Community Experience Distilled

Python Programming with Raspberry Pi Zero

Champion the skills of Python programming using the small yet powerful Raspberry Pi Zero

Sai Yamanoor
Srihari Yamanoor

[PACKT] PUBLISHING open source* community experience distilled

Table of Contents

1

Getting Started with Python and the Raspberry Pi Zero

Over the last few years, the Raspberry Pi family of Single Board Computers has proved to be a revolutionary set of tools of learning, fun and several serious projects! People over the world are now equipped with the means to learn computer architecture, computer programming, robotics, sensory systems, home automation and much more, with ease and without blowing a hole in their wallets. This book in particular, hopes to help you the reader, in the journey to learn programming in Python through Raspberry Pi. Among programming languages, Python is simultaneously one of the simplest and easiest to learn as well as one of the most versatile languages. Join us over the next few chapters as we first familiarize ourselves with the Raspberry Pi Zero, a unique and excitingly simple and cheap computer and Python, gradually building projects of increasing challenge and complexity.

...

Target audience of the book

This book intended for hobbyists and makers who are getting started with learning Python programming. We assume that you have a basic understanding of using electronic prototyping tools like a breadboard, soldering iron etc. We also assume that you are able to identify electronics components like LEDs, transistors etc. Some basic experience in using the Linux and its command line terminal is helpful but not necessary.

Let's get started!

In the first chapter, we will learn about the Raspberry Pi Zero, set things up for learning Python with this book and write our first piece of code in Python.

Things needed for this book

The following items are needed for this book. The sources provided are just a suggestion. The reader is welcome to buy them from an equivalent alternative source:

Name	**Link**	**Cost (in USD)**
Raspberry Pi Zero	(The purchase of the Pi would be discussed separately)	$5.00
USB Hub	http://amzn.com/B003M0NURK	$7.00 approx
USB OTG cable	https://www.adafruit.com/products/1099	$2.50
mini HDMI to HDMI	https://www.adafruit.com/products/1358	$6.95
USB WiFi adapter	http://amzn.com/B00LWE14TO	$9.45
micro USB power supply	http://amzn.com/B00DZLSEVI	$3.50
Electronics starter kit (or similar)	http://amzn.com/B00IT6AYJO	$25.00
2 x 20 headers	https://www.adafruit.com/products/2822	$0.95
NOOBS microSD card or a blank 8 GB microSD card	http://amzn.com/B00ENPQ1GK	$13.00

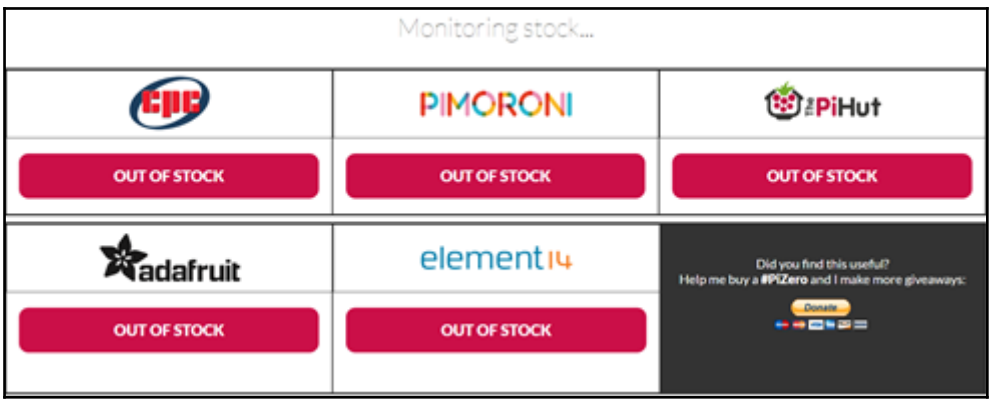
The other items needed for this include a USB mouse, USB keyboard and a monitor with HDMI output or DVI output. We will also need an HDMI cable (or DVI to HDMI cable if the monitor has an DVI output). Some vendors like the Pi Hut sell the Raspberry Pi Zero accessories as a kit (e.g. https://thepihut.com/collections/raspberry-pi-accessories/products/raspberry-pi-zero-essential-kit)

Note: Apart from the components mentioned in this section, we will also be discuss certain features of the Raspberry Pi Zero and Python programming using additional components like sensors, GPIO expanders etc. These components are optional but definitely useful while learning the different aspects of Python programming.

Note: The Electronics starter kit mentioned in the bill of materials is just an example. Feel free to order any beginners electronics kit (that contains a similar mix of electronic components).

Buying the Raspberry Pi Zero

The Raspberry Pi Zero is sold by distributors like the Newark Element 14, Pi Hut, Adafruit, and so on. At the time of writing this book, we encountered difficulties in buying a Raspberry Pi Zero. We recommend that monitoring websites like whereismypizero.com to find out when the Raspberry Pi Zero becomes available. We believe that it is rare to locate the Pi Zero because of its popularity. We are not aware if there might be abundant stock of the Raspberry Pi Zero in the future.



Pi Zero availability information provided by whereismypizero.com

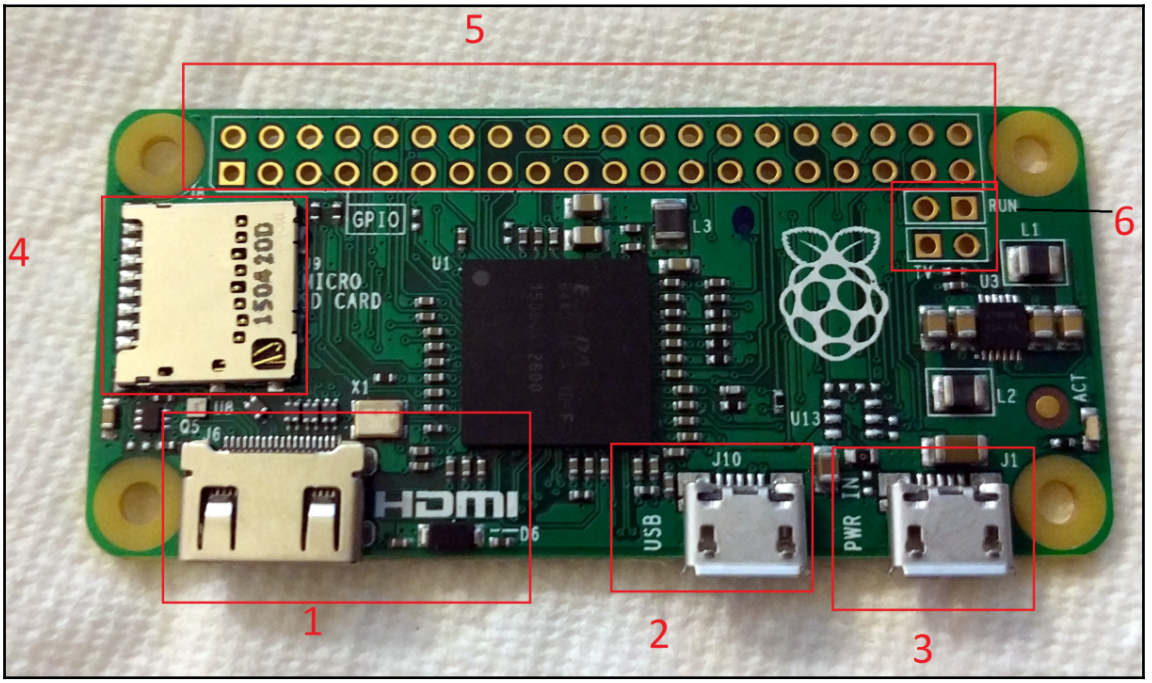
Introduction to the Raspberry Pi Zero

The **Raspberry Pi Zero** is a small computer that costs about $5 and smaller than a credit card, designed by the Raspberry Pi Foundation (a non-profit with the mission to teach computer science to students especially those who lack of access to the requisite tools). The Raspberry Pi Zero was preceded by the Raspberry Pi Models 1 and 2. A detailed history of the Raspberry Pi and the different models of the Raspberry Pi is available onhttp://elinux.org/RPi_General_History. The Raspberry Pi Zero was released on 26th November 2015 (Thanksgiving Day 2015).

One of the authors of this book, Sai Yamanoor drove from San Francisco to Los Angeles (700+ miles for a round trip in one day) on the day after Thanksgiving to buy the Raspberry Pi Zero from a local store.

Features of the Raspberry Pi Zero

The Raspberry Pi Zero is powered by a 1GHz BCM2835 processor and 512 MB RAM. Let us briefly discuss the features of the Raspberry Pi Zero using the picture marked with numbered rectangles.



The Raspberry Pi Zero

1. **mini HDMI interface:** The mini HDMI interface is used to connect a display to the Raspberry Pi Zero. The HDMI interface can be used to drive a display of maximum resolution of 1920×1080 pixels
2. **USB – 'On the Go' (USB – OTG) interface:** In the interest of keeping things low cost, the Raspberry Pi Zero comes with a **USB OTG** interface. This interface

enables interfacing USB devices like a mouse, keyboard, and so on. Using a USB OTG to USB-A female converter. We need a USB hub to interface any USB accessory.

3. **Power supply:** The micro-B USB adapter is used to power the Raspberry Pi zero and draws about a maximum of 200mA of current.
4. **microSD card slot:** The Raspberry Pi's operating system resides in a microSD card and the bootloader on the processor loads it upon powering up.
5. **GPIO interface:** The Raspberry Pi Zero comes with a 40 pin GPIO header (General Purpose Input Output) that is arranged in two rows of 20 pins. The GPIO header is used to interface sensors, control actuators and interface appliances. The GPIO header also consists of communication interfaces like UART, I2C, and so on. We will discuss the GPIO in detail in the second chapter.
6. **Run and TV pins:** There are two pins labeled **Run** below the GPIO header. These pins are used to reset the Raspberry Pi using a small tactile switch/push button. The TV pin is used to provide an analog output.

All these features of the Raspberry Pi have enabled them to be used by hobbyists in projects involving home automation, holiday decorations and more, limited only by your imagination. Scientists have used them in experiments including tracking of bees, tracking wildlife, perform computation intensive experiments. Engineers have used the Pi in building robots, mine bitcoins, check internet speeds to send twitter messages when the speeds are slow, and to order pizza!

Setup of the Raspberry Pi Zero

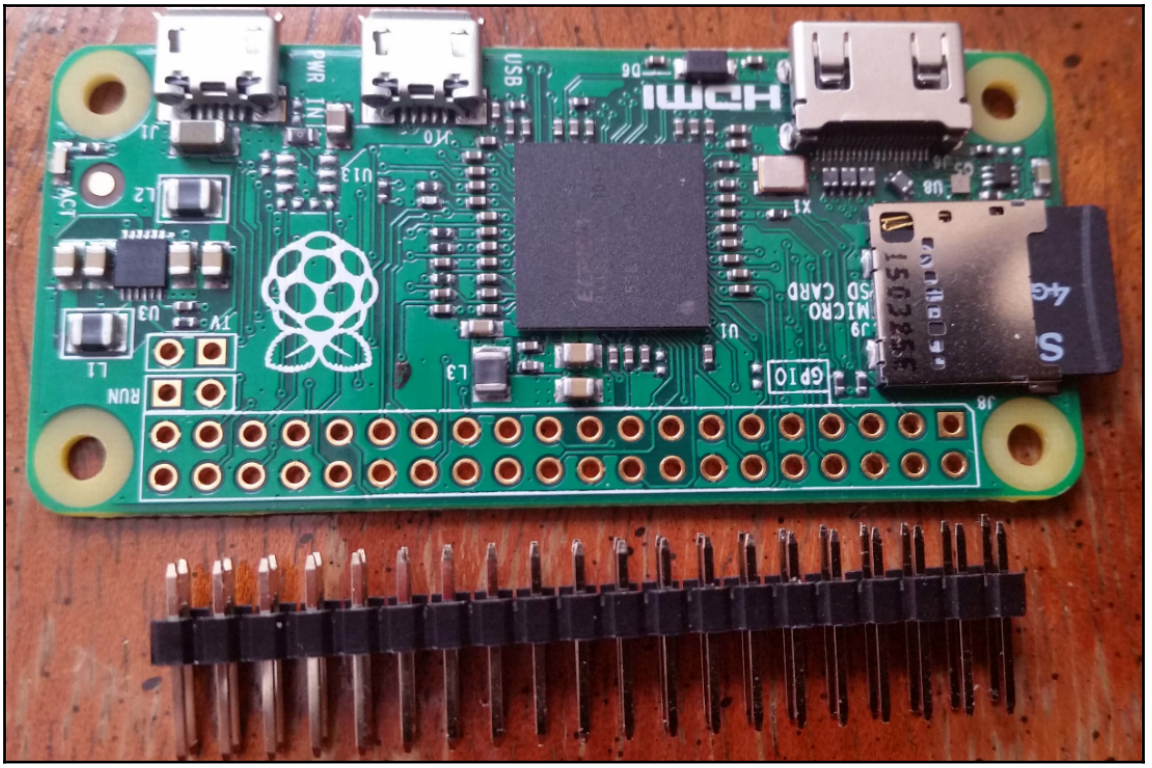
In this section, we will solder some headers onto the Raspberry Pi and load the operating system onto a microSD card and fire the Raspberry Pi Zero for the first example.

Soldering the GPIO headers

In this book, we will discuss the different aspects of Python programming using the Raspberry Pi's GPIO pins. The Raspberry Pi Zero ships without the GPIO header pins. Let us go ahead and solder the GPIO pins.

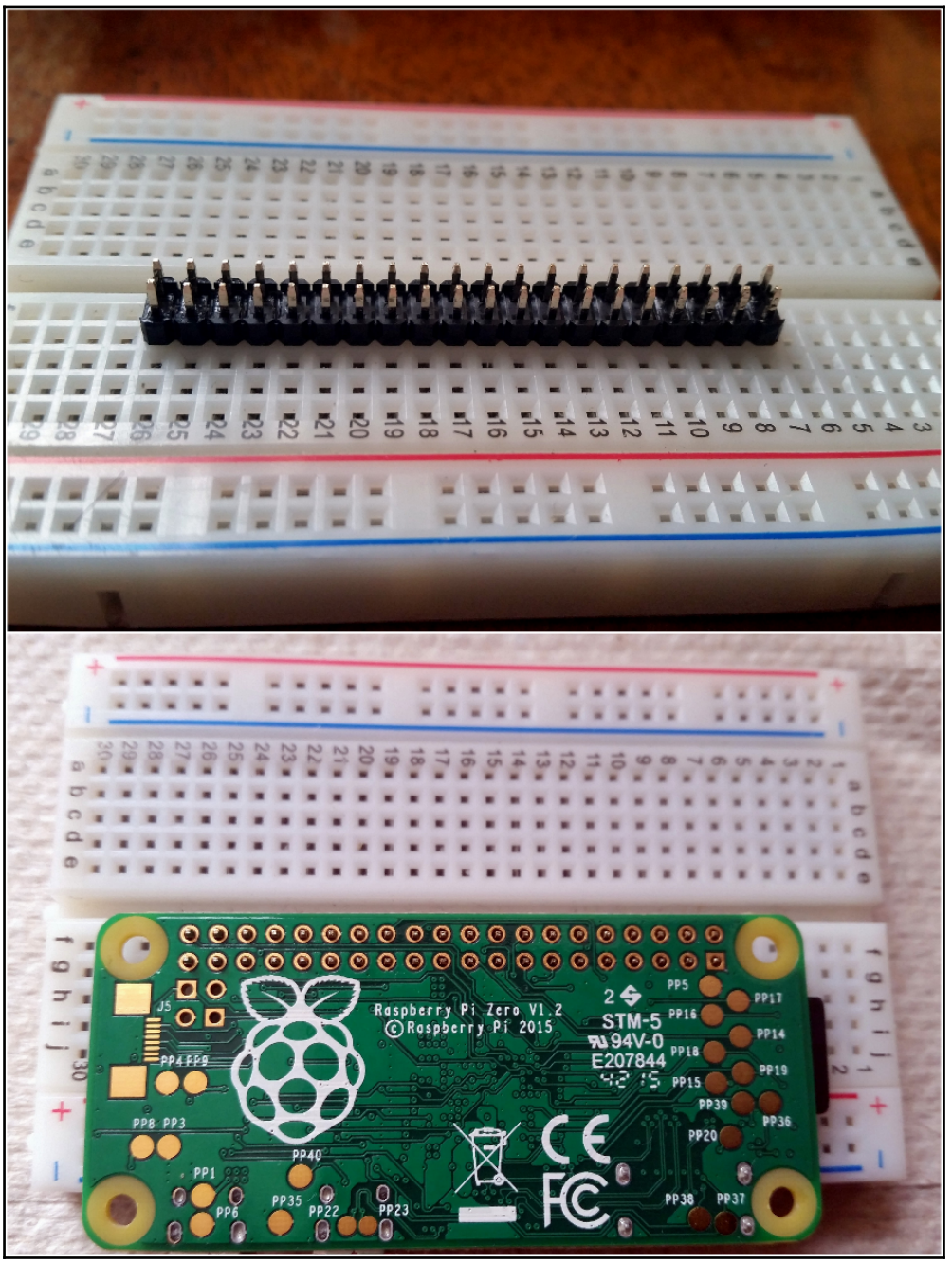
As mentioned before, the Raspberry Pi's GPIO section consists of 40 pins. This is arranged in two rows of 20 pins each. We will need either two sets of 20 pin male headers or a 20 pin double row male header. These are available from vendors like Digikey, Mouser, and so on. The headers for the Raspberry Pi are also sold as a kit by vendors like the *PiHut* (https://thepihut.com/collections/raspberry-pi-zero/products/raspberry-pi-zero-essential-

kit)



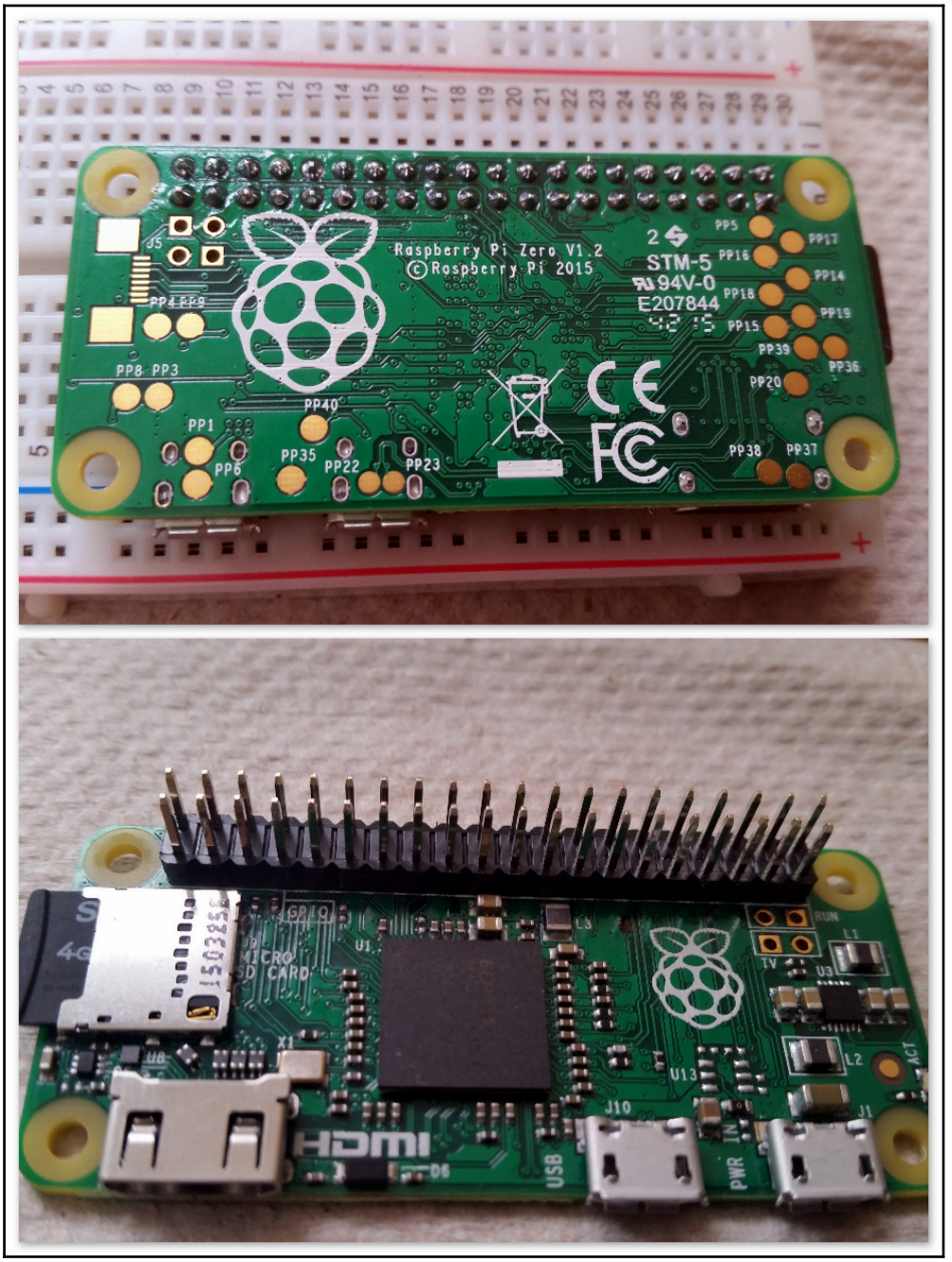
2×20 headers for the Raspberry Pi Zero

- In order to solder the headers onto the Raspberry Pi Zero, arrange the headers on a breadboard as shown in the figure below.


