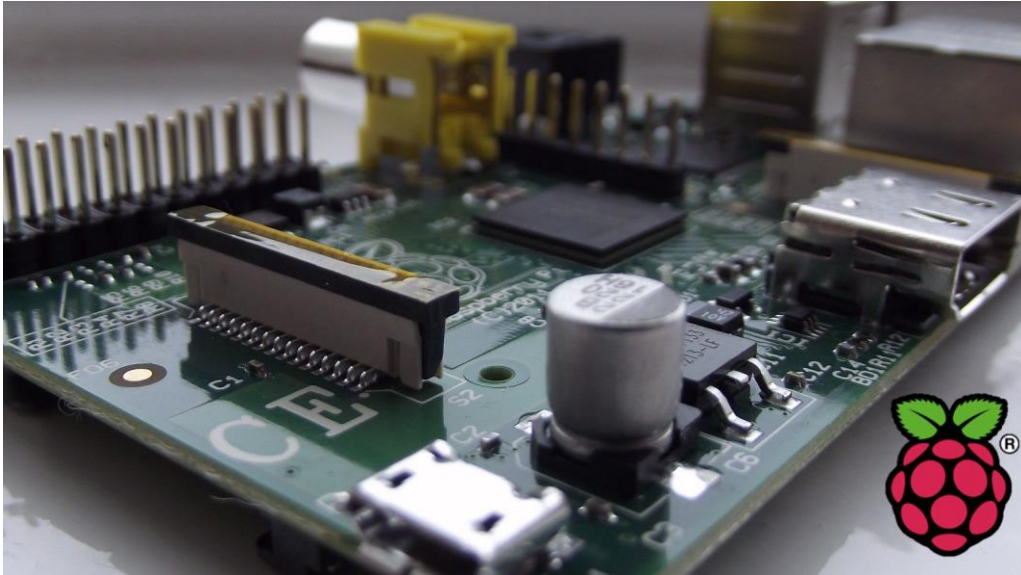


Introduction

The Raspberry Pi is an amazing *single board computer* (SBC) capable of running Linux and a whole host of applications. Python is a beginner-friendly programming language that is used in schools, web development, scientific research, and in many other industries. This guide will walk you through writing your own programs with Python to blink lights, respond to button pushes, read sensors, and log data on the Raspberry Pi.



Development Environments

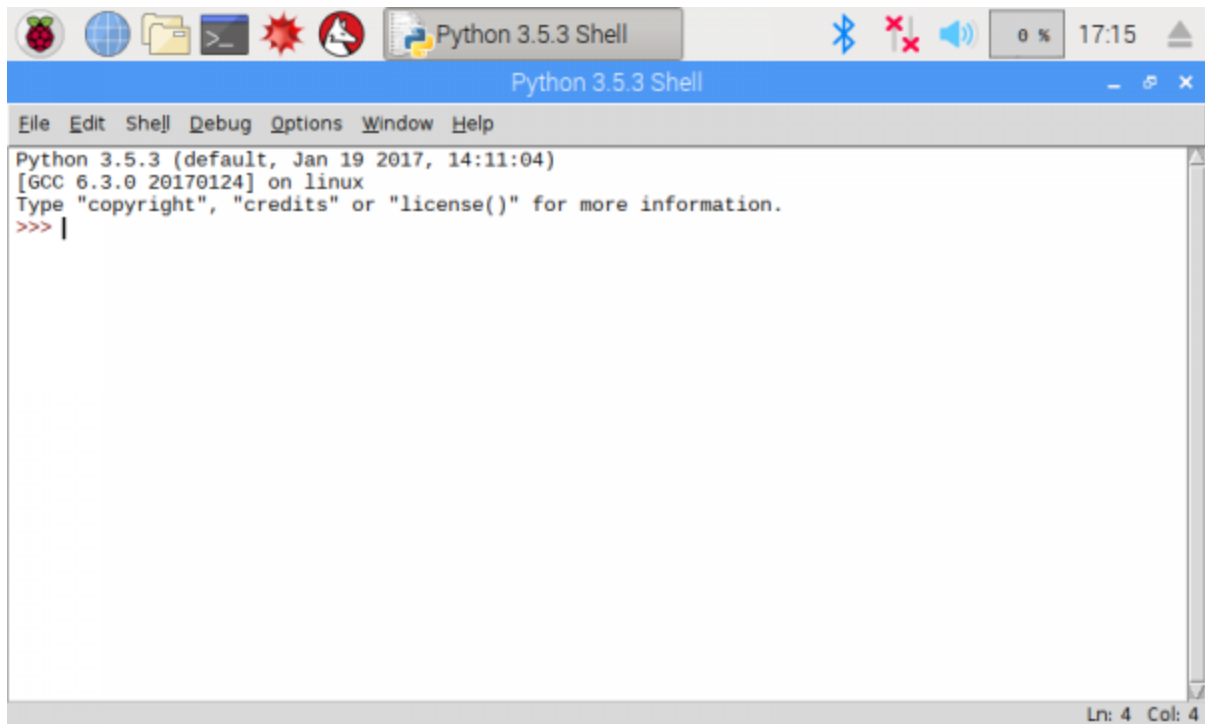
The simplest way to create Python programs is to write your code in a text editor (e.g. nano, vim, emacs, Midnight Commander, Leafpad, etc.), save it, and then run it from the terminal with the command `python <FILE>.py`. You are welcome to continue working through this guide using a text editor and command line.

Some users prefer to use an *integrated development environment* (IDE) when developing code. IDEs offer a number of benefits including syntax highlighting, code completion, one-click running, debugging hints, etc. However, most IDEs require a graphical interface to use, which means you will need to be on the full desktop version of Raspbian.

IDLE

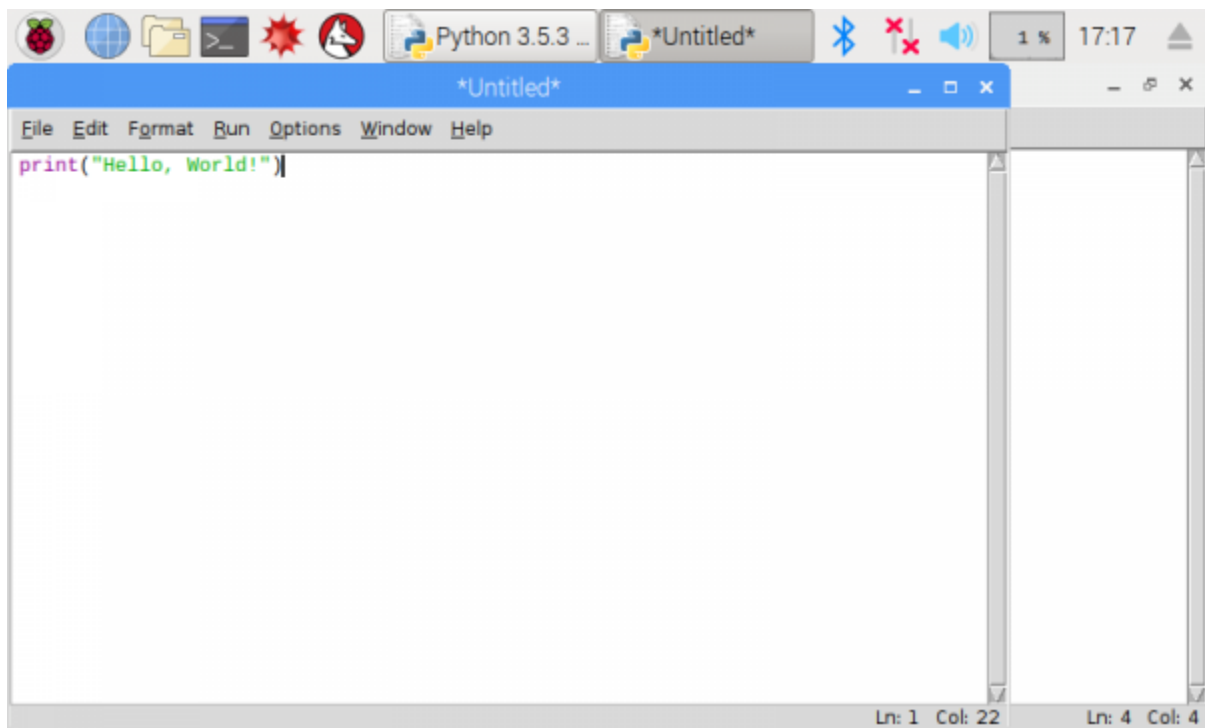
IDLE is the default Python editor that has been available on Raspbian for many generations. The good news is that it has a built-in interpreter, which allows you to run commands one at a time to test code. The bad news is that it doesn't show line numbers, and it only works with Python (but you're only here for Python anyway, right?).

Open IDLE by selecting the Raspberry Pi logo in the top-left, and click *Programming > Python 3 (IDLE)*. You should be presented with the Python interactive interpreter.



