, dass dieses Programm das Modul `random` importiert. Dieses Modul enthält mehrere Funktionen, die Zufälligkeit zu dem Programm hinzufügen.

Bei diesem Beispiel wird die Funktion `randint()` mit dem `start`-Wert 500 und dem `stop`-Wert 1500 verwendet. Diese Argumente bewirken in der Funktion die *Rückgabe* einer Zahl zwischen 500 and 1500, um die Länge des Ruhezustands variieren zu lassen. Die Farben werden jedoch immer in derselben Reihenfolge aufleuchten, auch wenn du das Programm mehrmals ausführst. Glücklicherweise enthält das Modul `random` noch andere Funktionen, um dem Programm noch mehr Zufälligkeit hinzuzufügen.

Endlosschleife

Du kannst auch eine `while`-Schleife verwenden, um etwas endlos zu wiederholen, anstatt eine bestimmte Zahl an Wiederholungen festzulegen. In Python lässt sich eine solche Schleife am einfachsten erstellen, indem man `while True:` schreibt und direkt dahinter die Programmzeile einfügt, die endlos ausgeführt werden soll. Beim nächsten Beispiel wird eine `while True`-Schleife verwendet, um eine kleine Disco-Show endlos laufen zu lassen. Oder bis du das Programm beendest.

```
import random
import time
from hub import light

while True:
    # Erzeuge eine Zufallszahl zwischen 1 und 9.
    random_color = random.randint(1, 9)

    # Lass das Licht in einer zufälligen Farbe leuchten.
    light.color(light.POWER, random_color)

    # Lass das Licht 0,5 bis 1,5 Sekunden lang eingeschaltet.
    sleep_time = random.randint(500, 1500)
    time.sleep_ms(sleep_time)
```

Du wirst feststellen, dass die Ein-/Aus-Taste in zufälligen Farben und mit einer zufälligen Verzögerung bei jedem Farbwechsel aufleuchtet. Bei diesem Beispiel wird wieder die Funktion `randint()` verwendet, um eine Zahl zwischen 1 und 9 (einschließlich 1 und 9) zu generieren, die den verschiedenen Farbcodes entspricht, allerdings ohne Schwarz (`0`) und Weiß (`10`).

Listen und Konstanten

Wenn du nur bestimmte Farben für die Lichtershow benutzen möchtest, kannst du diese Farben in eine *Liste* einfügen und dann eine Zufallsfarbe aus dieser Liste auswählen. Eine Liste erstellt man wie eine Variable: Zuerst schreibst du den Listennamen und dann das `=`-Zeichen. Schließlich setzt du die Werte in eckige Klammern und trennst sie mit Kommas. Eine einfache List mit mindestens zwei *Elementen* schreibt man `my_list = [1, 2]`. Du kannst beliebig viele Werte in die Liste eintragen.

Wie die vorigen Beispiele gezeigt haben, wird jede Farbe durch eine andere Zahl dargestellt. Du benutzt diese Zahl, um das Licht in dieser Farbe leuchten zu lassen. Die Zahl `9` lässt das Licht zum Beispiel rot leuchten. Wenn du jedoch Zahlen zur Darstellung von Farben benutzt, können andere Leserinnen und Leser vermutlich nur schwer nachvollziehen, was dein Programm macht. Du könntest Kommentare hinzufügen, um jeden Wert zu beschreiben. Besser wäre es jedoch, Variablen für jede Farbe zu erstellen. Das Modul `color` enthält eine Variable `RED`, damit du `color.RED` anstelle der Zahl `9` in dein Programm schreiben kannst. (Eine Variable, die in Großbuchstaben aufgelistet ist, ist eine *Konstante*, die du nicht verändern solltest.)

Im nachstehenden Beispiel wird das Modul `color` importiert. Außerdem werden einige Farbkonstanten verwendet, um die Liste `colors` zu erstellen. Dieses Mal bestimmt die Funktion `randint()`, wie oft die `for`-Schleife ausgeführt wird. Außerdem lässt das Beispielprogramm am Ende der zufälligen Lichtershow das Licht weiß leuchten.

```
import random
import time
import color
from hub import light

# Erstelle eine Liste mit unterschiedlichen Lichtfarben.
colors = [color.RED, color.GREEN, color.BLUE, color.YELLOW]

# Lass das Licht fünf bis zehn Mal wechseln.
times = random.randint(5, 10)

for i in range(times):
    # Wähle eine zufällige Farbe aus der Farbenliste aus.
    random_color = random.choice(colors)

    # Lass das Licht in einer zufälligen Farbe leuchten.
    light.color(light.POWER, random_color)

    # Lass das Licht 0,5 bis 1,5 Sekunden lang eingeschaltet.
    sleep_time = random.randint(500, 1500)
    time.sleep_ms(sleep_time)

# Lass das Licht weiß leuchten.
light.color(light.POWER, color.WHITE)
```

Du wirst feststellen, dass das Licht des Ein-/Aus-Schalters zufällig oft zu einer zufälligen Farbe aus der Liste wechselt, und zwar mit einer zufälligen Verzögerung zwischen jedem Farbwechsel. In dem Beispiel wird die Funktion `choice()` verwendet, um eine zufällige Farbe aus der Liste `colors` auszuwählen.

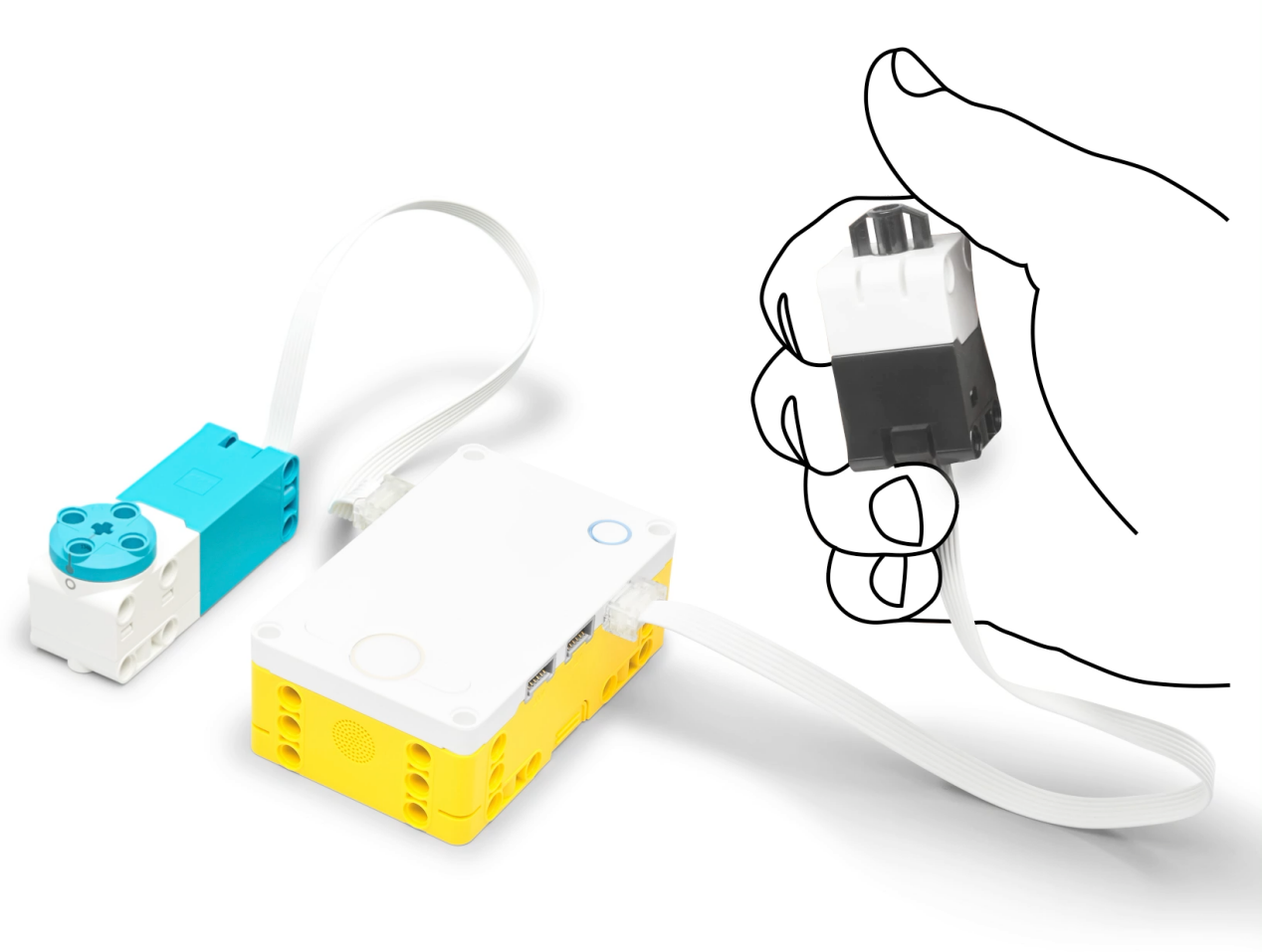
Aufgabe

Kann du das Programm so ändern, dass die Liste `colors` andere Farben enthält?

Sensorsteuerung

In den vorherigen Kapiteln hast du probiert, die Motoren und das Licht mithilfe von Variablen und Zufallszahlen zu steuern. Jetzt wirst du einen Sensorwert benutzen, um einen Motor zu steuern.

Schließe einen Motor an Port A (Ein-/Ausgang A)und einen Kraftsensor an Port B (Ein-/Ausgang B) an und probiere das nachstehende Programm aus.



```
import force_sensor
import motor
from hub import port

# Speichere die Kraft des Kraftsensors in einer Variablen.
force = force_sensor.force(port.B)

# Drucke die Variable auf der Konsole.
print(force)

# Lass den Motor laufen und verwende die Variable, um die
Geschwindigkeit einzustellen.
motor.run(port.A, force)
```

Drücke den Kraftsensor, während das Programm ausgeführt wird. Da ist jetzt nicht wirklich viel passiert, oder? Zum Glück wird in diesem Beispiel die

Funktion `print()` verwendet, um die Variable `force` auf der Konsole zu schreiben, damit du leicht erkennen kannst, was da schiefgegangen ist.

Die Konsole

Manchmal macht dein Programm nicht das, was du von ihm erwartest. Wenn das geschieht, kannst du die Funktion `print()` benutzen, um dein Programm zu *debuggen*. Die Funktion `print()` schreibt, was auch immer du als Argument weitergibst, in das Fenster der Konsole unter dem Code-Editor, in diesem Fall die Kraft des Kraftsensors. Führe das Programm erneut aus und achte auf den Wert, der in der Konsole angezeigt wird.

Du wirst eine einzelne Zahl auf der Konsole sehen, und sofern du beim Starten des Programms nicht den Kraftsensor gedrückt hast, handelt es sich bei dieser Zahl um die `0`. Wenn du einen Motor mit einer Geschwindigkeit von 0 Grad pro Sekunde laufen lässt, wird sich nicht viel rühren. Allerdings lässt sich dieser Fehler im Programm nicht so leicht feststellen, weil das Programm den Sensorwert nur ein einziges Mal beim Start des Programms prüft. Um die Motorgeschwindigkeit basierend auf der Kraft zu aktualisieren, solange das Programm ausgeführt wird, musst du wieder die `while True`-Schleife verwenden.

Die Konsole zeigt auch Fehlermeldungen an, wenn beim Ausführen deines Programms etwas schiefläuft. Wenn du ein Programm ausführst, das einen Motor steuert und die Werte eines Sensors liest, tritt häufig der Fehler auf, dass der fragliche Motor oder Sensor nicht angeschlossen ist. Trenne den Kraftsensor vom Hub und führe dasselbe Programm noch ein letztes Mal aus. In der Konsole wird ein Fehler angezeigt, der dir mitteilt, dass ein Problem aufgetreten ist, um welches Problem es sich handelt und in welcher Programmzeile der Fehler aufgetreten ist.

Fehlerbehebung

Die Konsole hat dir geholfen, zwei Fehler zu finden. Schließe den Kraftsensor wieder an Port B (Ein-/Ausgang B) an, um den zweiten Fehler zu beheben, und führe dann das nachstehende Programm aus, das den ersten Fehler behebt, indem es den Code in einer `while True`-Schleife *umschließt*.

```
import force_sensor
import motor
from hub import port
```

```
while True:
    # Speichere die Kraft des Kraftsensors in einer Variablen.
    force = force_sensor.force(port.B)

    # Drucke die Variable auf der Konsole.
    print(force)

    # Lass den Motor laufen und verwende die Variable, um die
Geschwindigkeit einzustellen.
    motor.run(port.A, force)
```

Drücke den Kraftsensor, während das Programm ausgeführt wird. Du wirst feststellen, dass der Motor beschleunigt oder langsamer wird, je nachdem, wie fest du den Kraftsensor drückst. Außerdem wirst du sehen, dass auf der Konsole jede Menge Variablenwerte geschrieben werden. Die Kraft des Kraftsensors wird in Dezinewton (dN) gemessen. Und da der Sensor maximal eine Kraft von 10 Newton messen kann, beträgt der Höchstwert 100 dN. Wenn ein Motor mit 100 Grad pro Sekunde läuft, ist das immer noch ziemlich langsam!

Rückgabewerte einer Funktion

Anstatt den Wert des Kraftsensors als Variable zu speichern, kannst du eine Funktion definieren, die diesen Wert *zurückgibt*. Wenn du die verschiedenen Teile deines Programms auf diese Weise voneinander trennst, ist es viel leichter, deinen Code zu strukturieren und etwaige Fehler zu beheben.