Raspberry Pi Zero V1.2
©Raspberry Pi 2015
STM-5
94V-0
E207844
J5
PP4 PP9
PP8 PP3
PP40
PP1
PP6
PP35
PP22
PP23
PP5
PP16
PP18
PP15
PP39
PP20
PP38
PP37
PP17
PP14
PP19
PP36

Arranging the headers for soldering on to the Raspberry Pi

- Arrange the Raspberry Pi on top of the headers upside down.
- Gently hold the Raspberry Pi (to make sure that the headers are positioned correctly while soldering) and solder the headers on to the Raspberry Pi.
- Inspect the board to ensure that the headers are soldered properly and carefully remove the Raspberry Pi Zero off the breadboard.


Raspberry Pi Zero V1.2
©Raspberry Pi 2015
STM-5
94V-0
E207844
MICRO SD CARD
HDMI
USB
PWR

Headers soldered on to the Raspberry Pi

We are all set to make use of the GPIO pins in this book! Let's move on to the next section.

Note: Soldering the headers onto the Raspberry Pi using a breadboard might damage the breadboard if the right temperature setting isn't used. The metal contacts of the breadboard might permanently expand resulting in permanent damage. Training in basic soldering techniques is crucial and there are plenty of tutorials on this topic.

Enclosure for the Raspberry Pi Zero

Setting up a Raspberry Pi zero inside an enclosure is completely optional but definitely useful while working on your projects. There are plenty of enclosures sold by vendors. Alternatively, you may download an enclosure design from Thingiverse and 3D print them. We found this enclosure to suit our needs: http://www.thingiverse.com/thing:1203246 as it provides access to the GPIO headers. 3D printing services like 3DHubs (3dhubs.com) would print the enclosure for a charge of $9 via a local printer. Alternately, you can also use pre-designed project enclosures, or design one that can be constructed using plexiglass or similar materials.


GPIO
RUN
HDMI

Raspberry Pi Zero in an enclosure

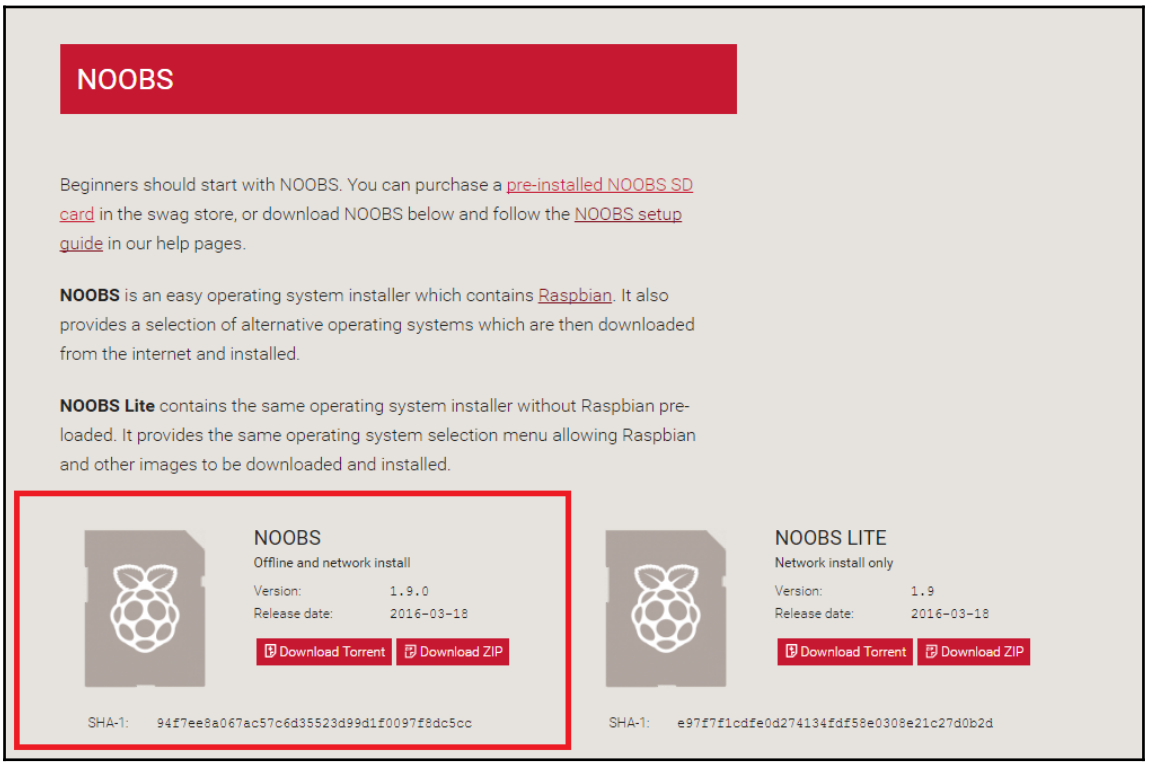
OS setup for the Raspberry Pi

Let's go ahead and prepare a micro SD card to set up the Raspberry Pi Zero. In this book, we will be working with the Raspbian Operating System (OS). The Raspbian OS has a wide user base and the operating system is officially supported by the Raspberry Pi foundation. Hence, it is easier to find support on forums while working on projects as more people are familiar with the operating system.

microSD card prep

If you had purchased a microSD card that comes pre-flashed with the Raspbian **NOOBS(New Out of the Box Software)** image, you may skip the microSD card prep.

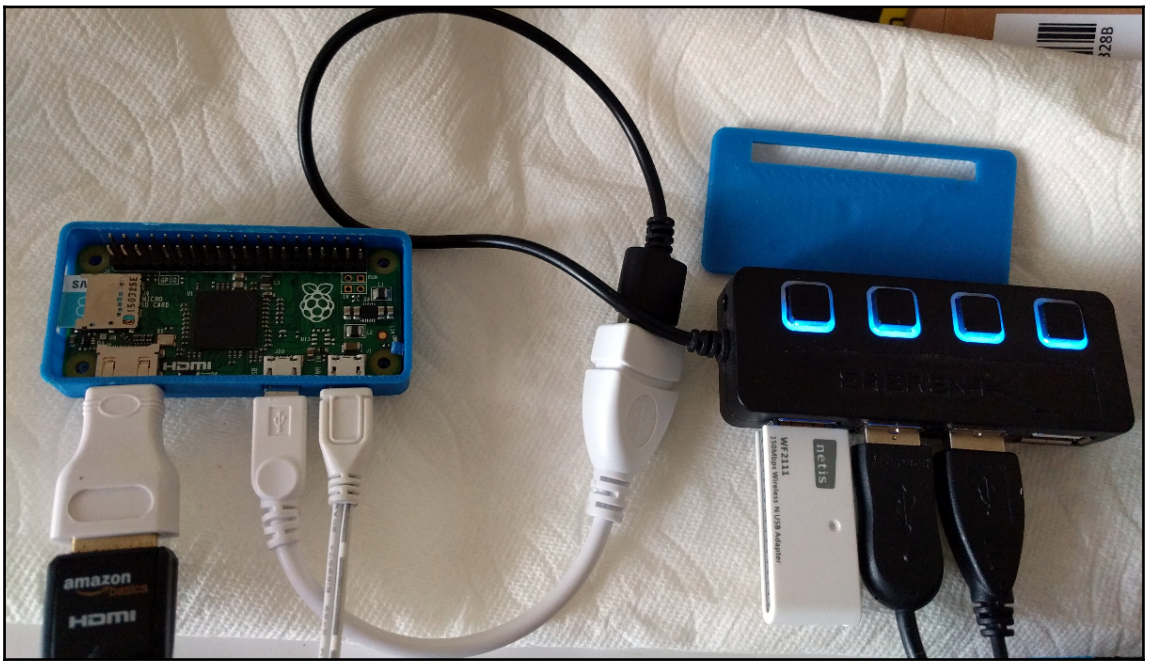
1. The first step is downloading the Raspbian NOOBS image. The image can be downloaded from here: https://www.raspberrypi.org/downloads/noobs/



Downloads the Raspberry Pi NOOBS image

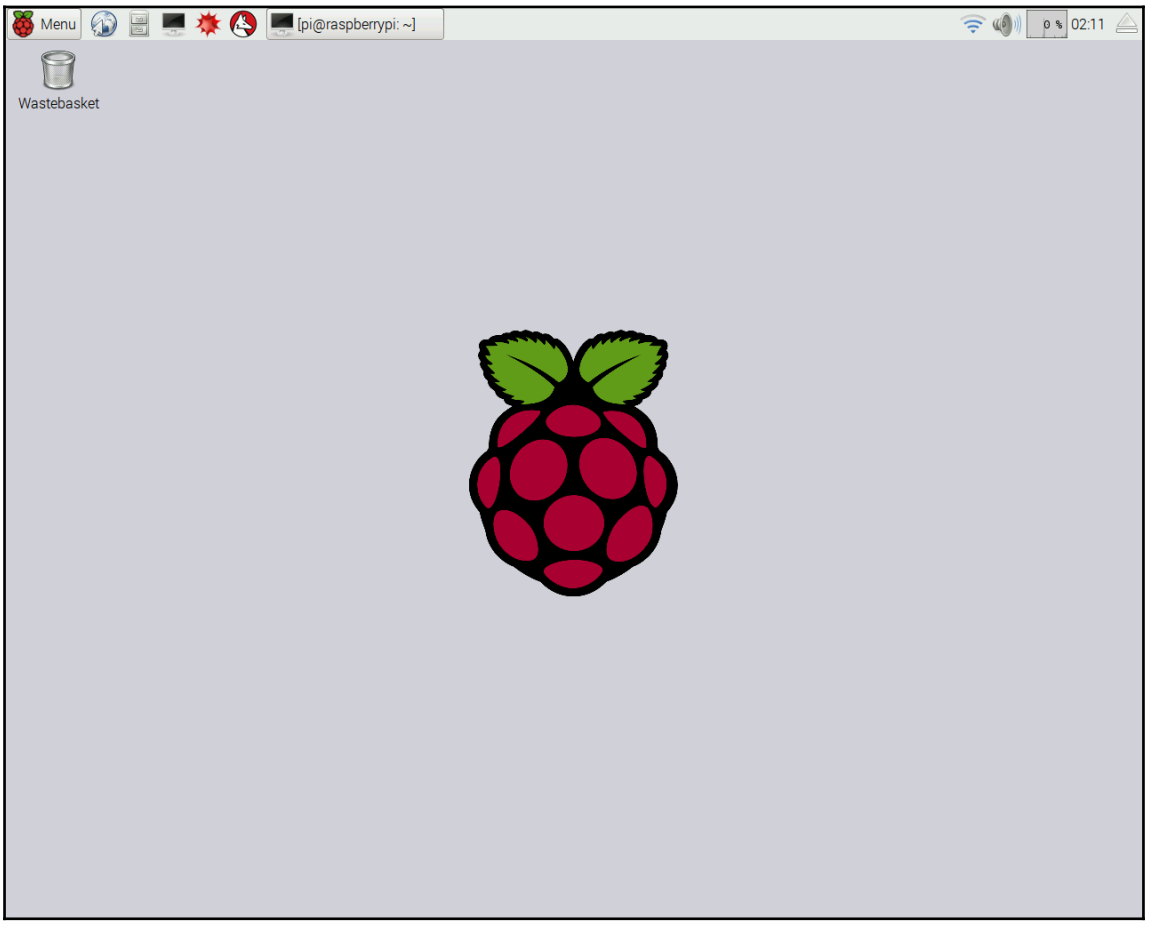
1. Format your SD card using the SD Formatter tool (available from https://www.sdcard.org/downloads/formatter_4/index.html)
2. Extract the downloaded zip file and copy the contents of the file to the formatted microSD card.
3. Setup the Raspberry Pi (not necessarily in the same order):

- Interface the HDMI cable from the monitor via the mini HDMI interface
- USB hub via the USB OTG interface of the Raspberry Pi Zero
- Micro USB cable to power the Raspberry Pi Zero.
- Plug in a WiFi adapter, keyboard and a mouse to the Raspberry Pi Zero.



Raspberry Pi Zero with the keyboard, mouse and the WiFi adapter

1. Power up the Raspberry Pi and it should automatically flash the OS onto the SD card and launch the desktop at startup.
2. When the installation is complete, connect the Raspberry Pi Zero to the wireless network (using the wireless tab on the top right).



Raspberry Pi desktop upon launch

The Raspberry Pi foundation hosts a video on its website that provides a visual aid to setting up the Raspberry Pi. This video is available at: https://vimeo.com/90518800

Let's learn Python!

Python is a high level programming language invented by Guido Van Rossum. It is advantageous to learn Python using the Raspberry Pi for the following reasons:

- It has a very simple syntax and hence very easy to understand
- It offers the flexibility of implementing ideas as a sequence of scripts. This is helpful to hobbyists to implement their ideas.
- There are python libraries to for the Raspberry Pi's GPIO. This enables easy interfacing of sensors/appliances to the Raspberry Pi.
- Python is used in a wide range of applications by tech giants such as Google. These applications range from simple robots to personal AI assistance and control modules in space.
- The Raspberry Pi has a growing fan base. This combined with the vast user group of Python means that there is no scarcity for learning resources or support for Projects

In this book, we will learn Python version 3.x. We will learn each aspect of Python programming using a demonstrative example. Find out the awesomeness of Python by learning to do things by yourself!

Hello World Example

Since we are done setting up the Raspberry Pi, let's get things rolling by writing our first piece of code in Python. While learning a new programming language, it is customary to get started by printing Hello World on the computer screen. Let's print the following message: *I am excited to learn Python programming with the Raspberry Pi Zero* using Python.

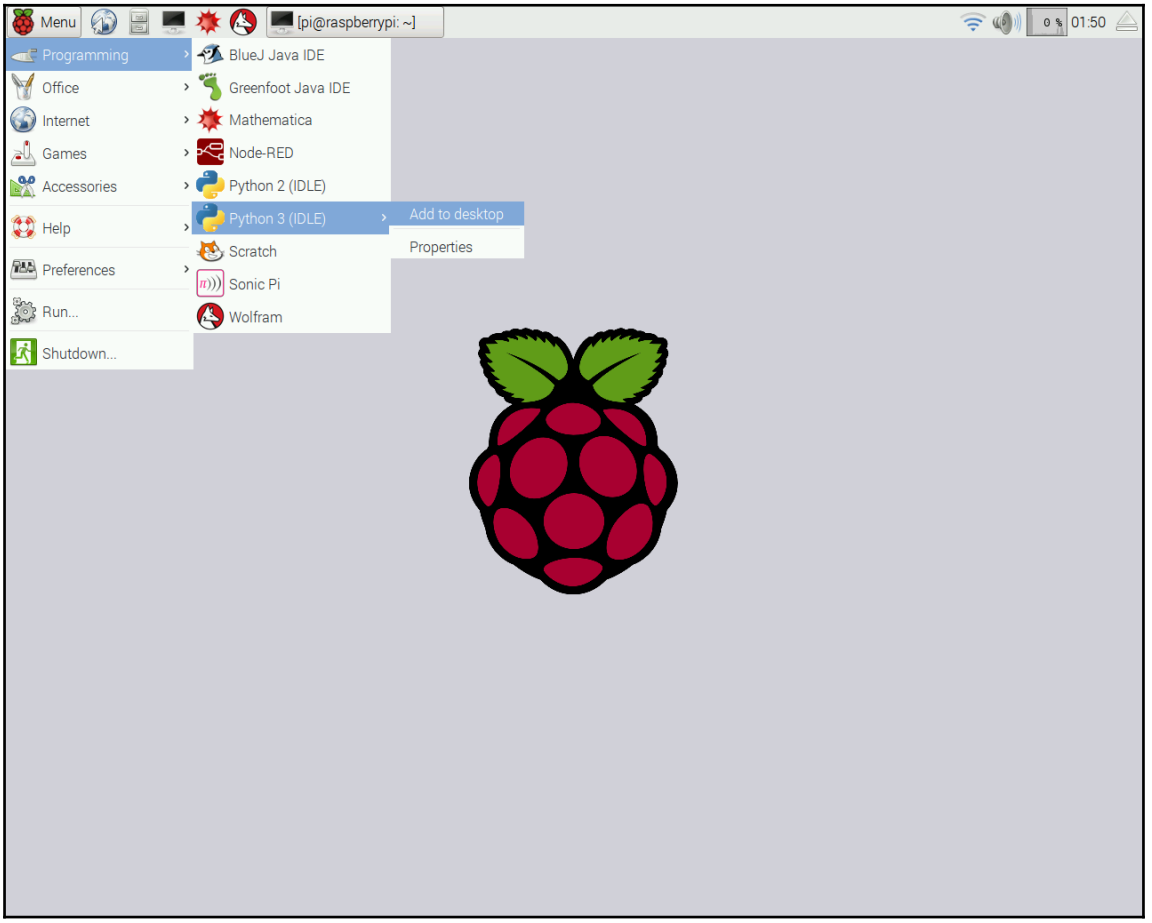
In this book, we will learn Python using the **IDLE(Integrated Development and Learning Environment)** tool. We chose*IDLE* for the following reasons:

- The tool is installed and shipped as a package in the Raspbian OS image. No installation required.
- It is equipped with an interactive tool that enables performing checks on a piece of code or a specific feature of the Python language
- It comes with a text editor that enables writing code according to the conventions of the Python programming language. The text editor provides a colour code for different elements of a Python script. This helps in writing a Python script with relative ease.
- The tool enables a step-by-step execution of any code sample and identify problems in it.

Setting up your Raspberry Pi Zero for Python programming

Before we get started, let's go ahead and setup the Raspberry Pi Zero to suit our needs:

1. Let's add a shortcut to *IDLE3* (for developing in Python 3.x) on the Raspberry Pi's desktop. Under**Programming** sub-menu, (located at the left top corner of your Raspberry Pi Zero's desktop), right click on**IDLE-3** and click on**Add to Desktop**. This adds a shortcut to the IDLE tool on your desktop, thus making it easily accessible.



Add shortcut to IDLE3 to the Raspberry Pi's desktop.

1. In order to save all the code samples, let's go ahead and create a folder called *code_samples* on the Raspberry Pi's desktop. Right click on your desktop and create a new folder.

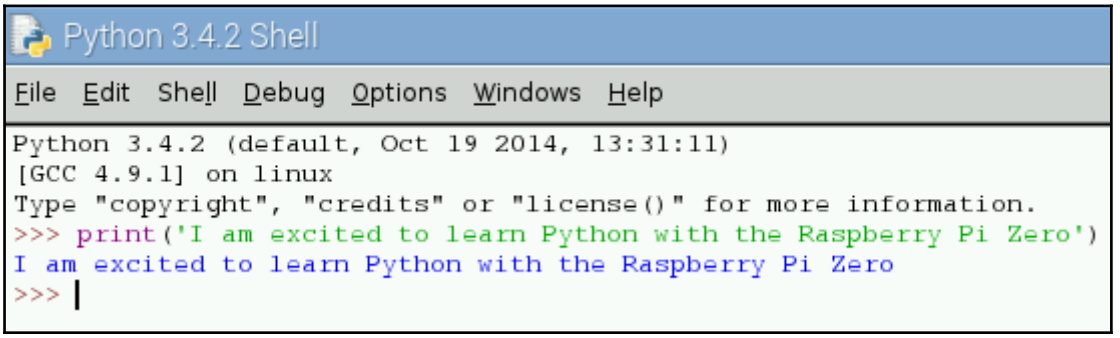
IDLE's interactive tool

Let's write our first example using IDLE's interactive tool:

1. Launch the IDLE3 (meant for Python 3.x) tool from the Raspberry Pi Zero's desktop by double clicking on it.
2. From the IDLE's interactive command line tool, type the following line:

```
print("I am excited to learn Python with the Raspberry Pi Zero")
```

1. This should print the following to the interactive command line tool's screen:

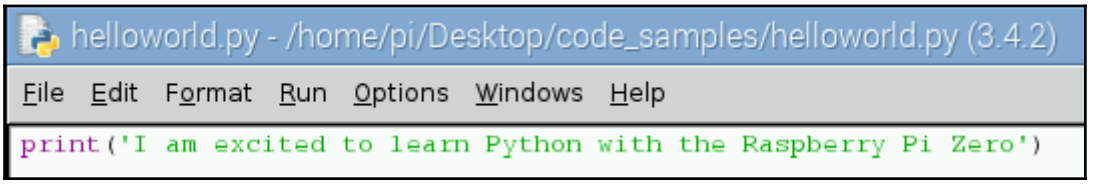


We did it! We wrote a single line that prints out a line of text to the Raspberry Pi's screen.

Text editor approach

The command line tool is useful to test coding logic but it is neither practical nor elegant to write code using the interactive tool. It is easier to write a bunch of code at a time and test it. Let's repeat the same example using IDLE's text editor.