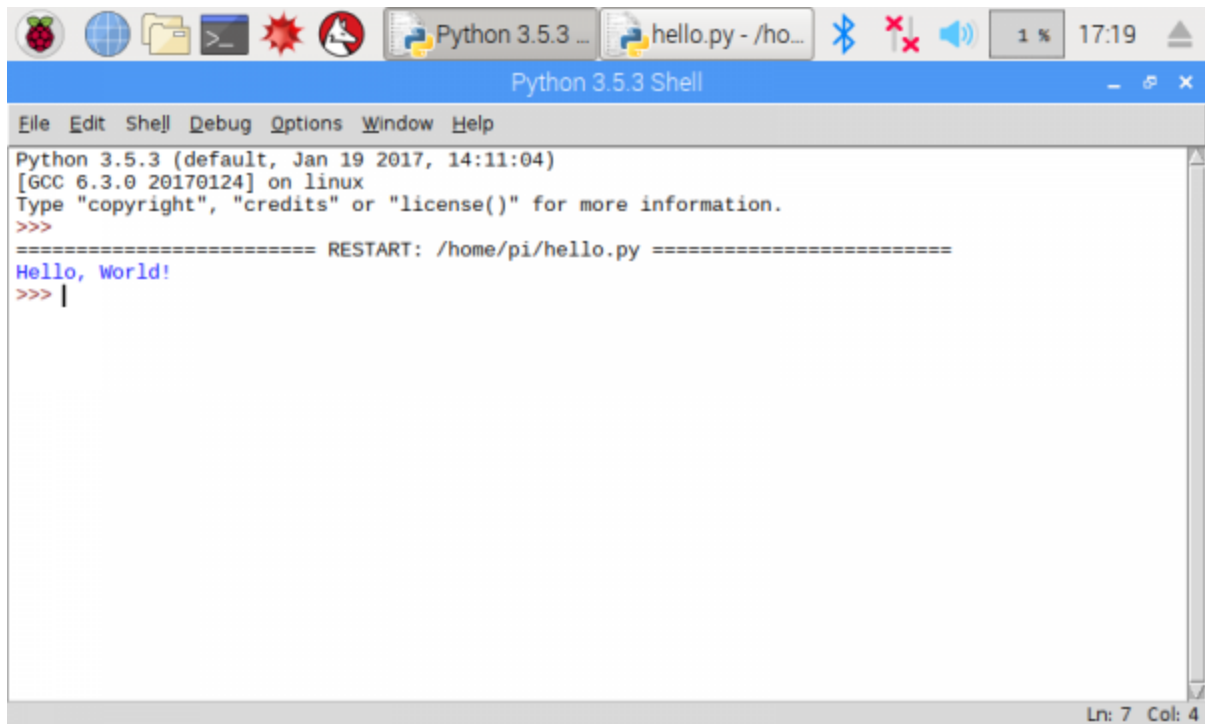
The screenshot shows a Raspberry Pi desktop environment. A terminal window titled "Python 3.5.3 Shell" is open. The terminal displays the following text: "Python 3.5.3 (default, Jan 19 2017, 14:11:04) [GCC 6.3.0 20170124] on linux Type 'copyright', 'credits' or 'license()' for more information. >>>". The terminal has a menu bar with "File", "Edit", "Shell", "Debug", "Options", "Window", and "Help". The status bar at the bottom right shows "Ln: 4 Col: 4".

To write a program, go to *File > New File*. Enter in your code.



The screenshot shows a Raspberry Pi desktop environment. A code editor window titled "*Untitled*" is open. The editor displays the following code: `print("Hello, World!")`. The editor has a menu bar with "File", "Edit", "Format", "Run", "Options", "Window", and "Help". The status bar at the bottom right shows "Ln: 1 Col: 22".

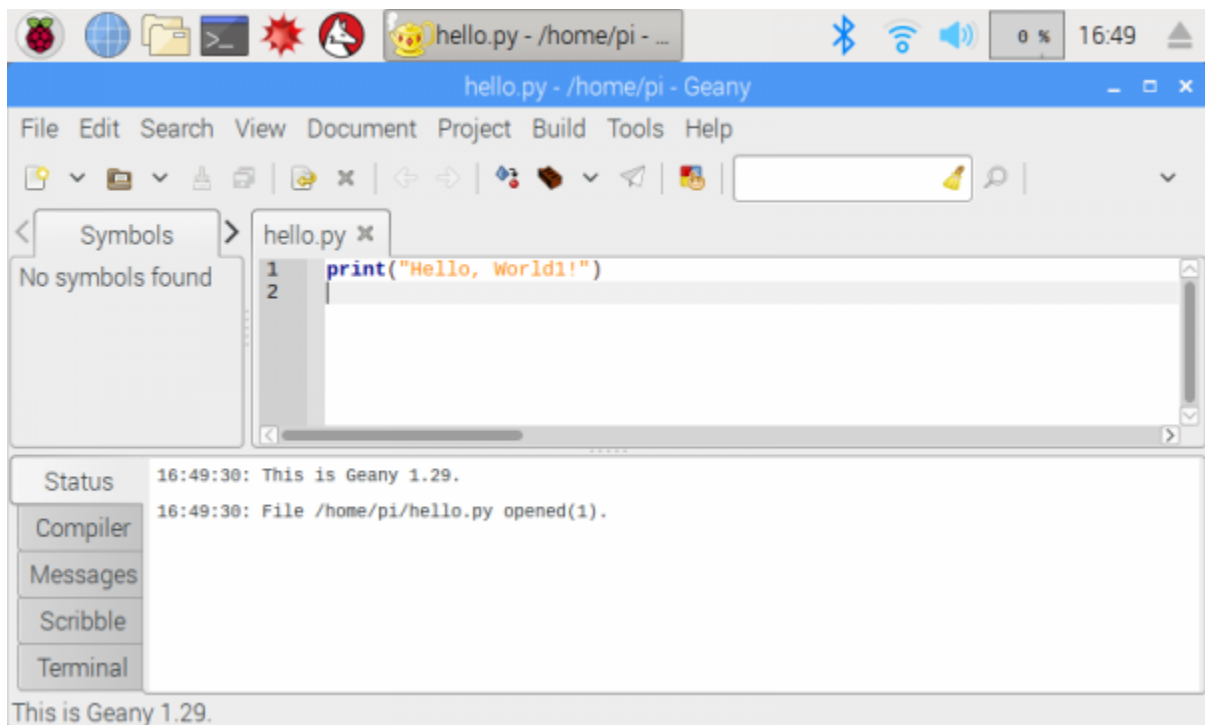
Click *File > Save As...* to save your code to a Python file (don't forget the *.py* suffix!). Click *Run > Run Module* to run your program.



```
Python 3.5.3 (default, Jan 19 2017, 14:11:04)
[GCC 6.3.0 20170124] on linux
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: /home/pi/hello.py =====
Hello, World!
>>> |
```

Geany

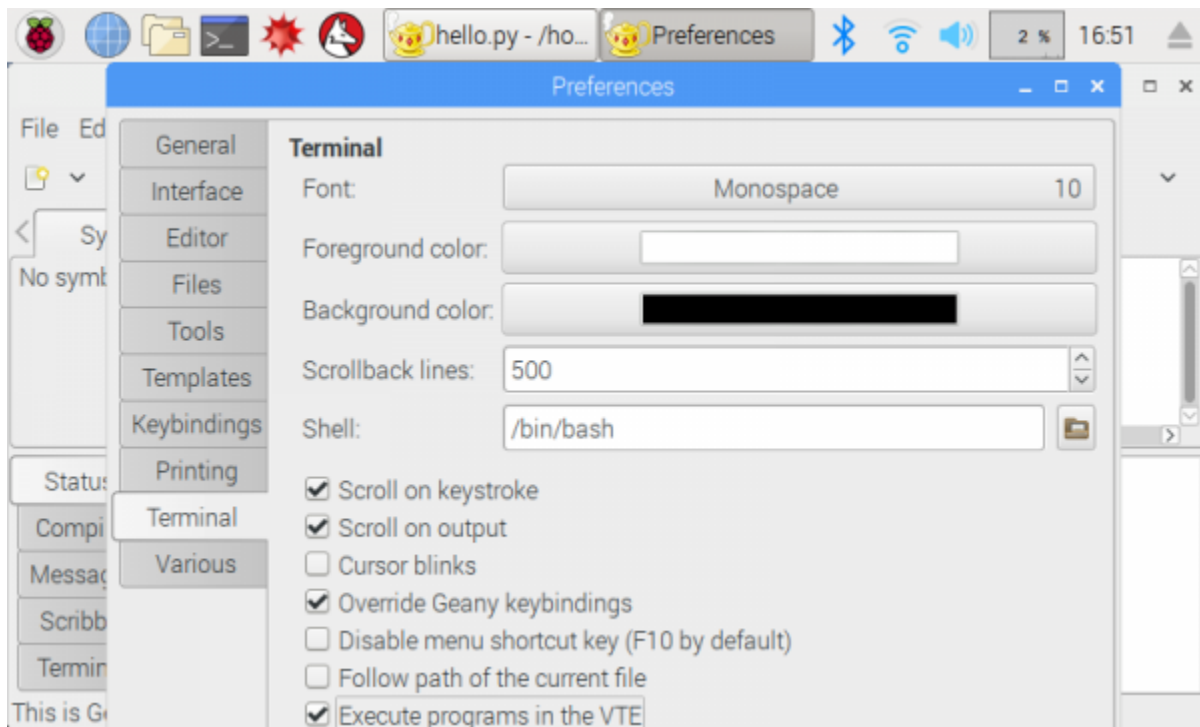
Geany is a great, beginner-friendly IDE that works with many different languages. However, it does not start up with a Python interactive interpreter. You can open Geany up by clicking on the Raspberry Pi logo in the top-left, and selecting *Programming >Geany*. Write your code in the file editor pane.



