

PYTHON PROGRAMMING

**THE ULTIMATE GUIDE TO LEARNING PYTHON
PROGRAMMING LANGUAGE IN 24 HOURS!**

**PRACTICAL PYTHON PROGRAMMING FOR BEGINNERS,
PYTHON COMMANDS, PYTHON LANGUAGE AND MUCH
MORE.**

Book Description

‘Python Programming Language in 24 hours!’ is the ultimate resource for newbies looking to learn a programming language that is easy to grasp in a very short time compared to other programming languages such as C++, Java and Pascal.

This book is promising to get you on your way to becoming a master of Python in no time. It endows you with the powers of a programmer with a solid foundation of the basics that you can use to create almost anything you want using Python in 24 hours or less.

The book has clear and straight forward examples that you can just copy and paste with explanations that are short, practical and easy to understand.

The book will teach you:

- How to install the latest Python 3 interpreter and using shell and the editor to create Python programs

- How to format and print information to the screen using Python commands

- How to define variables and perform operations such as arithmetic

- How to make your programs interactive by accepting user input and also receiving input from external files such as text files

- How to enable your programs to make decisions using conditional statements

- How to make your programs repeat a set of operations until a conditional is met through loops

- Functions, modules plus much more

Table of Content

[Introduction](#)

[Chapter 1: Introduction to the Python Language](#)

[Chapter 2 - Installation and First Program](#)

[Chapter 3 - Writing Our First Program](#)

[Chapter 4 - Data Storage and Manipulation](#)

[Chapter 5 - Basic Data Types](#)

[Chapter 6 - User Input and Output](#)

[Chapter 7 - Conditional Statements](#)

[Chapter 8 - Looping](#)

[Chapter 9 - Functions](#)

[Chapter 10 - Modules](#)

[Chapter 11 - Creating a Module](#)

[Chapter 12 - Working With Files in Python](#)

[Conclusion](#)

Introduction

It is always a good idea to learn how to program. It gives you the freedom to do whatever you want on your computer. Using it for the right purposes can allow you to save time by boosting your productivity efficiently and effectively. This will allow you to reach all new levels and heights while putting you ahead of anyone in your field.

Plus, once you learn to program, it is very fun.

Some programming languages are easy to learn and some not so much. The challenge is picking the best one that does exactly what you have in mind and can get you going as quickly as possible. Python is the language.

This book aims at getting you acquainted with the basics of Python in 24 hours or less while giving you a deep understanding of the concepts, and offering practical examples to get you going as quickly as possible while steering clear of any advanced topics.

By the end of this book you will be able to create Python programs that print information out to the user (there will be a lot of this in this book), tell Python how to store data, get input from the user then perform operations on that input such as making decisions and doing calculations, and also get information from external files such as text files and do more operation on that as well.

All this will be done in this book that can be read within 24 hours. All examples can be typed out. So there is no need to download any source files.

But before we get into all of that let us look at what Python is and what it can do for you after you have learned the basics.

Chapter 1: Introduction to the Python Language

Python is a general purpose, high-level programming language that was created in the late 1990s by Guido Van Rossum. It has a design philosophy that emphasizes on code readability and allowed programmers to easily express concepts very quickly and in fewer lines of codes than they would be able to in other programming languages, such as C++, Java and Pascal.

It was also designed to reduce development time by having fewer syntactical restrictions compared to other programming languages. This also helps in reducing many errors in programming that newbies make.

To reiterate some of the points mentioned in this chapter in a focused way (while adding more). Python is:

Easy to learn

This is why it is called the programming language for beginners. If you have a concept that you need to roll out very quickly then Python is for you because less code is required compared to other programming languages.

Powerful

It has high level functions, a huge standard library, and plenty of other easily available libraries and functions that can let you create anything you want while giving you productivity boost with a dynamic type system and automatic memory management for efficiency.

A scripting language

This is because it uses an interpreter to translate its code into machine code. As such, you don't need to compile your entire code before you execute it. The translator will read and execute each line of code as it comes across.

Object Oriented

It allows you to model real world objects using code and it also allows you to give those objects attributes and abilities.

Cross Platform

It has interpreters available for every major operating system. This means that any program you write in Windows will run the same on Mac OS X and Linux. You can even package the code into stand-alone packages using Py2exe or Pyinstaller for use on any operating system without needing to install an interpreter.

What Python Can Do For You

Like I mentioned before, Python can allow you to create anything and with third party resources that extend the capabilities of the language, the options become limitless. It is all up to the programmer and his/her imagination.

There are many ways to implement Python in many fields that it makes learning Python a no-brainer.

Python can be used for:

- System utility applications
- Graphical User Interfaces (GUIs)
- Internet scripting (for websites like Facebook, Twitter, Instagram etc.)
- Embedded scripting
- Database programming
- Game programming
- Mobile applications
- Artificial intelligence
- Image processing and many more

Now that you can't wait to start using Python, let's install the interpreter and write our first program so that you get properly initialized into the world of Python programming.

Chapter 2 - Installation and First Program

By the end of this chapter we will learn how to:

- Download and Install the Python 3 Interpreter
- Shell and IDLE
- Write Your First Python Program
- Comments
- A Bit about the Print Function

Programming gives you the ability to do things you could not do before with your computer. Here is where you start getting your abilities to do really useful things with Python.

We are going to talk about where to get python and installing it. After, we will get a taste of the action by creating a little program. Finally we will discuss writing comments in your code and a little bit about the print function (which you will use a lot throughout this book).

With all that said, let us get to it.

Downloading and Installing the Python 3 Interpreter

To be able to type Python commands on your computer and create programs, you need a Python interpreter. But what is an interpreter?

An **interpreter** is a computer program that translates human readable code into machine code that the computer can understand and execute (1s and 0s).

To get the latest version of the Python 3 interpreter, head over to

<https://www.python.org/downloads/>

Click on the **Download Python 3.5.1** button to download it. The latest version as of writing this book should be Python 3.5.1 as the picture below shows:



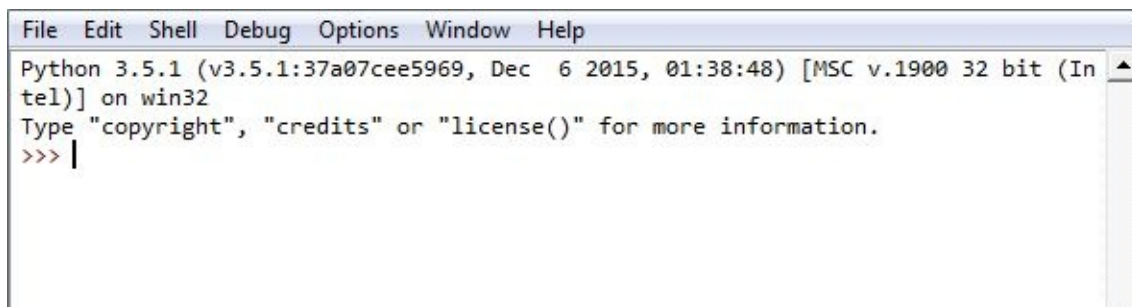
After the interpreter is downloaded, install it. Now you are good to go.

