CODING IN PYTHON

A COMPREHENSIVE BEGINNERS GUIDE TO LEARN THE REALMS OF CODING IN PYTHON

ROBERT C. MATTHEWS

Coding in Python

A Comprehensive Beginners Guide to Learn the Realms of Coding in Python

Table of Contents

Introduction

This book contains proven steps and strategies on how to code in Python. In this book, I have included all the basics of coding in Python, written in a lucid and easy-to-digest form. The book is divided into different chapters, each dealing with a specific topic. I started the book by explaining the process of downloading and installing Python on your operating system. As Windows is the most popular operating system in the world, I focused on Python installation on Windows.

After that, I moved on to Python variables and data types. I have explained them one by one. The most interesting of all the datatypes is the Python string, and it also is the most widely used in different Python programs that you will find in this book and that you will create yourself. Therefore, I emphasized it. Then I moved on to Python lists and tuples. I have tried to create a story in the book so that you find the content coherent, juicy, and highly valuable. To fulfill this purpose, I have created a game zone that is under development. The game revolves around a player who has a mission to equip an office with the cheapest in terms of price and with the most needed items so that the player can sell the office to the highest bidder. You can see the story kicking off with the start of the section on lists and tuples.

I will explain different Python codes regarding the game. As you find yourself inside the game development, you will learn different list methods such as the del method, the pop method, and list slicing. The game then moves on toward Python conditionals and dictionaries and gets more interesting. The conditionals add logic to the game, and dictionaries offer a way to store data most efficiently.

One of the most interesting programs you will see invites user input and integrates it into a while loop. You will also enjoy the chapter on functions that make the code smart and highly engaging. Functions are interesting because they can be easily paired up with different data types, and for and while loops.

Object-oriented programming will introduce to real programming. You will learn how to create classes, inheritance classes, and model real-life objects with the help of programming. You can create some very interesting programs with the help of Python classes.

I have clearly written the codes. Each code is coupled with its results. You can copy the code and paste it in the editor, do the edits of your choice, and get the result. This is how you can easily learn to code. I have written all the codes in Python 3.8 which is why I recommend you to download and install it. The process is already made a part of this book.

I wrote this book with the goal that a person who has to start Python coding from scratch can easily learn to code. The book is for those who are just getting started. I have explained each concept with due depth and comprehensiveness so that you beginners can understand it and digest it well. Even if you are just getting started in the world of coding, you can easily understand the concepts.

I recommend that you keep a notebook and a pen with you when you read this book so that you can note down the important points and code syntax to remember them and practice them. You also can bookmark them on a smart device and jump to the desired pages when you want to read them. Unlike other dry books of coding, this book will explain each step of the code so that you can easily reproduce what you have learned. I hope you have fun reading sessions once you get started. If you have got the fire to learn Python, this book should finish in five to six reading sessions.

Chapter One: Make a Start

This chapter will walk you through the process of making a start in Python coding. You will learn how to download and install Python on your operating system. You will also learn how to start the editor and write a code in Python text editor.

The first thing you should know about is setting up a Python environment on your operating system. Python is installed differently on different operating systems. There are currently two available versions of Python which are Python 2 and Python 3. Each programming language keeps evolving as developers add to it new concepts and ideas. Python developers have taken a step further by making regular developments to the language, making it more powerful and versatile as well. You can install both versions of your operating system.

Python 3 Installation

Python is one of the most widely used programming languages that were first launched in 1991. Since then Python has gained immense popularity. It also is considered one of the most popular as well as flexible programming languages. Unlike Linux, Windows operating systems don't have Python pre-installed by default. However, you always can install Python on the Windows operating system is a very easy step.

The prerequisites of installing Python 3 on your Windows system are that you must have Windows 10 installed on the system and have admin privileges. Also, you should have a command prompt running on the system. The next prerequisite is the availability of a remote desktop app. You can use it if you are planning to install Python on a remote Windows server.

The first step in the installation process is selecting the version you want to install. In this case, we are aiming at downloading Python 3. You should visit the official Python website python.org and download Python 3 installer for Windows. Now run the same on your system. As you are just getting started on learning the Python language, I recommend that you download and install Python 2 and Python 3 versions to test old and new projects simultaneously.

Executable Installer

You should download and install Python executable installer on your Windows operating system. The first step in this regard is to open your internet browser and move to the Downloads section of python.org. Make sure that you download and install only the version that has an executable installer available for Windows. The approximate download size is about 25 MB. If your Windows is a 32-bit operating system, you should select Windows x86 executable installer. If it is 64-bit, you should select Windows x86-64 executable installer. However, even if you install the wrong version, it is easy to uninstall it and reinstall it once again.

Run It

When the executable installer of Python has been downloaded, you need to run it. Select Install launcher for all users. This will ensure that all users on your system will be able to use it. Don’t forget to tick Add Python 3.8 to the PATH box. This links the interpreter to the path of execution and facilitates programming. Older versions don’t display this option.

In the next step, click on Install Now. When you have installed it, you will see that you will get Python IDLE and Pip. You also can verify Python on your system by navigating to the directory in which Python was installed or by entering the word Python in the search bar. You also can check it through Command Prompt by the name of the directory.

If you opt to install Python 2, you might not get Pip installed with it. Pip is a powerful package and is very helpful in building software and machine learning models. Therefore, you should make sure that the version you have installed has got Pip installed on it.

You can also confirm if Pip is installed on your operating system or not by opening the Start menu and typing the word cmb in the search bar. The command will open the Command Prompt application in which you can write pip –v. If the pip is installed on your system, you will see a line that starts with pip 19.0.3. Otherwise, the system will say that it doesn’t recognize pip as an external or internal command. When you open the Python interpreter or IDLE, it will have text like the following:

Python 3.8.5 (tags/v3.8.5:580fbb0, Jul 20 2020, 15:57:54) [MSC v.1924 64 bit (AMD64)] on win32

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

>>>

On the top is written the version of Python. If you have installed Python 2, the text will be as follows:

Python 2.7.8 (default, Jun 30 2014, 16:08:48) [MSC v.1500 64 bit (AMD64)] on win32

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

>>>

Chapter Two: Python Variables and Data Types

When you run a file with the extension .py on Python, it suggests that you are running a Python program. The editor runs the program in a Python interpreter. The Python interpreter spots Python keywords, reads them, identifies them, and acts accordingly. Take the example of the print keyword. When the Python interpreter sees this keyword, it can print whatever you put in the parenthesis. When you write code in the Python editor, you will see that the Python editor highlights your code in different colors. That is how it recognizes different types of code. This feature of Python is known as syntax highlighting and is very useful as you learn to write Python programs.

Python Variables

Python variables are like containers that you can fill in with different types of data values. Python differs from other programming languages because it does not have any command for the declaration of a variable. You can create a variable just by assigning it a specific value. In the following example, I will create a variable and assign it some value. Please keep in mind that I am using a Python interpreter for the initial stages. I will move on to the Python editor in the later chapters.

```
>>> greetings = "Hello, I am here to learn Python."

>>> print(greetings)

Hello, I am here to learn Python.

>>>
```

The name of the variable is *greetings*. The variable holds a full sentence as its unique value. The print keyword, as I said, is used to display the value of the variable. The value of a variable can be easily replaced if you pack it up with another value. See the following example.

```
>>> greetings = "Hello, I am here to learn Python."

>>> print(greetings)

Hello, I am here to learn Python.

>>>

>>> greetings = "This is the world of Python."
```

```
>>> print(greetings)
This is the world of Python.
>>>
```

Naming Variables

When you are naming a variable, you must adhere to a couple of guidelines and rules. If you break some of the rules, you will see errors in the code. Even if missing out on the rules does not trigger errors, breaking the rules will make the code look vague and abstract. The success of a programmer is that his code should be easy to understand and read. You must keep the following rules in mind when you are creating variables.

- The names of variables may contain numbers, letters, and underscores. They may start with an underscore and a letter, but they cannot start with a number. For example, a variable can be written as greetings_77 but it cannot be written as 77_greetings.
- You must avoid the use of Python keywords while naming a variable. For example, you cannot name a variable as print because print is a Python keyword.
- If you can keep the names of variables as much descriptive as possible, it will help you better read and understand the code when you come back to at a later stage. For example, greetings are better than simply writing g.

Writing variable names is a continuous practice in Python. You will learn how to create better variable names as you move further into the world of coding.

When you are creating variables, it is common to get errors. You should make sure that you are not misspelling different words. Take a look at the possible errors and try to learn how to avoid them and fix them.

I will use the same variable greetings to show how you can end up getting an error message.

```
>>> greetings = "This is the world of Python."
>>> print(greeting)
```

```
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    print(greeting)
NameError: name 'greeting' is not defined
>>>
```

All I did was writing the wrong spelling in the code. What makes Python different from other programming languages is that it displays the error message which tells you the exact line on which you have made the error. This helps rectify the errors and clean the code.

By writing the wrong spelling, I don’t suggest that Python recognizes the wrong or right spelling. I am trying to suggest that it reads the name of the variable and matches it with the command you have entered. In case of a mismatch, the error message pops up. So, even if you misspell the word while you allot a name to a variable, Python does not declare it an error.

```
>>> greetings = "This is the world of Python."
>>> print(gretings)
This is the world of Python.
>>>
```

I have deliberately misspelled the word greetings but still, Python read it and ran the code. A lot of programming errors happen just because of wrong spellings. You may call them typos. They can be easily managed if you read your code with close attention.

Python Datatypes

There are different types of data that you have to use in programming. Some are simple texts while others are integers or lists. Python supports a diversity of datatypes. However, you have to designate the datatype when you create one so that Python can recognize it easily and process it as per your wishes. In the following section, I will shed light on several data types that I will be using in various code snippets in the book.

Datatypes, in a brief look, are as under:

```
#This is Python list
```

```
>>> a = ["tomato, potato, garlic, ginger, pumpkin"]
>>> print(a)
['tomato, potato, garlic, ginger, pumpkin']
>>> #This is python frozenset
>>> a = frozenset({"tomato, potato, garlic, ginger, pumpkin"})
>>> print(a)
frozenset({'tomato, potato, garlic, ginger, pumpkin'})
>>> #This is python set
>>> a = {"tomato, potato, garlic, ginger, pumpkin"}
>>> print(a)
{'tomato, potato, garlic, ginger, pumpkin'}
>>> #This is python tuple
>>> a = ("tomato, potato, garlic, ginger, pumpkin")
>>> print(a)
tomato, potato, garlic, ginger, pumpkin
>>> #This is a dictionary
>>> a = {"fruit" : "tomato", "veg: potato", "veg" : "garlic", "veg" : "ginger", "veg" : "pumpkin"}
SyntaxError: invalid syntax
>>> a = {"fruit" : "tomato", "veg": "potato", "veg" : "garlic", "veg" : "ginger", "veg" : "pumpkin"}
>>> print(a)
{'fruit': 'tomato', 'veg': 'pumpkin'}
>>> a = {"fruit" : "tomato", "veg: potato", "veg1" : "garlic", "veg2" : "ginger", "veg3" : "pumpkin"}
SyntaxError: invalid syntax
>>> a = {"fruit" : "tomato", "veg": "potato", "veg1" : "garlic", "veg2" :
```

"ginger", "veg3" : "pumpkin"}

>>> print(a)

{'fruit': 'tomato', 'veg': 'potato', 'veg1': 'garlic', 'veg2': 'ginger', 'veg3': 'pumpkin'}

In the above I have turned a string into different datatypes. In the dictionary section of the code, you will see some errors. One error is because of a missing quote marks while the other error is because of similar dictionary names. Dictionaries are important as they allow users to store their important data in the form of pairs. They are used in a number of ways.

Python Strings

The first data type is Python strings. Most of the Python programs have to collect and process the data, store it, and use it. One of the most common datatypes is known as strings. They appear to be simple at first glance, however they can be used in several ways. A string is popularly written in the form of characters. Anything that comes inside the quotation marks is dubbed as a string. You can use single or double quotes to create a string. The flexibility of using different types of quote marks is not without reason. It serves several purposes. See that in the following code snippet.

While single quotes allow you to write the code easily, double quotes allow you to use apostrophes in the text. In the following example, you will learn to use both types of quotes and their dos and don’ts.

>>> a = "I am learning Python."

>>> print(a)

I am learning Python.

>>> a = 'I am learning Python.'

>>> print(a)

I am learning Python.

>>> a = "I am learning Tack's book on Python."

>>> print(a)

I am learning Tack's book on Python.

```
>>> a = 'I am learning Tacky's book on Python.'
SyntaxError: invalid syntax
>>>
```

Strings are very interesting if you get a full grasp of the concept. You can change the text into lower and upper texts. I will use the same string example and experiment on it to see how we can change its case.

```
>>> a = "I am learning Python."
>>> print(a.title())
I Am Learning Python.
>>> print(a.upper())
I AM LEARNING PYTHON.
>>> print(a.lower())
i am learning python.
>>>
```

One of the easiest tasks is changing the case as you have seen. There are three keywords: upper, lower, and title cases to change any string you have created for your code. You can see in the code that all the three keywords are accompanied by () brackets. It is called a method. A method can be defined as a kind of action that Python performs on a certain piece of data. Even a character as little as a dot is meaningful in Python coding. The dot that comes after the string's name directs Python to deploy the method, which can be the title, lower, upper cases. Each method has parenthesis at the end to fill in additional information. I have left the parenthesis empty in the above-mentioned code because I did not have to fill it in with additional information. However, in the next few chapters, I will explain how you can use the parenthesis to perform different types of tasks. They are quite interesting if you can use it in the right way.

Out of all the methods, the lower() method is specifically used to store data in Python programs. You might have browsed a website that asked you to write in small letters when you sought to enter some information on the database. Still, you very possibly might have entered the information in a capital case. The lower() method helps convert the strings into the lower case, even if a

user like you entered the information in the capital case.

Many Python programs invite data for storage purposes and then use it. Sometimes you have two or more strings that you have to combine into one. The process is known as string concatenation. For example, you can combine the name of the state and the country's name after you receive them independently. The odds are high that you invite the users' information separately because usually you have to create two columns on your interface to facilitate the users. When you receive them separately, you can combine them with a simple method. See the following example.

```
>>> state_name = "california"

>>> country_name = "United States"

>>> Location_info = state_name + " " + country_name

>>> print(Location_info)

california United States

>>>
```

You can see that there is something wrong with the code. The display is good, but it is not neat. We can add a comma to the code and see how we can use it. From here, I will shift from Python IDLE to Python text editor to give you a feel of how you will write a program. Let us see how to switch from Python IDLE to the text editor.