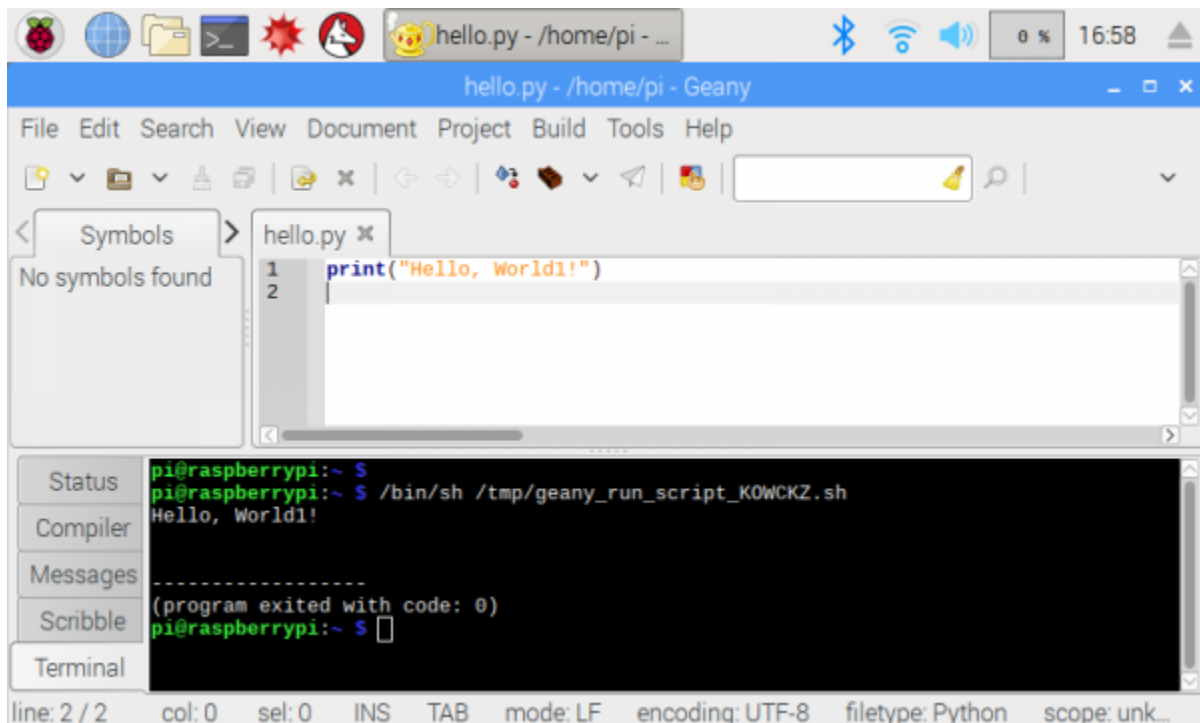
```
1 print("Hello, World!")
2
```

Save your code, making sure the filename ends with `.py`.

By default, Geany will attempt to open a new window to show the output of your code, which may or may not work on the Raspberry Pi. We can change it to run in the *Terminal* pane. Click *Edit > Preferences*. Select the *Terminal* tab and click to enable *Execute programs in the VTE*. Press *enter* to save and close the Preferences window.

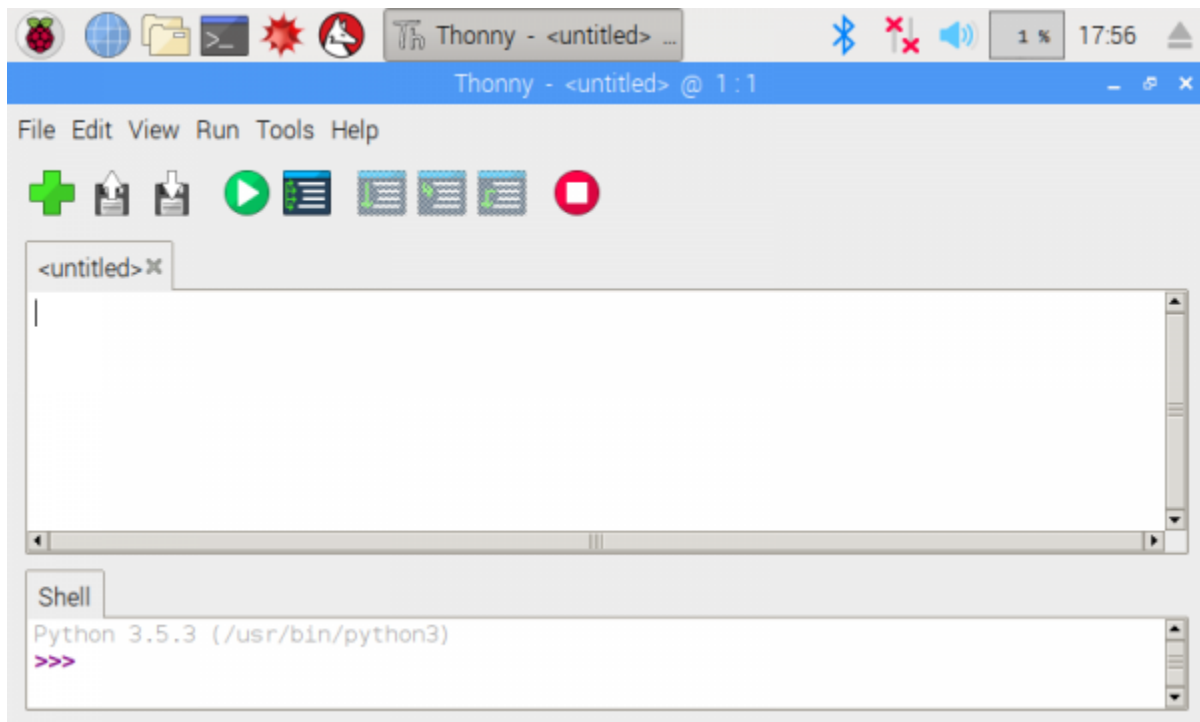


Click *Build > Execute* (or click the paper airplane icon) to run your code. You should see the output of your program appear in the Terminal pane of Geany.

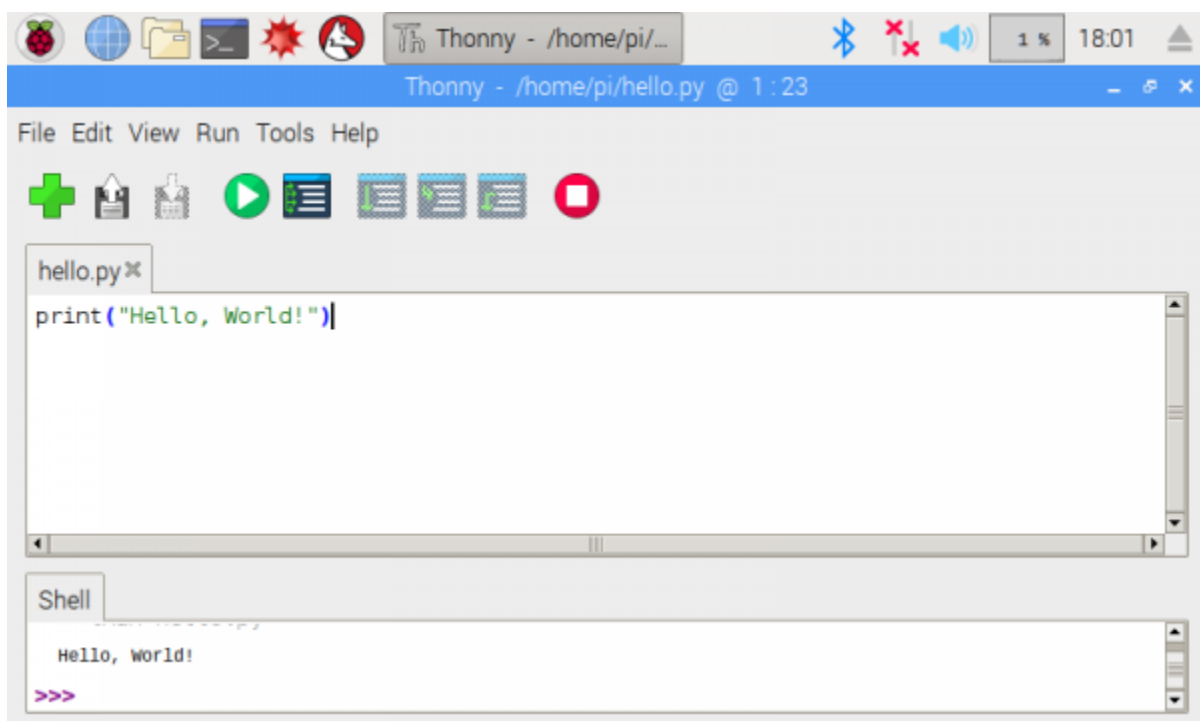


Thonny

Finally, Thonny is another great, easy-to-use IDE that comes pre-loaded on Raspbian. It focuses on Python and has an interactive environment when you load the program. Start Thonny by clicking on the Raspberry Pi icon followed by *Programming > Thonny Python IDE*



Write your program in the top pane, click *File > Save as...* to save it, and click *Run > Run current script* to execute the program. Output will appear in the bottom interpreter pane.



Opinion: If you are just starting your journey in programming, we recommend Thonny for a graphical IDE and using nano if you are using a headless Raspberry Pi setup.

Programming in Python

The bulk of this tutorial focuses on controlling hardware connected to the Raspberry Pi. To accomplish that, we will be using the Python programming language. As such, you should be familiar with some of the basics of Python, including literals, variables, operators, control flow, scope, and data structures. Depending on your comfort level with Python, we recommend the following:

- **Not familiar:** Read the recommended documentation and attempt the challenges
- **Somewhat familiar:** Attempt the challenges and refer to the documentation when you run into trouble
- **Very familiar:** Feel free to skip this whole section!