Das nächste Programm definiert die Funktion `motor_velocity()`, die die gewünschte Motorgeschwindigkeit basierend auf der Kraft des Kraftsensors zurückgibt, anstatt eine Variable zu verwenden.

```
import force_sensor
import motor
from hub import port

# Diese Funktion gibt die gewünschte Motorgeschwindigkeit zurück.
def motor_velocity():
    # Die Geschwindigkeit entspricht der fünffachen Kraft des
Kraftsensors.
    return force_sensor.force(port.B) * 5

while True:
    # Lass den Motor wie zuvor laufen.
    # Verwende den Rückgabewert der Funktion `motor_velocity()`
für die Geschwindigkeit.
    motor.run(port.A, motor_velocity())
```

Drücke den Kraftsensor, während das Programm ausgeführt wird. Du wirst feststellen, dass der Motor beschleunigt oder langsamer wird, je nachdem, wie fest du den Kraftsensor drückst. Die Funktion `motor_velocity()` multipliziert den Kraftwert mit 5, sodass die Geschwindigkeit zwischen 0 und 500 Grad pro Sekunde beträgt.

Aufgabe

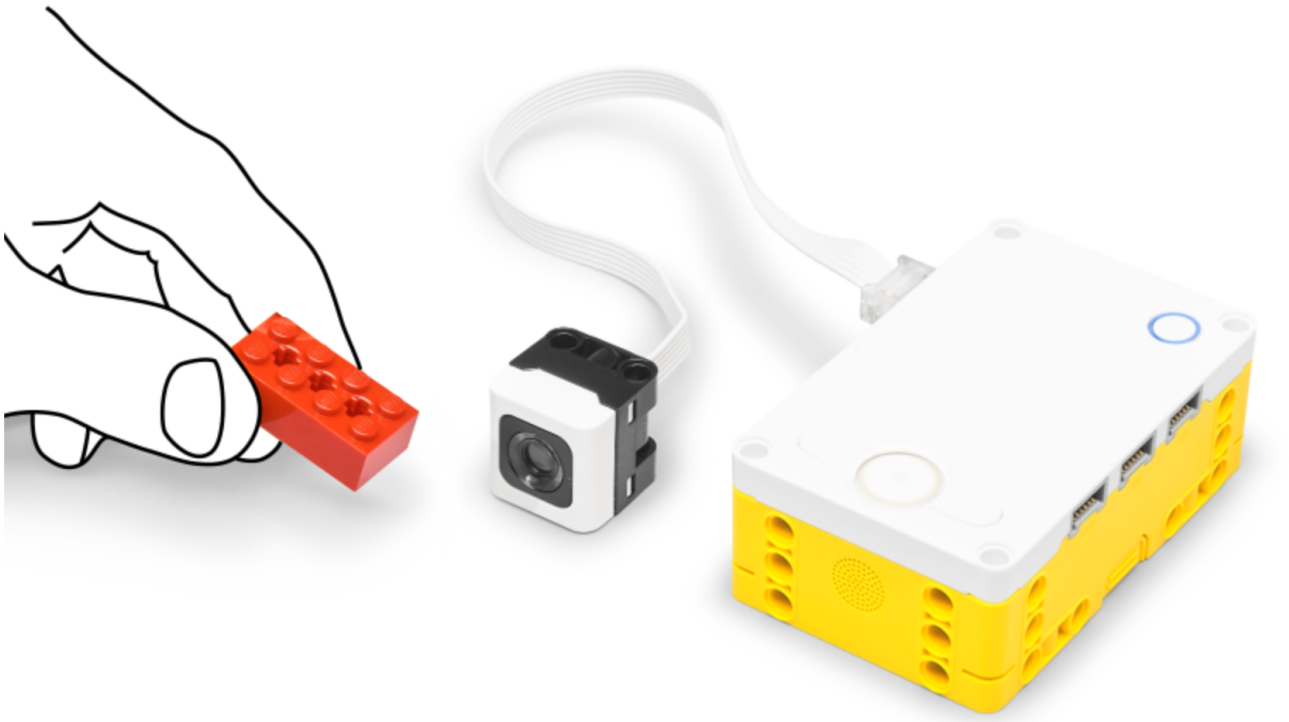
Kannst du das Programm so ändern, dass der Motor mit einer Geschwindigkeit von 1000 Grad pro Sekunde läuft, wenn der Kraftsensor voll gedrückt wird?

Sensorbedingungen

Du hast den Sensorwert verwendet, um den Motor direkt zu steuern. Man kann den *Ablauf* des Programms jedoch auch mithilfe von *Sensorbedingungen* und unter Verwendung einer `if`-Anweisung verändern. Die `if`-Anweisung ist ein wesentlicher Bestandteil der Programmierung und die einfachste Möglichkeit, den Ablauf deines Programms zu steuern.

Man erstellt eine `if`-Anweisung, indem man hinter `if` einen logischen Ausdruck und einen Doppelpunkt setzt. Logische Ausdrücke sind im Grunde genommen Ja- oder Nein-Fragen, zum Beispiel: „Ist die Farbe rot?“ oder „Ist die Taste gedrückt?“ Wenn die Frage mit „ja“ beantwortet wird, dann wird dieser Ausdruck als `True` (wahr) bewertet, ansonsten gilt der Ausdruck als `False` (falsch). Dies sind die beiden *Booleschen* Werte vom Typ `bool`, die in Python verwendet werden. Alle Programmzeilen derselben Einrückungsebene nach der `if`-Anweisung sind Teil des Programmblocks, der ausgeführt wird, wenn der Ausdruck `True` (wahr) ist.

Schließe zum Beispiel einen Farbsensor an Port A (Ein-/Ausgang A) an und führe das nachstehende Programm aus.



```
from hub import port, sound
import color
import color_sensor
import runloop

async def main():
    while True:
        # Prüfe, ob die Farbe Rot erkannt wird.
        if color_sensor.color(port.A) == color.RED:
            # Wenn Rot erkannt wird, ertönt ein sehr langer
Signalton.
            sound.beep(440, 1000000, 100)
            # Halte das Programm an, wenn Rot erkannt wird.
            while color_sensor.color(port.A) == color.RED:
                await runloop.sleep_ms(1)
            # Lass den Signalton verstummen, wenn Rot nicht mehr
erkannt wird.
            sound.stop()

runloop.run(main())
```

Schwenke einen roten LEGO® Stein vor dem Farbsensor, während das Programm ausgeführt wird. Du hörst einen Signalton, wenn die rote Farbe erkannt wird. Dieser Signalton verstummt, wenn die rote Farbe nicht mehr

erkannt wird. Bei diesem Beispiel wird eine `if`-Anweisung verwendet, um zu prüfen, ob Rot die vom Farbsensor erkannte Farbe ist. Hierzu wird der *Gleichheitsoperator* `==` verwendet, wobei der Farbwert des Farbsensors auf der linken Seite und die Konstante `color.RED` auf der rechten Seite steht. (Beachte bitte, dass der Gleichheitsoperator aus zwei `=`-Zeichen besteht. Für die Zuweisung von Werten zu Variablen hast du dagegen nur ein einzelnes `=`-Zeichen benutzt.) Wenn der Farbwert des Farbsensors mit der Konstante `color.RED` identisch ist, dann ist die Bedingung `True` (wahr) und der Programmblock hinter der `if`-Anweisung wird ausgeführt.

Es ist wichtig, den Programmblock in einer `while True`-Schleife zu umschließen. Andernfalls prüft der Farbsensor die Farbe nur für den Bruchteil einer Sekunde, wenn das Programm startet. Bisher hast du die `while`-Schleife mit der Konstante `True` verwendet, um Programmblöcke endlos zu wiederholen. Du kannst die `while`-Schleife auch mit einem logischen Ausdruck verwenden, um Programmblöcke zu wiederholen, bis der Ausdruck nicht mehr `True` (wahr) ist. Im Beispiel oben wird dieselbe Bedingung wie in der `if`-Anweisung in der inneren `while`-Schleife verwendet, um den Signalton ertönen zu lassen, solange die Farbe Rot erkannt wird. Wenn die Bedingung nicht mehr `True` (wahr) ist, *beendet* der Hub die `while`-Schleife und führt die nächste Programmzeile aus, um den Ton verstummen zu lassen.

Achte bitte in der inneren `while`-Schleife auf die Funktion `sleep_ms()` aus dem Modul `runloop`. Diese Funktion hält die Coroutine `main()` einige Millisekunden lang an, und zwar auf eine *nicht-blockierende* Weise. Da die Funktion das Schlüsselwort `await` enthält, können andere Aufgaben durchgeführt werden, während die Coroutine angehalten wird. Bei diesem Beispiel beträgt die Dauer der Pause eine Millisekunde. Auch wenn dies eine extrem kurze Zeitspanne sein mag, reicht sie dem Hub dennoch, um gleichzeitig viele Coroutinen auszuführen. Die Funktion `sleep_ms()` aus dem Modul `time`, die du in früheren Kapiteln verwendet hast, hält das Programm dagegen auf *blockierende* Weise an. Das heißt, das gesamte Programm wird angehalten, und nicht nur der Programmblock, in dem du die Pause aufrufst.

Weitere Hinweise zu Bedingungen

Du kannst mehr als eine Bedingung hinzufügen, indem du eine `if`-Anweisung mit einer `elif`-Anweisung erweiterst, die eine andere Bedingung prüft. Du kannst so viele dieser Anweisungen hinzufügen, wie benötigt werden. Sie folgen derselben Syntax wie die `if`-Anweisung. Die `elif`-Anweisung sollte sich auf derselben Einrückungsebene wie die `if`-Anweisung befinden, und dem Schlüsselwort `elif` folgen ein logischer Ausdruck und ein Doppelpunkt.

Rücke die nächste(n) Programmzeile(n) ein, wenn diese Bedingung `True` (wahr) ist.

Manchmal ist keine der Bedingungen in den `if`- und `elif`-Anweisungen `True` (wahr). In diesem Fall kannst du einige Programmzeilen ausführen, indem du eine `else`-Anweisung ohne Bedingungen hinzufügst. Dieses Programm wird ausgeführt, wenn alle vorherigen Bedingungen `False` (falsch) sind.

In dem nachstehenden Programm werden beispielsweise eine `elif`- und eine `else`-Anweisung hinzugefügt, damit auch ein Signalton ertönt, wenn die linke Taste gedrückt wird.

```
from hub import button, port, sound
import color
import color_sensor
import runloop

# Diese Funktion gibt `True` (wahr) zurück, wenn der Farbsensor
Rot erkennt.
def red_detected():
    return color_sensor.color(port.A) == color.RED

# Diese Funktion gibt `True` (wahr) zurück, wenn die linke Taste
gedrückt wird.
def left_pressed():
    return button.pressed(button.LEFT) > 0

async def main():
    while True:
        if red_detected():
            # Wenn Rot erkannt wird, ertönt ein sehr langer
Signalton.
            sound.beep(440, 1000000, 100)
            # Warte, bis die Farbe Rot nicht mehr erkannt wird.
            while red_detected():
                await runloop.sleep_ms(1)
        elif left_pressed():
            # Wenn die linke Taste gedrückt wird, soll ein kurzer
Signalton ertönen.
            sound.beep(880, 200, 100)
            # Warte, bis die linke Taste losgelassen wird.
            while left_pressed():
                await runloop.sleep_ms(1)
        else:
            # Lass den Ton andernfalls verstummen.
            sound.stop()

runloop.run(main())
```

Schwenke einen roten LEGO Stein vor dem Farbsensor und drücke die linke Taste am Hub, während das Programm ausgeführt wird. Du hörst einen

Signalton, solange die Farbe Rot erkannt wird. Und bei jedem Drücken der linken Taste ertönt ein kurzer Signalton.

In diesem Beispiel werden zwei Funktionen definiert, um die Logikprüfungen durchzuführen und das Ergebnis zurückzugeben. Die Funktion `red_detected()` prüft, ob Rot die vom Farbsensor erkannte Farbe ist, und gibt das Ergebnis `True` (wahr) oder `False` (falsch) zurück. Die Funktion `left_pressed()` verwendet die Funktion `pressed()` aus dem Modul `hub.button`, benutzt den `>`-Operator, um zu prüfen, ob der Wert *größer als* `0` ist, und gibt das Ergebnis zurück.

Der Code in der `if`-Anweisung ist weitgehend mit dem ersten Beispiel identisch, doch jetzt verwendet er die Funktion `red_detected()` an zwei Stellen, anstatt die Bedingung für die `if`- und `while`-Anweisungen zu wiederholen. Die `elif`-Anweisung verwendet die Funktion `left_pressed()`, um zu prüfen, ob die linke Taste am Hub länger als `0` Millisekunden gedrückt wird.

Beachte bitte, dass die `elif`-Anweisung nur ausgeführt wird, wenn die Bedingung der ersten `if`-Anweisung `False` (falsch) ist. Deshalb hat das Drücken der Taste keinen Effekt, solange der Farbsensor etwas Rotes erkennt. Du solltest dir die Reihenfolge deiner `if`- und `elif`-Bedingungen gut überlegen und die wichtigsten Bedingungen zuerst prüfen. Die `else`-Anweisung stoppt den Signalton, wenn keine der beiden Bedingungen `True` (wahr) ist.

Mehrere Bedingungen

Wenn du die `if`/`elif`/`else`-Anweisungen verwendest, um mehrere Bedingungen zu testen, wird nur einer der Blöcke ausgeführt. Diese Bedingungen sind *einander ausschließend*. Wie du gesehen hast, hatte das Drücken der linken Taste keinen Effekt, solange die Farbe Rot erkannt wurde. Um mehrere Bedingungen richtig zu prüfen, muss dies gleichzeitig erfolgen. In Python kannst du mehrere Coroutinen mit der Funktion `run()` aus dem Modul `runloop` ausführen. Das entspricht dem Hinzufügen mehrerer Stapel von Textblöcken. Bisher hast du die Funktion mit der Coroutine `main()` als ihr einziges Argument aufgerufen. Man kann jedoch auch mehrere Coroutinen als kommagetrennte Argumente weitergeben.