Python IDLE and Shell

Now that the interpreter is downloaded and installed, it is time to launch **IDLE** . This is the program that we will use to write all our Python codes and it comes bundled in with the interpreter. It is intended to be an easy-to-use **Integrated Development Environment (IDE)** that is ideal for learners.

Head over to the directory where you installed Python (or search for it on the start menu if you are using windows) and click the executable file called **IDLE (Python 3.5 32-bit)**.

This will open the Python Shell which looks something like this:

A screenshot of the Python IDLE Shell window. The window has a menu bar with 'File', 'Edit', 'Shell', 'Debug', 'Options', 'Window', and 'Help'. The main text area displays the following text: 'Python 3.5.1 (v3.5.1:37a07cee5969, Dec 6 2015, 01:38:48) [MSC v.1900 32 bit (Intel)] on win32', 'Type "copyright", "credits" or "license()" for more information.', and a prompt '>>> |' with a cursor. There is a small upward arrow icon in the top right corner of the text area.

```
File Edit Shell Debug Options Window Help
Python 3.5.1 (v3.5.1:37a07cee5969, Dec 6 2015, 01:38:48) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> |
```

The shell program is always in interactive mode. This means that you can enter a command and instantly see the results. After that, it will go to the next line and wait for the next command.

Now we are ready to start typing in some Python commands. Think of this as the conduit to access the new abilities you are about to gain.

Below we have the following simple Python commands you must type in the shell program one by one. Type them after the `>>>` which signifies input and hit Enter after each command (Do not worry, these commands will not tell your computer to self-destruct; we just want to make sure everything is working):

```
>>> 5 + 2
```

```
7
```

```
>>> 4 < 1
```

```
False
```

```
>>> print('Hello Word')
```

```
Hello Word
```

What is happening here? The first command is adding 5 and 2 and displaying 7 to us as the result. The second command is checking if 4 is less than 1 and the result is `False` because 4 is greater than 1. The third and final command just tells the shell to display the words *"Hello World"*.

You cannot write an entire program in shell because it does not save your code . Shell is useful for testing our commands to see how they work and instantly see the results, but that is just about it.

In the next section, we will look at how we can write out first program using the editor instead of just typing commands for shell to execute. This should be fun.

Chapter 3 - Writing Our First Program

While **Shell** is still open, headover to the top menu and click **File > New File** (or hit **Ctrl + N** on your keyboard) to bring up the editor we are going to use to write our awesome programs.

Any code written in the editor can be saved into a **py** or **.pywfile** .

Now type in the following code into the editor:

```
#print "Hello World" to the screen  
print('Hello World')
```

Now go back to the top menu and click **File > Save** (or hit **Ctrl + S** on the keyboard) and the "Save As" - dialog box will appear. Give the file a name like **helloworld.py** (do not forget to enter the file extension) and click **Save** .

Congratulation! You have created your first python program.

To run it, head to the top menu and click **File > Run > Run Module** (or hit **F5** on the keyboard) to run the program. This will bring up the shell program again with the output bellow:

Hello World

Comments

When you specify a certain piece of a code as a comment, the interpreter will ignore it.

Comments serve as a way of documenting your thoughts about what you were thinking when you wrote a particular piece of code. You do not want to go months without looking at your code only to look at it one day and start trying to figure it all out again.

They are also important if you are working with a team. Comments serve as little messages to the people maintaining your code. So, *do* comment.

Here is how you write single line comments:

```
#the interpreter will ignore this line
```

```
#the interpreter will ignore this too
```

```
#another line for the interpreter to ignore
```

```
#print ('this will not print because it is commented out')
```

Everything after the # (**hash**) will be ignored even if it is a line of code.

Single line comments are great, but you can also write multiline comments if you want to get very specific by placing them between triple quotes like these **“”** .

Here is how you do it:

”

**The interpreter will ignore
all of this as well. So
go crazy and comment your code**

”

If you noticed in our first program, we used a comment just before we printed Hello World.

A Bit about the Print Function

We have already seen the print function in action in our first program. But what is it?

The `print()` function gives you the ability to provide the user with feedback by outputting information to the screen. It can print pretty much any value from the basic data types (which we will look at in the next chapter).

Just place whatever you want to print out between the parentheses.

Here is an example:

```
print('Hi, my name is Richard')
```

You can even perform operations inside the print function:

```
print(5 + 2)
```

In the above example, 5 and 2 will be added together and 7 will be printed out to the screen. This is because it is not incased in quotes. But if it was, it would have printed '5 + 2' instead of the answer. You get why this is so in the next chapter. Don't worry about it now.

You can print a combination of text and numbers:

```
print('5 + 2 =', 5 + 2)
```

If you enter the above code into shell and hit enter, you will get the result below:

```
5 + 2 = 7
```

This is because the , (comma) is used to link the values you want to print out to the screen and it automatically adds a space between them.

Now that we have our environment installed and we know how to use Shell and IDLE to test and write our programs, let us look into how Python stores data in memory by looking into variables and some operations you can do in order to manipulate them.

Chapter 4 - Data Storage and Manipulation

Here is what we will cover by the end of this chapter:

- Variables
- Naming Conventions
- A Bit about Operators

You have an ability to tell your program to store and manipulate data to do really amazing things. This will not only enhance your programs, but it will also make them work properly.

We use variables and operators to store and manipulate data.

Variables

A variable is basically a container or place holder that you use to store and manipulate values. In Python, you can pretty much store anything.

To define a variable, you have to give it a name then assign a value to it. The syntax to define a variable is as follows:

```
variable_name = initial_value
```

The = sign in this example is not the mathematical equals sign, but an assignment operator which takes what is on the left hand side and assigns it on the right hand side.

Let us look at an example of how variables work. Open Shell and type in the following code, then press enter:

```
name = 'Richard'
```

Now that you have defined the variable, the computer will store that variable for you by allocating space for it in memory.

You can then refer to the variable by name and print it out. For example, if you enter `print(name)` in Shell and hit enter, you will get the following output:

Richard

You can also define more than one variable at once:

```
name, age = 'Richard', 25
```

In that example, the value 'Richard' will be assigned to the variable `name` and the value 25 will be assigned to the variable `age`.

Here a few things to remember about variables:

- Variable names must start with a letter and then you can use a combination of underscores, letters and numbers

- They cannot have the name of a keyword as a variable name because these have special meaning within Python and are reserved words. Some keywords are `class`, `while`, `global`, `break` and `not`. You can't use any of these

- Python is case sensitive. This means variable names cannot be used interchangeably even if they have the same meaning. A variable called `Name` is different from the variable `name`

Naming Conventions

Python programmers usually use two kinds of naming conventions. You can use either, just be sure to be consistent and stick to it for good programming practices.

You can do what is known as **camel-casing**. For example, you can name your variables this way:

randomNumber or playerName

Alternatively you can separate words with underscores like this:

random_number or player_name

I will stick to separating the variables using underscores. I think this improves readability. But you are free to use whatever naming convention you like.