The first step is to open Python IDLE or interpreter. Go to the File menu and click New File. A new window will pop up on your computer screen. This is a Python text editor. You can write the code on it. When you are done writing the code, you should click on Run on the top menu bar. The editor will ask you to save the code first and then run it. You need to save the code to your desired location and Python will run your code. One important point to note is that when you run the code, a new window of Python IDLE will pop up to display the result of the code. In the next code samples, you will see the code that is written in Python editor first, and attached to its tail will be the result of the code. The second result will start from the word Restart. This is how you will easily learn it and practice it on your computer system. Let us jump to the Python editor now.

```
state_name = "California"
```

```
country_name = "United States"
Location_info = state_name + " , " + country_name
print(Location_info)
= RESTART: C:/Users/saifia computers/Desktop/sample.py
California, United States
>>>
```

This code is neat and clean as I have added a comma to it. The most important thing to learn from this code sample is the plus operator used to combine the two strings. We can concatenate more than two strings as well. I will now add the name of the place to the concatenated string.

```
place_name = "Silicon Valley"
state_name = "California"
country_name = "United States"
location_info = place_name + " , " + state_name + " , " + country_name
print(location_info)
= RESTART: C:/Users/saifia computers/Desktop/sample.py
Silicon Valley, California , United States
>>>
```

I will now use the same code to display a message to the user after he has entered his location information.

```
place_name = "Silicon Valley"
state_name = "California"
country_name = "United States"
location_info = place_name + " , " + state_name + " , " + country_name
print("Hi, I want to visit " + location_info.title() + " in the next month.")
= RESTART: C:/Users/saifia computers/Desktop/sample.py
Hi, I want to visit Silicon Valley, California, United States in the next month.
```

>>>

Let change the case of the strings and see how it works in the program.

```
place_name = "Silicon Valley"
state_name = "California"
country_name = "United States"
location_info = place_name + " , " + state_name + " , " + country_name
print("Hi, I want to visit " + location_info.upper() + " in the next month.")
print("Hi, I want to visit " + location_info.lower() + " in the next month.")
```

= RESTART: C:/Users/saifia computers/Desktop/sample.py

Hi, I want to visit SILICON VALLEY, CALIFORNIA, UNITED STATES in the next month.

Hi, I want to visit silicon valley, california, united states in the next month.

>>>

You always have the option of creating a concatenated string and storing it in a variable so that you can use it later on as per your needs.

```
place_name = "Silicon Valley"
state_name = "California"
country_name = "United States"
location_info = place_name + " , " + state_name + " , " + country_name
info = "Hi, I want to visit " + location_info.title() + " in the next month."
print(info)
```

= RESTART: C:/Users/saifia computers/Desktop/sample.py

Hi, I want to visit Silicon Valley , California , United States in the next month.

>>>

Python allows you to neatly format your strings by adding tabs, spaces and other symbols.

```
place_name = "Silicon Valley"
state_name = "California"
country_name = "United States"
location_info = place_name + " , " + state_name + " , " + country_name
info = "\tHi, I want to visit " + location_info.title() + "\t in the next month."
print(info)
= RESTART: C:/Users/saifia computers/Desktop/sample.py
        Hi, I want to visit Silicon Valley , California , United States in the next month.
>>>
```

The tab feature has worked perfectly. Now I will use the \n feature to start each word of the sentence on a new line.

```
place_name = "Silicon Valley"
state_name = "California"
country_name = "United States"
location_info = place_name + " , " + state_name + " , " + country_name
info = "\nHi, \nI \nwant \nto \nvisit " + location_info.title() + " \nin the \nnext month."
print(info)
= RESTART: C:/Users/saifia computers/Desktop/sample.py
Hi,
I
want
to
visit Silicon Valley, California, United States
in the
next month.
```

>>>

Python Numbers

Numbers are more often used in programming to do calculations or store data. For example, if you are developing a Python game, you need to keep the scores calculated. Python integers are the simplest form of numbers that you can use. I will do some calculations by using different mathematical operators.

```
a = 3 + 6
print(a)
a = 10 - 5
print(a)
a = 3 * 6
print(a)
a = 18 / 6
print(a)
a = 3 ** 6
print(a)
a = 15 ** 6
print(a)
```

= RESTART: C:/Users/saifia computers/Desktop/sample.py

9

5

18

3.0

729

11390625

>>>

In the next example, I will use multiple operators to see how Python maintains the order of mathematics.

```
a = 3 + 6 * 18
print(a)
a = 10 - 5 + 45
print(a)
a = 3 * 6 + 50 / 30
print(a)
a = 18 / 6 * 50
print(a)
a = (3 * 6 )- 23
print(a)
a = (15 ** 6) + 100
print(a)
= RESTART: C:/Users/saifia computers/Desktop/sample.py
111
50
19.666666666666668
150.0
-5
11390725
>>>
```

The next number type is float. This is the decimal form of integer. See how to use them.

```
a = 3.0 + 6.344 * 18.234
print(a)
a = 10.1 - 5.56 + 45.22
```

```
print(a)
a = 3.45 * 6.3 + 50.09 / 30.1
print(a)
a = 18.22 / 6.1 * 50.89
print(a)
a = (3.21 * 6.23 )- 23.22
print(a)
a = (15.123 ** 6.23) + 100.456
print(a)
= RESTART: C:/Users/saifia computers/Desktop/sample.py
118.67649600000001
49.76
23.399119601328902
152.00259016393443
-3.2216999999999985
22343252.55463075
>>>
```

When you mix up the datatypes without proper procedure, you get an error in return. For example, you have to mix up date with day to display a string statement. This will trigger an error if you do not fill in the code with the right datatype and through the right process.

```
place_name = "Silicon Valley"
state_name = "California"
country_name = "United States"
location_info = place_name + " , " + state_name + " , " + country_name
date = 4
info = "Hi, I want to visit " + location_info.title() + " in the next month on" +
```

```
date + " th."
print(info)
= RESTART: C:/Users/saifia computers/Desktop/sample.py
Traceback (most recent call last):
  File "C:/Users/saifia computers/Desktop/sample.py", line 6, in <module>
    info = "Hi, I want to visit " + location_info.title() + " in the next month on" + date + " th."
TypeError: can only concatenate str (not "int") to str
>>>
```

The error is a type error. Python is unable to recognize the information you have put in. In simple words, the wrong method to fill in the code has confused Python. So, you have to turn the integer into a string first to display it in the right way. The technique is simple, but you will have to memorize it.

```
place_name = "Silicon Valley"
state_name = "California"
country_name = "United States"
location_info = place_name + " , " + state_name + " , " + country_name
date = 4
info = "Hi, I want to visit " + location_info.title() + " on the " + str(date) + "the of July"
print(info)
= RESTART: C:/Users/saifia computers/Desktop/sample.py
Hi, I want to visit Silicon Valley, California, United States on the 4th of July
>>>
```

I have told python that I have to add an integer to the string statement.

Chapter Three: Python Lists and Tuples

This chapter will walk you through the concept of Python lists. I will explain what Python lists are and how you can use them in Python programs. Python lists are one of the most amazing features of Python programming. They allow you to fill in loads of information in a succinct manner. They allow you to pack up tons of information in an easy-to-access format. You can add millions of items to Python lists. Python lists are considered one of the robust features of Python programming.

A Python list is packed up with a streak of items that are adjusted in a specific order. A list can be filled with alphabets and digits. As a list may contain more than one element, the traditional naming practice for lists suggests that you give them a plural name. Let us assume that you are developing a game where a player has to set up an office with necessary items and sell the office to the highest bidder.

officeitems = ['printer', 'scanner', 'fan', 'table', 'chair', 'computer system', 'table lights']

print(officeitems)

= RESTART: C:/Users/saifia computers/Desktop/sample.py

['printer', 'scanner', 'fan', 'table', 'chair', 'computer system', 'table lights']

>>>

The output is alright, except that you do not want your game users to see this output. It should be in an easy-to-digest form. Rather than printing the complete list, you can access certain elements in the list by a simple method. For example, the player in your game wants to see which item has been included in the office set up. To see that he should be able to access different items in the list. Here is the method you can include in your program to help your players confirm the inclusion of different items.

officeitems = ['printer', 'scanner', 'fan', 'table', 'chair', 'computer system', 'table lights']

print(officeitems[0])

print(officeitems[2])

```
print(officeitems[4])
print(officeitems[5])
print(officeitems[6])
= RESTART: C:/Users/saifia computers/Desktop/sample.py
printer
fan
chair
computer system
table lights
>>>
```

When you are writing the code for your game, you may run into a serious problem. Your player may want to access the item no 10 in the list, which does not exist in the first place as there are only seven items on the list. When the player tries to do that, the result will be an index error.

```
officeitems = ['printer', 'scanner', 'fan', 'table', 'chair', 'computer system', 'table lights']
print(officeitems[10])
= RESTART: C:/Users/saifia computers/Desktop/sample.py
Traceback (most recent call last):
  File "C:/Users/saifia computers/Desktop/sample.py", line 2, in <module>
    print(officeitems[10])
IndexError: list index out of range
>>>
```

You can deploy string methods to make the lists look neat and clean. I will use the title, lower and upper case methods to format the items from the list.

```
officeitems = ['printer', 'scanner', 'fan', 'table', 'chair', 'computer system', 'table lights']
print(officeitems[0].title())
```

```
print(officeitems[1].upper())
print(officeitems[2].lower())
= RESTART: C:/Users/saifia computers/Desktop/sample.py
Printer
SCANNER
fan
>>>
```

One important point to note regarding lists is that the first item in the list tends to start from zero. If you fill it in with 1, it means you are trying to access the second item on the list. If you want to access the fifth item on the list, you will have to use index number 3. There is another way to access items on the list. You can access the item by using negative indices. In the following example, I will access items both from positive and negative indices.

```
officeitems = ['printer', 'scanner', 'fan', 'table', 'chair', 'computer system', 'table lights']
print(officeitems[0])
print(officeitems[1])
print(officeitems[2])
print(officeitems[-1])
print(officeitems[-2])
print(officeitems[-3])
= RESTART: C:/Users/saifia computers/Desktop/sample.py
printer
scanner
fan
table lights
computer system
```

chair

>>>

While the positive index starts from the left side, the negative index starts from the right side. It will pick the values from the end of the list and display them to the user.

Let us make the game more interesting by adding statements and using items from the list to build those statements. Each time your player buys something from the market and adds it to the office, he will receive a message on the screen that informs him how much he has achieved. I will use individual values from the list and apply the method of concatenation to create a message.

officeitems = ['printer', 'scanner', 'fan', 'table', 'chair', 'computer system', 'table lights']

comment = "Dear player! You have successfully purchased a " + officeitems[2].title() + "."

print(comment)

comment = "Dear player! You have successfully purchased a " + officeitems[0].upper() + "."

print(comment)

comment = "Dear player! You have successfully purchased a " + officeitems[2].lower() + "."

print(comment)

= RESTART: C:/Users/saifia computers/Desktop/sample.py

Dear player! You have successfully purchased a Fan.

Dear player! You have successfully purchased a PRINTER.

Dear player! You have successfully purchased a fan.

>>>

You can use the same items in different ways. All it needs a pinch of creativity. When the player has installed the items in the office, you can display a message on the screen to show that the items are operational.

```
officeitems = ['printer', 'scanner', 'fan', 'table', 'chair', 'computer system', 'table lights']

comment = "Dear player! The " + officeitems[2].title() + " is operational."

print(comment)

comment = "Dear player! The " + officeitems[0].title() + " is operational."

print(comment)

comment = "Dear player! The " + officeitems[6].title() + " are operational."

print(comment)

= RESTART: C:/Users/saifia computers/Desktop/sample.py

Dear player! The Fan is operational.

Dear player! The Printer is operational.

Dear player! The Table Lights are operational.

>>>
```

Modifying Lists

When you have created a list, you can easily change its items, add more items to it, and remove certain items. In this sense, lists are very flexible. Most of the lists that you create are dynamic. You can offer your player a choice to add different items to your list and remove items from the list of office items to increase or decrease the office's value. The faster he sells or the bigger he sells matter for the overall gaming score of the player.

```
officeitems = ['printer', 'scanner', 'fan', 'table', 'chair', 'computer system', 'table lights']

print(officeitems)

# I will change the value of different items at different indices

officeitems[0] = 'water dispenser'

print(officeitems)

= RESTART: C:/Users/saifia computers/Desktop/sample.py

['printer', 'scanner', 'fan', 'table', 'chair', 'computer system', 'table lights']
```

['water dispenser', 'scanner', 'fan', 'table', 'chair', 'computer system', 'table lights']

>>>

Append Method

In the first line, I have displayed the original list. In the next line of code, I will change the first item's value and redisplay the list. The player has successfully replaced the item to boost the sale value of the office. You also can allow your player to keep adding more items to the office to beef up the value. One of the easiest and the most amazing way to add items to a list is by using the append method. This method is the simplest of adding new elements to a list. When you apply the append method, the element you want to add will be added to the end of the list. The player does not have a choice to add it to his favorite index.

officeitems = ['printer', 'scanner', 'fan', 'table', 'chair', 'computer system', 'table lights']

print(officeitems)

officeitems.append('water dispenser')

print(officeitems)

officeitems.append('multimedia projector')

print(officeitems)

officeitems.append('air conditioner')

print(officeitems)

officeitems.append('laptop')

print(officeitems)

= RESTART: C:/Users/saifia computers/Desktop/sample.py

['printer', 'scanner', 'fan', 'table', 'chair', 'computer system', 'table lights']

['printer', 'scanner', 'fan', 'table', 'chair', 'computer system', 'table lights', 'water dispenser']

['printer', 'scanner', 'fan', 'table', 'chair', 'computer system', 'table lights', 'water dispenser', 'multimedia projector']

['printer', 'scanner', 'fan', 'table', 'chair', 'computer system', 'table lights', 'water dispenser', 'multimedia projector', 'air conditioner']

['printer', 'scanner', 'fan', 'table', 'chair', 'computer system', 'table lights', 'water dispenser', 'multimedia projector', 'air conditioner', 'laptop']

>>>

The append method helps you build lists in perfect order. This is the most efficient way to build lists in a perfectly dynamic way. You can even start appending elements to an empty list. This is how your player will have more freedom to start right from scratch. He will see an empty office and equip it with the items he wants to add to it.

While the append method adds elements to the end of your lists, you can use the insert method to add items to the position of your choice. The first you need to do is to specify the index of each new element. See the following example.

officeitems = ['printer', 'scanner', 'fan', 'table', 'chair', 'computer system', 'table lights']

print(officeitems)

officeitems.insert(0, 'air conditioner')

print(officeitems)

officeitems.insert(2, 'multimedia projector')

print(officeitems)

officeitems.insert(3, 'water dispenser')

print(officeitems)

= RESTART: C:/Users/saifia computers/Desktop/sample.py

['printer', 'scanner', 'fan', 'table', 'chair', 'computer system', 'table lights']

['air conditioner', 'printer', 'scanner', 'fan', 'table', 'chair', 'computer system', 'table lights']

['air conditioner', 'printer', 'multimedia projector', 'scanner', 'fan', 'table', 'chair', 'computer system', 'table lights']

['air conditioner', 'printer', 'multimedia projector', 'water dispenser', 'scanner',

'fan', 'table', 'chair', 'computer system', 'table lights']

>>>

Del Method

There may be occasions in the game when a player wants to remove certain items from the office to adjust the office to his desires. You can easily remove different items from a list by a simple method. The first method is that of the del statement. Let us check how you can add this method to your code to make the game more interactive.

officeitems = ['printer', 'scanner', 'fan', 'table', 'chair', 'computer system', 'table lights']

print(officeitems)

del officeitems[0]

print(officeitems)

del officeitems[1]

print(officeitems)

del officeitems[2]

print(officeitems)

= RESTART: C:/Users/saifia computers/Desktop/sample.py

['printer', 'scanner', 'fan', 'table', 'chair', 'computer system', 'table lights']

['scanner', 'fan', 'table', 'chair', 'computer system', 'table lights']

['scanner', 'table', 'chair', 'computer system', 'table lights']

['scanner', 'table', 'computer system', 'table lights']

>>>

Pop Method

While the del method demands from you that you mention the index number, you always have the choice to use another method, the pop() method, to remove items from your lists. This is helpful if you want to give your players the freedom to randomly remove items from the office setup.