Das nachstehende Programm unterteilt den Code, der die erkannte Farbe und die gedrückte Taste prüft, in zwei Coroutinen. Die Funktion `run()` in der letzten Programmzeile startet beide Coroutinen gleichzeitig.

```
from hub import button, port, sound
```

```
import color
import color_sensor
import runloop

# Diese Funktion gibt `True` (wahr) zurück, wenn der Farbsensor
Rot erkennt.
def red_detected():
    return color_sensor.color(port.A) == color.RED

# Diese Funktion gibt `True` (wahr) zurück, wenn die linke Taste
gedrückt wird.
def left_pressed():
    return button.pressed(button.LEFT) > 0

# Diese Coroutine prüft ständig, ob der Farbsensor die Farbe Rot
erkennt.
async def check_color():
    while True:
        # Warte, bis Rot erkannt wird.
        while not red_detected():
            await runloop.sleep_ms(1)
        # Wenn Rot erkannt wird, ertönt ein sehr langer Signalton.
        sound.beep(440, 1000000, 100)
        # Warte, bis die Farbe Rot nicht mehr erkannt wird.
        while red_detected():
            await runloop.sleep_ms(1)
        # Lass den Ton verstummen, wenn Rot nicht mehr erkannt
wird.
        sound.stop()

# Diese Coroutine prüft ständig, ob die linke Taste gedrückt wird.
async def check_button():
    while True:
        # Warte, bis die linke Taste gedrückt wird.
        while not left_pressed():
            await runloop.sleep_ms(1)
        # Wenn sie gedrückt wird, soll ein kurzer Signalton
ertönen.
        sound.beep(880, 200, 100)
        # Warte, bis die linke Taste losgelassen wird.
        while left_pressed():
            await runloop.sleep_ms(1)

# Führe beide Coroutinen aus.
runloop.run(check_color(), check_button())
```

Schwenke einen roten LEGO Stein vor dem Farbsensor und drücke die linke Taste am Hub, während das Programm ausgeführt wird. Wie zuvor ist ein Signalton zu hören, solange die Farbe Rot erkannt wird. Und bei jedem Drücken der linken Taste ertönt ein kurzer Signalton. Dieses Mal kann man

die linke Taste drücken und einen Signalton hören, während die rote Farbe erkannt wird, da beide Funktionen gleichzeitig ausgeführt werden.

Beim Erstellen eigener Coroutinen solltest du Folgendes beachten:

- Deine Coroutinen sollten mindestens einen Befehl abwarten (eine `await`-Anweisung enthalten).
- Wenn du eine *enge* `while`-Schleife verwendest, solltest du in der Schleife `await runloop.sleep_ms(1)` benutzen, damit auch andere Coroutinen starten und ausgeführt werden können.

Aufgabe

Kannst du das Programm so verändern, dass es eine andere Farbe als Rot erkennt?

Nächste Schritte

In den vorangegangenen Kapiteln hast du

- die Grundlagen gelernt, wie man Python mit SPIKE Prime verwendet, und wie man die Lichtmatrix, das Licht, den Lautsprecher und die Tasten des Hubs und zudem die Motoren, den Farbsensor und den Kraftsensor benutzt.
- dich mit regulären und asynchronen Funktionen, lokalen und globalen Variablen sowie mit Datentypen wie `int`, `bool`, `str`, `tuple` und `list` vertraut gemacht.
- die `for`- und `while`-Schleifen sowie die `if`/`elif`/`else`-Anweisungen verwendet, um den Ablauf deines Programms zu steuern.
- gelernt, wie du Kommentare in deinem Programm verwendest und wie du Fehler in deinem Programm behebst (Debugging), wenn etwas schiefgeht.

Das ist eine beachtliche Leistung, auf die du zurecht sehr stolz sein darfst.

Es gibt noch weitere Informationsquellen, auf die du zugreifen kannst, um noch mehr über die Verwendung von Python mit SPIKE Prime zu erfahren.

Python-Benutzerhandbuch

Der Abschnitt **Erste Schritte** hat gerade mal angerissen, was mit Python und SPIKE Prime möglich ist. Erkunde auch die drei anderen Abschnitte des Python-Benutzerhandbuchs.

1. **Beispiel** Hier findest du Beispielprogramme, die zeigen, wie man Python verwendet, um diverse Aufgaben zu lösen. Kopiere die Beispielprogramme, probiere sie aus und passe sie an deine Bedürfnisse an.
2. **SPIKE Prime Module** Hier findest du die Dokumentation aller Funktionen und Variablen in den SPIKE Prime Modulen sowie kurze Beispiele, wie man sie verwendet.

Python-Lektionen

Wähle auf der Website LEGOeducation.com/lessons das Produkt **SPIKE™ Prime mit Python** aus. Dort findest du mehrere Einheitenpläne mit jeweils 6-8 Lektionen (nur auf Englisch verfügbar). Diese mehr als 50 Lektionen decken ein breites Themenspektrum ab und befassen sich unter anderem mit dem Debuggen, der Sensorsteuerung, einfachen Spielen oder Daten und mathematischen Funktionen. Entdecke die zahlreichen Möglichkeiten und werde eine echte Expertin beziehungsweise ein echter Experte für die Nutzung von Python mit SPIKE Prime.

Aufgabe

Erstelle ein neues Python-Projekt und mach dich ans Programmieren!

API-Module

App

Das Modul `app` (App) dient zur Kommunikation zwischen Hub und App

Untermodule

Balkendiagramm

Das Modul `bargraph` (Balkendiagramm) wird verwendet, um Balkendiagramme in der SPIKE App zu erstellen

Um das Modul `bargraph` (Balkendiagramm) zu verwenden, musst du es nur folgendermaßen importieren:

```
from app import bargraph
```

`bargraph` Details

Funktionen

change

change(color: int, value: float) -> None

Parameter

color: int

Eine Farbe aus dem Modul `color` (Farbe)

value: float

Der Wert

clear_all

clear_all() -> None

Parameter

get_value

get_value(color: int) -> Awaitable

Parameter

color: int

Eine Farbe aus dem Modul `color` (Farbe)

hide

hide() -> None

Parameter

set_value

set_value(color: int, value: float) -> None

Parameter

color: int

Eine Farbe aus dem Modul `color` (Farbe)

value: float

Der Wert

show

show(fullscreen: bool) -> None

Parameter

fullscreen: bool

Im Vollbildmodus anzeigen

Anzeige

Das Modul `display` (Anzeige) wird verwendet, um Bilder in der SPIKE App anzuzeigen

Um das Modul `display` (Anzeige) zu verwenden, musst du es nur folgendermaßen importieren:

```
from app import display
```

`display` Details

Funktionen

hide

hide() -> None

Parameter

image

image(image: int) -> None

Parameter

image: int

Die Kennung (ID) des anzuzeigenden Bildes Der Bereich der verfügbaren Bilder liegt zwischen 1 und 21. Für diese Bilder sind im Modul `display` (Anzeige) Konstanten verfügbar

show

show(fullscreen: bool) -> None

Parameter

fullscreen: bool

Im Vollbildmodus anzeigen

text

text(text: str) -> None

Parameter

text: str

Der anzuzeigende Text

Konstanten

app.display Konstanten

IMAGE_ROBOT_1 = 1

IMAGE_ROBOT_2 = 2

IMAGE_ROBOT_3 = 3

IMAGE_ROBOT_4 = 4

IMAGE_ROBOT_5 = 5

IMAGE_HUB_1 = 6

IMAGE_HUB_2 = 7

IMAGE_HUB_3 = 8

IMAGE_HUB_4 = 9

IMAGE_AMUSEMENT_PARK = 10

IMAGE_BEACH = 11

IMAGE_HAUNTED_HOUSE = 12

IMAGE_CARNIVAL = 13

IMAGE_BOOKSHELF = 14

IMAGE_PLAYGROUND = 15

IMAGE_MOON = 16

IMAGE_CAVE = 17

IMAGE_OCEAN = 18

IMAGE_POLAR_BEAR = 19

IMAGE_PARK = 20

IMAGE_RANDOM = 21

Liniendiagramm

Das Modul `linegraph` (Liniendiagramm) wird verwendet, um Liniendiagramme in der SPIKE App zu erstellen

Um das Modul `linegraph` (Liniendiagramm) zu verwenden, musst du es nur folgendermaßen importieren:

```
from app import linegraph
```

`linegraph` Details

Funktionen

clear

clear(color: int) -> None

Parameter

color: int

Eine Farbe aus dem Modul `color` (Farbe)

clear_all

clear_all() -> None

Parameter

get_average

get_average(color: int) -> Awaitable

Parameter

color: int

Eine Farbe aus dem Modul `color` (Farbe)

get_last

get_last(color: int) -> Awaitable

Parameter

color: int

Eine Farbe aus dem Modul `color` (Farbe)

get_max

get_max(color: int) -> Awaitable

Parameter

color: int

Eine Farbe aus dem Modul `color` (Farbe)

get_min

get_min(color: int) -> Awaitable

Parameter

color: int

Eine Farbe aus dem Modul `color` (Farbe)

hide

hide() -> None

Parameter

plot

plot(color: int, x: float, y: float) -> None

Parameter

color: int

Eine Farbe aus dem Modul `color` (Farbe)

x: float

Der X-Wert

y: float

Der Y-Wert

show

show(fullscreen: bool) -> None

Parameter

fullscreen: bool

Im Vollbildmodus anzeigen

Musik

Das Modul `music` (Musik) wird verwendet, um in der SPIKE App Musik zu machen

Um das Modul `music` (Musik) zu verwenden, musst du es nur folgendermaßen importieren:

```
from app import music
```

`music` Details

Funktionen

play_drum

play_drum(drum: int) -> None

Parameter

drum: int

Der Name des Schlagzeugs. Im Modul app.sound findest du alle verfügbaren Werte.

play_note

play_note(instrument: int, note: int, duration: int) -> None

Parameter

instrument: int

Der Name des Instruments. Im Modul app.sound findest du alle verfügbaren Werte.

note: int

Die abzuspielende MIDI-Note (0-130)

duration: int

Dauer in Millisekunden

Konstanten

app.music Konstanten

DRUM_BASS = 2

DRUM_BONGO = 13

DRUM_CABASA = 15

DRUM_CLAVES = 9

DRUM_CLOSED_HI_HAT = 6

DRUM_CONGA = 14

DRUM_COWBELL = 11

DRUM_CRASH_CYMBAL = 4

DRUM_CUICA = 18

DRUM_GUIRO = 16

DRUM_HAND_CLAP = 8

DRUM_OPEN_HI_HAT = 5

DRUM_SIDE_STICK = 3

DRUM_SNARE = 1

DRUM_TAMBOURINE = 7

DRUM_TRIANGLE = 12

DRUM_VIBRASLAP = 17

DRUM_WOOD_BLOCK = 10

INSTRUMENT_BASS = 6

INSTRUMENT_BASSOON = 14

INSTRUMENT_CELLO = 8

INSTRUMENT_CHOIR = 15

INSTRUMENT_CLARINET = 10

INSTRUMENT_ELECTRIC_GUITAR = 5

INSTRUMENT_ELECTRIC_PIANO = 2

INSTRUMENT_FLUTE = 12

INSTRUMENT_GUITAR = 4

INSTRUMENT_MARIMBA = 19

INSTRUMENT_MUSIC_BOX = 17

INSTRUMENT_ORGAN = 3

INSTRUMENT_PIANO = 1

INSTRUMENT_PIZZICATO = 7

INSTRUMENT_SAXOPHONE = 11

INSTRUMENT_STEEL_DRUM = 18

INSTRUMENT_SYNTH_LEAD = 20

INSTRUMENT_SYNTH_PAD = 21

INSTRUMENT_TROMBONE = 9

INSTRUMENT_VIBRAPHONE = 16

INSTRUMENT_WOODEN_FLUTE = 13

Klang

Das Modul `sound` (Klang) wird verwendet, um Geräusche in der SPIKE App abzuspielen

Um das Modul `sound` (Klang) zu verwenden, musst du es nur folgendermaßen importieren:

```
from app import sound
```

`sound` Details

Funktionen

play

play(sound_name: str, volume: int = 100, pitch: int = 0, pan: int = 0) -> Awaitable

Ein Geräusch in der SPIKE App abspielen

Parameter

sound_name: str

Der Name des Klangs, wie er in Sounderweiterung von Word Blocks angegeben ist

volume: int

Die Lautstärke (0-100)

pitch: int

Die Tonhöhe des Klangs

pan: int

Die Lautstärkeverteilung bestimmt, welcher Lautsprecher den Soundeffekt ausgibt. Dabei steht „-100“ nur für den linken Lautsprecher, „0“ für die Normaleinstellung und „100“ nur für den rechten Lautsprecher.

set_attributes

set_attributes(volume: int, pitch: int, pan: int) -> None

Parameter

volume: int

Die Lautstärke (0-100)

pitch: int

Die Tonhöhe des Klangs

pan: int

Die Lautstärkeverteilung bestimmt, welcher Lautsprecher den Soundeffekt ausgibt. Dabei steht „-100“ nur für den linken Lautsprecher, „0“ für die Normaleinstellung und „100“ nur für den rechten Lautsprecher.

stop

stop() -> None

Parameter

Color (Farbe)

Das Modul `color` (Farbe) enthält alle Farbkonstanten, die mit den Modulen `color_matrix` (Farbmatrix), `color_sensor` (Farbsensor) und `light` (Licht) verwendet werden können.