Comments

A *comment* is any text to the right of the pound (or hash) symbol `#`. The Python interpreter ignores this text, and it can be useful for writing notes to yourself or other programmers about what's going on in the code.

Example:

```
# This is a comment and is not seen by the interpreter
print("Hello, World!")
```

Literals

Literals, also known as *literal constants*, are fixed values and include integers (e.g. 42), floating-point numbers (e.g. 6.23), and strings (e.g. "Hello, World!"). Note that strings need to be in between single quotation marks (' ') or in between double quotation marks (" ").

Example:

```
print(42)
print("hi")
```

 **Challenge:** Change the `print("Hello, World!")` program we wrote earlier so that it prints out your name.

Variables

Variables are containers whose values can change. We can store a number or string in a variable and then retrieve that value later.

Example:

```
number=42
print(number)
```

 **Challenge:** Store your name in a variable and then print that variable's value to the terminal.

Logical Lines

So far, we've been writing one expression per line in our program. For example:

```
message="hello!"  
print(message)
```

You can combine these two lines into one line by separating them with a semicolon ;

```
message="hello"; print(message)
```

These two programs will execute in exactly the same fashion.

User Input

You can ask a user to enter information into the terminal by using the `input()` function. This will prompt the user to type out some text (including numbers) and then press *enter* to submit the text. Upon submission, the `input()` function will read the text and then return it as a string, which can be stored in a variable.

Whatever is in between the parentheses (known as *arguments*) will be printed to the screen prior to accepting user input.

Functions are sections of code that can be called by name. For example, `print()` is a function that takes the arguments and prints it out to the terminal. Notice in the example below that we separated two different arguments in `print()` by a comma. In this case, `print()` will print the different strings (or variables) in order on one line.

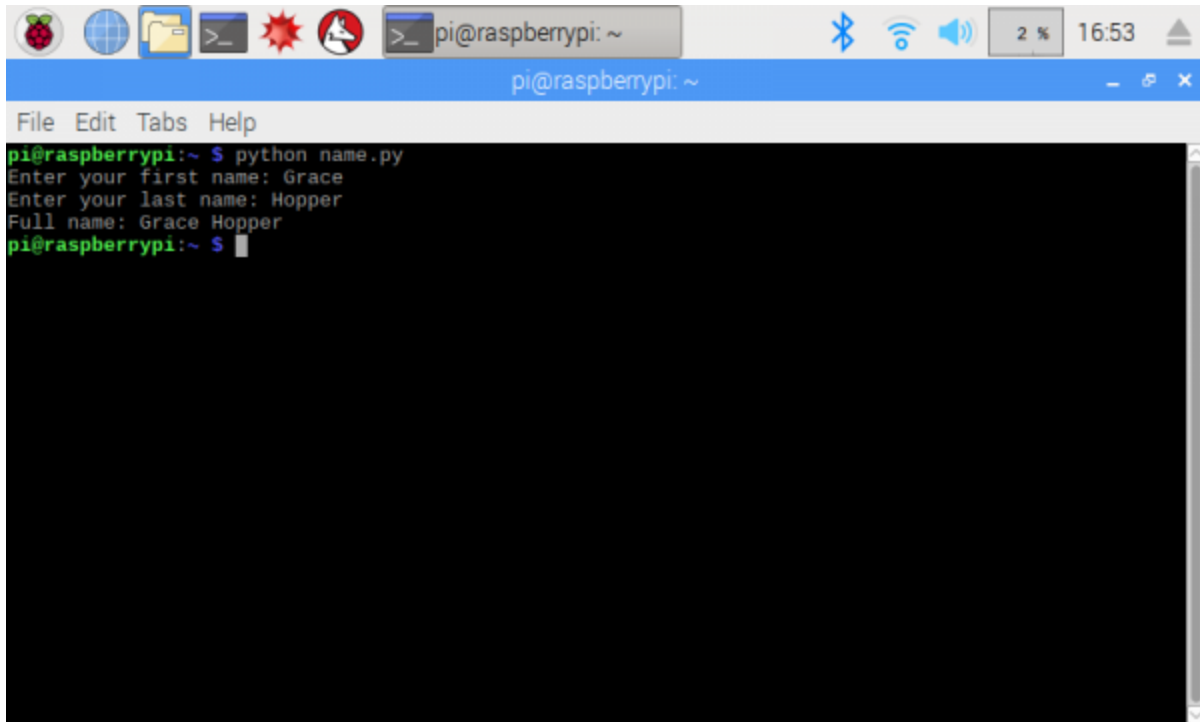
Note that you can use the `int()` function to turn a string into an integer (assuming the string is an integer).

Examples:

```
message= input("Type a message to yourself: ")  
print("You said:", message)
```

```
number=int(input("Type a number:"))  
print("You entered:", number)
```

✚ **Challenge:** Write a program that asks for the user's first name and last name (two separate `input()` calls) and then prints the user's first and last name on one line. An example of this program running should look like:

A screenshot of a Raspberry Pi desktop environment. The top panel shows various system icons including the Raspberry Pi logo, network status, volume, and battery. The main window is a terminal titled 'pi@raspberrypi: ~'. It displays the execution of a Python script named 'name.py'. The script prompts for a first name ('Grace') and a last name ('Hopper'), then prints the full name 'Grace Hopper'. The terminal window has a menu bar with 'File', 'Edit', 'Tabs', and 'Help'.

Indentation

White space (number of spaces) at the beginning of a line is important in Python. Statements that form a group together must have the same level of indentation (amount of white space in front of the line). This will be important when we get into control flow statements (e.g. `if`, `for`) and functions. If you have written programs in other languages before, you might be familiar with curly braces `{}`. In other languages, code in between these curly braces would form a group (or block) of code. In Python, a group (or block) of code is designated by the level of indentation of the individual lines of code.

Example:

```
COPY CODEanswer="yes"
guess= input("Is the sky blue? ")
if guess == answer:
    print("Correct!")
else:
    print("Try again")
```

`if` statements will be covered later, but note how the `print()` functions are indented, and thus form separate code groups underneath `if` and `else` statements.

Operators

An operator is a symbol that tells the interpreter to perform some mathematical, relational, or logical operation on one or more pieces of data and return the result.

Mathematical operators perform basic math operations on numbers:

Operator	Description	Example
+	Adds two numbers	2 + 3 returns 5
-	Subtracts one number from another	8 - 5 returns 3
*	Multiplies two numbers together	4 * 6 returns 24
**	Raises the first number to the power of the second number	2 ** 4 returns 16
/	Divides the first number by the second number	5 / 4 returns 1.25
//	Divides the two numbers and rounds down to the nearest integer (divide and floor)	5 / 4 returns 1
%	Divides the first number by the second number and gives the remainder (modulo)	19 % 8 returns 3

Logical operators compare two numbers and returns one of the *Boolean* values: `True` or `False`.


Operator	Description	Example
<	<code>True</code> if the first number is less than the second, <code>False</code> otherwise	5 < 3 returns <code>False</code>
>	<code>True</code> if the first number is greater than the second, <code>False</code> otherwise	5 > 3 returns <code>True</code>
<=	<code>True</code> if the first number is equal to or less than the second, <code>False</code> otherwise	2 <= 8 returns <code>True</code>
>=	<code>True</code> if the first number is equal to or greater than the second, <code>False</code> otherwise	2 >= 8 returns <code>False</code>
==	<code>True</code> if the first number is equal to the second, <code>False</code> otherwise	6 == 6 returns <code>True</code>

<code>!=</code>	True if the first number is not equal to the second, False otherwise (not equal)	<code>6 != 6</code> returns False
-----------------	--	-----------------------------------

Compound logical operators require Boolean inputs and give a Boolean answer.

Operator	Description	Example
<code>not</code>	Gives the opposite (True becomes False and vice versa)	<code>x = False; not x</code> returns True
<code>and</code>	Returns True if both operands are True, False otherwise	<code>x = True; y = False; x and y</code> returns False
<code>or</code>	Returns True if either of the operands are True, False otherwise	<code>x = True; y = False; x or y</code> returns True

Operator	Description	Example
<code>&</code>	Returns a 1 in each bit position for which the corresponding bits of both operands are 1 (bitwise AND)	<code>3 & 5</code> returns 1
<code> </code>	Returns a 1 in each bit position for which the corresponding bits of either or both operands are 1 (bitwise OR)	<code>3 5</code> returns 7
<code>^</code>	Returns a 1 in each bit position for which the corresponding bits of either but not both operands are 1 (bitwise XOR)	<code>3 ^ 5</code> returns 6
<code>~</code>	Inverts the bits in the given number (bitwise NOT)	<code>~5</code> returns -6
<code><<</code>	Shifts the bits of the first number to the left by the number of bits specified by the second number	<code>5 << 2</code> returns 20
<code>>></code>	Shifts the bits of the first number to the right by the number of bits specified by the second number	<code>5 >> 2</code> returns 1

 **Challenge:** Ask the user for two integers, and print the addition, subtraction, multiplication, division, and modulo of those numbers. For example, if you enter the numbers 6 and 7, the output should look like:

```
First number: 6
Second number: 7
13
```

```
-1
42
0.8571428571428571
6
```