The pop() method tends to eject items from the end of the list. However, you

can use the item later on. This is why the pop() method is different from the del() method. Imagine that the office items are stacked up one upon another and you get the opportunity to pick and throw out from the top one by one. This is how the pop() method works.

I will use the pop() method until the list stands empty to show how a player can empty an office off the items if he wants to.

officeitems = ['printer', 'scanner', 'fan', 'table', 'chair', 'computer system', 'table lights']

print(officeitems)

popped_officeitems = officeitems.pop()

print(officeitems)

print(popped_officeitems)

popped_officeitems = officeitems.pop()

print(officeitems)

print(popped_officeitems)

popped_officeitems = officeitems.pop()

print(officeitems)

print(popped_officeitems)

popped_officeitems = officeitems.pop()

print(officeitems)

print(popped_officeitems)

popped_officeitems = officeitems.pop()

print(officeitems)

print(popped_officeitems)

popped_officeitems = officeitems.pop()

print(officeitems)

print(popped_officeitems)

popped_officeitems = officeitems.pop()

```
print(officeitems)
print(popped_officeitems)
= RESTART: C:/Users/saifia computers/Desktop/sample.py
['printer', 'scanner', 'fan', 'table', 'chair', 'computer system', 'table lights']
['printer', 'scanner', 'fan', 'table', 'chair', 'computer system']
table lights
['printer', 'scanner', 'fan', 'table', 'chair']
computer system
['printer', 'scanner', 'fan', 'table']
chair
['printer', 'scanner', 'fan']
table
['printer', 'scanner']
fan
['printer']
scanner
[]
printer
>>>
```

Once there are no more items in the list but you still use the pop() method, it will return an error that will look like the following.

```
Traceback (most recent call last):
  File "C:/Users/saifia computers/Desktop/sample.py", line 24, in <module>
    popped_officeitems = officeitems.pop()
IndexError: pop from empty list
```

In the following section, I will show you how you can use the popped item in the code. If you want your player to explain to his boss why he removed an

item from the item, you can add another line of code to your program.

officeitems = ['printer', 'scanner', 'fan', 'table', 'chair', 'computer system', 'table lights']

print(officeitems)

popped_officeitems = officeitems.pop()

print("I have sold " + popped_officeitems.title() + " because it was not adding the desired value to the office.")

= RESTART: C:/Users/saifia computers/Desktop/sample.py

['printer', 'scanner', 'fan', 'table', 'chair', 'computer system', 'table lights']

I have sold Table Lights because it was not adding the desired value to the office.

>>>

Unlike what most people think, the pop() method allows you to remove items from a list at specific positions of your choice. You have to fill in the parenthesis with the desired index number.

officeitems = ['printer', 'scanner', 'fan', 'table', 'chair', 'computer system', 'table lights']

print(officeitems)

popped_officeitems = officeitems.pop(3)

print("I have sold the " + popped_officeitems.title() + " because it was not adding the desired value to the office.")

= RESTART: C:/Users/saifia computers/Desktop/sample.py

['printer', 'scanner', 'fan', 'table', 'chair', 'computer system', 'table lights']

I have sold the Table because it was not adding the desired value to the office.

>>>

Remove Method

Another method to remove items from a list by using the remove method. Many times you do not know the specific position of a value that you want to

remove. In that case, you can remove the item by using the value of that item. To do that you have to fill in the parenthesis of the remove() method with the value.

```
officeitems = ['printer', 'scanner', 'fan', 'table', 'chair', 'computer system', 'table lights']

print(officeitems)

officeitems.remove('scanner')

print(officeitems)

officeitems.remove('table')

print(officeitems)
```

```
= RESTART: C:/Users/saifia computers/Desktop/sample.py

['printer', 'scanner', 'fan', 'table', 'chair', 'computer system', 'table lights']

['printer', 'fan', 'table', 'chair', 'computer system', 'table lights']

['printer', 'fan', 'chair', 'computer system', 'table lights']

>>>
```

Just like you used the popped value later on in the code, you can also use the removed value from the item.

```
officeitems = ['printer', 'scanner', 'fan', 'table', 'chair', 'computer system', 'table lights']

print(officeitems)

removeditem = 'scanner'

officeitems.remove(removeditem)

print("I have sold the " + removeditem.title() + " because it was not adding the desired value to the office.")
```

```
= RESTART: C:/Users/saifia computers/Desktop/sample.py

['printer', 'scanner', 'fan', 'table', 'chair', 'computer system', 'table lights']

I have sold the Scanner because it was not adding the desired value to the office.
```

>>>

List Organization

More often when you create lists, they flow in a kind of unpredictable order. You cannot always control the order in which the users provide data to the program. However, you can bring that information into perfect order later on. It may happen more often than you want to make the information presentable. This is the reason you should bring it into perfect order. There are several ways by which you can order lists in Python.

The sort() Method

The sort method of Python makes it fun to sort a list. I will use the same list of officeitems and experiment on it to see how the list gets organized. The sort() method sorts lists in an alphabetical order.

```
officeitems = ['printer', 'scanner', 'fan', 'table', 'table lights']

print(officeitems)

officeitems.sort()

print(officeitems)

= RESTART: C:/Users/saifia computers/Desktop/sample.py

['printer', 'scanner', 'fan', 'table', 'table lights']

['fan', 'printer', 'scanner', 'table', 'table lights']

>>>
```

The sort() method has changed the list into a perfect alphabetical order. You cannot revert to the original order once you have sorted your list. The sort() can also be used to order the list in the reverse alphabetical order.

```
officeitems = ['printer', 'scanner', 'fan', 'table', 'table lights']

print(officeitems)

officeitems.sort(reverse=True)

print(officeitems)

= RESTART: C:/Users/saifia computers/Desktop/sample.py

['printer', 'scanner', 'fan', 'table', 'table lights']
```

['table lights', 'table', 'scanner', 'printer', 'fan']

>>>

Coupled with the sort() method is the sorted() function. While the sort() method permanently reorders a list, the sorted() function makes temporary changes.

officeitems = ['printer', 'scanner', 'fan', 'table', 'table lights']

print(officeitems)

print(sorted(officeitems))

print(officeitems)

= RESTART: C:/Users/saifia computers/Desktop/sample.py

['printer', 'scanner', 'fan', 'table', 'table lights']

['fan', 'printer', 'scanner', 'table', 'table lights']

['printer', 'scanner', 'fan', 'table', 'table lights']

>>>

You can see that the sorted() function temporarily changed the order of the list. When I printed the list without the sorted() function, it comes back into its original order.

Index Errors

As errors are common in coding, you may run into index errors in lists. One of the most common types of errors with lists is the index error. You may confront this type of error more often when you work on lists. However, you can easily avoid that if you know at which point the index starts. Let us see first how the error message looks like.

officeitems = ['printer', 'scanner', 'fan', 'table', 'table lights']

print(officeitems[5])

= RESTART: C:/Users/saifia computers/Desktop/sample.py

Traceback (most recent call last):

  File "C:/Users/saifia computers/Desktop/sample.py", line 2, in <module>

    print(officeitems[5])

IndexError: list index out of range

>>>

The list contains five items, but when I invoke the index number 5, I get an error message. Just as for strings, the index for lists also starts at zero.

Index errors come up more often if the list is a long one. The best practice to work with long lists is to know the exact length of the list. Once you know the length, you can sort out each item's index number in the list. See the following method to check the length of the list.

```
officeitems = ['printer', 'scanner', 'fan', 'table', 'table lights']

len(officeitems)
```

Create A Loop

Python lists can be operated with loops. Coming back to the game. If you want your player to have the option of displaying all the items that he has set up in his office, you can run a loop through your list. This is helpful if you have added another character to your game who asks the player about the total number of items that he has bought for the office. Each item in the list will be neatly displayed. The *for* loop in Python will repeat the same action with each item.

I will use the same list of office items and print each item by looping the list with a for loop. Let us now create and build a *for* loop to print out each item you have bought for the office.

```
officeitems = ['printer', 'scanner', 'fan', 'table', 'table lights']

for officeitem in officeitems:

    print(officeitem)
```

= RESTART: C:/Users/saifia computers/Desktop/sample.py

printer

scanner

fan

table

table lights

>>>

Python *for* loop has looped through and printed each item on the list. When you are writing this code, you may hit an error which can be hard to explain because there will be no exact error message on the interpreter screen. See the following example.

```
officeitems = ['printer', 'scanner', 'fan', 'table', 'table lights']

for officeitem in officeitems:

    print(officeitems)

= RESTART: C:/Users/saifia computers/Desktop/sample.py

['printer', 'scanner', 'fan', 'table', 'table lights']

['printer', 'scanner', 'fan', 'table', 'table lights']

['printer', 'scanner', 'fan', 'table', 'table lights']

['printer', 'scanner', 'fan', 'table', 'table lights']

['printer', 'scanner', 'fan', 'table', 'table lights']

>>>
```

All I did was to add an s to officeitem in the last line of code. Python interpreted it differently and looped through the entire list and displayed it repetitively. Instead of getting an error message, Python changes the results.

What exactly I did in the code. In the first line of code, I have defined a list, namely officeitems. In the next line, I have defined a *for* loop. This line instructs Python to pick a name from the list and store it in the newly created variable officeitem. It is not necessary to name a variable in this way, but it is easy to remember. You can name it as you like. In the next line, I told Python to print each name that I had stored in the new variable. Python repeats lines for each item in the list. To kill the confusion about the name of the variable, I will now change the variable's name.

```
officeitems = ['printer', 'scanner', 'fan', 'table', 'table lights']

for things in officeitems:

    print(things)

= RESTART: C:/Users/saifia computers/Desktop/sample.py
```

printer

scanner

fan

table

table lights

>>>

Looping is an important concept in computer programming because it is used to automate some recitative tasks. If your list is packed up with a million items, Python loops will repeat the steps a million times and in a very fast manner.

The Python *for* loop is amazing because it allows you to experiment with the office items quickly. You have to set the code right and the entire list will be fully automated. I will now take each item from the list and display a message on the Python interpreter screen. If you want your player to tell his boss he has purchased each item at a discount price and from a quality production house, you can slightly change the code and display the message most uniquely.

```
officeitems = ['printer', 'scanner', 'fan', 'table', 'table lights']

for officeitem in officeitems:

    print("I have purchased the " + officeitem.lower() + " at a discount price from TopQuality Productions.")
```

= RESTART: C:/Users/saifia computers/Desktop/sample.py

I have purchased the printer at a discount price from TopQuality Productions.

I have purchased the scanner at a discount price from TopQuality Productions.

I have purchased the fan at a discount price from TopQuality Productions.

I have purchased the table at a discount price from TopQuality Productions.

I have purchased the table lights at a discount price from TopQuality Productions.

>>>

So, this is getting interesting now. This is how you can develop your game in a brilliantly interactive manner. If you want the player to speak more than one line, you can pair up more sentences to the reply. Each line will be executed in the order you write it in the code. I will now add a second line of code to the response of the player.

```
officeitems = ['printer', 'scanner', 'fan', 'table', 'table lights']

for officeitem in officeitems:

    print("I have purchased the " + officeitem.lower() + " at a discount price from TopQuality Productions.")

    print ("I hope the " + officeitem.lower() + " will add more value to the office.")
```

= RESTART: C:/Users/saifia computers/Desktop/sample.py

I have purchased the printer at a discount price from TopQuality Productions.

I hope the printer will add more value to the office.

I have purchased the scanner at a discount price from TopQuality Productions.

I hope the scanner will add more value to the office.

I have purchased the fan at a discount price from TopQuality Productions.

I hope the fan will add more value to the office.

I have purchased the table at a discount price from TopQuality Productions.

I hope the table will add more value to the office.

I have purchased the table lights at a discount price from TopQuality Productions.

I hope the table lights will add more value to the office.

>>>

This is how you can add a hundred lines if your program or game requires that. You also can add a finishing note to the end of a block of code. The finishing block of code executes without repetition. I will now add a reply from the boss in the game who has heard what the player said about purchasing items.

```
officeitems = ['printer', 'scanner', 'fan', 'table', 'table lights']

for officeitem in officeitems:

    print("I have purchased the " + officeitem.lower() + " at a discount price from TopQuality Productions.")

    print ("I hope the " + officeitem.lower() + " will add more value to the office.")

print("Thanks for making the purchase. Hope you will be able to sell out the office this month. After all, you won't like to be deprived of your bonus.")

= RESTART: C:/Users/saifia computers/Desktop/sample.py

I have purchased the printer at a discount price from TopQuality Productions.

I hope the printer will add more value to the office.

I have purchased the scanner at a discount price from TopQuality Productions.

I hope the scanner will add more value to the office.

I have purchased the fan at a discount price from TopQuality Productions.

I hope the fan will add more value to the office.

I have purchased the table at a discount price from TopQuality Productions.

I hope the table will add more value to the office.

I have purchased the table lights at a discount price from TopQuality Productions.

I hope the table lights will add more value to the office.

Thanks for making the purchase. Hope you will be able to sell out the office this month. After all, you won't like to be deprived of your bonus.

>>>
```

All I did was to remove the space before the print statement by which I want to end the *for* loop. This explains how crucial a role spaces play in Python programming. You miss out on an indentation and you will see an error on the screen.

```
officeitems = ['printer', 'scanner', 'fan', 'table', 'table lights']
```

for officeitem in officeitems:

print(officeitem)

Expected an indented block

List Slicing

You can slice a list to work with a portion of a list. This is helpful if you are building a long list of items and you have to work only with a handful of items. I will use the same list for slicing.

officeitems = ['printer', 'scanner', 'fan', 'table', 'table lights']

print(officeitems[1:4])

= RESTART: C:/Users/saifia computers/Desktop/sample.py

['scanner', 'fan', 'table']

>>>

The result contains only the items that I have sliced out of the original list. The output comes in the form of the original structure of the list. The sliced part of the list can also be named as a subset of the list. If you omit the index's start, the subset of the list will start from the first item.

officeitems = ['printer', 'scanner', 'fan', 'table', 'table lights']

print(officeitems[:4])

= RESTART: C:/Users/saifia computers/Desktop/sample.py

['printer', 'scanner', 'fan', 'table']

>>>

Similarly, you can omit the second half of the range index.

officeitems = ['printer', 'scanner', 'fan', 'table', 'table lights']

print(officeitems[2:])

= RESTART: C:/Users/saifia computers/Desktop/sample.py

['fan', 'table', 'table lights']

>>>

The negative indexing is also available for Python lists.

```
officeitems = ['printer', 'scanner', 'fan', 'table', 'table lights']
print(officeitems[-2:])
= RESTART: C:/Users/saifia computers/Desktop/sample.py
['table', 'table lights']
>>>
```

You also can loop through the subset of a list or the slice of a list just like we did with a complete list.

```
officeitems = ['printer', 'scanner', 'fan', 'table', 'table lights']
print("Here are the items that have cost double the value of others.")
for officeitem in officeitems[:4]:
    print(officeitem.title())
= RESTART: C:/Users/saifia computers/Desktop/sample.py
Here are the items that have cost double the value of others.
Printer
Scanner
Fan
Table
>>>
```

Copying

You can create a copy of the list based on the original list. This is helpful if you have packed up customer data in a list and you want to save it elsewhere to secure it in the wake of data breaches or any other kind of cyberattack. The most common way of making a copy of the list is to create a slice with no starting or ending values. This slice instructs Python to slice the list right from the starting value to the ending value. This is how you end up creating a copy of your list.

```
officeitems = ['printer', 'scanner', 'fan', 'table', 'table lights']
c_officeitems = officeitems[:]
```

```
print("Here is the list of items that I have purchased.")
print(officeitems)
print("\nHere is the exact copy of the same items.")
print(c_officeitems)
= RESTART: C:/Users/saifia computers/Desktop/sample.py
Here is the list of items that I have purchased.
['printer', 'scanner', 'fan', 'table', 'table lights']
Here is the exact copy of the same items.
['printer', 'scanner', 'fan', 'table', 'table lights']
>>>
```

Let us now check if you really have two lists and that if removing or modifying one list will affect the copied list or not.

```
officeitems = ['printer', 'scanner', 'fan', 'table', 'table lights']
c_officeitems = officeitems[:]
officeitems.append('computer system')
c_officeitems.append('blinds')
print("Here is the list of items that I have purchased.")
print(officeitems)
print("\nHere is the new list of office items.")
print(c_officeitems)
= RESTART: C:/Users/saifia computers/Desktop/sample.py
Here is the list of items that I have purchased.
['printer', 'scanner', 'fan', 'table', 'table lights', 'computer system']
Here is the new list of office items.
['printer', 'scanner', 'fan', 'table', 'table lights', 'blinds']
>>>
```

You can see that the two lists exist independently and that you can modify them separately. This is helpful if you are building data lists for a big financial institution.