Um das Modul „Color“ (Farbe) zu verwenden, musst du deinem Projekt folgende import-Anweisung hinzufügen:

```
import color
```

Konstanten

color Konstanten

BLACK = 0

MAGENTA = 1

PURPLE = 2

BLUE = 3

AZURE = 4

TURQUOISE = 5

GREEN = 6

YELLOW = 7

ORANGE = 8

RED = 9

WHITE = 10

UNKNOWN = -1

Color Matrix (Farbmatrix)

Um das Modul „Color Matrix“ (Farbmatrix) zu verwenden, musst du deinem Projekt folgende import-Anweisung hinzufügen:

```
import color_matrix
```

Alle Funktionen im Modul sollten innerhalb des Moduls `color_matrix` (Motor) als Präfix wie folgt aufgerufen werden:

```
color_matrix.set_pixel(port.A, 1, 1, (color.BLUE, 10))
```

Funktionen

clear

clear(port: int) -> None

Schalte alle Pixel in einer Color Matrix (Farbmatrix) aus

```
from hub import port
import color_matrix

color_matrix.clear(port.A)
```

Parameter

port: int

Ein Anschluss aus dem Untermodul `port` (Anschluss) im Modul `hub` (Hub)

get_pixel

get_pixel(port: int, x: int, y: int) -> tuple[int, int]

Ruf ein bestimmtes Pixel ab, das als Tupel dargestellt wird, das die Farbe und Intensität enthält

```
from hub import port
import color_matrix

# Drucke die Farbe und Intensität des Pixels 0,0 auf der
Farbmatrix, die an Port A (Anschluss A) angeschlossen ist
print(color_matrix.get_pixel(port.A, 0, 0))
```

Parameter

port: int

Ein Anschluss aus dem Untermodul `port` (Anschluss) im Modul `hub` (Hub)

x: int

Der X-Wert (0 - 2)

y: int

Der Y-Wert, Bereich (0 - 2)

set_pixel

set_pixel(port: int, x: int, y: int, pixel: tuple[color: int, intensity: int]) -> None

Ändere ein einzelnes Pixel in einer Color Matrix (Farbmatrix)

```
from hub import port
import color
import color_matrix

# Ändere die Farbe des Pixels 0,0 auf der Farbmatrix, die an Port
A (Anschluss A) angeschlossen ist
color_matrix.set_pixel(port.A, 0, 0, (color.RED, 10))
```

```
# Drucke die Farbe des Pixels 0,0 auf der Farbmatrix, die an Port
A (Anschluss A) angeschlossen ist
print(color_matrix.get_pixel(port.A, 0, 0)[0])
```

Parameter

port: int

Ein Anschluss aus dem Untermodul `port` (Anschluss) im Modul `hub` (Hub)

x: int

Der X-Wert (0 - 2)

y: int

Der Y-Wert, Bereich (0 - 2)

pixel: tuple[color: int, intensity: int]

Tupel, das die Farbe und Intensität enthält und festlegt, wie hell das Pixel leuchten soll

show

show(port: int, pixels: list[tuple[int, int]]) -> None

Ändere alle Pixel in einer Color Matrix (Farbmatrix)

```
from hub import port
import color
import color_matrix

# Aktualisiere alle Pixel auf der Farbmatrix mithilfe der Funktion
„Programm anzeigen“

# Erstelle eine Liste mit 18 Elementen (Farb- und
Intensitätspaare)
pixels = [(color.BLUE, 10)] * 9

# Aktualisiere alle Pixel so, dass sie in derselben Farbe und
Intensität leuchten
color_matrix.show(port.A, pixels)
```

Parameter

port: int

Ein Anschluss aus dem Untermodul `port` (Anschluss) im Modul `hub` (Hub)

pixels: list[tuple[int, int]]

Eine Liste, die Farb- und Intensitätswertetupel für alle 9 Pixel enthält.

Color Sensor (Farbsensor)

Das Modul `color_sensor` (Farbsensor) ermöglicht es dir, Code zu schreiben, der auf bestimmte Farben oder die Intensität des reflektierten Lichts reagiert.

Um das Modul „Color Sensor“ (Farbsensor) zu verwenden, musst du deinem Projekt folgende import-Anweisung hinzufügen:

```
import color_sensor
```

Alle Funktionen im Modul sollten innerhalb des Moduls `color_sensor` (Farbsensor) als Präfix wie folgt aufgerufen werden:

```
color_sensor.reflection(port.A)
```

Der Farbsensor kann folgende Farben erkennen:

Rot
Grün
Blau
Magenta
Gelb
Orange
Hellblau
Schwarz
Weiß

Funktionen

color

color(port: int) -> int

Gibt den Farbwert der erkannten Farbe zurück. Verwende das Modul `color` (Farbe), um den Farbwert einer bestimmten Farbe zuzuordnen.

```
import color_sensor
from hub import port
import color

if color_sensor.color(port.A) is color.RED:
    print("Red detected")
```

Parameter

port: int

Ein Anschluss aus dem Untermodul `port` (Anschluss) im Modul `hub` (Hub)

reflection

reflection(port: int) -> int

Ruft die Intensität des reflektierten Lichts (0-100%) ab.

Parameter

port: int

Ein Anschluss aus dem Untermodul `port` (Anschluss) im Modul `hub` (Hub)

rgbi

rgbi(port: int) -> tuple[int, int, int, int]

Ruft die Gesamtfarbintensität und die Intensität von Rot, Grün und Blau ab.

Gibt tuple[red: int, green: int, blue: int, intensity: int] zurück

Parameter

port: int

Ein Anschluss aus dem Untermodul `port` (Anschluss) im Modul `hub` (Hub)

Device (Gerät)

Das Modul `device` (Gerät) ermöglicht es dir, Code zu schreiben, um Informationen zu den an den Hub angeschlossenen Geräten abzurufen.

Um das Modul „Device“ (Gerät) zu verwenden, musst du die folgende import-Anweisung zu deinem Projekt hinzufügen:

```
import device
```

Alle Funktionen im Modul sollten innerhalb des Moduls `device` (run-Schleife) als Präfix wie folgt aufgerufen werden:

```
device.device_id(port.A)
```

Funktionen

data

data(port: int) -> tuple[int]

Ruf die LPF-2-Rohdaten von einem Gerät ab.

Parameter

port: int

Ein Anschluss aus dem Untermodul `port` (Anschluss) im Modul `hub` (Hub)

device_id

device_id(port: int) -> int

Ruf die Gerätekennung eines Geräts ab. Jedes Gerät hat eine Kennung (ID), die auf seinem Typ basiert.

Parameter

port: int

Ein Anschluss aus dem Untermodul `port` (Anschluss) im Modul `hub` (Hub)

get_duty_cycle

get_duty_cycle(port: int) -> int

Ruf den Arbeitszyklus für ein Gerät ab. Die zurückgegebenen Werte liegen im Bereich von 0 bis 10000

Parameter

port: int

Ein Anschluss aus dem Untermodul `port` (Anschluss) im Modul `hub` (Hub)

ready

ready(port: int) -> bool

Wenn ein Gerät an den Hub angeschlossen ist, kann es eine kurze Zeit dauern, bis es zur Annahme von Anforderungen bereit ist.
Verwende die Option `ready` (bereit), um die Bereitschaft der angeschlossenen Geräte zu testen.

Parameter

port: int

Ein Anschluss aus dem Untermodul `port` (Anschluss) im Modul `hub` (Hub)

set_duty_cycle

set_duty_cycle(port: int, duty_cycle: int) -> None

Stell den Arbeitszyklus an einem Gerät ein. Bereich 0 bis 10000

Parameter

port: int

Ein Anschluss aus dem Untermodul `port` (Anschluss) im Modul `hub` (Hub)

duty_cycle: int

Der PWM-Wert (0-10000)

Distance Sensor (Abstandssensor)

Das Modul `distance_sensor` (Abstandssensor) ermöglicht es dir, Code zu schreiben, der auf bestimmte Entfernungen reagiert oder den Abstandssensor unterschiedlich leuchten lässt.

Um das Modul „Distance Sensor“ (Abstandssensor) zu verwenden, musst du deinem Projekt folgende import-Anweisung hinzufügen:

```
import distance_sensor
```

Alle Funktionen im Modul sollten innerhalb des Moduls `distance_sensor` (Abstandssensor) als Präfix wie folgt aufgerufen werden:

```
distance_sensor.distance(port.A)
```

Funktionen

clear

clear(port: int) -> None

Schaltet alle Lichter im Abstandssensor aus, der an den `port` (Anschluss) angeschlossen ist.

Parameter

port: int

Ein Anschluss aus dem Untermodul `port` (Anschluss) im Modul `hub` (Hub)

distance

distance(port: int) -> int

Ruf den Abstand in Millimetern ab, der von dem an den `port` (Anschluss) angeschlossenen Abstandssensor erfasst wurde. Wenn der Abstandssensor keinen gültigen Abstand lesen kann, gibt er -1 zurück.

Parameter

port: int

Ein Anschluss aus dem Untermodul `port` (Anschluss) im Modul `hub` (Hub)

get_pixel

get_pixel(port: int, x: int, y: int) -> int

Ruf die Intensität eines bestimmten Lichts an dem Abstandssensor ab, der an dem `port` (Anschluss) angeschlossenen ist.

Parameter

port: int

Ein Anschluss aus dem Untermodul `port` (Anschluss) im Modul `hub` (Hub)

x: int

Der X-Wert (0 - 3)

y: int

Der Y-Wert, Bereich (0 - 3)

set_pixel

set_pixel(port: int, x: int, y: int, intensity: int) -> None

Ändert die Intensität eines bestimmten Lichts an dem Abstandssensor, der an dem `port` (Anschluss) angeschlossen ist.

Parameter

port: int

Ein Anschluss aus dem Untermodul `port` (Anschluss) im Modul `hub` (Hub)

x: int

Der X-Wert (0 - 3)

y: int

Der Y-Wert, Bereich (0 - 3)

intensity: int

Wie hell das Pixel leuchten soll

show

show(port: int, pixels: list[int]) -> None

Ändere alle Lichter gleichzeitig.

```
from hub import port
import distance_sensor

# Aktualisiere alle Pixel auf dem Abstandssensor mithilfe der
Funktion „Programm anzeigen“

# Erstelle eine Liste mit 4 identischen Intensitätswerten
pixels = [100] * 4

# Aktualisiere alle Pixel so, dass sie dieselbe Intensität zeigen
distance_sensor.show(port.A, pixels)
```

Parameter

port: int

Ein Anschluss aus dem Untermodul `port` (Anschluss) im Modul `hub` (Hub)

pixels: bytes

Eine Liste, die Intensitätswerte für alle 4 Pixel enthält.

Force Sensor (Kraftsensor)

Das Modul `force_sensor` (run-Schleife) enthält alle Funktionen und Konstanten, um den Kraftsensor zu verwenden.

Um das Modul „Force Sensor“ (Kraftsensor) zu verwenden, musst du deinem Projekt folgende import-Anweisung hinzufügen:

```
import force_sensor
```

Alle Funktionen im Modul sollten innerhalb des Moduls `force_sensor` (run-Schleife) als Präfix wie folgt aufgerufen werden:

```
force_sensor.force(port.A)
```

Funktionen

force

force(port: int) -> int

Ruft die gemessene Kraft in Dezinewton ab. Wertebereich 0 bis 100

```
from hub import port
import force_sensor


print(force_sensor.force(port.A))
```

Parameter

port: int

Ein Anschluss aus dem Untermodul `port` (Anschluss) im Modul `hub` (Hub)

pressed

pressed(port: int) -> bool

Prüft, ob die Taste am Sensor gedrückt wird. Gibt „true“ (wahr) zurück, wenn der an den Anschluss angeschlossene Kraftsensor gedrückt wird.

```
from hub import port
import force_sensor


print(force_sensor.pressed(port.A))
```

Parameter

port: int

Ein Anschluss aus dem Untermodul `port` (Anschluss) im Modul `hub` (Hub)

raw

raw(port: int) -> int

Gibt den unverarbeiteten, nicht kalibrierten Kraftwert des an den `port` (Anschluss) angeschlossenen Kraftsensors zurück.

```
from hub import port
import force_sensor


print(force_sensor.raw(port.A))
```

Parameter

port: int

Ein Anschluss aus dem Untermodul `port` (Anschluss) im Modul `hub` (Hub)

Hub

Untermodule

Button (Taste)

Um das Modul „Button“ (Taste) zu verwenden, musst du die folgende import-Anweisung zu deinem Projekt hinzufügen:

```
from hub import button
```

Alle Funktionen im Modul sollten innerhalb des Moduls `button` (Motor) als Präfix wie folgt aufgerufen werden:

```
button.pressed(button.LEFT)
```

Funktionen

pressed

int pressed(button: int) -> int

Dieses Modul lässt dein Programm auf Tasten reagieren, die am Hub gedrückt werden. Um die Tasten verwenden zu können, musst du zunächst das Modul `button` (Taste) importieren.

```
from hub import button

left_button_press_duration = 0

# Warte, bis die linke Taste gedrückt wird
while not button.pressed(button.LEFT):
    pass

# Halte die linke Taste gedrückt und aktualisiere die Variable
`left_button_press_duration`
while button.pressed(button.LEFT):
    left_button_press_duration = button.pressed(button.LEFT)

print("Left button was pressed for " +
str(left_button_press_duration) + " milliseconds")
```

Parameter

button: int

Eine Taste aus dem Untermodul `button` (Taste) im Modul `hub` (Hub)

Konstanten

hub.button Konstanten

LEFT = 1
Taste links neben dem Ein-/Aus-Schalter am SPIKE Prime Hub
RIGHT = 2
Taste rechts neben dem Ein-/Aus-Schalter am SPIKE Prime Hub

Light (Licht)

Das Modul `light` (Licht) enthält Funktionen, um die Farbe des Lichts am SPIKE Prime Hub zu ändern.

Um das Modul „Light“ (Licht) zu verwenden, musst du deinem Projekt folgende import-Anweisung hinzufügen:

```
from hub import light
```

Alle Funktionen im Modul sollten innerhalb des Moduls `light` (Licht) als Präfix wie folgt aufgerufen werden:

```
light.color(color.RED)
```

Funktionen

color

color(light: int, color: int) -> None

Ändere die Farbe eines Lichts am Hub.

```
from hub import light
import color

# Ändere das Licht in Rot
light.color(light.POWER, color.RED)
```

Parameter

light: int

Das Licht am Hub

color: int

Eine Farbe aus dem Modul `color` (Farbe)

Konstanten

hub.light Konstanten

POWER = 0
Der Ein-/Aus-Schalter Am SPIKE Prime Hub befindet sich dieser Schalter zwischen der linken und der rechten Taste.
CONNECT = 1
Das Licht, das die Bluetooth-Verbindungstaste am SPIKE Prime Hub umringt.