A Bit about Operators

In Python, operators are what allow you to manipulate your data. The data they manipulate are called operands.

If we look at the expression $4 + 3 = 7$, the numbers **4** and **7** are the operands while the **+** sign is the operator.

Here are the most common operators that you will work with in your python programs:

Arithmetic Operators

Assignment Operators

Comparison Operators

Logical Operators

There are more operators than these but this will suffice for starters. In this chapter, we will look at only arithmetic and assignment operators. The other two will be covered when we look at conditional statements in next chapters.

Arithmetic Operators

Arithmetic operators are used to perform your basic mathematical operations like the ones you learned in school. Let us take a look at them one by one.

The + (Addition) operator will take the values of the operands on either side and add them together:

$$4 + 3 = 7$$

The - (Subtraction) operator subtracts the value of the left operand from the value of the right operand:

$$4 - 3 = 2$$

The * (Multiplication) operator will take values of the operands on either side and multiply them.

$$4 * 3 = 12$$

The / (Division) operator takes the value of the left operand and divides it by the value of the right operand:

$$4 / 3 = 1.333333333$$

The % (Modulus) operator divides the value of the left operand by the value of the right operand and then returns the remainder:

$$4 \% 3 = 1$$

The ** (Exponent) operator is used to perform exponential calculations:

$$4^{**}3 = 64$$

The // (Floor division) operator performs a division then throws away the remainder:

$$4 // 3 = 1$$

The Order of Operations in Python

The order of operations matters when performing arithmetic operations in Python. The order of operations is parenthesis, exponents, multiplication, division, addition and subtraction or PEDMAS, in other words.

This means that any operations in parenthesis will be performed first then any division or multiplication will be performed next and finally any addition or subtraction.

Let us look at an example of this in action so it makes a little more sense:

```
6 * (5 + 3) / 3 ** 2
```

Based on PEDMAS, the operations in parenthesis (5 + 3) will be performed first and then the exponential calculation 3 ** 2 will be performed next then followed by the multiplication and finally, the division. If you open Shell and type in that code, you will get the following output:

```
5.333333333333333
```


Assignment Operators

We already saw one of the assignment operators in action when we talked about variables. Let us take a look at a few of these.

The **+= (Add AND)** operator adds the values of the left and right operand together then assigns the resulting value to the left operand:

$x += y$ is the same as $x = x + y$

The **/= (Divide AND)** operator takes the value of the left operand and divides it by the value of the right operand then assigns the resulting value to the left operand:

$x /= y$ is the same as $x = x / y$

One more assignment operand we will look at is the **%= (Modulus AND)** operator. This divides the value of the right operand by the value of the left operand then assigns the remainder to the left operand:

$x \% = y$ is the same as $x = x \% y$

And the rest of the assignment operators work in pretty much the same way, such as **-= (Subtract AND)**, ***= (Multiply AND)**, ****= (Exponent AND)** and **//= (Floor Division)**.

In the next chapter we will look at some of the basic data types that your variables can hold in Python. We will look at the numeric, string and list types and how you can manipulate them to print out information to the user in very interesting ways.

Chapter 5 - Basic Data Types

Here is what we will cover by the end of this chapter:

- Numbers
- Strings
- Lists

It is always a good idea to know what you will be working with. Especially the type of values your variables contain and we will focus on that in this chapter.

We will be looking at numbers, strings, and lists. We will learn how to work with them to produce useful results.

This is essential knowledge that can help you hone your new abilities as a programmer. Plus you gain new abilities too. So much winning is happening here.

Numbers

Numbers are numeric data types used to store numeric values.

Integer

An **Integer** (or **int**) is a data type that stores positive or negative numeric values with no decimal parts (whole numbers) or quotation marks e.g. 1, 7, 19, 200 and so on.

You define an integer the same way you define any variable:

```
variable_name = initial_value
```

For example:

```
age = 28, number_of_students = 2000
```

Float

A float is a data type that stores positive or negative numeric values with decimal parts (real numbers) or quotation marks e.g. 7.5, 9.0454, 345.553, the number PI (3.14159285) and so on.

You define a float the same way you define any variable:

variable_name = initial_value

For example:

weight = 55.8, average_grade = 75.25, pi = 3.14159285

Basic Numeric Functions

We will discuss some of the functions you can use with numeric data types just to give a taste of what Python can do.

The `abs(x)` function takes in a numeric value `x` and returns its absolute value (positive distance between zero and the number). For example, `abs(-3)` returns 3 .

The `cmp(x, y)` function takes in two numeric values `x` and `y` then returns -1 if `x` is less than `y` , 1 if `x` is equal to `y` and 0 if `x` is greater than `y`. For example, `cmp(40, 100)` will return -1 because 40 is less than 100.

The `pow(x, y)` function takes in two numeric values `x` and `y` and returns the value of `x` to the power `y` or `x ** y` . For example, `pow(3, 5)` returns 243 .

There are many more functions. And I encourage you to explore.

Strings

A string is any value in quotation marks is a string e.g. **“Derek”**, **‘Richard’** , **‘Do not forget to buy some milk’** and so on

Defining a String

You define a string the same way you define any variable except that the initial value must be in quotation marks:

variable_name = ‘intitial_value’ or variable name = “initial_value”

For example:

message = ‘Hi, my name is Richard’ or message = “Hi, my name is Richard”

Connecting Strings

Just like you add two numbers together, you can also add two strings together using + operator. In this scenario is not called the addition operator anymore, but the string concatenation operator.

Here is an example of string concatenation in action. Open up the editor and place the following code in there:

```
name = "Hi, my name is Richard."
```

```
favorite_fruit = " I like bananas."
```

```
#using the + operator to connect two strings
```

```
message = name + favorite_fruit
```

What is happening here is that the variables **name** and **favorite_fruit** are being combined together using the + sign and the resulting string is being assigned to the variable `message`.

Now if you print the variable `message` with `print(message)` then **save and run the program** , you will get the following output:

```
Hi, my name is Richard. I like bananas.
```

As you can see, the two strings have been combined.

Formatting Strings Using Special Characters

Formatting a string is easy. Just use the %s inside your string where you want the string values and variables to appear. This special character acts like a place holder.

After that, type % then () (parentheses) with a comma separated list of values or variables in the order you want them to appear inside those parentheses.

An example can better illustrate what I just said. Open up the editor and place the following code in there:

```
name = 'Richard'

#formatting a string using the %s operator as place holder
message = 'Hi, my name is %s and i like %s.' % (name, 'bananas')

print(message)
```

In the above example, the first %s is replace with the value of the variable name and the second one is replaced with the string 'bananas'.

If you save that and run it, you will get the following output:

```
Hi, my name is Richard and i like bananas.
```

Use %d as a place holder for integers and %.4.2f as a place holder for floats. The 4 in %.4.2f represents the total length of the float and the 2 represents the number of decimal places of the float.

Lists

Lists allow you to create a collection of interrelated values and manipulate them. It can have a combination of any data types you want in there.

