

PYTHON

PROGRAMMING

The Ultimate Crash Course for Beginners
with all the Tools and Tricks to
Learn Coding with Python

with
PRACTICAL
EXAMPLES



HOWARD HAYES

PYTHON

PROGRAMMING

The Ultimate Crash Course for Beginners
with all the Tools and Tricks to
Learn Coding with Python



HOWARD HAYES

Python Programming

*The Ultimate Crash Course for Beginners
with all the Tools and Tricks to Learn
Coding with Python (with Practical
Examples)*

© Copyright 2019 - All rights reserved .

The content contained within this book may not be reproduced, duplicated or transmitted without direct written permission from the author or the publisher.

Under no circumstances will any blame or legal responsibility be held against the publisher, or author, for any damages, reparation, or monetary loss due to the information contained within this book, either directly or indirectly.

Legal Notice:

This book is copyright protected. It is only for personal use. You cannot amend,

distribute, sell, use, quote or paraphrase any part, or the content within this book, without the consent of the author or publisher.

Disclaimer Notice:

Please note the information contained within this document is for educational and entertainment purposes only. All effort has been executed to present accurate, up to date, reliable, complete information. No warranties of any kind are declared or implied. Readers acknowledge that the author is not engaging in the rendering of legal, financial, medical or professional advice. The content within this book has been derived from various sources. Please consult a licensed professional before attempting any techniques outlined in this book.

By reading this document, the reader agrees that under no circumstances is the author responsible for any losses, direct or indirect, that are incurred as a result of the use of information contained within this document, including, but not limited to, errors, omissions, or inaccuracies.

Table Of Contents

[Introduction](#)

[A Brief History of Python](#)

[Who Should Read This Book?](#)

[What Does This Book Have to Offer?](#)

[Chapter 1: Python Installation](#)

[How to Install Python on Windows Operating System](#)

[Installing Python on Linux](#)

[Installing Python on macOS](#)

[Python IDLE](#)

[How to Execute a Program in Python](#)

[Chapter 2: Python Syntax and Data Types](#)

[Python Syntax](#)

[Python Data Types](#)

[Creating Variables in the Python Shell](#)

[Chapter 3: Python Variables, Strings, Numbers](#)

[Python Variables](#)

[Python Strings](#)

[Multiline Strings](#)

[String Concatenation](#)

[Chapter 4: Python Loops](#)

[Using For Loop](#)

[Chapter 5 : Tuples and Lists](#)

[Python Tuples](#)

[Python Lists](#)

[Modification, Addition and Removal of Items From Lists](#)

[How to Organize a List?](#)

[Chapter 6: Python Input Function](#)

[The input\(\) Function](#)

[The int\(\) Method](#)

The While Loop

[Use of while Loops in Lists and Dictionaries](#)

Chapter 7: Python If Statements

The And Keyword

The Or Keyword

The Nested If Statement

You Can Use If Statements With Lists

Chapter 8: Python Functions and Dictionaries

How to Define a Function?

How to Pass Information to the Function?

A Look at Arguments and Parameters

[How to Pass Arguments to Python Functions](#)

[Positional Arguments](#)

[Keyword Arguments](#)

Filling the Function With Default Values

You Can Pass a List to a Function

[Global and Local Variables in Python](#)

[Using a Function With a While Loop](#)

Python Dictionaries

Python Calculator

Conclusion

References

Introduction

When I was a kid, I had no interest in computers, let alone programming languages. When I turned fifteen, my father bought me a personal computer system by IBM with the Windows Operating system pre-installed on it. On my first day, I liked it so much that I spent a full night playing with it. The night itself was very interesting. Dark clouds were hovering over the skies when I entered my room in the evening. That night, my computer was just close enough to the custom windows inside my room that I could see outside. Off and on the lightning made me feel her powerful presence. Coupled with the frightening lightning was the smashing wind that slapped the windows and tried to push through. I was on my computer playing games and watching video cartoons when it started raining wildly. Our house was drenched in rain from roof to basement. I was alone in the room but still extremely fearless because I had that machine in front of me. Suddenly, the rain stopped and the moon came out from behind the clouds. Its light was dazzling bright and it almost mesmerized me to the core. Ignoring the beauty of the full moon, I turned to the computer and started checking its functions and other things that it had to offer. It was then that I came across the command prompt. Unlike other kids who like colorful screens, I really found attraction in that black screen. That was the first moment when I fell in love with computer programming. I didn't know what that black screen was about, but it was the start of a new beginning. I kept entering different keys but understood nothing in the end.

After several failed attempts, I was disappointed and closed the command prompt and went to bed. The night was cold enough to send shivers down my spine. I had a sound sleep that night and the next day when I woke up, I got a new hobby, and that was to learn about that black screen. Gradually, I started to

learn computer programming languages. It was Python that really clicked for me when I came across it. The freedom of programming that Python gave to me was unmatched by any other programming language. I really loved the ease of use, the open community, and the freedom of implementation of Python programs. Python has really changed my life.

A Brief History of Python

Originally, Python was created by Guido van Rossum. He was a member of the National Research Institute of Mathematics and Computer Science. Rossum created this language in response to the ABC programming language. Python carried major features of the ABC programming language, and in addition, it was equipped with some advanced features like exception handling. So, Python followed in the footsteps of the ABC programming language but was better.

Just like all the other programming languages, Python too went through a number of versions. When it was first released in 1991, its version was Python 0.9.0. Python also had received the features of strings, lists, as well as classes. The feature of class placed Python among advanced languages to be used for the purpose of automation. Only the Python exception handling feature was amazing, as it allowed users to handle programming errors in a better way. Python went through upgradation in 2000 and the new version named as Python 2.0 was released. This version of Python was considered as an open source project. The creators added new features in Python like the garbage collector, list comprehension, as well as the supported Unicode. Python 3.0 is the latest version and was released in December 2008. Programmers consider Python 2 and 3 as copies of one another with slight differences. The major difference on which most programmers agree is the feature of print statement. In fact, the Python print statement has been replaced with the `print()` function.

Who Should Read This Book?

This book is produced for the people who are trying to push their way through the world of Python programming. You don't need to have prior knowledge of Python programming or any other programming language to get started with this book. In fact, this is one of the best qualities of Python—the fact that you can start from the very beginning. The syntax is easy to read, create, and run. You

should have a basic understanding of how a computer works, like how to download and install different applications. Python should be downloaded and installed on your computer operating system. We will talk about this topic in the first chapter.

What Does This Book Have to Offer?

This book is intended for beginners who are looking to learn Python programming. Let's see what it has to offer to you. I will dissect different chapters of the book in the following paragraphs. Let's roll on.

The first chapter walks you through the process of downloading and installing Python on your computer system. You will learn how to get it installed on your Windows operating system, Linux operating system, and macOS. I will also explain the importance of Python IDLE, an interactive Python editor to write Python coding. In addition, you will learn how you can run the Python program after you have written it in the editor. You will also learn where you will see the results of each program.

The next chapter talks about Python syntax and data types. You will be learning about different data types in Python.

The third chapter will shed light on Python variables. You will get to know about Python strings and the way you can use them and manipulate them. For example, the chapter includes how you can write multiline strings and you will also learn about string concatenation.

The fourth chapter is a short one and will talk about Python for loops. You will know how to use them, and for which purpose you can use them.

The next chapter focuses on Python tuples and lists. This will help you understand how list modification works and how you can add and remove items from lists. This chapter also contains live programs which you can use to learn and understand lists. In the end, you will learn what unordered lists are and how they are organized.

The next chapter focuses on the `input()` function. This is an advanced stage of programming. This chapter teaches you how you can take input from a user and then use it to process certain data and yield the output. Coupled with the `input()` function is the introduction of the while loop which you can use in the `input()` function, lists, as well as dictionaries.

The next chapter is centered around the if statement. This section includes creation of if statements, different options that can be used in the if statement, the nested if statements, and pairing up if statements with lists. At the end of the book, you will see a program that includes if else statements as a practical application.

The last chapter of the book is focused on Python functions and dictionaries. You will learn how to define a function, as well as pass information to it through different kinds of arguments and change in parameters. You will get to know about global and local variables. Also included in this chapter are Python dictionaries. You will learn how you can create a dictionary, access different items in a dictionary, add and remove items to and from a dictionary, and also use it. The chapter ends with the creation of a couple of programs like a calculator and a program to know whether a given year is a leap year or not. You can copy the source code, edit it to suit your understanding and preference, and then run it in the Python editor to see how it works.

In addition to the programs that are given in the end, the book also has a number of small programs scattered in different chapters. You can use those programs, throw them in the Python editor, edit them according to your own suitability, and then run it to see the results. The book contains basic and advanced level Python programming topics that you can use, learn, and master to become a Python programmer.

Chapter 1

Python Installation

If you want to get started with Python programming, you should have access to the Python interpreter. You can directly download Python 3 from the website www.python.org. You will be required to download the appropriate installer as per the specifications and requirements of your personal computer and operating system. If you are using Linux operating system, you will get your hands on a package manager which you can run for the installation of Python. For macOS users, you will have to install a separate package manager named Homebrew. Also, Python can be installed on mobiles and iOS. This is the best for more passionate programmers. You can practice on the go!

How to Install Python on Windows Operating System

You are unlikely to find Python pre-installed on your Windows operating system. As already said, you can directly download Python from www.python.org. Let's explain the steps involved in the installation of Python 3.

This requires you to click open an Internet browser and directly navigate to the download page to get the Python installer. Find the Python 3 version that matches your system. There are different versions available for 32 bit and 64 bit operating systems. For a 32 bit system, it is a must that you install Python 3 executable installer that specifically supports the 32 bit system. Python 64 bit

executable installer supports the 32 bit system as well as the 64 bit system. So, you can install either. Python installer for a 32 bit system will not take much memory, while on the other hand Python installer for a 64 system will be more efficient when it comes to writing complex codes.

Once you have downloaded the Python installer you need, it is time to run the installer. You can do that simply by double-clicking the field that you have downloaded. You will immediately see a dialog box on the screen. There will be an option 'Install now' in the middle. In the bottom, you will see the phrase 'Add Python to PATH' next to a check box. You have to tick that off and click on Install now. The installation process will kick off right away. Within a couple of minutes, you will be able to run Python on your computer system (Python 3 Installation & Setup Guide, n.d.).

Installing Python on Linux

Linux is amazing due to the fact that most of the time you will get Python pre-installed in the Linux distribution. The only problem is that it will have Python 2, which is the older version of Python. This version also works okay. But if you want to install the latest version that is Python 3, you will have to manually install it. So, first of all open the Linux terminal and enter the following command to check which version you have installed on the system.

```
$ python --version
```

The terminal will return with the value of the version that you have with the distribution. If you are using Ubuntu distribution on your Linux operating system, you can install it with the help of the following command.

```
# sudo apt-get install python3
```

I have been logged into a Linux system as a root user, and that's why I placed the # before the command. You are most likely to see \$ sign if you are not logged as a root user (Python 3 Installation & Setup Guide, n.d.).

Installing Python on macOS

Most macOS also has Python pre-installed, but this is an older version that is Python 2. So, if you want to install the latest version, you will have to install it

manually. You will have to install Python 3 with the help of Homebrew package manager. So, firstly we will have to install Homebrew.

Click open your Internet browser and enter the following url: <http://brew.sh/>. When the page has been fully downloaded, you have to copy the bootstrap code by pressing Cmd and C buttons. All the text in the code must be copied to the clipboard or the installation will fail.

Now click open the terminal app and paste the copied code in it. Then press the enter button. The installation will kick off right away. The users who have newly installed macOS will see another popped up window that will ask them to install the command line developer tools. Don't ignore it and close it. You are going to need those for the installation process. Click the install button and proceed. These tools will take some time to be installed on the operating system.

When they are installed, you can go on with installation of Homebrew. After that you can install Python. When the tools have been installed, you will be asked to confirm the installation. Now get back to the terminal. Press the enter button and go on with the installation process. Homebrew is going to take some time to get installed on your system. The faster the Internet connection, the speedier the installation of Homebrew will be. When you have installed Homebrew, you can now get back to the command prompt. Type and run the following command in the command prompt.

```
brew install python3
```

The above command will install Python 3 on your macOS. Now you have to click open the Terminal.app. Enter pip3 command in the terminal. Pip package manager will show text. If you get an error instead, repeat the installation process.

You can use Python interpreter in almost all the operating systems without any hassle. Just make sure that you have installed the version with which you are comfortable (Python 3 Installation & Setup Guide, n.d.).

Python IDLE

Python IDLE is an integral part of Python installation package. It is also known as Integrated Development Environment (IDE). It is a combo package of an editor for writing programs and a language environment for programmers. This is not mandatory for writing Python programs, as there are a wide range of other

IDEs available for composing programs in Python. IDLE stands out among them because it is a part of Python installation. In addition, IDLE is easy and comfortable to use for beginners.

As already stated, Python is a part of the installation package. If you have just installed Python on your Windows 10 operating system, you pop open the Windows menu. To the top of the menu you will find the Recently Added items list. Here you will also see Python IDLE. Hit it and launch it. If it is not there, you can trace it by scrolling down the list of items from top to bottom. Trace it and launch it. After launching it you will be able to see the Python Shell window on your screen. You will see the following text when you launch the shell window.

```
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 22:22:05) [MSC v.1916 64 bit (AMD64)] on win32
```

```
Type "help", "copyright", "credits" or "license()" for more information.
```

```
>>>
```

The above text shows the message that you will get if you have installed Python 3.7 on your computer system. In this shell window, you can use Python commands and also edit different Python programs. Unlike C++, Python is not a compiled language but rather an interpreted language. It is natural for an interpreted program to run slower than a compiled language program. Apart from this there is generally no difference. Interpreted languages are great in the sense that they offer programmers an interactive experience to test commands and algorithms. You don't have to write and finish a program to test it. Rather, you can test multiple commands in a single shell without finishing one program. This offers a faster way to learn commands. In simple words, you can test individual commands without writing a complete program in the interpreter.

This interactivity is the reason why an interpreter is used to code in Python. The three greater than symbols that you see in the Python shell denote the shell prompt. These symbols in the Python shell mean that the shell awaits you to write a command. You cannot just write anything or it will return an error. Let's see how Python IDLE works.

```
>>> michael
```

Traceback (most recent call last):

```
File "<pyshell#0>", line 1, in <module>
```

```
michael
```

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

```
>>>
```

You have to write something in a proper format.

```
>>> print("michael")
```

```
michael
```

```
>>>
```

Python shell treats text differently from integers. Let's see how.

```
>>> 45 * 55
```

```
2475
```

```
>>>
```

You don't have to use parentheses or double quotation marks to get the results in the Python shell when it comes to integers. Whenever you want the shell to display a piece of text, you can enclose it inside parentheses in a print statement (Using IDLE (Python's IDE), n.d).

How to Execute a Program in Python

You can play around the Python shell to try and test Python commands, but our ultimate goal is to write programs in Python. Launch the IDLE interpreter and click on the File option. Hit on the New Window option. A new window without a title will pop open. This window is an editing window and you can write Python commands in this window to create Python programs. In this window, you won't see the shell prompt and you also won't see any instant results of Python commands. You will have to execute the group of commands once you have filed a Python program.

```
name = input("You will have to enter your name: ")
```

```
print("Hi, " + name + "!")
```

```
print("Your name is a nice one.")
```

I have written the above program in the Python editor. When you are done with this, you have to run it. Go to the menu in the top bar. Click it. A list will drop

down. Hit the run module option. The shell will ask you to save the source code. Save it in the location you want it to be in. In the Python shell, you will immediately see the following statements. The shell prompt will demand you to write your name. When you are done with that, hit enter. Let's see what happens when you enter your name in the shell.

You will have to enter your name: Smith

Hi, Smith!

Your name is a nice one.

>>>

One important thing to remember is that when you run the program from the Python editor, you will see a Python shell window pop open on the screen. The program will be executed in the shell. Python editor had no role in displaying the executed program. You can see that the saved file in the Python editor carries the .py extension. When you are done with running the program, exit the Python shell by hitting the cross sign.

In this chapter we have learned how to install Python on Windows, Linux, and macOS operating systems. After that we learned how the Python IDLE works and how can you write and execute the Python program (Using IDLE (Python's IDE), n.d).

Chapter 2

Python Syntax and Data Types

Now that we have learned to install and use Python shell and editor, it is time to move on to the next level of coding. In order to successfully write and execute Python programs, we will have to learn about Python syntax and the type of data you can enter into the Python editor and the shell for execution. Unless you don't know the data types that are used for calculation and other programming purposes, you are unlikely to get everything right in the end.

Python Syntax

I have demonstrated a little bit of Python coding which definitely has given you a taste of Python syntax. Let's analyze the syntax to give you a brief introduction on how to write a simple statement in the Python shell where you can try and test your commands before you make them a part of the group of code for creation of a full program.

```
>>> print("This book is on Python programming")
```

```
This book is on Python programming
```

```
>>>
```

You can see that in the above code snippet, the three greater than signs denote the Python shell prompt. Next comes the print statement. I have enclosed it

inside parentheses after placing it inside double quotation marks. When I press enter, I get the display as under. The same happens when you put it in the Python editor and then execute the program. You can also run the .py file in the Python Command Line by invoking the file. An important thing about Python syntax is that it is sensitive to whitespaces. You miss an indentation and you will end up committing syntax error. Let's try another Python syntax and see how the indentation can ruin your code.

```
>>> if 7 > 5:
    print("Seven has greater value than five. Mind it.")
```

Seven has greater value than five. Mind it.

```
>>> if 7 > 5:
print("Seven has greater value than five. Mind it.")
SyntaxError: expected an indented block
```

```
>>>
```

You can see how a simple indentation error has ruined your program. Python shell will return an error telling you that you have suffered from an indented block. An interesting thing is that there is no restriction on the maximum spaces for the print statement. The only important thing is that the space should be at least one. Let's check it out.

```
>>> if 7 > 5:
    print("Seven has greater value than five. Mind it.")
```

Seven has greater value than five. Mind it.

```
>>> if 7 > 5:
    print("Seven has greater value than five. Mind it.")
```

Seven has greater value than five. Mind it.

```
>>> if 7 > 5:
```

```
print("Seven has greater value than five. Mind it.")
```

Seven has greater value than five. Mind it.

```
>>>
```

Python Data Types

In order to be a successful programmer, you need to get expertise in the data types that can be found as built-in in the Python shell. You can store different types of data in variables. Each data type has a specific job to do when it comes to functioning. I will be demonstrating how to store the data in the Python shell and then I will move on to explore the world of data types.

Creating Variables in the Python Shell

You can create Python variables in the Python shell by assigning a value to them in the following way.

```
>>> a = 85
```

```
>>> y = "I am learning Python."
```

```
>>> print(a)
```

85

```
>>> print(y)
```

I am learning Python.

```
>>>
```

I have stored an integer and a piece of text in two separate variables, and then I applied the print function to see the results. Now let's explore the types of data in the Python shell. You can specify the data type at the time of writing the code. This will help you remember what type of data you are using and also help you in minimizing the mistake percentage when writing a program. For your convenience, I will be storing each data type in the same variable. Each time I will store a new data type in the variable 'x,' it will overrun the existing data.

```
frozenset({'brinjal', 'radish', 'bell pepper', 'ginger', 'carrot', 'chilly', 'pumpkin'})
```

```
>>> x = tuple(("radish", "ginger", "pumpkin", "carrot", "brinjal", "chilly", "bell pepper"))
```

```
>>> print(x)
```

```
('radish', 'ginger', 'pumpkin', 'carrot', 'brinjal', 'chilly', 'bell pepper')
```

```
>>> x = bool(55)
```

```
>>> print(x)
```

```
True
```

```
>>> x = str("I am learning Python very well.")
```

```
>>> print(x)
```

```
I am learning Python very well.
```

```
>>> x = dict(word= "Sky", synonym= "heaven")
```

```
>>> print(x)
```

```
{'word': 'Sky', 'synonym': 'heaven'}
```

```
>>> x = int(55)
```

```
>>> print(x)
```

```
55
```

We can see that each data type when printed displays different data in different forms. Out of all the above, lists and tuples appear to be the same. But in reality, they have a great difference. Lists can be changed once they are created, while tuples cannot be changed. More on this in the upcoming chapters. Apart from this, there are dictionaries, sets, and bytes. Dictionary is very interesting. By using this data type, you can create your very own custom dictionary in a short span of time (Python Data Types, n.d).