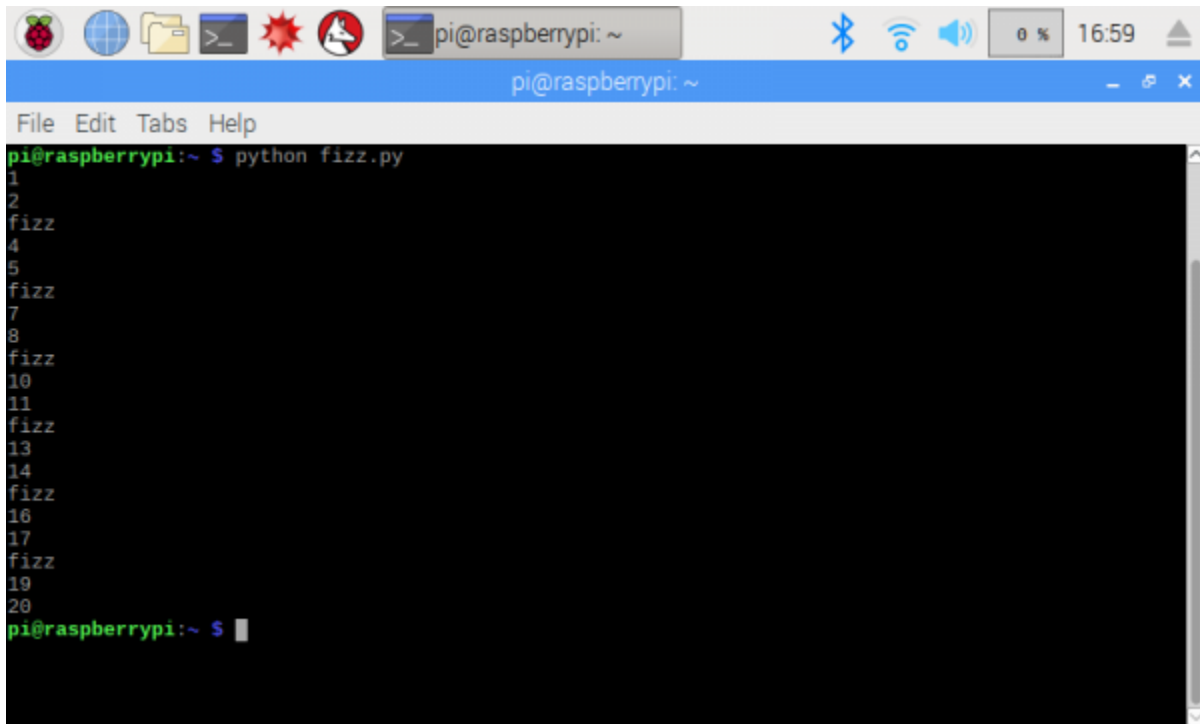
Control Flow

The Python interpreter executes statements in your code from the top to the bottom of the file, in sequential order. That is unless, of course, we employ some type of *control flow* statements to break this normal sequential flow.

We introduce the `range(x, y)` function in the examples below, which generates a list of numbers between the first number, `x` (inclusive), and the second number, `y` (exclusive).

Statement	Description	Example
<code>if elif else</code>	If a condition is true, execute the block of code underneath the <i>if statement</i> . If not, see if the condition is true in one or more <i>else if (elif)</i> statements. If one of those is true, execute the code block under that. Otherwise, execute the code block underneath the <i>else statement</i> . <code>elif</code> and <code>else</code> statements are optional.	<pre>number = 42 guess = int(input("Guess a number between 1-100: ")) if guess == number: print("You win!") elif guess < number: print("Nope") print("Too low") else: print("Nope") print("Too high") print("Run the program to try ...")</pre>
<code>while</code>	A <i>while loop</i> executes the block of code underneath it repeatedly as long as the condition is true.	<pre>counter = 15 while counter >= 5: print(counter) counter = counter - 1</pre>
<code>for..in</code>	Iterate over a sequence of numbers or objects. The variable declared in a <i>for loop</i> assumes the value of one of the numbers (or objects) during each iteration of the loop.	<pre>for i in range(1, 11): print(i)</pre>
<code>break</code>	Use the <i>break statement</i> to exit out of a loop.	<pre>while True: message = input("Tell me when to stop: ") if message == "stop": break print("OK")</pre>
<code>continue</code>	The <i>continue statement</i> works similar to <i>break</i> , but instead of exiting the loop, it stops the current iteration and returns to the top of the loop.	<pre>for i in range(1, 6): if i == 3: continue print(i)</pre>

🚩 **Challenge:** Write a program that prints integers counting up from 1 to 20, except that for every multiple of 3 (3, 6, 9, etc.), the word "fizz" is printed instead. The output should look like the following:



```
pi@raspberrypi:~ $ python fizz.py
1
2
fizz
4
5
fizz
7
8
fizz
10
11
fizz
13
14
fizz
16
17
fizz
19
20
pi@raspberrypi:~ $
```

Functions

Functions allow you to name a block of code and then reuse that code by calling its name. You can pass data to a function through variables known as *parameters* (note that the variables in the function definition are called *parameters* whereas the actual data itself being passed are known as *arguments*). Data can also be passed back to the calling statement using the `return` statement. An example of a function definition would look like:

```
def functionName(parameter1, parameter2):
    # Code goes here
```

You can call this function elsewhere in your code with `functionName(argument1, argument2)`. Note that variables declared inside the function definition are known as having *local scope*. This means that they cannot be accessed outside of that function. Variables declared at the top level of the program (i.e. outside any functions, loops, or classes) are known as having *global scope* and can be accessed anywhere in the program (including inside functions).

Important: Any functions you create must be defined *before* you use them! If you try to call a function higher up in the code (before its `def` definition), you'll likely get an error such as:

```
NameError: name 'FUNCTION_NAME' is not defined
```

Python has a number of built-in functions that can help you (we've already seen three: `print()`, `int()`, and `range()`). A list of these functions can be found in [the Python Tutorial](#).

Example:

```
def add(x, y):
    sum = x + y
    return sum

print(add(2, 3))
```

Challenge: Starting with the code below, implement the `sumTo()` function that takes an integer as a parameter (n), sums all the whole numbers from 1 to n (including n), and returns the sum. You may assume that the input, n , will always be a positive whole number (do not worry about handling negative numbers).

```
def sumTo(n):
    # YOUR CODE GOES HERE

    # Should be 1
    print(sumTo(1))

    # Should be 45
    print(sumTo(9))

    # Should be 5050
    print(sumTo(100))
```

Objects

We have not talked about *objects* yet, but in reality, you've been using them all along. The secret to Python is that everything is an object. That's right: *everything*, including functions and integers.

An object is simply a collection of data stored somewhere in your computer's memory. What makes an object special in a programming language is the ability for it to store information *and* perform actions. We've already seen an object's ability to store data (for example, `a = 3` means that `a` is an integer object and stores the information 3). But how do we get an object to perform an action? Objects are given a set of functions as defined by their *class*, which acts as a blueprint--telling the objects what they can and can't do or information it can and can't hold. When we talk about functions within a class (or objects), we call them *methods*.

For example, we can test if a floating point number is an integer by using the built-in `is_integer()` method. Note that this method is only accessible from `float` objects! We can't call `is_integer()` on its own, so we use the dot-notation (`.`) to have the `float` object call the `is_integer()` method from within itself.

Example:

```
a = 3.0
b = 7.32

print(a.is_integer())
print(b.is_integer())
```

Note that we can't use an integer as a float! For example, if we said `c = 8`, then `c` is an integer, not a float! If `c` is an integer, there is no `.is_integer()` method in integers, so calling `c.is_integer()` would fail (and throw an interpreter error). Try it! To force an integer value to be a floating point number, we simply add `.0` after it (just like we did with `a = 3.0`).

Challenge: Modify the code below so that the phrase stored in `my_string` is converted to all lower case letters and printed to the terminal.

```
COPY CODEmy_string="THiSiS A TESt!"
```

```
# Should print "this is a test!"
```


```
# YOUR CODE GOES HERE
```

Data Structures

In addition to *variables* and *objects*, Python has four other ways to store data: *list*, *tuple*, *dictionary*, *set*. These structures hold a collection of related data, and each of them has a slightly different way of interacting with it.