Defining a List

Defining a list is very easy in python. So let us do it. Open up the editor and place the following code in there:

```
fruits = ['bananas', 'oranges', 'mangoes', 'apples']
```

As you can see, you first type in the variable name followed by the assignment operator followed by [] (square brackets) with a comma separated list of values you want the list to contain.

To print out the list just add the following code in the editor:

```
print(fruits)
```

If you save and run the program, you should get the following output:

```
['bananas', 'oranges', 'mangoes', 'apples']
```

Each value has an index with the first value having an index of zero. This means 'bananas' will have an index of zero and 'oranges' will have an index of one and so on.

Accessing the values of a List

Accessing the values in a list is also very easy. You just type in the name of the list with square bracket at the end and the index of the item we want to access inside of them.

Let us try and accessing the values 'bananas' and 'apples' and store them in the variables named `bananas` and `apples` then printing them out to screen.

Add the following code in the editor:

```
bananas = fruits[0]
```

```
apples = fruits[3]
```

```
print(bananas)
```

```
print(apples)
```

If you save and run the program, you will get the following output:

```
bananas
```

```
apples
```

Changing a Value in a List

Let us say we do not like mangoes and would like to change the value. We can just assign a new value to index where 'mangoes' is. Let us say we want papayas instead.

Add the following code into the editor:

```
fruits[2] = 'papayas'
```

If you print the list now, you should get the following output:

```
['bananas', 'oranges', 'papayas', 'apples']
```

As you can see, 'mangoes' have been changed to 'papayas'.

List Manipulation Functions

We are going to look at functions that allow you to insert, append and remove items in a list. We will use shell for these purposes.

Open shell and place the following code in there to define a list:

```
grocery_list = ['milk', 'bread', 'butter', 'cheese']
```

Inserting an Item in a List

Here is how you insert an item in the third index of the list:

```
grocery_list.insert(2, 'cereal')
```

If you print out the list, you will get the following output with 'cereal' inserted in the third index of the list:

```
['milk', 'bread', 'cereal', 'butter', 'cheese']
```

Appending an Item to a List

If you want to append an item at the end of the list, this is how you do it:

```
grocery_list.append('sugar')
```

When you print the list now, this is what you will get with 'sugar' appended to the end of the list:

```
['milk', 'bread', 'cereal', 'butter', 'cheese', 'sugar']
```

Removing an Item in a List

Finally, if you want to delete an item in the list, this is what you would do:

```
grocery_list.remove('butter')
```

When you print the list now, this is what you will get with 'butter' missing from the list:

```
['milk', 'bread', 'cereal', 'cheese', 'sugar']
```

Many more functions exist that can allow you manipulate a list. I leave that up to you to explore.

Now that we have learned about the some of the basic data types let us get some action in these programs by giving the users the ability to enter data into our programs. We will also learn about converting one data type to another.

Chapter 6 - User Input and Output

By the end of this chapter you will learn:

- Input Function
- A Little Bit about Type Casting
- A Little Bit More about the Print Function

So far we have only been declaring variables and assigning values to them. Now let us make the programs a little more interactive by allowing the user to enter some input then performing operations based on the user input.

Adding input from the user makes your programs more usable.

We are going to look at some very important built-in functions that allow us to enter data in our program and display the data after we have done some processing to it. These functions are the conveniently named `input()` and `print()`

Enough talking let us let the user in on the action.

Input Function

We are going to imagine you are creating an awesome video game. The user just finished the level and you want to them to record their name and their score. Of course in an actual game, you would not allow a user to input their scores directly. But for the sake of keeping things simple, we shall allow it.

To get this data you use the `input()` function.

The code below shows you how you would allow the user to enter his name and score. Open up a new file and place the following code in there:

```
#Get user name from the user
user_name = input('Please enter your name: ')

#Get the score
score = input('Please enter you score: ')

#Display results to the player
print('Congratulations,' user_name + '. You scored', score, 'this level!')
```

Now save and run it. Let us break this program down line by line (well, except for the comments).

This line `user_name= input('Please enter your name: ')` will prompt the user to enter a name. The following will be the output of this to the screen:

Please enter your name:

At this point, the user will enter a name. Let us say they entered 'Richard' in there and pressed the Enter key. The input function will get the input and store it in the `user_name` variable.

The line `score = input ('Please enter you score: ')` will also prompt the user for input. The user has to enter their score in this case and the function will store the value into the `score` variable. Let us say they entered 4000 and hit the Enter key.

The line `print('Congratulations,' user_name + '. You scored', score, 'this level!')` will output all the information you entered to the screen:

Congratulations, Richard. You scored 4500 this level!

A Little Bit about Type Casting

Let us take a slight detour before we look into the print function a little more and talk a little bit about type casting. This is very important because the input function returns a string for the data the user has entered.

Don't just take my word for it. Try it out by opening shell and typing in the following code:

```
number = input('Please enter a number: ')
```

If you press the Enter key, it will prompt you for a number. Let us say you have entered 5, this will store the input into the variable `number`. Now press the Enter key again.

If you type `number` into Shell and hit Enter again, you should see exactly what I am talking about. It will output `'5'` with single quotes, meaning it is a string literal.

You cannot do any arithmetic operations on that. Again, i am not asking you take my word for it. So let us see an example.

While shell is still open, enter the following code:

```
number + 5
```

Press Enter and you should get a nice little error.

To solve this problem, you have to convert the string input into an integer using type casting. The function that does this is `input()` and you just place the string you wish to convert between the parenthesis.

Let us look at this in action using an example. Open up a new file and place the following code in there:

```
#gets input from the user and casts it into an integer
num_1 = int(input('Please the first number: '))
num_2 = int(input('Please the first number: '))

#multiplies the integers and prints the answer
answer = num_1 * num_2

#print the answer
print(answer)
```

What will happen in this program is that the lines `num_1 = int(input('Please the first number: '))` and `num_2 = int(input('Please the first number: '))` will prompt the user for input then cast the input into integers and store the values in the variables `num_1` and `num_2` respectively.

This line `answer = num_1 * num_2` multiplies the integer values and assigns the value into the variable `answer`.

Afterward the result is printed to the screen with `print(answer)`.

To Convert a float and string, use `float()` **and** `str()` functions respectively

A Little Bit More about the Print Function

We have been looking at simple ways to print information to screen and you have learned how to work with it a little. Let us take a look at the print function a little more.

Now we will look at two more things the print function can do just to make this a little bit more interesting.

We will look at multiline printing and escape characters.

Multiline Print

We have look at single line printing, but sometimes you might want to output information to the user over multiple lines using the print function. To do this, you use triple quotes (""") instead of double or singles quotes then spread the output across multiple lines.

The following example demonstrates this. Create a new file and place the following code in there:

```
#printing over multiple lines
print("""Nobody said it would be easy.
No one ever said it would be this hard.
I am going back to the start.""")
```

This should output the following text for you when you save and run it:

```
Nobody said it would be easy.
No one ever said it would be this hard.
I am going back to the start.
```

As you can see, the output is spread over multiple lines. Bonus points if you recognize who said those words above. The beg reveal is up next.

Okay, it is from the song **“Yellow”** by Coldplay. I am a huge fan.