You may create a copy of the list without using the slice method but you will not be able to modify the two lists independently of each other. They will not be able to exist independently of each other.

```
officeitems = ['printer', 'scanner', 'fan', 'table', 'table lights']

c_officeitems = officeitems

officeitems.append('computer system')

c_officeitems.append('blinds')

print("Here is the list of items that I have purchased.")

print(officeitems)

print("\nHere is the new list of office items.")

print(c_officeitems)
```

```
= RESTART: C:/Users/saifia computers/Desktop/sample.py

Here is the list of items that I have purchased.

['printer', 'scanner', 'fan', 'table', 'table lights', 'computer system', 'blinds']

Here is the new list of office items.

['printer', 'scanner', 'fan', 'table', 'table lights', 'computer system', 'blinds']

>>>
```

You can see that each list retained both newly added items. I will now add one item to one list only and see if the copied list retains that item or not.

```
officeitems = ['printer', 'scanner', 'fan', 'table', 'table lights']

c_officeitems = officeitems

officeitems.append('computer system')


print("Here is the list of items that I have purchased.")

print(officeitems)
```

```
print("\nHere is the new list of office items.")
print(c_officeitems)
= RESTART: C:/Users/saifia computers/Desktop/sample.py
Here is the list of items that I have purchased.
['printer', 'scanner', 'fan', 'table', 'table lights', 'computer system']
Here is the new list of office items.
['printer', 'scanner', 'fan', 'table', 'table lights', 'computer system']
>>>
```

So, as you can see that simply copying a list will not do the magic for you. Slicing helps you make the copy you desire for.

Python Tuples

While lists can be modified and are flexible, Python tuples are the opposite. Once you have created a tuple, you cannot change it. A tuple is similar to a list in the sense that you can fill it up with millions of items but at the same time, it is different from a list in the sense that you cannot add, remove or change the items in a tuple. This is helpful if you want to create a list that you do not want to be changed. The values in a tuple that cannot be changed are labeled as immutable in Python. So, you can label a rigid list or a tuple as an immutable list.

In appearance, a tuple looks mostly like a list except that you have to enclose the tuple items inside square brackets. Once you have created a tuple, you can easily access the tuples' items by using the index number. I will add a new feature to the game by creating a list of home items that the same player needs to sell. The difference is that the player cannot change the items of the house except in exceptional circumstances. He has to sell the house the way it is at the moment. He cannot add more items or remove the existing ones to tune the house's value as per the expectations and demands of buyers.

```
homeitems = ('dining table', 'cooking range', 'washing machine', 'refrigerator', 'air conditioner')
print(homeitems)
= RESTART: C:/Users/saifia computers/Desktop/sample.py
```

('dining table', 'cooking range', 'washing machine', 'refrigerator', 'air conditioner')

>>>

Let us see what happens when we try to change the value of an item in the tuple.

```
homeitems = ('dining table', 'cooking range', 'washing machine', 'refrigerator', 'air conditioner')

homeitems[0] = ('bed')

print(homeitems)
```

= RESTART: C:/Users/saifia computers/Desktop/sample.py

Traceback (most recent call last):

File "C:/Users/saifia computers/Desktop/sample.py", line 2, in <module>

homeitems[0] = ('bed')

TypeError: 'tuple' object does not support item assignment

>>>

You see an error because Python does not allow you to modify tuples. However, there is a way out by which you modify a tuple. You can do the modification by assigning new values to the same variable that carries the tuple. In the following code snippet, I will redefine the tuple.

```
homeitems = ('dining table', 'cooking range', 'washing machine', 'refrigerator', 'air conditioner')

print("Original items in the tuple:")

for homeitem in homeitems:

    print(homeitem)

homeitems = ('chairs', 'carpets', 'plates', 'oven')

print("Modified items in the tuple:")

for homeitem in homeitems:

    print(homeitem)
```

= RESTART: C:/Users/saifia computers/Desktop/sample.py

Original items in the tuple:

dining table

cooking range

washing machine

refrigerator

air conditioner

Modified items in the tuple:

chairs

carpets

plates

oven

>>>

Tuples are data structures that can be used to store values that cannot be changed through a program.

Looping

Just like lists, you can create a loop through your tuple.

```
homeitems = ('dining table', 'cooking range', 'washing machine', 'refrigerator', 'air conditioner')
for homeitem in homeitems:
    print(homeitem)
```

= RESTART: C:/Users/saifia computers/Desktop/sample.py

dining table

cooking range

washing machine

refrigerator

air conditioner

>>>

Chapter Four: Python Conditionals

A key part of programming is about examining a set of conditions and then taking appropriate action based on the conditions. The if statement in Python lets you test the state of a program and act on it appropriately. This chapter will walk you through the process of writing conditionals and checking them as well. The chapter will encompass simple and complex if statements. I will explain how you can pair up an if statement with lists.

```
homeitems = ['dining table', 'cooking range', 'washing machine', 'refrigerator', 'air conditioner']

for homeitem in homeitems:
    if homeitem == 'dining table':
        print(homeitem.upper())
    else:
        print(homeitem.title())
```

```
= RESTART: C:/Users/saifia computers/Desktop/sample.py
DINING TABLE
Cooking Range
Washing Machine
Refrigerator
Air Conditioner
>>>
```

The simplest conditional test tends to check if a particular variable's value stands equal to your value of interest. Sometimes you need to check if a value exists in a list or not. The player in the game might want to check if a certain item has been purchased or not before he lists the office for sale.

```
officeitems = ['printer', 'scanner', 'fan', 'table', 'table lights']

'scanner' in officeitems
```

You can use the conditional statements to check if a certain item appears in a

list. You also can use the item to display a message or comment.

```
officeitems = ['printer', 'scanner', 'fan', 'table', 'table lights']
item = 'scanner'
item1 = 'table'
item2 = 'extension cable'
if item in officeitems:
    print(item.title() + " exists in the office.")
if item1 in officeitems:
    print(item1.title() + " exists in the office.")
if item2 not in officeitems:
    print(item2.title() + " is not exist in the office. You should buy it as soon as possible.")
= RESTART: C:/Users/saifia computers/Desktop/sample.py
Scanner exists in the office.
Table exists in the office.
Extension Cable does not exist in the office. You should buy it as soon as possible.
>>>
```

Let us see what more you can do with the conditional statements. You can add the conditional statement to tell the player when the office will be ready for listing in the game.

```
officeitems = 10
if officeitems >= 10:
    print("You can list the office for sale")
= RESTART: C:/Users/saifia computers/Desktop/sample.py
You can list the office for sale
>>>
```

See the following example with an additional print statement.

```
officeitems = 12
if officeitems >= 10:
    print("You can list the office for sale")
    print("Have you listed it yet.")
= RESTART: C:/Users/saifia computers/Desktop/sample.py
You can list the office for sale
Have you listed it yet?
>>>
```

The if-else Statement

The condition can be interesting if you add to it the else statement to print a statement if the count of items has not matured yet. The if-else statement allows you to test the conditions in both ways. The else statement defines an action when a particular condition fails.

```
officeitems = 11
if officeitems >= 10:
    print("You can list the office for sale")
    print("Have you listed it yet?")
else:
    print("Sorry, you cannot list the office for sale.")
    print("Please buy and add more items to the office and list it again for sale.")
= RESTART: C:/Users/saifia computers/Desktop/sample.py
You can list the office for sale
Have you listed it yet?
>>>
```

As the items are more than 10, the condition has been tested passed. Now I will reduce the count of the items to test the else-statement.

```
officeitems = 9
if officeitems >= 10:
    print("You can list the office for sale")
    print("Have you listed it yet?")
else:
    print("Sorry, you cannot list the office for sale.")
    print("Please buy and add more items to the office and list it again for sale.")
= RESTART: C:/Users/saifia computers/Desktop/sample.py
Sorry, you cannot list the office for sale.
Please buy and add more items to the office and list it again for sale.
>>>
```

The if-elif-else Chain

The elif statement allows you to add one more condition to the block of code.

```
officeitems = 5
if officeitems < 10:
    print("You still can list the office but it will not bring you the desired amount of money.")
elif officeitems < 15:
    print("You can list the office for sale")
    print("Have you listed it yet?")

else:
    print("Sorry, you cannot list the office for sale.")
    print("Please buy and add more items to the office and list it again for sale.")
= RESTART: C:/Users/saifia computers/Desktop/sample.py
```

You still can list the office, but it will not bring you the desired amount of money.

>>>

You can test multiple conditions with conditional statements.

```
officeitems = ['printer', 'scanner', 'fan', 'table', 'table lights']
if 'printer' in officeitems:
    print("I have purchased the printer.")
if 'fan' in officeitems:
    print("I have purchased the fan.")
if 'scanner' in officeitems:
    print("I have purchased the scanner.")
```

= RESTART: C:/Users/saifia computers/Desktop/sample.py

I have purchased the printer.

I have purchased the fan.

I have purchased the scanner.

>>>

When the player has checked if he has purchased all the desired items, the game will give him a green signal to list the office. You can add a print statement at the end of the block of code.

```
officeitems = ['printer', 'scanner', 'fan', 'table', 'table lights']
if 'printer' in officeitems:
    print("I have purchased the printer.")
if 'fan' in officeitems:
    print("I have purchased the fan.")
if 'scanner' in officeitems:
    print("I have purchased the scanner.")
print("\nYou can list the office now.")
```

= RESTART: C:/Users/saifia computers/Desktop/sample.py

I have purchased the printer.

I have purchased the fan.

I have purchased the scanner.

You can list the office now.

>>>

The same code cannot work with an elif statement. It will stop working. See what happens when we remove the simple if statement and add an elif statement.

```
officeitems = ['printer', 'scanner', 'fan', 'table', 'table lights']
if 'printer' in officeitems:
    print("I have purchased the printer.")
elif 'fan' in officeitems:
    print("I have purchased the fan.")
elif 'scanner' in officeitems:
    print("I have purchased the scanner.")
print("\nYou can list the office now.")
```

= RESTART: C:/Users/saifia computers/Desktop/sample.py

I have purchased the printer.

You can list the office now.

>>>

If Statements and Lists

You can pair up if statements with lists. You can actually combine the two and do some amazing things. In the next example, I will pair up a loop with a list to make the game more interactive.

```
officeitems = ['printer', 'scanner', 'fan', 'table', 'table lights']
for officeitem in officeitems:
```

```
    print("I have purchased the " + officeitem + ".")
print("\nBoss: You can list the office for sale now.")
= RESTART: C:/Users/saifia computers/Desktop/sample.py
I have purchased the printer.
I have purchased the scanner.
I have purchased the fan.
I have purchased the table.
I have purchased the table lights.
Boss: You can list the office for sale now.
>>>
```

You have a straightforward result because the code contains a simple for loop. However, you can add a bit more complexity to the code. The boss may ask the employee about a certain item that must be included in the office set up before the office is put on sale. The employee has to handle the situation in the most appropriate way. I will add an if-else statement along with the for loop to make the code more flexible.

```
officeitems = ['printer', 'scanner', 'fan', 'table', 'table lights']
for officeitem in officeitems:
    if officeitem == 'cupboard':
        print("Sorry Boss, I have not purchased it yet.")
    else:
        print("However, I have purchased the " + officeitem + ".")
print("\nBoss: You can list the office for sale only after you purchase the cupboard and set it up in the office.")
= RESTART: C:/Users/saifia computers/Desktop/sample.py
However, I have purchased the printer.
However, I have purchased the scanner.
However, I have purchased the fan.
```

However, I have purchased the table.

However, I have purchased the table lights.

Boss: You can list the office for sale only after purchasing the cupboard and setting it up in the office.

>>>

Now, the Python interpreter checks each office item before it displays the message. The code confirms if the boss has asked for the cupboard. When he asks for the item, he gets a different response from the employee. The else block makes sure that the response is in affirmative for all the other items.

Multiple Lists

Up till now, we have worked with a single list. In this code sample, I will work on multiple lists. I will add a list of optional items that may or may not be purchased. However, purchasing these items will help increase the value of the office.

```
officeitems = ['printer', 'scanner', 'fan', 'table', 'table lights']

optionalitems = ['pen', 'paper', 'drafting pads', 'books', 'water dispenser']

for optionalitem in optionalitems:

    if optionalitem in officeitems:

        print("The office has been set up will all the essential and optional items.")

    else:

        print("Sorry, I have not purchased the " + optionalitem + ".")

print("\nBoss: You can list the office for sale only after you purchase the optional items and set them up in the office.")
```

= RESTART: C:/Users/saifia computers/Desktop/sample.py

Sorry, I have not purchased the pen.

Sorry, I have not purchased the paper.

Sorry, I have not purchased the drafting pads.

Sorry, I have not purchased the books.

Sorry, I have not purchased the water dispenser.

Boss: You can list the office for sale only after purchasing the optional items and setting them up in the office.

>>>

I defined the list of office items and optional items. I created a loop through the optional items to check if they also are added to the office or not. Upon the checking of each item, a message is displayed. In the end, the boss gives his verdict on whether to list the office or not. In the code mentioned above, the office does not contain any item from the list of optional items, therefore it displayed the same message. You can change that by including one or two optional items in the list of office items. See the following code.

```
officeitems = ['printer', 'paper', 'drafting pads', 'scanner', 'fan', 'table', 'table lights']

optionalitems = ['pen', 'paper', 'drafting pads', 'books', 'water dispenser']

for optionalitem in optionalitems:
    if optionalitem in officeitems:
        print("I have purchased the " + optionalitem + ".")
    else:
        print("Sorry, I have not purchased the " + optionalitem + ".")
print("\nBoss: You can list the office for sale only after you purchase all the optional items and set them up in the office.")
```

= RESTART: C:/Users/saifia computers/Desktop/sample.py

Sorry, I have not purchased the pen.

