Python and the Spike Prime Hub (v3)

Table of Contents

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:

1. Make sure the Code Editor is empty by deleting any existing code.
2. Press the Copy icon in the top right corner of the example below to copy the code.
3. Right-click or long-press the Code Editor and then choose Paste from the menu to paste the code.
4. 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*.

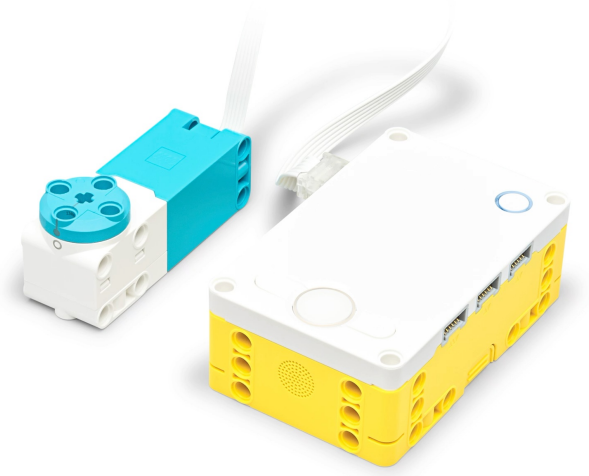
In the example below, the lines of code that blink the eyes are inside the new `blink()` function. The program then calls the function three times, so the eyes blink three times. Notice that this time, the comments describe only the main parts of the code to help readers understand what that code should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



```
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.

```
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.

```
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.

```
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)
```

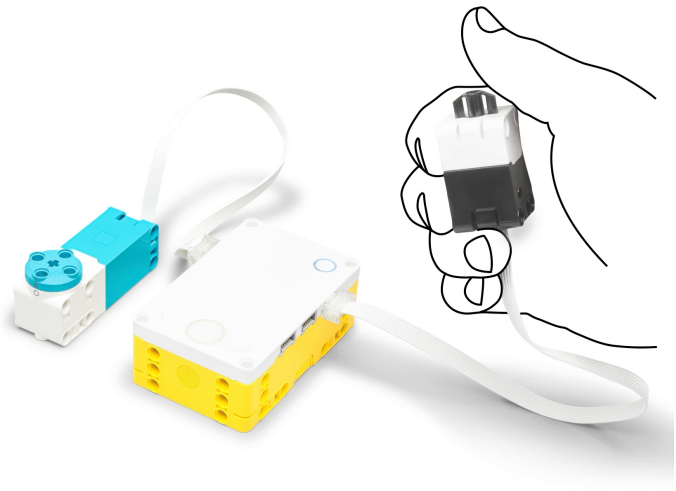
You'll see the Power Button light change to a random color from the list, for a random number of times, with a random delay between each color change. The example uses the `choice()` function to pick a random color from the `colors` list.

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.



```
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.

```
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.

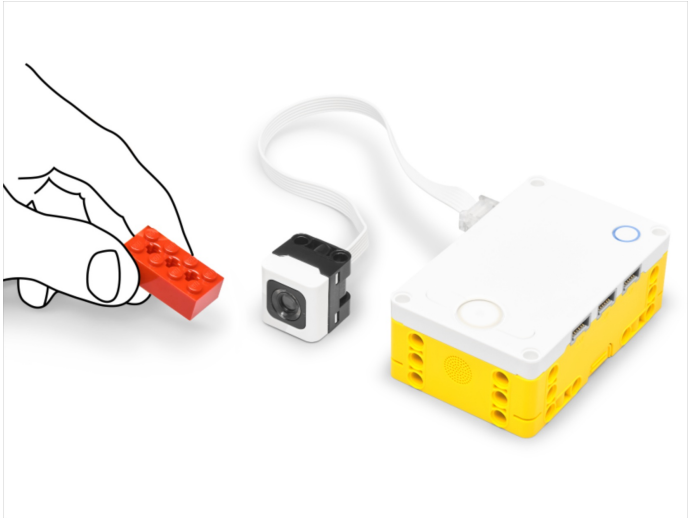
Challenge

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.

```
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!