Python and the Spike Prime Hub (v3)

Table of Contents

Getting Started 3

Introduction to Python 3

Hello, World! 4

Comments in Python 6

Controlling Motors 7

Variables 9

The Power of Random 11

Sensor Control 13

Sensor Conditions 15

Next Steps 19

API Modules 19

App 19

App.Bargraph 19

App.Display 21

App.Linegraph 22

App.Music 24

App.Sound 26

Color 27

Color Matrix 28

Color Sensor 30

Device 31

Distance Sensor 32

Force Sensor 34

Hub 35

Hub.Button 36

Hub.Light 37

Hub.Light Matrix 38

Hub.Motion Sensor 43

Hub.Port 46
Hub.Sound 46
Motor 48
Motor Pair 56
Orientation 66
Runloop 66

Getting Started

Python is a popular text-based coding language that is excellent for beginners because it's concise and easy-to-read. It's also useful for programmers because it's applicable to web and software development, as well as scientific applications like data analysis and machine learning.

This **Getting Started** section introduces the basics of using Python with LEGO® Education SPIKE™ Prime. It contains chapters where you'll:

Introduction to Python

Learn to use the *Code Editor* in the LEGO® Education SPIKE™ App to write Python code.

Hello, World!

Write a message on the Light Matrix of the SPIKE Prime Hub.

Comments in Python

Learn how comments can help you describe draft and finished programs.

Controlling Motors

Define and start *asynchronous functions* to control motors.

Variables

Control two motors with *local* and *global* variables.

The Power of Random

Discover ways to create fun and unpredictable programs that control the light on the Hub.

Sensor Control

Control a motor using the Force Sensor. Then learn ways to use the Console to *debug* your program.

Sensor Conditions

Use *logical expressions* to react to different conditions. Then learn to run different parts of your code together to react to multiple conditions.

Next Steps

Get suggestions for additional resources to learn more about using Python with SPIKE Prime.

Python Syntax

When learning a text programming language, the first step is to understand its *syntax*. This language syntax prescribes the rules for writing *statements* (lines of code), and how to indicate *code blocks* that consist of multiple statements.

In Python, each statement begins with a level of *indentation* and ends with a *line break*. Indentation is the number of spaces before a statement. Lines with the same number of spaces have the same *indentation level* and belong to the same code block. The SPIKE App uses 4 spaces for each indentation level.

You write code in the *Code Editor*, which has features to help you write it correctly. For example, when you start a new code block, like a *function* or `if` statement, the Editor indents the next line with four extra spaces. Also, it numbers each line to make it easier to navigate your code.

Syntax highlighting in the Code Editor shows *comments*, *keywords*, text, and numbers in different colors so the code is easier to read. In the code below, the comment on the first line is green, the keywords `print`, `if`, and `True` are blue, the text `'LEGO'` is magenta, and the number 123 is orange.

```
# This is a comment.
print('LEGO')
if True:
    print(123)
```

The code above is an *example program*, which you'll find throughout the **Getting Started** chapters. Each example has a Copy icon in the top right corner:

Press this icon to copy the whole example program. Then right-click or long-press the Code Editor and choose Paste from the menu to paste the code. You can also press `CTRL+V` on Windows or `Command+V` on Mac.

SPIKE Prime Modules

To control the SPIKE Prime Hub, sensors, and motors, you'll need the SPIKE Prime *modules*. Modules are used to organize related code. There's one for each SPIKE Prime component, e.g., the `motor` module contains the code to control the motors. To use the functionality of a module, first *import* it with the `import` statement:

```
import motor
```

Import the modules you need once at the beginning of your Python program. See the **SPIKE Prime Modules** section of this User Guide for more on the modules and their functionality.

MicroPython

The SPIKE Prime Hub is a small computer called a microcontroller, which has limited memory and processing power. Since the full Python programming language would use too much memory, the Hub runs *MicroPython*, a highly optimized version of the Python language that can run on microcontrollers. The modules to control the SPIKE Prime Hub, sensors, and motors are also highly optimized by using optimized *data types*.

You've seen that the Code Editor shows text and numbers in different colors – because they're different data types. Python further distinguishes between whole numbers and decimals. Whole numbers are also known as *integers*, or type `int`, which is optimized in MicroPython. Decimals use the unoptimized `float` type, so the SPIKE Prime modules avoid this data type. This means you have to stick to whole numbers or use different *units* to describe decimals. For half a second, you can use 500 milliseconds instead of 0.5 seconds.

Challenge

Can you copy some of the example code in this chapter and paste it to the Code Editor?

It's a tradition when learning a new programming language to create a "Hello, World!" program. You're going to write "Hello, World!" on the Light Matrix of the SPIKE Prime Hub. First, make sure your SPIKE Prime Hub is turned on and connected to the SPIKE App. Then follow these four steps:

Make sure the Code Editor is empty by deleting any existing code.
Press the Copy icon in the top right corner of the example below to copy the code.
Right-click or long-press the Code Editor and then choose Paste from the menu to paste the code.

Press the Play button to run the program.

```
from hub import light_matrix

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

You'll see the text "Hello, World!" scrolling across the Light Matrix.

Let's look at the code line by line.

The first line imports the `light_matrix` module from the hub module, which controls the Light Matrix of the Hub. After importing a module, you can use its various functions.

The final line *calls* the `write()` *function* from the `light_matrix` module to write "Hello, World!" on the Light Matrix.

Define a Function

In the previous example, you used the `write()` function. A function is a block of code that performs a task when you call it. You define a function with the `def` keyword, followed by the function name, parentheses, and a colon. The *body* of the function is indented and contains all the code that runs when you call the function. You call a function by writing the function name and parentheses. Make sure to *unindent* the function call, otherwise it's part of the function body.

The example below defines the `hello()` function, which writes "Hello, World!" on the Light Matrix, and calls the function once. Try to run the example code. Remember to first delete any existing code from the Code Editor, before you copy, paste, and run the code.

```
from hub import light_matrix

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

hello()
```

Add a Parameter

In the example above, the `hello()` function has no *parameters*, so it writes "Hello World" on the Light Matrix each time you call it. To make it more dynamic, add a parameter name inside the parentheses of the function definition. The code block in the function body can then use this parameter to do something different based on its value. An example of this is the `write()` function, which has one required parameter: the text to write on the Light Matrix.

The example below adds a name parameter to the `hello()` function, which then writes `'Hello, ' + name + '!'` on the Light Matrix. Notice the + *operator*, which lets you add pieces of text together. Text is also known as a *string*, or type `str`. It is surrounded by either single (`'`) or double (`"`) straight quotation marks, using the same type around a given text string. The updated `hello()` function has one required parameter of type `str`, so you can *pass* the string `'World'` as an *argument* when you call it to write "Hello, World!" on the Light Matrix.

Try running the example code. Don't forget to delete any existing code from the Code Editor before you copy, paste, and run your new code.

```
from hub import light_matrix

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

hello('World')
```

You'll see the text "Hello, World!" scrolling across the Light Matrix once again.

Challenge

Can you change the code so the Hub greets *you* instead of the world?

It's easier to use code when you know what it should do. You can describe this in everyday language by adding comments. Comments aren't part of the code that runs on the Hub, so they don't influence its functionality.

The # character marks the start of a comment. You'd usually place a comment before the code it describes, but you can also place short comments after a code statement.

```
# This is a comment.
from hub import light_matrix
# This is another comment.
```

Sometimes part of your program doesn't behave like you want it to. In such cases, it's useful to *comment out* parts of the code by adding the # character at the start of the line. These lines then become comments and no longer run as a part of your program. Commenting out parts of a program can help you *debug* or find and correct the problem. To quickly comment out multiple lines of code, select them and then press `CTRL+/` on Windows or `Command+/` on Mac. To change multiple comments back to code, select them and press the same key combination.

```
# The next line is commented out:
# light_matrix.write('Hello, World!')
```

You can also use comments to describe your code before writing the working code. That's called *pseudocode* and it can help you describe in everyday language what your program should do. The example below uses comments as pseudocode for a blinking-eyes animation on the Light Matrix.

```
# Show a happy face with eyes on the Light Matrix.
# Wait for some time.
# Show a smile without eyes on the Light Matrix.
# Wait for a short time.
# Show the first image again on the Light Matrix.
```

Blinking Eyes Program

This program will show a face with blinking eyes on the Light Matrix of the Hub. Copy the code below and paste it into the Code Editor. Then run the program. As always, delete any existing code from the Code Editor before pasting the new code.

When you run this program, you'll see that the smiley face blinks after a second. The program calls the `show_image()` function from the `hub.light_matrix` module to show an image on the Light Matrix. The program uses the `sleep_ms()` function from the `time` module to add delays for a number of milliseconds between different images. In the code, each comment describes what the next line of code should do.

```
import time

from hub import light_matrix

# Show a happy face on the Light Matrix.
light_matrix.show_image(light_matrix.IMAGE_HAPPY)

# Wait for one second.
time.sleep_ms(1000)

# Show a smile on the Light Matrix.
light_matrix.show_image(light_matrix.IMAGE_SMILE)

# Wait for 0.2 seconds.
time.sleep_ms(200)

# Show a happy face on the Light Matrix.
light_matrix.show_image(light_matrix.IMAGE_HAPPY)
```

"WET" or "DRY"?

Although commenting every line of code is tempting, the result is that you'll *Write Everything Twice*. These *WET* comments don't help readers if the code is self-explanatory. Instead, follow the *DRY* principle and *Don't Repeat Yourself*.

In the example below, the lines of code that blink the eyes are inside the new `blink()` function. The program then calls the function three times, so the eyes blink three times. Notice that this time, the comments describe only the main parts of the code to help readers understand what that code should do.