I have purchased the paper.

I have purchased the drafting pads.

Sorry, I have not purchased the books.

Sorry, I have not purchased the water dispenser.

Boss: You can list the office for sale only after purchasing all the optional items and setting them up in the office.

>>>

Chapter Five: Python Dictionaries

This chapter will walk you through the concept of Python dictionaries, which are the most important part of Python coding. You can store information in a dictionary in the form of pairs. You can easily access the information, modify it, and delete it at will. Dictionaries are amazing in the sense that they allow you to store unlimited information. Just like lists, I will explain how you can pair up a dictionary with a loop.

When you have a good grasp of Python dictionaries, you will learn how to model an object with a dictionary's help. Creating a dictionary is simple, but updating it and using it in a code can be tricky. I will move through this chapter step by step. In the first code sample, I will create a simple dictionary.

```
officeitems = {'printer' : 'HP', 'paper': 'A4', 'drafting pads': 'blank', 'scanner': 'hybrid', 'table': 'wood', 'table lights': 'LED'}

print(officeitems)
```

```
= RESTART: C:/Users/saifia computers/Desktop/sample.py

{'printer': 'HP', 'paper': 'A4', 'drafting pads': 'blank', 'scanner': 'hybrid', 'table': 'wood', 'table lights': 'LED'}

>>>
```

There is another way to access and display selected information from a dictionary. You can use one of the pairs' values and use them to access the other value of the pair. See the following code example.

```
officeitems = {'printer' : 'HP', 'paper': 'A4', 'drafting pads': 'blank', 'scanner': 'hybrid', 'table': 'wood', 'table lights': 'LED'}

print(officeitems['printer'])

print(officeitems['paper'])

print(officeitems['table lights'])

print(officeitems['table'])

print(officeitems['drafting pads'])
```

= RESTART: C:/Users/saifia computers/Desktop/sample.py

HP

A4

LED

wood

blank

>>>

Dictionaries are more complex than lists, therefore you need more programming practice to handle them. You can see that a dictionary contains key-value pairs where each key is automatically connected to its value. Each key's value can be a string or an integer or even a list in some cases. It also can be a dictionary in a more complex code form. You have to wrap up a dictionary in curly braces or the dictionary will display an error. A key has directed association with its value.

Accessing values from a dictionary is easy. As you have seen in the above code sample, I tried to access each value with the help of a key or a dictionary. Dictionaries are also very dynamic, and they allow you to add as many key-value pairs to the dictionary as you desire. I will now take an empty dictionary and fill it up with key-value pairs of my choice.

```
officeitems = {}

officeitems['printer'] = 'HP'

officeitems['paper'] = 'A4'

officeitems['drafting pads'] = 'blank'

officeitems['scanner'] = 'hybrid'

officeitems['table lights'] = 'LED'

print(officeitems)
```

= RESTART: C:/Users/saifia computers/Desktop/sample.py

{'printer': 'HP', 'paper': 'A4', 'drafting pads': 'blank', 'scanner': 'hybrid', 'table lights': 'LED'}

>>>

You also can modify the value of a key as you deem fit. In order to do that, you have to mention the name of the dictionary and write the key in square brackets. Then you have to write the new value for the same key.

officeitems = {'printer' : 'HP', 'paper': 'A4', 'drafting pads': 'blank', 'scanner': 'hybrid', 'table': 'wood', 'table lights': 'LED'}

print("I have purchased a printer by " + officeitems['printer'] + ".")

officeitems['printer'] = 'dell'

print("However, I have also purchased one more now by " + officeitems['printer'] + ".")

= RESTART: C:/Users/saifia computers/Desktop/sample.py

I have purchased a printer by HP.

However, I have also purchased one more now by dell.

>>>

Removing Pairs

When you don't need a certain key-value pair, you can remove it easily. You can apply the del statement to remove the key-value pair. All you have to do is to mention the name of your dictionary and key. I will remove different items from the dictionary I have created earlier on.

officeitems = {'printer' : 'HP', 'paper': 'A4', 'drafting pads': 'blank', 'scanner': 'hybrid', 'table': 'wood', 'table lights': 'LED'}

print(officeitems)

del officeitems['printer']

print(officeitems)

del officeitems['paper']

print(officeitems)

del officeitems['drafting pads']

print(officeitems)

del officeitems['scanner']

```
print(officeitems)

del officeitems['table']

print(officeitems)

del officeitems['table lights']

print(officeitems)

= RESTART: C:/Users/saifia computers/Desktop/sample.py

{'printer': 'HP', 'paper': 'A4', 'drafting pads': 'blank', 'scanner': 'hybrid', 'table': 'wood', 'table lights': 'LED'}

{'paper': 'A4', 'drafting pads': 'blank', 'scanner': 'hybrid', 'table': 'wood', 'table lights': 'LED'}

{'drafting pads': 'blank', 'scanner': 'hybrid', 'table': 'wood', 'table lights': 'LED'}

{'scanner': 'hybrid', 'table': 'wood', 'table lights': 'LED'}

{'table': 'wood', 'table lights': 'LED'}

{'table lights': 'LED'}

{}

>>>
```

You can see that when all the pairs are removed, the result is an empty dictionary. One important thing to keep in mind is that the del statement removes a pair completely from the dictionary. Therefore, only use the del statement when you are sure that you do not need a certain key-value pair.

A dictionary allows you to use the values in a print statement to display certain messages.

```
officeitems = {'printer' : 'HP', 'paper': 'A4', 'drafting pads': 'blank', 'scanner': 'hybrid', 'table': 'wood', 'table lights': 'LED'}

print("I bought a printer by " + officeitems['printer'] + ".")

print("I also bought a scanner by " + officeitems['scanner'] + ".")

print("The paper if of the size " + officeitems['paper'] + ".")

print("The drafting pads are " + officeitems['drafting pads'] + ".")
```

= RESTART: C:/Users/saifia computers/Desktop/sample.py

I bought a printer by HP.

I also bought a scanner by hybrid.

The paper if of the size A4.

The drafting pads are blank.

>>>

I have used the print keyword in the code. Then I added the appropriate statement to the code. After that came the part of the concatenation operator. This is how you can use the values of a dictionary to display messages in your code.

Looping

Just like we formed a loop through a list, we can form the same through a dictionary as well. A Python dictionary may contain a few millions of key-value pairs. As a dictionary carries big amounts of data, Python allows you to create a loop through it to easily see each key-value pair and use it in a program. In the first example, I will loop through each item in your dictionary.

```
officeitems = {'printer' : 'HP', 'paper': 'A4', 'drafting pads': 'blank', 'scanner': 'hybrid', 'table': 'wood', 'table lights': 'LED'}

for key, value in officeitems.items():
    print("\nThe Key: " + key)
    print("The Value: " + value)
```

= RESTART: C:/Users/saifia computers/Desktop/sample.py

The Key: printer

The Value: HP

The Key: paper

The Value: A4

The Key: drafting pads

The Value: blank

The Key: scanner

The Value: hybrid

The Key: table

The Value: wood

The Key: table lights

The Value: LED

>>>

There is another method to display the values of each key-value pair. See the following example.

```
officeitems = {'printer' : 'HP', 'paper': 'A4', 'drafting pads': 'blank', 'scanner': 'hybrid', 'table': 'wood', 'table lights': 'LED'}

for k, v in officeitems.items():
    print("\nThe Key: " + k)
    print("The Value: " + v)
```

= RESTART: C:/Users/saifia computers/Desktop/sample.py

The Key: printer

The Value: HP

The Key: paper

The Value: A4

The Key: drafting pads

The Value: blank

The Key: scanner

The Value: hybrid

The Key: table

The Value: wood

The Key: table lights

The Value: LED

>>>

One important thing to consider before moving on is the order in which Python stores the key-value pairs. When you create and run a loop through a dictionary, Python does not care about the order in which you had created the dictionary. It only tracks down the keys and their respective values.

officeitems = {'printer' : 'Produced by HP', 'paper': 'A4 type', 'drafting pads': 'It is blank', 'scanner': 'it is hybrid', 'table': 'made of wood', 'table lights': 'They are LED'}

for items, features in officeitems.items():

    print(items.title() + " carries the following feature: " + features.title())

= RESTART: C:/Users/saifia computers/Desktop/sample.py

Printer carries the following feature: Produced By Hp

Paper carries the following feature: A4 Type

Drafting Pads carries the following feature: It Is Blank

Scanner carries the following feature: It Is Hybrid

Table carries the following feature: Made Of Wood

Table Lights carries the following feature: They Are Led

>>>

The code instructs Python to loop through the key-value pairs inside of the dictionary. As the code loops through each pair, Python first stores each key inside the variable named items. It stores each value inside the variable named features. The same variables are then added to the print statement that runs and displays related messages.

You can opt for looping through all the keys or values separately. For example, sometimes you need to work just with the keys and only want to display them. There is a way out. See the following example.

officeitems = {'printer' : 'Produced by HP', 'paper': 'A4 type', 'drafting pads': 'It is blank', 'scanner': 'it is hybrid', 'table': 'made of wood', 'table lights': 'They are LED'}

```
for items in officeitems.keys():
    print(items.title())

= RESTART: C:/Users/saifia computers/Desktop/sample.py
Printer
Paper
Drafting Pads
Scanner
Table
Table Lights
>>>
```

Now I will form a loop through each key's values and display the result in the interpreter.

```
officeitems = {'printer' : 'Produced by HP', 'paper': 'A4 type', 'drafting pads': 'It is blank', 'scanner': 'it is hybrid', 'table': 'made of wood', 'table lights': 'They are LED'}
for items in officeitems.values():
    print(items.title())
= RESTART: C:/Users/saifia computers/Desktop/sample.py
Produced By Hp
A4 Type
It Is Blank
It Is Hybrid
Made Of Wood
They Are Led
>>>
```

The sorted() Method

To make your loops more interesting, you can add to them the sorted() method. A dictionary is not in order, therefore you need to bring it up in the order you want it to be. You can use the sorted() method to make that happen.

```
officeitems = {'printer' : 'Produced by HP', 'paper': 'A4 type', 'drafting pads': 'It is blank', 'scanner': 'it is hybrid', 'table': 'made of wood', 'table lights': 'They are LED'}

for items in sorted(officeitems.keys()):

    print(items.title() + " has been purchased at a discount price. I hope it will help earn a handsome amount from the sale of the office")
```

```
= RESTART: C:/Users/saifia computers/Desktop/sample.py

Drafting Pads has been purchased at a discount price. I hope it will help earn a handsome amount from the sale of the office

Paper has been purchased at a discount price. I hope it will help earn a handsome amount from the sale of the office

Printer has been purchased at a discount price. I hope it will help earn a handsome amount from the sale of the office

Scanner has been purchased at a discount price. I hope it will help earn a handsome amount from the sale of the office

Table has been purchased at a discount price. I hope it will help earn a handsome amount from the sale of the office

Table Lights have been purchased at a discount price. I hope it will help earn a handsome amount from the sale of the office

>>>
```

You can see that the result is in perfect alphabetical order.

Nesting

Ever wondered if you can make a dictionary more complex than it already is. You can nest a long dictionary inside another dictionary. The process is dubbed as nesting. You also can nest more than one dictionaries in a list. You can diversify the process of nesting by several methods.

I am going to pack up multiple dictionaries inside a list. Coming to the back, I will create three different dictionaries about different items of an office and

then cram them all inside a list.

```
officeitem1 = {'printer' : 'Produced by HP', 'scanner': 'it is hybrid', 'laptop': 'dell'}

officeitem2 = {'paper': 'A4 type', 'drafting pads': 'It is blank', 'pen': 'parker'}

officeitem3 = {'table': 'made of wood', 'table lights': 'They are LED', 'office chair': 'boss'}

officeitems = [officeitem1, officeitem2, officeitem3]

for officeitem in officeitems:
    print(officeitem)
```

```
= RESTART: C:/Users/saifia computers/Desktop/sample.py
{'printer': 'Produced by HP', 'scanner': 'it is hybrid', 'laptop': 'dell'}
{'paper': 'A4 type', 'drafting pads': 'It is blank', 'pen': 'parker'}
{'table': 'made of wood', 'table lights': 'They are LED', 'office chair': 'boss'}
>>>
```

The three dictionaries denote each section of the office items. One deals with IT set up, the second denotes stationary while the third section denotes office furniture.

Random Dictionary Methods

Dictionaries are flexible in the sense that they allow you to do several things. For example, you can check if a certain key exists in the dictionary or not. I will use the if statement in the code.

```
officeitem1 = {'printer' : 'Produced by HP', 'scanner': 'it is hybrid', 'laptop': 'dell', 'paper': 'A4 type', 'drafting pads': 'It is blank', 'pen': 'parker', 'table': 'made of wood', 'table lights': 'They are LED', 'office chair': 'boss'}

if "scanner" in officeitem1:
    print("Yes, I have got 'scanner' in the office.")
else:
    print("Sorry, I do not have that item.")
```

= RESTART: C:/Users/saifia computers/Desktop/sample.py

Yes, I have got 'scanner' in the office.

>>>

There is another method known as the clear() method that will empty your dictionary. See the following code sample.

officeitem1 = {'printer' : 'Produced by HP', 'scanner': 'it is hybrid', 'laptop': 'dell', 'paper': 'A4 type', 'drafting pads': 'It is blank', 'pen': 'parker', 'table': 'made of wood', 'table lights': 'They are LED', 'office chair': 'boss'}

print(officeitem1)

officeitem1.clear()

print(officeitem1)

= RESTART: C:/Users/saifia computers/Desktop/sample.py

{'printer': 'Produced by HP', 'scanner': 'it is hybrid', 'laptop': 'dell', 'paper': 'A4 type', 'drafting pads': 'It is blank', 'pen': 'parker', 'table': 'made of wood', 'table lights': 'They are LED', 'office chair': 'boss'}

{}

>>>

You can see that the clear method has emptied the dictionary. Python allows you to create perfect copies of your dictionary. You can create as many copies as you want to. The method is dubbed as the copy() method. It is a built-in Python method.

officeitem1 = {'printer' : 'Produced by HP', 'scanner': 'it is hybrid', 'laptop': 'dell', 'paper': 'A4 type', 'drafting pads': 'It is blank', 'pen': 'parker', 'table': 'made of wood', 'table lights': 'They are LED', 'office chair': 'boss'}

print(officeitem1)

officeitem2 = officeitem1.copy()

print(officeitem2)

= RESTART: C:/Users/saifia computers/Desktop/sample.py

{'printer': 'Produced by HP', 'scanner': 'it is hybrid', 'laptop': 'dell', 'paper': 'A4

type', 'drafting pads': 'It is blank', 'pen': 'parker', 'table': 'made of wood', 'table lights': 'They are LED', 'office chair': 'boss'}

{'printer': 'Produced by HP', 'scanner': 'it is hybrid', 'laptop': 'dell', 'paper': 'A4 type', 'drafting pads': 'It is blank', 'pen': 'parker', 'table': 'made of wood', 'table lights': 'They are LED', 'office chair': 'boss'}

>>>

There is another built-in method to create copy of the dictionary. The method is labeled as the dict() method.

officeitem1 = {'printer' : 'Produced by HP', 'scanner': 'it is hybrid', 'laptop': 'dell', 'paper': 'A4 type', 'drafting pads': 'It is blank', 'pen': 'parker', 'table': 'made of wood', 'table lights': 'They are LED', 'office chair': 'boss'}

print(officeitem1)

officeitem2 = dict(officeitem1)

print(officeitem2)

= RESTART: C:/Users/saifia computers/Desktop/sample.py

{'printer': 'Produced by HP', 'scanner': 'it is hybrid', 'laptop': 'dell', 'paper': 'A4 type', 'drafting pads': 'It is blank', 'pen': 'parker', 'table': 'made of wood', 'table lights': 'They are LED', 'office chair': 'boss'}

{'printer': 'Produced by HP', 'scanner': 'it is hybrid', 'laptop': 'dell', 'paper': 'A4 type', 'drafting pads': 'It is blank', 'pen': 'parker', 'table': 'made of wood', 'table lights': 'They are LED', 'office chair': 'boss'}

>>>

We had the exact copy of the same dictionary. There is a bit of difference in writing the code.