Chapter 3

Python Variables, Strings, Numbers

This chapter is connected to the last one because of the presence of Python variables. We have already learned how to store value in a variable in the Python shell. Variables are basically containers that are used to store different values. There is no specific command for declaration of a variable in the Python shell.

Python Variables

As you have read in the last chapter, all you have to do is simply write the variable, put an equal sign next to it, and then enter the value in the end. Press the enter button. You have successfully stored whatever value you have in the variable. We have also seen this in the data type section. Another important aspect of variables is their flexibility when it comes to reassigning the value. I have reassigned different values to a single variable `x` in the last chapter with the utmost ease. Let's repeat the process to explain it further.

```
>>> x = 55 # this is an integer
```

```
>>> print(x)
```

```
55
```

```
>>> x = "Michael" # this is a string
```

```
>>> print(x)
```

```
Michael
```

```
>>>
```

Also, you can assign multiple values to multiple variables in a single line.

```
>>> x, y, z = "radish", "carrot", "pumpkin"
```

```
>>> print(x)
```

```
radish
```

```
>>> print(y)
```

```
carrot
```

```
>>> print(z)
```

```
pumpkin
```

```
>>> x = y = z = "radish"
```

```
>>> print(x)
```

```
radish
```

```
>>> print(y)
```

```
radish
```

```
>>> print(z)
```

```
radish
```

```
>>>
```

In the last code snippet, I have assigned the same value to three different variables, and it worked perfectly. This is how we can navigate around the Python shell, experimenting with variables. You can manipulate variables in different ways. For example, you can add text to a variable by inserting the plus sign in between the two. Let's see how we can do that.

```
>>> x = "radish"
```

```
>>> print("I am going to cook " + x + " in the evening.")
```

```
I am going to cook radish in the evening.
```

```
>>>
```

Another interesting thing about Python variables is that you can pair up two variables by inserting the plus sign in between them. Let's demonstrate this in the Python shell. There are two ways to pair up variables. See the following code snippet.

```
>>> x = "I am going to cook "  
>>> y = "radish in the evening."  
>>> z = x + y  
>>> print(z)  
I am going to cook radish in the evening.  
>>> print(x + y)  
I am going to cook radish in the evening.  
>>>
```

Also, Python shell offers you the best mathematical operator. You can assign mathematical values to Python variables and then run the mathematical functions. Let's see how it is done.

```
>>> x = 35  
>>> y = 25  
>>> print(x + y)  
60  
>>>
```

As with all programming languages, Python has its limitations. You cannot just combine an integer and a string. If you try to do that, you will see an error in return.

```
>>> x = 1000  
>>> y = "I am going to cook in the evening."  
>>> print(x + y)
```

Traceback (most recent call last):

```
File "<pyshell#149>", line 1, in <module>  
    print(x + y)
```

TypeError: unsupported operand type(s) for +: 'int' and 'str'

```
>>>
```

We can also pair up variables with functions. We can create them outside a function and then use them inside as per requirement.

```
>>> x = "radish"
```

```
>>> def vegifunct():
```

```
    print("I am going to cook " + x + " in the evening.")
```

```
>>> vegifunct()
```

I am going to cook radish in the evening.

```
>>>
```

We will talk more about functions in the coming chapters.

Python Strings

Now that you have hopefully grasped the concept of Python variables, I will move on to explaining Python strings. To understand something, you first have to be able to identify it. Strings can be identified by the single and double quotation marks that surround them. Let's proceed with working out strings in the Python shell.

```
>>> x = "I am going to cook in the evening."
```

```
>>> print(x)
```

I am going to cook in the evening.

```
>>> x = 'I am going to cook in the evening.'
```

```
>>> print(x)
```

I am going to cook in the evening.

```
>>>
```

Strings don't discriminate with the quotation marks. For strings, single quotation marks are the same as double quotation marks. You already know that you can

assign a string to a variable.

Multiline Strings

You can spread a string to multiple lines instead of a single one. For the purpose, you will have to use triple quotation marks.

```
>>> x = """I get back at 5 from the office.
```

```
Then I take a shower and have lunch.
```

```
I go on studying for a while.
```

```
In the evening I go out for a walk.
```

```
By 11 I am fast asleep in the comforts of my bedroom."""
```

```
>>> print(x)
```

```
I get back at 5 from the office.
```

```
Then I take a shower and have lunch.
```

```
I go on studying for a while.
```

```
In the evening I go out for a walk.
```

```
By 11 I am fast asleep in the comforts of my bedroom.
```

```
>>> x = "I get back at 5 from the office.
```

```
Then I take a shower and have lunch.
```

```
I go on studying for a while.
```

```
In the evening I go out for a walk.
```

```
By 11 I am fast asleep in the comforts of my bedroom."
```

```
>>> print(x)
```

```
I get back at 5 from the office.
```

```
Then I take a shower and have lunch.
```

```
I go on studying for a while.
```

```
In the evening I go out for a walk.
```

```
By 11 I am fast asleep in the comforts of my bedroom.
```

```
>>>
```

Here too, single quotation marks can be interchangeably used with double quotation marks. Strings basically are arrays that are represented by Unicode characters. You can find out the Unicode numbers by the following method.

```
>>> x = 'I am going to cook in the evening.'
```

```
>>> print(x[5])
```

g

You can slice through strings and divide the syntax. This can be helpful if you want to view a specific number of characters in the list in the midst of programming. Let's demonstrate it in the Python shell and see how it is done.

```
>>> x = "I get back at 5 from the office. Then I take a shower and have lunch. I  
go on studying for a while. In the evening I go out for a walk. By 11 I am fast  
asleep in the comforts of my bedroom."
```

```
>>> print(x[5:7])
```

b

```
>>> print(x[10:50])
```

at 5 from the office. Then I take a sho

```
>>> print(x[1:60])
```

get back at 5 from the office. Then I take a shower and ha

```
>>> print(x[15:70])
```

from the office. Then I take a shower and have lunch.

```
>>> print(x[10:150])
```

at 5 from the office. Then I take a shower and have lunch. I go on studying for a
while. In the evening I go out for a walk. By 11 I am fas

```
>>>
```

You can test negative indexing when it comes to strings. Let's see how to do that.

```
>>> x = "I get back at 5 from the office. Then I take a shower and have lunch. I  
go on studying for a while. In the evening I go out for a walk. By 11 I am fast  
asleep in the comforts of my bedroom."
```

```
>>> print(x[-70:-5])
```

out for a walk. By 11 I am fast asleep in the comforts of my bed

```
>>> print(x[-100:-10])
```

r a while. In the evening I go out for a walk. By 11 I am fast asleep in the comforts of m

```
>>>
```

You can go on to check the length of the string with the following function. Let's see how to do that.

```
>>> x = "I get back at 5 from the office. Then I take a shower and have lunch. I go on studying for a while. In the evening I go out for a walk. By 11 I am fast asleep in the comforts of my bedroom."
```

```
>>> print(len(x))
```

```
189
```

```
>>>
```

We have Python methods to further streamline the process of programming.

```
>>> x = "        I get        back at 5 from the office. Then I take a shower and have        lunch. I go on studying for a while. In the evening I go out        for a walk. By 11 I am fast asleep in the comforts of my        bedroom.        "
```

```
>>> print(x.strip()) # this method removes any whitespaces from the start and the end of the string.
```

```
I get        back at 5 from the office. Then I take a shower and have        lunch. I go on studying for a while. In the evening I go out        for a walk. By 11 I am fast asleep in the comforts of my        bedroom.
```

```
>>>
```

You can observe that Python shell has removed the whitespaces that were in the start and the end of the string. One important thing to be noted is that this strip method strips the string of only the whitespaces from the start and end of the string, and leaves the whitespaces intact that are inside the strings. I have deliberately left some big whitespaces inside the string that are returned intact when I execute the function. Let's further explore the world of strings. If you are a game-lover, you must have seen the messages that are displayed during

playthrough. These messages can be a way for the player to communicate with the game. You can choose how to display these messages by trying different methods on the strings. Now I will look at the options I have for making your game messages sprightly and interactive.

```
>>> x = "I get back at 5 from the office. Then I take a shower and have lunch. I go on studying for a while. In the evening I go out for a walk. By 11 I am fast asleep in the comforts of my bedroom."
```

```
>>> print(x.lower())
```

```
i get back at 5 from the office. then i take a shower and have lunch. i go on studying for a while. in the evening i go out for a walk. by 11 i am fast asleep in the comforts of my bedroom.
```

```
>>> print(x.upper())
```

```
I GET BACK AT 5 FROM THE OFFICE. THEN I TAKE A SHOWER AND HAVE LUNCH. I GO ON STUDYING FOR A WHILE. IN THE EVENING I GO OUT FOR A WALK. BY 11 I AM FAST ASLEEP IN THE COMFORTS OF MY BEDROOM.
```

We have a split method that splits a string along the commas that are present in the text. Let's see how to dissect the string.

```
>>> x = "I get back at 5 from the office, Then I take a shower and have lunch, I go on studying for a while. In the evening I go out for a walk, By 11 I am fast asleep in the comforts of my bedroom."
```

```
>>> print(x.split(","))
```

```
['I get back at 5 from the office', ' Then I take a shower and have lunch', ' I go on studying for a while. In the evening I go out for a walk', ' By 11 I am fast asleep in the comforts of my bedroom.']
```

```
>>>
```

There is another interesting aspect of the Python string. You can check for any item to see if it is present in the string or not. This is helpful when you are creating lengthy messages and want to check for a specific word that you wanted to include in the string for your players to see. There is a specific method for the purpose. Let's see how to do that.

```
>>> x = "I get back at 5 from the office. Then I take a shower and have lunch. I go on studying for a while. In the evening I go out for a walk. By 11 I am fast
```

asleep in the comforts of my bedroom."

```
>>> y = "walk" in x
```

```
>>> print(y)
```

True

```
>>> y = "fast" in x
```

```
>>> print(y)
```

True

```
>>>
```

You can also go on and check if a certain word is not in the string. That's the opposite of what we did in the latest method.

```
>>> x = "I get back at 5 from the office. Then I take a shower and have lunch. I  
go on studying for a while. In the evening I go out for a walk. By 11 I am fast  
asleep in the comforts of my bedroom."
```

```
>>> y = "snazzy" not in x
```

```
>>> print(y)
```

True

```
>>> y = "bedroom" not in x
```

```
>>> print(y)
```

False

```
>>>
```

String Concatenation

String concatenation is all about combining two strings by using a simple + operator. Let's do that.

```
>>> x = "I get back at 5 from the office. Then I take a shower and have lunch. I  
go on studying for a while. In the evening I go out for a walk. By 11 I am fast  
asleep in the comforts of my bedroom."
```

```
>>> y = "Last but not least is that I don't forget to read a book while I am on the  
bed."
```

```
>>> z = x + y
```

```
>>> print(z)
```

I get back at 5 from the office. Then I take a shower and have lunch. I go on studying for a while. In the evening I go out for a walk. By 11 I am fast asleep in the comforts of my bedroom. Last but not least is that I don't forget to read a book while I am on the bed.

```
>>> z = x + " " + y
```

```
>>> print(z)
```

I get back at 5 from the office. Then I take a shower and have lunch. I go on studying for a while. In the evening I go out for a walk. By 11 I am fast asleep in the comforts of my bedroom. Last but not least is that I don't forget to read a book while I am on the bed.

```
>>>
```

In the last code snippet, I added a white space and then concatenated the two strings. You can do a lot with Python strings like the following. The following capitalize method capitalizes just the first letter of the string.

```
>>> x = "I get back at 5 from the office. Then I take a shower and have lunch. I go on studying for a while. In the evening I go out for a walk. By 11 I am fast asleep in the comforts of my bedroom."
```

```
>>> y = x.capitalize()
```

```
>>> print(y)
```

I get back at 5 from the office. then i take a shower and have lunch. i go on studying for a while. in the evening i go out for a walk. by 11 i am fast asleep in the comforts of my bedroom.

```
>>>
```

The next string method converts upper case in the string into lower case, hence the keyword casefold. I am going to use the same string and see what happens when we print it.

```
>>> x = "I Get Back at 5 from The Office. Then I take a shower and have lunch. I go on Studying For a while. In the evening I go out for a walk. By 11 I am fast asleep in the comforts of my bedroom."
```

```
>>> y = x.casefold()
```

```
>>> print(x)
```

I Get Back at 5 from The Office. Then I take a shower and have lunch. I go on Studying For a while. In the evening I go out for a walk. By 11 I am fast asleep in the comforts of my bedroom.

```
>>> print(y)
```

i get back at 5 from the office. then i take a shower and have lunch. i go on studying for a while. in the evening i go out for a walk. by 11 i am fast asleep in the comforts of my bedroom.

```
>>>
```

Chapter 4

Python Loops

Loops are an important function in Python. The for loop that is focused on the keyword for is usually used for iteration. Iteration means that it loops through a sequence that usually is a tuple, a list or a dictionary. In addition, it can loop through a string or a set of items and display them accordingly on the Python shell. In Python, the for keyword is different from other programming languages. Its tendency to operate like an iterator places Python among object-oriented programming languages. And of course, owing to its ease of use, Python tends to stay at the top among all.

Using For Loop

Let's see how to use the for keyword with different data types like tuple, list and set, etc.

```
general_ammunition = ['revolver', 'AK-47', 'machine gun', 'tank', 'bunker',  
'pistol','shotgun', 'handgun', 'magazines', 'hand grenades', 'sub machine gun',  
'rocket launcher', 'bazooka', 'steel stick', 'EM bombs', 'steel gloves']
```

```
for y in general_ammunition:
```

```
    print(y)
```

```
=====
```

RESTART:

D:/python

```
programs/PythonProgram.py =====
```

```
revolver
```

```
AK-47
```

```
machine gun
```

```
tank
```

```
bunker
```

```
pistol
```

```
shotgun
```

```
handgun
```

```
magazines
```

```
hand grenades
```

```
sub machine gun
```

```
rocket launcher
```

```
bazooka
```

```
steel stick
```

```
EM bombs
```

```
steel gloves
```

```
>>>
```

If we convert the same list into a string by removing square brackets and inserting double quotation marks.

```
for y in ["revolver", "AK-47", "machine gun", "tank", "bunker", "pistol", "shotgun",
"handgun", "magazines", "hand grenades", "sub machine gun", "rocket launcher",
"bazooka", "steel stick", "EM bombs", "steel gloves"]:
```

```
    print(y)
```

```
===== RESTART: D:/python programs/PythonProgram.py
=====
```

```
,
```

```
r
```

e
v
o
l
v
e
r
,
,
,
A
K
-
4
7
,
,
,
m
a
c
h
i
n
e

g
u
n
,
,
,
t
a
n
k
,
,
,
b
u
n
k
e
r
,
,
p
i
s

t
o
l
,
,
s
h
o
t
g
u
n
,
,
,
h
a
n
d
g
u
n
,
,
,

m

a

g

a

z

i

n

e

s

,

,

,

h

a

n

d

g

r

e

n

a

d

e

s

,

,

'
s
u
b

m
a
c
h
i
n
e

g
u
n
,

,

,

r
o
c
k
e
t

l
a

u
n
c
h
e
r
,
,
,
b
a
z
o
o
k
a
,
,
,
s
t
e
e
l
s

t
i
c
k
,
,
,
E
M

b
o
m
b
s
,
,
,
s
t
e
e
l

g
l
o

```
v
e
s
,

>>>
```

(Python For Loops, n.d)

It individually read each word of the string and displayed it accordingly. Coming back to using for loops in the list. We have seen how we can iterate through each item of the list. Sometimes we have to stop at a particular point to break the loop and display the items that have been iterated by the loop. Let's see how to do that.

```
general_ammunition = ['revolver', 'AK-47', 'machine gun', 'tank', 'bunker',
'pistol','shotgun', 'handgun', 'magazines', 'hand grenades', 'sub machine gun',
'rocket launcher', 'bazooka', 'steel stick', 'EM bombs', 'steel gloves']
```

```
print(general_ammunition)
```

```
for y in general_ammunition:
```

```
    print(y)
```

```
    if y == "sub machine gun":
```

```
        break
```

```
for y in general_ammunition:
```

```
    print(y)
```

```
    if y == "machine gun":
```

```
        break
```

Let's run the program and see how the information is displayed to the user.

```
===== RESTART:  D:/python  programs/PythonProgram.py
=====
```

```
['revolver', 'AK-47', 'machine gun', 'tank', 'bunker', 'pistol', 'shotgun', 'handgun',
'magazines', 'hand grenades', 'sub machine gun', 'rocket launcher', 'bazooka',
'steel stick', 'EM bombs', 'steel gloves']
```

```
revolver
```

AK-47

machine gun

tank

bunker

pistol

shotgun

handgun

magazines

hand grenades

sub machine gun

>>>

```
===== RESTART: D:/python programs/PythonProgram.py  
=====
```

```
['revolver', 'AK-47', 'machine gun', 'tank', 'bunker', 'pistol', 'shotgun', 'handgun',  
'magazines', 'hand grenades', 'sub machine gun', 'rocket launcher', 'bazooka',  
'steel stick', 'EM bombs', 'steel gloves']
```

revolver

AK-47

machine gun

>>>

I have broken the loop at two different points to see how efficiently it works. If you can see, I have used the break keyword after the print statement. This might confuse some readers and they may start pondering over how Python printed a list while still applying the break statement. Thankfully, Python allows us to clear our minds on this. In the following code snippet, I will take the break keyword from that place and put it before the print statement. If you code like this, it will allow you and your users alike to read your code in an easy way. There is no impact on the results of the program. Let's demonstrate.

```
general_ammunition = ['revolver', 'AK-47', 'machine gun', 'tank', 'bunker',  
'pistol', 'shotgun', 'handgun', 'magazines', 'hand grenades', 'sub machine gun',  
'rocket launcher', 'bazooka', 'steel stick', 'EM bombs', 'steel gloves']
```

```

print(general_ammunition)
for y in general_ammunition:
    if y == "sub machine gun":
        break
    print(y)

```

```

===== RESTART: D:/python programs/PythonProgram.py
=====

```

```

['revolver', 'AK-47', 'machine gun', 'tank', 'bunker', 'pistol', 'shotgun', 'handgun',
'magazines', 'hand grenades', 'sub machine gun', 'rocket launcher', 'bazooka',
'steel stick', 'EM bombs', 'steel gloves']

```

```

revolver
AK-47
machine gun
tank
bunker
pistol
shotgun
handgun
magazines
hand grenades
>>>

```

Beside the break statement we have the continue statement that has the power to stop the loop from iterating and make it continue with the next statement. A practical demonstration will allow you to understand it in a better way.

```

general_ammunition = ['revolver', 'AK-47', 'machine gun', 'tank', 'bunker',
'pistol','shotgun', 'handgun', 'magazines', 'hand grenades', 'sub machine gun',
'rocket launcher', 'bazooka', 'steel stick', 'EM bombs', 'steel gloves']

print(general_ammunition)

for y in general_ammunition:

```

```

        if y == "sub machine gun":
            continue
        print(y)
===== RESTART: D:/python programs/PythonProgram.py
=====
['revolver', 'AK-47', 'machine gun', 'tank', 'bunker', 'pistol', 'shotgun', 'handgun',
'magazines', 'hand grenades', 'sub machine gun', 'rocket launcher', 'bazooka',
'steel stick', 'EM bombs', 'steel gloves']
revolver
AK-47
machine gun
tank
bunker
pistol
shotgun
handgun
magazines
hand grenades
rocket launcher
bazooka
steel stick
EM bombs
steel gloves
>>>

```

If you take a closer look, you can see that the item sub machine gun is missing from the formatted list that has been displayed on the Python shell. The continue statement reaches the item that has been mentioned in the code. It breaks the loop there, skips the item, and then continues with the iteration to the end of the loop. You can put it at the start or the end. Its function is going to remain the same all along.

```
general_ammunition = ['revolver', 'AK-47', 'machine gun', 'tank', 'bunker',  
'pistol', 'shotgun', 'handgun', 'magazines', 'hand grenades', 'sub machine gun',  
'rocket launcher', 'bazooka', 'steel stick', 'EM bombs', 'steel gloves']
```

```
print(general_ammunition)
```

```
for y in general_ammunition:
```

```
    if y == "revolver":
```

```
        continue
```

```
        print(y)
```

```
===== RESTART: D:/python programs/PythonProgram.py  
=====
```

```
['revolver', 'AK-47', 'machine gun', 'tank', 'bunker', 'pistol', 'shotgun', 'handgun',  
'magazines', 'hand grenades', 'sub machine gun', 'rocket launcher', 'bazooka',  
'steel stick', 'EM bombs', 'steel gloves']
```

```
AK-47
```

```
machine gun
```

```
tank
```

```
bunker
```

```
pistol
```

```
shotgun
```

```
handgun
```

```
magazines
```

```
hand grenades
```

```
sub machine gun
```

```
rocket launcher
```

```
bazooka
```

```
steel stick
```

```
EM bombs
```

```
steel gloves
```

```
>>>
```

The continue loop skipped the first item and continued with the second item. Now let's put it at the end of the list.

```
general_ammunition = ['revolver', 'AK-47', 'machine gun', 'tank', 'bunker',  
'pistol','shotgun', 'handgun', 'magazines', 'hand grenades', 'sub machine gun',  
'rocket launcher', 'bazooka', 'steel stick', 'EM bombs', 'steel gloves']
```

```
print(general_ammunition)
```

```
for y in general_ammunition:
```

```
    if y == "steel gloves":
```

```
        continue
```

```
    print(y)
```

```
===== RESTART: D:/python programs/PythonProgram.py  
=====
```

```
['revolver', 'AK-47', 'machine gun', 'tank', 'bunker', 'pistol', 'shotgun', 'handgun',  
'magazines', 'hand grenades', 'sub machine gun', 'rocket launcher', 'bazooka',  
'steel stick', 'EM bombs', 'steel gloves']
```

```
revolver
```

```
AK-47
```

```
machine gun
```

```
tank
```

```
bunker
```

```
pistol
```

```
shotgun
```

```
handgun
```

```
magazines
```

```
hand grenades
```

```
sub machine gun
```

```
rocket launcher
```

```
bazooka
```

```
steel stick
```

EM bombs

>>>

Python for loop offers more flexibility of programming to users by integrating in itself the else statement. The else keyword is an extra block of code in the for loop that is executed when the loop ends.

```
general_ammunition = ['revolver', 'AK-47', 'machine gun', 'tank', 'bunker',  
'pistol','shotgun', 'handgun', 'magazines', 'hand grenades', 'sub machine gun',  
'rocket launcher', 'bazooka', 'steel stick', 'EM bombs', 'steel gloves']
```

```
print(general_ammunition)
```

```
for y in general_ammunition:
```

```
    print(y)
```

```
else:
```

```
    print("We are running out of ammunications.")
```

```
===== RESTART: D:/python programs/PythonProgram.py  
=====
```

```
['revolver', 'AK-47', 'machine gun', 'tank', 'bunker', 'pistol', 'shotgun', 'handgun',  
'magazines', 'hand grenades', 'sub machine gun', 'rocket launcher', 'bazooka',  
'steel stick', 'EM bombs', 'steel gloves']
```

```
revolver
```

```
AK-47
```

```
machine gun
```

```
tank
```

```
bunker
```

```
pistol
```

```
shotgun
```

```
handgun
```

```
magazines
```

```
hand grenades
```

```
sub machine gun
```

rocket launcher

bazooka

steel stick

EM bombs

steel gloves

We are running out of ammunitions.

>>>

Also, we can nest one for loop inside the other for loop. The loop that is inside the other for loop gets executed each time for all iterations of the main for loop.

```
qualities = ["juicy, farm fresh, clean, cheap"]
```

```
vegetables_fruits = ['pumpkin', 'garlic', 'ginger', 'tomato', 'potato', 'radish', 'carrot',  
'brinjal', 'pepper', 'onion', 'banana', 'oranges', 'figs', 'watermelon', 'melon', 'blue  
berry']
```

```
for y in qualities:
```

```
    for x in vegetables_fruits:
```

```
        print(y, x)
```

```
===== RESTART: D:/python programs/PythonProgram.py  
=====
```

```
juicy, farm fresh, clean, cheap pumpkin
```

```
juicy, farm fresh, clean, cheap garlic
```

```
juicy, farm fresh, clean, cheap ginger
```

```
juicy, farm fresh, clean, cheap tomato
```

```
juicy, farm fresh, clean, cheap potato
```

```
juicy, farm fresh, clean, cheap radish
```

```
juicy, farm fresh, clean, cheap carrot
```

```
juicy, farm fresh, clean, cheap brinjal
```

```
juicy, farm fresh, clean, cheap pepper
```

```
juicy, farm fresh, clean, cheap onion
```

juicy, farm fresh, clean, cheap banana

juicy, farm fresh, clean, cheap oranges

juicy, farm fresh, clean, cheap figs

juicy, farm fresh, clean, cheap watermelon

juicy, farm fresh, clean, cheap melon

juicy, farm fresh, clean, cheap blue berry

>>>

(Python For Loops, n.d)

Chapter 5

Tuples and Lists

This chapter will walk you through what Python tuples and lists are about and what flexibility tuples can offer you when it comes to coding and creating programs. So similar looking are both that they can be easily mixed into one another. The only difference is that Python tuples are enclosed within round brackets. You can use tuples and lists for a lot of purposes. Their usage depends on your priorities as a programmer and the requirements for creating a certain program or game.

Python Tuples

Python tuples are collections that are usually in ordered form. One particular attribute of Python tuples is that you cannot change them once they have been created. Let's jump to creating the tuples.

```
>>> ourtuple = ("guns", "toys", "horses", "swords", "knives", "balls", "beds",  
"wood", "metal", "petal")  
>>> print(ourtuple)  
(  
'guns', 'toys', 'horses', 'swords', 'knives', 'balls', 'beds', 'wood', 'metal', 'petal')  
>>>
```

If you want to access different items from the tuple, you can easily do that by using the index number that Python shell gives each item in the tuple.

```
>>> ourtuple = ("guns", "toys", "horses", "swords", "knives", "balls", "beds",  
"wood", "metal", "petal")
```

```
>>> print(ourtuple[0])
```

```
guns
```

```
>>> print(ourtuple[5])
```

```
balls
```

```
>>> print(ourtuple[4])
```

```
knives
```

```
>>> print(ourtuple[2])
```

```
horses
```

```
>>> print(ourtuple[-1])
```

```
petal
```

```
>>> print(ourtuple[-5])
```

```
balls
```

```
>>> print(ourtuple[-6])
```

```
knives
```

```
>>> print(ourtuple[-10])
```

```
guns
```

```
>>> print(ourtuple[-11])
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#14>", line 1, in <module>
```

```
    print(ourtuple[-11])
```

```
IndexError: tuple index out of range
```

```
>>> print(ourtuple[-8])
```

```
horses
```

```
>>>
```

You can see that you can use positive as well as negative indexing to bring out items from the tuple. Now let's see how can we insert the range function to bring out a set of items that you want instead of printing individual items. The line code that returned the error is also for learning purposes. It happened because I entered the index number that didn't exist in the tuple. Coming back to the range function, let's see how it is done in the Python shell.

```
>>> ourtuple = ("guns", "toys", "horses", "swords", "knives", "balls", "beds",  
"wood", "metal", "petal")  
>>> print(ourtuple[3:6])  
( 'swords', 'knives', 'balls' )  
>>> print(ourtuple[1:8])  
( 'toys', 'horses', 'swords', 'knives', 'balls', 'beds', 'wood' )  
>>>
```

We can use negative indexing in the range function.

```
>>> ourtuple = ("guns", "toys", "horses", "swords", "knives", "balls", "beds",  
"wood", "metal", "petal")  
>>> print(ourtuple[-6:-1])  
( 'knives', 'balls', 'beds', 'wood', 'metal' )  
>>> print(ourtuple[-10:-1])  
( 'guns', 'toys', 'horses', 'swords', 'knives', 'balls', 'beds', 'wood', 'metal' )  
>>>
```

I have already told you that you cannot change values of tuples once you have created them. But, there is a way around. You can actually edit a tuple but only by navigating it through a method. Let's see what that method is and how can you actually edit a tuple and change its values.

```
>>> ourtuple = ("guns", "toys", "horses", "swords", "knives", "balls", "beds",  
"wood", "metal", "petal")  
>>> x = list(ourtuple)  
>>> x[0] = "cherry"  
>>> ourtuple = tuple(x)
```

```

>>> print(ourtuple)
('cherry', 'toys', 'horses', 'swords', 'knives', 'balls', 'beds', 'wood', 'metal', 'petal')
>>> x = list(ourtuple)
>>> x[2] = "cherry"
>>> ourtuple = tuple(x)
>>> print(ourtuple)
('cherry', 'toys', 'cherry', 'swords', 'knives', 'balls', 'beds', 'wood', 'metal', 'petal')
>>> x = list(ourtuple)
>>> x[2] = "cherry"
>>> ourtuple = tuple(x)
>>> print(ourtuple)
('cherry', 'toys', 'cherry', 'swords', 'knives', 'balls', 'beds', 'wood', 'metal', 'petal')
>>> x = list(ourtuple)
>>> x[5] = "cherry"
>>> ourtuple = tuple(x)
>>> print(ourtuple)
('cherry', 'toys', 'cherry', 'swords', 'knives', 'cherry', 'beds', 'wood', 'metal', 'petal')
>>>

```

You can see that I converted the tuple into a list, changed it, and then converted it back to a tuple. We can also loop through the Python tuple such as the following.

```

>>> ourtuple = ("guns", "toys", "horses", "swords", "knives", "balls", "beds",
"wood", "metal", "petal")
>>> for y in ourtuple:
    print(y)

```

guns

toys
horses
swords
knives
balls
beds
wood
metal
petal
>>>

You can check if any item is present in the Python tuple or not.

```
>>> ourtuple
('guns', 'toys', 'horses', 'swords', 'knives', 'balls', 'beds', 'wood', 'metal', 'petal')
>>> if "wood" in ourtuple:
    print("You should be happy now because we have plenty of wood in our
stock.")
```

You should be happy now because we have plenty of wood in our stock.

The following method helps you in determining the length of the tuple. Let's see.

```
>>> ourtuple = ("guns", "toys", "horses", "swords", "knives", "balls", "beds",
"wood", "metal", "petal")
>>> print(len(ourtuple))
10
>>>
```

You saw how I replaced items with new ones in Python lists. Let's see what happens when we try to add new items to Python tuples. If I am right and Python tuple doesn't receive any new item, the Python shell will return an error. Let's

see the output of a simple method of adding items to Python shell.

```
>>> ourtuple = ("guns", "toys", "horses", "swords", "knives", "balls", "beds",  
"wood", "metal", "petal")
```

```
>>> ourtuple[5] = "cherry"
```

Traceback (most recent call last):

```
File "<pyshell#66>", line 1, in <module>
```

```
    ourtuple[5] = "cherry"
```

TypeError: 'tuple' object does not support item assignment

```
>>>
```

You cannot add items to Python tuples. Similarly, you cannot remove items from tuples once they have been created. But you have the option to delete the entire tuple from Python. See the following method to do that.

```
>>> ourtuple = ("guns", "toys", "horses", "swords", "knives", "balls", "beds",  
"wood", "metal", "petal")
```

```
>>> del ourtuple
```

```
>>> print(ourtuple)
```

Traceback (most recent call last):

```
File "<pyshell#71>", line 1, in <module>
```

```
    print(ourtuple)
```

NameError: name 'ourtuple' is not defined

```
>>>
```

We are seeing the above mentioned error because the tuple named as ourtuple doesn't exist anymore after you have deleted it. Beware that the deletion is permanent. Once you do away with a tuple, you cannot get it back at any cost. An interesting thing about Python tuples is that you can add two tuples with the help of a plus sign. Yes, it is that simple.

```
>>> ourtuple = ("guns", "toys", "horses", "swords", "knives", "balls", "beds",  
"wood", "metal", "petal")
```

```
>>> ourtuple2 = ("jewels", "nails")
```

```
>>> ourtuple3 = ourtuple + ourtuple2
```

```
>>> print(ourtuple3)
```

```
('guns', 'toys', 'horses', 'swords', 'knives', 'balls', 'beds', 'wood', 'metal', 'petal',  
'jewels', 'nails')
```

```
>>>
```

So, Python tuples can be interesting things if you are on the verge of getting started with creating your own game in Python. Yes, they are great. Players cannot change the values of the tuple and the game goes on smoothly (Python Tuples, n.d).

Python Lists

Quite similar to a tuple in Python is the list data type. Of course, there is a difference in the creation, performance, and usage. You can use lists for different tasks like for storing information in a single place. Lists allow you to cram items ranging from a few hundred to millions. You can estimate the power of lists just by the single fact that they can store data of an entire ministry of a government. Generally, a list is a collection of items in an ordered form. You can put random items like names of vegetables or metals, or you can fill it in with organized names of items. Not a compulsion but rather a tradition with lists is that you should create their names in plural form, as your lists are most likely going to have more than one item. In addition, you will have to enclose the items inside square brackets and single quotation marks. Let's create a list and print it in the Python shell.

```
randoms = ['guns', 'toys', 'horses', 'swords', 'knives', 'balls', 'beds', 'wood', 'metal',  
'petal', 'jewels', 'nails']
```

```
print(randoms)
```

```
===== RESTART: D:/python programs/PythonProgram.py  
=====
```

```
['guns', 'toys', 'horses', 'swords', 'knives', 'balls', 'beds', 'wood', 'metal', 'petal',  
'jewels', 'nails']
```

```
>>>
```

Just like tuples, you can easily access items from a list. If you are creating a game in which a player has to sell vegetables in a given amount of time, you can

create a list of vegetables and allow the player to access them when they need it. For the purpose, you will have to specify the index number of the item and it will be accessed accordingly. Let's see how you can do that.

```
randoms = ['potato', 'tomato', 'ginger', 'garlic', 'cauliflower', 'cabbage', 'pumpkin',  
'brinjal', 'peas', 'onion', 'radish', 'carrot']
```

```
print(randoms)
```

```
===== RESTART: D:/python programs/PythonProgram.py  
=====
```

```
['potato', 'tomato', 'ginger', 'garlic', 'cauliflower', 'cabbage', 'pumpkin', 'brinjal',  
'peas', 'onion', 'radish', 'carrot']
```

```
>>>
```

Let's access different items from the list.

```
randoms = ['potato', 'tomato', 'ginger', 'garlic', 'cauliflower', 'cabbage', 'pumpkin',  
'brinjal', 'peas', 'onion', 'radish', 'carrot']
```

```
print(randoms[0])
```

```
print(randoms[1])
```

```
print(randoms[6])
```

```
print(randoms[9])
```

```
print(randoms[3])
```

```
print(randoms[2])
```

```
print(randoms[5])
```

```
print(randoms[8])
```

```
print(randoms[10])
```

```
print(randoms[7])
```

```
print(randoms[4])
```

```
===== RESTART: D:/python programs/PythonProgram.py  
=====
```

```
potato
```

```
tomato
```

```
pumpkin
onion
garlic
ginger
cabbage
peas
radish
brinjal
cauliflower
>>>
```

The output by this function is neatly formatted and ready to be presented to your users. In addition, Python list allows you to apply different string methods to make formatting more fun.

```
randoms = ['potato', 'tomato', 'ginger', 'garlic', 'cauliflower', 'cabbage', 'pumpkin',
'brinjal', 'peas', 'onion', 'radish', 'carrot']
```

```
print(randoms[0].title())
```

```
print(randoms[0].upper())
```

```
print(randoms[0].lower())
```

```
print(randoms[10].title())
```

```
print(randoms[10].upper())
```

```
print(randoms[10].lower())
```

```
print(randoms[5].title())
```

```
print(randoms[5].upper())
```

```
print(randoms[5].lower())
```

```
===== RESTART: D:/python programs/PythonProgram.py
=====
```

```
Potato
```

```
POTATO
```

potato

Radish

RADISH

radish

Cabbage

CABBAGE

cabbage

>>>

One important thing to note is that the index starts at 0. We can display the item of the lists in the title case, upper, and lower cases. It depends on our particular requirements. Python also allows negative indexing, which means that your last item will be accessed by the index -1. Let's see how negative indexing works.

```
randoms = ['potato', 'tomato', 'ginger', 'garlic', 'cauliflower', 'cabbage', 'pumpkin',  
'brinjal', 'peas', 'onion', 'radish', 'carrot']
```

```
print(randoms[-1].title())
```

```
print(randoms[-1].upper())
```

```
print(randoms[-1].lower())
```

```
print(randoms[-3].title())
```

```
print(randoms[-4].upper())
```

```
print(randoms[-5].lower())
```

```
print(randoms[-9].title())
```

```
print(randoms[-10].upper())
```

```
print(randoms[-7].lower())
```

```
print(randoms[-1])
```

```
print(randoms[-1])
```

```
print(randoms[-1])
```

```
print(randoms[-3])
```

```
print(randoms[-4])
```

```
print(randoms[-5])
print(randoms[-9])
print(randoms[-10])
print(randoms[-7])
```

```
===== RESTART: D:/python programs/PythonProgram.py
=====
```

```
Carrot
CARROT
carrot
Onion
PEAS
brinjal
Garlic
GINGER
cabbage
carrot
carrot
carrot
onion
peas
brinjal
garlic
ginger
cabbage
>>>
```

Let's see how to use a value from a list. If you want your player to shout out what item is going to sell next, you can add a simple function that will use one of

the items from the list and display the result. Let's see.

```
vegetables = ['potato', 'tomato', 'ginger', 'garlic', 'cauliflower', 'cabbage',  
'pumpkin', 'brinjal', 'peas', 'onion', 'radish', 'carrot']
```

```
txt = "I am going to sell " + vegetables[5].title() + "."
```

```
print(txt)
```

```
txt = "I am going to sell " + vegetables[3].title() + "."
```

```
print(txt)
```

```
txt = "I am going to sell " + vegetables[6].title() + "."
```

```
print(txt)
```

```
txt = "I am going to sell " + vegetables[10].title() + "."
```

```
print(txt)
```

```
txt = "I am going to sell " + vegetables[4].title() + "."
```

```
print(txt)
```

```
===== RESTART: D:/python programs/PythonProgram.py  
=====
```

```
I am going to sell Cabbage.
```

```
I am going to sell Garlic.
```

```
I am going to sell Pumpkin.
```

```
I am going to sell Radish.
```

```
I am going to sell Cauliflower.
```

```
>>>
```

(Matthes, 2016)

The output will be neat and clean, ready to be displayed to the user.