```
import time

from hub import light_matrix

# This function blinks the eyes.
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)

# Blink three times.
blink()
blink()
blink()
```

Challenge

Can you change the code to keep the eyes open longer each time they blink?

You're ready to connect and use the motors. Connect a motor to port A and try the program below.

```python
import motor
from hub import port

# Run a motor on port A for 360 degrees at 720 degrees per second.
motor.run_for_degrees(port.A, 360, 720)
```

You should see the motor run 360 degrees (one complete rotation) at 720 degrees (two rotations) per second.

Let's examine the code line by line.

The first line imports the `motor` module that controls the motors.

The second line imports `port` from the hub module, which holds the value for each port. You can write `port.A` for port A, `port.B` for port B, and so on to specify the port(s) you want.

The final line calls the `run_for_degrees()` function with three *arguments*:

The first parameter specifies which motor to run, using the port value.
The second parameter specifies the number of degrees to run.
The third parameter specifies at what velocity to run the motor, in degrees per second.

Multiple Motors

Now connect a second motor to port B and try the program below.

```python
import motor
from hub import port

# Run two motors on ports A and B for 360 degrees at 720 degrees per second.
# The motors run at the same time.
motor.run_for_degrees(port.A, 360, 720)
motor.run_for_degrees(port.B, 360, 720)
```

Notice that both motors run 360 degrees (one rotation) at 720 degrees per second, starting and ending at the same time. Since the two motor statements are on separate lines, you might expect them to run one by one. However, they run at the same time because the `run_for_degrees()` is an *awaitable* function. That means you *can* wait for it to complete, but you don't have to. By default, the program immediately continues to the next line of code while the awaitable code runs to completion in the background. This makes it possible to run multiple commands at the same time.

Run Loop, Async, and Await

To effectively use awaitable code with the flexibility to run commands either concurrently or sequentially, you must run your code in an *asynchronous function* using a *run loop*. The `runloop` module controls the run loop on the Hub, and lets you run asynchronous functions with its `run()` function. An asynchronous function, also known as a *coroutine*, is an awaitable that uses the `async` keyword before the function definition. The convention is to name the coroutine containing your main program `main()`. The code below shows the general structure of a program using a run loop.

```python
import runloop

async def main():
    # Write your program here.

runloop.run(main())
```

In the body of a coroutine, you can use the `await` keyword before calling an awaitable command. This pauses the coroutine until the command completes. Without the keyword, the program immediately continues to the next line of code in the coroutine. You can still use regular (not awaitable) code inside the coroutine. However, doing so will always pause or *block* the whole program until the command completes.

The program below defines the `main()` coroutine, which uses the `await` keyword before the two `run_for_degrees()` function calls. It uses the `run()` function from the `runloop` module to run the `main()` coroutine on the final line of code.

```
import motor
import runloop
from hub import port

async def main():
    # Run two motors on ports A and B for 360 degrees at 720 degrees per second.
    # The motors run after each other.
    await motor.run_for_degrees(port.A, 360, 720)
    await motor.run_for_degrees(port.B, 360, 720)

runloop.run(main())
```

Try the sample code. You should see that both motors run 360 degrees (one rotation) at 720 degrees per second, one at a time.

Challenge

Can you change the code to run both motors at the same time again?

Sometimes, you find yourself writing the same number again and again. For example, the motor commands in the previous chapter ran for the same number of degrees at the same velocity each time. In cases like this, using variables makes changing multiple commands easier.

You create a variable by writing the variable name, followed by a single = sign, and the initial value for the variable. If you want to change the value of an existing variable, you use the exact same format to *assign* a new value to it.

Connect motors to ports A and B and try the program below.

```
import motor
import runloop
from hub import port

async def main():
    # Create a variable `velocity` with a value of 720.
    velocity = 720

    # Run two motors on ports A and B for 360 degrees.
    # Use the value of the `velocity` variable for the motor velocity.
    await motor.run_for_degrees(port.A, 360, velocity)
    await motor.run_for_degrees(port.B, 360, velocity)

runloop.run(main())
```

As in the previous chapter, you'll see both motors run 360 degrees (one rotation) at 720 degrees per second, one at a time. The example here creates a `velocity` variable and uses it in the `run_for_degrees()` function calls. Because we used a variable, it's easy to change the motor velocity for all the motor commands. Try changing the value of the `velocity` variable and run the program again.

Variable Scope

It's important to understand that it matters *where* a variable is created. When you create a variable inside a function, it's only available to that function. This is called a *local* variable. If you want to use a variable across different functions in your program, you must create the variable outside the functions, for example underneath your `import` statements. This is called a *global* variable.

```
import motor
import runloop
from hub import port

# Create a global variable `velocity` with a value of 720.
velocity = 720

async def main():
    # Create a local variable `degrees` with a value of 360.
    degrees = 360

    # Run two motors on ports A and B.
    # Use the value of the `degrees` variable for the number of degrees.
    # Use the value of the `velocity` variable for the motor velocity.
    await motor.run_for_degrees(port.A, degrees, velocity)
    await motor.run_for_degrees(port.B, degrees, velocity)

runloop.run(main())
```

Once again, you'll see both motors run 360 degrees (one rotation) at 720 degrees per second, one at a time. This time the `velocity` variable has global *scope* and a new `degrees` variable has local scope. You can use the global `velocity` variable both inside and outside of the `main()` function, but you can only use the local `degrees` variable inside the `main()` function where it is defined.

It can be tempting to define all your variables at the top of your program so they have global scope, because then you can conveniently use them in your whole program. However, this also means that the value of those variables can be changed from *anywhere* in your program, with undesired side effects. Instead, *tightly scope* your variables, so only the parts of your program that need to use and change them have access.

Variables in Loops

In Python, the simplest way to repeat some code a number of times is to use a `for` loop with the built-in `range()` function. For example, to repeat something four times, you write `for i in range(4):` followed by the code you want to run four times. You can think of `range(4)` as the *tuple* `(0, 1, 2, 3)`. Tuples and *lists* such as `[1, 2, 3]` are *iterables*. The `for` loop takes an iterable and *loops over* its values until it reaches the end.

When a `for` loop *iterates* over a tuple or list, it changes the value of a local variable on each iteration. So far, you've explicitly created variables and assigned them a value using the = sign. In a `for` loop, the name of the local variable is defined after the `for` keyword, in this case `i`. Each time the loop runs, the value of this local variable `i` changes. It will be `0` the first time the loop runs and 3 the last time it runs, corresponding to the values in `(0, 1, 2, 3)`.

The next example uses a `for` loop to change the global `velocity` variable four times to run the motor on port A with a different velocity each time. To enable changing the global variable `velocity` in the *local context* of the `main()` function, you need to use the `global` keyword before `velocity` at the start of the function body.

```
import motor
import runloop
from hub import port

# Create a global variable `velocity` with a value of 450.
velocity = 450

async def main():
    # Use the `global` keyword to enable changing `velocity` here.
    global velocity

    # Create a local variable `degrees` with a value of 360.
    degrees = 360

    # The `for` loop creates a local variable `i` and repeats 4 times.
    # The values of the `i` variable are 0, 1, 2, and 3.
    for i in range(4):
        # Change the global variable `velocity` by adding `i`*90 each time.
        # The values of the `velocity` variable are 450, 540, 720, and 990.
        velocity = velocity + i*90
        await motor.run_for_degrees(port.A, degrees, velocity)

    # The value of the `velocity` variable outside the `for` loop is 990.
    await motor.run_for_degrees(port.B, degrees, velocity)

runloop.run(main())
```

Run the example code. You'll see the motor on port A run 360 degrees four times, at four different velocities, faster each time. The final time, the motor on port B runs 360 degrees once at 990 degrees per second.

Challenge

Can you change the code so the motor on port A runs a different number of degrees each time?

Sometimes, the best program is an unpredictable one. When you don't know what a program will do next, it seems more alive. To achieve this result, add some randomness.

The program below will set the Power Button light on the SPIKE Prime Hub to ten different colors, with a random delay between each color change.

```
import random
import time
from hub import light

for color in range(11):
    # Set the light to the current color.
    light.color(light.POWER, color)

    # Keep the light on for 0.5 to 1.5 seconds.
    sleep_time = random.randint(500, 1500)
    time.sleep_ms(sleep_time)
```

Each color is represented by a different number. The `for` loop iterates over `range(11)` and assigns the value to the color variable. It will be 0 (black) which turns the light off the first time the loop runs and 10 (white) in the last iteration. Notice that this program imports the random module, which contains several functions to add randomness.

This example uses the randint() function, with a start value of 500 and a stop value of 1500. With these arguments, the function *returns* a number between 500 and 1500 to add some variation to the sleep time. However, the colors will always light up in the same order even if you run the program several times. Luckily, the random module has some other functions to add even more randomness to the program.

Loop Forever

You can also use a while loop to repeat something forever instead of a specific number of times. In Python, the simplest way to create such a loop is to write while True: followed by the code you want to run forever. The next example uses a while True loop to run a little disco show forever or until you stop the program.