If you have to create a dictionary from scratch, you can use the dict() constructor to do that. I will take an empty dictionary and fill it in with the keys and values by using the dict() constructor.

officeitem1 = dict(printer = 'Produced by HP', scanner = 'it is hybrid', laptop = 'dell', paper = 'A4 type', draftingpads = 'blank', pen = 'parker', table = 'made of wood')

print(officeitem1)

= RESTART: C:/Users/saifia computers/Desktop/sample.py

{'printer': 'Produced by HP', 'scanner': 'it is hybrid', 'laptop': 'dell', 'paper': 'A4 type', 'draftingpads': 'blank', 'pen': 'parker', 'table': 'made of wood'}

>>>

The keys should not contain any spaces while you are constructing a dictionary by the dict() constructor. Please take a look at how I wrote drafting pads. If you leave any spaces between the keys, Python interpreter will return syntax error.

Chapter Six: Input and Python Loops

Programs are written to solve different problems. Some programs are made to collect information from users. These programs demand special functions that could collect the information and process it to the system's database. When you put your office on sale, you can introduce a special function that invites the buyers' quotations. The user input program will take the input, analyze it, and respond to the user.

In this chapter, I'll explain how you can build a program that accepts user input and processes it. I will use the input() function to develop the program. The user input and while loop will be explained together as it is the while loop that keeps the program running. The while loop runs the program as long as a particular condition stands true.

The input() Function

It is an interesting function and very helpful in program building. The function pauses your program and allows the user to fill in the program with the requisite information. Once the function receives the information, it forwards it to a variable for storage purposes.

```
pgm = input("This program repeats whatever you write: ")

print(pgm)

= RESTART: C:/Users/saifia computers/Desktop/sample.py

This program repeats whatever you write: I am learning Python and I am enjoying it well.

I am learning Python and I am enjoying it well.

>>>

= RESTART: C:/Users/saifia computers/Desktop/sample.py

This program repeats whatever you write: Do you know Python can be used to educate robots.

Do you know Python can be used to educate robots.

>>>
```

I entered some statements which the program repeats as they are. The important point is that you have to rerun the program once it has repeated one statement. When you run the program, it pauses and waits for the user to write something. Once the program senses input, it waits for the user to press Enter. After that, it displays the results. I will create a program that asks users to enter the bidding price to buy the office that has already been set up by the player in your game.

```
pgm = input("Please enter the bidding price at which you want to buy the office: ")

print("I want to buy the office at " + pgm + " million dollars.")
```

```
= RESTART: C:/Users/saifia computers/Desktop/sample.py

Please enter the bidding price at which you want to buy the office: five

I want to buy the office for five million dollars.

>>>
```

The program is suitable only if the user enters the value in the form of string. Therefore you will have to leave a note, instructing the user to write only in words. However, there is a way out to solve this problem. You can allow users to enter the price in numbers without causing an error.

```
pgm = input("Please enter the bidding price at which you want to buy the office: ")

print("I want to buy the office at " + str(pgm) + " million dollars.")
```

```
= RESTART: C:/Users/saifia computers/Desktop/sample.py

Please enter the bidding price at which you want to buy the office: 5

I want to buy the office for 5 million dollars.

>>>
```

In the next sample, I will create a program that has more than lines.

```
pgm = input("If you are interested in buying the office, please proceed to fill in the price box. ")

pgm += "\nPlease enter the bidding price at which you want to buy the office. "
```

```
username = input(pgm)
print("I want to buy the office for " + username + " million dollars.")
= RESTART: C:/Users/saifia computers/Desktop/sample.py
If you are interested in buying the office, please proceed to fill in the price box.
Please enter the bidding price at which you want to buy the office. 3
I want to buy the office for 3 million dollars.
>>>
```

This is how you can easily build a multiline string in the user input function.

While Loops

This section will shed light on how you can create and use Python while loops. You have already encountered the for loop which runs through a list of items and applies the code to each item in the list. The while loop is a bit different. It runs through a set of items as long as a certain condition stands true. While loop is interesting in the sense that you can use it to execute different interesting mathematical functions. The simplest and the most interesting thing is counting the numbers.

```
my_number = 1
while my_number <= 15:
    print(my_number)
    my_number += 1
= RESTART: C:/Users/saifia computers/Desktop/sample.py
1
2
3
4
5
6
```

7
8
9
10
11
12
13
14
15
>>>
See another mathematical example of the use of a while loop.

```
my_number = 1
while my_number <= 100:
    print(my_number)
    my_number += 5
```

= RESTART: C:/Users/saifia computers/Desktop/sample.py
1
6
11
16
21
26
31
36
41
46

```
51
56
61
66
71
76
81
86
91
96
>>>
```

I set the value to numbers 1 and 5, respectively. The while loop reads it and keeps running until it reaches 15 and 100, respectively. The code guides the loop to calculate the numbers and display the result on the interpreter. The loops get repeated as long as its condition remains true. Your player needs a while loop to exit the game. Only a while loop helps you end a game and shutdown it properly. Otherwise, it will hang the system each time you try to shut it down.

This demands that you let the users quit the game when they want to. I will not pack up the program in a while loop and then define the quit value for the same so that users can exit it by entering the quit value.

```
pgm = input("This program repeats whatever you tell it: ")
pgm += "\nYou have to enter 'q' to exit the program. "
msg = ""
while msg != 'q':
    msg = input(pgm)
    print(msg)

= RESTART: C:/Users/saifia computers/Desktop/sample.py
```

This program repeats whatever you tell it: I am determined to learn Python in six months.

I am determined to learn Python in six months.

You have to enter 'q' to exit the program. I am determined to build my programs in the first month of learning.

I am determined to build my programs in the first month of learning.

I am determined to learn Python in six months.

You have to enter 'q' to exit the program. q

q

>>>

I defined the prompt namely pgm in the first line of code. It gives the user two options; one to enter a message and another to quit the program. I also set a variable that stored the information the user enters. The while loop runs until the user enters q and breaks the loop. It can run a million times on end if the user does not end it.

```
pgm = input("This program repeats whatever you tell it: ")
pgm += "\nYou have to enter 'q' to exit the program. "
msg = ""
while msg != 'q':
    msg = input(pgm)
    print(msg)
```

= RESTART: C:/Users/saifia computers/Desktop/sample.py

This program repeats whatever you tell it: hi

hi

You have to enter 'q' to exit the program. how are you

how are you

hi

You have to enter 'q' to exit the program. What is your name?

What is your name?

hi

You have to enter 'q' to exit the program. Are you fine?

Are you fine?

hi

You have to enter 'q' to exit the program. I am looking forward to doing business with you.

I am looking forward to doing business with you.

hi

You have to enter 'q' to exit the program. q

q

>>>

You can see that the while loop ran until I entered the keyword q that broke the loop. The program is perfect except for the fact that it displays q as an actual message. If I add an *if* clause to the code, it will work just fine.

```
pgm = input("This program repeats whatever you tell it: ")
pgm += "\nYou have to enter 'q' to exit the program. "
msg = ""
while msg != 'q':
    msg = input(pgm)
    if msg != 'q':
        print(msg)
```

= RESTART: C:/Users/saifia computers/Desktop/sample.py

This program repeats whatever you tell it: Hi

Hi

You have to enter 'q' to exit the program. My name is Jack.

My name is Jack.

Hi

You have to enter 'q' to exit the program. I am here to do business with you.

I am here to do business with you.

Hi

You have to enter 'q' to exit the program. I want to sell an office to the highest bidder. Have a look at the pictures.

I want to sell an office to the highest bidder. Have a look at the pictures.

Hi

You have to enter 'q' to exit the program. I think you are not interested. Thank you!

I think you are not interested. Thank you!

Hi

You have to enter 'q' to exit the program. q

>>>

The program did not display the word q as a message. It simply lets the user exit the program.

The Break Keyword

If you want to exit the loop without running the code that remains, you can add a break statement to the program. The break statement tends to redirect the flow of a program and allow you to execute the code of your choice.

```
pgm = input("Please enter the name of the office item that you have purchased:")
pgm += "\n(You have to enter 'q' to exit the program.) "
while True:
    item = input(pgm)
    if item == 'q':
```

```
        break
    else:
        print("I have purchased the " + item.title())
= RESTART: C:/Users/saifia computers/Desktop/sample.py
Please enter the name of the office item that you have purchased:table
table
(You have to enter 'q' to exit the program.)
I have purchased the
table
(You have to enter 'q' to exit the program.) laptop
I have purchased the Laptop
table
(You have to enter 'q' to exit the program.) computer system
I have purchased the Computer System
table
(You have to enter 'q' to exit the program.) stack of paper.
I have purchased the Stack Of Paper.
table
(You have to enter 'q' to exit the program.) air conditioner
I have purchased the Air Conditioner
table
(You have to enter 'q' to exit the program.) q
>>>
```

You have another choice as well. Instead of breaking out of the loop, you can integrate into the block of code a continue statement that will take the code back to the start after the condition stands tested. See the following mathematical example.

```
num = 0
while num < 40:
    num += 1
    if num %4 == 0:
        continue
    print(num)
= RESTART: C:/Users/saifia computers/Desktop/sample.py
1
2
3
5
6
7
9
10
11
13
14
15
17
18
19
21
22
23
25
```

```
26
27
29
30
31
33
34
35
37
38
39
>>>
```

You can see that the continue statement returned the code after a pause at the point Python tested the condition. The num started at 0. I kept the figure under 40 so the loop ran until 4, checked if the current number is divisible by 4 and then executed the rest of the code because the number was not divisible by 4. Let us try another example to clear the concept fully.

```
num = 4
while num < 80:
    num += 3
    if num %4 == 0:
        continue
    print(num)
= RESTART: C:/Users/saifia computers/Desktop/sample.py
7
10
13
```

19

22

25

31

34

37

43

46

49

55

58

61

67

70

73

79

82

>>>

Loops, Lists, Dictionaries

The three go side by side. I have already given some code snippets that showed what a while loop could be used for. I will give a comprehensive overview of how you can pair up loops, lists, and dictionaries. The examples will be a bit more complex than the previous examples.

items_tobuy = ['printer', 'scanner', 'laptop', 'paper', 'drafting pads', 'pen', 'table', 'table lights', 'office chair']

items_bought = []

while items_tobuy:

```
    officeitems = items_tobuy.pop()
    print("I am purchasing the " + officeitems.title())
    items_bought.append(officeitems)
    print("\nI have purchased the following items:")
    for item_bought in items_bought:
        print(item_bought.title())
= RESTART: C:/Users/saifia computers/Desktop/sample.py
I am purchasing the Office Chair
I have purchased the following items:
Office Chair
I am purchasing the Table Lights
I have purchased the following items:
Office Chair
Table Lights
I am purchasing the Table
I have purchased the following items:
Office Chair
Table Lights
Table
I am purchasing the Pen
I have purchased the following items:
Office Chair
Table Lights
Table
Pen
I am purchasing the Drafting Pads
```

I have purchased the following items:

Office Chair

Table Lights

Table

Pen

Drafting Pads

I am purchasing the Paper

I have purchased the following items:

Office Chair

Table Lights

Table

Pen

Drafting Pads

Paper

I am purchasing the Laptop

I have purchased the following items:

Office Chair

Table Lights

Table

Pen

Drafting Pads

Paper

Laptop

I am purchasing the Scanner

I have purchased the following items:

Office Chair

Table Lights

Table

Pen

Drafting Pads

Paper

Laptop

Scanner

I am purchasing the Printer

I have purchased the following items:

Office Chair

Table Lights

Table

Pen

Drafting Pads

Paper

Laptop

Scanner

Printer

>>>

When a player purchases an item, he will get a clear message that an item has been bought and added to the office. So, this has definitely made the game more interesting.

While loop can also help you in removing multiple instances of a particular value. If an item's value is being repeated in the list, you can set up a while loop to remove all instances of the same. For a small list, manually removing it is not a problem. However, this feature of the while loops becomes a must when you are dealing with long lists.

officeitems = ['printer', 'scanner', 'laptop', 'paper', 'drafting pads', 'scanner',

'pen', 'table', 'scanner', 'table lights', 'office chair']

```
print(officeitems)
while 'scanner' in officeitems:
    officeitems.remove('scanner')
print(officeitems)
```

= RESTART: C:/Users/saifia computers/Desktop/sample.py

['printer', 'scanner', 'laptop', 'paper', 'drafting pads', 'scanner', 'pen', 'table', 'scanner', 'table lights', 'office chair']

['printer', 'laptop', 'paper', 'drafting pads', 'pen', 'table', 'table lights', 'office chair']

>>>

In the next example, I will build a dictionary with the user input with a while loop. I will create a program that will ask the user to tell about the office items he wants to buy and the brand name. The input will be forwarded to a dictionary and used to create the desired output for the user. The program will display purchase statistics in a neat and summarized way.

```
officeitems = {}
buying = True
while buying:
    buyer_item = input("\nWhat do want to buy? ")
    officeitem = input("Of what brand do you want to buy? ")
    officeitems[buyer_item] = officeitem
    repeat = input("Would you like to buy another item for the office? (yes/ no ")
    if repeat == 'no':
        buying = False
print("\nPurchase Statistics")
for buyer_item, officeitem in officeitems.items():
```

```
    print("I have purchased " + buyer_item + " from the brand " + officeitem
+ ".")
= RESTART: C:/Users/saifia computers/Desktop/sample.py
What do you want to buy? table
Of what brand do you want to buy? interwood
Would you like to buy another item for the office? (yes/ noyes
What do you want to buy?
= RESTART: C:/Users/saifia computers/Desktop/sample.py
What do you want to buy? table
Of what brand do you want to buy? interwood
Would you like to buy another item for the office? (yes/ no yes
What do you want to buy? air conditioner
Of what brand do you want to buy? orient
Would you like to buy another item for the office? (yes/ no yes
What do you want to buy? laptop
Of what brand do you want to buy? dell
Would you like to buy another item for the office? (yes/ no yes
What do you want to buy? office chair
Of what brand do you want to buy? boss
Would you like to buy another item for the office? (yes/ no no
Purchase Statistics
I have purchased a table from the brand interwood.
I have purchased an air conditioner from the brand orient.
I have purchased a laptop from the brand dell.
I have purchased an office chair from the brand boss.
>>>
```

The same program can be redesigned to collect names and email IDs of users who visit your eCommerce website. You can then use the information to send your prospects direct mail ads and boost your business. All you need is a bit of tweaking to the existing code and your program will be ready to boost your marketing campaign.

```
users_id = {}
info = True
while info:
    user_name = input("\nWhat is your name? ")
    user_id = input("what is your email id? ")
    users_id[user_name] = user_id
    repeat = input("Would you like to add another username or id? (yes/ no ")
    if repeat == 'no':
        info = False
print("\nUser Info")
for user_name, user_id in users_id.items():
    print("My name is " + user_name + " and my email ID is " + user_id + ".")
= RESTART: C:/Users/saifia computers/Desktop/sample.py
What is your name? johnson
what is your email id? johnson@gmail.com
Would you like to add another username or id? (yes/ no yes
What is your name? emily
what is your email id? emily@yahoo.com
Would you like to add another username or id? (yes/ no yes
What is your name? emilia
what is your email id? emilia@outlook.com
Would you like to add another username or id? (yes/ no yes
```

What is your name? mark

what is your email id? mark@rocketmail.com

Would you like to add another username or id? (yes/ no yes

What is your name? jasmine

what is your email id? jasmine@gmail.com

Would you like to add another username or id? (yes/ no

User Info

My name is johnson and my email ID is johnson@gmail.com.