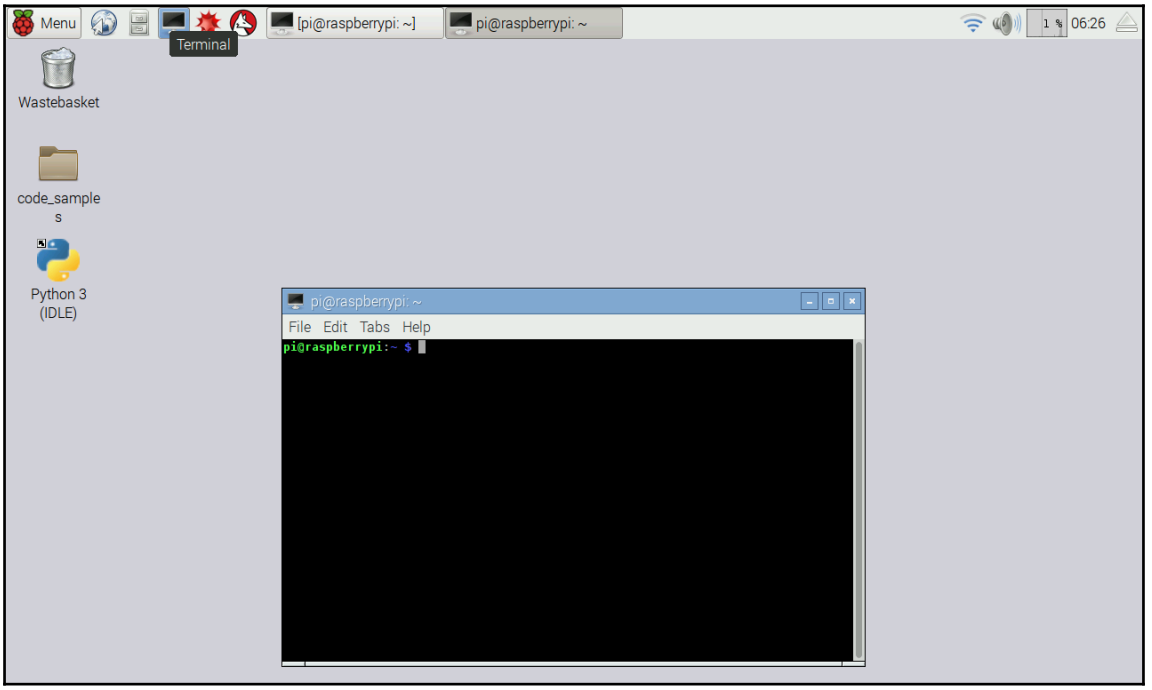
1. Launch IDLE's text editor (In IDLE, **File** | **New File**), enter the hello world line discussed in the previous section and save it as helloworld.py.
2. Now. the code could be executed either pressing the F5 key or clicking on **Run** module from the dropdown menu **Run** and you will get the output as shown in the figure below.



Launching the Python interpreter via the Linux terminal

It is also possible to use the Python interpreter via the Linux terminal. Programmers mostly use this to test their code or refer to the Python documentation tool: *pydoc*. This approach is convenient if the readers plan to use a text editor other than IDLE.

1. Launch the Raspberry Pi's command line terminal from the desktop toolbar.

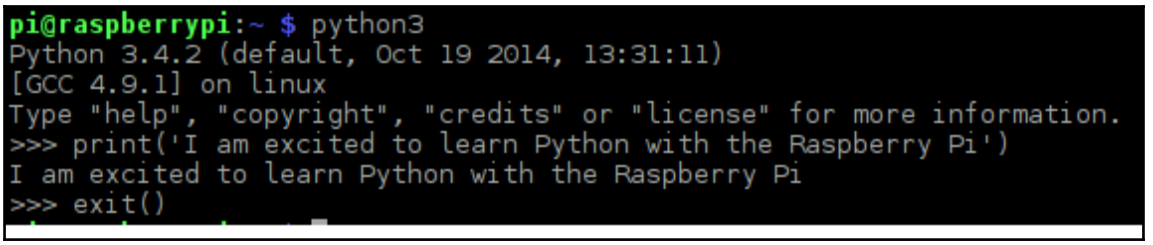


Launching the command line terminal

- Type the command: python3 and press Enter

- This should launch python 3.x on the terminal
- Now, trying running the same piece of code discussed in the previous section as shown in the following screenshot:

```
print("I am excited to learn Python with the Raspberry Pi Zero")
```



- The result should be similar to that of the previous two sections.
- The Python interpreter in the Linux terminal may be closed by typing exit() and pressing the return key.

Executing python scripts using the linux terminal

It is possible to execute code written using any text editor via the Linux terminal. For example: Let's say the file helloworld.py is saved in a folder called code_samples on the Raspberry Pi's desktop. This file may be executed as follows from the Linux terminal:

1. On the Linux terminal, switch to directory where the python script is located:

cd /home/pi/Desktop/code_samples

1. Execute the python script as follows:

python3 helloworld.py

1. Alternatively, the python script could be executed using its absolute location path:

python3 /home/pi/Desktop/code_samples/hello_world.py

We did it! We just wrote our first piece of code and discussed different ways to execute the code.

The print() function

In our first helloworld example, we discussed printing something to the screen. We used the print() function to obtain our result. In Python, a **function** is a pre-defined task that is executed when it is used in a program with the recommended syntax.

For example: The function print() prints any combination of alphanumeric characters mentioned between the quotes. It is also possible to write custom function to execute a repetitive task required by the user. In this chapter, the function print() executed the string **I am excited to learn Python programming with the Raspberry Pi Zero** (We will discuss strings in the later section of this book).

Similarly, the function exit() executes the pre-defined task of exiting the Python interpreter at the user's call.

help() function

While getting started, it is going to be difficult to remember the syntax of every function in Python. It is possible to refer to a function's documentation and syntax using the help function in Python. For example: In order to find the use of the print function in Python, we can call help on the command line terminal or the interactive shell as follows:

```
help(print)
```

This would return a detailed description of the function and its syntax.

Help on built-in function print in module built-ins:

```
print(...)
   print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
```

Prints the values to a stream, or to sys.stdout by default.

Optional keyword arguments:

- file: a file-like object (stream); defaults to the current sys.stdout.
- sep: string inserted between values, default a space.
- end: string appended after the last value, default a newline.
- flush: whether to forcibly flush the stream

Summary

That's it! You are now ready and on your way to learn Python with the Raspberry Pi. In the next chapter, we will dig deeper and learn more about the GPIO pins while executing a simple project that makes LEDs blink.

2
Arithmetic Operations, Loops and Blinky Lights

In the previous chapter, we discussed printing a line of text on the screen. In this chapter, we will review arithmetic operations and variables in Python. We will also discuss strings and accepting user's inputs in Python. We will learn about the Raspberry Pi's GPIO and its features and write code in Python that blinks an LED using the Raspberry Pi's GPIO. We will also discuss a practical application of controlling the Raspberry Pi's GPIO.

...

Hardware required for this chapter

In this chapter, we will be discussing examples where we will be controlling the Raspberry Pi's GPIO. We will need a breadboard, jumper wires, LEDs and some 330 ohm resistors to discuss these examples.

We will also need some optional hardware that we will discuss in the last section of this chapter.

Arithmetic operations

Python enables performing all the standard arithmetic operations. Let's launch the python interpreter and learn more:

- **Addition:**

```
>>> 123+456
579
```

- **Subtraction:**

```
>>> 456-123
333
>>> 123-456
-333
```

- **Multiplication:**

```
>>> 123*456
56088
```

- **Division:**

```
>>> 456/22
>>> 456/2.0
228.0
>>> int(456/228)
2
```

- **Modulus operator:** In Python, the modulus operator (%) returns the reminder of a division operation:

```
>>> 4%2

>>> 3%2
1
```

- There is one other arithmetic operator in Python called the **floor operator** (//). This operator returns the floor of the quotient:

```
>>> 9//7
1
>>> 7//3
2
>>> 79//25
3
```

Bitwise operators in Python

In Python, it is possible to perform bit levels operations on numbers. This is especially helpful while parsing information from certain sensors (discussed in detail in upcoming chapters). Consider the numbers 3 and 2 whose binary equivalents are 011 and 010 respectively. Let's take a look at different operators that perform the operation on every bit of the number.

- **AND operator:**

```
>>> 3&2
2
```

- **OR operator:**

```
>>> 3|2
3
```

- **NOT operator:**

```
>>> ~1
-2
```

- **XOR operator:**

```
>>> 3^2
1
```

- **Left shift operator:** The left shift operator enables shifting the bits of a given value to the left by the desired number of places. For example: Bit shifting the number 3 to the left gives us the number 6. The binary representation of the number 3 is 011. Left shifting the bits by 1 position will give us 110 i.e. the number 6.

```
>>> 3<<1
6
```

- **Right shift operator:** The right shift operator enables shifting the bits of a given value to the right by the desired number of places. For example: Bit shifting the number 6 to the right by one position gives us the number 3.

```
>>> 6>>1
3
```

Logical operators

Logical operators are used for used to check different conditions and execute the code accordingly. For example, detecting a button interfaced to the Raspberry Pi's GPIO and executing a specific task. Let's discuss the basic logical operators:

- **EQUAL operator '==' :** The **EQUAL** operator is used to compare if two values are equal.

```
>>> 3==3
True
>>> 3==2
False
NOT EQUAL '!=':The NOT EQUAL operator compares two values and returns True if they are not equal.
>>> 3!=2
True
>>> 2!=2
False
```

- **GREATER THAN '>':** This operator returns *True* if one value is greater than the other value:

```
>>> 3>2
True
>>> 2>3
False
```

- **LESS THAN '<':** This operator compares two values and returns *True* if one value is smaller than the other.

```
>>> 2<3
True
>>> 3<2
False
```

- **GREATER THAN OR EQUAL TO '>=':** This operator compares two values and returns *True* if one value is greater/bigger than or equal to the other value.

```
>>> 4>=3
True
>>> 3>=3
True
>>> 2>=3
False
```

- **LESS THAN OR EQUAL TO '<=':** This operator compares two values and

returns True if one value is smaller than or equal to the other value.

```
>>> 2<=2
True
>>> 2<=3
True
>>> 3<=2
False
```

Data types and variables in Python

In Python, **variables** are used to store a result or a value in the computer's memory during the execution of a program. Variables enable easy access to a specific location on the computer's memory and enables writing user readable code.

For example, let's consider a scenario where a person wants a new ID card from an office or a university. The person would be asked to fill out an application form with relevant information including their name, department, emergency contact information, and so on. The form would have the requisite fields. This would enable the office manager to refer to the form while creating a new ID card.

Similarly, variables simplify code development by providing means to store information in the computer's memory. It would be very difficult to write code if one had to write code keeping the storage memory map in mind. Imagine writing code to store data at a specific address like 0x3745092.

There are different kinds of data types in Python. Let's review the different data types:

- In general names, street addresses, and so on are a combination of alphabets and numbers. In Python, they are stored as **strings**. Strings in Python are represented and stored in variables as follows:

```
>>> name = 'John Smith'
>>> address = '123 Main Street'
```

- **Numbers** in Python could be stored as follows:

```
>>> age = 29
>>> employee_id = 123456
>>> height = 179.5
>>> zip_code = 94560
```

- Python also enables storing **boolean** variables. For example: A person's organ

donor status can be either as True or False.

```
>>> organ_donor = True
```

- It is possible to **assign** values to multiple variables at the same time.

```
>>> a = c= 1
>>> b = a
```

- A variable may be **deleted** as follows:

```
>>> del(a)
```

There are other data types in Python including lists, tuples and dictionaries. We will discuss this in detail in the next chapter.

Reading inputs from the user

In the previous chapter, we printed something on the screen. Now that we have discussed the data types in Python, let's take some user input and print it out.

In Python, user input to a Python program can be provided using the input() program. The input function is used to take user input using a prompt and store the input in a variable:

```
>>> name = input("What is your name? ")
What is your name? Sai
>>> print(name)
Sai
```

Let's build a console/command-line application that takes inputs from the user and print it to the screen. Let's create a new file called input_test.py, (available along with this chapter's downloads) take some user inputs and print them to the screen.

```
name = input("What is your name? ")
address = input("What is your address? ")
age = input("How old are you? ")

print("My name is " + name)
print("I am " + age + " years old")
print("My address is " + address)
```

Execute the program and see what happens:

```
What is your name? Sai
What is your address? 123 Main Street, Newark, CA
```

```
How old are you? 29
My name is Sai
I am 29 years old
My address is 123 Main Street, Newark, CA
```

Concatenating strings

In the above example, we printed the user inputs in combination with another string. For example: We took the user input 'name' and printed the sentence as "My name is Sai". The process of appending one string to another is called concatenation.

In Python, strings can be concatenated by adding a '+' between two strings:

```
name = input("What is your name? ")
print("My name is " + name)
```

It is possible to concatenate two strings but it is not possible to concatenate an integer. Let's consider the following example:

```
id = 5
print("My id is " + id)
```

It would throw an error implying that integers and strings cannot be combined:

```
Traceback (most recent call last):
  File "<pyshell#35>", line 1, in <module>
    print(name + age)
TypeError: Can't convert 'int' object to str implicitly
```

It is possible to convert an integer to string and concatenate it to another string:

```
print("My id is " + str(id))
```

Loops in Python

Sometimes, a specific task has to be repeated several times. In such cases, we could use loops. In Python, there are two types of loops namely for loop and while loop. Let's review them with specific examples.

for loop

In Python, a for loop is used to execute a task for *n* times. For example: The variable i could

be used to count the loop execution as follows:

```
for i in range(0, 10):
  print("Loop execution no: ", i)
```

In the above example, the print statement is executed 10 times:

Loop execution no: 0

```
Loop execution no:  1
Loop execution no:  2
Loop execution no:  3
Loop execution no:  4
Loop execution no:  5
Loop execution no:  6
Loop execution no:  7
Loop execution no:  8
Loop execution no:  9
```

In order to execute the print task 10 times, the range function was used. The range generates a sequence of numbers when a start value and a stop value is specified. In this case, a sequence of numbers from 0 to 9 is generated in increments of 1. The for loop iterates through the code for each step. The range function enables incrementing in steps of 2:

```
Loop execution no:  0
Loop execution no:  2
Loop execution no:  4
Loop execution no:  6
Loop execution no:  8
Loop execution no:  10
Loop execution no:  12
Loop execution no:  14
Loop execution no:  16
Loop execution no:  18
```

The range function also helps count down from a given number. Let's say we would like to count down from 10 to 1:

```
for i in range(10, 1, -1):
  print("Count down no: ", i)
```

The output would be something like:

```
Count down no:  10
Count down no:  9
Count down no:  8
Count down no:  7
Count down no:  6
```

```
Count down no:  5
Count down no:  4
Count down no:  3
Count down no:  2
```

The general syntax of the range function is range(start, stop, step_count). It generates a sequence of numbers from start to n-1 where n is the stop value.

Indentation

Note the **indentation** in the for loop block:

```
for i in range(10, 1, -1):
  print("Count down no: ", i)
```

Python executes the block of code under the for loop statement. It is one of the features of the Python programming language. It executes any piece of code under the for loop as long as it has same level of indentation.

```
for i in range(0,10):
  #start of block
  print("Hello")
  #end of block
```

The indentation has two uses:

- It makes the code readable.
- It helps the identify the block of code to be executed in a loop

It is important to pay attention to indentation in Python as it directly affects how a piece of code is executed.

Nested loops

In Python, it is possible to implement *a loop within a loop.* For example: Let's say we have to print x and y coordinates of a map. We can use nested loops to implement this:

```
for x in range(0,3):
  for y in range(0,3):
      print(x,y)
```

The expected output is:

```
0 0
0 1
```

```
0 2
1 0
1 1
1 2
2 0
2 1
2 2
```

Be careful about code indentation in nested loops as it may throw errors. For example: Consider the following example:

```
for x in range(0,10):
  for y in range(0,10):
  print(x,y)
```

The python interpreter would throw the following error:

SyntaxError: expected an indented block

Hence it is important to pay attention to indentation in Python (especially nested loops) to successfully execute the code. IDLE's text editor automatically indents code as you write them. This should aid with understanding indentation in Python.

while loop

while loops are used when a specific task is supposed to be executed until a specific condition is met. while loops are commonly used to execute code in an infinite loop. Let's look at a specific example where we would like to print the value of i from 0 to 9:

```
i=0
while i<10:
  print("The value of i is ",i)
  i+=1
```

This example would execute the code until the value of i is less than 10. It is also possible to execute something in an infinite loop:

```
i=0
while True:
  print("The value of i is ",i)
  i+=1
```

The execution of this infinite loop can be stopped by pressing Ctrl+C on your keyboard.

It is also possible to have nested while loops:

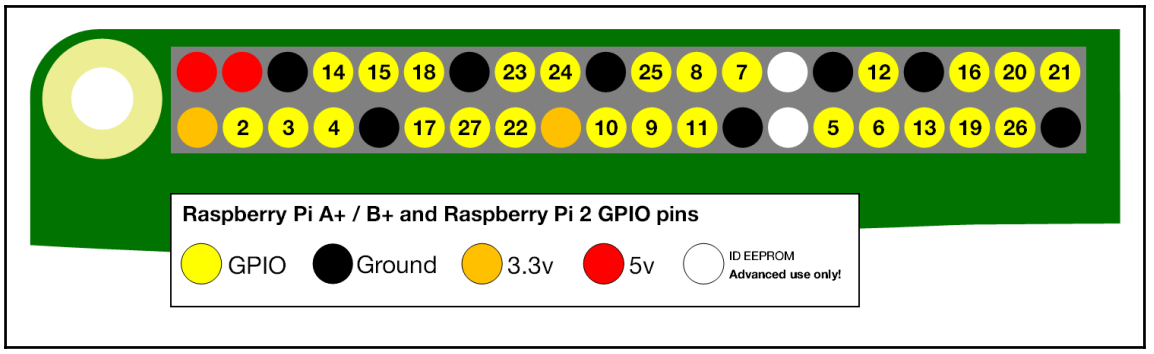
```
i=0
j=0
while i<10:
  while j<10:
    print("The value of i,j is ",i,",",j)
    i+=1
    j+=1
```

Similar to for loops, while loops also rely on the indented code block to execute a piece of code.

Python enables printing a combination of strings and integers as long as they are presented as arguments to the print function separated by commas. In the above example, "The value of i,j is ", i are arguments to the print function. We will learn more about functions and arguments in the next chapter. This feature enables formatting the output string to suit our needs.

Raspberry Pi's GPIO

The term **GPIO** refers to **General Pupose Input/Output**. The Raspberry Pi Zero comes with a 40 pin GPIO header. Out of these 40 pins, we should be able to use 26 pins. This means that we should be able to use the 26 pins to wither read inputs (from sensors) or control outputs. The other pins are power supply pins (5V, 3.3V and Ground pins).



Raspberry Pi Zero GPIO mapping (Source: https://www.raspberrypi.org/documentation/usage/gpio-plus-and-raspi2/README.md)

We should be able to use the 26 pins of the Raspberry Pi's GPIO to interface appliances and control them. But, there are certain pins which have an alternative function which will be

discussed in the later chapter.

The above image shows the mapping of the Raspberry Pi's GPIO pins. The numbers in the circle correspond to the pin numbers on the Raspberry Pi's processor. For example GPIO pin 2 (second pin from the left on the bottom row) corresponds to the GPIO pin 2 on the Raspberry Pi's processor.

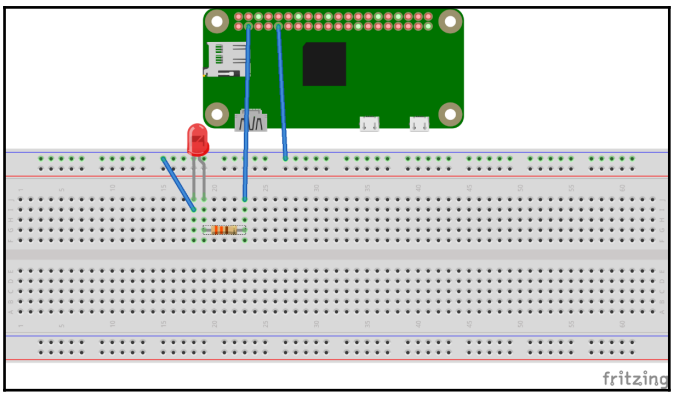
In the beginning, it might be confusing to try and understand the pin mapping. Keep a GPIO pin handout (available for download along with this chapter) for your reference. It takes some time to get used to the GPIO pin mapping of the Raspberry Pi Zero.

The Raspberry Pi Zero's GPIO pins are 3.3V tolerant that is, if a voltage greater than 3.3V is applied to the pin, it may permanently damage the pin. When set to *HIGH*, the pins are set to 3.3V and 0V when the pins are set to *LOW*.

Blinky lights

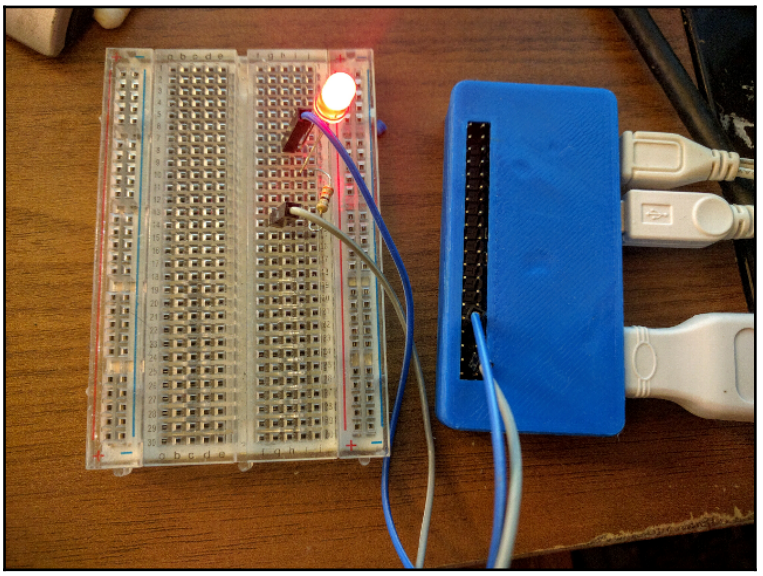
Let's discuss an example where we make use of the Raspberry Pi Zero's GPIO. We will interface an LED to the Raspberry Pi Zero and make it blink *on* and *off* with a one second interval.