Escape characters

Printing over multiple lines is pretty cool, I must admit, but what if you want to print things like quotes?

You could do something like `print(““this is a quote””) and surely enough, this will print “this is a quote” for you.`

Alternatively, you can use what is known as escape characters. An example of an escape character is the `\` (**black slash**) character. This allows you to print characters that are hard to type like quotation marks, tabs or new lines.

To print quotation marks, use `\`. For example, `print(“This is a quote”)` will print:

`‘This is a quote’`

To print a tab space, just use `\t`. For example, `print(‘I have put a\tTAB here’)` will print:

`I have put a TAB here`

Printing a new line is also very easy. You just enter the `\n`. For example, `print(‘This is a line.\nThis is another line.’)` will print:

`This is a line.`

`This is another line`

Use `\` to print a back slash. For example, `print(‘This is a \ (back slash)’)` will print:

`This is a \ (back slash)`

In the next chapter we will look at how you can enable your programs to make decisions using conditional statements. This is very import if you want the ability to control the flow of the programming and execute pieces of code based on certain conditions.

We will also look at more operators.

Chapter 7 - Conditional Statements

Here is what we will cover by the end of this chapter:

- More on Operators
- If Statement
- If-else Statement
- Elif Statement (Else-if Statement)

You are a smart person. No doubt. You made a decision and started reading this book instead of watching TV or playing your favorite video game because you wanted to be an awesome programmer.

So it makes sense to give you programs the ability to do certain make decisions and perform certain operations depending on the condition.

Here is an example of a condition:

```
x == y
```

Before we get into this, let us look at those operators I briefly mentioned in **Chapter 4**. For now it just matters that you know what a conditional is.

More on Operators

We are going to look at some operators that you will most likely use with these conditional statements

Comparison Operators

A comparison operator takes the values of operands on both sides, compares them and returns either true or false.

For the examples below, we are going to assume that the variable `x` equals 6 and the variable `y` equals 1. This will make everything I say after this point make a lot more sense.

The `==` (Equal to) operator returns `True` if the both operands are equal to each other; otherwise it returns `False`:

`x == y` **returns False.**

The `!=` (Not equal to) operator returns `True` if the operands are not equal to each other; otherwise it returns `False`:

`x != y` **returns True.**

The `>` (Greater than) operator returns `True` if the left operand is greater than the right operand; otherwise it returns `False`:

`x > y` **returns True**

The `<` (Less than) operator returns `True` if the left operand is less than the right operand; otherwise it returns `False`:

`x < y` **returns False**

The \geq (Greater than or equal to) operator returns `True` if the left operand is greater than or equal to the right operand; otherwise it returns `False`:

`x \geq y` **returns** `True`

The \leq (Less than or equal to) operator returns `True` if the left operand is less than or equal to the right operand; otherwise it returns `False`:

`x \leq y` **returns** `False`

Logical Operators

Below are some examples of the logical operators that you will be working with. Let us assume that `x` is equal to 4 and `y` is equal to 3.

The `and` operator returns `True` if all conditions are true; otherwise, it returns `False`:

`(x == 4) and (y == 2)` returns `True`

The `or` operator returns `True` if at least one of the conditions is true; otherwise, if all the condition are false, it returns `False`:

`(x == 6) or (y == 3)` returns `False`

The `not` operator negates the result of a condition. So if the result was `True`, it becomes `False` and if the result was `False`, it becomes `True`:

`not (x == 6)` returns `True`

Now let us get into those conditional statements.

If Statement

The if statement executes a block of code when the condition is met; Otherwise, it ignores it.

The syntax for the if statement is as follows:

If [condition is true]:

do something

White space is used to group the block of code after the:(semicolon). Everything indented after that is considered to be part of that if statement. To exit the code block, just start typing on a new line without any white spaces.

This is best demonstrated with the code:

```
option = int(input('Press 1: '))
```

```
if option == 1:
```

```
    print('You get a point!')
```

Here we prompt the user to enter the number 1 then we cast it to an integer with `int(input('Press 1: '))`. When all that is successful we assign the result into the variable `option`. Just remember to enter an integer or the program will crash if you enter a float or even a string.

Then we check if `option` is equal to 1 with `if option == 1:` and if the condition is met we print 'You get a point!' to the screen. If the condition is not met the code to print is ignored and the program ends.

Now save and run the program to test it yourself.

If-else Statement

The if-else statement is executed when a condition is met; otherwise, it does something else. It ignores the if statement and executes what is in the else statement instead.

We are just going to modify the previous syntax to tell the program to do something else when the condition in the if statement is false. The syntax should look as follows:

```
if [condition is true]:
```

```
    do something
```

```
else:
```

```
    do something else
```


This can be demonstrated with an example to see the if-else statement in action. Open up the editor and place the following code in there:

```
option = int(input('Enter 1: '))
```

```
if option == 1:
```

```
    print('You get a point!')
```

```
else:
```

```
    print('You get no points!')
```

When the user enters 1, the program outputs 'You get a point!' to the screen, because that condition has been met. If not and the user enters another number, the program outputs 'You get no points!' instead.

Now save and run the program to test it yourself.

Elif Statement (else-if Statement)

The elif statement is used to check for multiple conditions. When the condition is met, the rest are ignored.

Let us further modify the syntax from the previous if-else to include some elif statements. The code should look as follows:

```
if [condition 1 is true ]:  
    do something  
elif [condition 2 is true ]:  
    do something  
elif [condition 3 is true]:  
    do something  
else:  
    do something else if all conditions are false
```

The following code demonstrates the elif-statement. Open up the editor and place the following code in there:

```
option = int(input('Enter 1 or 2: '))  
  
if option == 1:  
    print('You get a point!')  
elif option == 2:  
    print('You get 2 points!')  
else:  
    print ('You get no points!')
```

In the code above, if the user enters 1, the program prints 'You get a point!' to the screen and the rest of the statements are ignored because the condition `if option == 1:` has been satisfied.

If the user enters 2, 'You get 2 points!' is printed to the screen because the condition for the elif statement `elif option == 2:` has been satisfied.

If none of the conditions are met, 'You get no points!' is outputted to the screen instead.

Now save and run the program to test it yourself.

In the next chapter we shall look at loops which perform operations on a block of code until a condition is met. This is very import and will save you a world of pain, especially when you want to print out items from a list. Printing them out one by one manually would be a pain if the list was very big. Loops make things like these a breeze.

Find out how to write loops in the next chapter. Things get a little more interesting.

Chapter 8 - Looping

Here is what we will by the end cover of this chapter:

- While loop
- For loop
- Break Statement
- Continue Statement

You are learning to program to make your life easier. So it follows that you don't want to type the items in a list one by one all the time. This is already tedious if the list contains ten items. Now imagine a hundred items. Madness!

What if you want to count down from a hundred? That can get old very fast because that is not easy living.

Lucky for you, Python can make things easy for you by allowing you to give your program the ability to loop through things like lists and even strings to manipulate the data however you want.

While Loop

The while loop executes a block of code until a predefined condition is met.