Light Matrix (Lichtmatrix)

Um das Modul „Light Matrix“ (Lichtmatrix) zu verwenden, musst du deinem Projekt die folgende import-Anweisung hinzufügen:

```
from hub import light_matrix
```

Alle Funktionen im Modul sollten innerhalb des Moduls `light_matrix` (Lichtmatrix) als Präfix wie folgt aufgerufen werden:

```
light_matrix.write("Hello World")
```

Funktionen

clear

clear() -> None

Schaltet alle Pixel auf der Lichtmatrix aus.

```
from hub import light_matrix
import time
# Aktualisiere die Pixel so, dass ein Bild auf der Lichtmatrix angezeigt wird. Schalte die Pixel dann mithilfe der Löschfunktion aus.

# Lass ein kleines Herz auf der Lichtmatrix leuchten
light_matrix.show_image(2)

# Warte zwei Sekunden
time.sleep_ms(2000)

# Schalte das Herz aus
light_matrix.clear()
```

Parameter

get_orientation

get_orientation() -> int

Rufe die aktuelle Ausrichtung der Lichtmatrix ab.
Kann mit den folgenden Konstanten verwendet werden: `orientation.UP`(Ausrichtung: NACH OBEN) `orientation.LEFT` (Ausrichtung: NACH LINKS, `orientation.RIGHT` (Ausrichtung: NACH RECHTS), `orientation.DOWN` (Ausrichtung: NACH UNTEN)

Parameter

get_pixel

get_pixel(x: int, y: int) -> int

Ruf die Intensität eines bestimmten Pixels auf der Lichtmatrix ab.

```
from hub import light_matrix

# Lass ein Herz leuchten
light_matrix.show_image(1)

# Drucke den Wert der Intensität des mittleren Pixels
print(light_matrix.get_pixel(2, 2))
```

Parameter

x: int

Der X-Wert, Bereich (0 - 4)

y: int

Der Y-Wert, Bereich (0 - 4)

set_orientation

set_orientation(top: int) -> int

Ändere die Ausrichtung der Lichtmatrix. Für alle nachfolgenden Aufrufe wird die neue Ausrichtung verwendet.
Kann mit den folgenden Konstanten verwendet werden: `orientation.UP`(Ausrichtung: NACH OBEN) `orientation.LEFT` (Ausrichtung: NACH LINKS, `orientation.RIGHT` (Ausrichtung: NACH RECHTS), `orientation.DOWN` (Ausrichtung: NACH UNTEN)

Parameter

top: int

Die Seite des Hubs, die nach oben zeigen soll

set_pixel

set_pixel(x: int, y: int, intensity: int) -> None

Stellt die Helligkeit eines Pixels (1 von 25 LEDs) auf der Lichtmatrix ein.

```
from hub import light_matrix
# Schalte das Pixel in der Mitte des Hubs ein
light_matrix.set_pixel(2, 2, 100)
```

Parameter

x: int

Der X-Wert, Bereich (0 - 4)

y: int

Der Y-Wert, Bereich (0 - 4)

intensity: int

Wie hell das Pixel leuchten soll

show

show(pixels: list[int]) -> None

Ändere alle Lichter gleichzeitig.

```
from hub import light_matrix
# Aktualisiere alle Pixel auf der Lichtmatrix mithilfe der
Funktion „Programm anzeigen“

# Erstelle eine Liste mit 25 identischen Intensitätswerten
pixels = [100] * 25

# Aktualisiere alle Pixel so, dass sie mit derselben Intensität
leuchten
light_matrix.show(pixels)
```

Parameter

pixels: Iterable

Eine Liste, die Lichtintensitätswerte für alle 25 Pixel enthält.

show_image

show_image(image: int) -> None

Zeig eines der integrierten Bilder auf dem Display an.

```
from hub import light_matrix
# Aktualisiere die Pixel mithilfe der Funktion „show_image“ so,
dass auf der Lichtmatrix ein Bild angezeigt wird

# Zeig ein lächelndes Gesicht an
light_matrix.show_image(light_matrix.IMAGE_HAPPY)
```

Parameter

image: int

Die Kennung (ID) des anzuzeigenden Bildes Der Bereich der verfügbaren Bilder liegt zwischen 1 und 67. Für diese Bilder sind im Modul `light_matrix` (Lichtmatrix) Konstanten verfügbar.

write

write(text: str, intensity: int = 100, time_per_character: int = 500) -> Awaitable

Zeigt einen Text auf der Lichtmatrix als einzelne Buchstaben an, die von rechts nach links durchlaufen. Wenn nur einzelnes Zeichen angezeigt werden soll, läuft dieses nicht von links nach rechts durch

```
from hub import light_matrix
# Weiß ist eine Nachricht an den Hub
light_matrix.write("Hello, world!")
```

Parameter

text: str

Der anzuzeigende Text

intensity: int

Wie hell das Pixel leuchten soll

time_per_character: int

Wie lange jedes Zeichen auf dem Display angezeigt werden soll

Konstanten

hub.light_matrix Konstanten

IMAGE_HEART = 1
IMAGE_HEART_SMALL = 2
IMAGE_HAPPY = 3
IMAGE_SMILE = 4
IMAGE_SAD = 5
IMAGE_CONFUSED = 6
IMAGE_ANGRY = 7
IMAGE_ASLEEP = 8
IMAGE_SURPRISED = 9
IMAGE_SILLY = 10
IMAGE_FABULOUS = 11
IMAGE_MEH = 12
IMAGE_YES = 13
IMAGE_NO = 14
IMAGE_CLOCK12 = 15
IMAGE_CLOCK1 = 16
IMAGE_CLOCK2 = 17
IMAGE_CLOCK3 = 18

IMAGE_CLOCK4 = 19

IMAGE_CLOCK5 = 20

IMAGE_CLOCK6 = 21

IMAGE_CLOCK7 = 22

IMAGE_CLOCK8 = 23

IMAGE_CLOCK9 = 24

IMAGE_CLOCK10 = 25

IMAGE_CLOCK11 = 26

IMAGE_ARROW_N = 27

IMAGE_ARROW_NE = 28

IMAGE_ARROW_E = 29

IMAGE_ARROW_SE = 30

IMAGE_ARROW_S = 31

IMAGE_ARROW_SW = 32

IMAGE_ARROW_W = 33

IMAGE_ARROW_NW = 34

IMAGE_GO_RIGHT = 35

IMAGE_GO_LEFT = 36

IMAGE_GO_UP = 37

IMAGE_GO_DOWN = 38

IMAGE_TRIANGLE = 39

IMAGE_TRIANGLE_LEFT = 40

IMAGE_CHESSBOARD = 41

IMAGE_DIAMOND = 42

IMAGE_DIAMOND_SMALL = 43

IMAGE_SQUARE = 44

IMAGE_SQUARE_SMALL = 45

IMAGE_RABBIT = 46

IMAGE_COW = 47

IMAGE_MUSIC_CROTCHET = 48

IMAGE_MUSIC_QUAVER = 49

IMAGE_MUSIC_QUAVERS = 50

IMAGE_PITCHFORK = 51

IMAGE_XMAS = 52

IMAGE_PACMAN = 53

IMAGE_TARGET = 54

IMAGE_TSHIRT = 55

IMAGE_ROLLERSKATE = 56

IMAGE_DUCK = 57

IMAGE_HOUSE = 58

IMAGE_TORTOISE = 59

IMAGE_BUTTERFLY = 60

IMAGE_STICKFIGURE = 61

IMAGE_GHOST = 62

IMAGE_SWORD = 63

IMAGE_GIRAFFE = 64

IMAGE_SKULL = 65

IMAGE_UMBRELLA = 66

IMAGE_SNAKE = 67

Motion Sensor (Bewegungssensor)

Um das Modul „Motion Sensor“ (Bewegungssensor) zu verwenden, musst du deinem Projekt folgende import-Anweisung hinzufügen:

```
from hub import motion_sensor
```

Alle Funktionen im Modul sollten innerhalb des Moduls `motion_sensor` (Bewegungssensor) als Präfix wie folgt aufgerufen werden:

```
motion_sensor.up_face()
```

Funktionen

acceleration

acceleration(raw_unfiltered: bool) -> tuple[int, int, int]

Gibt ein Tupel zurück, das x-, y- und z-Beschleunigungswerte als ganze Zahlen enthält. Die Werte sind in Milli-G (1/1000 G) angegeben

Parameter

raw_unfiltered: bool

Wenn wir die Daten unverarbeitet und ungefiltert zurückhaben wollen

angular_velocity

angular_velocity(raw_unfiltered: bool) -> tuple[int, int, int]

Gibt ein Tupel zurück, das x-, y- und z-Winkelgeschwindigkeitswerte als ganze Zahlen enthält. Die Werte sind in Dezigrad pro Sekunde angegeben

Parameter

raw_unfiltered: bool

Wenn wir die Daten unverarbeitet und ungefiltert zurückhaben wollen

gesture

gesture() -> int

Gibt die erkannte Geste zurück.

Folgende Werte sind möglich:

```
motion_sensor.TAPPED
motion_sensor.DOUBLE_TAPPED
motion_sensor.SHAKEN
motion_sensor.FALLING
motion_sensor.UNKNOWN
```

Parameter

get_yaw_face

get_yaw_face() -> int

Ruf die Seite des Hubs ab, auf die sich das Gieren bezieht.
Wenn du den Hub so auf eine ebene Fläche stellst, dass die zurückgegebene Seite nach oben zeigt, wird nur der Gierwert aktualisiert, wenn du den Hub drehst.
`motion_sensor.TOP` Die Seite des SPIKE Prime Hubs mit dem USB-Ladeanschluss.
`motion_sensor.FRONT` Die Seite des SPIKE Prime Hubs mit der Lichtmatrix.
`motion_sensor.RIGHT` Die rechte Seite des SPIKE Prime Hubs (von vorne betrachtet).
`motion_sensor.BOTTOM` Die Seite des SPIKE Prime Hubs, wo sich der Akku befindet.
`motion_sensor.BACK` Die Seite des SPIKE Prime Hubs, an der sich der Lautsprecher befindet.
`motion_sensor.LEFT` Die linke Seite des SPIKE Prime Hubs (von vorne betrachtet).

Parameter

quaternion

quaternion() -> tuple[float, float, float, float]

Gibt die Quaternion der Hub-Ausrichtung als tuple[w: float, x: float, y: float, z: float] zurück.

Parameter

reset_tap_count

reset_tap_count() -> None

Setz die Tap-Anzahl zurück, die von der Funktion `tap_count` (Tap-Anzahl) zurückgegeben wird.

Parameter

reset_yaw

reset_yaw(angle: int) -> None

Ändere den Versatz des Gierwinkels.
Der eingestellte Winkel ist der neue Gierwert.

Parameter

angle: int

set_yaw_face

set_yaw_face(up: int) -> bool

Ändere die Seite des Hubs, die als Gierebene benutzt wird. Wenn du den Hub so auf eine ebene Fläche stellst, dass diese Seite nach oben zeigt, wird nur der Gierwert aktualisiert, wenn du den Hub drehst.

Parameter

up: int

Die Seite des Hubs, die nach oben zeigen soll
Die verfügbaren Werte sind:

`motion_sensor.TOP` Die Seite des SPIKE Prime Hubs mit dem USB-Ladeanschluss.
`motion_sensor.FRONT` Die Seite des SPIKE Prime Hubs mit der Lichtmatrix.
`motion_sensor.RIGHT` Die rechte Seite des SPIKE Prime Hubs (von vorne betrachtet).
`motion_sensor.BOTTOM` Die Seite des SPIKE Prime Hubs, wo sich der Akku befindet.
`motion_sensor.BACK` Die Seite des SPIKE Prime Hubs, an der sich der Lautsprecher befindet.
`motion_sensor.LEFT` Die linke Seite des SPIKE Prime Hubs (von vorne betrachtet).

stable

stable() -> bool

Prüft, ob der Hub auf einer ebenen Fläche steht.

Parameter

tap_count

tap_count() -> int

Gibt die Anzahl der Taps zurück, die seit dem Start des Programms oder dem letzten Aufruf von `motion_sensor.reset_tap_count()` erkannt wurden.

Parameter

tilt_angles

tilt_angles() -> tuple[int, int, int]

Gibt ein Tupel zurück, das Gier-, Neigungs- und Rollwerte als ganze Zahlen enthält. Die Werte sind in Dezigrad angegeben

Parameter

up_face

up_face() -> int

Gibt die Seite des Hubs zurück, die zurzeit nach oben zeigt.
`motion_sensor.TOP` Die Seite des SPIKE Prime Hubs mit dem USB-Ladeanschluss.
`motion_sensor.FRONT` Die Seite des SPIKE Prime Hubs mit der Lichtmatrix.
`motion_sensor.RIGHT` Die rechte Seite des SPIKE Prime Hubs (von vorne betrachtet).
`motion_sensor.BOTTOM` Die Seite des SPIKE Prime Hubs, wo sich der Akku befindet.
`motion_sensor.BACK` Die Seite des SPIKE Prime Hubs, an der sich der

Lautsprecher befindet.
`motion_sensor.LEFT` Die linke Seite des SPIKE Prime Hubs (von vorne betrachtet).

Parameter

Konstanten

hub.motion_sensor Konstanten

TAPPED = 0

DOUBLE_TAPPED = 1

SHAKEN = 2

FALLING = 3

UNKNOWN = -1

TOP = 0
Die Seite des SPIKE Prime Hubs mit der Lichtmatrix.
FRONT = 1
Die Seite des SPIKE Prime Hubs, an der sich der Lautsprecher befindet.
RIGHT = 2
Die rechte Seite des SPIKE Prime Hubs (von vorne betrachtet).
BOTTOM = 3
Die Seite des SPIKE Prime Hubs, wo sich der Akku befindet.
BACK = 4
Die Seite des SPIKE Prime Hubs mit dem USB-Ladeanschluss.
LEFT = 5
Die linke Seite des SPIKE Prime Hubs (von vorne betrachtet).