Let's wire up the Raspberry Pi zero to get started.



Blinky Schematic generated using Fritzing

In the above schematic, the GPIO pin 2 is connected to the anode of the LED. The cathode of the LED is connected to the ground pin of the Raspberry Pi Zero. A 330 ohm current limiting resistor to limit the flow of the current.



Breadboard connections to the Raspberry Pi Zero

Code

We will make use of the python3-gpiozero library. The Raspbian Jessie OS image comes with the pre-installed library. It is very simple to use and it is the best option to get started as a beginner. It supports a standard set of devices that helps us get started easily.

For example: In order to interface an LED, we need to import the LED*class* from the gpiozero library.

```
from gpiozero import LED
```

We will be turning the LED *on* and *off* at a 1 second interval. In order to do so, we will be 'importing' the time library. In Python, we need to import a library to make use of it. Since we interfaced the LED to the GPIO pin 2, let's make a mention of that in our code.

```
import time

led = LED(2)
```

We just created a variable called led and defined that we will be making use of GPIO pin 2 in the LED class. Let's make use of a while loop to turn the LED on and off with a one second interval.

The gpiozero library's LED class comes with functions called on() and off() to set the GPIO pin 2 to high and low respectively.

```
while True:
    led.on()
    time.sleep(1)
    led.off()
    time.sleep(1)
```

In Python's time library, there is a sleep function that enables introducing a 1 second delay between turning on/off the LED. This is executed in an infinite loop! We just built a practical example using the Raspberry Pi Zero.

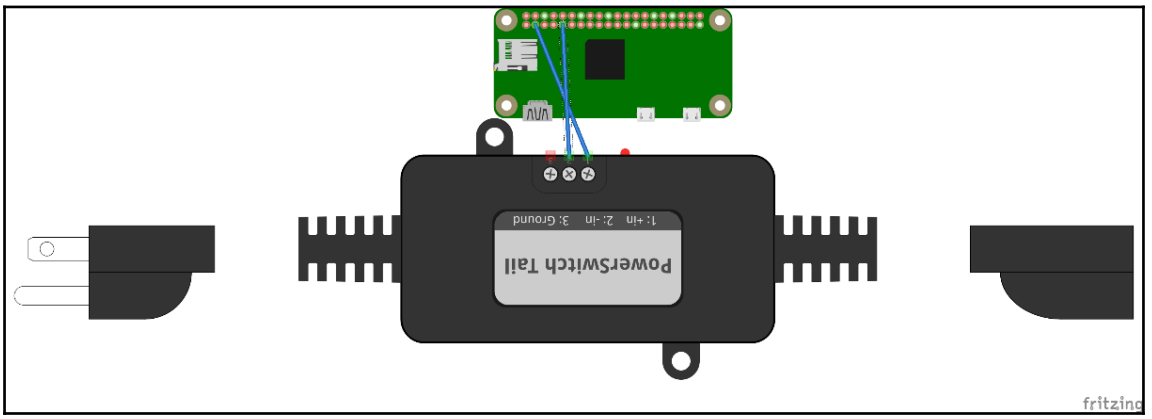
Putting all the code together in a file called blinky.py (available for download along with this book) and run the code from the command line terminal (alternatively, you may use IDLE3)

```
python3 blinky.py
```

Applications of GPIO control

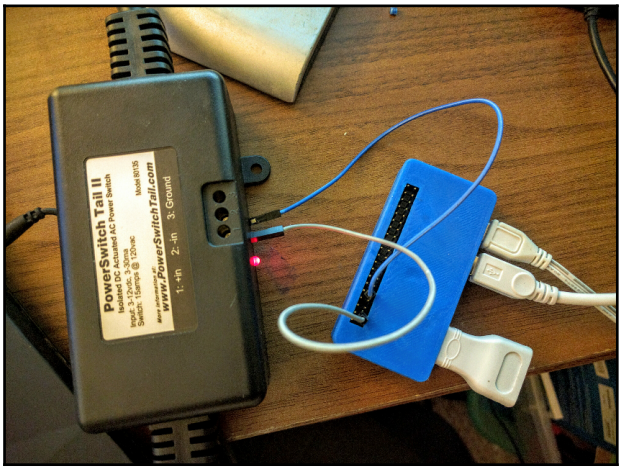
Now that we have implemented our first example, let's discuss some possible applications of being able to control the GPIO. We could use the Raspberry Pi's GPIO to control the lights in our homes. We will make use of the same example to control a table lamp!

There is a product called the Powerswitch Tail 2 (http://www.powerswitchtail.com/Pages/default.aspx) that enable interfacing AC appliances like a table lamp to a Raspberry Pi. The Powerswitch Tail comes with control control pins (that can take a 3.3V *HIGH* signal) that could be used to turn on/off a lamp. The switch comes with the requisite circuitry/protection to interface it directly to a Raspberry Pi Zero.



Pi Zero interfaced to the Powerswitch Tail II

Let's take the same example from the previous section and connect the GPIO pin 2 to the *in+* pin of the Powerswitch tail. Let's connect the ground pin of the Raspberry Pi Zero's GPIO header to the Powerswitch tail's *in-* pain. The Powerswitch tail should be connected to the AC mains. The lamp should be connected the AC output of the switch. If we use the same piece of code and connect a lamp to the Powerswitch tail, we should be able to turn on/off with a 1 second interval.



Powerswitch Tail II connected to a Raspberry Pi Zero

Note: This appliance control using the LED blinking code is just an example. It is not a recommended to turn on/off a table lamp at such short intervals. In future chapters, we will make use of the Raspberry Pi Zero's GPIO to control appliances from anywhere on the internet.

Summary

In this chapter, we reviewed the basic data types in Python, arithmetic operations and logical operators in Python. We also discussed accepting user inputs and loops. We introduced ourselves to the Raspberry Pi Zero's GPIO and discussed an LED blinking example. We took the same example to control a table lamp!

Have you heard of the chat application called *Slack*? How about controlling a table lamp at home from your laptop at work? If that peeks your interest, work with us towards the next few chapters.

3
Conditional Statements, Functions and Lists

In this chapter, we will build upon what we learned in the previous chapter. We will learn about conditional statements and we will learn how to make use of logical operators to check conditions using conditional statements. Next, we will learn to write simple functions in Python and discuss interfacing inputs to the Raspberry Pi's GPIO header using a tactile switch (momentary push button). We will also discuss motor control (this is a run-up to the final project) using the Raspberry Pi Zero and control the motors using the switch inputs. Let's get to it!

...

Conditional Statements

In Python, conditional statements are used to determine if a specific condition is met by testing whether a condition is True or False. Conditional statements are used to determine how a program is executed. For example: Conditional statements could be used to determine whether it is time to turn on the lights. The syntax is:

```
if condition_is_true:
  do_something()
```

The condition is usually tested using a logical operator and the set of tasks under the indented block is executed. For example: Let's consider the example: check_address.py (available for download with this chapter) where the user input to a program needs to be verified using a yes or no question.

```
check_address = input("Is your address correct(yes/no)? ")
```

```
if check_address == "yes":
  print("Thanks. Your address has been saved")
if check_address == "no":
  del(address)
  print("Your address has been deleted. Try again")
```

In this example, the program expects a yes or no input. If the user provides the input yes, the condition if check_address == "yes" is true, the message **Your address has been saved** is printed to the screen.

Likewise, if the user input is no, the program executes the indented code block under the logical test condition if check_address == "no" and deletes the variable address.

if – else statement:

In the above example, we used an if-statement to test each condition. In Python, there is an alternative option called the *if-else* statement. The if-else statement enables testing an alternative condition if the main condition is not true.

```
check_address = input("Is your address correct(yes/no)? ")
if check_address == "yes":
  print("Thanks. Your address has been saved")
else:
  del(address)
  print("Your address has been deleted. Try again")
```

In this example, if the user input is yes, the indented code block under if is executed. Otherwise, the code block under else is executed.

if-elif-else statement:

In the above example, the program executes any piece of code under the else block for any user input other than yes that is if the user pressed the return key without providing any input or provided random characters instead of no, The *if-elif-else* statement works as follows:

```
check_address = input("Is your address correct(yes/no)? ")
if check_address == "yes":
  print("Thanks. Your address has been saved")
elif check_address == "no":
  del(address)
  print("Your address has been deleted. Try again")
else:
```

```
print("Invalid input. Try again")
```

If the user input is yes, the indented code block under the *if-statement* is executed. If the user input is no, the indented code block under elif (else-if) is executed. If the user input is something else, the program prints the message: **Invalid input. Try again**.

The indented code block is important to identify the code block has to be executed while branching off upon executing a conditional statement. If the condition under if is true, the indented code block under if is executed.

In the three examples that we discussed so far, it could be noted that an if-statement does not need to be complemented by an else statement. The else and elif statements need to have a preceding if statement or the program execution would result in an error.

Breaking out of loops

Conditional statements can be used to break out of a loop execution (for loop and while loop). When a specific condition is met, an if statement can be used to break out of a loop.

```
i = 0
while True:
  print("The value of i is ", i)
  i += 1
  if i > 100:
    break
```

In the above example, the while loop is executed in an infinite loop. The value of i is incremented and printed to the screen. The program breaks out of the while loop when the value of i is greater than 100 and the value of i is printed from 1 to 100.

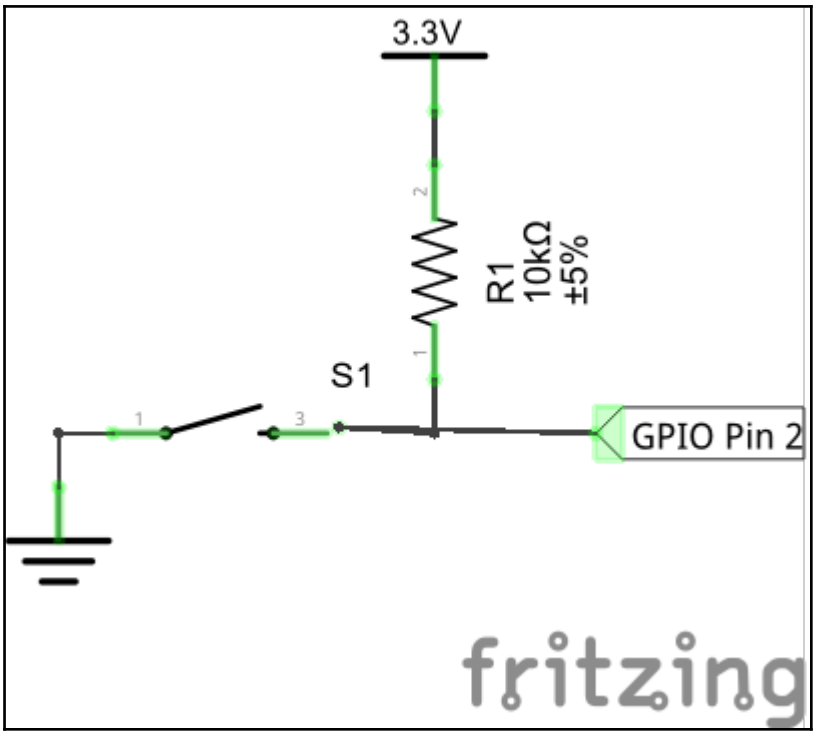
Applications of conditional statements: Reading GPIO Inputs

In the last chapter, we discussed interfacing outputs to the Raspberry Pi's GPIO. Let's discuss an example where a simple push button is pressed. A button press is detected by reading the GPIO pin state.

Let's connect a button to the Raspberry Pi's GPIO. A button, pull-up resistor, and a few jumper wires. The figure below shows an illustration on connecting the push button to the Raspberry Pi Zero. One of the push button's terminals is connected to the ground pin of the Raspberry Pi Zero's GPIO pin.

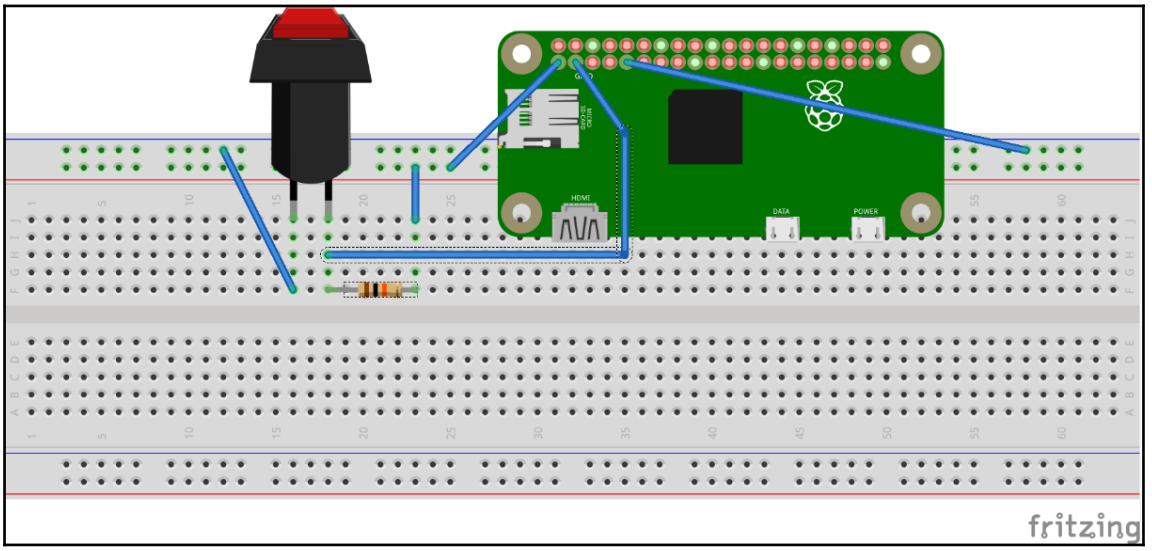
The other terminal of the push button is pulled up to 3.3V using a 10K resistor. The junction of the push button terminal and the 10K resistor is connected to the GPIO pin 2 (Refer to the BCM GPIO pin map shared in the earlier chapter).

The schematic of the button's interface is shown below:



Raspberry Pi GPIO schematic

...



Interfacing the push button to the Raspberry Pi Zero's GPIO – Image generated using Fritzing

Let's review the code required to review the button state. We make use of loops and conditional statements to read the button inputs using the Raspberry Pi Zero.

We will be making use of the gpiozero library introduced in the previous chapter. The code sample for this section is GPIO_button_test.py and available for download along with this chapter.

Let's get started with importing the gpiozero library and instantiate the Button class of the gpiozero library (We will discuss Python's classes, objects and their attributes in a later chapter). The button is interfaced to GPIO pin 2.

```
from gpiozero import Button

#button is interfaced to GPIO 2
button = Button(2)
```

The gpiozero library's documentation is available from here: http://gpiozero.readthedocs.io/en/v1.2.0/api_input.html.According to the documentation, there is a variable called is_pressed in the Button class that could be tested using a conditional statement to determine if the button is pressed:

```
if button.is_pressed:
   print("Button pressed")
```

Whenever the button is pressed, the message **Button pressed** is printed on the screen. Let's stick this code snippet inside an infinite loop:

```
from gpiozero import Button

#button is interfaced to GPIO 2
button = Button(2)

while True:
  if button.is_pressed:
    print("Button pressed")
```

When the check the button state in an infinite while loop, the program constantly checks for a button press and prints the message as long as the button is being pressed. Once the button is released, it goes back to checking whether the button is pressed.

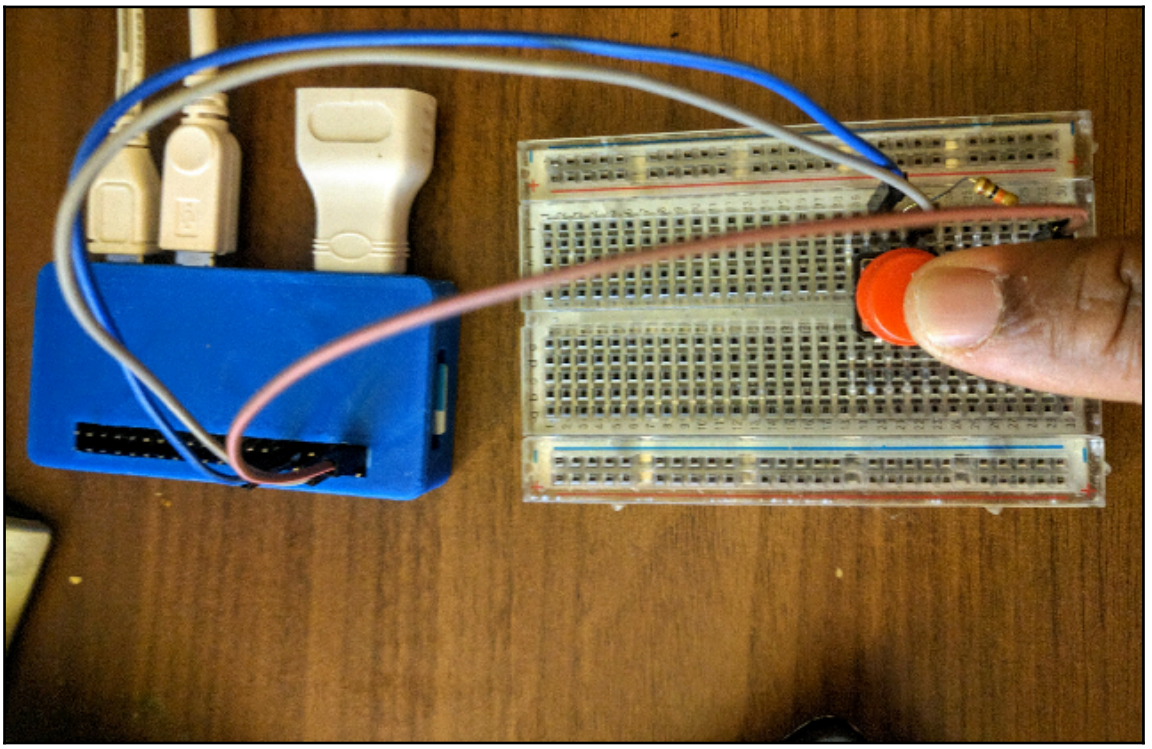
Breaking out a loop by counting button press

Let's review another example where we would like to count the number of button presses and break out of the infinite loop when the button has received a pre-determined number of presses.

```
while True:
  if button.is_pressed:
    button.wait_for_release()
    i += 1
    print("Button pressed")
  if i >= 10:
    break
```

In this example, the program checks for the state of the is_pressed variable. Upon receiving a button press, the program can be paused until the button is released using the method wait_for_release. When the button is released, the variable used to store the number of presses is incremented by 1.

The program breaks out of the infinite loop, when the button has received 10 presses.



A red momentary push button interfaced to Raspberry Pi Zero GPIO pin 2.

Functions in Python

We briefly discussed functions in Python. Functions execute a pre-defined set of task. print is one example of a function in Python. It enables printing something to the screen. Let's discuss writing our own functions in Python.

A function can be declared in Python using the def keyword. A function could be defined as follows:

```
def my_func():
  print("This is a simple function")
```

In this function my_func, the print statement is written under an indented code block. Any block of code that is indented under the function definition is executed when the function is called during the code execution. The function could be executed as follows: my_func().

Passing arguments to a function:

A function is always defined with parentheses. The parentheses are used to pass any requisite arguments to a function. Arguments are parameters required to execute a function. In the above example, there are no arguments passed to the function.

Let's review an example where we pass an argument to a function:

```
def add_function(a, b):
  c = a + b
  print("The sum of a and b is ", c)
```

In this example, a and b are arguments to the function. The function adds a and b and prints the sum to the screen. When the function add_function is called by passing the arguments 3 and 2 as: add_function(3,2) where a=3 and b=2 respectively.

Hence the arguments a and b are required to execute function or calling the function without the arguments would result in an error. Errors related to missing arguments could be avoided by setting default values to the arguments:

```
def add_function(a=0, b=0):
  c = a + b
  print("The sum of a and b is ", c)
```

Now, the arguments to the function as follows: add_function(a=3) or add_function(b=2). When an argument is not furnished while calling a function, it defaults to zero (declared in the function).

Similarly the print function prints any variable passed as an argument. If the print function is called without any arguments, a blank line is printed.

Returning values from a function

Functions can perform a set of defined operations and finally return a value at the end. Let's consider the following example:

```
def square(a):
   return a**2
```

In this example, the function returns a square of the argument. In Python, the return keyword is used to return a value requested upon completion of execution.