The syntax for the while loop is as follows:

```
While [condition]:
```

```
    dosomething
```

Just the conditional statements, white space is used to group the block of code after the: (semicolon). Everything indented after that is considered to be part of that loop. To exit the code block, just start typing on a new line without any white spaces.

The code below will demonstrate a simple count down to a missile launch. Don't panic. No real missiles will be fired. Open up the editor and place the following code in there:

```
count_down = 10

print('Missile will launch in')

#runs the loop until count_down is equal to or less than zero

while (count_down >= 0):
    if (count_down == 0):
        print('LAUNCH!')
    else:
        print(count_down)

    count_down -= 1
```

In this example we assign the value 10 to the variable `count_down`. This will be our iterator that will determine the number of times the loop will execute. Then we check if the condition has been met **with** `while (count_down >= 0):`

If the condition evaluates to true, then the loop will stop execution, otherwise it will continue to execute until the condition evaluates to true.

I have placed an if-else statement inside the loop just to demonstrate you can put them in there too. The loop just checks if the variable `count_down` is greater than or equal to zero and if it is, it prints LAUNCH! If it is false, it just prints the value of `count_down` **instead**.

While still in the loop, we subtract 1 from the variable `count_down` before the condition is checked again. This is very important to avoid going into an infinite loop that goes on forever (or until your program crashes, which is more likely).

Sometimes while loops are used when you don't know ahead of time how many times you are going loop. We shall revisit this in **Chapter 9** after you are familiar with modules.

For Loop

A for loop allows you to perform operations a set number of times.

A for loop is meant to be a little more variable than a while loop, although, it can essentially be used for the same tasks as a while loop such as a count-down.

The syntax for the for loop is as follows:

```
for iterator_variable in iterable_sequence:  
    do something
```

An `iterable_sequence`, in this case, is anything that can be looped over such as a list, tuple, string or dictionary.

The code below demonstrates how a for loop can be used to list the items in a list one by one. Open the editor a place the following code in there:

```
party_list = ['Jane', 'Alex', 'William', 'Dennis', 'Natasha']
```

```
for name in party_list:  
    print(name)
```

The first time this loop executes it will take the value of `party_list[0]` and assign it into the variable `name` then print it to the screen. As the loop continues to execute, it will do the same for the other items in the list one by one.

Now save run it. You will see all the names of the party list printed out to the screen on separate lines.

A for loop can also be used to list the items in a string one by one. Let us just look at an example:

```
message = 'This is a message'
```

```
for m in message:  
    print(m)
```

This loop will output the each character in the string to screen one by one (white spaces

included) on separate lines.

Break Statement

A break statement is used to get out of the loop even though the condition has not been met.

The following example shows the break statement in action. We are going to modify the while loop we did a little earlier for this purpose. Place the following code into a new file:

```
count_down = 10

print('Missile will launch in')

while (count_down >= 0):
    if (count_down == 0):
        print('LAUNCH!')
    else:
        print(count_down)

    if (count_down == 3):
        print('Something went wrong... ABORT!')
        break                #will force the loop to exit

    count_down -= 1
```

If you save and run it, you will see that when **count_down reaches 3**, the if statement will kick in and print 'Something went wrong... ABORT!'

After that the break statement will be executed and the loop will end there.

Continue Statement

The continue statement ignores everything that comes after it in the loop and returns to the top of the loop to begin execution again.

Let us remove the break statement we added earlier in our while loop and put in the continue statement. Now place the following code in a new python script:

```
count_down = 10

print('Missile will launch in')

while (count_down >= 0):
    if (count_down == 0):
        print('LAUNCH!')
    else:
        print(count_down)

    if (count_down == 6):
        count_down = 3
        continue                #everything after this will be ignored and the loop will start again

    count_down -= 1
```

When you save and run it, when the **variable**count_down**is** equal to 6 we will accelerate it a little by assigning the value 3 to the **variable**count_down.

Then continuing statement will be executed and the **expression**count_down -= 1**will** be ignored. The loop will return to the top, check the condition and start executing as normal again.

The count_down variable will now be equal to 3 and the loop will start running from there.

In the next chapter we will look out how to write re-usable code and bundle them up into what are known as functions. You have been using functions like print() all throughout the book but what are the functions?

Find out in the next chapter.

Chapter 9 - Functions

Here is what we will cover by the end of this chapter:

- Built-In Functions
- User Made Functions
- Variable Scope

So far we have been using functions throughout the book, but what are functions?

A function is basically a block of code that can be used repeatedly in a program. Think of it as writing a little program within your program and running it whenever you need it instead of having to re-write that code every time.

They also make our code more readable and more manageable.

Python allows you to write your own functions and also has a host of different built-in functions at your disposal to enhance your programs and provide extra functionality.

Built-In Functions

Python comes bundled with a lot of built-in functions that you can use to speed up programming. We have already looked at some of them like the `input()` and `print()` functions. We even used the `int()` function to cast strings into integers.

But there is more.

So many more functions that if you took the time to explore, you would find whatever functionality you are looking to add to your program.

Here is a small list of other built in function that use can use in your programs:

- **`range()` – This returns an iterable sequence. Very useful in for loops**
- **`open()` – This is used to open a file then return the corresponding file object**
- **`sorted()` – sorts a list then return the sorted list**
- **`help()` – used to invoke the help system that is built into Python**
- **`abs()` – returns the absolute value of a number**

Take time to explore. But for now we see how we can create our own functions.

User Defined Functions

To define your own function in Python, the syntax is as follows:

```
def name_of_function(parameters):
```

```
    Process data or variables
```

```
    Return [SomeExpression or variable]
```

The `def` keyword is what is used when defining a function.

It is followed by the name of function, which has to be unique and should not be the same as any of the other keywords in that have special meaning in Python.

After you give the function a name, you can put in optional parameters for the function within parenthesis. You can have as many parameters as you like. As long you separate them with a comma.

Then you can return a value based on the result of an expression or the value of a variable. This too is also optional.

As with conditional statements and loops, whitespace is used to denote a function block after the semicolon. So everything you type below, that is indented with white space, is part of the function.

Basic Function

Here is an example of a basic function that just outputs to the screen 'I am in a function!' and the sum of 2 and 3, which is 5. Open a new file and place the following code in there:

```
def example():  
    print('I am in a function!')  
  
    x = 2 + 3  
    print (x)
```

As you can see, it is easy to create a function in Python. But if you save and run the code above, nothing happens. You might be wondering why the program did not output anything to screen. This is because functions need to be called in order for them to execute the block of code in them.

To call a function you simply do the following:

```
example()
```

This should now print I am in a function!. It will also print 5 as well.

Try it.

Function Parameters

Now that we know how to create a basic function, let us work with parameters.

Parameters are data or variables that you pass along to a function and the function does operations of them.

The following example demonstrates an addition function that adds two numbers together. The numbers are passed as parameters. Open a new file and paste the following code in there:

```
def addition(num1, num2):  
    answer = num1 + num2  
    print(answer)
```

The code below shows how we can call the function with parameters and pass in values 2 and 3:

```
addition(2, 3)
```

In the above example, the function assumes you are entering numeric values. If you do not enter a numeric value and enter a string instead, you will get an error. Also, if you enter string values for both parameters, what will happen is a string concatenation.