Port (Anschluss)

Dieses Modul enthält Konstanten, die einfachen Zugriff auf die Ports (Anschlüsse) am SPIKE Prime Hub bieten. Verwende die Konstanten in allen Funktionen, die den Parameter `port` (Anschluss) erfordern.

Um das Modul „Port“ (Anschluss) zu verwenden, musst du die folgende import-Anweisung zu deinem Projekt hinzufügen:

```
from hub import port
```
Alle Funktionen im Modul sollten innerhalb des Moduls `port` (Anschluss) als Präfix wie folgt aufgerufen werden:

```
port.A
```

Konstanten

hub.port Konstanten

A = 0
Der Port (Anschluss) am Hub, der mit „A“ gekennzeichnet ist.
B = 1
Der Port (Anschluss) am Hub, der mit „B“ gekennzeichnet ist.
C = 2
Der Port (Anschluss) am Hub, der mit „C“ gekennzeichnet ist.
D = 3
Der Port (Anschluss) am Hub, der mit „D“ gekennzeichnet ist.
E = 4
Der Port (Anschluss) am Hub, der mit „E“ gekennzeichnet ist.
F = 5
Der Port (Anschluss) am Hub, der mit „F“ gekennzeichnet ist.

Speaker (Lautsprecher)

Um das Modul „Speaker“ (Lautsprecher) zu verwenden, musst du deinem Projekt folgende import-Anweisung hinzufügen:

```
from hub import sound
```
Alle Funktionen im Modul sollten innerhalb des Moduls `sound` (Motor) als Präfix wie folgt aufgerufen werden:

```
sound.stop()
```

Funktionen

beep

beep(frequency: int = 440, duration: int = 500, volume: int = 100, *, attack: int = 0, decay: int = 0, sustain: int = 100, release: int = 0, transition: int = 10, waveform: int = WAVEFORM_SINE, channel: int = DEFAULT) -> Awaitable

Lässt am Hub einen Piepton ertönen

Parameter

frequency: int

Die abzuspielende Frequenz

duration: int

Dauer in Millisekunden

volume: int

Die Lautstärke (0-100)

Optionale Schlüsselwortargumente:

attack: int

Die Dauer des Hochfahrens von Null bis zum Spitzenwert ab dem Zeitpunkt, wenn die Taste gedrückt wird.

decay: int

Die Dauer des anschließenden Rundowns vom Attack-Pegel auf den vorgesehenen Sustain-Pegel.

sustain: int

Der Pegel während der Hauptsequenz der Tondauer, bis die Taste losgelassen wird.

release: int

Die Dauer, bis der Pegel nach dem Loslassen der Taste vom Sustain-Pegel auf Null abfällt

transition: int

Zeit in Millisekunden bis zum Übergang in den Ton, wenn bereits etwas im Kanal abgespielt wird

waveform: int

Die synthetisierte Wellenform. Verwende eine der Konstanten im Modul `hub.sound`.

channel: int

Der fürs Abspielen gewünschte Kanal. Die Optionen sind `sound.DEFAULT` und `sound.ANY`

stop

stop() -> None

Stoppt alle Geräusche aus dem Hub

Parameter

volume

volume(volume: int) -> None

Parameter

volume: int

Die Lautstärke (0-100)

Konstanten

hub.sound Konstanten

ANY = -2

DEFAULT = -1

WAVEFORM_SINE = 1

WAVEFORM_SAWTOOTH = 3

WAVEFORM_SQUARE = 2

WAVEFORM_TRIANGLE = 1

Funktionen

device_uuid

device_uuid() -> str

Ruf die Gerätekennung ab.

Parameter

hardware_id

hardware_id() -> str

Ruf die Hardware-Kennung ab

Parameter

power_off

power_off() -> int

Schaltet den Hub aus.

Parameter

temperature

temperature() -> int

Ruf die Hub-Temperatur ab. Gemessen in Dezigrad Celsius (d°C), was 1/10 Grad Celsius (°C) entspricht

Parameter

Motor

Um einen Motor zu verwenden, musst du deinem Projekt die folgende import-Anweisung hinzufügen:

```
import motor
```

Alle Funktionen im Modul sollten innerhalb des Moduls `motor` (Motor) als Präfix wie folgt aufgerufen werden:

```
motor.run(port.A, 1000)
```

Funktionen

absolute_position

absolute_position(port: int) -> int

Ruf die absolute Position eines Motors ab

Parameter

port: int

Ein Anschluss aus dem Untermodul `port` (Anschluss) im Modul `hub` (Hub)

get_duty_cycle

get_duty_cycle(port: int) -> int

Ruf die PWM eines Motors ab

Parameter

port: int

Ein Anschluss aus dem Untermodul `port` (Anschluss) im Modul `hub` (Hub)

relative_position

relative_position(port: int) -> int

Ruf die relative Position eines Motors ab

Parameter

port: int

Ein Anschluss aus dem Untermodul `port` (Anschluss) im Modul `hub` (Hub)

reset_relative_position

reset_relative_position(port: int, position: int) -> None

Ändere die Position, die als Offset dient, wenn die Funktion `run_to_relative_position` verwendet wird.

Parameter

port: int

Ein Anschluss aus dem Untermodul `port` (Anschluss) im Modul `hub` (Hub)

position: int

Der Gradzahl (Position) des Motors

run

run(port: int, velocity: int, *, acceleration: int = 1000) -> None

Starte einen Motor mit konstanter Drehzahl

```
from hub import port
import motor, time

# Starte den Motor
motor.run(port.A, 1000)
```

Parameter

port: int

Ein Anschluss aus dem Untermodul `port` (Anschluss) im Modul `hub` (Hub)

velocity: int

Die Geschwindigkeit in Grad/s

Die Wertebereiche hängen vom Motortyp ab.

Kleiner Motor (unverzichtbar): -660 bis 660
Mittlerer Motor: -1110 bis 1110
Großer Motor: -1050 bis 1050

Optionale Schlüsselwortargumente:

acceleration: int

Die Beschleunigung (Grad/s²) (0 - 10000)

run_for_degrees

run_for_degrees(port: int, degrees: int, velocity: int, *, stop: int = BRAKE, acceleration: int = 1000, deceleration: int = 1000) -> Awaitable

Lass einen Motor eine bestimmte Anzahl an Umdrehungen drehen
Wenn die awaited-Funktion einen Bewegungsstatus zurückgibt, der einer der folgenden Konstanten entspricht:

```
motor.READY
motor.RUNNING
motor.STALLED
motor.CANCELED
motor.ERROR
motor.DISCONNECTED
```

Parameter

port: int

Ein Anschluss aus dem Untermodul `port` (Anschluss) im Modul `hub` (Hub)

degrees: int

Die Anzahl an Grad

velocity: int

Die Geschwindigkeit in Grad/s

Die Wertebereiche hängen vom Motortyp ab.

Kleiner Motor (unverzichtbar): -660 bis 660
Mittlerer Motor: -1110 bis 1110
Großer Motor: -1050 bis 1050

Optionale Schlüsselwortargumente:

stop: int

Das Verhalten des Motors, nachdem er gestoppt wurde. Verwende die Konstanten im Modul `motor` (Motor).

Folgende Werte sind möglich:
`motor.COAST`, um den Motor bis zum Stillstand auslaufen zu lassen
`motor.BREAK`, um zu bremsen und nach dem Anhalten noch weiter zu bremsen
`motor.HOLD`, um dem Motor mitzuteilen, dass er seine Position halten soll
`motor.CONTINUE`, um dem Motor mitzuteilen, dass er mit seiner aktuellen Geschwindigkeit weiterlaufen soll, bis er einen weiteren Befehl erhält
`motor.SMART_COAST`, um den Motor bis zum Stillstand bremsen zu lassen und dann auszulaufen und Ungenauigkeiten im nächsten Befehl auszugleichen
`motor.SMART_BRAKE`, um den Motor bremsen zu lassen und nach dem Anhalten weiter zu bremsen und Ungenauigkeiten im nächsten Befehl auszugleichen

acceleration: int

Die Beschleunigung (Grad/s²) (0 - 10000)

deceleration: int

Die Verzögerung (Grad/s²) (0 - 10000)

run_for_time

run_for_time(port: int, duration: int, velocity: int, *, stop: int = BRAKE, acceleration: int = 1000, deceleration: int = 1000) -> Awaitable

Lass einen Motor eine begrenzte Zeit laufen
Wenn die await-Funktion einen Bewegungsstatus zurückgibt, der einer der folgenden Konstanten entspricht:

`motor.READY`
`motor.RUNNING`
`motor.STALLED`
`motor.ERROR`
`motor.DISCONNECTED`

```
from hub import port
import runloop
import motor

async def main():
```

```
    # 1 Sekunde lang mit Geschwindigkeit 1000 laufen lassen
    await motor.run_for_time(port.A, 1000, 1000)

    # 1 Sekunde lang mit Geschwindigkeit 280 laufen lassen
    await motor_pair.run_for_time(port.A, 1000, 280)

    # 10 Sekunden lang mit Geschwindigkeit 280 laufen und
allmählich langsamer werden lassen
    await motor_pair.run_for_time(port.A, 10000, 280,
deceleration=10)

runloop.run(main())
```

Parameter

port: int

Ein Anschluss aus dem Untermodul `port` (Anschluss) im Modul `hub` (Hub)

duration: int

Dauer in Millisekunden

velocity: int

Die Geschwindigkeit in Grad/s

Die Wertebereiche hängen vom Motortyp ab.

Kleiner Motor (unverzichtbar): -660 bis 660
Mittlerer Motor: -1110 bis 1110
Großer Motor: -1050 bis 1050

Optionale Schlüsselwortargumente:

stop: int

Das Verhalten des Motors, nachdem er gestoppt wurde. Verwende die Konstanten im Modul `motor` (Motor).

Folgende Werte sind möglich:
`motor.COAST`, um den Motor bis zum Stillstand auslaufen zu lassen
`motor.BREAK`, um zu bremsen und nach dem Anhalten noch weiter zu

bremsen
`motor.HOLD`, um dem Motor mitzuteilen, dass er seine Position halten soll
`motor.CONTINUE`, um dem Motor mitzuteilen, dass er mit seiner aktuellen Geschwindigkeit weiterlaufen soll, bis er einen weiteren Befehl erhält
`motor.SMART_COAST`, um den Motor bis zum Stillstand bremsen zu lassen und dann auszulaufen und Ungenauigkeiten im nächsten Befehl auszugleichen
`motor.SMART_BRAKE`, um den Motor bremsen zu lassen und nach dem Anhalten weiter zu bremsen und Ungenauigkeiten im nächsten Befehl auszugleichen

acceleration: int

Die Beschleunigung (Grad/s²) (0 - 10000)

deceleration: int

Die Verzögerung (Grad/s²) (0 - 10000)

run_to_absolute_position

run_to_absolute_position(port: int, position: int, velocity: int, *, direction: int, stop: int = BRAKE, acceleration: int = 1000, deceleration: int = 1000) -> Awaitable

Lass einen Motor zu einer absoluten Position drehen
Wenn die await-Funktion einen Bewegungsstatus zurückgibt, der einer der folgenden Konstanten entspricht:

```
motor.READY
motor.RUNNING
motor.STALLED
motor.CANCELED
motor.ERROR
motor.DISCONNECTED
```

Parameter

port: int

Ein Anschluss aus dem Untermodul `port` (Anschluss) im Modul `hub` (Hub)

position: int

Der Gradzahl (Position) des Motors

velocity: int

Die Geschwindigkeit in Grad/s

Die Wertebereiche hängen vom Motortyp ab.

Kleiner Motor (unverzichtbar): -660 bis 660
Mittlerer Motor: -1110 bis 1110
Großer Motor: -1050 bis 1050

Optionale Schlüsselwortargumente:

direction: int

Die gewünschte Drehrichtung:
Folgende Optionen sind verfügbar:

```
motor.CLOCKWISE
motor.COUNTERCLOCKWISE
motor.SHORTEST_PATH
motor.LONGEST_PATH
```

stop: int

Das Verhalten des Motors, nachdem er gestoppt wurde. Verwende die Konstanten im Modul `motor` (Motor).

Folgende Werte sind möglich:
`motor.COAST`, um den Motor bis zum Stillstand auslaufen zu lassen
`motor.BREAK`, um zu bremsen und nach dem Anhalten noch weiter zu bremsen
`motor.HOLD`, um dem Motor mitzuteilen, dass er seine Position halten soll
`motor.CONTINUE`, um dem Motor mitzuteilen, dass er mit seiner aktuellen Geschwindigkeit weiterlaufen soll, bis er einen weiteren Befehl erhält
`motor.SMART_COAST`, um den Motor bis zum Stillstand bremsen zu lassen und dann auszulaufen und Ungenauigkeiten im nächsten Befehl auszugleichen
`motor.SMART_BRAKE`, um den Motor bremsen zu lassen und nach dem Anhalten weiter zu bremsen und Ungenauigkeiten im nächsten Befehl auszugleichen

acceleration: int

Die Beschleunigung (Grad/s²) (0 - 10000)

deceleration: int

Die Verzögerung (Grad/s²) (0 - 10000)

run_to_relative_position

run_to_relative_position(port: int, position: int, velocity: int, *, stop: int = BRAKE, acceleration: int = 1000, deceleration: int = 1000) -> Awaitable

Lass einen Motor zu einer Position relativ zur aktuellen Position drehen
Wenn die await-Funktion einen Bewegungsstatus zurückgibt, der einer der folgenden Konstanten entspricht:

```
motor.READY
motor.RUNNING
motor.STALLED
motor.CANCELED
motor.ERROR
motor.DISCONNECTED
```

Parameter

port: int

Ein Anschluss aus dem Untermodul `port` (Anschluss) im Modul `hub` (Hub)

position: int

Der Gradzahl (Position) des Motors

velocity: int

Die Geschwindigkeit in Grad/s

Die Wertebereiche hängen vom Motortyp ab.