Scope of variables in a function

There are two types of variables in a Python program: **local** and **global** variables. Local variables are local to a variable that is a variable declared within a function is accessible within that function only. For example:

```
def add_function():
  a = 3
  b = 2
  c = a + b
  print("The sum of a and b is ", c)
```

In this example, the variables a and b are local to the function add_function. Let's consider an example of a global variable:

```
a = 3
b = 2
def add_function():
  c = a + b
  print("The sum of a and b is ", c)
add_function()
```

In this case, the variables a and b are declared in the main body of the Python script. They are accessible across the entire program. Now, let's consider this example:

```
a = 3
def my_function():
  a = 5
  print("The value of a is ", a)
my_function()
print("The value of a is ", a)
```

In this case, when my_function is called, the value of a is 5 and the value of a is 3 in the print statement of the main body of the script. In Python, it is not possible to explicitly modify the value of global variables inside functions. In order to modify the value of a global variable, we need to make use of the global keyword.

```
a = 3
def my_function():
  global a
  a = 5
  print("The value of a is ", a)
my_function()
print("The value of a is ", a)
```

In general, it is not recommended to modify variables inside functions. The best practice would be passing variables as arguments and returning the modified value. For example:

```
a = 3
def my_function(a):
  a = 5
  print("The value of a is ", a)
  return a
a = my_function(a)
print("The value of a is ", a)
```

...

GPIO 'callback' functions

Let's review some uses of functions with the GPIO example. Functions can be used for handling specific events related to the GPIO pins of the Raspberry Pi. For example. the gpiozero library provides the capability of calling a function either when a button is pressed or released.

```
from gpiozero import Button

def button_pressed():
  print("button pressed")

def button_released():
  print("button released")

#button is interfaced to GPIO 2
button = Button(2)
button.when_pressed = button_pressed
button.when_released = button_released

while True:
  pass
```

In this example, we make use of the attributes when_pressed and when_released of the library's GPIO class. When the button is pressed, the function button_pressed is executed. Likewise, when the button is released, the function button_released is executed. This capability of being able to execute different functions for different events is useful in applications like Home automation. For example: It could be used to turn on lights when it is dark and vice versa.

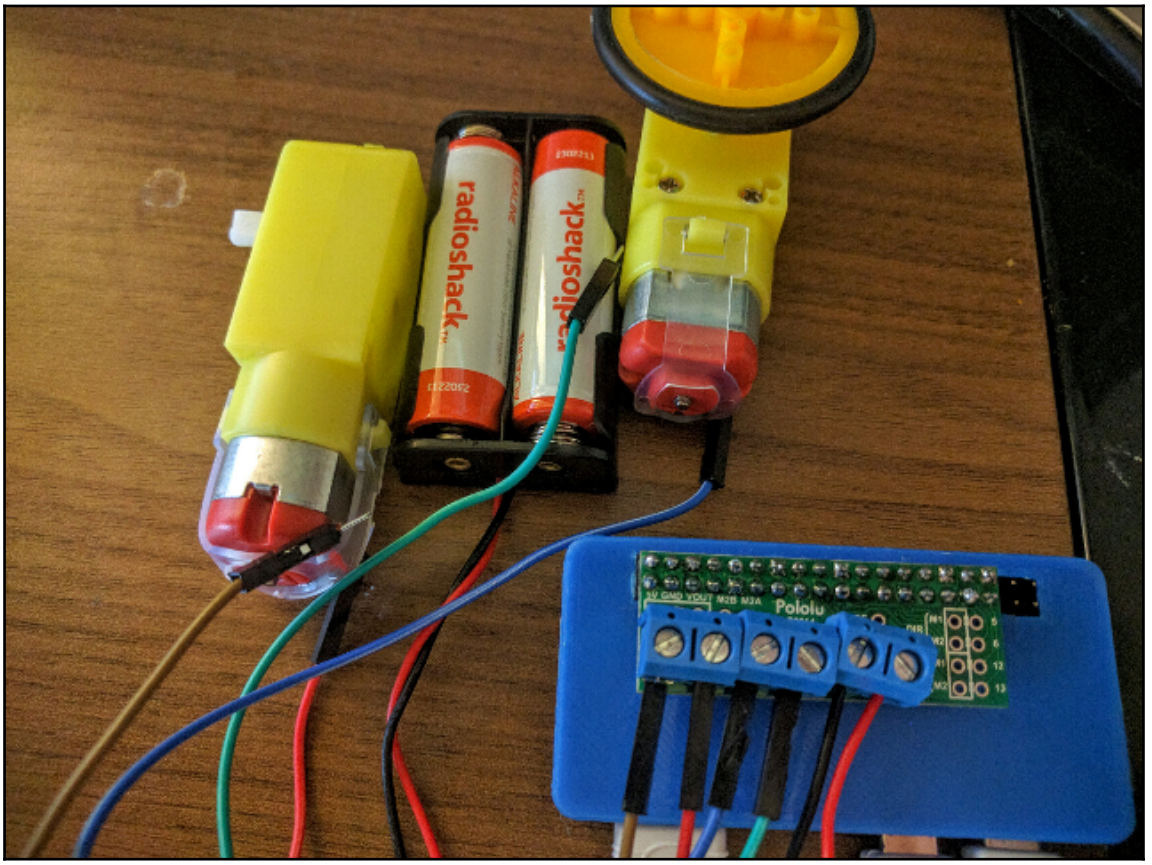
DC Motor Control in Python

In this section, we will discuss motor control using the Raspberry Pi Zero. Why discuss motor control? As we progress through different topics in this book, we will culminate in building a mobile robot. Hence, we need to discuss writing code in Python for controlling a motor using a Raspberry Pi.

In order to control a motor, we need an H Bridge motor driver (Discussing H bridge is beyond scope. There are several resources for H bridge motor drivers: http://www.mcmanis.com/chuck/robotics/tutorial/h-bridge/). There are several motor driver kits designed for the Raspberry Pi. In this section, we will make use of the following kit: https://www.pololu.com/product/2753

The Pololu product page also provides instructions on how to connect the motor. Let's get to writing some Python code for operating the motor:

```
from gpiozero import Motor
from gpiozero import OutputDevice
import time

motor_1_direction = OutputDevice(13)
motor_2_direction = OutputDevice(12)

motor = Motor(5, 6)

motor_1_direction.on()
motor_2_direction.on()

motor.forward()

time.sleep(10)

motor_1_speed.stop()
motor_2_speed.stop()

motor_1_direction.off()
motor_2_direction.off()
```



Raspberry Pi based motor control

In order to control the motor, let's declare the pins, the motor's speed pins and direction pins. Per the motor driver's documentation, the motors are controlled by GPIO pins 12,13 and 5,6 respectively.

```
from gpiozero import Motor
from gpiozero import OutputDevice
import time

motor_1_direction = OutputDevice(13)
motor_2_direction = OutputDevice(12)

motor = Motor(5, 6)
```

- Controlling the motor is as simple as calling the attribute `on()` of the motor class:

```
motor.forward()
```

- Similarly, reversing the motor direction could be done by calling the method reverse(). Stopping the motor could be done by:

```
motor.stop()
```

Some mini-project challenges for the reader:

1. In this chapter, we discussed interfacing inputs for the Raspberry Pi and controlling motors. Think about a project where we could drive a mobile robot that reads inputs from whisker switches and operate a mobile robot. Is it possible to build a wall following robot in combination with the limit switches and motors?
2. We discussed controlling a DC motor in this chapter. How do we control a stepper motor using a Raspberry Pi?
3. How can we interface a motion sensor to control the lights at home using a Raspberry Pi Zero.
4. Read on to find out!

Bonus: Interested in playing tricks on your friends with your Raspberry Pi Zero? Check this book's website!

Summary

...

4

Communication Interfaces

So far, we discussed loops, conditional statements and functions in Python. We also discussed interfacing output devices and interfacing simple digit input devices. Most sensors today come with communication interfaces like **I2C**, **SPI** or **UART**. In this chapter, we will discuss different examples of using these communication interfaces (sensors, displays, and so on).

...

Note: We will be making use of different sensors/electronic components to demonstrate writing code in Python for these interfaces. We leave it up to you to pick a component of your choice to explore these communication interfaces.

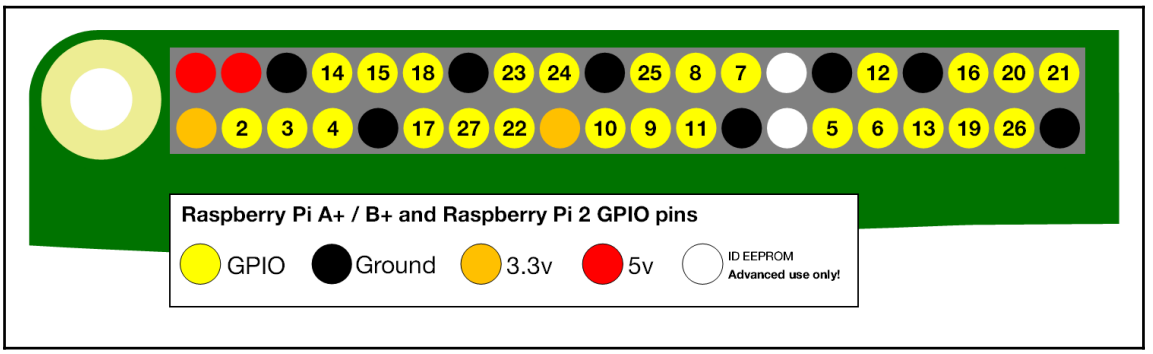
UART (Serial) Port

UART stands for **Universal Asynchronous Receiver Transmitter**. A Serial port is a communication interface where the data is transmitted serially in bits from a sensor to the host computer. Serial port is one of the oldest forms of communication protocol. It is used in data logging where microcontrollers collect data from sensors and transmit the data via serial port. There are also sensors that transmit data via serial communication as responses to incoming commands.

We will not go into the theory behind serial port communications (Plenty of theory available on the web: http://www.computerhope.com/jargon/s/seriport.htm). We will be discussing the use of the serial port to interface different sensors to the Raspberry Pi.

Raspberry Pi Zero's UART port

Typically, UART ports consist of a receiver (*Rx*) and a transmitter (*Tx*) pin that receive and transmit data. The Raspberry Pi's GPIO header comes with an UART port. The GPIO pins 14 (is the *Tx* Pin) and 15 (is the *Rx* Pin) serve as the UART port for the Raspberry Pi.



GPIO Pins 14 and 15 are the UART Pins – Image source: https://www.raspberrypi.org/documentation/usage/gpio-plus-and-raspi2/README.md

Setting up the Raspberry Pi Zero serial port

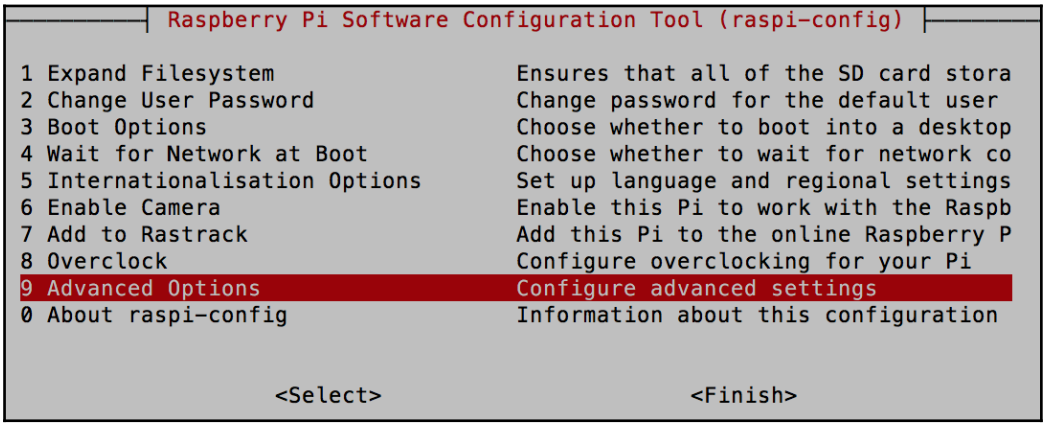
In order to use the serial port to talk to sensors, the serial port login/console needs to be disabled. In the Raspbian OS image, this is enabled by default as it enables easy debugging (demonstrated in a later chapter).

The serial port login can be disabled via **raspi-config**:

1. Launch the terminal and run the command:

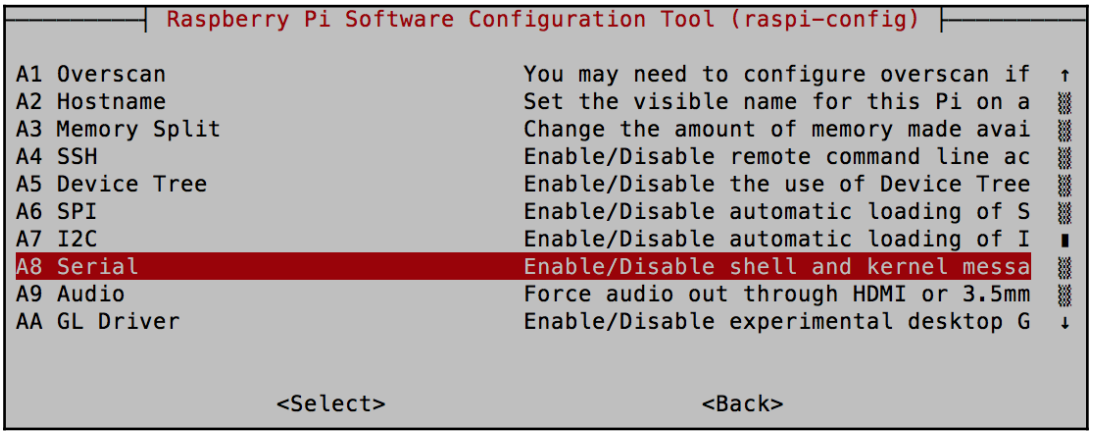
```
sudo raspi-config
```

2. Select advanced options from the main menu of **raspi-config**.



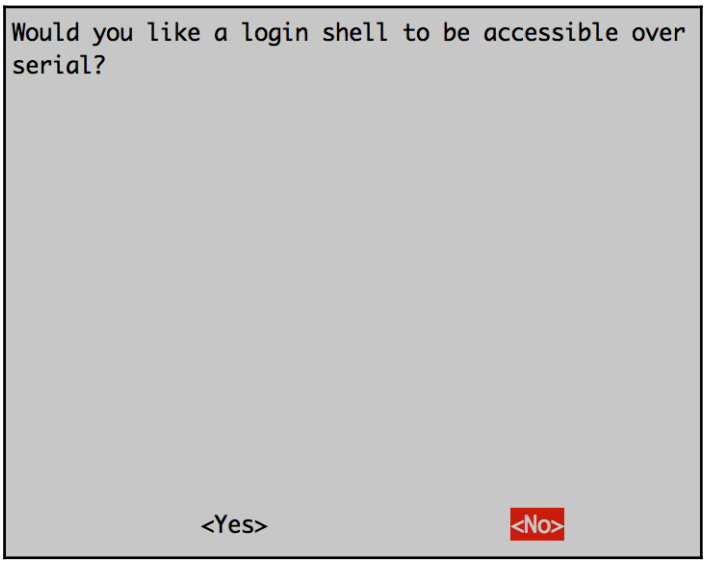
Select advanced options from the raspi-config menu

3. Select option **A8 Serial** from the drop down menu.



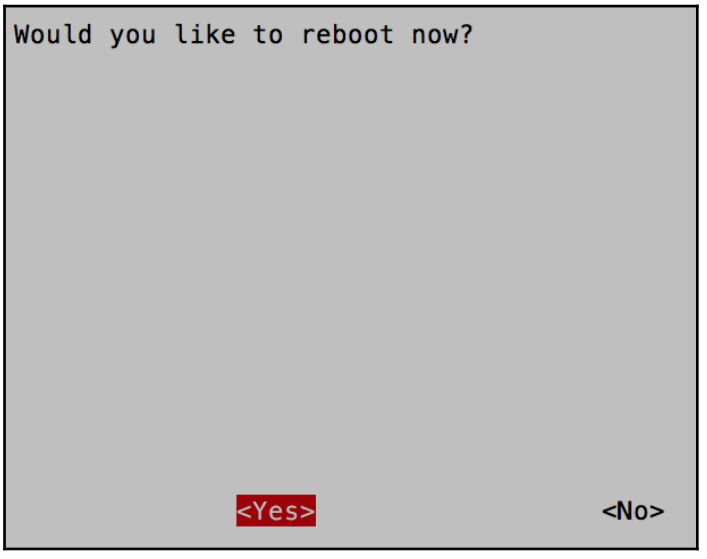
Select A8 Serial from the drop down

4. Disable Serial login



Disable serial login

5. Finish the config and reboot at the end

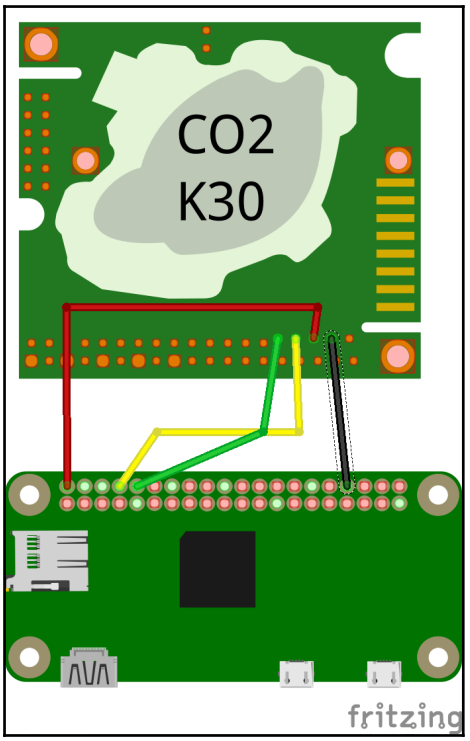


Save config and reboot

Example 1: Interfacing a carbon dioxide sensor to the Raspberry Pi

We will be making use of the K-30 carbon dioxide sensor (Documentation: http://co2meters.com/Documentation/Datasheets/DS30-01%20-%20K30.pdf). It has a range of 0-10,000 ppm and the sensor provides it carbon dioxide concentration readings via serial port as a respond to certain commands from the Raspberry Pi.

The figure below shows the connections between the Raspberry Pi and the K-30 carbon dioxide sensor.



K-30 carbon dioxide sensor interfaced to the Raspberry Pi

The receiver (*Rx*) pin of the sensor is connected to the transmitter (*Tx* – GPIO 14) pin of the Raspberry Pi Zero (yellow figure in the figure below). The transmitter (*Tx*) pin of the sensor is connected to the receiver (*Rx* – GPIO 15) pin of the Raspberry Pi Zero (green wire in the figure below).

In order to power the sensor, the G+ pin of the sensor (red wire in the figure above) is connected to the 5V pin of the Raspberry Pi Zero. The G0 pin of the sensor is connected to the ground pin of the Raspberry Pi Zero (black wire in the figure above).

Typically, serial port communication is initiated by specifying the baud rate, number of bits in a frame, stop bit and flow control.

Python code for serial port communication

We will make use of the pyserial library (https://pyserial.readthedocs.io/en/latest/shortintro.html#opening-serial-ports) for interfacing the carbon dioxide sensor.

1. As per the sensor's documentation, the sensor output can be read by initiating the serial port at a baud rate: 9600, no parity, 8 bits and 1 stop bit. The GPIO serial port is ttyAMA0. The first step in interfacing with the sensor is initiating serial port communication:

```
import serial
ser = serial.Serial("/dev/ttyAMA0")
```

2. As per the sensor documentation (http://co2meters.com/Documentation/Other/SenseAirCommGuide.zip), the sensor responds to the following command for the carbon dioxide concentration:

Reading CO2

Request:

Description	Address 1byte	Command 1-byte	Address (see I2C guide) 2-bytes		N- Bytes to Read 1-byte	Checksum 2-bytes	
Example (reads CO2)	0xFE	0x44	0x00	0x08	0x02	0x9F	0x25

Command Bytes: 0x46- EEPROM Read, 0x44 – RAM Read

Command to read carbon dioxide concentration from the sensor – borrowed from the sensor datasheet

3. The command can be transmitted to the sensor as follows:

```
ser.write(bytearray([0xFE, 0x44, 0x00, 0x08, 0x02, 0x9F, 0x25]))
```

4. The sensor responds with a 7-byte response which can be read as follows:

```
resp = ser.read(7)
```

5. The sensor response is in the following format:

Response

Description	Address 1byte	Command 1-byte	Count 1-byte	N- Bytes Read n-bytes		Checksum 2-bytes	
Example (cont.)	0xFE	0x44	0x02	0x01	0x90		

Carbon dioxide sensor response

6. The message from the sensor could be parsed to calculate the concentration as follows:

```
high = resp[3]
low = resp[4]
co2 = (high*256) + low
```

7. Putting it all together:

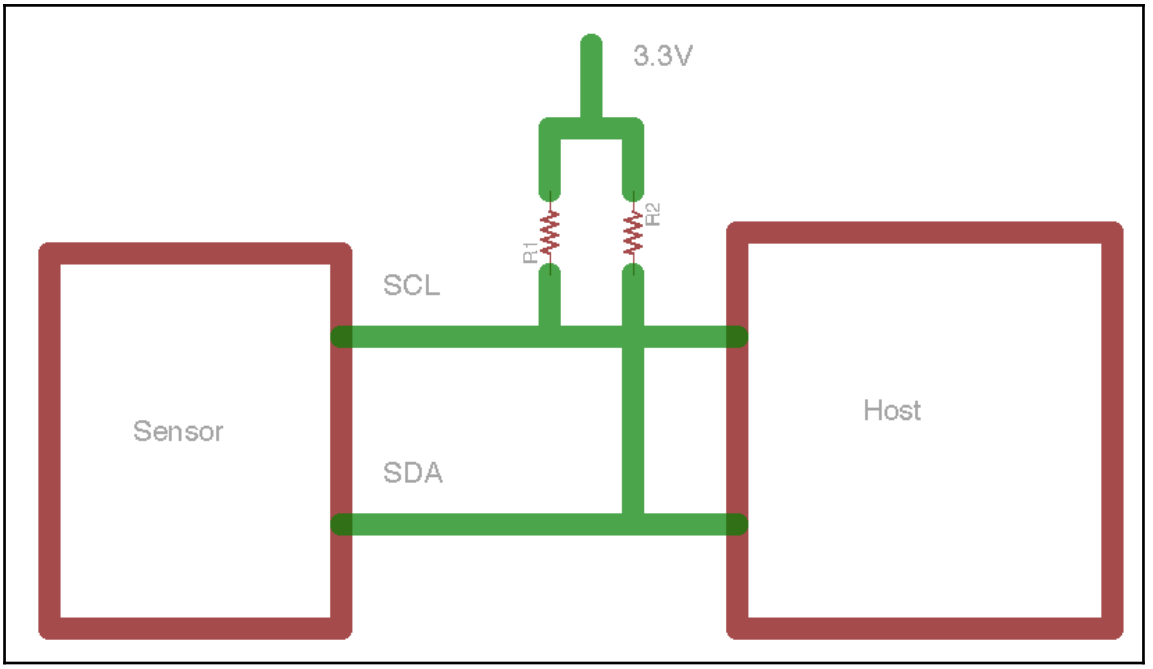
```
import serial
import time
import array
ser = serial.Serial("/dev/ttyAMA0")
print("Serial Connected!")
ser.flushInput()
time.sleep(1)

while True:
   ser.write(bytearray([0xFE, 0x44, 0x00, 0x08, 0x02, 0x9F, 0x25]))
   # wait for sensor to respond
   time.sleep(.01)
   resp = ser.read(7)
   high = resp[3]
   low = resp[4]
   co2 = (high*256) + low
   print()
   print()
   print("Co2 = " + str(co2))
   time.sleep(1)
```

8. Save the code to a file and try executing it.

I²C communication

I2C communication (*Inter-Integrated Circuit*) is a type of serial communication that allows interfacing multiple sensors to the computer. I²C communication consists of two wires of a clock and a data line. The Raspberry Pi Zero's clock and data pins for I²C communication are GPIO 3 and GPIO 2 respectively. In order to communicate with multiple sensors over the same bus, sensors/actuators that communicate via I²C protocol are usually addressed by their 7-bit address. It is possible to have two or more Raspberry Pi boards talking to the same sensor on the same I²C bus. This enables building a sensor network around the Raspberry Pi.