```
import random
import time
from hub import light

while True:
    # Generate a random number between 1 and 9.
    random_color = random.randint(1, 9)

    # Set the light to the random color.
    light.color(light.POWER, random_color)

    # Keep the light on for 0.5 to 1.5 seconds.
    sleep_time = random.randint(500, 1500)
    time.sleep_ms(sleep_time)
```

Notice that the Power Button lights up in random colors, with a random delay between each color change. The example uses the randint() function again to generate a number between 1 and 9 (both included), which corresponds to the different color numbers excluding black (0) and white (10).

Lists and Constants

If you want the light show to only include certain colors, you can put them in a *list* and then choose a random color from it. You create a new list like a variable, first writing the list name, then the = sign, and finally the values inside square brackets, separated with commas. For a simple list with at least two *items*, write my_list = [1, 2]. You can add as many values as you want.

As you've seen in the previous examples, each color is represented by a different number. You use this number to set the light to that color, for example, number 9 will set the light to red. However, using numbers for colors can make it harder for other readers to know what your code will do. You could add comments to describe each value, but a better way is to make variables for each color. The color module has a variable RED so you can write color.RED instead of the number 9 in your code. (A variable listed with all capital letters is a *constant*, which means you shouldn't change it.)

The example below imports the color module and uses some of the color constants to create the list colors. This time, the randint() function determines how many times the for loop runs, and the light turns white at the end of this little random light show.

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

# Create a list with some different light colors.
colors = [color.RED, color.GREEN, color.BLUE, color.YELLOW]

# Change the light five to ten times.
times = random.randint(5, 10)

for i in range(times):
    # Choose a random color from the list of colors.
    random_color = random.choice(colors)

    # Set the light to the random color.
    light.color(light.POWER, random_color)

    # Keep the light on for 0.5 to 1.5 seconds.
    sleep_time = random.randint(500, 1500)
    time.sleep_ms(sleep_time)

# Set the light to white.
light.color(light.POWER, color.WHITE)
```

You'll see the Power Button light change to a random color from the list, for a random number of times, with a random delay between each color change. The example uses the `choice()` function to pick a random color from the `colors` list.

Challenge

Can you change the code so there are different colors in the `colors` list?

In the previous chapters, you tried using variables and random numbers to control the motors and the light. Now you'll use a sensor value to control a motor.

Connect a motor to port A and a Force Sensor to port B and try the program below.

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

# Store the force of the Force Sensor in a variable.
force = force_sensor.force(port.B)

# Print the variable to the Console.
print(force)

# Run the motor and use the variable to set the velocity.
motor.run(port.A, force)
```

Press the Force Sensor while the program is running. That didn't do much, right? Luckily, the example uses the built-in `print()` function to write the `force` variable to the Console, so that you can easily see what went wrong.

The Console

Sometimes your program doesn't do what you expect it to do. You can use the `print()` function to *debug* your program when that happens. The `print()` function writes whatever you pass as the argument to the Console window below the Code Editor, in this case the force of the Force Sensor. Run the program again and notice the value that appears in the Console.

You'll see a single number in the console, and unless you were pressing the Force Sensor when you started the program, that number is `0`. Running a motor at 0 degrees per second doesn't do much, so the problem is that the program only checks the sensor value once at the start of the program. To update the motor velocity based on the force for as long as the program runs, you'll need to use the `while True` loop again.

The Console also displays error messages when something goes wrong while running your program. One common error happens when you run a program to control a motor or read a sensor that isn't connected. Disconnect the Force Sensor and run the same program one last time. You'll see an error in the Console informing you that there was a problem, what the problem was, and on what line of code it happened.

Fix the Bugs

The Console helped you find two bugs. Reconnect the Force Sensor to port B to fix the second bug and then run the program below that fixes the first bug by *wrapping* the code in a `while True` loop.

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

while True:
    # Store the force of the Force Sensor in a variable.
    force = force_sensor.force(port.B)

    # Print the variable to the Console.
    print(force)

    # Run the motor and use the variable to set the speed.
    motor.run(port.A, force)
```

Press the Force Sensor while the program is running. You'll see the motor speeding up or slowing down depending on how hard you press the Force Sensor. You'll also see a lot of variable values written in the Console. The Force Sensor force is measured in decinewtons (dN) and since the maximum force it can measure is 10 newtons, the maximum value in dN is 100. Running a motor at 100 degrees per second still isn't very fast!

Function Return Values

Instead of storing the value of the Force Sensor in a variable, you can also define a function that *returns* this value. Separating the different parts of your program this way makes it easier to organize your code and fix bugs if they happen.

The next program defines a `motor_velocity()` function that returns the desired motor velocity based on the force of the Force Sensor instead of using a variable.

```
import force_sensor
import motor
from hub import port

# This function returns the desired motor velocity.
def motor_velocity():
    # The velocity is five times the force of the Force Sensor.
    return force_sensor.force(port.B) * 5

while True:
    # Run the motor like before.
    # Use the `motor_velocity()` function return value for velocity.
    motor.run(port.A, motor_velocity())
```

Press the Force Sensor while the program is running. You'll see the motor speeding up or slowing down depending on how hard you press the Force Sensor. The `motor_velocity()` function multiplies the force value by 5, so the velocity will be between 0 and 500 degrees per second.

Challenge

Can you change the code to run the motor at 1000 degrees per second when the Force Sensor is fully pressed?

You've used the sensor value to control a motor directly, but it's also possible to change the *flow* of the program using sensor *conditions* and an `if`statement. The `if` statement is an essential part of programming and the simplest way to control the flow of your program.

You create an `if` statement by writing `if` followed by a logical expression and a colon. Logical expressions are basically yes / no questions, such as "Is the color red?" or "Is the button pressed?" If the answer to the question is "yes," the expression is evaluated to be `True` and otherwise it is `False`. These are the two *Boolean* values used in Python, and are of type `bool`. All the lines of code with the same indentation level after the `if` statement are part of the code block that runs if the expression is `True`.

For an example, connect a Color Sensor to port A and run the program below.

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

async def main():
    while True:
        # Check if the red color is detected.
        if color_sensor.color(port.A) == color.RED:
            # If red is detected start a very long beep.
            sound.beep(440, 1000000, 100)
            # Pause the program while red is detected.
            while color_sensor.color(port.A) == color.RED:
                await runloop.sleep_ms(1)
            # Stop the sound when red is no longer detected.
            sound.stop()

runloop.run(main())
```

Wave a red LEGO® brick in front of the Color Sensor while the program is running. You'll hear a beep when the red color is detected, which stops when the red color is no longer detected. The example uses an `if` statement to check if the color detected by the Color Sensor is red. It does this using the *equality operator* == with the Color Sensor color value on the left and the `color.RED` constant on the right. (Note that there are two = signs vs. the single = sign you used to assign values to variables.) If the Color Sensor color value is the same as the `color.RED` constant, the condition is `True` and the code block after the `if` statement runs.

It's important to wrap the code in a `while True` loop. Otherwise, the Color Sensor only checks the color for a split second when the program starts. So far, you used the `while` loop with the `True` constant to repeat code forever. You can also use the `while` loop with a logical expression to repeat code only as long as that expression evaluates to `True`. The example above uses the same condition as the `if` statement in the inner `while` loop, to continue playing the beep while the red color is detected. When the condition is no longer `True`, the Hub *exits* the `while` loop and runs the next line of code to stop the sound.

Inside the inner `while` loop, notice the `sleep_ms()` function from the `runloop` module. This function pauses the `main()` coroutine for a number of milliseconds in a *non-blocking* way. Because it uses the `await` keyword, other tasks can run while that coroutine is paused. In the example, the pause is one millisecond. This may seem like very little time, but it's enough for the Hub to run many coroutines concurrently. The `sleep_ms()` function from the `time` module, which you used in earlier chapters, pauses the program in a *blocking* way. This means that it pauses the entire program and not just the code block where you call it.

What Else?

You can add more than one condition by extending the `if` statement with an `elif` statement that checks another condition. You can add as many of these as you need, and they follow the same syntax as the `if` statement. The `elif` should be on the same indentation level as the first `if` statement, and the `elif` keyword is followed by a logical expression and a colon. Indent the next line(s) of code that should run when this condition is `True`.

Sometimes, none of the conditions in the `if` and `elif` statements are `True`. In this case, you can run some code by adding an `else` statement without any condition. This runs when all the previous conditions are `False`.

For example, the program below adds an `elif` and `else` statement to also beep if the left button is pressed.

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

# This function returns `True` if the Color Sensor detects red.
def red_detected():
    return color_sensor.color(port.A) == color.RED

# This function returns `True` if the left button is pressed.
def left_pressed():
    return button.pressed(button.LEFT) > 0

async def main():
    while True:
        if red_detected():
            # If red is detected start a very long beep.
            sound.beep(440, 1000000, 100)
            # Wait until red is no longer detected.
            while red_detected():
                await runloop.sleep_ms(1)
        elif left_pressed():
            # If the left button is pressed make a short beep.
            sound.beep(880, 200, 100)
            # Wait until the left button is released.
            while left_pressed():
                await runloop.sleep_ms(1)
        else:
            # Otherwise, stop the sound.
            sound.stop()

runloop.run(main())
```

Wave a red LEGO brick in front of the Color Sensor and press the left button on the Hub while the program is running. You'll hear a beep for as long as the red color is detected, and a short beep each time the left button is pressed.