Modification, Addition and Removal of Items From Lists

Once you have created lists, you have the freedom to modify them, add items to them, and remove items from them. That's what makes lists different from

tuples. Tuples are unchangeable. Let's take our game of vegetables. If you want your player to buy and add more vegetables to his stock, you can use the add function. Similarly, the vegetables that are out of stock can be removed from the list. That's how lists can make your game immensely interactive and interesting, as players have greater freedom of play. Modification, addition, and removal of elements are also very easy-to-use functions. Let's go to the Python editor to demonstrate how you can use these functions in your games.

First of all, we will focus on modification of the list. This works in a similar way as accessing an item because of the fact that you have to mention the index number for modifying the item. This function is amazing for your game. You can give your player the choice to change a vegetable even after he has purchased it from the market.

```
vegetables = ['potato', 'tomato', 'ginger', 'garlic', 'cauliflower', 'cabbage',  
'pumpkin', 'brinjal', 'peas', 'onion', 'radish', 'carrot']  
print(vegetables)
```

```
vegetables[1] = 'lintels'  
print(vegetables)
```

```
vegetables[5] = 'apple'  
print(vegetables)
```

```
===== RESTART: D:/python programs/PythonProgram.py  
=====
```

```
['potato', 'tomato', 'ginger', 'garlic', 'cauliflower', 'cabbage', 'pumpkin', 'brinjal',  
'peas', 'onion', 'radish', 'carrot']
```

```
['potato', 'lintels', 'ginger', 'garlic', 'cauliflower', 'cabbage', 'pumpkin', 'brinjal',  
'peas', 'onion', 'radish', 'carrot']
```

```
['potato', 'lintels', 'ginger', 'garlic', 'cauliflower', 'apple', 'pumpkin', 'brinjal', 'peas',  
'onion', 'radish', 'carrot']
```

```
>>>
```

You can see how easy it is to modify certain items in a list. While modifying is fun, adding elements to the list is greater fun. Instead of modifying, you can

allow your player to buy more items to fill his stock. For example, he wants to add fruits to his stock along with vegetables for greater profit. One of the most popular methods of doing this is adding items to the end of the list. The append function will keep adding no matter how many items you want to add. Let's see how to do that.

```
vegetables = ['potato', 'tomato', 'ginger', 'garlic', 'cauliflower', 'cabbage',  
'pumpkin', 'brinjal', 'peas', 'onion', 'radish', 'carrot']  
print(vegetables)
```

```
vegetables.append('apples')  
print(vegetables)
```

```
vegetables.append('strawberry')  
print(vegetables)
```

```
vegetables.append('blueberry')  
print(vegetables)
```

```
vegetables.append('watermelon')  
print(vegetables)
```

```
vegetables.append('melon')  
print(vegetables)
```

```
vegetables.append('grapes')  
print(vegetables)
```

```
===== RESTART: D:/python programs/PythonProgram.py  
=====
```

```
['potato', 'tomato', 'ginger', 'garlic', 'cauliflower', 'cabbage', 'pumpkin', 'brinjal',  
'peas', 'onion', 'radish', 'carrot']
```

```
['potato', 'tomato', 'ginger', 'garlic', 'cauliflower', 'cabbage', 'pumpkin', 'brinjal',
```

```
'peas', 'onion', 'radish', 'carrot', 'apples']
```

```
['potato', 'tomato', 'ginger', 'garlic', 'cauliflower', 'cabbage', 'pumpkin', 'brinjal',  
'peas', 'onion', 'radish', 'carrot', 'apples', 'strawberry']
```

```
['potato', 'tomato', 'ginger', 'garlic', 'cauliflower', 'cabbage', 'pumpkin', 'brinjal',  
'peas', 'onion', 'radish', 'carrot', 'apples', 'strawberry', 'blueberry']
```

```
['potato', 'tomato', 'ginger', 'garlic', 'cauliflower', 'cabbage', 'pumpkin', 'brinjal',  
'peas', 'onion', 'radish', 'carrot', 'apples', 'strawberry', 'blueberry', 'watermelon']
```

```
['potato', 'tomato', 'ginger', 'garlic', 'cauliflower', 'cabbage', 'pumpkin', 'brinjal',  
'peas', 'onion', 'radish', 'carrot', 'apples', 'strawberry', 'blueberry', 'watermelon',  
'melon']
```

```
['potato', 'tomato', 'ginger', 'garlic', 'cauliflower', 'cabbage', 'pumpkin', 'brinjal',  
'peas', 'onion', 'radish', 'carrot', 'apples', 'strawberry', 'blueberry', 'watermelon',  
'melon', 'grapes']
```

```
>>>
```

The `append()` function can also be used to build up lists. For example, you put an empty stock in the game and ask the player to fill it himself. He can add individual items to the list by the same method. Let's create an empty list and then add items to it by the `append()` function.

```
vegetables_fruits = []
```

```
vegetables_fruits.append('pumpkin')
```

```
print(vegetables_fruits)
```

```
vegetables_fruits.append('garlic')
```

```
print(vegetables_fruits)
```

```
vegetables_fruits.append('ginger')
```

```
print(vegetables_fruits)
```

```
vegetables_fruits.append('tomato')
```

```
print(vegetables_fruits)
```

```
vegetables_fruits.append('potato')
```

```
print(vegetables_fruits)
```

```
vegetables_fruits.append('radish')
```

```
print(vegetables_fruits)
vegetables_fruits.append('carrot')
print(vegetables_fruits)
vegetables_fruits.append('brinjal')
print(vegetables_fruits)
vegetables_fruits.append('pepper')
print(vegetables_fruits)
vegetables_fruits.append('onion')
print(vegetables_fruits)
vegetables_fruits.append('banana')
print(vegetables_fruits)
vegetables_fruits.append('oranges')
print(vegetables_fruits)
vegetables_fruits.append('figs')
print(vegetables_fruits)
vegetables_fruits.append('watermelon')
print(vegetables_fruits)
vegetables_fruits.append('melon')
print(vegetables_fruits)
vegetables_fruits.append('blueberry')
print(vegetables_fruits)
```

```
===== RESTART: D:/python programs/PythonProgram.py
=====
```

```
['pumpkin']
['pumpkin', 'garlic']
['pumpkin', 'garlic', 'ginger']
['pumpkin', 'garlic', 'ginger', 'tomato']
```

```
['pumpkin', 'garlic', 'ginger', 'tomato', 'potato']  
['pumpkin', 'garlic', 'ginger', 'tomato', 'potato', 'radish']  
['pumpkin', 'garlic', 'ginger', 'tomato', 'potato', 'radish', 'carrot']  
['pumpkin', 'garlic', 'ginger', 'tomato', 'potato', 'radish', 'carrot', 'brinjal']  
['pumpkin', 'garlic', 'ginger', 'tomato', 'potato', 'radish', 'carrot', 'brinjal', 'pepper']  
['pumpkin', 'garlic', 'ginger', 'tomato', 'potato', 'radish', 'carrot', 'brinjal', 'pepper',  
'onion']  
['pumpkin', 'garlic', 'ginger', 'tomato', 'potato', 'radish', 'carrot', 'brinjal', 'pepper',  
'onion', 'banana']  
['pumpkin', 'garlic', 'ginger', 'tomato', 'potato', 'radish', 'carrot', 'brinjal', 'pepper',  
'onion', 'banana', 'oranges']  
['pumpkin', 'garlic', 'ginger', 'tomato', 'potato', 'radish', 'carrot', 'brinjal', 'pepper',  
'onion', 'banana', 'oranges', 'figs']  
['pumpkin', 'garlic', 'ginger', 'tomato', 'potato', 'radish', 'carrot', 'brinjal', 'pepper',  
'onion', 'banana', 'oranges', 'figs', 'watermelon']  
['pumpkin', 'garlic', 'ginger', 'tomato', 'potato', 'radish', 'carrot', 'brinjal', 'pepper',  
'onion', 'banana', 'oranges', 'figs', 'watermelon', 'melon']  
['pumpkin', 'garlic', 'ginger', 'tomato', 'potato', 'radish', 'carrot', 'brinjal', 'pepper',  
'onion', 'banana', 'oranges', 'figs', 'watermelon', 'melon', 'blueberry']  
>>>
```

(Matthes, 2016)

You can also insert a specific item right at the position you want it to be placed. You can do that by filling the function with the index number. Let's add some items by mentioning the index numbers in the code.

```
vegetables_fruits = ['pumpkin', 'garlic', 'ginger', 'tomato', 'potato', 'radish', 'carrot',  
'brinjal', 'pepper', 'onion', 'banana', 'oranges', 'figs', 'watermelon', 'melon',  
'blueberry']  
vegetables_fruits.insert(1, 'strawberry')  
print(vegetables_fruits)  
  
vegetables_fruits.insert(10, 'strawberry')
```

```
print(vegetables_fruits)
```

```
===== RESTART: D:/python programs/PythonProgram.py  
=====
```

```
['pumpkin', 'strawberry', 'garlic', 'ginger', 'tomato', 'potato', 'radish', 'carrot',  
'brinjal', 'pepper', 'onion', 'banana', 'oranges', 'figs', 'watermelon', 'melon',  
'blueberry']
```

```
['pumpkin', 'strawberry', 'garlic', 'ginger', 'tomato', 'potato', 'radish', 'carrot',  
'brinjal', 'pepper', 'strawberry', 'onion', 'banana', 'oranges', 'figs', 'watermelon',  
'melon', 'blueberry']
```

```
>>>
```

Now I will move on to removing certain elements from the same list. We can once again refer to the game we were talking about. When the player sells a vegetable, it should be removed from the stock. This is how you are able to add more life to your game. Rigid things just mar the game. That's why you need flexibility. There are a couple of ways to do that. One popular method is the del statement. You can use this statement if you know the index number. Let's see how to do that.

```
vegetables_fruits = ['pumpkin', 'garlic', 'ginger', 'tomato', 'potato', 'radish', 'carrot',  
'brinjal', 'pepper', 'onion', 'banana', 'oranges', 'figs', 'watermelon', 'melon',  
'blueberry']
```

```
print(vegetables_fruits)
```

```
del vegetables_fruits[0]
```

```
print(vegetables_fruits)
```

```
del vegetables_fruits[5]
```

```
print(vegetables_fruits)
```

```
del vegetables_fruits[2]
```

```
print(vegetables_fruits)
```

```
del vegetables_fruits[6]
```

```
print(vegetables_fruits)
```

```
del vegetables_fruits[7]
```

```
print(vegetables_fruits)
```

```
del vegetables_fruits[9]
```

```
print(vegetables_fruits)
```

```
del vegetables_fruits[8]
```

```
print(vegetables_fruits)
```

```
del vegetables_fruits[10]
```

```
print(vegetables_fruits)
```

```
===== RESTART: D:/python programs/PythonProgram.py  
=====
```

```
['pumpkin', 'garlic', 'ginger', 'tomato', 'potato', 'radish', 'carrot', 'brinjal', 'pepper',  
'onion', 'banana', 'oranges', 'figs', 'watermelon', 'melon', 'blueberry']
```

```
['garlic', 'ginger', 'tomato', 'potato', 'radish', 'carrot', 'brinjal', 'pepper', 'onion',  
'banana', 'oranges', 'figs', 'watermelon', 'melon', 'blueberry']
```

```
['garlic', 'ginger', 'tomato', 'potato', 'radish', 'brinjal', 'pepper', 'onion', 'banana',  
'oranges', 'figs', 'watermelon', 'melon', 'blueberry']
```

```
['garlic', 'ginger', 'potato', 'radish', 'brinjal', 'pepper', 'onion', 'banana', 'oranges',  
'figs', 'watermelon', 'melon', 'blueberry']
```

```
['garlic', 'ginger', 'potato', 'radish', 'brinjal', 'pepper', 'banana', 'oranges', 'figs',  
'watermelon', 'melon', 'blueberry']
```

```
['garlic', 'ginger', 'potato', 'radish', 'brinjal', 'pepper', 'banana', 'figs', 'watermelon',  
'melon', 'blueberry']
```

```
['garlic', 'ginger', 'potato', 'radish', 'brinjal', 'pepper', 'banana', 'figs', 'watermelon',  
'blueberry']
```

```
['garlic', 'ginger', 'potato', 'radish', 'brinjal', 'pepper', 'banana', 'figs', 'blueberry']
```

Traceback (most recent call last):

File "D:/python programs/PythonProgram.py", line 25, in <module>

```
del vegetables_fruits[10]
```

IndexError: list assignment index out of range

>>>

As I kept adding the index numbers, Python recognized them and deleted the items accordingly. In the last code when I tried to enter 10 as index number, Python returned range error because there were not enough items left in the list. The del statement permanently deleted items from the list as I directed it to do. You cannot access the items later on.

Another popular method of removing items from the list is the pop() method. This method does the same thing as the del statement. The only difference is that you can access the values even after you have removed them. Take the example of the game once again. After the player has sold the vegetables and fruits, they may be removed but can still be visible to the player so that he knows which items he had and how many he had sold. Let's see how to use the pop() method in our programs. I will be using the same list that I have used for the del statement. I will refill it before using the pop() method.

```
vegetables_fruits = ['pumpkin', 'garlic', 'ginger', 'tomato', 'potato', 'radish', 'carrot',  
'brinjal', 'pepper', 'onion', 'banana', 'oranges', 'figs', 'watermelon', 'melon',  
'blueberry']
```

```
print(vegetables_fruits)
```

```
popped_vegetables_fruits = vegetables_fruits.pop()
```

```
print(vegetables_fruits)
```

```
print(popped_vegetables_fruits)
```

```
popped_vegetables_fruits = vegetables_fruits.pop()
```

```
print(vegetables_fruits)
```

```
print(popped_vegetables_fruits)
```

```
popped_vegetables_fruits = vegetables_fruits.pop()
```

```
print(vegetables_fruits)
print(popped_vegetables_fruits)

popped_vegetables_fruits = vegetables_fruits.pop()
print(vegetables_fruits)
print(popped_vegetables_fruits)
```

```
popped_vegetables_fruits = vegetables_fruits.pop()
print(vegetables_fruits)
print(popped_vegetables_fruits)
```

```
popped_vegetables_fruits = vegetables_fruits.pop()
print(vegetables_fruits)
print(popped_vegetables_fruits)
```

```
popped_vegetables_fruits = vegetables_fruits.pop()
print(vegetables_fruits)
print(popped_vegetables_fruits)
```

```
===== RESTART: D:/python programs/PythonProgram.py
=====
```

```
['pumpkin', 'garlic', 'ginger', 'tomato', 'potato', 'radish', 'carrot', 'brinjal', 'pepper',
'onion', 'banana', 'oranges', 'figs', 'watermelon', 'melon', 'blueberry']
```

```
['pumpkin', 'garlic', 'ginger', 'tomato', 'potato', 'radish', 'carrot', 'brinjal', 'pepper',
'onion', 'banana', 'oranges', 'figs', 'watermelon', 'melon']
```

```
blueberry
```

```
['pumpkin', 'garlic', 'ginger', 'tomato', 'potato', 'radish', 'carrot', 'brinjal', 'pepper',
'onion', 'banana', 'oranges', 'figs', 'watermelon']
```

```
melon
```

```
['pumpkin', 'garlic', 'ginger', 'tomato', 'potato', 'radish', 'carrot', 'brinjal', 'pepper',
```

```
'onion', 'banana', 'oranges', 'figs']
```

```
watermelon
```

```
['pumpkin', 'garlic', 'ginger', 'tomato', 'potato', 'radish', 'carrot', 'brinjal', 'pepper',  
'onion', 'banana', 'oranges']
```

```
figs
```

```
['pumpkin', 'garlic', 'ginger', 'tomato', 'potato', 'radish', 'carrot', 'brinjal', 'pepper',  
'onion', 'banana']
```

```
oranges
```

```
['pumpkin', 'garlic', 'ginger', 'tomato', 'potato', 'radish', 'carrot', 'brinjal', 'pepper',  
'onion']
```

```
banana
```

```
['pumpkin', 'garlic', 'ginger', 'tomato', 'potato', 'radish', 'carrot', 'brinjal', 'pepper']
```

```
onion
```

```
>>>
```

Let's analyze how we did that. First of all, I defined and printed the list on which I was going to work. After that I used the pop() method on the list and each time accessed the popped item to confirm whether it was accessible or not after removal. The last to be removed from the list is onion. When it was removed, I accessed the same with a special method, and it was likewise displayed. The pop() method removes items from the end of the list. You can also pair up the pop() method with a print statement that will tell the player which items had been sold from the stock. Let's try to add the print statement in the program.

```
vegetables_fruits = ['pumpkin', 'garlic', 'ginger', 'tomato', 'potato', 'radish', 'carrot',  
'brinjal', 'pepper', 'onion', 'banana', 'oranges', 'figs', 'watermelon', 'melon',  
'blueberry']
```

```
print(vegetables_fruits)
```

```
sold_items = vegetables_fruits.pop()
```

```
print("You have run out of stock for " + sold_items.title() + ".")
```

```
sold_items = vegetables_fruits.pop()
```

```
print("You have run out of stock for " + sold_items.title() + ".")
```

```
sold_items = vegetables_fruits.pop()
```

```
print("You have run out of stock for " + sold_items.title() + ".")
```

```
sold_items = vegetables_fruits.pop()
```

```
print("You have run out of stock for " + sold_items.title() + ".")
```

```
sold_items = vegetables_fruits.pop()
```

```
print("You have run out of stock for " + sold_items.title() + ".")
```

```
sold_items = vegetables_fruits.pop()
```

```
print("You have run out of stock for " + sold_items.title() + ".")
```

```
===== RESTART: D:/python programs/PythonProgram.py  
=====
```

```
['pumpkin', 'garlic', 'ginger', 'tomato', 'potato', 'radish', 'carrot', 'brinjal', 'pepper',  
'onion', 'banana', 'oranges', 'figs', 'watermelon', 'melon', 'blueberry']
```

```
You have run out of stock for Blueberry.
```

```
You have run out of stock for Melon.
```

```
You have run out of stock for Watermelon.
```

```
You have run out of stock for Figs.
```

```
You have run out of stock for Oranges.
```

```
You have run out of stock for Banana.
```

```
>>>
```

(Matthes, 2016)

You can see that Python returns a cleanly printed statement, telling the user that a particular item has been removed from the stock. You also can use the pop() method to remove items from any positions in the list. For the purpose you have to include the index number in the method. Let's see how to do that.

```
vegetables_fruits = ['pumpkin', 'garlic', 'ginger', 'tomato', 'potato', 'radish', 'carrot',  
'brinjal', 'pepper', 'onion', 'banana', 'oranges', 'figs', 'watermelon', 'melon',  
'blueberry']
```

```
print(vegetables_fruits)
```

```
sold_items = vegetables_fruits.pop(1)
```

```
print("You have run out of stock for " + sold_items.title() + ".")
```

```
sold_items = vegetables_fruits.pop(4)
```

```
print("You have run out of stock for " + sold_items.title() + ".")
```

```
sold_items = vegetables_fruits.pop(7)
```

```
print("You have run out of stock for " + sold_items.title() + ".")
```

```
sold_items = vegetables_fruits.pop(8)
```

```
print("You have run out of stock for " + sold_items.title() + ".")
```

```
sold_items = vegetables_fruits.pop(2)
```

```
print("You have run out of stock for " + sold_items.title() + ".")
```

```
sold_items = vegetables_fruits.pop(0)
```

```
print("You have run out of stock for " + sold_items.title() + ".")
```

```
===== RESTART: D:/python programs/PythonProgram.py  
=====
```

```
['pumpkin', 'garlic', 'ginger', 'tomato', 'potato', 'radish', 'carrot', 'brinjal', 'pepper',  
'onion', 'banana', 'oranges', 'figs', 'watermelon', 'melon', 'blueberry']
```

```
You have run out of stock for Garlic.
```

```
You have run out of stock for Radish.
```

```
You have run out of stock for Onion.
```

```
You have run out of stock for Oranges.
```

You have run out of stock for Tomato.