The I²C communication lines are open drain lines and hence they are pulled up using resistors as shown in the figure below:



I2C setup

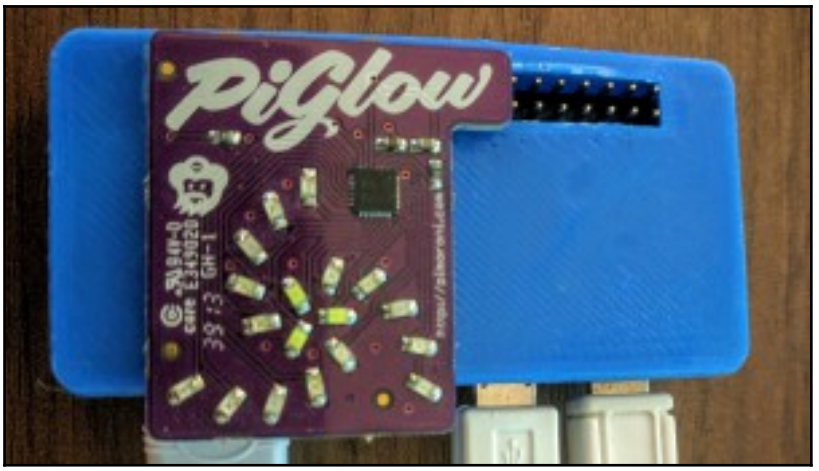
Let's review an example of I²C communication using an example.

Example 2: PiGlow

The **PiGlow** is an add-on hardware for the Raspberry Pi that consists of 18 LEDs interfaced to the SN3218 chip. This chip enables controlling the LEDs via the I^2C interface. The chip's 7-bit address is 0x54.

The add-on hardware is interfaced as follows: The SCL pin is connected to GPIO 3, SDA pin to GPIO 2, the ground pins and the power supply pins are connected to the counterparts of the add-on hardware respectively.

The PiGlow comes with a library that comes with abstracts the I^2C communication: https://github.com/pimoroni/piglow



PiGlow stacked on top of the Raspberry Pi

Installing libraries

The PiGlow library may be installed by running the following from the command line terminal:

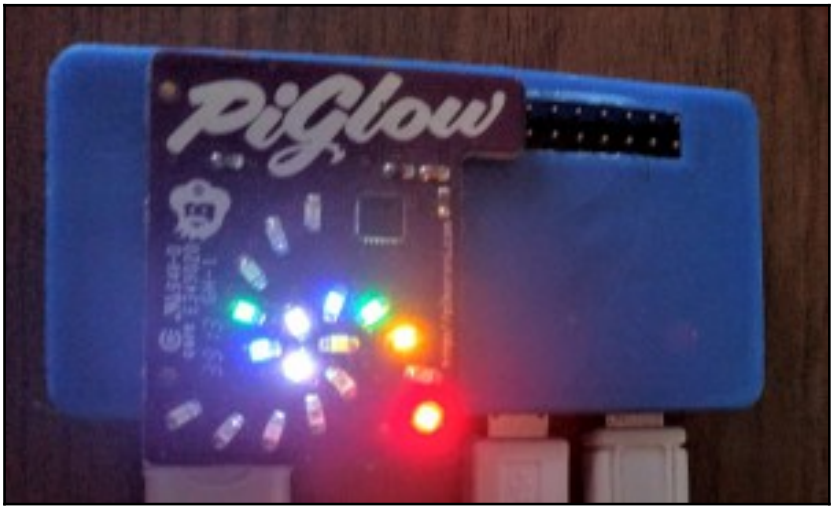
```
curl get.pimoroni.com/piglow | bash
```

Example

Upon completion of installation, switch to the example folder and run one of the examples:

```
python3 bar.py
```

It should run *blinky* light effects as shown in the figure below:



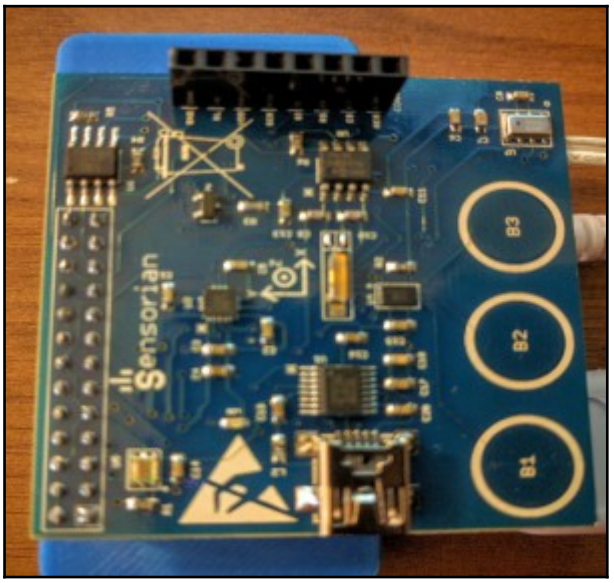
Blinky lights on the PiGlow

Similarly, there are libraries to talk to real time clocks, LCD displays etc. using I^2C communication. If you are interested in writing your own interface that provides the nitty-gritty detail of I^2C communication with sensors/output devices, check out this book's accompanying website for some examples.

Example 3: Sensorian add-on hardware for the Raspberry Pi

The **Sensorian** is an add-on hardware designed for the Raspberry Pi. This add-on hardware comes with different types of sensors including a light sensor, barometer, accelerometer, LCD display interface, Flash memory, capacitive touch sensors and a real time clock.

The sensors on this add-on hardware is sufficient to learn using all the communication interfaces discussed in this chapter.



Sensorian hardware stacked on top of the Raspberry Pi Zero

In this section, we will discuss an example where we will measure the ambient light levels using a Raspberry Pi Zero via the I^2C interface. The sensor on the add-on hardware board is the APDS-9300 sensor (www.avagotech.com/docs/AV02-1077EN).

I2C drivers for the lux sensor

The drivers are available from the GitHub repository for the Sensorian hardware (https://github.com/sensorian/sensorian-firmware.git). Let's clone the repository from the command line terminal

```
git clone https://github.com/sensorian/sensorian-firmware.git
```

Let's make use of the drivers (available in the folder: ~/sensorian-firmware/Drivers_Python/APDS-9300) to read the values from the two ADC channels of the sensor:

```
import time
import APDS9300 as LuxSens
import sys
```

```
AmbientLight = LuxSens.APDS9300()
while True:
  time.sleep(1)
  channel1 = AmbientLight.readChannel(1)
  channel2 = AmbientLight.readChannel(0)
  Lux = AmbientLight.getLuxLevel(channel1,channel2)
  print("Lux output: %d." % Lux)
```

With the ADC values available from both the channel, the ambient light value can be calculated by the driver using the following formula (retrieved from the sensor datasheet):

CH1/CH0	Sensor Lux Formula
0 ≤ CH1/CH0 ≤ 0.52	Sensor Lux = (0.0315 x CH0) – (0.0593 x CH0 x ((CH1/CH0)$^{1.4}$))
0.52 ≤ CH1/CH0 ≤ 0.65	Sensor Lux = (0.0229 x CH0) – (0.0291 x CH1)
0.65 ≤ CH1/CH0 ≤ 0.80	Sensor Lux = (0.0157 x CH0) – (0.0180 x CH1)
0.80 ≤ CH1/CH0 ≤ 1.30	Sensor Lux = (0.00338 x CH0) – (0.00260 x CH1)
CH1/CH0 ≥ 1.30	Sensor Lux = 0

Ambient light levels calculated using the ADC values

This calculation is performed by the attribute getLuxLevel. Under normal lighting conditions, the ambient light level (measured in lux) was around 2. The measured output was 0 when we covered the lux sensor with the palm. This sensor could be used to measure ambient light and adjust the room lighting accordingly.

Challenge

We discussed measuring ambient light levels using the lux sensor. How do we make use of the lux output (ambient light levels) to control the room lighting?

...

SPI interface

There is another type of serial communication interface called the **Serial Peripheral Interface**. This interface has to be enabled via **raspi-config** (this is similar to enabling serial port interface earlier in this chapter). Using the SPI interface is similar to that of I^2C interface and the serial port.

Typically, an SPI interface consists of a Clock line, Data-In, Data-Out and a Slave Select line. Unlike I^2C communication (where we could connect multiple masters), there can be only one master (the Raspberry Pi Zero) but multiple slaves on the same bus. The Slave Select pin enables selecting a specific sensor that the Raspberry Pi Zero is reading/writing data when there are multiple sensors connected to the same bus.

Example 4: Writing to external memory chip

Let's review an example where we write to a Flash memory chip on the Sensorian add-on hardware via the SPI interface. The drivers for the SPI interface and the memory chip are available from the same Github repository.

Since we already have the drivers downloaded, let's review an example available with drivers:

```
import sys
import time
import S25FL204K as Memory
```

Let's initialize and write the message 'hello' to the memory:

```
Flash_memory = Memory.S25FL204K()
Flash_memory.writeStatusRegister(0x00)
message = "hello"
flash_memory.writeArray(0x000000,list(message), message.len())
```

Now, let's try to read the data we just wrote to the external memory:

```
data = flash_memory.readArray(0x000000, message.len())
print("Data Read from memory: ")
print(''.join(data))
```

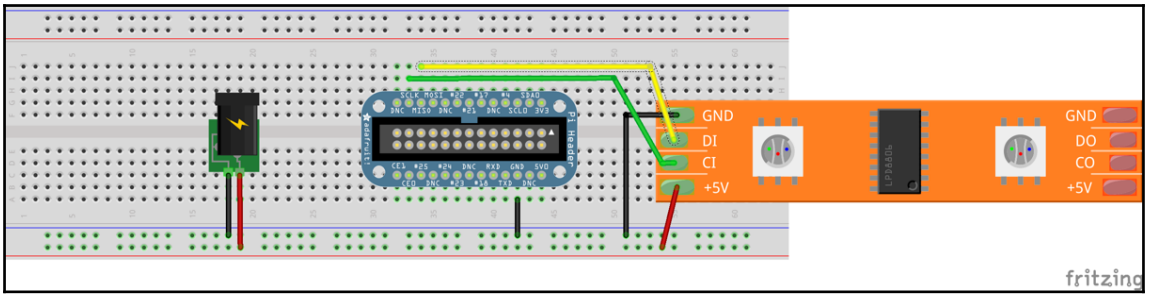
The code sample is available for download with this chapter (memory_test.py).

We were able to demonstrate using the **S**erial **P**eripheral **I**nterface to read/write to an external memory chip.

Note: We have specifically avoided discussing I2C and SPI keeping the beginner in Python programming in mind. There are open source libraries readily available to interface sensors that come with I^2C or SPI interfaces. In order to enable readers, write their own libraries for a specific sensor, we have explained writing device level drivers for I^2C and SPI interfaces on this book's website.

Challenge to the Reader

As you may have guessed by now, we love LEDs and decorative lights. In the figure below, there is an LED strip (https://www.adafruit.com/product/306) interfaced to the SPI interface of the Raspberry Pi add on hardware, the Adafruit Cobbler (https://www.adafruit.com/product/914). We are providing a clue on how to interface the LED strip to the Raspberry Pi Zero. We would like to see if you are able to find a solution to interface the LED strip by yourself.



LED strip interfaced to the Adafruit Cobbler for the Raspberry Pi Zero

If you are not able to get the SPI interface working, don't sweat it! We are using the same LED strip to send out a visual alert when an incoming email alert is received.

Summary

In this chapter, we discussed different communication interfaces to interact with sensors including I^2C, SPI and UART in Python. In the next chapter, we will discuss object oriented programming and its distinct advantages.

5
Object-oriented Programming in Python

In this chapter, we will discuss object oriented programming in Python. We will discuss what it means, why do we need to make use of object oriented programming in python and how to write object oriented code in python for Raspberry Pi based projects (for e.g. Using object-oriented programming to control appliances at home). We will discuss making use of object-oriented programming in a Raspberry Pi Zero project. Finally, we will discuss lists in Python.

Objected-oriented programming in Python

Object-oriented Programming is a concept that helps simplifying your code and eases application development. It is especially useful in reusing your code. Object-oriented code enables reusing your code for sensors that use the communications interface. For e.g.: All sensors that are equipped with a UART port could be grouped together using object-oriented code.

A great example of object-oriented programming is the GPIO Zero library (https://www.raspberrypi.org/blog/gpio-zero-a-friendly-python-api-for-physical-computing/) used in previous chapters. In fact, everything is an object in Python.

Object-oriented code is especially helpful in collaboration with other people on a project. For e.g.: You could implement a sensor driver using object-oriented code in Python and document its usage. This enables other developers to develop an application without paying attention to the nitty-gritty detail behind the sensor's interface. Object-oriented programming provides modularity to an application that simplifies application development. We are going to review an example in this chapter that demonstrates the

advantage of object-oriented programming. In this chapter, we will be making use of object-oriented programming to bring modularity to our project.

Let's get started!

Revisiting the student id card example

Let's revisit the ID card example from chapter 2 (input_test.py). We discussed writing a simple program that captures and prints the information belonging to a student. A student's contact information could be retrieved and stored as follows:

```
name = input("What is your name? ")
address = input("What is your address? ")
age = input("How old are you? ")
```

Now, consider a scenario where the information of 10 students has to be saved and retrieved at any point during program execution. We would need to come up with a nomenclature for the variables used to save the student information. It would be a clutter if we use 30 different variables to store information belonging to each student. This is where object oriented programming can be really helpful.

Let's rewrite this example using object-oriented programming to simplify the problem. The first step in object-oriented programming is declaring a structure for the object. This is done by defining a Class. The Class determines the functions of an object. Let's write a python class that defines the structure for a student object.

Class

Since we are going to save student information, the class is going to be called Student. A class is defined using the class keyword as follows:

```
class Student(object):
```

Thus, a class called Student has been defined. Whenever a new object is created, the method __init__() (the underscore indicate that the init method is a magic method i.e. it is a function that is called by Python when an object is created)is called internally by Python. This method is defined within the class:

```
class Student(object):
    """A Python class to store student information"""

    def __init__(self, name, address, age):
        self.name = name
```

```
        self.address = address
        self.age = age
```

In this example, the arguments to the __init__ method include name, age and address. These arguments are called **attributes**. These attributes enable creating a unique object that belongs to the Student class. Hence, in this example, while creating an instance of the Student class, the attributes name, age and address are required arguments.

Let's create an object (also called an instance) belonging to the student class:

```
student1 = Student("John Doe", "29", "123 Main Street, Newark, CA")
```

In this example, we created an object belonging to the Student class called student1 where John Doe (name), 29 (age) and 123 Main Street, Newark, CA(address) are attributes required to create an object. When we create an object that belongs to the Student class by passing the requisite arguments (declared earlier in the __init__() method of the Student class), the __init__() method is automatically called to initialize the object. Upon initialization, the information related to student1 is stored under the object student1.

Now, the information belonging to student1 could be retrieved as follows:

```
print(student1.name)
print(student1.age)
print(student1.address)
```

Now, let's create another object called student2:

```
student2 = Student("Jane Doe", "27", "123 Main Street, San Jose, CA")
```

We created two objects called student1 and student2. Each objects attributes are accessible as student1.name, student2.name and so on. In the absence of object oriented programming, we will have to create variables like student1_name, student1_age, student1_address, student2_name, student2_age and student2_address and so on. Thus, object-oriented programming enables modularizing the code.

Adding methods to a class

Let's add some methods to our Student class that would help retrieve a student's information:

```
class Student(object):
    """A Python class to store student information"""

    def __init__(self, name, age, address):
```

```
        self.name = name
        self.address = address
        self.age = age

    def return_name(self):
        """return student name"""
        return self.name

    def return_age(self):
        """return student age"""
        return self.age

    def return_address(self):
        """return student address"""
        return self.address
```

In this example, we have added three methods namely return_name(), return_age() and return_address() that returns the attributes name, age and address respectively. These methods of a class are called **callable attributes**. Let's review a quick example where we make use of these callable attributes to print an object's information.

```
student1 = Student("John Doe", "29", "123 Main Street, Newark, CA")
print(student1.return_name())
print(student1.return_age())
print(student1.return_address())
```

So far, we discussed methods that retrieves information about a student. Let's include a method in our class that enables updating information belonging to a student. Now, let's add another method to the class that enables updating address by a student:

```
def update_address(self, address):
    """update student address"""
    self.address = address
    return self.address
```

Let's compare student1's address before and after updating the address:

```
print(student1.address())
print(student1.update_address("234 Main Street, Newark, CA"))
```

This would print the following output to your screen:

```
123 Main Street, Newark, CA
234 Main Street, Newark, CA
```

Thus, we have written our first object-oriented code that demonstrates the ability to modularize the code. The above code sample is available for download along with this

chapter as *student_info.py*.

Doc strings in Python

In the object oriented example, you might have noticed a sentence enclosed in triple double quotes:

```
"""A Python class to store student information"""
```

This is called a **doc string**. The doc string is used to document information about a class or a method. Doc strings are especially helpful while trying to store information related to the usage of a method or a class (this will be demonstrated later in this chapter). Doc strings are also used at the beginning of a file to store multi-line comments related to an application or a code sample. Doc strings are ignored by the python interpreter and they are meant to provide documentation about a class to fellow programmers.

Similarly, the python interpreter ignores any single line comment that starts with a '#' sign. Single line comments are generally used to make a specific note on a block of code. The practice of including well-structured comments makes your code readable.

For example, the following code snippet informs the reader that a random number between 0 and 9 is generated and stored in the variable rand_num:

```
# generate a random number between 0 and 9
rand_num = random.randrange(0,10)
```

On the contrary, a comment that provides no context is going to confuse someone who is reviewing your code:

```
# Todo: Fix this later
```

It is quite possible that you may not be able to recall what needs fixing when you revisit the code at a later time.

self

In our object oriented example, the first argument to every method had an argument called self. self refers to the instance of the class in use and the self keyword is used as the first argument in methods that interact with the instances of the class. In the above example, self refers to the object student1. It is equivalent to initializing an object and accessing it as follows:

```
Student(student1, "John Doe", "29", "123 Main Street, Newark, CA")
```

```
Student.return_address(student1)
```

The self keyword simplifies how we access an object's attributes in this case. Now, let's review some examples where we make use of object-oriented programming involving the Raspberry Pi.

Speaker controller

Let's write a python class (tone_player.py in **downloads**) that plays a musical tone indicating that the boot-up of your Raspberry Pi is complete. For this section, you will need a USB sound card and a speaker interfaced to the USB hub of the Raspberry Pi.

Let's call our class TonePlayer. This class should be capable of controlling the speaker volume and playing any file passed as an argument while creating an object.

```
class TonePlayer(object):
   """A Python class to play boot-up complete tone"""

   def __init__(self, file_name):
      self.file_name = file_name
```

In this case, the file that has to be played by the TonePlayer class has to be passed an argument. For example:

```
tone_player = TonePlayer("/home/pi/tone.wav")
```

We also need to be able to set the volume level at which the tone has to be played. Let's add a method to do the same:

```
def set_volume(self, value):
      """set tone sound volume"""
      subprocess.Popen(["amixer", "set", "'PCM'", str(value)], shell=False)
```

In the method set_volume, we make use of Python's subprocess module to run the Linux system command that adjusts the sound drive volume.