The example defines two functions to do the logic tests and return the result. The `red_detected()` function checks if the color detected by the Color Sensor is red and returns the result `True` or `False`. The `left_pressed()` function uses the `pressed()` function from the `hub.button` module and uses the > operator to check if the value is *greater than* `0` and returns the result.

The code in the `if` statement here is largely the same as the first example, but now it uses the `red_detected()` function in two places instead of repeating the condition for the `if` and `while` statements. The `elif` statement uses the `left_pressed()` function to check if the left button on the Hub is pressed for more than `0` milliseconds.

Note that the `elif` statement only runs when the condition of the first `if` statement is `False`. Therefore, pressing the button while the Color Sensor detects something red has no effect. You should carefully consider the order of your `if` and `elif` conditions and check the most important ones first. The `else` statement stops the sound if neither condition is `True`.

Multiple Conditions

When you use `if`/`elif`/`else` statements to test for multiple conditions, only one of the blocks will run. These conditions are *mutually exclusive*. You've seen that while the color red was detected, pressing the left button had no effect. To truly check multiple conditions, you must check them at the same time. Like adding several stacks of Word Blocks, in Python you can run multiple coroutines with the `run()` function from the `runloop` module. Until now, you've called it with the `main()` coroutine as its only argument, but it's possible to pass multiple coroutines as comma-separated arguments.

For example, the program below divides the code that checks the detected color and the pressed button into two coroutines. The `run()` function on the final line of code starts both coroutines at the same time.

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

# This function returns `True` if the Color Sensor detects red.
def red_detected():
    return color_sensor.color(port.A) == color.RED

# This function returns `True` if the left button is pressed.
def left_pressed():
    return button.pressed(button.LEFT) > 0

# This coroutine continuously checks if the Color Sensor detects red.
async def check_color():
    while True:
        # Wait until red is detected.
        while not red_detected():
            await runloop.sleep_ms(1)
        # When it's detected start a very long beep.
        sound.beep(440, 1000000, 100)
        # Wait until red is no longer detected.
        while red_detected():
            await runloop.sleep_ms(1)
        # When red is no longer detected stop the sound.
        sound.stop()

# This coroutine continuously checks if the left button is pressed.
async def check_button():
    while True:
        # Wait until the left button is pressed.
        while not left_pressed():
            await runloop.sleep_ms(1)
        # When it's pressed make a short beep.
        sound.beep(880, 200, 100)
        # Wait until the left button is released.
        while left_pressed():
            await runloop.sleep_ms(1)

# Run both coroutines.
runloop.run(check_color(), check_button())
```

Wave a red LEGO brick in front of the Color Sensor and press the left button on the Hub while the program is running. Like before, you'll hear a beep for as long as the red color is detected, and a short beep each time the left button is pressed. This time, it's possible to press the left button and hear a beep while the red color is detected because both functions run at the same time.

When you create your own coroutines, remember that:

- Your coroutines should at least `await` one command.
- When you use a *tight* `while` loop, use `await runloop.sleep_ms(1)` inside the loop to give other coroutines a chance to start and run.

Challenge

Can you change the code to detect a different color than red?

In the previous chapters, you

- learned the basics of using Python with SPIKE Prime, and how to use the Hub Light Matrix, Light, Speaker, and Buttons, as well as the motors, Color Sensor, and Force Sensor.
- became familiar with regular and asynchronous functions, local and global variables, and data types such as `int`, `bool`, `str`, `tuple`, and `list`.
- used `for` and `while` loops, as well as `if/elif/else` statements to control the flow of your program.
- learned how to use comments in your code, and when things go wrong, how to debug the program.

This is quite an achievement – you can be proud of yourself!

There are additional resources you can access to learn even more about using Python with SPIKE Prime.

Python User Guide

This **Getting Started** section has barely scratched the surface of what's possible with Python and SPIKE Prime. Explore these two other sections of the Python User Guide.

Examples Find example programs that show how to use Python to solve various tasks. Copy and try them, then modify them to suit your needs.
SPIKE Prime Modules Find documentation of all the functions and variables in the SPIKE Prime modules with short examples of how to use them.

Python Lessons

On the LEGOeducation.com/lessons website, select the product **SPIKE™ Prime with Python**. You'll find several unit plans with 6–8 lessons each (available in English only). These 50+ lessons cover a wide range of topics, from debugging to sensor control, and from simple games to data and math functions. Discover the many possibilities and become an expert using Python with SPIKE Prime.

Challenge

Create a new Python project and get coding!

API Modules

The app module is used communicate between hub and app

The `bargraph` module is used make bar graphs in the SPIKE App

To use the `bargraph` module simply import the module like so:

```
from app import bargraph
```

`bargraph` details

Functions

change

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

Parameters

color: int

A color from the `color` module

value: float

The value

clear_all

clear_all() -> None

Parameters

get_value

get_value(color: int) -> Awaitable

Parameters

color: int

A color from the `color` module

hide

hide() -> None

Parameters

set_value

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

Parameters

color: int

A color from the `color` module

value: float

The value

show

show(fullscreen: bool) -> None

Parameters

fullscreen: bool

Show in full screen

The `display` module is used show images in the SPIKE App

To use the `display` module simply import the module like so:

```python
from app import display
```

`display` details

Functions

hide

hide() -> None

Parameters

image

image(image: int) -> None

Parameters

image: int

The id of the image to show. The range of available images is 1 to 21. There are consts on the `display` module for these

show

show(fullscreen: bool) -> None

Parameters

fullscreen: bool

Show in full screen

text

text(text: str) -> None

Parameters

text: str

The text to display

app.display Constants

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

The `linegraph` module is used make line graphs in the SPIKE App

To use the `linegraph` module simply import the module like so:

```
from app import linegraph
```

`linegraph` details

Functions

clear

clear(color: int) -> None

Parameters

color: int

A color from the `color` module

clear_all

clear_all() -> None

Parameters

get_average

get_average(color: int) -> Awaitable

Parameters

color: int

A color from the `color` module

get_last

get_last(color: int) -> Awaitable

Parameters

color: int

A color from the `color` module

get_max

get_max(color: int) -> Awaitable

Parameters

color: int

A color from the `color` module

get_min

get_min(color: int) -> Awaitable

Parameters

color: int

A color from the `color` module

hide

hide() -> None

Parameters

plot

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

Parameters

color: int

A color from the `color` module

x: float

The X value

y: float

The Y value

show

show(fullscreen: bool) -> None

Parameters

fullscreen: bool

Show in full screen

The `music` module is used make music in the SPIKE App

To use the `music` module simply import the module like so:

```
from app import music
```

`music` details

Functions

play_drum

play_drum(drum: int) -> None

Parameters

drum: int

The drum name. See all available values in the app.sound module.

play_instrument

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

Parameters

instrument: int

The instrument name. See all available values in the app.music module.

note: int

The midi note to play (0-130)

duration: int

The duration in milliseconds

app.music Constants

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

The sound module is used play sounds in the SPIKE App

To use the sound module simply import the module like so:

```
from app import sound
```

sound details

Functions

play

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

Play a sound in the SPIKE App

Parameters

sound_name: str

The sound name as seen in the Word Blocks sound extension

volume: int

The volume (0-100)

pitch: int

The pitch of the sound

pan: int

The pan effect determines which speaker is emitting the sound, with "-100" being only the left speaker, "0" being normal, and "100" being only the right speaker.

set_attributes

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

Parameters

volume: int

The volume (0-100)

pitch: int

The pitch of the sound

pan: int

The pan effect determines which speaker is emitting the sound, with "-100" being only the left speaker, "0" being normal, and "100" being only the right speaker.

stop

stop() -> None

Parameters

The `color` module contains all the color constants to use with the `color_matrix`, `color_sensor` and `light` modules.

To use the Color module add the following import statement to your project:

```
import color
```

color Constants

BLACK = 0

MAGENTA = 1

PURPLE = 2

BLUE = 3

AZURE = 4

TURQUOISE = 5

GREEN = 6

YELLOW = 7

ORANGE = 8

RED = 9

WHITE = 10

UNKNOWN = -1

To use the Color Matrix module add the following import statement to your project:

```
import color_matrix
```

All functions in the module should be called inside the `color_matrix` module as a prefix like so:

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

Functions

clear

clear(port: int) -> None

Turn off all pixels on a Color Matrix

```
from hub import port
import color_matrix

color_matrix.clear(port.A)
```

Parameters

port: int

A port from the `port` submodule in the hub module

get_pixel

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

Retrieve a specific pixel represented as a tuple containing the color and intensity

```
from hub import port
import color_matrix

# Print the color and intensity of the 0,0 pixel on the Color Matrix connected to port A
print(color_matrix.get_pixel(port.A, 0, 0))
```

Parameters

port: int

A port from the `port` submodule in the hub module

x: int

The X value (0 - 2)

y: int

The Y value, range (0 - 2)

set_pixel

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

Change a single pixel on a Color Matrix

```
from hub import port
import color
import color_matrix

# Change the color of the 0,0 pixel on the Color Matrix connected to port A
color_matrix.set_pixel(port.A, 0, 0, (color.RED, 10))

# Print the color of the 0,0 pixel on the Color Matrix connected to port A
print(color_matrix.get_pixel(port.A, 0, 0)[0])
```

Parameters

port: int

A port from the `port` submodule in the hub module

x: int

The X value (0 - 2)

y: int

The Y value, range (0 - 2)

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

Tuple containing color and intensity, meaning how bright to light up the pixel

show

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

Change all pixels at once on a Color Matrix

```
from hub import port
import color
import color_matrix

# Update all pixels on Color Matrix using the show function

# Create a list with 18 items (color and intensity pairs)
pixels = [(color.BLUE, 10)] * 9

# Update all pixels to show same color and intensity
color_matrix.show(port.A, pixels)
```

Parameters

port: int

A port from the `port` submodule in the hub module

pixels: list[tuple[int, int]]

A list containing color and intensity value tuples for all 9 pixels.

The `color_sensor` module enables you to write code that reacts to specific colors or the intensity of the reflected light.

To use the Color Sensor module add the following import statement to your project:

```
import color_sensor
```

All functions in the module should be called inside the `color_sensor` module as a prefix like so:

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

The Color Sensor can recognize the following colors:

Red
Green
Blue
Magenta
Yellow
Orange
Azure
Black
White

Functions

color

color(port: int) -> int

Returns the color value of the detected color. Use the `color` module to map the color value to a specific color.