My name is emily and my email ID is emily@yahoo.com.

My name is emilia and my email ID is emilia@outlook.com.

My name is mark and my email ID is mark@rocketmail.com.

My name is jasmine and my email ID is jasmine@gmail.com.

>>>

You can see how easy it is to build a program with a while loop and user input function to collect crucial prospect data that can ultimately help you shape your marketing campaign. You can give this program a brilliant interface and run it as part of your landing page design. As I have defined an empty dictionary at the start, you can fill it with as much information as you want to. It can carry over a million items.

Chapter Seven: Python Functions

This chapter will walk you through the process of writing functions. Functions can be defined as blocks of code that have just one job to perform. When you want to do a simple task that you have defined in your function, you can just call the function you have written to do the job. If you are looking forward to performing the same task more than once throughout the program, you can just make a call to the same function and Python will execute the entire block of code. Functions make your programs simple and easy to write and run.

In this chapter, I will explain how you can create functions, pass crucial information to the same, and repeat multiple times the task that functions perform. I will also explain how you can store a function in the form of modules.

Defining Functions

Defining a function is a simple job. The keyword I will use is called *def*. The keyword will be followed by the name of the function and parenthesis. Parenthesis is a function that is very important as it can be used for different purposes, like adding default information and passing information to functions at a later stage of writing a program.

```
def user_info():
    print("My name is Joe and I am a new user.")
user_info()
```

```
= RESTART: C:/Users/saifia computers/Desktop/sample.py
My name is Joe and I am a new user.
>>>
```

In the first line of code, I have defined the function. The second line carries the usual print statement while the last line is where I made a function call to display the function's information. This can be dubbed as the simplest structure of a function. The def keyword defines the function.

From this point, I will make it a bit complex by passing information to the function. I will have to modify it a little bit to suit our needs. The parenthesis

now will no longer be empty. I will fill them up with some information.

```
def user_info(username):
    print("My name is " + username.title() + " and I am a new user.")
user_info('Joe')
user_info('Jimmy')
user_info('Emily')
user_info('Emilia')
user_info('Mark')
= RESTART: C:/Users/saifia computers/Desktop/sample.py
My name is Joe and I am a new user.
My name is Jimmy and I am a new user.
My name is Emily and I am a new user.
My name is Emilia and I am a new user.
My name is Mark and I am a new user.
>>>
```

The most important point to note here is that functions help you cut short the block of code. You do not have to rewrite a block of code again and again to do the same job. All you need is to call the function and use the parenthesis to use new information for the same block of code. Programmers love functions because they save their time and energy when they write lengthy programs.

Arguments and Parameters

I have packed up the variable inside the parenthesis and named it as the username; this is labeled as a parameter. The values Joe and Emily that I have put in the parenthesis function are known as arguments. Arguments and parentheses are often confused with each other. People use them interchangeably. That is not the right thing to do.

There can be multiple arguments for a function; you can pass them to the function in many ways and put them in a position. They are called positional arguments and they are also known as keyword arguments. Each argument

may include the name of a variable, a list, or a dictionary.

Positional Arguments

When you make a function call, Python ought to watch the arguments with a specific parameter in the definition of a function. The matching of values are dubbed as positional arguments.

```
def user_info(username, email_id):
    print("My name is " + username.title() + " and I am a new user.")
    print("My email ID is " + email_id.title() + ".")
user_info('Joe', 'joe@gmail.com')
user_info('Jimmy', 'jimmy@outlook.com')
user_info('Emily', 'emily@gmail.com')
user_info('Emilia', 'emilia@yahoo.com')
user_info('Mark', 'mark@outlook.com')
= RESTART: C:/Users/saifia computers/Desktop/sample.py
My name is Joe and I am a new user.
My email ID is Joe@Gmail.Com.
My name is Jimmy and I am a new user.
My email ID is Jimmy@Outlook.Com.
My name is Emily and I am a new user.
My email ID is Emily@Gmail.Com.
My name is Emilia and I am a new user.
My email ID is Emilia@Yahoo.Com.
My name is Mark and I am a new user.
My email ID is Mark@Outlook.Com.
>>>
```

The output neatly displays the name of the user and his or her email ID. In the above code sample, I have called the function more than once. Multiple

function calls are the most efficient way to do a job. As soon as a new user fills in the information and you make a function call, the entire block of code will run and execute the information. There is virtually no limit to the number of function calls. One important thing to keep in mind while making a function call is to remember the position of arguments. If you change the position, you are likely to get funny results.

```
def user_info(username, email_id):
    print("My name is " + username.title() + " and I am a new user.")
    print("My email ID is " + email_id.title() + ".")
user_info('joe@gmail.com', 'Joe')
user_info('jimmy@outlook.com', 'Jimmy' )
user_info('emily@gmail.com', 'Emily')
user_info('Emilia', 'emilia@yahoo.com')
user_info('Mark', 'mark@outlook.com')
>>>= RESTART: C:/Users/saifia computers/Desktop/sample.py
My name is Joe@Gmail.Com and I am a new user.
My email ID is Joe.
My name is Jimmy@Outlook.Com and I am a new user.
My email ID is Jimmy.
My name is Emily@Gmail.Com and I am a new user.
My email ID is Emily.
My name is Emilia and I am a new user.
My email ID is Emilia@Yahoo.Com.
My name is Mark and I am a new user.
My email ID is Mark@Outlook.Com.
>>>
```

I have changed the position for the first three function calls and the results are ridiculous.

Keyword Arguments

There is another way out. You can use keyword arguments to avoid this kind of mix up. A keyword argument is like a name-value pair that is passed to a function. A keyword argument allows you to create a link between the name and the value inside an argument. When you pass the argument to the function, Python cannot mistake it. It eliminates the confusion and you do not have to worry about bringing your arguments in order.

```
def user_info(username, email_id):
    print("My name is " + username.title() + " and I am a new user.")
    print("My email ID is " + email_id.title() + ".")
user_info(email_id = 'joe@gmail.com', username = 'Joe')
user_info( email_id = 'jimmy@outlook.com', username = 'Jimmy' )
user_info( email_id = 'emily@gmail.com', username = 'Emily')
user_info(username = 'Emilia', email_id = 'emilia@yahoo.com')
user_info(username = 'Mark', email_id = 'mark@outlook.com')
= RESTART: C:/Users/saifia computers/Desktop/sample.py
My name is Joe and I am a new user.
My email ID is Joe@Gmail.Com.
My name is Jimmy and I am a new user.
My email ID is Jimmy@Outlook.Com.
My name is Emily and I am a new user.
My email ID is Emily@Gmail.Com.
My name is Emilia and I am a new user.
My email ID is Emilia@Yahoo.Com.
My name is Mark and I am a new user.
My email ID is Mark@Outlook.Com.
>>>
```

I have changed the positions of the arguments and it hardly affected the

results. Keyword arguments help you create a functional program.

Default Values

When you are writing a program, you may come up with information that you have to use repeatedly. This means that you will have to fill in the function call with the required arguments each you need that information to be executed. This may result in a waste of time and energy, and may also cause frustration. If you create default values for the function, you will be able to execute the excessively used information fast and efficiently. When you leave the function call empty, it will use the default arguments. You can use the default information as many times as you want to. The default values tend to simply a program and declutter the code. I will fill in the same example with the default arguments and also use the default values multiple times.

```
def user_info(username = 'Dora', email_id = 'dora@outlook.com'):
    print("My name is " + username.title() + " and I am a new user.")
    print("My email ID is " + email_id.title() + ".")
user_info(email_id = 'joe@gmail.com', username = 'Joe')
user_info()
user_info( email_id = 'jimmy@outlook.com', username = 'Jimmy' )
user_info()
user_info( email_id = 'emily@gmail.com', username = 'Emily')
user_info(username = 'Emilia', email_id = 'emilia@yahoo.com')
user_info(username = 'Mark', email_id = 'mark@outlook.com')
user_info()
= RESTART: C:/Users/saifia computers/Desktop/sample.py
My name is Joe and I am a new user.
My email ID is Joe@Gmail.Com.
My name is Dora and I am a new user.
My email ID is Dora@Outlook.Com.
```

```
My name is Jimmy and I am a new user.
My email ID is Jimmy@Outlook.Com.
My name is Dora and I am a new user.
My email ID is Dora@Outlook.Com.
My name is Emily and I am a new user.
My email ID is Emily@Gmail.Com.
My name is Emilia and I am a new user.
My email ID is Emilia@Yahoo.Com.
My name is Mark and I am a new user.
My email ID is Mark@Outlook.Com.
My name is Dora and I am a new user.
My email ID is Dora@Outlook.Com.
>>>
```

You can use the keyword arguments, the positional arguments and the default values at the same time.

```
def user_info(username = 'Dora', email_id = 'dora@outlook.com'):
    print("My name is " + username.title() + " and I am a new user.")
    print("My email ID is " + email_id.title() + ".")
user_info(email_id = 'joe@gmail.com', username = 'Joe')
user_info()
user_info( username = 'Jimmy' )
user_info()
user_info( email_id = 'emily@gmail.com', username = 'Emily')
user_info('Emilia', email_id = 'emilia@yahoo.com')
user_info('Mark', 'mark@outlook.com')
user_info()
```

= RESTART: C:/Users/saifia computers/Desktop/sample.py

My name is Joe and I am a new user.

My email ID is Joe@Gmail.Com.

My name is Dora and I am a new user.

My email ID is Dora@Outlook.Com.

My name is Jimmy and I am a new user.

My email ID is Dora@Outlook.Com.

My name is Dora and I am a new user.

My email ID is Dora@Outlook.Com.

My name is Emily and I am a new user.

My email ID is Emily@Gmail.Com.

My name is Emilia and I am a new user.

My email ID is Emilia@Yahoo.Com.

My name is Mark and I am a new user.

My email ID is Mark@Outlook.Com.

My name is Dora and I am a new user.

My email ID is Dora@Outlook.Com.

>>>

The most important thing to note in the code mentioned above is that in one function call when I missed out on writing the email ID, the program picked it up from the default values and ran it. If you leave one argument in the function call but have a default argument in place, you will have it covered by the default values.

You may run an error if you fail to fill in the function call with the arguments.

```
def user_info(username, email_id ):
    print("My name is " + username.title() + " and I am a new user.")
    print("My email ID is " + email_id.title() + ".")
```

```
user_info()
= RESTART: C:/Users/saifia computers/Desktop/sample.py
Traceback (most recent call last):
  File "C:/Users/saifia computers/Desktop/sample.py", line 6, in <module>
    user_info()
TypeError: user_info() missing 2 required positional arguments: 'username' and 'email_id'
>>>
```

Returning Values

A function does not have to display the output in a direct form. You can make the function process a bunch of data and return the value in an indirect form. The return statement picks up a value from the function and forwards it to the line that made a function call.

```
def user_info(username, email_id ):
    info = "My name is " + username.title() + " and I am a new user, and my email ID is " + email_id.title() + "."
    return info.title()
newuser = user_info('Dora', 'dora@gmail.com')
print(newuser)
newuser1 = user_info('John', 'john@gmail.com')
print(newuser1)
newuser2 = user_info('Jimmy', 'jimmy@gmail.com')
print(newuser2)
= RESTART: C:/Users/saifia computers/Desktop/sample.py
My Name Is Dora And I Am A New User, And My Email Id Is Dora@Gmail.Com.
My Name Is John And I Am A New User, And My Email Id Is John@Gmail.Com.
```

My Name Is Jimmy And I Am A New User, And My Email Id Is Jimmy@Gmail.Com.

>>>

In the next code sample, I will add another argument to the code. I will also experiment on making an argument optional so that the users who do not want to fill in a value, can leave it without running an error in the program.

```
def user_info(username, email_id, gender ):
    info = "My name is " + username.title() + " and I am a new user, and my email ID is " + email_id.title() + ". My gender is " + gender.title() + "."
    return info.title()
newuser = user_info('Dora', 'dora@gmail.com', 'female')
print(newuser)
newuser1 = user_info('John', 'john@gmail.com', 'male')
print(newuser1)
newuser2 = user_info('Jimmy', 'jimmy@gmail.com', 'male')
print(newuser2)
```

= RESTART: C:/Users/saifia computers/Desktop/sample.py

My Name Is Dora And I Am A New User, And My Email Id Is Dora@Gmail.Com. My Gender Is Female.

My Name Is John And I Am A New User, And My Email Id Is John@Gmail.Com. My Gender Is Male.

My Name Is Jimmy And I Am A New User, And My Email Id Is Jimmy@Gmail.Com. My Gender Is Male.