The most essential method for this class is the play command. When the play method is called, we need to play the tone sound using Linux's aplay command:

def play(self):

“”"play the wav file"""

subprocess.Popen([“aplay”, self.file_name], shell=False)

Putting it altogether:

```
import subprocess

class TonePlayer(object):
    """A Python class to play boot-up complete tone"""

    def __init__(self, file_name):
        self.file_name = file_name

    def set_volume(self, value):
        """set tone sound volume"""
        subprocess.Popen(["amixer", "set", "'PCM'", str(value)], shell=False)

    def play(self):
        """play the wav file"""
        subprocess.Popen(["aplay", self.file_name], shell=False)


if __name__ == "__main__":
    tone_player = TonePlayer("/home/pi/tone.wav")
    tone_player.set_volume(75)
    tone_player.play()
```

Save the TonePlayer class to your Raspberry Pi (save it to a file called tone_player.py) and use a tone sound file from sources like *freesound* (https://www.freesound.org/people/zippi1/sounds/18872/). Save it to a location of your choice and try running the code. It should play the tone sound at the desired volume!

Now, edit /etc/rc.local and add the following line to the end of the file (Right before the line exit 0):

```
python3 /home/pi/toneplayer.py
```

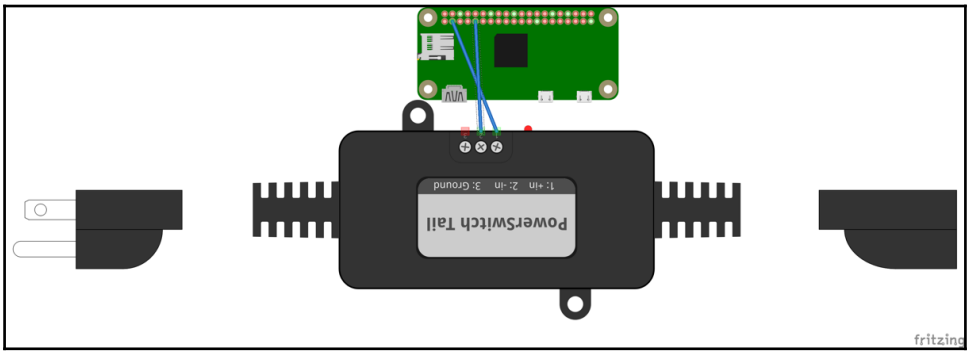
This should play a tone whenever the Pi boots up!

Light Control Daemon

Let's review another example where we implement a simple daemon using object-oriented programming that turns on/off lights at specified times of the day. In order to be able to perform tasks at scheduled times, we will make use of the schedule library (https://github.com/dbader/schedule). It could be installed as follows:

```
pip3 install schedule
```

Let's call our class LightScheduler. It should be capable of accepting start and top times to turn on/off lights at given times. It should also provide override capabilities to let the user turn on/off lights as necessary. Let's assume that the light is controlled using a PowerSwitch Tail ii (http://www.powerswitchtail.com/Pages/default.aspx). It is interfaced as follows:



Raspberry Pi Zero interfaced to the PowerSwitch Tail ii

```
class LightScheduler(object):
    """A Python class to turn on/off lights"""

    def __init__(self, start_time, stop_time):
        self.start_time = start_time
        self.stop_time = stop_time
        # lamp is connected to GPIO pin2.
        self.lights = OutputDevice(2)
```

Whenever an instance of the LightScheduler is created, the GPIO pin is initialized to control the PowerSwitch Tail II. Now, let's add methods to turn on/off lights.

```
def init_schedule(self):
        # set the schedule
        schedule.every().day.at(self.start_time).do(self.on)
        schedule.every().day.at(self.stop_time).do(self.off)

    def on(self):
        """turn on lights"""
        self.lights.on()

    def off(self):
        """turn off lights"""
        self.lights.off()
```

In the method init_schedule(), the start and stop times that were passed as arguments are used to initialize schedule to turn on/off the lights at the specified times.

Putting it together, we have:

```
import schedule
import time
from gpiozero import OutputDevice

class LightScheduler(object):
    """A Python class to turn on/off lights"""

    def __init__(self, start_time, stop_time):
        self.start_time = start_time
        self.stop_time = stop_time
        # lamp is connected to GPIO pin2.
        self.lights = OutputDevice(2)

    def init_schedule(self):
        # set the schedule
        schedule.every().day.at(self.start_time).do(self.on)
        schedule.every().day.at(self.stop_time).do(self.off)

    def on(self):
        """turn on lights"""
        self.lights.on()

    def off(self):
        """turn off lights"""
        self.lights.off()


if __name__ == "__main__":
    lamp = LightScheduler("18:30", "9:30")
    lamp.on()
    time.sleep(50)
    lamp.off()
    lamp.init_schedule()
    while True:
        schedule.run_pending()
        time.sleep(1)
```

In the above example, the lights are scheduled to be turned on at 6:30 p.m. and turned off at 9:30 a.m. Once the jobs are scheduled, the program enters an infinite loop where it awaits task execution. This example could be run as a daemon by executing the file at start-up (add a line called light_scheduler.py to /etc/rc.local). After scheduling the job, it will continue to run as a daemon in the background.

Note: This is just a basic introduction to object-oriented programming and its applications (keeping the beginner in mind). Refer to this book's website for more examples on object-oriented programming.

Let's switch gears and discuss one major type of *collections in Python* i.e. lists.

Lists

The topics discussed in this chapter can be difficult to grasp unless used in practice. Any example that is represented using this notation: >>> indicates that it could be tested using the Python interpreter.

In Python, a *list* is a datatype (Documentation: https://docs.python.org/3.4/tutorial/datastructures.html#) that could be used to store elements in a sequence. A list may consist of strings, objects (discussed in detail in this chapter) or numbers, and so on. For example: the following is an example of a list:

```
>>> sequence = [1, 1, 2, 3, 4, 5, 6, 7]
>>> example_list = ['apple', 'orange', 1.0, 2.0, 3]
```

In the above example, the list sequence consists of numbers between 1 and 6 while the list example_list consists of a combination of strings, integer and floating-point numbers. A list is represented by square brackets ([]). Items can be added to a list separated by commas.

```
>>> type(sequence)
<class 'list'>
```

Since a list is an ordered sequence of elements, the elements of a list could be fetched by iterating through the list elements using a for-loop as follows:

```
for item in sequence:
  print("The number is ", item)
```

The output is something like:

```
The number is  1
The number is  2
The number is  3
The number is  4
The number is  5
The number is  6
```

Since Python's loop can iterate through a sequence of elements, it fetches each element and assigns it to item. This item is printed on the console.

Operations that could be performed on a list

In Python, the attributes of a datatype can be retrieved using the method dir(). For example, the attributes available for the list sequence can be retrieved as follows:

```
>>> dir(sequence)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
'__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__',
'__hash__', '__iadd__', '__imul__', '__init__', '__iter__', '__le__', '__len__', '__lt__', '__mul__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__',
'__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'append', 'clear', 'copy',
'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

These attributes enable performing different operations on a list. Let's discuss each attribute in detail:

Append element to list:

It is possible to add an element using the method append():

```
>>> sequence.append(7)
>>> sequence
[1, 2, 3, 4, 5, 6, 7]
```

Remove element from list:

The method remove() finds the first instance of the element (passed an argument) and removes it from the list. Let's consider the following examples:

Example 1:

```
>>> sequence = [1, 1, 2, 3, 4, 7, 5, 6, 7]
>>> sequence.remove(7)
>>> sequence
[1, 1, 2, 3, 4, 5, 6, 7]
```

Example 2:

```
>>> sequence.remove(1)
>>> sequence
```

```
[1, 2, 3, 4, 5, 6, 7]
```

Example 3:

```
>>> sequence.remove(1)
>>> sequence
[2, 3, 4, 5, 6, 7]
```

Retrieving the index of an element

The method index() returns the position of an element in a list:

```
>>> index_list = [1, 2, 3, 4, 5, 6, 7]
>>> index_list.index(5)
4
```

In this example, the method returns the index of the element 5. Since Python uses zero-based indexing that is the index is counted from 0 and hence the index of the element 5 is 4.

```
random_list = [2, 2, 4, 5, 5, 5, 6, 7, 7, 8]
>>> random_list.index(5)
3
```

In this example, the method returns the position of the first instance of the element. The element 5 is located at the third position.

Popping an element from the list

The method pop() enables removing an element from a specified position and return it:

```
>>> index_list = [1, 2, 3, 4, 5, 6, 7]
>>> index_list.pop(3)
4
>>> index_list
[1, 2, 3, 5, 6, 7]
```

In this example, the list index_list consists of numbers between 1 and 7. When the third element is popped by passing the index position (3) as an argument, the number 4 is removed from the list and returned.

If no arguments are provided for the index position, the last element is popped and returned:

```
>>> index_list.pop()
7
```

```
>>> index_list
[1, 2, 3, 5, 6]
```

In this example, the last element (7) was popped and returned.

Counting the instances of an element:

The method count() returns the number of times an element appears in a list. For example, the element appears twice in the list: random_list.

```
>>> random_list = [2, 9, 8, 4, 3, 2, 1, 7]
>>> random_list.count(2)
2
```

Inserting element at a specific position:

The method insert() enables adding an element at a specific position in the list. For example, let's consider the following example:

```
>>> day_of_week = ['Monday', 'Tuesday', 'Thursday', 'Friday', 'Saturday']
```

In the list, Wednesday is missing. It needs to be positioned between Tuesday and Thursday at position 2 (Python uses **zero based indexing** that is the positions/indexes of elements are counted as 0,1,2 and so on.). It could be added using insert as follows:

```
>>> day_of_week.insert(2, 'Wednesday')
>>> day_of_week
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday']
```

Challenge to the reader

In the above list, Sunday is missing. Use the insert attribute of lists to insert it at the correct position.

Extending a list

Two lists can be combined together using the method extend(). The lists day_of_week and sequence can be combined as follows:

```
>>> day_of_week.extend(sequence)
>>> day_of_week
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 1,
```

```
2, 3, 4, 5, 6]
```

Lists can also be combined as follows:

```
>>> [1, 2, 3] + [4, 5, 6]
[1, 2, 3, 4, 5, 6]
```

It is also possible to add a list as an element to another list:

```
sequence.insert(6, [1, 2, 3])
>>> sequence
[1, 2, 3, 4, 5, 6, [1, 2, 3]]
```

Clearing the elements of a list

All the elements of a list could be deleted using the method clear().

```
>>> sequence.clear()
>>> sequence
[]
```

Sorting the elements of a list

The elements of a list could be sorted using the sort() method.

```
random_list = [8, 7, 5, 2, 2, 5, 7, 5, 6, 4]
>>> random_list.sort()
>>> random_list
[2, 2, 4, 5, 5, 5, 6, 7, 7, 8]
```

When a list consists of a collection of strings, they are sorted in the alphabetical order.

```
>>> day_of_week = ['Monday', 'Tuesday', 'Thursday', 'Friday', 'Saturday']
>>> day_of_week.sort()
>>> day_of_week
['Friday', 'Monday', 'Saturday', 'Thursday', 'Tuesday']
```

Reverse the order of elements in list

The method reverse() enables the reversing the order of the list elements:

```
>>> random_list = [8, 7, 5, 2, 2, 5, 7, 5, 6, 4]
>>> random_list.reverse()
>>> random_list
[4, 6, 5, 7, 5, 2, 2, 5, 7, 8]
```

Create copies of a list

The method copy() enables creating copies of a list:

```
>>> copy_list = random_list.copy()
>>> copy_list
[4, 6, 5, 7, 5, 2, 2, 5, 7, 8]
```

Accessing list elements

The list elements could be accessed by specifying the index position of the element list_name[i]. For example: The zeroth list element of the list random_list could be accessed as follows:

```
random_list = [4, 6, 5, 7, 5, 2, 2, 5, 7, 8]
>>> random_list[0]4>>> random_list[3]7
```

Accessing a set of elements within a list

- It is possible to access elements between specified indices. For example: It is possible to retrieve all elements between indices 2 and 4:

```
>>> random_list[2:5]
[5, 7, 5]
```

- The first six elements of a list could be accessed as follows:

```
>>> random_list[:6]
[4, 6, 5, 7, 5, 2]
```

- The elements of a list could be printed in the reverse order as follows:

```
>>> random_list[::-1]
[8, 7, 5, 2, 2, 5, 7, 5, 6, 4]
```

- Every second element in the list could be fetched as follows:

```
>>> random_list[::2]
[4, 5, 5, 2, 7]
```

- It is also possible to fetch every second element after the second element after skipping the first two elements

```
>>> random_list[2::2]
[5, 5, 2, 7]
```

List membership

It is possible to check if a value is a member of a list using in keyword. For example:

```
random_list = [2, 1, 0, 8, 3, 1, 10, 9, 5, 4]
```

In this list, we could check if the number 6 is a member:

```
>>> 6 in random_list
False
>>> 4 in random_list
True
```

Let's build a simple game!

This exercise consists of two parts. In the first part, we will review building a list containing ten random numbers between and 10. The second part is a challenge to the reader.

1. The first step is creating an empty list. Let's create an empty list called random_list. An empty list can be created as follows:

   ```
   random_list = []
   ```

2. We will be making use of Pythons random module (https://docs.python.org/3/library/random.html) to generate random numbers. In order to generate random numbers between and 10, we will make use of the method randint() from the random module:

   ```
   random_number = random.randint(0,10)
   ```

3. Let's append the generated number to the list. This operation is repeated 10 times using a for loop:

   ```
   for index in range(0,10):
       random_number = random.randint(0, 10)
       random_list.append(random_number)
   print("The items in random_list are  ")
   print(random_list)
   ```

4. The generated list looks something like this:

   ```
   The items in random_list are
   [2, 1, 0, 8, 3, 1, 10, 9, 5, 4]
   ```

We discussed generating a list of random numbers. The next step is taking user input where

we ask the user to make a guess for a number between and 10. If the number is a member of the list, the message **Your guess is correct** is printed to the screen, else, the message **Sorry! Your guess is incorrect** is printed. We leave the second part as a challenge to the reader. Get started with the code sample list_generator.py available for download with this chapter.

Note: There are other data types including dicts and tuples that we have not discussed in this chapter. We will discuss them in the upcoming chapters.

Summary

In this chapter, we discussed lists and the advantages of object-oriented programming. We discussed object-oriented programming examples using the Raspberry Pi as the center of the examples. Since the book is targeted mostly towards beginners, we decided to stick to the basics of object-oriented programming while discussing examples. There are advanced aspects that are beyond the scope of the book. We leave it up to the reader to learn advanced concepts using other examples available on this book's site.

6
File I/O and Python utilities

In this chapter, we are going to extensively discuss file i/o i.e. reading, writing and appending to file. We are also going to discuss python utilities that enable manipulating files and interacting with the operating system. Each topic has a different level of complexity that we will discuss using an example. Let's get started!

File I/O

We are discussing File I/O for two reasons:

- In the world of Linux operating systems, everything is a file. Interaction with peripherals on the Raspberry Pi is similar to reading from/writing to a file. For example: In chapter 4, we discussed serial port communication. You should be able to observe that serial port communication similar to a file read/write operation.
- We use File I/O in some form in every project. For example: Writing sensor data to a csv file or reading pre-configured options for a webserver, and so on.

Hence, we thought it would be useful to discuss File I/O in Python as its own chapter (Detailed documentation available from here: https://docs.python.org/3/tutorial/inputoutput.html#reading-and-writing-files) and discuss examples where it could play a role while developing applications on the Raspberry Pi Zero.

Reading from a file

Let's create a simple text file read_file.txt with the following text: **I am learning Python Programming using the Raspberry Pi Zero** and save it to the code samples directory (or

any location of your choice).

In order to read from a file, we need to make use of the python's in-built function: open to open the file. Let's take a quick look at a code snippet that demonstrates opening a text file to read its content and print it to the screen:

```
if __name__ == "__main__":
    # open text file to read
    file = open('read_file.txt', 'r')
    # read from file and store it to data
    data = file.read()
    print(data)
    file.close()
```

Let's discuss this code snippet in detail:

1. The first step in reading the contents of the text file is opening the file using the in-built function open. The file in question needs to be passed as an argument along with a flag r that indicates we are opening the file to read the contents (We will discuss other flag options as we discuss each reading/writing files.)
2. Upon opening the file, the open function returns a pointer (address to the file object) that is stored in the variable file.

   ```
   file = open('read_file.txt', 'r')
   ```

3. This file pointer is used to read the contents of the file and print it to the screen:

   ```
   data = file.read()
   print(data)
   ```

4. After reading the contents of the file, the file is closed by calling the function close().

Note: It is absolutely not necessary to close files after the reading its contents. The files are closed while performing routine clean-ups (known as garbage collection) or when the program has completed execution. In complex programs, these clean-ups may happen at random occasions. This might result in memory leaks. As the saying goes, *Clean up after yourself*.

Run the above code snippet (available for download along with this chapter – read_from_file.py) using IDLE3 or the command line terminal. The contents of the text file would be printed to the screen as follows:

```
I am learning Python Programming using the Raspberry Pi Zero
```

Reading Lines

Sometimes, it is necessary to read the contents of a file by line-by-file. In Python, there are two options to do this: readline() and readlines().

readline

As the name suggests, this in-built function enables reading one line at a time. Let's review this using an example:

```
if __name__ == "__main__":
  # open text file to read
  file = open('read_line.txt', 'r')

  # read a line from the file
  data = file.readline()
  print(data)

  # read another line from the file
  data = file.readline()
  print(data)

  file.close()
```

- When the above code snippet is executed (available for download as read_line_from_file.pyalong with this chapter), the file read_line.txt is opened and a single line is returned by the readline() function. This line is stored in the variable data. Since the function is called twice in this program, the output is as follows:

  I am learning Python Programming using the Raspberry Pi Zero.

  This is the second line.

- A new line is returned every time the readline function is called and it returns an empty string when the end-of-file has reached.

readlines

This function reads the entire content of a file in lines and stores each it to a list:

```
if __name__ == "__main__":
  # open text file to read
  file = open('read_lines.txt', 'r')

  # read a line from the file
```

```
data = file.readlines()
for line in data:
    print(line)

file.close()
```

- Since the lines of the files is stored as a list, it could be retrieved by iterating through the list:

```
data = file.readlines()
  for line in data:
      print(line)
```

- The above code snippet is available for download along with this chapter as read_lines_from_file.py.

Writing to a file

1. The first step in writing to a file is opening a file with the write flag: w. If the file name that was passed as an argument doesn't exist, a new file is created.

```
file = open('write_file.txt', 'w')
```

2. Once the file is open, the next step is passing the string to be written as argument to the write() function.

```
file.write('I am excited to learn Python using Raspberry Pi
Zero')
```

3. Let's put the code together where we write a string to a text file, close it, re-open the file and print the contents of the file to the screen.

```
if __name__ == "__main__":
  # open text file to read
  file = open('write_file.txt', 'w')
  # read a line from the file
  file.write('I am excited to learn Python using Raspberry Pi
  Zero')
  file.close()

  file = open('write_file.txt', 'r')
  data = file.read()
  print(data)
  file.close()
```

4. The above code snippet is available for download along with this chapter (write_to_file.py).
5. When the above code snippet is executed, the output is something like shown below:

```
I am excited to learn Python using Raspberry Pi Zero
```

Appending to a file

- Whenever a file is opened using the write flag w, the contents of the file are deleted and opened afresh to write data. There is an alternative flag a that enables appending data to the end of the file. This flag also creates a new file if the file (that is passed as an argument to open) doesn't exist. Let's consider the code snippet below where we append a line to the text file write_file.txt from the previous section:

```
if __name__ == "__main__":
  # open text file to read
  file = open('write_file.txt', 'a')
  # read a line from the file
  file.write('This is a line appended to the file')
  file.close()

  file = open('write_file.txt', 'r')
  data = file.read()
  print(data)
  file.close()
```

- When the above code snippet is executed (available for download along with this chapter – append_to_file.py), the string **This is a line appended to the file** is appended to the end of the text of the file. The contents of the file will include the following:

```
I am excited to learn Python using Raspberry Pi Zero
This is a line appended to the file
```

seek

Once a file is opened, the file pointer that is used in file I/O moves from the beginning to the end of the file. It is possible to move the pointer to a specific position and read the data from that position. This is especially useful when we are interested in a specific line of a file. Let's

consider the text file write_file.txt from the previous example. The contents of the file include:

```
I am excited to learn Python using Raspberry Pi Zero
This is a line appended to the file
```

Let's try to skip the first line and read only the second line using seek:

```
if __name__ == "__main__":
  # open text file to read

  file = open('write_file.txt', 'r')
  # read the second line from the file
  file.seek(53)
  data = file.read()
  print(data)
  file.close()
```

In the above example (available for download along with this chapter as seek_in_file.py), the seek function is used to move the pointer to byte 53 that is the end of first line. Then the file's contents are read and stored into the variable. When this code snippet is executed, the output is as follows:

```
This is a line appended to the file
```

Thus, seek enables moving the file pointer to a specific position.

Read 'n' bytes

The seek function enables moving the pointer to a specific position and reading a byte or n bytes from that position. Let's re-visit reading write_file.txt and try to read the word *excited* in the sentence *I am excited to learn Python using Raspberry Pi Zero*.