```
import color_sensor
from hub import port
import color

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

Parameters

port: int

A port from the `port` submodule in the hub module

reflection

reflection(port: int) -> int

Retrieves the intensity of the reflected light (0-100%).

Parameters

port: int

A port from the `port` submodule in the hub module

rgbi

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

Retrieves the overall color intensity and intensity of red, green and blue.

Returns tuple[red: int, green: int, blue: int, intensity: int]

Parameters

port: int

A port from the `port` submodule in the hub module

The `device` module enables you to write code to get information about devices plugged into the hub.

To use the Device module add the following import statement to your project:

```
import device
```

All functions in the module should be called inside the `device` module as a prefix like so:

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

Functions

data

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

Retrieve the raw LPF-2 data from a device.

Parameters

port: int

A port from the `port` submodule in the hub module

id

id(port: int) -> int

Retrieve the device id of a device. Each device has an id based on its type.

Parameters

port: int

A port from the `port` submodule in the hub module

get_duty_cycle

get_duty_cycle(port: int) -> int

Retrieve the duty cycle for a device. Returned values is in range 0 to 10000

Parameters

port: int

A port from the `port` submodule in the hub module

ready

ready(port: int) -> bool

When a device is attached to the hub it might take a short amount of time before it's ready to accept requests. Use `ready` to test for the readiness of the attached devices.

Parameters

port: int

A port from the `port` submodule in the hub module

set_duty_cycle

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

Set the duty cycle on a device. Range 0 to 10000

Parameters

port: int

A port from the `port` submodule in the hub module

duty_cycle: int

The PWM value (0-10000)

The `distance_sensor` module enables you to write code that reacts to specific distances or light up the Distance Sensor in different ways.

To use the Distance Sensor module add the following import statement to your project:

```
import distance_sensor
```

All functions in the module should be called inside the `distance_sensor` module as a prefix like so:

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

Functions

clear

clear(port: int) -> None

Turns off all the lights in the Distance Sensor connected to `port`.

Parameters

port: int

A port from the `port` submodule in the hub module

distance

distance(port: int) -> int

Retrieve the distance in millimeters captured by the Distance Sensor connected to `port`. If the Distance Sensor cannot read a valid distance it will return -1.

Parameters

port: int

A port from the `port` submodule in the hub module

get_pixel

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

Retrieve the intensity of a specific light on the Distance Sensor connected to `port`.

Parameters

port: int

A port from the `port` submodule in the hub module

x: int

The X value (0 - 3)

y: int

The Y value, range (0 - 3)

set_pixel

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

Changes the intensity of a specific light on the Distance Sensor connected to `port`.

Parameters

port: int

A port from the `port` submodule in the hub module

x: int

The X value (0 - 3)

y: int

The Y value, range (0 - 3)

intensity: int

How bright to light up the pixel

show

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

Change all the lights at the same time.

```
from hub import port
import distance_sensor

# Update all pixels on Distance Sensor using the show function

# Create a list with 4 identical intensity values
pixels = [100] * 4

# Update all pixels to show same intensity
distance_sensor.show(port.A, pixels)
```

Parameters

port: int

A port from the `port` submodule in the `hub` module

pixels: bytes

A list containing intensity values for all 4 pixels.

The `force_sensor` module contains all functions and constants to use the Force Sensor.

To use the Force Sensor module add the following import statement to your project:

```
import force_sensor
```

All functions in the module should be called inside the `force_sensor` module as a prefix like so:

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

Functions

force

force(port: int) -> int

Retrieves the measured force as decinewton. Values range from 0 to 100

```
from hub import port
import force_sensor


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

Parameters

port: int

A port from the `port` submodule in the hub module

pressed

pressed(port: int) -> bool

Tests whether the button on the sensor is pressed. Returns true if the force sensor connected to port is pressed.

```
from hub import port
import force_sensor


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

Parameters

port: int

A port from the `port` submodule in the hub module

raw

raw(port: int) -> int

Returns the raw, uncalibrated force value of the force sensor connected on port `port`

```
from hub import port
import force_sensor


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

Parameters

port: int

A port from the `port` submodule in the hub module

Functions

device_uuid

device_uuid() -> str

Retrieve the device id.

Parameters

hardware_id

hardware_id() -> str

Retrieve the hardware id.

Parameters

power_off

power_off() -> int

Turns off the hub.

Parameters

temperature

temperature() -> int

Retrieve the hub temperature. Measured in decidegrees celsius (d°C) which is 1 / 10 of a degree celsius (°C)

Parameters

To use the Button module add the following import statement to your project:

```
from hub import button
```

All functions in the module should be called inside the `button` module as a prefix like so:

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

Functions

pressed

int pressed(button: int) -> int

This module allows you to react to buttons being pressed on the hub. You must first import the `button` module to use the buttons.

```
from hub import button

left_button_press_duration = 0

# Wait for the left button to be pressed
while not button.pressed(button.LEFT):
    pass

# As long as the left button is being pressed, update the `left_button_press_duration` va
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")
```

Parameters

button: int

A button from the `button` submodule in the hub module

hub.button Constants

LEFT = 1
Left button next to the power button on the SPIKE Prime hub
RIGHT = 2
Right button next to the power button on the SPIKE Prime hub

The `light` module includes functions to change the color of the light on the SPIKE Prime hub.

To use the Light module add the following import statement to your project:

```
from hub import light
```

All functions in the module should be called inside the `light` module as a prefix like so:

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

Functions

color

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

Change the color of a light on the hub.

```
from hub import light
import color

# Change the light to red
light.color(light.POWER, color.RED)
```

Parameters

light: int

The light on the hub

color: int

A color from the `color` module

hub.light Constants

POWER = 0
The power button. On SPIKE Prime it's the button between the left and right buttons.
CONNECT = 1
The light around the Bluetooth connect button on SPIKE Prime.

To use the Light Matrix module add the following import statement to your project:

```
from hub import light_matrix
```

All functions in the module should be called inside the `light_matrix` module as a prefix like so:

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

Functions

clear

clear() -> None

Switches off all of the pixels on the Light Matrix.

```
from hub import light_matrix
import time
# Update pixels to show an image on Light Matrix, and then turn them off using the clear

# Show a small heart
light_matrix.show_image(2)

# Wait for two seconds
time.sleep_ms(2000)

# Switch off the heart
light_matrix.clear()
```

Parameters

get_orientation

get_orientation() -> int

Retrieve the current orientation of the Light Matrix.
Can be used with the following constants: `orientation.UP`, `orientation.LEFT`, `orientation.RIGHT`, `orientation.DOWN`

Parameters

get_pixel

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

Retrieve the intensity of a specific pixel on the Light Matrix.

```
from hub import light_matrix

# Show a heart
light_matrix.show_image(1)

# Print the value of the center pixel's intensity
print(light_matrix.get_pixel(2, 2))
```

Parameters

x: int

The X value, range (0 - 4)

y: int

The Y value, range (0 - 4)

set_orientation

set_orientation(top: int) -> int

Change the orientation of the Light Matrix. All subsequent calls will use the new orientation.
Can be used with the following constants: `orientation.UP`, `orientation.LEFT`, `orientation.RIGHT`, `orientation.DOWN`

Parameters

top: int

The side of the hub to be the top

set_pixel

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

Sets the brightness of one pixel (one of the 25 LEDs) on the Light Matrix.

```
from hub import light_matrix
# Turn on the pixel in the center of the hub
light_matrix.set_pixel(2, 2, 100)
```

Parameters

x: int

The X value, range (0 - 4)

y: int

The Y value, range (0 - 4)

intensity: int

How bright to light up the pixel

show

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

Change all the lights at the same time.

```
from hub import light_matrix
# Update all pixels on Light Matrix using the show function

# Create a list with 25 identical intensity values
pixels = [100] * 25

# Update all pixels to show same intensity
light_matrix.show(pixels)
```

Parameters

pixels: Iterable

A list containing light intensity values for all 25 pixels.

show_image

show_image(image: int) -> None

Display one of the built in images on the display.