Structure	Description	Example
List	A <i>list</i> is a sequence of ordered items. You can access items in a list using brackets <code>[]</code> and an index (e.g. <code>list[2]</code>). Note that indices are 0-based, which means <code>list[0]</code> will access the first item in the list. Because lists can be modified, they are known as <i>mutable</i> .	<pre>my_list = [1, 5, 73, -3] my_list[2] = -42 # Get third item in list print(my_list[2]) # Get all but first item in list print(my_list[1:])</pre>
Tuple	A <i>tuple</i> works just like a list (an ordered set). The difference is that a tuple is <i>immutable</i> , which means you cannot change the values once they are set. A tuple is usually specified by parentheses <code>()</code> . While the parentheses are not necessary, they are highly recommended to make your code easier to read.	<pre>my_tuple = ("bird", "plane", 5, "train") # Get one item from the tuple print(my_tuple[0]) # Get a tuple of second and third items print(my_tuple[1:3]) # Can't do this because tuples are immutable! my_tuple[1] = "skyscraper"</pre>
Dictionary	A <i>dictionary</i> is a collection of associated <i>key</i> and <i>value</i> pairs. Similar to how a phonebook works: you look up someone's name (key) and you find their phone number (value). Dictionaries are defined by curly braces <code>{}</code> , and key/value	<pre>my_dictionary = {"name": "Bruce", "aka": "The Hulk"} my_dictionary["name"] = "Banner" print(my_dictionary["name"])</pre>

	pairs are separated by a colon <code>:</code> . Dictionaries are mutable.	
Set	A <i>set</i> is a mutable, unordered collection with no duplicate elements. Sets are optimized for determining if an item is in the set (and you don't care about the order of items). Sets are great if you want to implement any sort of <i>set theory</i> in Python.	<pre> my_set_1 = set(["orange", "banana"]) my_set_2 = set(["apple"]) my_set_2.add("orange") print(my_set_1) print(my_set_2) print("apple" in my_set_2) print("strawberry" in my_set_2) print(my_set_1.union(my_set_2)) print(my_set_1.intersection(my_set_2)) </pre>

 **Challenge:** Starting with the code below, implement the `average()` function to compute the average of the numbers given to it in list form. Hint: you will probably want to use the [len\(\) function](#).

```

def average(num_list):
# YOUR CODE GOES HERE

# Should print 5.0
list_1 = [4, 7, 9, 0]
print(average(list_1))

# Should print 4.4063333333333
list_2 = [-3.2, 6.419, 10]
print(average(list_2))

# Should print 42.0
list_3 = [42]
print(average(list_3))

```

Modules

Modules are another way to reuse code and help you organize your program. They are simply files that are *imported* into your main program. After importing a module, you can use a module in much the same way you would an object: access constants and functions using the dot-notation.

Example:

Save the following file as *stringmod.py*:

```

a = 42

def string_to_list(s):
    c_list = []
    for c in s:


```

```
c_list.append(c)
return c_list
```

In the same folder as *stringmod.py*, run the following code (either in the interpreter or saved as a file):

```
import stringmod

s = "Hello!"
print(stringmod.a)
print(stringmod.string_to_list(s))
```

 **Challenge:** Python comes with several standard modules that you can import into your program. One of them is the [math module](#), which you can use with `import math`. Use the constants and functions found in the `math` module to perform the following actions:

- Print the ceiling of 3.456 (should be 4)
- Print the square root of 9216 (should be 96.0)
- Calculate and print the area of a circle whose radius is 2 (should be 12.566370614359172)

Finding and Fixing Bugs

Programs almost never work right away, so don't sweat it! The art and science of finding and fixing problems in your code is known as *debugging*. The most helpful debugging tool is the standard Python output. If there is a problem with your code, it will likely tell you where the error is and what's wrong with it.

For example, let's take the following code snippet. In it, we forget to indent our code underneath our *for loop*.

```
a = [1, 2, 3, 4]

for n in a:
print(n)
```

If you run this code, you'll see that the Python interpreter is super helpful by saying it was looking for an indented piece of code around line 4.

```
File "test_01.py", line 4
print(n)
^
```

IndentationError: expected an indented block

Here's another example. Can you spot the error?

```
a = [1, 2, 3, 4]

for n in a
print(n)
```

We forgot the colon after the *for loop*! The interpreter will let us know by telling us:

File "test_01.py", line 3

```
for n in a
    ^
```

SyntaxError: invalid syntax

"Invalid syntax" is a little vague, but it tells you to look around line 3 for something that might be wrong, such as a missing colon, too many parentheses, a single quote instead of a double quote, etc.

What happens if your code runs, but it doesn't output the value(s) you expect? This is on you, the programmer, to find and fix! Adding `print()` statements throughout your code can help you identify where something might have went wrong.

Try running this code:

```
for i in range(1, 10):
```

```
    if i == 10:
```

```
        print("end")
```

Why don't you see "end" appear in the terminal? To help diagnose this problem, we can add a `print()` statement to see what's going on:

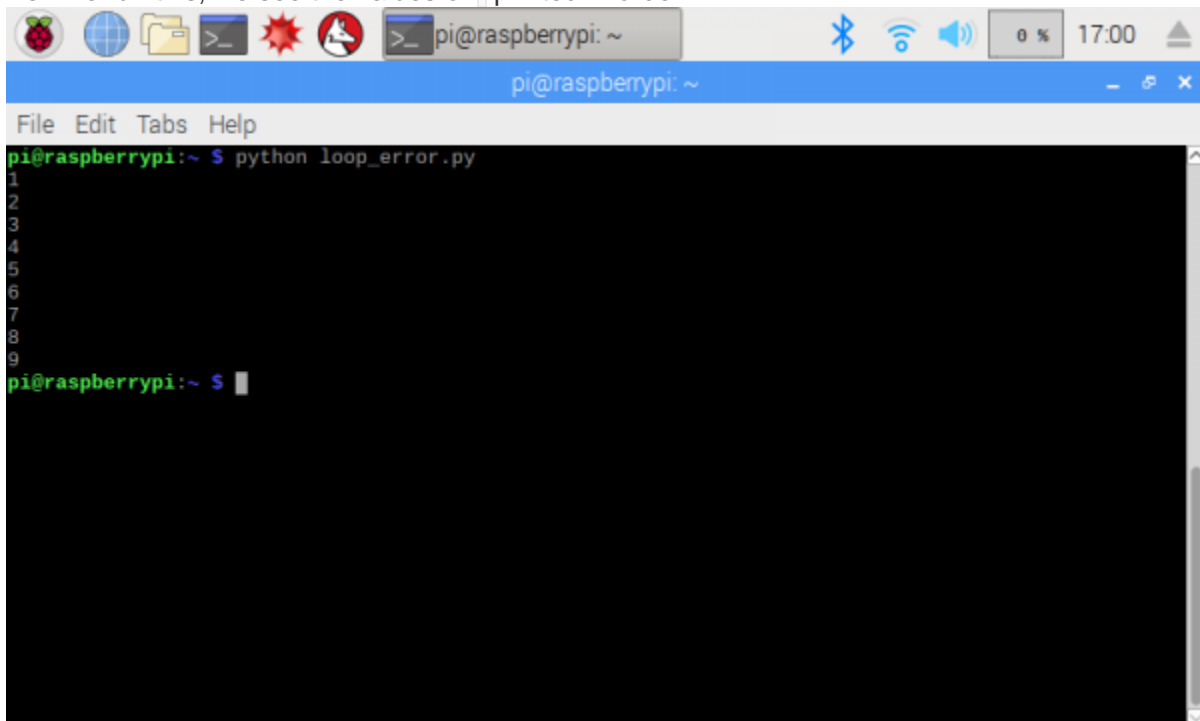
```
for i in range(1, 10):
```

```
    print(i)
```

```
    if i == 10:
```

```
        print("end")
```

When we run this, we see the values of `i` printed in order.



```
pi@raspberrypi: ~
File Edit Tabs Help
pi@raspberrypi:~ $ python loop_error.py
1
2
3
4
5
6
7
8
9
pi@raspberrypi:~ $
```

A-ha! It turns out that `i` never reaches 10. That's because the second number in the `range()` function is exclusive. If we need it to count to 10, then we should change it to:

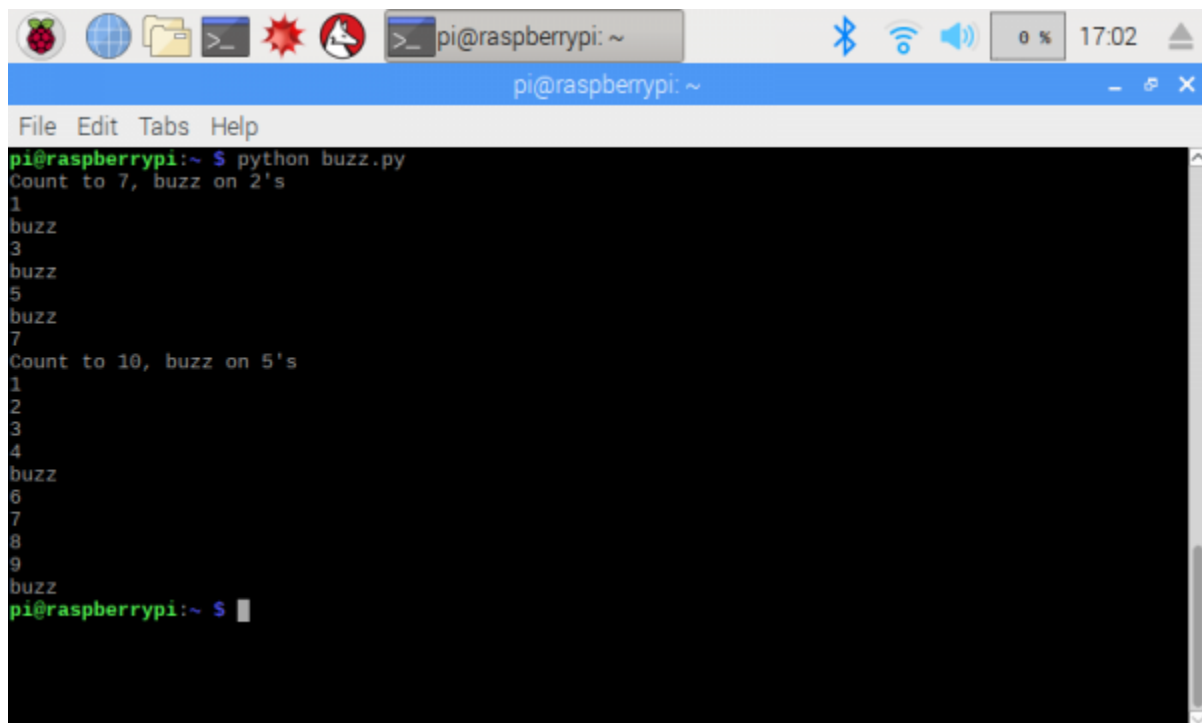
```
for i in range(1, 11):  
    if i==10:  
        print("end")
```

Thanks for the help, `print()`!