Kleiner Motor (unverzichtbar): -660 bis 660
Mittlerer Motor: -1110 bis 1110
Großer Motor: -1050 bis 1050

Optionale Schlüsselwortargumente:

stop: int

Das Verhalten des Motors, nachdem er gestoppt wurde. Verwende die Konstanten im Modul `motor` (Motor).

Folgende Werte sind möglich:
`motor.COAST`, um den Motor bis zum Stillstand auslaufen zu lassen
`motor.BREAK`, um zu bremsen und nach dem Anhalten noch weiter zu bremsen
`motor.HOLD`, um dem Motor mitzuteilen, dass er seine Position halten soll
`motor.CONTINUE`, um dem Motor mitzuteilen, dass er mit seiner aktuellen Geschwindigkeit weiterlaufen soll, bis er einen weiteren Befehl erhält
`motor.SMART_COAST`, um den Motor bis zum Stillstand bremsen zu lassen und dann auszulaufen und Ungenauigkeiten im nächsten Befehl auszugleichen
`motor.SMART_BRAKE`, um den Motor bremsen zu lassen und nach dem Anhalten weiter zu bremsen und Ungenauigkeiten im nächsten Befehl auszugleichen

acceleration: int

Die Beschleunigung (Grad/s²) (0 - 10000)

deceleration: int

Die Verzögerung (Grad/s²) (0 - 10000)

set_duty_cycle

set_duty_cycle(port: int, pwm: int) -> None

Starte einen Motor mit einer bestimmten PWM

Parameter

port: int

Ein Anschluss aus dem Untermodul `port` (Anschluss) im Modul `hub` (Hub)

pwm: int

Der PWM-Wert (-10000-10000)

stop

stop(port: int, *, stop: int = BRAKE) -> None

Stoppt einen Motor

```
from hub import port
import motor, time

# Starte den Motor
motor.run(port.A, 1000)

# Warte 2 Sekunden
time.sleep_ms(2000)

# Stoppe den Motor
motor.stop(port.A)
```

Parameter

port: int

Ein Anschluss aus dem Untermodul `port` (Anschluss) im Modul `hub` (Hub)

Optionale Schlüsselwortargumente:

stop: int

Das Verhalten des Motors, nachdem er gestoppt wurde. Verwende die Konstanten im Modul `motor` (Motor).

Folgende Werte sind möglich:
`motor.COAST`, um den Motor bis zum Stillstand auslaufen zu lassen

`motor.BREAK`, um zu bremsen und nach dem Anhalten noch weiter zu bremsen
`motor.HOLD`, um dem Motor mitzuteilen, dass er seine Position halten soll
`motor.CONTINUE`, um dem Motor mitzuteilen, dass er mit seiner aktuellen Geschwindigkeit weiterlaufen soll, bis er einen weiteren Befehl erhält
`motor.SMART_COAST`, um den Motor bis zum Stillstand bremsen zu lassen und dann auszulaufen und Ungenauigkeiten im nächsten Befehl auszugleichen
`motor.SMART_BRAKE`, um den Motor bremsen zu lassen und nach dem Anhalten weiter zu bremsen und Ungenauigkeiten im nächsten Befehl auszugleichen

velocity

velocity(port: int) -> int

Ruf die Geschwindigkeit (Grad/s) eines Motors ab

Parameter

port: int

Ein Anschluss aus dem Untermodul `port` (Anschluss) im Modul `hub` (Hub)

Konstanten

motor Konstanten

READY = 1

RUNNING = 2

STALLED = -1

CANCELED = -2

ERROR = -3

DISCONNECTED = 0

COAST = 1

BRAKE = 2

HOLD = 3

CONTINUE = 0

SMART_COAST = 4

SMART_BRAKE = 5

CLOCKWISE = 0

COUNTERCLOCKWISE = 1

SHORTEST_PATH = 2

LONGEST_PATH = 3

Motor-Paar

Das Modul `motor_pair` (Motor-Paar) wird verwendet, um zwei Motoren synchronisiert laufen zu lassen. Dieses Modul ist die optimale Lösung, wenn zwei Motoren gleichzeitig starten und stoppen sollen.

Um das Modul `motor_pair` (Motor-Paar) zu verwenden, musst du es nur folgendermaßen importieren:

```
import motor_pair
```

Alle Funktionen im Modul sollten innerhalb des Moduls `motor_pair` (Motor-Paar) als Präfix wie folgt aufgerufen werden:

```
motor_pair.pair(motor_pair.PAIR_1, port.A, port.B)
```

Funktionen

move

move(pair: int, steering: int, *, velocity: int = 360, acceleration: int = 1000) -> None

Lass ein Motor-Paar mit konstanter Drehzahl laufen, bis ein neuer Befehl erteilt wird.

```
from hub import port
import runloop
```

```
import motor_pair

async def main():
    # Kopple zwei Motoren an den Anschlüssen A und B
    motor_pair.pair(motor_pair.PAIR_1, port.A, port.B)

    await runloop.sleep_ms(2000)

    # Starte die Bewegung mit Standardgeschwindigkeit
    motor_pair.move(motor_pair.PAIR_1, 0)

    await runloop.sleep_ms(2000)

    # Lass das Fahrzeug mit einer bestimmten Geschwindigkeit
geradeaus fahren
    motor_pair.move(motor_pair.PAIR_1, 0, velocity=280)

    await runloop.sleep_ms(2000)

    # Lass das Fahrzeug mit einer bestimmten Geschwindigkeit und
Beschleunigung geradeaus fahren
    motor_pair.move(motor_pair.PAIR_1, 0, velocity=280,
acceleration=100)

runloop.run(main())
```

Parameter

pair: int

Der doppelte Steckplatz des Motor-Paars.

steering: int

Die Lenkung (-100 bis 100)

Optionale Schlüsselwortargumente:

velocity: int

Die Geschwindigkeit in Grad/s

Die Wertebereiche hängen vom Motortyp ab.

Kleiner Motor (unverzichtbar): -660 bis 660
Mittlerer Motor: -1110 bis 1110
Großer Motor: -1050 bis 1050

acceleration: int

Die Beschleunigung (Grad/s²) (0 - 10000)

move_for_degrees

move_for_degrees(pair: int, degrees: int, steering: int, *, velocity: int = 360, stop: int = motor.BRAKE, acceleration: int = 1000, deceleration: int = 1000) -> Awaitable

Lass ein Motor-Paar eine bestimmte Anzahl von Umdrehungen mit konstanter Drehzahl laufen.
Wenn die awaited-Funktion einen Bewegungsstatus zurückgibt, der einer der folgenden Konstanten aus dem Modul „Motor“ entspricht:

```
motor.READY
motor.RUNNING
motor.STALLED
motor.CANCELED
motor.ERROR
motor.DISCONNECTED
```

```
from hub import port
import runloop
import motor_pair

async def main():
    # Kopple zwei Motoren an den Anschlüssen A und B
    motor_pair.pair(motor_pair.PAIR_1, port.A, port.B)

    # Lass die beiden Motoren mit Standardgeschwindigkeit um 90
Grad drehen und das Fahrzeug dabei geradeaus fahren
    await motor_pair.move_for_degrees(motor_pair.PAIR_1, 90, 0)

    # Lass das Fahrzeug mit einer bestimmten Geschwindigkeit
geradeaus fahren
    await motor_pair.move_for_degrees(motor_pair.PAIR_1, 360, 0,
velocity=280)

    # Lass das Fahrzeug mit einer bestimmten Geschwindigkeit
geradeaus fahren und allmählich langsamer werden
```

```
    await motor_pair.move_for_degrees(motor_pair.PAIR_1, 360, 0,
velocity=280, deceleration=10)

runloop.run(main())
```

Parameter

pair: int

Der doppelte Steckplatz des Motor-Paars.

degrees: int

Die Anzahl an Grad

steering: int

Die Lenkung (-100 bis 100)

Optionale Schlüsselwortargumente:

velocity: int

Die Geschwindigkeit in Grad/s

Die Wertebereiche hängen vom Motortyp ab.

Kleiner Motor (unverzichtbar): -660 bis 660
Mittlerer Motor: -1110 bis 1110
Großer Motor: -1050 bis 1050

stop: int

Das Verhalten des Motors, nachdem er gestoppt wurde. Verwende die Konstanten im Modul `motor` (Motor).

Folgende Werte sind möglich:
`motor.COAST`, um den Motor bis zum Stillstand auslaufen zu lassen
`motor.BREAK`, um zu bremsen und nach dem Anhalten noch weiter zu bremsen
`motor.HOLD`, um dem Motor mitzuteilen, dass er seine Position halten soll
`motor.CONTINUE`, um dem Motor mitzuteilen, dass er mit seiner aktuellen

Geschwindigkeit weiterlaufen soll, bis er einen weiteren Befehl erhält `motor.SMART_COAST`, um den Motor bis zum Stillstand bremsen zu lassen und dann auszulaufen und Ungenauigkeiten im nächsten Befehl auszugleichen `motor.SMART_BRAKE`, um den Motor bremsen zu lassen und nach dem Anhalten weiter zu bremsen und Ungenauigkeiten im nächsten Befehl auszugleichen

acceleration: int

Die Beschleunigung (Grad/s²) (0 - 10000)

deceleration: int

Die Verzögerung (Grad/s²) (0 - 10000)

move_for_time

move_for_time(pair: int, duration: int, steering: int, *, velocity: int = 360, stop: int = motor.BRAKE, acceleration: int = 1000, deceleration: int = 1000) -> Awaitable

Lass ein Motor-Paar eine bestimmte Dauer mit konstanter Drehzahl laufen. Wenn die await-Funktion einen Bewegungsstatus zurückgibt, der einer der folgenden Konstanten aus dem Modul „Motor“ entspricht:

```
motor.READY
motor.RUNNING
motor.STALLED
motor.CANCELED
motor.ERROR
motor.DISCONNECTED
```

```
from hub import port
import runloop
import motor_pair

async def main():
    # Kopple zwei Motoren an den Anschlüssen A und B
    motor_pair.pair(motor_pair.PAIR_1, port.A, port.B)

    # Lass das Fahrzeug 1 Sekunde lang mit Standardgeschwindigkeit
geradeaus fahren
    await motor_pair.move_for_time(motor_pair.PAIR_1, 1000, 0)
```

```
    # Lass das Fahrzeug 1 Sekunde lang mit einer bestimmten
Geschwindigkeit geradeaus fahren
    await motor_pair.move_for_time(motor_pair.PAIR_1, 1000, 0,
velocity=280)

    # Lass das Fahrzeug 10 Sekunden lang mit einer bestimmten
Geschwindigkeit geradeaus fahren und allmählich langsamer werden
    await motor_pair.move_for_time(motor_pair.PAIR_1, 10000, 0,
velocity=280, deceleration=10)

runloop.run(main())
```

Parameter

pair: int

Der doppelte Steckplatz des Motor-Paars.

duration: int

Dauer in Millisekunden

steering: int

Die Lenkung (-100 bis 100)

Optionale Schlüsselwortargumente:

velocity: int

Die Geschwindigkeit in Grad/s

Die Wertebereiche hängen vom Motortyp ab.

Kleiner Motor (unverzichtbar): -660 bis 660
Mittlerer Motor: -1110 bis 1110
Großer Motor: -1050 bis 1050

stop: int

Das Verhalten des Motors, nachdem er gestoppt wurde. Verwende die Konstanten im Modul `motor` (Motor).

Folgende Werte sind möglich:
`motor.COAST`, um den Motor bis zum Stillstand auslaufen zu lassen
`motor.BREAK`, um zu bremsen und nach dem Anhalten noch weiter zu bremsen
`motor.HOLD`, um dem Motor mitzuteilen, dass er seine Position halten soll
`motor.CONTINUE`, um dem Motor mitzuteilen, dass er mit seiner aktuellen Geschwindigkeit weiterlaufen soll, bis er einen weiteren Befehl erhält
`motor.SMART_COAST`, um den Motor bis zum Stillstand bremsen zu lassen und dann auszulaufen und Ungenauigkeiten im nächsten Befehl auszugleichen
`motor.SMART_BRAKE`, um den Motor bremsen zu lassen und nach dem Anhalten weiter zu bremsen und Ungenauigkeiten im nächsten Befehl auszugleichen

acceleration: int

Die Beschleunigung (Grad/s²) (0 - 10000)

deceleration: int

Die Verzögerung (Grad/s²) (0 - 10000)

move_tank

move_tank(pair: int, left_velocity: int, right_velocity: int, *, acceleration: int = 1000) -> None

Lass ein Motor-Paar mit konstanter Drehzahl gegenläufig laufen, um das Fahrzeug auf der Stelle drehen zu lassen (Tank Move), bis ein neuer Befehl erteilt wird.

```
from hub import port
import runloop
import motor_pair

async def main():
    # Kopple zwei Motoren an den Anschlüssen A und B
    motor_pair.pair(motor_pair.PAIR_1, port.A, port.B)

    # Starte die Bewegung mit Standardgeschwindigkeit
    motor_pair.move_tank(motor_pair.PAIR_1, 1000, 1000)

    await runloop.sleep_ms(2000)

    # Lass das Fahrzeug nach rechts abbiegen
```

```
    motor_pair.move_tank(motor_pair.PAIR_1, 0, 1000)

    await runloop.sleep_ms(2000)

    # Lass das Fahrzeug wie einen Panzer auf der Stelle wenden
(Tank Move)
    motor_pair.move_tank(motor_pair.PAIR_1, 1000, -1000)

runloop.run(main())
```

Parameter

pair: int

Der doppelte Steckplatz des Motor-Paars.

left_velocity: int

Die Geschwindigkeit (Grad/s) des linken Motors.

right_velocity: int

Die Geschwindigkeit (Grad/s) des rechten Motors.