If you save and run the code, it should output 5 after the function adds 2 and 3 together.

A few things to remember about parameters:

- The order in which you specified the parameters matters. So all values meant for `num1` should be inserted where `num1` is located in the parentheses and the variables for `num2` should also be placed where `num2` is located and so on
- Also the quantity of the parameters matters. If your function requires two

parameters, then you must enter two values. In the above example, doing this `addition(2, 3, 6)` will cause an error

Return Value

Not only can functions receive value using parameters, but they can also return values to us to print to the screen or assign them to a variable to manipulate them further.

In order to return a value, you put the **return** keyword followed by the expression or variable you wish to return a value. When the `return` keyword is encountered, the execution of the function will end.

The following example demonstrates the same additional function that has been modified in order to return a value from the addition. Open up the editor and place the code below in there:

```
def addition(num1, num2):  
    x = num1 + num2  
    return x
```

The function now adds the variables `num1` and `num2` and assigns the resulting value of the addition into the variable `x` with this line of code `x = num1 + num2`.

We then use the **return** keyword followed by the variable `x` to return the value by typing `return x`.

We call the function while passing in the parameters 2 and 3 to perform the addition:

```
answer = addition(2, 3)
```

We use the following code to output 5 to the screen:

```
print(answer)
```

Now save and run it.

Variable Scope

The scope of a variable denotes where the variable is accessible within our program.

A local variable is a variable that is only accessible in the function it is defined while a global function, once defined, is accessible anywhere within the program.

The following example will demonstrate this. Place the code below into a new file:

```
#declaring a global variable
global_variable = 'I am a global variable!'

def message():
    #accessing the global variable from within the function
    print(global_variable)

    #declaring a local variable
local_variable = 'I am a local variable!'
    print (local_variable)
```

In the example above, the **line** `global_variable = 'I am a global variable!'` declares a global variable and assigns a string with a message to it. Then we define a simple function which outputs the value of `global_variable` then declares a local variable with the line `local_variable = 'I am a local variable!'` and then outputs that to the screen as well.

We then call the function from the outside:

```
message()
```

If you save and run the program, this will be the output:

```
I am a global variable
I am a local variable
```

If you try you to try to this line `print(global_variable)` outside the function it will work as

expected because it is a global variable and it is accessible anywhere.

But if you try to print the local variable outside of the function it is defined with this line `print(local_variable)`, you will get the following error:

NameError: name 'local_variable' is not defined

In the next chapter we shall look at how to access many of Python's built in variables, functions and classes using modules that further extend the functionality of your program. We shall also create our own module while we are at it.

Chapter 10 - Modules

Here is what we will cover by the end of this chapter:

- Importing a Module
- Creating a Module

Modules contain a lot of built-in codes that are written inside Python files (file with py or pyw extension). You have probably created modules all along without even knowing it. These modules give you built-in functions, variables, classes and other things that help you speed up the writing of your programs.

All you have to do is import them.

Let us take a look at importing Python modules.

Importing a Module

To import a module is very easy and Python has three different ways for you to do it.

The following syntax shows you how you would go about it the first and most common way:

```
import module_name
```

Just use the `import` keyword followed by the module name you wish to add. And bingo! You have added extra functionality to your program that you did not have to code yourself from scratch. You can even add more than one module at a time; just separate the module names with commas.

Here is an example of how you would go about importing the `math` module in Python:

```
import math
```

Now you have all the built in capabilities of the `math` module. Here is an example of how you can use the `sqrt()` function of the `math` module:

```
print(math.sqrt(9))
```

This should output the square root of 9, which is 3

You can import the `math` module another way if you find the first way is giving you hassles because you have to type `math` all the time. Here is how you would do it:

```
import math as m
```

Now you can display the value of PI from the `math` module by accessing the `pi` variable:

```
pi = m.pi  
print(pi)
```

This should output:

3.141592653589793

The third way is to just import the specific functions you want to use. Here is the syntax:

```
from module_name import function_name
```

You can import as many functions as you want. Just separate the function names by commas.

Here is an example:

```
from math import sqrt
```

Now in order to use the `sqrt()` function, you just type the function name only. The following example demonstrates how to display the square root of 9:

```
print(sqrt(9))
```

We shall look at one more example that involves the while loop which I briefly mentioned in **Chapter 6**. I said the while loop is also useful for looping if you don't know how many times you are going to loop.

Now that you have a good understating of functions and importing modules, the following code will make a lot more sense. We are going to import the `random` module and use the `randrange()` method to loop until a certain random number we are looking for is found.

Open up a new file and place the following code in there:

```
import random
```

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

```
#execute the loop while random_number is not equal to 4
```



```
while random_number != 4:  
    print(random_number)  
    random_number = random.randrange(0, 10)
```

The first line will import the module `random`. This is why I excluded this example in Chapter 6. This would have made less sense.

The next line is `random_number = random.randrange(0, 10)`. This line will assign a random number to the variable `random_number` using the function `randrange` from the `random` module.

We enter two parameters in the function telling the range to start at 0 and stop at 10. This will return a number between those two values including the number 0, which will be included in the range but 10 will be excluded from the range.

Next we start the loop and check if `random_number` is not equal to 4 with this line `while random_number != 4`. If it is not, we print the value and get another number with the `randrange()` function. If `random_number` is equal to 4, the loop ends.

Now save and run the program. It will display a list of numbers until the condition is met.

Now that you know how to extend the capabilities of your program using built-in modules, we will look into how you can go about creating your own modules.

Chapter 11 - Creating a Module

Sometimes you may want to extend the functionality of a Python program yourself by creating your own modules. Here is how you go about it.

Open shell and create a new Python script and save it as **sub_mod.py** somewhere on your desktop. Place the following code into the file and save it:

```
def subtraction (num1, num2):  
    answer = num1 - num2  
    return (answer)
```

Congratulations, you have created a module. It has a function named `subtraction` that takes in two numbers `num1` and `num2` as parameters. The line `answer = num1 - num2` subtracts them and assigns them to the `answer` variable. Finally, we use this line `return (answer)` to return a value.

Now you can use this module in any program however you like. Just import it.

Now create another file and call it **sub_imp.py** and save it on your desktop as well. Place the code below in that file:

```
import sub_mod
```

```
answer = sub_mod.subtraction(8, 3)
```

```
print(answer)
```

As you can see, all we had to do is import the module we created using the line `import sub_mod`.

We then call `subtraction()` function from the `sub_mod` module, passing in two values then assigning the returned value to a variable with this line `answer = sub_mod.subtraction(8, 3)`

Then we use the variable to output the answer with the line `print(answer)`. The output of subtracting 3 from 8 is 5 and that is what will be printed out.

For this to work, both files need to be in the same directory. You can place them in different directories then append the file path where the file is located to the list of directories that python goes through to search for files and modules. But we won't be discussing that in this book to keep things simple.