```
from hub import light_matrix
# Update pixels to show an image on Light Matrix using the show_image function

# Show a smiling face
light_matrix.show_image(light_matrix.IMAGE_HAPPY)
```

Parameters

image: int

The id of the image to show. The range of available images is 1 to 67. There are consts on the `light_matrix` module for these.

write

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

Displays text on the Light Matrix, one letter at a time, scrolling from right to left except if there is a single character to show which will not scroll

```
from hub import light_matrix
# White a message to the hub
light_matrix.write("Hello, world!")
```

Parameters

text: str

The text to display

intensity: int

How bright to light up the pixel

time_per_character: int

How long to show each character on the display

hub.light_matrix Constants

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

To use the Motion Sensor module add the following import statement to your project:

```
from hub import motion_sensor
```

All functions in the module should be called inside the `motion_sensor` module as a prefix like so:

```
motion_sensor.up_face()
```

Functions

acceleration

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

Returns a tuple containing x, y & z acceleration values as integers. The values are mili G, so 1 / 1000 G

Parameters

raw_unfiltered: bool

If we want the data back raw and unfiltered

angular_velocity

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

Returns a tuple containing x, y & z angular velocity values as integers. The values are decidegrees per second

Parameters

raw_unfiltered: bool

If we want the data back raw and unfiltered

gesture

gesture() -> int

Returns the gesture recognized.

Possible values are:

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

Parameters

get_yaw_face

get_yaw_face() -> int

Retrieve the face of the hub that yaw is relative to.
If you put the hub on a flat surface with the face returned pointing up, when you rotate the hub only the yaw will update
`motion_sensor.TOP` The SPIKE Prime hub face with the USB charging port.
`motion_sensor.FRONT` The SPIKE Prime hub face with the Light Matrix.
`motion_sensor.RIGHT` The right side of the SPIKE Prime hub when facing the front hub face.
`motion_sensor.BOTTOM` The side of the SPIKE Prime hub where the battery is.
`motion_sensor.BACK` The SPIKE Prime hub face where the speaker is.
`motion_sensor.LEFT` The left side of the SPIKE Prime hub when facing the front hub face.

Parameters

quaternion

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

Returns the hub orientation quaternion as a tuple[w: float, x: float, y: float, z: float].

Parameters

reset_tap_count

reset_tap_count() -> None

Reset the tap count returned by the `tap_count` function

Parameters

reset_yaw

reset_yaw(angle: int) -> None

Change the yaw angle offset.
The angle set will be the new yaw value.

Parameters

angle: int

set_yaw_face

set_yaw_face(up: int) -> bool

Change what hub face is used as the yaw face.If you put the hub on a flat surface with this face pointing up, when you rotate the hub only the yaw will update

Parameters

up: int

The hub face that should be set as the upwards facing hub face.
Available values are:

`motion_sensor.TOP` The SPIKE Prime hub face with the USB charging port.
`motion_sensor.FRONT` The SPIKE Prime hub face with the Light Matrix.
`motion_sensor.RIGHT` The right side of the SPIKE Prime hub when facing the front hub face.
`motion_sensor.BOTTOM` The side of the SPIKE Prime hub where the battery is.
`motion_sensor.BACK` The SPIKE Prime hub face where the speaker is.
`motion_sensor.LEFT` The left side of the SPIKE Prime hub when facing the front hub face.

stable

stable() -> bool

Whether or not the hub is resting flat.

Parameters

tap_count

tap_count() -> int

Returns the number of taps recognized since the program started or last time `motion_sensor.reset_tap_count()` was called.

Parameters

tilt_angles

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

Returns a tuple containing yaw pitch and roll values as integers. Values are decidegrees

Parameters

up_face

up_face() -> int

Returns the Hub face that is currently facing up
`motion_sensor.TOP` The SPIKE Prime hub face with the USB charging port.
`motion_sensor.FRONT` The SPIKE Prime hub face with the Light Matrix.
`motion_sensor.RIGHT` The right side of the SPIKE Prime hub when facing the front hub face.
`motion_sensor.BOTTOM` The side of the SPIKE Prime hub where the battery is.
`motion_sensor.BACK` The SPIKE Prime hub face where the speaker is.
`motion_sensor.LEFT` The left side of the SPIKE Prime hub when facing the front hub face.

Parameters

hub.motion_sensor Constants

TAPPED = 0

DOUBLE_TAPPED = 1

SHAKEN = 2

FALLING = 3

UNKNOWN = -1

TOP = 0
The SPIKE Prime hub face with the Light Matrix.
FRONT = 1
The SPIKE Prime hub face where the speaker is.
RIGHT = 2
The right side of the SPIKE Prime hub when facing the front hub face.
BOTTOM = 3
The side of the SPIKE Prime hub where the battery is.
BACK = 4
The SPIKE Prime hub face with the USB charging port.
LEFT = 5
The left side of the SPIKE Prime hub when facing the front hub face.

This module contains constants that enables easy access to the ports on the SPIKE Prime hub. Use the constants in all functions that takes a `port` parameter.

To use the Port module add the following import statement to your project:

```
from hub import port
```

All functions in the module should be called inside the `port` module as a prefix like so:

```
port.A
```

hub.port Constants

A = 0
The Port that is labelled 'A' on the Hub.
B = 1
The Port that is labelled 'B' on the Hub.
C = 2
The Port that is labelled 'C' on the Hub.
D = 3
The Port that is labelled 'D' on the Hub.
E = 4
The Port that is labelled 'E' on the Hub.
F = 5
The Port that is labelled 'F' on the Hub.

To use the Sound module add the following import statement to your project:

```
from hub import sound
```

All functions in the module should be called inside the sound module as a prefix like so:

```
sound.stop()
```

Functions

beep

beep(freq: 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

Plays a beep sound from the hub

Parameters

freq: int

The frequency to play

duration: int

The duration in milliseconds

volume: int

The volume (0-100)

Optional keyword arguments:

attack: int

The time taken for initial run-up of level from nil to peak, beginning when the key is pressed.

decay: int

The time taken for the subsequent run down from the attack level to the designated sustain level.

sustain: int

The level during the main sequence of the sound's duration, until the key is released.

release: int

The time taken for the level to decay from the sustain level to zero after the key is released

transition: int

time in milliseconds to transition into the sound if something is already playing in the channel

waveform: int

The synthesized waveform. Use one of the constants in the hub.sound module.

channel: int

The desired channel to play on, options are `sound.DEFAULT` and `sound.ANY`

stop

stop() -> None

Stops all noise from the hub

Parameters

volume

volume(volume: int) -> None

Parameters

volume: int

The volume (0-100)

hub.sound Constants

ANY = -2

DEFAULT = -1

WAVEFORM_SINE = 1

WAVEFORM_SAWTOOTH = 3

WAVEFORM_SQUARE = 2

WAVEFORM_TRIANGLE = 1

To use a Motor add the following import statement to your project:

```
import motor
```

All functions in the module should be called inside the `motor` module as a prefix like so:

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

Functions

absolute_position

absolute_position(port: int) -> int

Get the absolute position of a Motor

Parameters

port: int

A port from the `port` submodule in the hub module

get_duty_cycle

get_duty_cycle(port: int) -> int

Get the pwm of a Motor

Parameters

port: int

A port from the `port` submodule in the hub module

relative_position

relative_position(port: int) -> int

Get the relative position of a Motor

Parameters

port: int

A port from the `port` submodule in the hub module

reset_relative_position

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

Change the position used as the offset when using the `run_to_relative_position` function.

Parameters

port: int

A port from the `port` submodule in the hub module

position: int

The degree of the motor

run

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

Start a Motor at a constant speed

```
from hub import port
import motor, time

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

Parameters

port: int

A port from the `port` submodule in the hub module

velocity: int

The velocity in degrees/sec

Value ranges depends on motor type.

Small motor (essential): -660 to 660
Medium motor: -1110 to 1110
Large motor: -1050 to 1050

Optional keyword arguments:

acceleration: int

The acceleration (deg/sec²) (1 - 10000)

run_for_degrees

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

Turn a motor for a specific number of degrees
When awaited returns a status of the movement that corresponds to one of the following constants:

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

Parameters

port: int

A port from the `port` submodule in the hub module

degrees: int

The number of degrees

velocity: int

The velocity in degrees/sec

Value ranges depends on motor type.

Small motor (essential): -660 to 660
Medium motor: -1110 to 1110
Large motor: -1050 to 1050

Optional keyword arguments:

stop: int

The behavior of the Motor after it has stopped. Use the constants in the `motor` module.

Possible values are
`motor.COAST` to make the motor coast until a stop

`motor.BRAKE` to brake and continue to brake after stop
`motor.HOLD` to tell the motor to hold it's position
`motor.CONTINUE` to tell the motor to keep running at whatever velocity it's running at until it gets another command
`motor.SMART_COAST` to make the motor brake until stop and then coast and compensate for inaccuracies in the next command
`motor.SMART_BRAKE` to make the motor brake and continue to brake after stop and compensate for inaccuracies in the next command

acceleration: int

The acceleration (deg/sec^2) (1 - 10000)

deceleration: int

The deceleration (deg/sec^2) (1 - 10000)

run_for_time

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

Run a Motor for a limited amount of time
When awaited returns a status of the movement that corresponds to one of the following constants:

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

```
from hub import port
import runloop
import motor

async def main():
    # Run at 1000 velocity for 1 second
    await motor.run_for_time(port.A, 1000, 1000)

    # Run at 280 velocity for 1 second
    await motor_pair.run_for_time(port.A, 1000, 280)

    # Run at 280 velocity for 10 seconds with a slow deceleration
    await motor_pair.run_for_time(port.A, 10000, 280, deceleration=10)

runloop.run(main())
```

Parameters

port: int

A port from the `port` submodule in the hub module

duration: int

The duration in milliseconds

velocity: int

The velocity in degrees/sec

Value ranges depends on motor type.