You have run out of stock for Pumpkin.

>>>

(Matthes, 2016)

I have inserted the index number in the pop() method to remove the item of my choice and also print the statement. Once again the output comes neat and clean, perfectly tailored for the user. There is another method to remove items from the list and that is the remove() method. This method is helpful in conditions when you lose track of the index numbers of the items that are stored in the list. Let's try to remove certain items by entering their values in the remove() method.

```
vegetables_fruits = ['pumpkin', 'garlic', 'ginger', 'tomato', 'potato', 'radish', 'carrot',  
'brinjal', 'pepper', 'onion', 'banana', 'oranges', 'figs', 'watermelon', 'melon',  
'blueberry']
```

```
print(vegetables_fruits)
```

```
vegetables_fruits.remove('pumpkin')
```

```
print(vegetables_fruits)
```

```
vegetables_fruits.remove('ginger')
```

```
print(vegetables_fruits)
```

```
vegetables_fruits.remove('tomato')
```

```
print(vegetables_fruits)
```

```
===== RESTART: D:/python programs/PythonProgram.py  
=====
```

```
['pumpkin', 'garlic', 'ginger', 'tomato', 'potato', 'radish', 'carrot', 'brinjal', 'pepper',  
'onion', 'banana', 'oranges', 'figs', 'watermelon', 'melon', 'blueberry']
```

```
['garlic', 'ginger', 'tomato', 'potato', 'radish', 'carrot', 'brinjal', 'pepper', 'onion',  
'banana', 'oranges', 'figs', 'watermelon', 'melon', 'blueberry']
```

```
['garlic', 'tomato', 'potato', 'radish', 'carrot', 'brinjal', 'pepper', 'onion', 'banana',  
'oranges', 'figs', 'watermelon', 'melon', 'blueberry']
```

```
['garlic', 'potato', 'radish', 'carrot', 'brinjal', 'pepper', 'onion', 'banana', 'oranges',  
'figs', 'watermelon', 'melon', 'blueberry']
```

```
>>>
```

The code tells Python to remove certain items from the list when you name them. You can also print a reason for the remove of the item. Like the pop() method, the remove() method allows users to access the removed item and also use it for other purposes. Let's see how to do that.

```
vegetables_fruits = ['pumpkin', 'garlic', 'ginger', 'tomato', 'potato', 'radish', 'carrot',  
'brinjal', 'pepper', 'onion', 'banana', 'oranges', 'figs', 'watermelon', 'melon',  
'blueberry']
```

```
print(vegetables_fruits)
```

```
sold_item = 'garlic'
```

```
vegetables_fruits.remove(sold_item)
```

```
print(vegetables_fruits)
```

```
print("\nI have already sold " + sold_item.title() + ".")
```

```
sold_item = 'pumpkin'
```

```
vegetables_fruits.remove(sold_item)
```

```
print(vegetables_fruits)
```

```
print("\nI have already sold " + sold_item.title() + ".")
```

```
sold_item = 'ginger'
```

```
vegetables_fruits.remove(sold_item)
```

```
print(vegetables_fruits)
```

```
print("\nI have already sold " + sold_item.title() + ".")
```

```
sold_item = 'radish'
```

```
vegetables_fruits.remove(sold_item)
```

```
print(vegetables_fruits)
```

```
print("\nI have already sold " + sold_item.title() + ".")
```

```
===== RESTART: D:/python programs/PythonProgram.py  
=====
```

```
['pumpkin', 'garlic', 'ginger', 'tomato', 'potato', 'radish', 'carrot', 'brinjal', 'pepper',  
'onion', 'banana', 'oranges', 'figs', 'watermelon', 'melon', 'blueberry']
```

```
['pumpkin', 'ginger', 'tomato', 'potato', 'radish', 'carrot', 'brinjal', 'pepper', 'onion',  
'banana', 'oranges', 'figs', 'watermelon', 'melon', 'blueberry']
```

I have already sold Garlic.

```
['ginger', 'tomato', 'potato', 'radish', 'carrot', 'brinjal', 'pepper', 'onion', 'banana',  
'oranges', 'figs', 'watermelon', 'melon', 'blueberry']
```

I have already sold Pumpkin.

```
['tomato', 'potato', 'radish', 'carrot', 'brinjal', 'pepper', 'onion', 'banana', 'oranges',  
'figs', 'watermelon', 'melon', 'blueberry']
```

I have already sold Ginger.

```
['tomato', 'potato', 'carrot', 'brinjal', 'pepper', 'onion', 'banana', 'oranges', 'figs',  
'watermelon', 'melon', 'blueberry']
```

I have already sold Radish.

```
>>>
```

If you noticed, I have created a variable named `sold_items`. When I remove the items from the list, they automatically are stored in the variable. In the next lines of code, I use this variable inside a print statement to display information about the removed item.

How to Organize a List?

When you create a list in Python, it remains as such unless you try to bring it into an orderly form. In addition, if you leave the creation of lists to the users, you are at a disadvantage because you cannot control the order of the input by

users. If you want to change the order of your lists, you can do that by the following method. There are a number of ways of changing the order of Python lists. I will explore them here one by one. Let's see.

The sort() Method: The sort() method makes it easier to sort Python lists. Let's apply it to the list that we have been working on. The sort() method is supposed to order the list in alphabetical form.

```
vegetables_fruits = ['pumpkin', 'garlic', 'ginger', 'tomato', 'potato', 'radish', 'carrot', 'brinjal', 'pepper', 'onion', 'banana', 'oranges', 'figs', 'watermelon', 'melon', 'blueberry']
```

```
vegetables_fruits.sort()
```

```
print(vegetables_fruits)
```

```
===== RESTART: D:/python programs/PythonProgram.py =====
```

```
['banana', 'blueberry', 'brinjal', 'carrot', 'figs', 'garlic', 'ginger', 'melon', 'onion', 'oranges', 'pepper', 'potato', 'pumpkin', 'radish', 'tomato', 'watermelon']
```

```
>>>
```

We have transformed the list into alphabetical order. The problem with this method is that you cannot just revert the list back to the original order in which you had created it. But one advantage is that you can reorder the list in the reverse alphabetical order. Let's see how to do that.

```
vegetables_fruits = ['pumpkin', 'garlic', 'ginger', 'tomato', 'potato', 'radish', 'carrot', 'brinjal', 'pepper', 'onion', 'banana', 'oranges', 'figs', 'watermelon', 'melon', 'blueberry']
```

```
vegetables_fruits.sort(reverse=True)
```

```
print(vegetables_fruits)
```

```
===== RESTART: D:/python programs/PythonProgram.py =====
```

```
['watermelon', 'tomato', 'radish', 'pumpkin', 'potato', 'pepper', 'oranges', 'onion', 'melon', 'ginger', 'garlic', 'figs', 'carrot', 'brinjal', 'blueberry', 'banana']
```

```
>>>
```

In addition, you can also order the list on temporary basis. It will come back to its original form after a short while. You don't have to do extra coding for the

purpose.

```
vegetables_fruits = ['pumpkin', 'garlic', 'ginger', 'tomato', 'potato', 'radish', 'carrot',  
'brinjal', 'pepper', 'onion', 'banana', 'oranges', 'figs', 'watermelon', 'melon',  
'blueberry']
```

```
print("Please see the list in the original form: ")
```

```
print(vegetables_fruits)
```

```
print("\nNow you can see the list in the sorted form: ")
```

```
print(sorted(vegetables_fruits))
```

```
print("\nPlease see the list reverted back to the original form: ")
```

```
print(vegetables_fruits)
```

```
===== RESTART: D:/python programs/PythonProgram.py  
=====
```

Please see the list in the original form:

```
['pumpkin', 'garlic', 'ginger', 'tomato', 'potato', 'radish', 'carrot', 'brinjal', 'pepper',  
'onion', 'banana', 'oranges', 'figs', 'watermelon', 'melon', 'blueberry']
```

Now you can see the list in the sorted form:

```
['banana', 'blue berry', 'brinjal', 'carrot', 'figs', 'garlic', 'ginger', 'melon', 'onion',  
'oranges', 'pepper', 'potato', 'pumpkin', 'radish', 'tomato', 'watermelon']
```

Please see the list reverted back to the original form:

```
['pumpkin', 'garlic', 'ginger', 'tomato', 'potato', 'radish', 'carrot', 'brinjal', 'pepper',  
'onion', 'banana', 'oranges', 'figs', 'watermelon', 'melon', 'blueberry']
```

```
>>>
```

Regardless of the alphabetical order, you can set the list on the reverse course.

```
vegetables_fruits = ['pumpkin', 'garlic', 'ginger', 'tomato', 'potato', 'radish', 'carrot',  
'brinjal', 'pepper', 'onion', 'banana', 'oranges', 'figs', 'watermelon', 'melon',  
'blueberry']
```

```
print(vegetables_fruits)
```

```
vegetables_fruits.reverse()
```

```
print(vegetables_fruits)
```

```
===== RESTART: D:/python programs/PythonProgram.py  
=====
```

```
['pumpkin', 'garlic', 'ginger', 'tomato', 'potato', 'radish', 'carrot', 'brinjal', 'pepper',  
'onion', 'banana', 'oranges', 'figs', 'watermelon', 'melon', 'blueberry']
```

```
['blueberry', 'melon', 'watermelon', 'figs', 'oranges', 'banana', 'onion', 'pepper',  
'brinjal', 'carrot', 'radish', 'potato', 'tomato', 'ginger', 'garlic', 'pumpkin']
```

```
>>>
```

This method simply reverses the list regardless of the order of the list. Lists are amazing but sometimes you may get errors in return when you try to locate an item by indexing. The problem with accessing items from the list through indexing is that you just cannot remember all the indices. Let's see the type of error that you can get when you put in the wrong index.

```
vegetables_fruits = ['pumpkin', 'garlic', 'ginger', 'tomato', 'potato', 'radish', 'carrot',  
'brinjal', 'pepper', 'onion', 'banana', 'oranges', 'figs', 'watermelon', 'melon',  
'blueberry']
```

```
print(vegetables_fruits[2])
```

```
print(vegetables_fruits[15])
```

```
print(vegetables_fruits[20])
```

```
===== RESTART: D:/python programs/PythonProgram.py  
=====
```

```
ginger
```

```
blueberry
```

```
Traceback (most recent call last):
```

```
File "D:/python programs/PythonProgram.py", line 4, in <module>
```

```
    print(vegetables_fruits[20])
```

```
IndexError: list index out of range
```

```
>>>
```

A similar type of error can happen when you try to access items from the list through negative indexing.

```
vegetables_fruits = ['pumpkin', 'garlic', 'ginger', 'tomato', 'potato', 'radish', 'carrot',  
'brinjal', 'pepper', 'onion', 'banana', 'oranges', 'figs', 'watermelon', 'melon',  
'blueberry']
```

```
print(vegetables_fruits[-1])
```

```
print(vegetables_fruits[-20])
```

```
===== RESTART: D:/python programs/PythonProgram.py  
=====
```

```
blueberry
```

```
Traceback (most recent call last):
```

```
File "D:/python programs/PythonProgram.py", line 3, in <module>
```

```
    print(vegetables_fruits[-20])
```

```
IndexError: list index out of range
```

```
>>>
```

You can avoid this error by printing the list first, checking the indices, and then entering them in the code line (Matthes, 2016).

Chapter 6

Python Input Function

Programmers are mostly handed tasks for development of certain programs that would solve the problem of an end user. To accomplish the job, you need information from the user's end. If you want to build a program that will take a user's age as input and return the message of whether he qualifies for a driving license or not, you will be needing information from the user. The program will ask the user to fill in the program with his or her age, and will return the message that you feed to the system. The information that the user will provide is called input. That's where we will be needing the `input()` function to come into play. This chapter will explain how you can use the `input()` function to build programs. The chapter will walk you through the development of real programs. You will learn about the while loops that play a crucial role when it comes to controlling the flow of programs.

The `input()` Function

The `input()` function requests the user to put in information in the form of text or numbers. Python gets the information and immediately stores it in a variable so that you can work on that information and also use it later on. Let's create a program that takes some information from the user and displays it back. It is like an echo. Time to move on to Python editor, where we will write this program

and then run to have the display in the Python shell.

```
txt = input("This program takes information from the user and displays the same  
back to him. Please enter what you want to say: ")
```

```
print(txt)
```

```
===== RESTART: D:/python programs/a.py  
=====
```

```
This program takes information from the user and displays the same back to him.  
Please enter what you want to say: I am learning Python programming
```

```
I am learning Python programming
```

```
>>>
```

```
===== RESTART: D:/python programs/a.py  
=====
```

```
>>>
```

```
This program takes information from the user and displays the same back to him.  
Please enter what you want to say: Do you repeat what I will tell you to?
```

```
Do you repeat what I will tell you to?
```

```
>>>
```

Our program takes a single argument. You can enter whatever information you want the program to repeat, and it will do exactly that. I have tried twice and received the same piece of text that I entered when the shell prompt asked me to enter information. If you notice, you can understand that the program runs and stops at the prompt where it demands information from the user. It remains paused until the user enters some information. When the information has been passed on to the program, the program once again starts running. Please remember that you have to press ENTER once you have entered information. Only then will the shell prompt pass on the information.

Let's create more interesting programs to get a better knowhow of the input() function. One thing to keep in mind is that we should write clearly when creating a prompt. Clear writing means that what we want to ask from the user should be spelled and structured clearly. In addition, we should also take care while handling whitespaces. Let's create a program that can be installed at a casino's virtual reception. Users will be prompted to enter their names and in return a welcome message will be displayed by the casino reception.

```
txt = input("Will you please fill in your name please?: ")
print("Dear, " + txt.upper() + " I welcome you on behalf of our casino's
management. Have a good evening!")
```

```
===== RESTART: D:/python programs/a.py
=====
```

Will you please fill in your name please?: Harry

Dear, HARRY I welcome you on behalf of our casino's management. Have a good evening!

>>>

All went well except for the clarity of the message that I displayed in the program. You can spot the comma after the word Dear. In an ideal situation there should an exclamation mark after Harry. Let's fix this.

```
txt = input("Will you please fill in your name please?: ")
print("Dear " + txt.upper() + "! I welcome you on behalf of our casino's
management. Have a good evening!")
```

```
===== RESTART: D:/python programs/a.py
=====
```

Will you please fill in your name please?: Harry

Dear HARRY! I welcome you on behalf of our casino's management. Have a good evening!

>>>

Still the display of the name in all capital seems awkward, so we will have to fix it. One important thing to note is that your user can enter his or her name in capital or lower case. Our program will handle it.

```
txt = input("Will you please fill in your name please?: ")
print("Dear " + txt.title() + "! I welcome you on behalf of our casino's
management. Have a good evening!")
```

```
===== RESTART: D:/python programs/a.py
=====
```

Will you please fill in your name please?: harry

Dear Harry! I welcome you on behalf of our casino's management. Have a good

evening!

>>>

Please see that whenever the prompt happens in the shell, you see space between the text message and the name I enter. This happens because when writing the program, I left a whitespace to the end of the text just before the closing double quotation marks. Let's move on to create a program that includes more than one string. I'm going to write the program first and explain it later on.

```
csntxt = "We are honored to have you as guest player at our casino."
```

```
csntxt += "\nWe hope you will get the best experience with us."
```

```
csntxt += ("\nWill you please fill in your name please?: ")
```

```
name = input(csntxt)
```

```
print("\nDear " + name + "! I welcome you on behalf of our casino's  
management. Have a good evening!")
```

```
===== RESTART: D:/python programs/a.py  
=====
```

We are honored to have you as guest player at our casino.

We hope you will get the best experience with us.

Will you please fill in your name please?: harry

Dear harry! I welcome you on behalf of our casino's management. Have a good evening!

>>>

You can see how easy it is to add multiple strings to Python programs. I have stored all the lines in a single variable and separated the lines by adding `\n`. In addition, the operate changed from `=` to `+=` from the second line. This operator helps add the new string to the end of the previous string. This is how you can add as many strings as you want to.

The int() Method

The input() function is created as such to interpret everything that the user enters as a string. Let's try to feed the input() function with numbers and see what can be the possible output. I will create a program in the editor that will prompt the user to enter his or age to qualify for a driving license.

```
age = input("Please enter your age to check whether you qualify for a driving
license or not?")
```

```
===== RESTART: D:/python programs/a.py
=====
```

```
>>>
```

```
Please enter your age to check whether you qualify for a driving license or not?
```

```
>>> 45
```

```
45
```

```
>>>
```

The program will display whatever age number you enter. One important thing to note is that when you run the program from the Python editor, you will enter Python shell. Your input message will be displayed in the Python shell. Now press ENTER to start the shell prompt and enter your age. Still, we have a serious problem. We are only getting the display of the number that we are entering in the shell prompt. Let's see what happens when we try to use numbers as input.

```
age = input("Please enter your age to check whether you qualify for a driving
license or not?")
```

```
age >= 20
```

```
===== RESTART: D:/python programs/a.py
=====
```

```
Please enter your age to check whether you qualify for a driving license or not?
```

Traceback (most recent call last):

```
File "D:/python programs/a.py", line 2, in <module>
```

```
    age >= 20
```

TypeError: '>=' not supported between instances of 'str' and 'int'

We get an error in return because the input() function only takes strings as input from the user. The program is unable to integrate the number with the string hence returns an error. This problem can be solved by including the int() function inside the input() method. The int() function tells Python to treat numbers as input value. Let's see how to fix the error.

```
age = input("Please enter your age to check whether you qualify for a driving license or not?")
```

```
age = int(age)
```

```
if age >= 18:
```

```
    print("\nCongratulations! You are qualified for license.")
```

```
else:
```

```
    print("\nSorry, you cannot get your license. Please come back later when you grow up.")
```

The While Loop

While loop is special because it controls the flow of a program. A while loop in a program will keep the program running as long as a certain condition remains true. The moment the condition turns out to be false, the loop ends and the program stops. Let's create a program that does some calculation.

```
my_num = 10
```

```
while my_num <= 50:
```

```
    print(my_num)
```

```
    my_num += 5
```

```
===== RESTART: D:/python programs/a.py =====
```

```
10
```

```
15
```

```
20
```

25

30

35

40

45

50

>>>

I will now change the conditions and see what happens.

```
my_num = 100
```

```
while my_num <= 500:
```

```
    print(my_num)
```

```
    my_num += 20
```

```
=====
=====
```

RESTART:

D:/python

programs/a.py

100

120

140

160

180

200

220

240

260

280

300

320

340

360

380

400

420

440

460

480

500

>>>

(Matthes, 2016)

In the latest program, Python starts counting from 100 and keeps adding 20 until the number is equal to or less than 500. Similarly, we can add a while loop that stops a program when the condition turns out to be true. If you are creating a game, you will have to provide players with the facility to quit the game. Let's see an example.

```
txt = "\nThis program will echo whatever you enter: "
```

```
txt += "\nPlease press 'quit' to exit from the program."
```

```
msg = ""
```

```
while msg != 'quit':
```

```
    msg = input(txt)
```

```
    print(msg)
```

```
===== RESTART: D:/python programs/a.py
=====
```

This program will echo whatever you enter:

Please press 'quit' to exit from the program. This winter I am planning to go to Iceland and see mountains covered with thick ice.

This winter I am planning to go to Iceland and see mountains covered with thick

ice.

This program will echo whatever you enter:

Please press 'quit' to exit from the program.After that I will visit the United States of America.

After that I will visit the United States of America.