In the final chapter of this book, we shall look at how we can work with text files. Python grants you the ability to read and write and also rename and delete files with ease. This goes beyond asking input from user only to actually processing data from external files and even storing data in them.

All this is in the next chapter. Keep reading.

Chapter 12 - Working With Files in Python

Here is what we will cover by the end of this chapter:

- Opening and Reading a Text File
- Writing to a Text File
- Renaming and Deleting Text file

So far we have been able to get input from the user using **the `input()` function**. We did this in **Chapter 6**. As you recall, this prompted the user to enter something. But that is not all Python can do.

Python can also retrieve data from external files such as text, binary and image files then print that information to the user. We are going to be working with text files in this chapter.

We will be opening, closing, reading, writing, renaming and deleting text files. There are a lot of things to do in this chapter. Shall we get started?

Open and Reading a Text File

Before we open a text file, it needs to exist. Open up any text editor enter the following three lines and save in the same directory as were the program will be saved. Name the file **example.txt**:

Lights will guide you home

And ignite your bones

And I will try to fix you

Okay, you got me. They are lyrics from another Coldplay song called “Fix You”. Did I mention how I really love Coldplay?

The syntax to open a file in Python looks something like this:

open(‘name of the file’, ‘file mode’)

The first parameter is where you enter the path to where the file is located. Since our file is located in the same directory as the program, the path will simply be ‘example.txt’.

The second parameter is for the file mode we shall be opening the file in. Here is a list of some of the file modes we can use:

- **‘r’ mode – This means the file has been opened in read-only mode**
- **‘w’ mode – This means the file has been opened in write-mode for only writing to it. Also, if the file does not exist yet, it will be created. But if the file exists, all existing data will be erased**
- **‘a’ mode – This opens the file in append-mode for appending data at the end of the file. If the file does not exist, it will be created and if it exists, the data will not be erased**
- **‘r+’ mode – This will open the file in both read and write mode.**

Now that we know a bit about **the open() function**, to open and read from a file in Python is easy. Create a new file and place the following code in it:

```
#opening a the text file example.txt
example_text_file = open('example.txt', 'r')

#read some lines and print them to the screen
first_line = example_text_file.readline()
second_line = example_text_file.readline()

print(first_line)
print(second_line)

example_text_file.close()
```

To begin with, the line `example_text_file = open('example.txt', 'r')` will open a text file and set it to read-only mode then assign the resulting text object in to the variable `example_text_file`.

To read from a text file, you use the `readline()` function which always reads the next line from a file. So the first `example_text_file.readline()` line will read the first line from the text file then assign the resulting string value to variable `first_line` and the second one will read the second line and assign it to the variable `second_line`.

Reading from a text file is as easy as that.

The next lines will just print out the values of the `first_line` **and** `second_line` variables to the screen,

We then use the `close()` function to close the file since we are done using it. This will free up

space in memory for our program. The line `example_text_file.close()` does this for us in our program.

If you save and run the program, you should get the following output:

Lights will guide you home

And ignite your bones

Okay, now let us look at writing to the same text file.

Writing to a Text File

The code for writing and appending to a text file is basically the same. All you change is the file mode. Since write-mode will erase all the data, I will use append-mode instead and just add a line of text to the end of the text file that we created in the previous section.

Just note that opening it in this mode means that we can no longer read from it.

Create a new file and save it the same folder as the **example.txt** then place the following code in there for appending:

```
#opening example.txt in append mode to avoid erasing the data
```

```
example_text_file = open('example.txt', 'a')
```

```
#this appends a line of text at the end of example.txt
```

```
example_text_file.write("\nI forgot how the rest of it goes")
```

```
example_text_file.write("\nI need to look up the lyrics")
```

```
example_text_file.close()
```

Since you already know what the `open()` function does, I bet you have a pretty good idea what the line `example_text_file= open('example.txt', 'a')` does.

Yes. It opens the text file `example.txt` in append mode.

We then use the `write()` function to write to our file and since the file is in append mode, the lines `example_text_file.write('\nI forgot how the rest of it goes')` **and** `example_text_file.write('\nI need to look up the lyrics')` will be added to the end of the file. We use the `\n` (new line) escape character to write each of them on a new line; otherwise, they will both be added to the last line of the file.

The last line in our program just closes the file since we are done using it.

If you save and the run the program, the txt file **example.txt** should look something like this when you open it:

Lights will guide you home

And ignite your bones

And i will try to fix you.

I forgot how the rest of it goes

I need to look up the lyrics

Renaming and Deleting the Text File

The functions we need to use for renaming and deleting a file are in the `os` module. So you need to import them into your program before you can use them.

The syntax for renaming a file in Python is `rename('old file name', 'new file name')`. So if you wanted to rename the text file **example.txt** to something like **renamed.txt**, you would write `rename('example.txt', 'renamed.txt')`.

Removing a file is even easier. The syntax to remove a file in Python is `remove('file name')`. To remove the **example.txt**, you would write `remove('example.txt')`.

That is enough fun with files. In fact, that is enough fun with this book as well. You have made it to the end. But before you go, let us just quickly recap what we have learned as I conclude the book in the page.

Conclusion

It is my sincere hope that if you here, then this book endowed you with awesome programming abilities that you can improve upon as you head into the awesome world of programming. And I also hope that you use your abilities for good because, as you already know, with great power comes great responsibility. It is now up to you to create programs that help you or others. Programming makes life easier.

Let us briefly recap what we covered in the chapters of this book.

In the first chapter, we covered how to install the Python 3 interpreter to be able to run our programs. We also looked at Shell and IDLE then used them to create a simple ‘Hello World’ program. Finally, we looked at comments and the print function a little.

In the second chapter, we dove deeper into Python by looking at variables and how they act as containers for data types. We created them, assigned values to them and looked at some arithmetic and assignment operators we can use to manipulate them.

In the third chapter, we looked at the various basic data types that the Python language supports, such as the numeric, string and list data types. We looked at various functions for the numeric and list types that you can use to manipulate them and also how to combine and format strings.

The fourth chapter looked at how we can get input from the user and how to convert it into different data types. We also saw a little more of the print function in action by looking at multiline printing and how to use escape characters.

In the fifth chapter, we were able to control the flow of our program by enabling it to be able to make decisions on which code blocks to execute based on conditional statements. But not before we looked at the comparison and logical operators we can use in conditional statements.

In the sixth chapter, we were able to introduce how to loop through code blocks until a certain condition was met then used that to print out information from lists and create countdowns.

In the seventh chapter we covered functions. We looked at built-in functions then how to create our own functions; call them, pass them information and have them return back information for us to use in our program.

In the eighth chapter we looked at how to import modules into our programs that give us access to various variables and functions that enable us to extend the functionality of our program. We even learned how to create and import our very own models

In the ninth and final chapter, we looked at looking getting input from external files such as text files and how to manipulate them.

Now that you have mastered the basics, you are at a position to tackle the advanced topics and look into object oriented programming and how you can create classes. You can also look into advanced collection types like dictionaries or how to retrieve and display images or read binary files.

You have so much more to explore. But don't despair. I know you can do it.