Small motor (essential): -660 to 660
Medium motor: -1110 to 1110
Large motor: -1050 to 1050

Optional keyword arguments:

stop: int

The behavior of the Motor after it has stopped. Use the constants in the `motor` module.

Possible values are
`motor.COAST` to make the motor coast until a stop
`motor.BRAKE` to brake and continue to brake after stop
`motor.HOLD` to tell the motor to hold it's position
`motor.CONTINUE` to tell the motor to keep running at whatever velocity it's running at until it gets another command
`motor.SMART_COAST` to make the motor brake until stop and then coast and compensate for inaccuracies in the next command
`motor.SMART_BRAKE` to make the motor brake and continue to brake after stop and compensate for inaccuracies in the next command

acceleration: int

The acceleration (deg/sec^2) (1 - 10000)

deceleration: int

The deceleration (deg/sec^2) (1 - 10000)

run_to_absolute_position

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

Turn a motor to an absolute position.
When awaited returns a status of the movement that corresponds to one of the following constants:

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

Parameters

port: int

A port from the `port` submodule in the hub module

position: int

The degree of the motor

velocity: int

The velocity in degrees/sec

Value ranges depends on motor type.

Small motor (essential): -660 to 660
Medium motor: -1110 to 1110
Large motor: -1050 to 1050

Optional keyword arguments:

direction: int

The direction to turn.
Options are:

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

stop: int

The behavior of the Motor after it has stopped. Use the constants in the `motor` module.

Possible values are
`motor.COAST` to make the motor coast until a stop
`motor.BRAKE` to brake and continue to brake after stop
`motor.HOLD` to tell the motor to hold it's position
`motor.CONTINUE` to tell the motor to keep running at whatever velocity it's running at until it gets another command
`motor.SMART_COAST` to make the motor brake until stop and then coast and compensate for inaccuracies in the next command
`motor.SMART_BRAKE` to make the motor brake and continue to brake after stop and compensate for inaccuracies in the next command

acceleration: int

The acceleration (deg/sec^2) (1 - 10000)

deceleration: int

The deceleration (deg/sec^2) (1 - 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

Turn a motor to a position relative to the current position.
When awaited returns a status of the movement that corresponds to one of the following constants:

`motor.READY`
`motor.RUNNING`
`motor.STALLED`

```
motor.CANCELED
motor.ERROR
motor.DISCONNECTED
```

Parameters

port: int

A port from the `port` submodule in the hub module

position: int

The degree of the motor

velocity: int

The velocity in degrees/sec

Value ranges depends on motor type.

Small motor (essential): -660 to 660
Medium motor: -1110 to 1110
Large motor: -1050 to 1050

Optional keyword arguments:

stop: int

The behavior of the Motor after it has stopped. Use the constants in the `motor` module.

Possible values are
`motor.COAST` to make the motor coast until a stop
`motor.BRAKE` to brake and continue to brake after stop
`motor.HOLD` to tell the motor to hold it's position
`motor.CONTINUE` to tell the motor to keep running at whatever velocity it's running at until it gets another command
`motor.SMART_COAST` to make the motor brake until stop and then coast and compensate for inaccuracies in the next command
`motor.SMART_BRAKE` to make the motor brake and continue to brake after stop and compensate for inaccuracies in the next command

acceleration: int

The acceleration (deg/sec^2) (1 - 10000)

deceleration: int

The deceleration (deg/sec^2) (1 - 10000)

set_duty_cycle

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

Start a Motor with a specific pwm

Parameters

port: int

A port from the `port` submodule in the hub module

pwm: int

The PWM value (-10000-10000)

stop

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

Stops a motor

```
from hub import port
import motor, time

# Start motor
motor.run(port.A, 1000)

# Wait for 2 seconds
time.sleep_ms(2000)

# Stop motor
motor.stop(port.A)
```

Parameters

port: int

A port from the `port` submodule in the hub module

Optional keyword arguments:

stop: int

The behavior of the Motor after it has stopped. Use the constants in the `motor` module.

Possible values are
`motor.COAST` to make the motor coast until a stop
`motor.BRAKE` to brake and continue to brake after stop
`motor.HOLD` to tell the motor to hold it's position
`motor.CONTINUE` to tell the motor to keep running at whatever velocity it's running at until it gets another command
`motor.SMART_COAST` to make the motor brake until stop and then coast and compensate for inaccuracies in the next command
`motor.SMART_BRAKE` to make the motor brake and continue to brake after stop and compensate for inaccuracies in the next command

velocity

velocity(port: int) -> int

Get the velocity (deg/sec) of a Motor

Parameters

port: int

A port from the `port` submodule in the hub module

motor Constants

READY = 0

RUNNING = 1

STALLED = 2

CANCELLED = 3

ERROR = 4

DISCONNECTED = 5

COAST = 0

BRAKE = 1

HOLD = 2

CONTINUE = 3

SMART_COAST = 4

SMART_BRAKE = 5

CLOCKWISE = 0

COUNTERCLOCKWISE = 1

SHORTEST_PATH = 2

LONGEST_PATH = 3

The `motor_pair` module is used to run motors in a synchronized fashion. This mode is optimal for creating drivebases where you'd want a pair of motors to start and stop at the same time.

To use the `motor_pair` module simply import the module like so:

```
import motor_pair
```

All functions in the module should be called inside the `motor_pair` module as a prefix like so:

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

Functions

move

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

Move a Motor Pair at a constant speed until a new command is given.

```python
from hub import port
import runloop
import motor_pair

async def main():
    # Pair motors on port A and B
    motor_pair.pair(motor_pair.PAIR_1, port.A, port.B)

    await runloop.sleep_ms(2000)

    # Move straight at default velocity
    motor_pair.move(motor_pair.PAIR_1, 0)

    await runloop.sleep_ms(2000)

    # Move straight at a specific velocity
    motor_pair.move(motor_pair.PAIR_1, 0, velocity=280)

    await runloop.sleep_ms(2000)

    # Move straight at a specific velocity and acceleration
    motor_pair.move(motor_pair.PAIR_1, 0, velocity=280, acceleration=100)

runloop.run(main())
```

Parameters

pair: int

The pair slot of the Motor Pair.

steering: int

The steering (-100 to 100)

Optional keyword arguments:

velocity: int

The velocity in degrees/sec

Value ranges depends on motor type.

Small motor (essential): -660 to 660
Medium motor: -1110 to 1110
Large motor: -1050 to 1050

acceleration: int

The acceleration (deg/sec^2) (1 - 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

Move a Motor Pair at a constant speed for a specific number of degrees.
When awaited returns a status of the movement that corresponds to one of the following constants from the motor module:

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

```
from hub import port
import runloop
import motor_pair

async def main():
    # Pair motors on port A and B
    motor_pair.pair(motor_pair.PAIR_1, port.A, port.B)

    # Move straight at default velocity for 90 degrees
    await motor_pair.move_for_degrees(motor_pair.PAIR_1, 90, 0)

    # Move straight at a specific velocity
    await motor_pair.move_for_degrees(motor_pair.PAIR_1, 360, 0, velocity=280)

    # Move straight at a specific velocity with a slow deceleration
    await motor_pair.move_for_degrees(motor_pair.PAIR_1, 360, 0, velocity=280, decelerati

runloop.run(main())
```

Parameters

pair: int

The pair slot of the Motor Pair.

degrees: int

The number of degrees

steering: int

The steering (-100 to 100)

Optional keyword arguments:

velocity: int

The velocity in degrees/sec

Value ranges depends on motor type.

Small motor (essential): -660 to 660
Medium motor: -1110 to 1110
Large motor: -1050 to 1050

stop: int

The behavior of the Motor after it has stopped. Use the constants in the `motor` module.

Possible values are
`motor.COAST` to make the motor coast until a stop
`motor.BRAKE` to brake and continue to brake after stop
`motor.HOLD` to tell the motor to hold it's position
`motor.CONTINUE` to tell the motor to keep running at whatever velocity it's running at until it gets another command
`motor.SMART_COAST` to make the motor brake until stop and then coast and compensate for inaccuracies in the next command
`motor.SMART_BRAKE` to make the motor brake and continue to brake after stop and compensate for inaccuracies in the next command

acceleration: int

The acceleration (deg/sec^2) (1 - 10000)

deceleration: int

The deceleration (deg/sec^2) (1 - 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

Move a Motor Pair at a constant speed for a specific duration.
When awaited returns a status of the movement that corresponds to one of the following constants from the motor module:

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

```python
from hub import port
import runloop
import motor_pair

async def main():
    # Pair motors on port A and B
    motor_pair.pair(motor_pair.PAIR_1, port.A, port.B)

    # Move straight at default velocity for 1 second
    await motor_pair.move_for_time(motor_pair.PAIR_1, 1000, 0)

    # Move straight at a specific velocity for 1 second
    await motor_pair.move_for_time(motor_pair.PAIR_1, 1000, 0, velocity=280)

    # Move straight at a specific velocity for 10 seconds with a slow deceleration
    await motor_pair.move_for_time(motor_pair.PAIR_1, 10000, 0, velocity=280, deceleratic

runloop.run(main())
```

Parameters

pair: int

The pair slot of the Motor Pair.

duration: int

The duration in milliseconds

steering: int

The steering (-100 to 100)

Optional keyword arguments:

velocity: int

The velocity in degrees/sec

Value ranges depends on motor type.