This program will echo whatever you enter:

Please press 'quit' to exit from the program.quit

quit

>>>

So, our program has just two options. For the first option a user can enter information that the program will echo. The second option will ask the user to quit the program. He can use the second option whenever he likes. There is a special function != that is used for allowing the user to quit the program. The while loop has not set at the condition of receiving quit from the user. It will keep running as long as the user doesn't type quit. You might be thinking that our program is not so sophisticated as we have to get the word quit printed out in order to exit from the program. We can fix this by using a simple if test. Let's see how to do that.

```
txt = "\nThis program will echo whatever you enter: "
```

```
txt += "\nPlease press 'quit' to exit from the program."
```

```
msg = ""
```

```
while msg != 'quit':
```

```
    msg = input(txt)
```

```
    if msg != 'quit':
```

```
        print(msg)
```

```
===== RESTART: D:/python programs/a.py
=====
```

This program will echo whatever you enter:

Please press 'quit' to exit from the program.This winter I am planning to go to Iceland and see mountains covered with thick ice.

This winter I am planning to go to Iceland and see mountains covered with thick ice.

This program will echo whatever you enter:

Please press 'quit' to exit from the program.After that I will visit the United States of America.

After that I will visit the United States of America.

This program will echo whatever you enter:

Please press 'quit' to exit from the program.quit

>>>

The program kept running until the condition was true. When the user entered quit, it stopped. In addition, it didn't print the word quit while pausing the program and allow users to exit out of it. This works pretty well if you are developing a game. But for more complex programs, things are different. Say you are developing a game in which a player has to kill enemies and win a battle, but if he gets killed himself or runs out of ammunition, the game ends. So, we have three conditions to end the game. Here we have to fill the program in with three conditions so that it keeps running as long as they are true. You have to use a special variable named as flag to achieve this special goal.

```
txt = "\nThis program will echo whatever you enter: "
```

```
txt += "\nPlease press 'quit' to exit from the program."
```

```
current = True
```

```
while current:
```

```
    msg = input(txt)
```

```
    if msg == 'quit':
```

```
        current = False
```

```

        else:
            print(msg)
===== RESTART: D:/python programs/a.py
=====

```

This program will echo whatever you enter:
Please press 'quit' to exit from the program.Hello
Hello

This program will echo whatever you enter:
Please press 'quit' to exit from the program.quit
>>>

In the above program, I have set the flag in the program to end it. Some users don't wait for the game to end. In this case, certain conditions cannot be true for the game to stop. Suppose a gamer has to close the game midway. For this purpose you need to break the while loop to allow the gamer to end it midway.

```

txt = "\nThis program will echo whatever you enter: "
txt += "\nPlease press 'quit' to exit from the program."

```

```

while True:
    msg = input(txt)
    if msg == 'quit':
        break
    else:
        print(msg)

```

```

===== RESTART: D:/python programs/a.py
=====

```

This program will echo whatever you enter:

Please press 'quit' to exit from the program.I am going to California
I am going to California

This program will echo whatever you enter:

Please press 'quit' to exit from the program.I will come back soom.
I will come back soom.

This program will echo whatever you enter:

Please press 'quit' to exit from the program.quit

>>>

This is how you can insert the break statement in the game to exit midway. This loop is special as it starts with while True. The loop is infinite. You can keep it running as long as you want to.

There also are infinite loops. A single mistake in the syntax will set your loop on running at an infinite level. Let's create a loop.

x = 50

while x<= 500:

 print(x)

 x += 10

=====
=====

RESTART:

D:/python

programs/a.py

50

60

70

80

90

100

110

120

130

140

150

160

170

180

190

200

210

220

230

240

250

260

270

280

290

300

310

320

330

340

350

360

370

380

```
390
400
410
420
430
440
450
460
470
480
490
500
>>>
```

Let's omit the last line from the code.

```
x = 50
```

```
while x<= 500:
```

```
    print(x)
```

(Matthes, 2016)

Now this loop runs forever. It will not stop.

Use of while Loops in Lists and Dictionaries

Strings and number best suit simple programs with single end users. If you want to build complex programs that include input from multiple users, you should create a while loop by integrating a list or a dictionary instead of simple string and integer. Let's see how can we pair up a while loop and a list or a dictionary. This tendency of Python programming allows you to collect information, store it, and then organize the input. Let's move on to the Python editor for writing a code.

```
# We will create a list of students that have taken an exam
# There will be an empty list that contains name of the students who have passed
the exam.
```

```
student_names = ['Jimmy', 'twinkle', 'jasmine', 'mary', 'andy', 'hillary', 'john',
'adam', 'jack']
passed_students = []
```

```
#the program will transfer the names of passed students to the empty list.
```

```
while student_names:
    current_name = student_names.pop()

    print("Passed student name: " + current_name.title())
    passed_students.append(current_name)
```

```
# We will have a fully prepared list of passed students.
```

```
print("\nThese students have passed the exam:")
```

```
for passed_students in passed_students:
```

```
    print(passed_students.title())
```

```
===== RESTART: D:/python programs/PythonProgram.py
=====
```

```
Passed student name: Jack
```

```
Passed student name: Adam
```

```
Passed student name: John
```

```
Passed student name: Hillary
```

```
Passed student name: Andy
```

```
Passed student name: Mary
```

```
Passed student name: Jasmine
```

Passed student name: Twinkle

Passed student name: Jimmy

These students have passed the exam:

Jack

Adam

John

Hillary

Andy

Mary

Jasmine

Twinkle

Jimmy

>>>

Initially we had a general list of students who were taking a university exam. We added to the program an empty list that was named as `passed_students`. You can see that the square brackets were empty at the start of the program. After that I created a while loop that was conditioned to run as long as the general list of students had a name in it. It had to run until the last student name was transferred to the list of passed students. I integrated the `pop()` function in the loop that removed each student from the end of the list and added it to the empty list that I had just created. I have also added a text message that explains how many students have actually passed the exam.

We can use Python while loops for removing all instances of a particular item in the lists. This is very advantageous due to the fact that you don't have to remove similar items individually from the list. Let's create a list that has similar instance of a particular item.

```
students = ['jasmine', 'johnny', 'jasmine', 'adam', 'jasmine', 'harry', 'tommy']
```

```
print(students)
```

```
while 'jasmine' in students:
```

```
students.remove('jasmine')
```

```
print(students)
```

```
===== RESTART: D:/python programs/PythonProgram.py  
=====
```

```
['jasmine', 'johnny', 'jasmine', 'adam', 'jasmine', 'harry', 'tommy']
```

```
['johnny', 'adam', 'harry', 'tommy']
```

```
>>>
```

Chapter 7

Python If Statements

In Python programming, you need to set certain conditions in the coding. Python reads the conditions and responds accordingly. Python if statements are really helpful when you have to build a program that could make logical decisions. Python supports certain logical conditions such as equals and greater than or less than. You can use these conditions in a number of ways. This chapter will walk you through a number of conditional tests. You will learn how to write if statements, if-else, and if-elif statements. Let's see the construct of the if statement. The if in the if statement is a Python keyword. Let's try our first if statement.

```
x = 500
```

```
y = 1000
```

```
if y > x:
```

```
    print("y has a greater value than that of x.")
```

```
===== RESTART: D:/python programs/PythonProgram.py
=====
```

```
y has a greater value than that of x.
```

```
>>>
```

So, we have our first if statement. When Python found the condition true, it returned the message that was enclosed inside the print function. If the condition is not true, nothing will be there except the shell prompt.

```
x = 1100
```

```
y = 1000
```

```
if y > x:
```

```
    print("y has a greater value than that of x.")
```

```
===== RESTART: D:/python programs/PythonProgram.py  
=====
```

```
>>>
```

An important thing to keep in mind when you are writing if statements is that you must take great care about indentation. Anything wrong in indentation will lead to a syntax error that will be displayed in the Python shell.

```
>>> x = 1100
```

```
>>> y = 1000
```

```
>>> if y > x:
```

```
print("y has a greater value than that of x.")
```

SyntaxError: expected an indented block

```
>>>
```

There should be a whitespace at the beginning of a line. While other programming languages are dependent on curly brackets for the purpose, Python heavily relies on whitespaces. If you want to understand the importance of whitespaces, compare them each time with curly brackets.

Another important feature of if statements is the elif statement. The keyword elif can be compared with the statement that if the previous conditions remained false, please try this one. The elif statement is crucial for Python programs. Let's see how to use it when writing a program.

```
x = 1100
```

```
y = 1000
```

```
if y > x:
    print("y has a greater value than that of x.")
elif x == y:
    print("y has equal value to that of the x.")
(Python If ... Else, n.d)
```

The first condition is false, as you can see, so Python shell goes on to read the elif statement, and displays the result. Even if the elif statement is false, what should you do then? There is a solution for that. We have the else statement that succeeds the elif statement and displays the result even if the elif statement remains wrong. Let's see the syntax and the result accordingly.

```
>>> x = 1100
>>> y = 1000
>>> if y > x:
    print("y has a greater value than that of x.")
elif x == y:
    print("y has equal value to that of the x.")
else:
    print("x has greater value than y.")
```

x has greater value than y.

```
>>>
```

(Python If ... Else, n.d)

I will change the values and see different results.

```
x = 1000
y = 1000
if y > x:
    print("y has a greater value than that of x.")
elif x == y:
    print("y has equal value to that of the x.")
```

else:

```
    print("x has greater value than y.")
```

```
===== RESTART: D:/python programs/PythonProgram.py  
=====
```

y has equal value to that of the x.

>>>

(Python If ... Else, n.d)

Let's try another condition.

```
x = 500
```

```
y = 1000
```

```
if y > x:
```

```
    print("y has a greater value than that of x.")
```

```
elif x == y:
```

```
    print("y has equal value to that of the x.")
```

```
else:
```

```
    print("x has greater value than y.")
```

```
===== RESTART: D:/python programs/PythonProgram.py  
=====
```

y has a greater value than that of x.

>>>

(Python If ... Else, n.d)

To write an else statement, elif statement is not a must. You can write an else statement without an if statement. Let's see how to do that.

```
x = 500
```

```
y = 1000
```

```
if y > x:
```

```
    print("y has a greater value than that of x.")
```

```
else:
```

```
print("x has greater value than y.")
```

```
===== RESTART: D:/python programs/PythonProgram.py  
=====
```

y has a greater value than that of x.

```
>>>
```

(Python If ... Else, n.d)

Let's fill it in with different values.

```
x = 5000
```

```
y = 1000
```

```
if y > x:
```

```
    print("y has a greater value than that of x.")
```

```
else:
```

```
    print("x has greater value than y.")
```

```
===== RESTART: D:/python programs/PythonProgram.py  
=====
```

x has greater value than y.

```
>>>
```

(Python If ... Else, n.d)

Instead of writing the code on more than one line, you can write it on a single line. Let's see how to do that.

```
x = 1100
```

```
y = 1000
```

```
print("y has a greater value than that of x.")if y > x else print("y has equal value  
to that of the x.") if x>y else print("x has greater value than y.")
```

```
===== RESTART: D:/python programs/PythonProgram.py  
=====
```

y has equal value to that of the x.

```
>>>
```

The And Keyword

The and keyword in Python is also a logical operator. You can use it to pair up multiple conditional statements.

```
x = 1100
y = 1000
z = 1500
if x > y and z > x:
    print("Both conditions are true")
```

```
===== RESTART: D:/python programs/PythonProgram.py
=====
```

```
Both conditions are true
```

```
>>>
```

```
(Python If ... Else, n.d)
```

The Or Keyword

Apart from the and keyword we have the or keyword which also is a logical operator. You can use it to pair up different conditional statements. Let's see how to do that.

```
x = 1100
y = 1000
z = 1500
if x > y or z > x:
    print("At least one of the conditions is true!")
```

```
===== RESTART: D:/python programs/PythonProgram.py
=====
```

```
At least one of the conditions is true!
```

```
>>>
```

(Python If ... Else, n.d)

The Nested If Statement

Also, you can nest an if statement inside another if statement. The enclosed if statement is known as the nested if statement. Let's see how to do that.

```
x = 50
```

```
if x > 40:
```

```
    print("x is bigger than 40!")
```

```
    if x > 45:
```

```
        print("x is greater than 45!")
```

```
    else:
```

```
        print("x is not above 45")
```

```
===== RESTART: D:/python programs/PythonProgram.py
=====
```

```
x is bigger than 40!
```

```
x is greater than 45!
```

```
>>>
```

There is a pass keyword to include in the if statement. If you miss the print statement, the pass keyword will keep the code from error and pass you on to the shell prompt.

```
x = 50
```

```
y = 100
```

```
if y > x:
```

```
    pass
```

```
===== RESTART: D:/python programs/PythonProgram.py
=====
```

```
>>>
```

You Can Use If Statements With Lists

Python allows you to combine the if statements with lists. In this way you can manage the availability of items. Let me take you back to the list of vegetables and fruits. We can integrate an if statement inside the list to test conditions. You can easily manage changing conditions effectively. For example, if you run out of an item, you can put a condition there to check the availability and return a message to the customer. Let's jump to the Python editor. I will be creating a list and using the for loop to iterate through every single item in the list.

```
vegetables_fruits = ['pumpkin', 'garlic', 'ginger', 'tomato', 'potato', 'radish', 'carrot',  
'brinjal', 'pepper', 'onion', 'banana', 'oranges', 'figs', 'watermelon', 'melon',  
'blueberry']
```

```
for vegetables_fruit in vegetables_fruits:
```

```
    print("Selling " + vegetables_fruit + ".")
```

```
print("\nSold the stock!")
```

```
===== RESTART: D:/python programs/PythonProgram.py  
=====
```

```
Selling pumpkin.
```

```
Selling garlic.
```

```
Selling ginger.
```

```
Selling tomato.
```

```
Selling potato.
```

```
Selling radish.
```

```
Selling carrot.
```

Selling brinjal.

Selling pepper.

Selling onion.

Selling banana.

Selling oranges.

Selling figs.

Selling watermelon.

Selling melon.

Selling blueberry.

Sold the stock!

>>>

If a customer demands from you a certain vegetable which has been sold, what will you tell the customer? Here, you need an if statement to test the condition of availability. If it is available, the condition is false. If not, the condition will stand true and return a neatly formatted reply to the customer. You can add more spice to your game this way.

```
vegetables_fruits = ['pumpkin', 'garlic', 'ginger', 'tomato', 'potato', 'radish', 'carrot',  
'brinjal', 'pepper', 'onion', 'banana', 'oranges', 'figs', 'watermelon', 'melon',  
'blueberry']
```

```
for vegetables_fruit in vegetables_fruits:
```

```
    if vegetables_fruit == 'ginger':
```

```
        print("I am really sorry! We have sold out gingers.")
```

```
    else:
```

```
        print("Selling " + vegetables_fruit + ".")
```

```
print("\nSold the stock!")
```

```
===== RESTART: D:/python programs/PythonProgram.py  
=====
```

Selling pumpkin.

Selling garlic.

I am really sorry! We have sold out gingers.

Selling tomato.

Selling potato.

Selling radish.

Selling carrot.

Selling brinjal.

Selling pepper.

Selling onion.

Selling banana.

Selling oranges.

Selling figs.

Selling watermelon.

Selling melon.

Selling blueberry.

Sold the stock!

>>>

You can add if statement in the full program. Let's write a lengthy program and integrate the if statement inside on multiple occasions.

```
vegetables_fruits = ['pumpkin', 'garlic', 'ginger', 'tomato', 'potato', 'radish', 'carrot',  
'brinjal', 'pepper', 'onion', 'banana', 'oranges', 'figs', 'watermelon', 'melon',  
'blueberry']
```

```
for vegetables_fruit in vegetables_fruits:
```

```
    if vegetables_fruit == 'ginger':
```

```
        print("I am really sorry! We have sold out gingers.")
```

```
    else:
```

```
print("Selling " + vegetables_fruit + ".")
```

```
print("\nSold the stock!")
```

```
vegetables_fruits = ['pumpkin', 'garlic', 'ginger', 'tomato', 'potato', 'radish', 'carrot',  
'brinjal', 'pepper', 'onion', 'banana', 'oranges', 'figs', 'watermelon', 'melon',  
'blueberry']
```

```
for vegetables_fruit in vegetables_fruits:
```

```
    if vegetables_fruit == 'blueberry':
```

```
        print("I am really sorry! We have sold out blueberry.")
```

```
    else:
```

```
        print("Selling " + vegetables_fruit + ".")
```

```
print("\nSold the stock!")
```

```
vegetables_fruits = ['pumpkin', 'garlic', 'ginger', 'tomato', 'potato', 'radish', 'carrot',  
'brinjal', 'pepper', 'onion', 'banana', 'oranges', 'figs', 'watermelon', 'melon',  
'blueberry']
```

```
for vegetables_fruit in vegetables_fruits:
```

```
    if vegetables_fruit == 'ginger':
```

```
        print("I am really sorry! We have sold out tomato.")
```

```
    else:
```

```
        print("Selling " + vegetables_fruit + ".")
```

```
print("\nSold the stock!")
```

```
vegetables_fruits = ['pumpkin', 'garlic', 'ginger', 'tomato', 'potato', 'radish', 'carrot',  
'brinjal', 'pepper', 'onion', 'banana', 'oranges', 'figs', 'watermelon', 'melon',
```

```
'blueberry']
```

```
for vegetables_fruit in vegetables_fruits:
```

```
    if vegetables_fruit == 'ginger':
```

```
        print("I am really sorry! We have sold out melon.")
```

```
    else:
```

```
        print("Selling " + vegetables_fruit + ".")
```

```
print("\nSold the stock!")
```

```
===== RESTART: D:/python programs/PythonProgram.py  
=====
```

```
Selling pumpkin.
```

```
Selling garlic.
```

```
I am really sorry! We have sold out gingers.
```

```
Selling tomato.
```

```
Selling potato.
```

```
Selling radish.
```

```
Selling carrot.
```

```
Selling brinjal.
```

```
Selling pepper.
```

```
Selling onion.
```

```
Selling banana.
```

```
Selling oranges.
```

```
Selling figs.
```

```
Selling watermelon.
```

```
Selling melon.
```

```
Selling blueberry.
```

Sold the stock!

Selling pumpkin.

Selling garlic.

Selling ginger.

Selling tomato.

Selling potato.

Selling radish.

Selling carrot.

Selling brinjal.

Selling pepper.

Selling onion.

Selling banana.

Selling oranges.

Selling figs.

Selling watermelon.

Selling melon.

I am really sorry! We have sold out blueberry.

Sold the stock!

Selling pumpkin.

Selling garlic.

I am really sorry! We have sold out tomato.

Selling tomato.

Selling potato.

Selling radish.

Selling carrot.

Selling brinjal.

Selling pepper.

Selling onion.

Selling banana.

Selling oranges.

Selling figs.

Selling watermelon.

Selling melon.

Selling blueberry.

Sold the stock!

Selling pumpkin.

Selling garlic.

I am really sorry! We have sold out melon.

Selling tomato.

Selling potato.

Selling radish.

Selling carrot.

Selling brinjal.

Selling pepper.

Selling onion.

Selling banana.

Selling oranges.

Selling figs.

Selling watermelon.

Selling melon.

Selling blueberry.

Sold the stock!

>>>

We have checked the availability of different vegetables and fruits through the if statement. The if else statement tested the condition and returned with the message that tells the users about the status of a particular vegetable and fruit in the stock. The else statement ensures that the player in the game is able to make a sales pitch for all the other available items before the customers. When he has sold all the remaining items, the game displays the message that the stock has been sold.