```
if __name__ == "__main__":
  # open text file to read and write
  file = open('write_file.txt', 'r')
  # set the pointer to the desired position
  file.seek(5)
  data = file.read(1)
  print(data)
  # rewind the pointer
  file.seek(5)
  data = file.read(7)
  print(data)
  file.close()
```

1. In the first step, the file is opened using the read flag and the file pointer is set to

the 5th byte (using seek) – the position of the letter **e** in the contents of the text file.

2. Now, we read one byte from the file by passing it as an argument to the read function. When an integer is passed as an argument, the read function returns the corresponding number of bytes from the file. When no argument is passed, it reads the entire file. The read function returns an empty string if the file is empty.

```
file.seek(5)
data = file.read(1)
print(data)
```

3. In the second part, we try to read the word *excited* from the text file. We rewind the position of the pointer back to the 5th byte. Then we read seven bytes from the file (length of the word *excited*).
4. When the code snippet is executed (available for download along with this chapter as seek_to_read.py), the program should print the letter e and the word excited.

```
file.seek(5)
data = file.read(7)
print(data)
```

r+

We discussed reading and writing to files using the r and w flags. There is another called r+. This flag enables reading and writing to a file. Let's review an example that enables us to understand this flag.

Let's review the contents of write_file.txt once again:

```
I am excited to learn Python using Raspberry Pi Zero
This is a line appended to the file
```

Let's modify the second line to read: **This is a line that was modified**. The code sample is available for download along with this chapter as seek_to_write.py.

```
if __name__ == "__main__":
  # open text file to read and write
  file = open('write_file.txt', 'r+')
  # set the pointer to the desired position
  file.seek(68)
  file.write('that was modified   ')
  # rewind the pointer to the beginning of the file
  file.seek(0)
  data = file.read()
```

```
print(data)
file.close()
```

Let's review how this example works:

1. The first step in this example is opening the file using the r+ flag. This enables reading and writing to the file.
2. The next step is moving to the 68th byte of the file
3. The string 'that was modified' is written to the file at this position. The spaces at the end of the string are used to overwrite the original content of the second sentence.
4. Now, the file pointer is set to the beginning of the file and its contents are read.
5. When the above code snippet is executed, the modified file contents are printed to the screen as shown below:

```
I am excited to learn Python using Raspberry Pi Zero
This is a line that was modified
```

There is another flag a+ that enables appending data to the end of the file and reading at the same time. We will leave this to the reader to figure out using the examples discussed so far.

Note: We have discussed different examples on reading and writing to files in Python. It can be overwhelming without sufficient experience in programming. We strongly recommend working through the different code samples provided in this chapter

Challenge to the reader:

Use the a+ flag to open the file write_file.txt (discussed in different examples) and append a line to the file. Set the file pointer using seek and print its contents. You may open the file only once in the program.

The 'with' keyword

So far, we discussed different flags that could be used to open files in different modes. The examples we discussed followed a common pattern – Open the file, perform read/write operations and close the file. There is an elegant way of interacting with files using the with keyword. If there are any errors during the execution of the code block that interacts with a file, the with keyword ensures that the file is closed and the associated resources are cleaned up on exiting the code block. As always, let's review the with keyword with an example.

```
if __name__ == "__main__":
  with open('write_file.txt', 'r+') as file:
      # read the contents of the file and print to the screen
      print(file.read())
      file.write("This is a line appended to the file")

      #rewind the file and read its contents
      file.seek(0)
      print(file.read())
  # the file is automatically closed at this point
  print("Exited the with keyword code block")
```

In the above example (with_keyword_example), we skipped closing the file as the with keyword takes care of closing the file once the execution of the indented code block is complete. The with keyword also takes care of closing the file while leaving the code block due to an error. This ensures that the resources are cleaned up properly in any scenario. Going forward, we will be using the with keyword for file i/o.

configparser

Let's discuss some aspects of Python programming that is especially helpful while developing applications using the Raspberry Pi. One such tool is the configparser available in Python. The configparser module (https://docs.python.org/3.4/library/configparser.html) is used to read/write config files for applications.

In software development, config files are generally used to store constants such as access credentials, device id, and so on In the context of a Raspberry Pi, configparser could be used to store the list of all GPIO pins in use, addresses of sensors interfaced via the I^2C interface, and so on. Let's discuss three examples where we learn making use of the configparser module. In the first example we will create a config file using the configparser. In the second example, we will make use of the configparser to read the config values and in the third example, we will discuss modifying config files in the final example.

Example 1

In the first example, let's create a config file that stores information including device id, GPIO pins in use, sensor interface address, debug switch and access credentials.

```
import configparser

if __name__ == "__main__":
  # initialize ConfigParser
  config_parser = configparser.ConfigParser()
```

```
# Let's create a config file
with open('raspi.cfg', 'w') as config_file:
    #Let's add a section called ApplicationInfo
    config_parser.add_section('AppInfo')

    #let's add config information under this section
    config_parser.set('AppInfo', 'id', '123')
    config_parser.set('AppInfo', 'gpio', '2')
    config_parser.set('AppInfo', 'debug_switch', 'True')
    config_parser.set('AppInfo', 'sensor_address', '0x62')

    #Let's add another section for credentials
    config_parser.add_section('Credentials')
    config_parser.set('Credentials', 'token', 'abcxyz123')
    config_parser.write(config_file)
print("Config File Creation Complete")
```

Let's discuss the above code example (available for download along with this chapter as config_parser_write.py) in detail:

1. The first step is importing the configparser module and creating an instance of the ConfigParser class. This instance is going to be called config_parser:

   ```
   config_parser = configparser.ConfigParser()
   ```

2. Now, we open a config file called raspi.cfg using the with keyword. Since the file doesn't exist, a new config file is created.
3. The config file is going to consist of two sections namely AppInfo and Credentials.
4. The two sections could be created using the method add_section as follows:

   ```
   config_parser.add_section('AppInfo')
   config_parser.add_section('Credentials')
   ```

5. Each section is going to consist of different set of constants. Each constant could be added to the relevant section using the set method. The required arguments to the set method include the section name (under which the parameter/constant is going to be located), the name of the parameter/constant and its corresponding value. For example: The parameter id can be added to the section AppInfo and assigned a value of 123 as follows:

   ```
   config_parser.set('AppInfo', 'id', '123')
   ```

6. The final step is saving these config values to the file. This is accomplished using the config_parser method, write. The file is closed once the program exits the indented block under the with keyword.

```
config_parser.write(config_file)
```

Note: We strongly recommend trying the code snippets yourself and use these snippets as a reference. You will learn a lot by making mistakes and possibly arrive with a better solution than the one discussed here.

When the above code snippet is executed, a config file called raspi.cfg is created. The contents of the config file would include the contents shown below:

```
[AppInfo]
id = 123
gpio = 2
debug_switch = True
sensor_address = 0x62

[Credentials]
token = abcxyz123
```

Example 2

- Let's discuss an example where we read config parameters from a config file created in the previous example:

Note: As long as the config files are created in the format shown above, the ConfigParser class should be able to parse it. It is absolutely not necessary to create config files using a python program. We just wanted to show programmatic creation of config files as it is easier to programmatically create config files for multiple devices at the same time.

```
import configparser

if __name__ == "__main__":
  # initialize ConfigParser
  config_parser = configparser.ConfigParser()

  # Let's read the config file
  config_parser.read('raspi.cfg')

  # Read config variables
  device_id = config_parser.get('AppInfo', 'id')
  debug_switch = config_parser.get('AppInfo', 'debug_switch')
  sensor_address = config_parser.get('AppInfo',
  'sensor_address')

 # execute the code if the debug switch is true
 if debug_switch == "True":
```

```
print("The device id is " + device_id)
print("The sensor_address is " + sensor_address)
```

The above example is available for download along with this chapter (config_parser_read.py). Let's discuss how this code sample works:

1. The first step is initializing an instance of the ConfigParser class called config_parser.
2. The second step is loading and reading the config file using the instance method read.
3. Since we know the structure of the config file, let's go ahead and read some constants available under the section AppInfo. The config file parameters can be read using the method get. The required arguments include the section under which the config parameter is located and the name of the parameter. For example: The config parameter id is located under the section AppInf'. Hence, the required arguments to the method include AppInfo and id.

   ```
   device_id = config_parser.get('AppInfo', 'id')
   ```

4. Now that the config parameters are read into variables, let's make use of it in our program. For example: Let's test if the variable debug_switch (a switch to determine if the program is in debug mode) and print the other config parameters that were retrieved from the file.

   ```
   if debug_switch == "True":
   print("The device id is " + device_id)
     print("The sensor_address is " + sensor_address)
   ```

Example 3

Let's discuss an example where we would like to modify an existing config file. This is especially useful in situations where we need to update the firmware version number in the config file after performing a firmware update.

The code snippet shown below is available for download as config_parser_modify.pyalong with this chapter.

```
import configparser

if __name__ == "__main__":
  # initialize ConfigParser
  config_parser = configparser.ConfigParser()

  # Let's read the config file
```

```
config_parser.read('raspi.cfg')

# Set firmware version
config_parser.set('AppInfo', 'fw_version', 'A3')

# write the updated config to the config file
with open('raspi.cfg', 'w') as config_file:
config_parser.write(config_file)
```

Let's discuss how this works:

1. As always, the first step is initializing an instance of the ConfigParser class. The config file is loaded using the method read.

```
# initialize ConfigParser
config_parser = configparser.ConfigParser()
# Let's read the config file
config_parser.read('raspi.cfg')
```

2. The required parameter is updated using the set method (discussed in a previous example).

```
# Set firmware version
config_parser.set('AppInfo', 'fw_version', 'A3')
```

3. The updated config is saved to the config file using the write method.

```
with open('raspi.cfg', 'w') as config_file:
  config_parser.write(config_file)
```

Challenge to the reader

Using example 3 as a reference, update the config parameter debug_switch to the value False. Repeat example 2 and see what happens.

Reading/Writing to CSV files

In this section, we are going to discuss reading/writing to csv files. This module (https://docs.python.org/3.4/library/csv.html) is useful in datalogging applications. Since we will be discussing datalogging in the next chapter, let's review reading/writing to csv files.

Writing to CSV files

Let's consider a scenario where we are reading data from different sensors. This data needs to be recorded to a csv file where each column corresponds to a reading from a specific sensor. We are going to discuss an example where we record the value 123, 456 and 789 in the first row of the csv file and the second row is going to consist of values including Red, Green and Blue.

1. The first step in writing to a csv file is opening a csv file using the with keyword.

```
with open("csv_example.csv", 'w') as csv_file:
```

2. The next step is initializing an instance of the writer class of the csv module:

```
    csv_writer = csv.writer(csv_file)
```

3. Now, each row is added to the file by creating a list that contains all the elements that need to be added to a row. For example: The first row can be added to the list as follows:

```
    csv_writer.writerow([123, 456, 789])
```

4. Putting it altogether, we have:

```
import csv
if __name__ == "__main__":
  # initialize csv writer
  with open("csv_example.csv", 'w') as csv_file:
      csv_writer = csv.writer(csv_file)

      csv_writer.writerow([123, 456, 789])
      csv_writer.writerow(["Red", "Green", "Blue"])
```

5. When the above code snippet is executed (available for download as csv_write.py along with this chapter), a csv file is created in the local directory with the following contents:

```
123,456,789
Red,Green,Blue
```

Reading from CSV files

Let's discuss an example where we read the contents of the csv file created in the previous section:

1. The first step in reading a csv file is opening it in read mode.

```
with open("csv_example.csv", 'r') as csv_file:
```

2. Next, we initialize an instance of the reader class from the csv module. The contents of the csv file are loaded into the object csv_reader.

```
csv_reader = csv.reader(csv_file)
```

3. Now that the contents of the csv file are loaded, each row of the csv file could be retrieved as follows:

```
for row in csv_reader:
  print(row)
```

4. Putting it altogether:

```
import csv

if __name__ == "__main__":
  # initialize csv writer
  with open("csv_example.csv", 'r') as csv_file:
      csv_reader = csv.reader(csv_file)

      for row in csv_reader:
          print(row)
```

5. When the above code snippet is executed (available for download along with this chapter as csv_read.py), the contents of the file are printed row-by-row where each row is a list that contains the comma separated values.

```
['123', '456', '789']
['Red', 'Green', 'Blue']
```

Python utilities

Python comes with a number of utilities that enables interacting with other files and the operating system itself. We have identified all those python utilities that we have used in our past projects. Let's discuss the different modules and their uses as we might use them in the final project of this book:

The os module

As the name suggests, this module (https://docs.python.org/3.1/library/os.html) enables interacting with the operating system. Let's discuss some of its applications with examples:

Checking a file's existence

The os module could be used to check if a file exists in a specific directory. For example: We extensively made use of the file write_file.txt. Before opening this file to read or write, we could check the file's existence.

```
import os

if __name__ == "__main__":
    # Check if file exists
    if os.path.isfile('/home/pi/Desktop/code_samples/write_file.txt'):
        print('The file exists!')
    else:
        print('The file does not exist!')
```

In the above code snippet, we make use of the function isfile(), available with the module os.path. When a file's location is passed an argument to the function, it returns True if the file exists at that location. In this example, since the file write_file.txt exists in the code examples directory, the function returns True. Hence the message, **The file exists** is printed to the screen.

```
if os.path.isfile('/home/pi/Desktop/code_samples/write_file.txt'):
    print('The file exists!')
else:
    print('The file does not exist!')
```

Checking for a folder's existence

Similar to os.path.isfile(), there is another function called os.path.isdir(). It returns True if a folder exists at a specific location. We have been reviewing all code samples from a folder called code_samples located on the Raspberry Pi's Desktop. It's existence could be confirmed as follows:

```
# Confirm code_samples' existence
if os.path.isdir('/home/pi/Desktop/code_samples'):
    print('The directory exists!')
else:
    print('The directory does not exist!')
```

Delete files

The os module also enables deleting files using the remove() function. Any file that is passed as an argument to the function is deleted. In the file i/o section, we discussed reading from files using the text file, read_file.txt. Let's delete the file by passing it as an

argument to the remove() function.

```
os.remove('/home/pi/Desktop/code_samples/read_file.txt')
```

Kill a process

It is possible to kill an application running on the Raspberry Pi by passing process pid to the kill() function. In the previous chapter, we discussed the light_scheduler example that runs as a background process on the Raspberry Pi. In order to demonstrate killing a process, we are going to attempt killing that process. We need to determine the process pid of the light_scheduler process (you may pick an application that was started by you as a user and not do not touch root processes). The process pid could be retrieved from the command line terminal using the following command:

```
ps aux
```

It spits out the processes currently running on the Raspberry Pi (shown in the figure below). The process pid for the light_scheduler application is 1815:

```
pi         822  0.0  1.1   6916  5000 pts/0    Ss   Jul10   0:02 -bash
root      1548  0.0  0.0      0     0 ?        S    Jul10   0:00 [kworker/u2:1]
pi        1815  0.1  1.9  12636  8804 pts/0    S+   Jul10   0:01 python3 light_scheduler.py
root      1817  0.0  1.1  12064  5280 ?        Ss   Jul10   0:00 sshd: pi [priv]
pi        1827  0.0  0.7  12064  3504 ?        S    Jul10   0:00 sshd: pi@pts/1
pi        1830  0.0  1.0   6320  4476 pts/1    Ss   Jul10   0:00 -bash
```

light_scheduler daemon's PID.

Assuming we know the process pid of the application that needs to be killed, let's review killing the function using kill(). The arguments required to kill the function include the process pid and the signal (signal.SIGKILL) that needs to be sent to the process to kill the application:

```
import os
import signal

if __name__ == "__main__":
    #kill the application
    try:
        os.kill(1815, signal.SIGKILL)
    except OSError as error:
        print("OS Error " + str(error))
```

The signal module (https://docs.python.org/3/library/signal.html) contains the constants that

represents the signals that could be used to stop an application. In this code snippet, we make use of the SIGKILL signal. Try running the ps command (ps aux) and you will notice that the light_scheduler application has been killed.

Monitoring a process

In the previous example, we discussed killing an application using the kill() function. You might have noticed that we made use of something called the try/except keywords to attempt killing the application. We will discuss these keywords in detail in the next chapter.

It is also possible to monitor whether an application is running using the kill() function using the try/except keywords. We will discuss monitoring processes using the kill() function after introducing the concept of trapping exceptions using try/except keywords.

All examples discussed in the os module is available for download along with this chapter as os_utils.py

The glob module

The glob module (https://docs.python.org/3/library/glob.html) enables identifying files of a specific extension or files that have a specific pattern. For example, it is possible to list all python files in a folder as follows:

```
# List all files
for file in glob.glob('*.py'):
   print(file)
```

The glob() function returns a list of files that contains the .py extension. A for loop is used to iterate through the list and print each file. When the above code snippet is executed, the output contains the list of all code samples belonging to this chapter (output truncated for representation):

```
read_from_file.py
config_parser_read.py
append_to_file.py
read_line_from_file.py
config_parser_modify.py
python_utils.py
config_parser_write.py
csv_write.py
```

This module is especially helpful with listing files that have a specific pattern. For example: Let's consider a scenario where you would like to upload files that were created from different trials of an experiment. You are only interested in files that are of the following format: file1xx.txt where x stands for any digit between and 9. Those files could be sorted

and listed as follows:

```
# List all files of the format 1xx.txt
for file in glob.glob('txt_files/file1[0-9][0-9].txt'):
   print(file)
```

In the above example, [0-9] means that the file name could contain any digit between 0 and 9. Since we are looking for files of the format file1xx.txt, the search pattern that is passed an argument to the glob() function is file1[0-9][0-9].txt.

When the above code snippet is executed, the output contains all text files of the specified format:

```
txt_files/file126.txt
txt_files/file125.txt
txt_files/file124.txt
txt_files/file123.txt
txt_files/file127.txtz
```

We came across this article that explains the use of expressions for sorting files: http://www.linuxjournal.com/content/bash-extended-globbing. The same concept can be extended to searching for files using the glob module.

Challenge to the reader

The examples discussed with the glob module are available for download along with this chapter as glob_example.py. In one of the examples, we discussed listing files of a specific format. How would you go about listing files that are of the following format: filexxxx.*? (Here x represents any number between and 9. * represents any file extension.)

The shutil module

The shutil module (https://docs.python.org/3/library/shutil.html) enables moving and copying files between folders using the methods move() and copy(). In the previous section, we listed all text files within the folder, txt_files. Let's move these files to the current directory (where the code is being executed) using move(), make a copy of these files once again in txt_files and finally remove the text files from the current directory.

```
import glob
import shutil
import os
if __name__ == "__main__":
   # move files to the current directory
   for file in glob.glob('txt_files/file1[0-9][0-9].txt'):
      shutil.move(file, '.')
   # make a copy of files in the folder 'txt_files' and delete them
```

```
for file in glob.glob('file1[0-9][0-9].txt'):
    shutil.copy(file, 'txt_files')
    os.remove(file)
```

In the above example (available for download along with this chapter as shutil_example.py), the files are being moved as well as copied from the origin to the destination by specifying the source and the destination as the first and second arguments respectively.

The files to be moved (or copied) are identified using the glob module. Then, each file is moved or copied using their corresponding methods.

The subproc module

We briefly discussed this module in the previous chapter. The subproc module (https://docs.python.org/3.2/library/subprocess.html) enables launching another program from within a python program. One of the commonly used functions from the subproc module is Popen. Any process that needs to be launched from within the program needs to be passed as a list argument to the Popen function:

```
import subprocess
if __name__ == "__main__":
    subprocess.Popen(['aplay', 'tone.wav'])
```

In the above example, tone.wav (wave file that needs to be played) and the command that needs to be run are passed as a list argument to the function. There are several other commands from the subprocess module that serve a similar purpose. We leave it to your exploration.

The sys module

The sys module (https://docs.python.org/3/library/sys.html) allows interacting with the Python run-time interpreter. One of the functions of the sys module is parsing command line arguments provided as inputs to the program. Let's write a program that reads and prints the contents of the file that is passed as an argument to the program.

```
import sys
if __name__ == "__main__":
    with open(sys.argv[1], 'r') as read_file:
        print(read_file.read())
```

Try running the above example as follows:

```
python3 sys_example.py read_lines.txt
```

The above example is available for download along with this chapter as sys_example.py. The list of command line arguments passed while running the program are available as a list

argv in the sys module. argv[0] is usually the name of the python program and argv[1] is usually the first argument passed to the function.

When sys_example.py is executed with read_lines.txt as an argument, the program should print the contents of the text file:

```
I am learning Python Programming using the Raspberry Pi Zero.
This is the second line.
Line 3.
Line 4.
Line 5.
Line 6.
Line 7.
```

Summary

In this chapter, we discussed file i/o – reading and writing to files, different flags used to read, write and append to files. We talked about moving file pointers to different points in a file to retrieve specific content or overwrite the contents of a file at a specific location. We discussed the ConfigParser module in Python and its application in storing/retrieving config parameters for applications along with reading and writing to csv files.

Finally, we discussed different Python utilities that have a potential use in our project. We will be extensively making use of file i/o and the discussed python utilities in our final project. We strongly recommend familiarizing with the concepts discussed in this chapter before moving onto the final projects discussed in this book.

In the upcoming chapters, we will discuss uploading sensor data stored in csv files to the cloud and logging errors encountered during the execution of an application. See you in the next chapter!