Small motor (essential): -660 to 660
Medium motor: -1110 to 1110
Large motor: -1050 to 1050

stop: int

The behavior of the Motor after it has stopped. Use the constants in the `motor` module.

Possible values are
`motor.COAST` to make the motor coast until a stop
`motor.BRAKE` to brake and continue to brake after stop
`motor.HOLD` to tell the motor to hold it's position
`motor.CONTINUE` to tell the motor to keep running at whatever velocity it's running at until it gets another command
`motor.SMART_COAST` to make the motor brake until stop and then coast and compensate for inaccuracies in the next command
`motor.SMART_BRAKE` to make the motor brake and continue to brake after stop and compensate for inaccuracies in the next command

acceleration: int

The acceleration (deg/sec^2) (1 - 10000)

deceleration: int

The deceleration (deg/sec^2) (1 - 10000)

move_tank

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

Perform a tank move on a Motor Pair at a constant speed until a new command is given.

```
from hub import port
import runloop
import motor_pair

async def main():
    # Pair motors on port A and B
    motor_pair.pair(motor_pair.PAIR_1, port.A, port.B)

    # Move straight at default velocity
    motor_pair.move_tank(motor_pair.PAIR_1, 1000, 1000)

    await runloop.sleep_ms(2000)

    # Turn right
    motor_pair.move_tank(motor_pair.PAIR_1, 0, 1000)

    await runloop.sleep_ms(2000)

    # Perform tank turn
    motor_pair.move_tank(motor_pair.PAIR_1, 1000, -1000)

runloop.run(main())
```

Parameters

pair: int

The pair slot of the Motor Pair.

left_velocity: int

The velocity (deg/sec) of the left motor.

right_velocity: int

The velocity (deg/sec) of the right motor.

Optional keyword arguments:

acceleration: int

The acceleration (deg/sec^2) (1 - 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

Perform a tank move on a Motor Pair at a constant speed until a new command is given.
When awaited returns a status of the movement that corresponds to one of the following constants from the motor module:

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

```
from hub import port
import runloop
import motor_pair

async def main():
    # Pair motors on port A and B
    motor_pair.pair(motor_pair.PAIR_1, port.A, port.B)

    # Move straight at default velocity for 360 degrees
    await motor_pair.move_tank_for_degrees(motor_pair.PAIR_1, 360, 1000, 1000)

    # Turn right for 180 degrees
    await motor_pair.move_tank_for_degrees(motor_pair.PAIR_1, 180, 0, 1000)

    # Perform tank turn for 720 degrees
    await motor_pair.move_tank_for_degrees(motor_pair.PAIR_1, 720, 1000, -1000)

runloop.run(main())
```

Parameters

pair: int

The pair slot of the Motor Pair.

degrees: int

The number of degrees

left_velocity: int

The velocity (deg/sec) of the left motor.

right_velocity: int

The velocity (deg/sec) of the right motor.

Optional keyword arguments:

stop: int

The behavior of the Motor after it has stopped. Use the constants in the `motor` module.

Possible values are
`motor.COAST` to make the motor coast until a stop
`motor.BRAKE` to brake and continue to brake after stop
`motor.HOLD` to tell the motor to hold it's position
`motor.CONTINUE` to tell the motor to keep running at whatever velocity it's running at until it gets another command
`motor.SMART_COAST` to make the motor brake until stop and then coast and compensate for inaccuracies in the next command
`motor.SMART_BRAKE` to make the motor brake and continue to brake after stop and compensate for inaccuracies in the next command

acceleration: int

The acceleration (deg/sec^2) (1 - 10000)

deceleration: int

The deceleration (deg/sec^2) (1 - 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

Perform a tank move on a Motor Pair at a constant speed for a specific amount of time.
When awaited returns a status of the movement that corresponds to one of the following constants from the motor module:

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

```python
from hub import port
import runloop
import motor_pair

async def main():
    # Pair motors on port A and B
    motor_pair.pair(motor_pair.PAIR_1, port.A, port.B)

    # Move straight at default velocity for 1 second
    await motor_pair.move_tank_for_time(motor_pair.PAIR_1, 1000, 1000, 1000)

    # Turn right for 3 seconds
    await motor_pair.move_tank_for_time(motor_pair.PAIR_1, 0, 1000, 3000)

    # Perform tank turn for 2 seconds
    await motor_pair.move_tank_for_time(motor_pair.PAIR_1, 1000, -1000, 2000)

runloop.run(main())
```

Parameters

pair: int

The pair slot of the Motor Pair.

duration: int

The duration in milliseconds

left_velocity: int

The velocity (deg/sec) of the left motor.

right_velocity: int

The velocity (deg/sec) of the right motor.

Optional keyword arguments:

stop: int

The behavior of the Motor after it has stopped. Use the constants in the `motor` module.

Possible values are
`motor.COAST` to make the motor coast until a stop
`motor.BRAKE` to brake and continue to brake after stop
`motor.HOLD` to tell the motor to hold it's position
`motor.CONTINUE` to tell the motor to keep running at whatever velocity it's running at until it gets another command
`motor.SMART_COAST` to make the motor brake until stop and then coast and compensate for inaccuracies in the next command
`motor.SMART_BRAKE` to make the motor brake and continue to brake after stop and compensate for inaccuracies in the next command

acceleration: int

The acceleration (deg/sec^2) (1 - 10000)

deceleration: int

The deceleration (deg/sec^2) (1 - 10000)

pair

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

pair two motors (`left_motor` & `right_motor`) and store the paired motors in `pair`.
Use `pair` in all subsequent `motor_pair` related function calls.

```
import motor_pair
from hub import port

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

Parameters

pair: int

The pair slot of the Motor Pair.

left_motor: int

The port of the left motor. Use the `port` submodule in the hub module.

right_motor: int

The port of the right motor. Use the `port` submodule in the hub module.

stop

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

Stops a Motor Pair.

```
import motor_pair

motor_pair.stop(motor_pair.PAIR_1)
```

Parameters

pair: int

The pair slot of the Motor Pair.

Optional keyword arguments:

stop: int

The behavior of the Motor after it has stopped. Use the constants in the `motor` module.

Possible values are
`motor.COAST` to make the motor coast until a stop
`motor.BRAKE` to brake and continue to brake after stop
`motor.HOLD` to tell the motor to hold it's position
`motor.CONTINUE` to tell the motor to keep running at whatever velocity it's running at until it gets another command
`motor.SMART_COAST` to make the motor brake until stop and then coast and compensate for inaccuracies in the next command
`motor.SMART_BRAKE` to make the motor brake and continue to brake after stop and compensate for inaccuracies in the next command

unpair

unpair(pair: int) -> None

Unpair a Motor Pair.

```
import motor_pair

motor_pair.unpair(motor_pair.PAIR_1)
```

Parameters

pair: int

The pair slot of the Motor Pair.

motor_pair Constants

PAIR_1 = 0
First Motor Pair
PAIR_2 = 1
Second Motor Pair
PAIR_3 = 2
Third Motor Pair

The `orientation` module contains all the orientation constants to use with the `light_matrix` module.

To use the orientation module add the following import statement to your project:

```
import orientation
```

orientation Constants

UP = 0

RIGHT = 1

DOWN = 2

LEFT = 3

The `runloop` module contains all functions and constants to use the Runloop.

To use the Runloop module add the following import statement to your project:

```
import runloop
```

All functions in the module should be called inside the `runloop` module as a prefix like so:

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

Functions

run

run(*functions: Awaitable) -> None

Start any number of parallel `async` functions. This is the function you should use to create programs with a similar structure to Word Blocks.

Parameters

***functions: awaitable**

The functions to run

sleep_ms

sleep_ms(duration: int) -> Awaitable

Pause the execution of the application for any amount of milliseconds.

```python
from hub import light_matrix
import runloop

async def main():
    light_matrix.write("Hi!")
    # Wait for ten seconds
    await runloop.sleep_ms(10000)
    light_matrix.write("Are you still here?")

runloop.run(main())
```

Parameters

duration: int

The duration in milliseconds

until

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

Returns an awaitable that will return when the condition in the function or lambda passed is `True` or when it times out

```python
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():
    # Wait until Color Sensor sees red
    await runloop.until(is_color_red)
    print("Red!")

runloop.run(main())
```

Parameters

function: Callable[[], bool]

A callable with no parameters that returns either `True` or `False`.
Callable is anything that can be called, so a `def` or a `lambda`

timeout: int

A timeout for the function in milliseconds.
If the callable does not return `True` within the timeout, the `until` still resolves after the timeout.
0 means no timeout, in that case it will not resolve until the callable returns `True`