The if else statement also tests if the list is empty or full. We can check the stock if it is full or empty before we go on to make a sales pitch. If there is nothing in the stock, we can say sorry to the customer and ask him to visit the shop again. Let's see how to do that.

```
vegetables_fruits = []
```

```
if vegetables_fruits:
```

```
    for vegetables_fruit in vegetables_fruits:
```

```
        print("Selling " + vegetables_fruit + ".")
```

```
    print("\nSold the stock.")
```

```
else:
```

```
    print("I am really sorry! Our stock is empty at the moment.")
```

```
===== RESTART: D:/python programs/PythonProgram.py
=====
```

```
I am really sorry! Our stock is empty at the moment.
```

```
>>>
```

Suppose the list is not empty. Let's fill it up and see what happens with the result when the stock is full.

```
vegetables_fruits = ['pumpkin', 'garlic', 'ginger', 'tomato', 'potato', 'radish', 'carrot',  
'brinjal', 'pepper', 'onion', 'banana', 'oranges', 'figs', 'watermelon', 'melon',  
'blueberry']
```

```
if vegetables_fruits:
```

```
    for vegetables_fruit in vegetables_fruits:
```

```
    print("Selling " + vegetables_fruit + ".")
    print("\nSold the stock.")
else:
    print("I am really sorry! Our stock is empty at the moment.")
```

```
===== RESTART: D:/python programs/PythonProgram.py
=====
```

```
Selling pumpkin.
Selling garlic.
Selling ginger.
Selling tomato.
Selling potato.
Selling radish.
Selling carrot.
Selling brinjal.
Selling pepper.
Selling onion.
Selling banana.
Selling oranges.
Selling figs.
Selling watermelon.
Selling melon.
Selling blueberry.
```

```
Sold the stock.
```

```
>>>
```

(Matthes, 2016)

You can also add multiple lists to the same program to build up a complex

system. For example, if the player in your game is running a super store, it can have other items for sale. You have to use lists as well as if statements in order to make sure the input by the users are catered accordingly. Let's include another list in the same program. The second list will be of fast food items. Customers will also demand the other items. Let's see how to build such a program.

```
general_items = ['potato chips', 'burgers', 'pizza', 'roasted chicken', 'lays',  
'pringles','pumpkin', 'garlic', 'ginger', 'tomato', 'potato', 'radish', 'carrot', 'brinjal',  
'pepper', 'onion', 'banana', 'oranges', 'figs', 'watermelon', 'melon', 'blueberry']
```

```
fast_foods = ['potato chips', 'burgers', 'pizza', 'roasted chicken', 'lays', 'pringles']
```

```
for fast_food in fast_foods:
```

```
    if fast_food in general_items:
```

```
        print("Selling " + fast_food + ".")
```

```
    else:
```

```
        print("I am really sorry! Our stock is empty at the moment.")
```

```
print("\nSold the entire stock")
```

```
===== RESTART: D:/python programs/PythonProgram.py  
=====
```

```
Selling potato chips.
```

```
Selling burgers.
```

```
Selling pizza.
```

```
Selling roasted chicken.
```

```
Selling lays.
```

```
Selling pringles.
```

```
Sold the entire stock
```

```
>>>
```

I have defined a general item list and a fast food item list in the start. I have looped through the list by a for loop to see whether the fast food items are

available in the list or not. The for loop iterates through all the items in the fast food list and comes up with the result. When the for loop finds that fast food items are no longer in the list, it will run the else block, and print the statement that we have added to the else block. The statement informs the customer that the desired item has been sold out. (Matthes, 2016)

```
general_items = ['potato chips', 'burgers', 'pizza', 'roasted chicken', 'lays',  
'pringles','pumpkin', 'garlic', 'ginger', 'tomato', 'potato', 'radish', 'carrot', 'brinjal',  
'pepper', 'onion', 'banana', 'oranges', 'figs', 'watermelon', 'melon', 'blueberry']
```

```
fast_foods = ['potato chips', 'paratha rolls', 'burgers', 'pizza', 'Cheesy sandwich',  
'roasted chicken', 'lays', 'pringles']
```

```
for fast_food in fast_foods:
```

```
    if fast_food in general_items:
```

```
        print("Selling " + fast_food + ".")
```

```
    else:
```

```
        print("I am really sorry! Our stock is empty at the moment for this  
particular item.")
```

```
print("\nSold the entire stock")===== RESTART: D:/python  
programs/PythonProgram.py =====
```

```
Selling potato chips.
```

```
I am really sorry! Our stock is empty at the moment for this particular item.
```

```
Selling burgers.
```

```
Selling pizza.
```

```
I am really sorry! Our stock is empty at the moment for this particular item.
```

```
Selling roasted chicken.
```

```
Selling lays.
```

```
Selling pringles.
```

```
Sold the entire stock
```

```
>>>
```

(Matthes, 2016)

You can see that whenever the loop found that a particular item was not there in the list, it started running the else block that showed the buyer that a particular item was no longer in the stock, and that the stock had been empty. This is how you can build your own programs. For example, you can create a program for a game in which a player goes to a gun store and asks for multiple guns to use later on for battling an enemy in the core of mountains. This program is quite suitable for the occasion. You can have a pretty lively and interactive conversation and business dealing between the shopkeeper and the buyer. Let's create such a program.

```
general_ammunition = ['revolver', 'AK-47', 'machine gun', 'tank', 'bunker',  
'pistol','shotgun', 'handgun', 'magazines', 'hand grenades', 'sub machine gun',  
'rocket launcher', 'bazooka', 'steel stick', 'EM bombs', 'steel gloves']
```

```
player1_guns = ['revolver', 'AK-47', 'machine gun', 'tank', 'bunker',  
'pistol','shotgun', 'handgun', 'magazines', 'apples', 'guava', 'strawberry',  
'blueberry']
```

```
for player1_gun in player1_guns:
```

```
    if player1_gun in general_ammunition:
```

```
        print("Selling " + player1_gun + ".")
```

```
    else:
```

```
        print("I am really sorry! Our stock is empty at the moment for this  
particular item.")
```

```
print("\nSold the entire stock")
```

```
===== RESTART: D:/python programs/PythonProgram.py  
=====
```

```
Selling revolver.
```

```
Selling AK-47.
```

```
Selling machine gun.
```

```
Selling tank.
```

Selling bunker.

Selling pistol.

Selling shotgun.

Selling handgun.

Selling magazines.

I am really sorry! Our stock is empty at the moment for this particular item.

I am really sorry! Our stock is empty at the moment for this particular item.

I am really sorry! Our stock is empty at the moment for this particular item.

I am really sorry! Our stock is empty at the moment for this particular item.

Sold the entire stock

>>>

Chapter 8

Python Functions and Dictionaries

Now that you have gained a good practical knowledge of basic and advanced Python coding, it is time to move on to more advanced options. This chapter will walk you through Python functions. These are blocks of code that are created as such to do a certain task. Firstly, you have to define a function which you can call by the name in the middle of a program. Python functions are amazing because you can call them multiple times whenever you need them to build a program. Basically, a function call asks Python to run the code that is enclosed inside the function. Functions basically make it easier to run your programs and also read, write, and test them.

You can also pass information to the functions after they have been successfully created. Python function are also helpful as they are used to return crucial information to process. Let's get it started with Python functions.

How to Define a Function?

As you know, a function is a block of code that runs whenever it is called. The data that you pass on to the function is named as parameter. Usually, a Python function effectively returns data after each calling.

```
def motor_func():
```

```
"""The function will tell users about super cars."""  
print("I have bought a brand new car.")
```

```
motor_func()
```

```
===== RESTART: D:/python programs/PythonProgram.py  
=====
```

```
I have bought a brand new car.
```

```
>>>
```

You can see the structure of the function. This is one of the simplest functions that displays the information about the purchase of a brand new car. In the first line you can find the `def` keyword that defines the function. After that you have to enter the name of the function. Next to the name are parentheses which are used for a lot of important purposes. We will discuss their importance in the later sections of this chapter.

As we have no extra information at the moment to pass on to the Python function, we have left the parentheses empty. In the second line I have explained the reason behind the creation of the function. This is called docstring and is enclosed inside triple quotation marks as you can see in the code snippet. If you miss a quotation mark and enclose it in double, Python will fail to recognize it as a docstring and most likely will return an error. When we have defined the function, we can call it by writing its name. The information we have entered in the print statement in this case will be displayed in the Python shell. This print statement is in reality the real line of code in the Python function. In the last line of code I have called the function by writing its name. You don't have to do anything except that. Just write the name and Python will run the function.

How to Pass Information to the Function?

Functions allow you to pass information midway while the program is running. You don't have to define them from the start to pass the information. For the purpose we have parentheses. You can fill them in with arguments that are a name given to the data you enter. Let's continue with the function that we have defined and attempt to pass some additional information to functions so that the same can be displayed before the users. I will explain the steps involved in the

process later on. Firstly, we have to open Python editor.

```
def motor_func(car_name):
```

```
    """The function will tell users about super cars."""
```

```
    print("I have bought a brand new car that is named as " + car_name.title() + ".")
```

```
motor_func('BMW')
```

```
===== RESTART: D:/python
programs/PythonProgram.py =====
```

```
I have bought a brand new car that is named as Bmw.
```

```
>>>
```

Let's try it by filling it in with another car name.

```
def motor_func(car_name):
```

```
    """The function will tell users about super cars."""
```

```
    print("I have bought a brand new car that is named as " + car_name.title() + ".")
```

```
motor_func('lamborghini')
```

```
===== RESTART: D:/python
programs/PythonProgram.py =====
```

```
I have bought a brand new car that is named as Lamborghini.
```

```
>>>
```

(Matthes, 2016)

You can change the case of the output by a change in the method.

```
def motor_func(car_name):
```

```
    """The function will tell users about super cars."""
```

```
    print("I have bought a brand new car that is named as " + car_name.upper() + ".")
```

```
motor_funct('lamborghini')
```

```
===== RESTART: D:/python programs/PythonProgram.py  
=====
```

```
I have bought a brand new car that is named as LAMBORGHINI.
```

```
>>>
```

So, everything went nicely. I filled in the parentheses with a variable named carname. Later on I integrated it inside the print statement. When I called the function, I filled in the parentheses with the name of the car, and it automatically passed the information to the variable in the function.

A Look at Arguments and Parameters

You can see how the function integrated the argument to the variable named carname. When we called it, it printed the desired line right away without any error. The information BMW and lamborghini are called parameters. They are pieces of information that are regularly changed as per your requirements. In the world of Python we call them arguments. These values are then passed to the parameter which in this case is the variable 'carname.'

How to Pass Arguments to Python Functions

We have seen how easy it is to pass arguments. Can we pass multiple arguments to make our function appear more sophisticated and complex? We can pass multiple arguments to Python functions in a great number of ways. Let's explore different kinds in Python. On top of all are positional arguments. Let's delve deeper into the topic.

Positional Arguments

Whenever you are about to call a function, the arguments you enter in the function call must be able to match the parameter inside the function definition. So, in simple words, you have to place your arguments in matching position so that the function reads them carefully and likewise integrates them with the parameters and displays the result. Because the position of the arguments is

important, they are named as positional arguments so that the programmer can remember them easily. Let's move on to our function that is meant to display information about cars. I am not creating a function so that it can be easier for you to follow the old one in order to get a good grasp of the process.

```
def motor_func(car_name, car_price):  
    """The function will tell users about super cars."""  
    print("\nI have bought a brand new car that is named as " +  
car_name.upper() + ".")  
    print("\nThis car is worth " + car_price.title() + ".")  
  
motor_func('lamborghini', 'ten million dollars')
```

```
===== RESTART: D:/python  
programs/PythonProgram.py =====
```

I have bought a brand new car that is named as LAMBORGHINI.

This car is worth Ten Million Dollars.

>>>

Let's try different arguments.

```
def motor_func(car_name, car_price):  
    """The function will tell users about super cars."""  
    print("\nI have bought a brand new car that is named as " +  
car_name.upper() + ".")  
    print("\nThis car is worth " + car_price.title() + ".")  
  
motor_func('ferrari', 'five million dollars')
```

```
===== RESTART: D:/python  
programs/PythonProgram.py =====
```

I have bought a brand new car that is named as FERRARI.

This car is worth Five Million Dollars.

>>>

Now let's see what happens when we change the position of the arguments.

```
def motor_func(car_name, car_price):
```

```
    """The function will tell users about super cars."""
```

```
    print("\nI have bought a brand new car that is named as " +  
car_name.upper() + ".")
```

```
    print("\nThis car is worth " + car_price.title() + ".")
```

```
motor_func('five million dollars', 'ferrari')
```

```
===== RESTART: D:/python  
programs/PythonProgram.py =====
```

I have bought a brand new car that is named as FIVE MILLION DOLLARS.

This car is worth Ferrari.

>>>

(Matthes, 2016)

We can observe that the definition was the same except that I added one more parameter in the function. After that came the docstring that explained the nature of the function. When I am done with the print statement, I move on to writing the arguments that are stored in the parameters as Python matched them. In addition to accepting multiple arguments, Python also allows multiple function calls. Let's see how to make them.

```
def motor_func(car_name, car_price):
```

```
    """The function will tell users about super cars."""
```

```
    print("\nI have bought a brand new car that is named as " +  
car_name.upper() + ".")
```

```
    print("\nThis car is worth " + car_price.title() + ".")
```

```
motor_func('ferrari', 'ten million dollars')
motor_func('lamborghini', 'five million dollars')
motor_func('land rover', 'two million dollars')
```

```
===== RESTART: D:/python
programs/PythonProgram.py =====
```

I have bought a brand new car that is named as FERRARI.

This car is worth Ten Million Dollars.

I have bought a brand new car that is named as LAMBORGHINI.

This car is worth Five Million Dollars.

I have bought a brand new car that is named as LAND ROVER.

This car is worth Two Million Dollars.

>>>

Calling a function multiple times allows programmers to be at ease while coding as they don't have to repeat the same information again and again. I mean they don't have to write the same function on different occasions for different purposes. They just include new arguments in the function call, and set the entirety to work on the basis of the new arguments. We have successfully described three cars in the above example just by calling the function. In addition, if you want to you can include more parameters in the function to add more information about the object you are trying to describe by making the function call. Let's move on to Python editor and add more parameters to the function to test its expanding capabilities.

```
def motor_func(car_name, car_price, car_make):
```

```
    """The function will tell users about super cars."""
```

```
    print("\nI have bought a brand new car that is named as " +
car_name.upper() + ".")
```

```
print("\nThis car is worth " + car_price.title() + ".")
print("\nThe model I bought was manufactured and assembled in " +
str(car_make) + ".")
```

```
motor_func('ferrari', 'ten million dollars', 2009)
motor_func('lamborghini', 'five million dollars', 2004)
motor_func('land rover', 'two million dollars', 2019)
```

```
===== RESTART: D:/python programs/PythonProgram.py
=====
```

I have bought a brand new car that is named as FERRARI.

This car is worth Ten Million Dollars.

The model I bought was manufactured and assembled in 2009.

I have bought a brand new car that is named as LAMBORGHINI.

This car is worth Five Million Dollars.

The model I bought was manufactured and assembled in 2004.

I have bought a brand new car that is named as LAND ROVER.

This car is worth Two Million Dollars.

The model I bought was manufactured and assembled in 2019.

>>>

Let's see what happens if you miss an argument and run the program in Python.

```
def motor_func(car_name, car_price, car_make):
```

```

        """The function will tell users about super cars."""
        print("\nI have bought a brand new car that is named as " + car_name.upper()
+ ".")
        print("\nThis car is worth " + car_price.title() + ".")
        print("\nThe model I bought was manufactured and assembled in " +
str(car_make) + ".")

```

```

motor_func('ferrari', 'ten million dollars', 2009)
motor_func('lamborghini', 'five million dollars', 2004)
motor_func('land rover', 'two million dollars', )

```

```

===== RESTART:  D:/python  programs/PythonProgram.py
=====

```

I have bought a brand new car that is named as FERRARI.

This car is worth Ten Million Dollars.

The model I bought was manufactured and assembled in 2009.

I have bought a brand new car that is named as LAMBORGHINI.

This car is worth Five Million Dollars.

The model I bought was manufactured and assembled in 2004.

Traceback (most recent call last):

File "D:/python programs/PythonProgram.py", line 9, in <module>

motor_func('land rover', 'two million dollars',)

TypeError: motor_func() missing 1 required positional argument: 'car_make'

>>>

You will receive the above mentioned error and the program will stop running.

Let's make this error in the first function call. It will definitely halt the program and the next function calls will not be able to execute.

```
def motor_func(car_name, car_price, car_make):  
    """The function will tell users about super cars."""  
    print("\nI have bought a brand new car that is named as " +  
car_name.upper() + ".")  
    print("\nThis car is worth " + car_price.title() + ".")  
    print("\nThe model I bought was manufactured and assembled in " +  
str(car_make) + ".")  
  
motor_func('ferrari', 'ten million dollars', )  
motor_func('lamborghini', 'five million dollars', 2004)  
motor_func('land rover', 'two million dollars', 2019)
```

```
===== RESTART: D:/python programs/PythonProgram.py  
=====
```

Traceback (most recent call last):

File "D:/python programs/PythonProgram.py", line 7, in <module>

```
    motor_func('ferrari', 'ten million dollars', )
```

TypeError: motor_func() missing 1 required positional argument: 'car_make'

```
>>>
```

Keyword Arguments

A keyword argument is the most practical way to avoid errors that you can commit by changing the position of an argument, only to get a mismatch in the print statements, and hitting an ultimate disaster. Keyword arguments are the perfect solution to this problem. With keyword arguments you don't have to worry about the position of the arguments. Even if you misplace them, you will get the right result.

```
def motor_func(car_name, car_price, car_make):  
    """The function will tell users about super cars."""
```

```
print("\nI have bought a brand new car that is named as " +  
car_name.upper() + ".")
```

```
print("\nThis car is worth " + car_price.title() + ".")
```

```
print("\nThe model I bought was manufactured and assembled in " +  
str(car_make) + ".")
```

```
motor_func(car_name = 'ferrari', car_price = 'ten million dollars', car_make =  
2005)
```

```
motor_func(car_name = 'lamborghini', car_make = 2004, car_price = 'five  
million dollars')
```

```
motor_func(car_price = 'two million dollars', car_name = 'land rover', car_make  
= 2019)
```

```
===== RESTART: D:/python programs/PythonProgram.py  
=====
```

I have bought a brand new car that is named as FERRARI.

This car is worth Ten Million Dollars.

The model I bought was manufactured and assembled in 2005.

I have bought a brand new car that is named as LAMBORGHINI.

This car is worth Five Million Dollars.

The model I bought was manufactured and assembled in 2004.

I have bought a brand new car that is named as LAND ROVER.

This car is worth Two Million Dollars.

The model I bought was manufactured and assembled in 2019.

```
>>>
```

(Matthes, 2016)

Filling the Function With Default Values

In addition to positional and keyword arguments, there are default values which can be filled in the function. There is a default value for each parameter. So, whenever you make a function call, you will see the default value. If you fill in the function with arguments, it will return the same but if you leave it vacant, the function will display the default values that you have already fed it with. Default values tend to simplify your function calls.

Let's fill default values to a function and see how it returns it with and without the arguments. This will help you understand how Python functions work.

```
def motor_func(car_name = 'bmw', car_price = 'fifty million dollars', car_make = 2018):
```

```
    """The function will tell users about super cars."""
```

```
    print("\nI have bought a brand new car that is named as " + car_name.upper() + ".")
```

```
    print("\nThis car is worth " + car_price.title() + ".")
```

```
    print("\nThe model I bought was manufactured and assembled in " + str(car_make) + ".")
```

```
motor_func(car_name = 'ferrari', car_price = 'ten million dollars', car_make = 2005)
```

```
motor_func(car_name = 'lamborghini', car_make = 2004, car_price = 'five million dollars')
```

```
motor_func(car_price = 'two million dollars', car_name = 'land rover', car_make = 2019)
```

```
===== RESTART: D:/python programs/PythonProgram.py =====
```

I have bought a brand new car that is named as FERRARI.

This car is worth Ten Million Dollars.

The model I bought was manufactured and assembled in 2005.

I have bought a brand new car that is named as LAMBORGHINI.

This car is worth Five Million Dollars.

The model I bought was manufactured and assembled in 2004.

I have bought a brand new car that is named as LAND ROVER.

This car is worth Two Million Dollars.