🚩 **Challenge:** Find the 7 errors in the program below. When you fix the errors and run it, it should print numbers from 1 to the first argument, replacing multiples of the second argument with the word "buzz".

```
print("Count to 7, buzz on 2's")  
buzz(7, 2)  
  
print('Count to 10, buzz on 5's')  
buzz(10, 5)  
  
def buzz(n):  
    for i in range(1, n):  
        if i % 2 == 0:  
            print("buzz")  
        else  
            print(n)
```

Here is the program running successfully:



```
pi@raspberrypi: ~  
File Edit Tabs Help  
pi@raspberrypi:~ $ python buzz.py  
Count to 7, buzz on 2's  
1  
buzz  
3  
buzz  
5  
buzz  
7  
Count to 10, buzz on 5's  
1  
2  
3  
4  
buzz  
6  
7  
8  
9  
buzz  
pi@raspberrypi:~ $
```

Python (RPi.GPIO) API

We'll use the [RPi.GPIO module](#) as the driving force behind our Python examples. This set of Python files and source is **included with Raspbian**, so assuming you're running that most popular Linux distribution, you don't need to download anything to get started.

On this page we'll provide an overview of the basic function calls you can make using this module.

Setup Stuff

In order to use RPi.GPIO throughout the rest of your Python script, you need to put this statement at the **top of your file**:

```
import RPi.GPIO as GPIO
```

That statement "includes" the RPi.GPIO module, and goes a step further by providing a local name - `GPIO` -- which we'll call to reference the module from here on.

Pin Numbering Declaration

After you've included the RPi.GPIO module, the next step is to determine which of the two **pin-numbering schemes** you want to use:

1. `GPIO.BOARD` -- Board numbering scheme. The pin numbers follow the pin numbers on header P1.
2. `GPIO.BCM` -- Broadcom chip-specific pin numbers. These pin numbers follow the lower-level numbering system defined by the Raspberry Pi's Broadcom-chip brain.

If you're using the Pi Wedge, we recommend using the `GPIO.BCM` definition -- those are the numbers silkscreened on the PCB. The `GPIO.BOARD` may be easier if you're wiring directly to the header. To specify in your code which number-system is being used, use the `GPIO.setmode()` function. For example...

```
GPIO.setmode(GPIO.BCM)
```

...will activate the Broadcom-chip specific pin numbers.

Both the `import` and `setmode` lines of code are **required**, if you want to use Python.

Setting a Pin Mode

If you've used Arduino, you're probably familiar with the fact that you have to declare a "pin mode" before you can use it as either an input or output. To set a pin mode, use the `setup([pin], [GPIO.IN, GPIO.OUT])` function. So, if you want to set pin 18 as an output, for example, write:

```
GPIO.setup(18, GPIO.OUT)
```

Remember that the pin number will change if you're using the board numbering system (instead of 18, it'd be 12).

Outputs

Digital Output

To write a pin high or low, use the `GPIO.output([pin], [GPIO.LOW, GPIO.HIGH])` function. For example, if you want to set pin 18 high, write:

```
GPIO.output(18, GPIO.HIGH)
```

Writing a pin to `GPIO.HIGH` will drive it to 3.3V, and `GPIO.LOW` will set it to 0V. For the lazy, alternative to `GPIO.HIGH` and `GPIO.LOW`, you can use either `1`, `True`, `0` or `False` to set a pin value.

PWM ("Analog") Output

PWM on the Raspberry Pi is about as limited as can be -- one, single pin is capable of it: 18 (i.e. board pin 12).

To initialize PWM, use `GPIO.PWM([pin], [frequency])` function. To make the rest of your script-writing easier you can assign that instance to a variable. Then use `pwm.start([duty cycle])` function to set an initial value. For example...

```
pwm= GPIO.PWM(18, 1000)
pwm.start(50)
```

...will set our PWM pin up with a frequency of 1kHz, and set that output to a 50% duty cycle.

To adjust the value of the PWM output, use the `pwm.ChangeDutyCycle([duty cycle])` function. `[duty cycle]` can be any value between 0 (i.e 0%/LOW) and 100 (i.e.e 100%/HIGH). So to set a pin to 75% on, for example, you could write:

```
pwm.ChangeDutyCycle(75)
```

To turn PWM on that pin off, use the `pwm.stop()` command.
Simple enough! Just don't forget to set the pin as an output before you use it for PWM.

Inputs

If a pin is configured as an input, you can use the `GPIO.input([pin])` function to read its value. The `input()` function will return either a `True` or `False` indicating whether the pin is HIGH or LOW. You can use an if statement to test this, for example...

```
if GPIO.input(17):
    print("Pin 11 is HIGH")
else:
    print("Pin 11 is LOW")
```

...will read pin 17 and print whether it's being read as HIGH or LOW.

Pull-Up/Down Resistors

Remember back to the `GPIO.setup()` function where we declared whether a pin was an input or output? There's an optional third parameter to that function, which you can use to set pull-up or pull-down resistors. To use a pull-up resistor on a pin, add `pull_up_down=GPIO.PUD_UP` as a third parameter in `GPIO.setup`. Or, if you need a pull-down resistor, instead use `pull_up_down=GPIO.PUD_DOWN`.

For example, to use a pull-up resistor on GPIO 17, write this into your setup:

```
GPIO.setup(17, GPIO.IN, pull_up_down=GPIO.PUD_UP)
```

If nothing is declared in that third value, both pull-resistors will be disabled.

Delays

If you need to slow your Python script down, you can add delays. To incorporate delays into your script, you'll need to include another module: `time`. This line, at the top of your script, will do it for you:

```
include time
```

Then, throughout the rest of your script, you can use `time.sleep([seconds])` to give your script a rest. You can use decimals to precisely set your delay. For example, to delay 250 milliseconds, write:

```
time.sleep(0.25)
```

The time module includes all sorts of useful functions, on top of `sleep`.

Code Part 1: Blinking an LED

Depending on your version of Raspbian, you may or may not have to install the RPi.GPIO package (e.g. Raspbian Lite does not come with some Python packages pre-installed). In a terminal, enter the following:

```
pip install rpi.gpio
```

In a new file, enter the following code:

```
COPY CODEimport time
import RPi.GPIO as GPIO

# Pin definitions
led_pin=12

# Suppress warnings
GPIO.setwarnings(False)

# Use "GPIO" pin numbering
GPIO.setmode(GPIO.BCM)

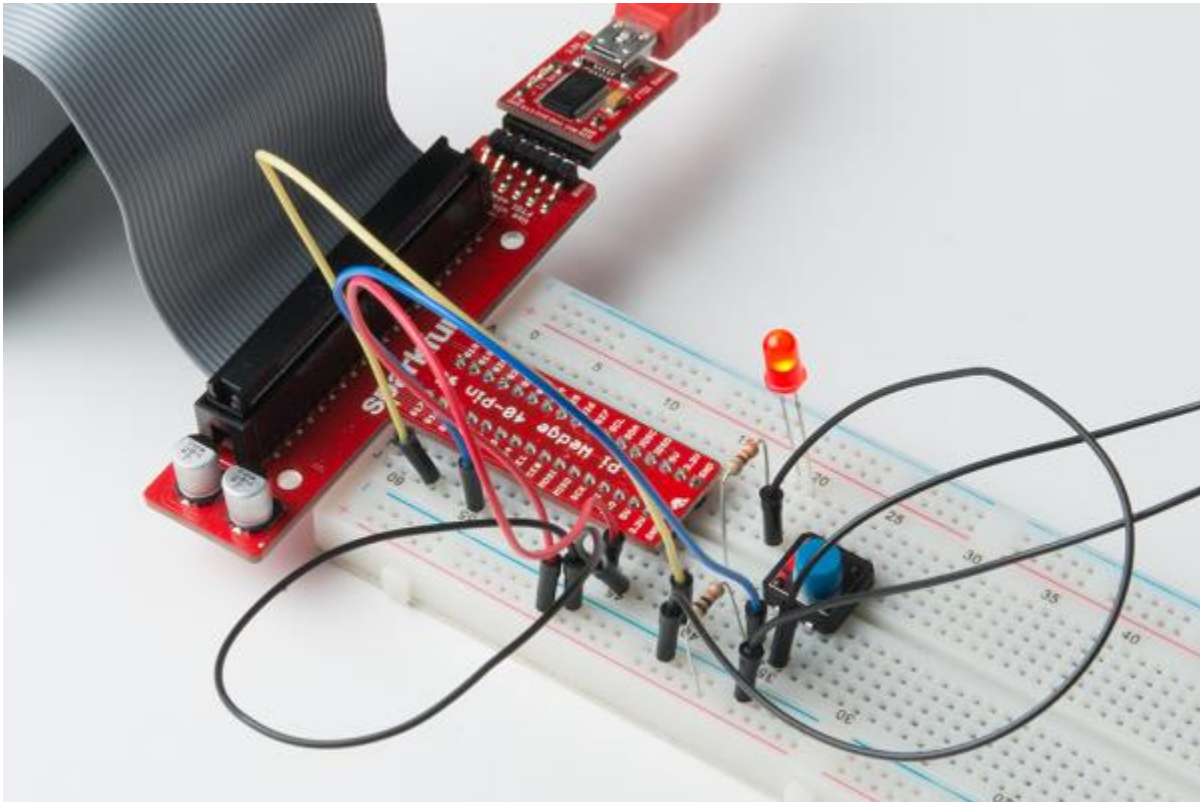
# Set LED pin as output
GPIO.setup(led_pin, GPIO.OUT)

# Blink forever
while True:
    GPIO.output(led_pin, GPIO.HIGH) # Turn LED on
    time.sleep(1)                  # Delay for 1 second
    GPIO.output(led_pin, GPIO.LOW) # Turn LED off
    time.sleep(1)                  # Delay for 1 second
```

Save the file (I named my file *blink.py*). Run the code from the terminal by entering:


```
python blink.py
```

You should see your LED begin to blink on and off every second:



Once you've gotten bored of watching the LED, end the program by pressing `ctrl + c`.