Optionale Schlüsselwortargumente:

acceleration: int

Die Beschleunigung (Grad/s²) (0 - 10000)

move_tank_for_degrees

move_tank_for_degrees(pair: int, degrees: int, left_velocity: int, right_velocity: int, *, stop: int = motor.BRAKE, acceleration: int = 1000, deceleration: int = 1000) -> Awaitable

Lass ein Motor-Paar mit konstanter Drehzahl gegenläufig laufen, um das Fahrzeug auf der Stelle drehen zu lassen (Tank Move), bis ein neuer Befehl erteilt wird.
Wenn die await-Funktion einen Bewegungsstatus zurückgibt, der einer der folgenden Konstanten aus dem Modul „Motor“ entspricht:

```
motor.READY
motor.RUNNING
motor.STALLED
motor.CANCELED
motor.ERROR
motor.DISCONNECTED
```

```
from hub import port
import runloop
import motor_pair

async def main():
    # Kopple zwei Motoren an den Anschlüssen A und B
    motor_pair.pair(motor_pair.PAIR_1, port.A, port.B)

    # Lass die Motoren mit Standardgeschwindigkeit um 360 Grad
drehen und das Fahrzeug dabei geradeaus fahren
    await motor_pair.move_tank_for_degrees(motor_pair.PAIR_1, 360,
1000, 1000)

    # Lass das Fahrzeug eine Rechtsdrehung um 180 Grad ausführen
    await motor_pair.move_tank_for_degrees(motor_pair.PAIR_1, 180,
0, 1000)

    # Lass das Fahrzeug wie einen Panzer um 720 Grad auf der
Stelle wenden (Tank Move)
    await motor_pair.move_tank_for_degrees(motor_pair.PAIR_1, 720,
1000, -1000)

runloop.run(main())
```

Parameter

pair: int

Der doppelte Steckplatz des Motor-Paars.

degrees: int

Die Anzahl an Grad

left_velocity: int

Die Geschwindigkeit (Grad/s) des linken Motors.

right_velocity: int

Die Geschwindigkeit (Grad/s) des rechten Motors.

Optionale Schlüsselwortargumente:

stop: int

Das Verhalten des Motors, nachdem er gestoppt wurde. Verwende die Konstanten im Modul `motor` (Motor).

Folgende Werte sind möglich:
`motor.COAST`, um den Motor bis zum Stillstand auslaufen zu lassen
`motor.BREAK`, um zu bremsen und nach dem Anhalten noch weiter zu bremsen
`motor.HOLD`, um dem Motor mitzuteilen, dass er seine Position halten soll
`motor.CONTINUE`, um dem Motor mitzuteilen, dass er mit seiner aktuellen Geschwindigkeit weiterlaufen soll, bis er einen weiteren Befehl erhält
`motor.SMART_COAST`, um den Motor bis zum Stillstand bremsen zu lassen und dann auszulaufen und Ungenauigkeiten im nächsten Befehl auszugleichen
`motor.SMART_BRAKE`, um den Motor bremsen zu lassen und nach dem Anhalten weiter zu bremsen und Ungenauigkeiten im nächsten Befehl auszugleichen

acceleration: int

Die Beschleunigung (Grad/s²) (0 - 10000)

deceleration: int

Die Verzögerung (Grad/s²) (0 - 10000)

move_tank_for_time

move_tank_for_time(pair: int, left_velocity: int, right_velocity: int, duration: int, *, stop: int = motor.BRAKE, acceleration: int = 1000, deceleration: int = 1000) -> Awaitable

Lass ein Motor-Paar eine bestimmte Dauer mit konstanter Drehzahl gegenläufig laufen, um das Fahrzeug auf der Stelle drehen zu lassen (Tank Move).

Wenn die await-Funktion einen Bewegungsstatus zurückgibt, der einer der folgenden Konstanten aus dem Modul „Motor“ entspricht:

```
motor.READY
motor.RUNNING
motor.STALLED
motor.CANCELLED
motor.ERROR
motor.DISCONNECTED
```

```
from hub import port
import runloop
import motor_pair

async def main():
    # Kopple zwei Motoren an den Anschlüssen A und B
    motor_pair.pair(motor_pair.PAIR_1, port.A, port.B)

    # Lass das Fahrzeug 1 Sekunde lang mit Standardgeschwindigkeit
geradeaus fahren
    await motor_pair.move_tank_for_time(motor_pair.PAIR_1, 1000,
1000, 1000)

    # Lass das Fahrzeug 3 Sekunden lang nach rechts drehen
    await motor_pair.move_tank_for_time(motor_pair.PAIR_1, 0,
1000, 3000)

    # Lass das Fahrzeug wie einen Panzer 2 Sekunden lang auf der
Stelle drehen (Tank Move)
    await motor_pair.move_tank_for_time(motor_pair.PAIR_1, 1000,
-1000, 2000)

runloop.run(main())
```

Parameter

pair: int

Der doppelte Steckplatz des Motor-Paars.

duration: int

Dauer in Millisekunden

left_velocity: int

Die Geschwindigkeit (Grad/s) des linken Motors.

right_velocity: int

Die Geschwindigkeit (Grad/s) des rechten Motors.

Optionale Schlüsselwortargumente:

stop: int

Das Verhalten des Motors, nachdem er gestoppt wurde. Verwende die Konstanten im Modul `motor` (Motor).

Folgende Werte sind möglich:
`motor.COAST`, um den Motor bis zum Stillstand auslaufen zu lassen
`motor.BREAK`, um zu bremsen und nach dem Anhalten noch weiter zu bremsen
`motor.HOLD`, um dem Motor mitzuteilen, dass er seine Position halten soll
`motor.CONTINUE`, um dem Motor mitzuteilen, dass er mit seiner aktuellen Geschwindigkeit weiterlaufen soll, bis er einen weiteren Befehl erhält
`motor.SMART_COAST`, um den Motor bis zum Stillstand bremsen zu lassen und dann auszulaufen und Ungenauigkeiten im nächsten Befehl auszugleichen
`motor.SMART_BRAKE`, um den Motor bremsen zu lassen und nach dem Anhalten weiter zu bremsen und Ungenauigkeiten im nächsten Befehl auszugleichen

acceleration: int

Die Beschleunigung (Grad/s²) (0 - 10000)

deceleration: int

Die Verzögerung (Grad/s²) (0 - 10000)

pair

pair(pair: int, left_motor: int, right_motor: int) -> None

Kopple zwei Motoren (`left_motor` & `right_motor`) und speichere die gekoppelten Motoren in einem `pair` (Paar):

Verwende die Option `pair` (Paar) in allen nachfolgenden Aufrufen der Funktion `motor_pair` (Motor-Paar).

```
import motor_pair
from hub import port

motor_pair.pair(motor_pair.PAIR_1, port.A, port.B)
```

Parameter

pair: int

Der doppelte Steckplatz des Motor-Paars.

left_motor: int

Der Port (Anschluss) am linken Motor. Verwende das Untermodul `port` (Anschluss) im Modus `hub` (Hub).

right_motor: int

Der Anschluss am rechten Motor Verwende das Untermodul `port` (Anschluss) im Modus `hub` (Hub).

stop

stop(pair: int, *, stop: int = motor.BRAKE) -> None

Stoppt ein Motor-Paar.

```
import motor_pair

motor_pair.stop(motor_pair.PAIR_1)
```

Parameter

pair: int

Der doppelte Steckplatz des Motor-Paars.

Optionale Schlüsselwortargumente:

stop: int

Das Verhalten des Motors, nachdem er gestoppt wurde. Verwende die Konstanten im Modul `motor` (Motor).

Folgende Werte sind möglich:
`motor.COAST`, um den Motor bis zum Stillstand auslaufen zu lassen
`motor.BREAK`, um zu bremsen und nach dem Anhalten noch weiter zu bremsen
`motor.HOLD`, um dem Motor mitzuteilen, dass er seine Position halten soll
`motor.CONTINUE`, um dem Motor mitzuteilen, dass er mit seiner aktuellen Geschwindigkeit weiterlaufen soll, bis er einen weiteren Befehl erhält
`motor.SMART_COAST`, um den Motor bis zum Stillstand bremsen zu lassen und dann auszulaufen und Ungenauigkeiten im nächsten Befehl auszugleichen
`motor.SMART_BRAKE`, um den Motor bremsen zu lassen und nach dem Anhalten weiter zu bremsen und Ungenauigkeiten im nächsten Befehl auszugleichen

unpair

unpair(pair: int) -> None

Entkopple ein Motor-Paar.

```
import motor_pair

motor_pair.unpair(motor_pair.PAIR_1)
```

Parameter

pair: int

Der doppelte Steckplatz des Motor-Paars.

Konstanten

motor_pair Konstanten

PAIR_1 = 0
Erstes Motor-Paar
PAIR_2 = 1
Zweites Motor-Paar
PAIR_3 = 2
Drittes Motor-Paar

Orientation (Ausrichtung)

Das Modul `orientation` (Ausrichtung) beinhaltet alle Ausrichtungskonstanten, die mit dem Modul `light_matrix` (Lichtmatrix) verwendet werden können.

Um das Modul „Orientation“ (Ausrichtung) zu verwenden, musst du die folgende import-Anweisung zu deinem Projekt hinzufügen:

```
import orientation
```

Konstanten

orientation Konstanten

UP = 0

RIGHT = 1

DOWN = 2

LEFT = 3

Runloop (run-Schleife)

Das Modul `runloop` (run-Schleife) enthält alle Funktionen und Konstanten, um die run-Schleife zu verwenden.

Um das Modul „Runloop“ (run-Schleife) zu verwenden, musst du deinem Projekt folgende import-Anweisung hinzufügen:

```
import runloop
```

Alle Funktionen im Modul sollten innerhalb des Moduls `runloop` (run-Schleife) als Präfix wie folgt aufgerufen werden:

```
runloop.run(some_async_function())
```

Funktionen

run

run(*functions: Awaitable) -> None

Starte beliebig viele parallele asynchrone Funktionen vom Typ `async`. Diese Funktion solltest du zur Erstellung von Programmen verwenden, die ähnlich wie Textblöcke strukturiert sind.

Parameter

*functions: awaitable

The functions to run

sleep_ms

sleep_ms(duration: int) -> Awaitable

Unterbrich die Ausführung der Anwendung für eine beliebige Anzahl von Millisekunden.

```
from hub import light_matrix
import runloop

async def main():
    light_matrix.write("Hi!")
    # Warte zehn Sekunden
    await runloop.sleep_ms(10000)
    light_matrix.write("Are you still here?")
```

```
runloop.run(main())
```

Parameter

duration: int

Dauer in Millisekunden

until

until(function: Callable[[], bool], timeout: int = 0) -> Awaitable

Gibt ein Awaitable zurück, wenn die Bedingung in der abgelaufenen Funktion oder Lambda `True` (wahr) oder die Zeit abgelaufen ist.

```
import color_sensor
import color
from hub import port
import runloop

def is_color_red():
    return color_sensor.color(port.A) is color.RED

async def main():
    # Warte, bis der Farbsensor die Farbe Rot erkennt
    await runloop.until(is_color_red)
    print("Red!")

runloop.run(main())
```

Parameter

function: Callable[[], bool]

Ein aufrufbares Element ohne Parameter, das entweder `True` (wahr) oder `False` (falsch) zurückgibt.
Ein Callable ist jedes aufrufbare Element, also eine `def` oder ein `lambda`

timeout: int

Ein Timeout für die Funktion in Millisekunden.
Wenn das aufrufbare Element nicht innerhalb des Timeouts den Wert `True` (wahr) zurückgibt, wird `until` nach dem Timeout trotzdem aufgelöst.
0 bedeutet kein Timeout. In diesem Fall wird es erst aufgelöst, wenn das aufrufbare Element `True` (wahr) zurückgibt.

```
1
2

3
4
```

```
from hub import light_matrix

light_matrix.write('Hello, World!')
```


```
from hub import light_matrix

light_matrix.write('Hello, World!')
```


14:55:17: Compiled

In den vorangegangenen Kapiteln hast du

- die Grundlagen gelernt, wie man Python mit SPIKE Prime verwendet, und wie man die Lichtmatrix, das Licht, den Lautsprecher und die Tasten des Hubs und zudem die Motoren, den Farbsensor und den Kraftsensor benutzt.
- dich mit regulären und asynchronen Funktionen, lokalen und globalen Variablen sowie mit Datentypen wie `int`, `bool`, `str`, `tuple` und `list` vertraut gemacht.
- die `for`- und `while`-Schleifen sowie die `if`/`elif`/`else`-Anweisungen verwendet, um den Ablauf deines Programms zu steuern.
- gelernt, wie du Kommentare in deinem Programm verwendest und wie du Fehler in deinem Programm behebst (Debugging), wenn etwas schiefgeht.

Das ist eine beachtliche Leistung, auf die du zurecht sehr stolz sein darfst.

Es gibt noch weitere Informationsquellen, auf die du zugreifen kannst, um noch mehr über die Verwendung von Python mit SPIKE Prime zu erfahren.

Python-Benutzerhandbuch

Der Abschnitt **Erste Schritte** hat gerade mal angerissen, was mit Python und SPIKE Prime möglich ist. Erkunde auch die drei anderen Abschnitte des Python-Benutzerhandbuchs.

1. **Beispiel** Hier findest du Beispielprogramme, die zeigen, wie man Python verwendet, um diverse Aufgaben zu lösen. Kopiere die Beispielprogramme, probiere sie aus und passe sie an deine Bedürfnisse an.
2. **SPIKE Prime Module** Hier findest du die Dokumentation aller Funktionen und Variablen in den SPIKE Prime Modulen sowie kurze Beispiele, wie man sie verwendet.

Python-Lektionen

Wähle auf der Website LEGOeducation.com/lessons das Produkt **SPIKE™ Prime mit Python** aus. Dort findest du mehrere Einheitenpläne mit jeweils 6-8 Lektionen (nur auf Englisch verfügbar). Diese mehr als 50 Lektionen decken ein breites Themenspektrum ab und befassen sich unter anderem mit dem Debuggen, der Sensorsteuerung, einfachen Spielen oder Daten und mathematischen Funktionen. Entdecke die zahlreichen Möglichkeiten und werde eine echte Expertin beziehungsweise ein echter Experte für die Nutzung von Python mit SPIKE Prime.

Aufgabe

Erstelle ein neues Python-Projekt und mach dich ans Programmieren!