The model I bought was manufactured and assembled in 2019.

>>>

You can see how I filled the function with the default values and ran the code. The display was the same. All the changes that I have made are in the first line of code. I have changed the parameters by pairing them up with the default values. But these values don't have any impact on the display of information because I already have filled in the function call with relevant arguments. Let's remove the arguments and then make the function call.

```
def motor_func(car_name = 'bmw', car_price = 'fifty million dollars', car_make = 2018):
```

```
    """The function will tell users about super cars."""
```

```
    print("\nI have bought a brand new car that is named as " + car_name.upper() + ".")
```

```
    print("\nThis car is worth " + car_price.title() + ".")
```

```
    print("\nThe model I bought was manufactured and assembled in " + str(car_make) + ".")
```

```
motor_func()
```

```
===== RESTART: D:/python programs/PythonProgram.py
=====
```

I have bought a brand new car that is named as BMW.

This car is worth Fifty Million Dollars.

The model I bought was manufactured and assembled in 2018.

```
>>>
```

As expected, the Python shell returned the statement by including the default values in the results. Default values can be manipulated to do specific tasks. For example, you can fix default values for a parameter and leave the rest of parameters open for change.

```
def motor_func(car_name, car_price, car_make, car_color = 'red'):
```

```
    """The function will tell users about super cars."""
```

```
    print("\nI have bought a brand new car that is named as " +
car_name.upper() + ".")
```

```
    print("\nThis car is worth " + car_price.title() + ".")
```

```
    print("\nThe model I bought was manufactured and assembled in " +
str(car_make) + ".")
```

```
    print("\nIts color is " + car_color.title() + ".")
```

```
motor_func(car_name = 'ferrari', car_price = 'ten million dollars', car_make =
2005)
```

```
motor_func(car_name = 'lamborghini', car_make = 2004, car_price = 'five
million dollars')
```

```
motor_func(car_price = 'two million dollars', car_name = 'land rover', car_make
= 2019)
```

```
===== RESTART: D:/python
```

programs/PythonProgram.py =====

I have bought a brand new car that is named as FERRARI.

This car is worth Ten Million Dollars.

The model I bought was manufactured and assembled in 2005.

Its color is Red.

I have bought a brand new car that is named as LAMBORGHINI.

This car is worth Five Million Dollars.

The model I bought was manufactured and assembled in 2004.

Its color is Red.

I have bought a brand new car that is named as LAND ROVER.

This car is worth Two Million Dollars.

The model I bought was manufactured and assembled in 2019.

Its color is Red.

>>>

You can also make equivalent function calls for better functionality. You can manipulate arguments and parameters as per your particular needs as I have demonstrated by multiple examples. You can run these codes in the Python editor and see the results. When you start coding yourself, you are highly likely to see argument errors. Let's see some of them to give you feel of what an error will look like and how can you fix it.

```
def motor_func(car_name, car_price, car_make, car_color = 'red'):
```

```

        """The function will tell users about super cars."""
        print("\nI have bought a brand new car that is named as " +
car_name.upper() + ".")
        print("\nThis car is worth " + car_price.title() + ".")
        print("\nThe model I bought was manufactured and assembled in " +
str(car_make) + ".")
        print("\nIts color is " + car_color.title() + ".")

```

```
motor_func()
```

```

===== RESTART: D:/python programs/PythonProgram.py
=====

```

Traceback (most recent call last):

File "D:/python programs/PythonProgram.py", line 8, in <module>

```
    motor_func()
```

TypeError: motor_func() missing 3 required positional arguments: 'car_name', 'car_price', and 'car_make'

```
>>>
```

Python is special among programming languages owing to how it displays its errors. It allows programmers to not only see them, but also read them and understand them, hence it gives them the opportunity to eliminate the error. In the first line you can traceback the core of the problem. The last line of the error tells us that three positional arguments are missing. It even tells us the names of the three arguments. That's how we can rectify our errors in a better way.

You Can Pass a List to a Function

Until now we have learned how can we pass information to a function in the form of arguments. Python also allows you to pass a list to the function. The list can be anything like numbers, names, and complex objects like dictionaries. When you try to pass a list to a function, the function has direct access to its contents. We will print an incomplete statement and complete it with the items in the list. Let's see how you can do that.

```
def motor_funcnt(car_names):
    """The function will tell users about super cars."""
    for car_name in car_names:
        txt = "I have bought a brand new car that is named as " + car_name.upper()
        + "."
        print(txt)
```

```
supercars = ['bmw', 'ferrari', 'lamborghini', 'mercedes bez', 'land rover', 'wagon r',
'suzuki', 'toyota']
```

```
motor_funcnt(supercars)
```

```
===== RESTART: D:/python programs/PythonProgram.py
=====
```

```
I have bought a brand new car that is named as BMW.
```

```
I have bought a brand new car that is named as FERRARI.
```

```
I have bought a brand new car that is named as LAMBORGHINI.
```

```
I have bought a brand new car that is named as MERCEDES BEZ.
```

```
I have bought a brand new car that is named as LAND ROVER.
```

```
I have bought a brand new car that is named as WAGON R.
```

```
I have bought a brand new car that is named as SUZUKI.
```

```
I have bought a brand new car that is named as TOYOTA.
```

```
>>>
```

Global and Local Variables in Python

Python has global and local variables which you can define and declare out of a function but can use them inside the function. Let's see how to create them and use them in a function.

```
def globalv():
```

```
    x = "I am learning Python."
```

```
print (s)
```

```
s = "I am trying to learn Python."
```

```
globalv()
```

```
print (s)
```

```
===== RESTART: D:/python programs/PythonProgram.py  
=====
```

```
I am trying to learn Python.
```

```
I am trying to learn Python.
```

```
>>>
```

```
Let's print x.
```

```
def globalv():
```

```
    x = "I am learning Python."
```

```
    print (x)
```

```
s = "I am trying to learn Python."
```

```
globalv()
```

```
print (s)
```

```
===== RESTART: D:/python programs/PythonProgram.py  
=====
```

```
I am learning Python.
```

```
I am trying to learn Python.
```

```
>>>
```

(Global and Local Variables in Python, n.d)

If a function is not designated as global, it is a local variable. In order to make a global variable, we need to define them.

```
def fmma():
```

```
    global x
```

```
print (x)
x = "I am learning Python."
print (x)
```

```
x = "I am trying to learn Python."
```

```
fmma()
```

```
print (x)
```

```
===== RESTART: D:/python programs/PythonProgram.py
=====
```

```
I am trying to learn Python.
```

```
I am learning Python.
```

```
I am learning Python.
```

```
>>>
```

(Global and Local Variables in Python, n.d)

Using a Function With a While Loop

Python allows us to pair up a function with a while loop. Let's see how to do that.

```
def motor_func(car_name, car_price, car_make):
    """The function will tell users about super cars."""
    txt = car_name + ' ' + car_price + ' ' + car_make
    return txt.title()
```

```
#Let's create a loop
```

```
while True:
```

```
    print("\nPlease fill in info about the car you want to sell.")
    c_name = input("Car name: ")
    c_price = input("Car price: ")
    c_make = input("Car make: ")
```

```
formatted_information = motor_funct(c_name, c_price, c_make)
print("\nSo, you want to sell this car: " + formatted_information + "!")
===== RESTART: D:/python programs/PythonProgram.py
=====
```

Please fill in info about the car you want to sell.

Car name: ferrari

Car price: ten million dollars

Car make: 2002

So, you want to sell this car: Ferrari Ten Million Dollars 2002!

Please fill in info about the car you want to sell.

Car name: bmw

Car price: five million dollars

Car make: 2003

So, you want to sell this car: Bmw Five Million Dollars 2003!

Please fill in info about the car you want to sell.

Car name: lamborghini

Car price: fifty million dollars

Car make: 2009

So, you want to sell this car: Lamborghini Fifty Million Dollars 2009!

Please fill in info about the car you want to sell.

Car name:

(Matthes, 2016)

This is an infinite for loop that goes on end and serves your customers. You can

use this program in an auto showroom that does business in sales and purchasing of used cars.

Python Dictionaries

This section will explain the creation and usage of Python dictionaries. You will learn how to use them and connect different pieces of information. When you have completed this section, you will be able to access different items that you store in a dictionary and can also modify items. Just like lists, dictionaries also can store limitless information. If you are creating a car model, you can store the related information about the car in a dictionary. There is no limit to the information that you can store in the dictionary. For example, you can add information like the color of the car, its make, and height. A dictionary stores information in the form of key pairs. Each key is linked to its respective value that you store in it. The value of the key can be anything like a list, a number or a string. You can identify a dictionary by the curly braces with a number of key pairs wrapped up inside. Let's create a dictionary.

```
mydict = {'color': 'red', 'make': 2005, 'height': 4}
```

```
print(mydict)
```

```
===== RESTART: D:/python programs/PythonProgram.py  
=====
```

```
{'color': 'red', 'make': 2005, 'height': 4}
```

```
>>>
```

(Python Dictionaries, n.d)

You can easily access items from a dictionary. Let's see how to do that.

```
mydict = {'color': 'red', 'make': 2005, 'height': 4}
```

```
print(mydict)
```

```
a = mydict["color"]
```

```
print(a)
```

```
===== RESTART: D:/python programs/PythonProgram.py
```

```
=====
```

```
{'color': 'red', 'make': 2005, 'height': 4}
```

```
red
```

```
>>>
```

(Python Dictionaries, n.d)

You can also change the values of items in the dictionary.

```
mydict = {
```

```
    'color': 'red',
```

```
    'make': 2005,
```

```
    'height': 4
```

```
}
```

```
mydict["color"] = 'green'
```

```
print(mydict)
```

```
mydict["height"] = 3
```

```
print(mydict)
```

```
===== RESTART: D:/python programs/PythonProgram.py  
=====
```

```
{'color': 'green', 'make': 2005, 'height': 4}
```

```
{'color': 'green', 'make': 2005, 'height': 3}
```

```
>>>
```

Python Calculator

I will be creating a simple calculator that would do four different mathematical calculations. For the purpose, I will define each mathematical operator in separate functions. Next come the print statements that will prompt user to take action and choose which mathematical function they want to use. Next comes the option of taking input from the user. After that I will add if else statements to decide test conditions on the basis of the input by the users. If the input is not valid, the program will display the result as invalid option.

```
# This program will create a calculator that has mathematical functions.
```

```
#I will be using functions for adding mathematical operators.
```

```
# This function is created to add two numbers
```

```
def addition(a, b):
```

```
    return a + b
```

```
# This function is created to subtract
```

```
def subtraction(a, b):
```

```
    return a - b
```

```
# This function is created to multiply
```

```
def multiplication(a, b):
```

```
    return a * b
```

```
# This function is created to divide
```

```
def division(a, b):
```

```
    return a / b
```

```
print("Dear user, you have to select the requisite operation.")
```

```
print("1.Please add the numbers.")
```

```
print("2.Please subtract the numbers.")
```

```
print("3.Please multiply the numbers.")
```

```
print("4.Please divide the numbers.")
```

```
# Time to take input
```

```

your_choice = input("Enter choice(1/2/3/4):")
digit1 = int(input("Please enter the first digit: "))
digit2 = int(input("Please enter the second digit: "))
if your_choice == '1':
    print(digit1,"+",digit2,"=", addition(digit1,digit2))
elif your_choice == '2':
    print(digit1,"-",digit2,"=", subtraction(digit1,digit2))
elif your_choice == '3':
    print(digit1,"*",digit2,"=", multiplication(digit1,digit2))
elif your_choice == '4':
    print(digit1,"/",digit2,"=", division(digit1,digit2))
else:
    print("Invalid input")

```

```

===== RESTART:  D:/python  programs/PythonProgram.py
=====

```

Dear user, you have to select the requisite operation.

- 1.Please add the numbers.
- 2.Please subtract the numbers.
- 3.Please multiply the numbers.
- 4.Please divide the numbers.

Enter choice(1/2/3/4):4

Please enter the first digit: 10

Please enter the second digit: 5

10 / 5 = 2.0

>>>

```

===== RESTART:  D:/python  programs/PythonProgram.py
=====

```

Dear user, you have to select the requisite operation.

1.Please add the numbers.

2.Please subtract the numbers.

3.Please multiply the numbers.

4.Please divide the numbers.

Enter choice(1/2/3/4):

Please enter the first digit: 1

Please enter the second digit: 50

Invalid input

```
>>> 100
```

```
100
```

```
>>>
```

```
>>>
```

```
===== RESTART: D:/python programs/PythonProgram.py
=====
```

Dear user, you have to select the requisite operation.

1.Please add the numbers.

2.Please subtract the numbers.

3.Please multiply the numbers.

4.Please divide the numbers.

Enter choice(1/2/3/4):2

Please enter the first digit: 100

Please enter the second digit: 50

100 - 50 = 50

```
>>>
```

```
===== RESTART: D:/python programs/PythonProgram.py
=====
```

Dear user, you have to select the requisite operation.

- 1.Please add the numbers.
- 2.Please subtract the numbers.
- 3.Please multiply the numbers.
- 4.Please divide the numbers.

Enter choice(1/2/3/4):3

Please enter the first digit: 50

Please enter the second digit: 2

$50 * 2 = 100$

>>>

```
===== RESTART: D:/python programs/PythonProgram.py
=====
```

Dear user, you have to select the requisite operation.

- 1.Please add the numbers.
- 2.Please subtract the numbers.
- 3.Please multiply the numbers.
- 4.Please divide the numbers.

Enter choice(1/2/3/4):3

Please enter the first digit: -50

Please enter the second digit: -50

$-50 * -50 = 2500$

>>>

```
===== RESTART: D:/python programs/PythonProgram.py
=====
```

Dear user, you have to select the requisite operation.

- 1.Please add the numbers.
- 2.Please subtract the numbers.
- 3.Please multiply the numbers.

4.Please divide the numbers.

Enter choice(1/2/3/4):1

Please enter the first digit: -500

Please enter the second digit: -500

-500 + -500 = -1000

>>>

(Python Program to Make a Simple Calculator, n.d)

Let's create a program that will calculate the leap year when you enter a particular year as input.

This Python program will check if the year a user has entered as input is a leap year or not

input_year = 2020

You will have to to get year (that will be in the form of integer input) from the user

Let's create the conditional statement. year = int(input("Enter a year: "))

if (input_year % 4) == 0:

 if (input_year % 100) == 0:

 if (input_year % 400) == 0:

 print("{0} will be a leap year".format(input_year))

 else:

 print("{0} will not be a leap year".format(input_year))

 else:

 print("{0} will be a leap year".format(input_year))

else:

 print("{0} will not be a leap year".format(input_year))

```
===== RESTART: D:/python programs/PythonProgram.py
=====
```

2020 will be a leap year

>>>

Let's try another year in the input.

This Python program will check if the year a user has entered as input is a leap year or not

```
input_year = 2025
```

You will have to to get year (that will be in the form of integer input) from the user

Let's create the conditional statement. year = int(input("Enter a year: "))

```
if (input_year % 4) == 0:
```

```
    if (input_year % 100) == 0:
```

```
        if (input_year % 400) == 0:
```

```
            print("{0} will be a leap year".format(input_year))
```

```
        else:
```

```
            print("{0} will not be a leap year".format(input_year))
```

```
    else:
```

```
        print("{0} will be a leap year".format(input_year))
```

```
else:
```

```
    print("{0} will not be a leap year".format(input_year))
```

```
===== RESTART: D:/python programs/PythonProgram.py
=====
```

2025 will not be a leap year

>>>

(Python Program to Check Leap Year, n.d)

Conclusion

Python is considered to be among the most beloved programming languages in any circle of programmers. Software engineers, hackers, and data scientists alike are in love with the versatility that Python has to offer. In addition, the object-oriented feature of Python coupled with its flexibility are also some of the major attractions for this language. Programmers are now developing a wide range of mobile as well as web applications that we enjoy on an everyday basis.

In addition to having the liberty to develop a lot of applications, the abundant libraries of Python are also an added benefit for programmers who fly at an advanced level. Python libraries offer the best way to develop data science projects as well as deep learning projects. In addition, Python also offers a wide range of modules as well as file extensions. In addition, Python offers great help in building of micro-projects as well as macro projects.

Python is great for automation projects like deep learning neural networks, machine learning models, and robotics. Today, companies are developing driverless cars and image recognition systems in addition to machine learning models for boosting up business revenue. From this angle Python may seem to be too complex a language to learn and even understand. But the reality is different. Python is very easy to learn and understand because of its clean syntax. You have experienced this by now after reading this book. The clean syntax of Python has made it easy for both beginners as well as experts. In addition, owing to the popularity of Python, you can self-study and learn.

Python is among the top programming languages that universities and industries are preferring to teach and use respectively. The charm of Python is hidden in the fact that it has extremely large applications in a wide range of fields. Most people abhor Python because of its use in building artificial intelligence models. They fear that these Python-powered AI models will drive people out of

different industries and snatch their jobs. They quote the example of Tesla's driverless taxi program by which Tesla tends to replace Uber's taxis in the US market. But the reality is different. In fact, Python-powered AI models will create many more jobs than they are going to remove. For example, building these models will become an independent industry. In addition, the implementation of these AI models will become a new job sector.

Data science is going to take the corporate world by storm. Data science is based on Python programming, as more and more companies are now moving in a neck-on-neck competition. All they crave is for a way to take an edge over their competitors. They do each and every thing for power and to get ahead. In this regard, Python seems to be promising. Python-backed data science tends to equip industries with sophisticated data about past and present sales patterns, which can help corporate sector CEOs make wiser decisions about sales and development of marketing strategies.

The biggest advantage for learners of Python is that you don't have to compile the code. In C++, you have to compile the entire program first and then run it. Only then will you be able to see whether your program runs or returns an error. Python offers the same level of programming and even at a higher stage, but still it is an interpreted language which can be easily written, edited, and corrected.

Now that you have reached the end of the book, I hope you have gained sufficient knowledge and fluency. Python is very easy to read and learn. Even you can easily read source codes for different programs that are created by other programmers. But no matter how easy it is on the outside to read and learn, it needs, like all the other programming languages, dedicated practice. You will have to get to the Python editor and practice all codes. In the beginning you can take a syntax and just paste it in the editor to see the results. In the second phase, you can make minor edits to the code and see the results. In the third phase you will be able to completely reshape a program and see how it runs in the Python shell. Given the increasing applications of Python, learning it is extremely profitable from the angle of the global job market. Python can give you the much needed edge over others when it comes to securing high paid jobs.

References

Eric Matthes. 2016. Python Crash Course[PDF file]. San Francisco: William Pollock. Retrieved from http://bedford-computing.co.uk/learning/wp-content/uploads/2015/10/No.Starch.Python.Oct_.2015.ISBN_.1593276036.pdf

Python 3 Installation & Setup Guide. n.d. Retrieved from <https://realpython.com/installing-python/#windows>

Python Data Types. n.d. Retrieved from https://www.w3schools.com/python/python_datatypes.asp

Python Tuples, n.d. Retrieved from https://www.w3schools.com/python/python_tuples.asp

Python If ... Else, n.d. Retrieved from https://www.w3schools.com/python/python_conditions.asp

Python For Loops. n.d. Retrieved from https://www.w3schools.com/python/python_for_loops.asp

Global and Local Variables in Python. n.d. Retrieved from <https://www.geeksforgeeks.org/global-local-variables-python/>

Python Dictionaries. n.d. Retrieved from https://www.w3schools.com/python/python_dictionaries.asp

Python Program to Make a Simple Calculator. n.d. Retrieved from <https://www.programiz.com/python-programming/examples/calculator>

Python Program to Check Leap Year. n.d. Retrieved from <https://www.programiz.com/python-programming/examples/leap-year>

Using IDLE (Python's IDE). n.d. Retrieved from <http://www2.cs.arizona.edu/people/mccann/usingidle>