Troubleshooting: If you see the message "ModuleNotFoundError: No module named 'rpi'" you will need to install the RPi.GPIO package by entering `pip install RPi.GPIO` in a terminal.

Code to Note:

To control hardware from the Raspberry Pi, we rely on the [RPi.GPIO module](#). This module (likely known as a "library" in other languages) is specifically designed to help us toggle pins and talk to other pieces of hardware. Lucky for us, it comes pre-packaged with Raspbian!

In the first two lines, you see that we imported modules, but we added a few things onto those imports. First up, we used the keyword `as`:

```
import RPi.GPIO as GPIO
```

RPi.GPIO is the name of the module. By saying `as GPIO`, we change how we want to refer to that module in the rest of the program. This allows us to type

```
GPIO.output(led_pin, GPIO.HIGH)
```

instead of the much longer

```
RPi.GPIO.output(led_pin, RPi.GPIO.HIGH)
```

While it's generally not a good idea to disable warnings while coding, we added the following line:

```
GPIO.setwarnings(False)
```

Without it, you'll get a warning from the interpreter when you try to run the blink program again:

```
blink.py:14: RuntimeWarning: This channel is already in use, continuing anyway. Use
GPIO.setwarnings(False) to disable warnings.
GPIO.setup(led_pin, GPIO.OUT)
```

This is because we did not shut down the GPIO 12 pin nicely when we exited the program. To do this, we would want to add a `GPIO.cleanup()` line at the end of our program. However, because we wrote our program to run forever, we have to interrupt the program to stop it (and a call to `cleanup()` would never occur). For the time being, it's enough to just ignore the warnings.

🔧 **Challenge:** Change the program to make the LED blink like a heartbeat: 2 quick flashes in succession and then a longer delay.

Raspberry Pi 3 GPIO Pinout

GPIO_GEN	Functions	GPIO	Pin	Pin	GPIO	Functions	GPIO_GEN
		3.3V	1		2	5V	
	SDA1 (I ² C)	GPIO2	3		4	5V	
	SCL1 (I ² C)	GPIO3	5		6	GND	
GCLK		GPIO4	7		8	GPIO14	TXD0 (UART)
		GND	9		10	GPIO15	RXD0 (UART)
GEN0		GPIO17	11		12	GPIO18	PWM0, CLK (PCM)
GEN2		GPIO27	13		14	GND	
GEN3		GPIO22	15		16	GPIO23	
		3.3V	17		18	GPIO24	
	MOSI (SPI)	GPIO10	19		20	GND	
	MISO (SPI)	GPIO9	21		22	GPIO25	
	SCLK (SPI)	GPIO11	23		24	GPIO8	CE0 (SPI)
		GND	25		26	GPIO7	CE1 (SPI)
		ID_SD	27		28	ID_SC	
		GPIO5	29		30	GND	
		GPIO6	31		32	GPIO12	PWM0
	PWM1	GPIO13	33		34	GND	
	FS (PCM), PWM1	GPIO19	35		36	GPIO16	
		GPIO26	37		38	GPIO20	DIN (PCM)
		GND	39		40	GPIO21	DOUT (PCM)

Code Part 2: Fading an LED with PWM

We've seen how to turn an LED on and off, but how do we control its brightness levels? An LED's brightness is determined by controlling the amount of current flowing through it, but that requires a lot more hardware components. A simple trick we can do is to flash the LED faster than the eye can see!

By controlling the amount of time the LED is on versus off, we can change its perceived brightness. This is known as *pulse width modulation* (PWM). We have two separate PWM channels for our use: PWM0 and PWM1. We can output a PWM signal on PWM0, which will show up on GPIO12 and GPIO18. Additionally, PWM1 controls the signal for GPIO13 and GPIO19.

Copy the following code into a file (e.g. *pwm.py*):

```
import time
import RPi.GPIO as GPIO

# Pin definitions
led_pin=12

# Use "GPIO" pin numbering
GPIO.setmode(GPIO.BCM)

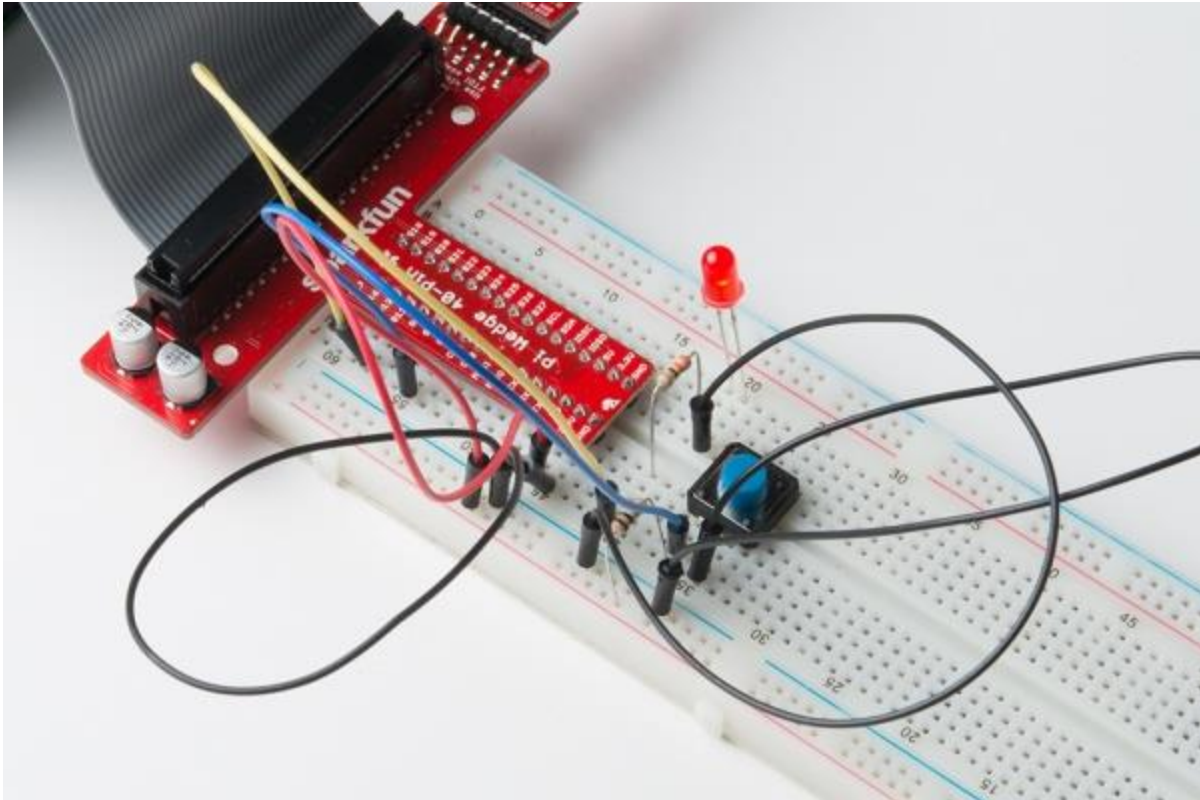
# Set LED pin as output
GPIO.setup(led_pin, GPIO.OUT)

# Initialize pwm object with 50 Hz and 0% duty cycle
pwm= GPIO.PWM(led_pin, 50)
pwm.start(0)

# Set PWM duty cycle to 50%, wait, then to 90%
pwm.ChangeDutyCycle(50)
time.sleep(2)
pwm.ChangeDutyCycle(90)
time.sleep(2)

# Stop, cleanup, and exit
pwm.stop()
GPIO.cleanup()
```

Run it (e.g. `python pwm.py`), and you should see the LED start dim, wait 2 seconds, grow brighter, wait another 2 seconds, and then turn off before exiting the program.



Code to Note:

In the first part, we use the `.output()` function of the `GPIO` module to toggle the LED. Here, we create `PWM` object and store it in the variable `pwm`. We do this with the line:

```
pwm= GPIO.PWM(led_pin, 50)
```

From there, we can control the PWM by calling methods within that object. For example, we change the brightness by calling:

```
pwm.ChangeDutyCycle(t)
```

where `t` is some number between 0-100 (0 being off and 100 being always on). Putting in the number 50 would mean that the LED is on half the time and off the other half the time (it's just toggling so fast that you can't see it!).

Also, we left out the `GPIO.setwarnings()` call, since we can actually call `GPIO.cleanup()` at the end of our program! If you try to run the PWM code twice, you should not see any warnings.

- 🔧 **Challenge:** Make the LED slowly fade from off to fully bright over the course of about 2 seconds. Once it has reached maximum brightness, the LED should turn off and repeat the fading process again. Have the LED fade on over and over again forever.