>>>

Suppose someone wants to leave the email option aside. You can add a conditional statement to the existing code to allow users to make a choice at will.

```
def user_info(username, gender, email_id=" ):
    if email_id:
```

```
        info = "My name is " + username.title() + " and I am a new user, and my email ID is " + email_id.title() + ". My gender is " + gender.title() + "."
    else:
        info = "My name is " + username.title() + " and my gender is " + gender.title() + "."
    return info.title()
newuser = user_info('Dora', 'female')
print(newuser)
newuser1 = user_info('John', 'john@gmail.com', 'male')
print(newuser1)
newuser2 = user_info('Jimmy', 'jimmy@gmail.com', 'male')
print(newuser2)
= RESTART: C:/Users/saifia computers/Desktop/sample.py
My Name Is Dora And My Gender Is Female.
My Name Is John And I Am A New User, And My Email Id Is Male. My Gender Is John@Gmail.Com.
My Name Is Jimmy And I Am A New User, And My Email Id Is Male. My Gender Is Jimmy@Gmail.Com.
>>>
```

Function and Dictionary

You can pair up a dictionary with a function. Take the example of the following function.

```
def user_info(username, gender, email_id):
    user = {'uname': username, 'gender': gender, 'email address': email_id}
    return user
newuser = user_info('Johnson', 'johnson@gmail.com', 'male')
print(newuser)
newuser1 = user_info('John', 'john@gmail.com', 'male')
```

```
print(newuser1)
newuser2 = user_info('Jimmy', 'jimmy@gmail.com', 'male')
print(newuser2)
newuser3 = user_info('Dora', 'dora@gmail.com', 'female')
print(newuser3)
= RESTART: C:/Users/saifia computers/Desktop/sample.py
{'uname': 'Johnson', 'gender': 'johnson@gmail.com', 'email address': 'male'}
{'uname': 'John', 'gender': 'john@gmail.com', 'email address': 'male'}
{'uname': 'Jimmy', 'gender': 'jimmy@gmail.com', 'email address': 'male'}
{'uname': 'Dora', 'gender': 'dora@gmail.com', 'email address': 'female'}
>>>
```

The function user_info takes the requisite information about the name, gender and email address of a user, and fill them up into a dictionary. Each value is stored in the designated key. The function receives information in raw form and turns it into textual information in a meaningful data structure. Up till now, I have stored and processed the information in the form of strings. You may confront situations where you have to store data in numerical form as well. There is an easy way out.

```
def user_info(username, gender, email_id, age=''):
    user = {'uname': username, 'gender': gender, 'email address': email_id}
    if age:
        user['age'] = age
    return user
newuser = user_info('Johnson', 'johnson@gmail.com', 'male', age=55)
print(newuser)
newuser1 = user_info('John', 'john@gmail.com', 'male', age=33)
print(newuser1)
newuser2 = user_info('Jimmy', 'jimmy@gmail.com', 'male', age= 54)
```

```
print(newuser2)
newuser3 = user_info('Dora', 'dora@gmail.com', 'female', age= 24)
print(newuser3)
= RESTART: C:/Users/saifia computers/Desktop/sample.py
{'uname': 'Johnson', 'gender': 'johnson@gmail.com', 'email address': 'male', 'age': 55}
{'uname': 'John', 'gender': 'john@gmail.com', 'email address': 'male', 'age': 33}
{'uname': 'Jimmy', 'gender': 'jimmy@gmail.com', 'email address': 'male', 'age': 54}
{'uname': 'Dora', 'gender': 'dora@gmail.com', 'email address': 'female', 'age': 24}
>>>
```

I have added a new parameter to the function's definition and have also assigned this parameter a kind of empty default value.

Function and While Loop

You can pair up a function with a while loop. Let us jump to the text editor to see how you can do that.

```
def user_info(username, email_id, gender ):
    info = "My name is " + username.title() + " and I am a new user, and my email ID is " + email_id.title() + ". My gender is " + gender.title() + "."
    return info.title()
while True:
    print("\nPlease tell me about yourself.")
    user_name = input("Please enter your name: ")
    email = input("Please enter your email address: ")
    gen = input("Please enter your gender: ")

newuser = user_info(user_name, email, gen)
```

print("\n The user information is as follows: " + newuser + ".")

= RESTART: C:/Users/saifia computers/Desktop/sample.py

Please tell me about yourself.

Please enter your name: John

Please enter your email address: john@gmail.com

Please enter your gender: male

Please tell me about yourself.

Please enter your name: jimmy

Please enter your email address: jimmy@gmail.com

Please enter your gender: male

Please tell me about yourself.

Please enter your name: dora

Please enter your email address: dora@gmail.com

Please enter your gender: female

Please tell me about yourself.

Please enter your name:

The while loop lacks a quit condition therefore it will run on end and will keep asking about the name of users even after all the users have filled in their personal information. I will add a break statement in the same code so that users can exit the program when they have entered all the information.

```
def user_info(username, email_id, gender ):
    info = "My name is " + username.title() + " and I am a new user, and my email ID is " + email_id.title() + ". My gender is " + gender.title() + "."
    return info.title()
while True:
    print("\nPlease tell me about yourself.")
    print("(If you want to exit the program, enter 'q')")
```

```
    uname = input("Please enter your name: ")
    if uname == 'q':
        break

    uemail = input("Please enter your email address: ")
    if uemail == 'q':
        break
    ugen = input("Please enter your gender: ")
    if ugen == 'q':
        break

newuser = user_info(uname, uemail, ugen)
print("\n The user information is as follows: " + newuser + ".")
```

This program will keep running until someone enters ‘q.’ Functions are flexible in inviting and using different types of data structures. You can easily pass a list to a specific function. Whether the list is of numbers, names, and even complex objects like dictionaries. When you do that, the function gets access to the specific contents of the list.

```
def user_info(usersinfo):
    for userinfo in usersinfo:
        mg = "Hi, My name is " + userinfo.title() + ". I am here to take a walk-in interview."
        print(mg)
candidate_names = ['jimmy', 'john', 'dora', 'johnson', 'james']
user_info(candidate_names)
= RESTART: C:/Users/saifia computers/Desktop/sample.py
Hi, My name is Jimmy. I am here to take a walk-in-interview.
Hi, My name is John. I am here to take a walk-in-interview.
Hi, My name is Dora. I am here to take a walk-in-interview.
```

Hi, My name is Johnson. I am here to take a walk-in-interview.

Hi, My name is James. I am here to take a walk-in-interview.

>>>

If you run into an error, it can possibly be due to a missing argument. See the following error type.

```
def user_info(usersinfo):
    for userinfo in usersinfo:
        mg = "Hi, My name is " + userinfo.title() + ". I am here to take a walk-in interview."
        print(mg)
candidate_names = ['jimmy', 'john', 'dora', 'johnson', 'james']
user_info()
```

= RESTART: C:/Users/saifia computers/Desktop/sample.py

```
Traceback (most recent call last):
  File "C:/Users/saifia computers/Desktop/sample.py", line 7, in <module>
    user_info()
TypeError: user_info() missing 1 required positional argument: 'usersinfo'
```

>>>

Therefore, you should not leave the parenthesis of the function empty.

Functions allow you to modify different data types such as lists. You can first pass a list and then modify it as well. The changes you introduce to a list are permanent and allow a person to work efficiently. The following will pass the list without functions.

```
to_buy_items = ['printer', 'scanner', 'fan', 'table', 'chair', 'computer system', 'table lights']
bought_items = []
while to_buy_items:
    office = to_buy_items.pop()
```

```
    print("I am buying the " + office)
    bought_items.append(office)
print("\nI have purchased and set up the following items in the office:")
for bought_item in bought_items:
    print(bought_item)
= RESTART: C:/Users/saifia computers/Desktop/sample.py
I am buying the table lights
I am buying the computer system
I am buying the chair
I am buying the table
I am buying the fan
I am buying the scanner
I am buying the printer
I have purchased and set up the following items in the office:
table lights
computer system
chair
table
fan
scanner
printer
>>>
```

Now I will write two functions for two separate jobs. The code will be more efficient and interactive.

```
def officeitems (to_buy_items, bought_items):
    while to_buy_items:
```

```
        office = to_buy_items.pop()
        print("I am buying the " + office)
        bought_items.append(office)
def o_bought_items(bought_items):
    print("\nI have purchased and set up the following items in the office:")
    for bought_item in bought_items:
        print(bought_item)
to_buy_items = ['printer', 'scanner', 'fan', 'table', 'chair', 'computer system', 'table lights']
bought_items = []
officeitems(to_buy_items, bought_items)
o_bought_items(bought_items)
= RESTART: C:/Users/saifia computers/Desktop/sample.py
I am buying the table lights
I am buying the computer system
I am buying the chair
I am buying the table
I am buying the fan
I am buying the scanner
I am buying the printer
I have purchased and set up the following items in the office:
table lights
computer system
chair
table
fan
```

scanner

printer

>>>

Chapter Eight: Object-Oriented Programming

Object-oriented programming is the spirit of Python. It is one of the most effective approaches to develop software. Object-oriented programming suggests that you write effective classes to represent real-world situations and objects. While writing a class, you get the actual feel of automation. You get to build an object from a class and add appropriate personality traits to the same. The process of building objects from a class is dubbed as instantiation.

In this chapter, I will explain how to write Python classes and how to create a lot of instances in a single class. I will also define the actions that I want to attribute to an object. You will also be able to store the classes in the form of modules and then import them to your program files.

Python classes help you build complex programs and give you a feel for programming. You will get to know your code and the bigger concepts behind these codes. Classes can help you wrap up a lot of work in a short amount of time and meet complex challenges in the simplest ways. A class can turn a random program into sophisticated software.

You can model any real-world object with the help of Python classes. In the next code snippet, I will write a code that will be modeled on a leopard. I will give the leopard a name, age, and color. I will add behavioral attributes to the class as well.

Leopard Class

After writing the leopard class, I will add instances to the same that will store the name, age and color of the object.

```
class Leopard():
    """This class will build the model of a leopard."""
    def__init__(self, lname, lage, lcolor):
        """here I will initialize the name, age and color attributes of the class."""
        self.lname = lname
        self.lage = lage
```

```
        self.lcolor = lcolor
    def run(self):
        print(self.lname.title() + " is running fast out in the wild.")
    def attack(self):
        print(self.lname.title() + " is now attacking a deer who is grazing in the meadow.")
```

This is how you can write a class and add attributes to it. I have created the class and add a couple of functions.

Explaining the __init__() Method

It is a special method that is automatically run by Python when you create a new instance from the main Leopard class. The method two underscores in the front and two in the trail.

I have defined the __init__() method and given it three attributes for the name, age, and the color of the leopard. Then I added two more methods that are about the behavioral traits of the leopard we are creating. These methods will print messages about the running and attacking of the leopard. If you want to understand it in a simpler form, you can consider the leopard a robot leopard. This will help you understand how Python helps in automating machines by modeling them on real-life objects.

Now that we have the structure of the class, we can move on to create different objects. I will add an instance to the Leopard class.

```
class Leopard():
    """This class will build the model of a leopard."""

    def __init__(self, lname, lage, lcolor):
        """here I will initialize the name, age and color attributes of the class."""
        self.lname = lname
        self.lage = lage
        self.lcolor = lcolor
```

```
    def run(self):
        print(self.lname.title() + " is running fast out in the wild.")
    def attack(self):
        print(self.lname.title() + " is now attacking a deer who is grazing in the meadow.")
leopard1 = Leopard('Tame', 9, 'yellow')
print("The name of the leopard is " + leopard1.lname.title() + ".")
print("The age of the leopard is " + str(leopard1.lage) + ".")
print("The color of the leopard is " + leopard1.lcolor.title() + ".")
= RESTART: C:/Users/saifia computers/Desktop/sample.py
The name of the leopard is Tame.
The age of the leopard is 9.
The color of the leopard is Yellow.
>>>
```

Now I will add more instances to the same class.

```
class Leopard():
    """This class will build the model of a leopard."""

    def __init__(self, lname, lage, lcolor):
        """here I will initialize the name, age and color attributes of the class."""
        self.lname = lname
        self.lage = lage
        self.lcolor = lcolor
    def run(self):
        print(self.lname.title() + " is running fast out in the wild.")
    def attack(self):
```

```
        print(self.lname.title() + " is now attacking a deer who is grazing in the meadow.")
leopard1 = Leopard('Tame', 9, 'yellow')
print("The name of the leopard is " + leopard1.lname.title() + ".")
print("The age of the leopard is " + str(leopard1.lage) + ".")
print("The color of the leopard is " + leopard1.lcolor.title() + ".")

leopard2 = Leopard('Fame', 8, 'snow white')
print("The name of the leopard is " + leopard2.lname.title() + ".")
print("The age of the leopard is " + str(leopard2.lage) + ".")
print("The color of the leopard is " + leopard2.lcolor.title() + ".")

leopard3 = Leopard('Storm', 11, 'yellow')
print("The name of the leopard is " + leopard3.lname.title() + ".")
print("The age of the leopard is " + str(leopard3.lage) + ".")
print("The color of the leopard is " + leopard3.lcolor.title() + ".")
= RESTART: C:/Users/saifia computers/Desktop/sample.py
The name of the leopard is Tame.
The age of the leopard is 9.
The color of the leopard is Yellow.
The name of the leopard is Fame.
The age of the leopard is 8.
The color of the leopard is Snow White.
The name of the leopard is Storm.
The age of the leopard is 11.
The color of the leopard is Yellow.
>>>
```

Now that I have created an instance for the Leopard class, I will now add to it some additional methods that will make the robot leopard run wildly and attack the prey to hunt his meal. This is going to be quite interesting.

```
class Leopard():
    """This class will build the model of a leopard."""

    def __init__(self, lname, lage, lcolor):
        """here I will initialize the name, age and color attributes of the class."""
        self.lname = lname
        self.lage = lage
        self.lcolor = lcolor
    def run(self):
        print(self.lname.title() + " is running fast out in the wild.")
    def attack(self):
        print(self.lname.title() + " is now attacking a deer who is grazing in the meadow.")
leopard1 = Leopard('Tame', 9, 'yellow')
print("The name of the leopard is " + leopard1.lname.title() + ".")
print("The age of the leopard is " + str(leopard1.lage) + ".")
print("The color of the leopard is " + leopard1.lcolor.title() + ".")
leopard1.run()
leopard1.attack()

leopard2 = Leopard('Fame', 8, 'snow white')
print("The name of the leopard is " + leopard2.lname.title() + ".")
print("The age of the leopard is " + str(leopard2.lage) + ".")
print("The color of the leopard is " + leopard2.lcolor.title() + ".")
```

```
leopard2.run()
leopard2.attack()
leopard3 = Leopard('Storm', 11, 'yellow')
print("The name of the leopard is " + leopard3.lname.title() + ".")
print("The age of the leopard is " + str(leopard3.lage) + ".")
print("The color of the leopard is " + leopard3.lcolor.title() + ".")
leopard3.run()
leopard3.attack()
= RESTART: C:/Users/saifia computers/Desktop/sample.py
The name of the leopard is Tame.
The age of the leopard is 9.
The color of the leopard is Yellow.
Tame is running fast out in the wild.
Tame is now attacking a deer who is grazing in the meadow.
The name of the leopard is Fame.
The age of the leopard is 8.
The color of the leopard is Snow White.
Fame is running fast out in the wild.
Fame is now attacking a deer who is grazing in the meadow.
The name of the leopard is Storm.
The age of the leopard is 11.
The color of the leopard is Yellow.
Storm is running fast out in the wild.
Storm is now attacking a deer who is grazing in the meadow.
>>>
```

Python creates two separate instances if you keep the name, age and color of

the leopard same. See the following example.

```
class Leopard():
    """This class will build the model of a leopard."""

    def __init__(self, lname, lage, lcolor):
        """here I will initialize the name, age and color attributes of the class."""
        self.lname = lname
        self.lage = lage
        self.lcolor = lcolor
    def run(self):
        print(self.lname.title() + " is running fast out in the wild.")
    def attack(self):
        print(self.lname.title() + " is now attacking a deer who is grazing in the meadow.")
leopard1 = Leopard('Tame', 9, 'yellow')
print("The name of the leopard is " + leopard1.lname.title() + ".")
print("The age of the leopard is " + str(leopard1.lage) + ".")
print("The color of the leopard is " + leopard1.lcolor.title() + ".")
leopard2 = Leopard('Tame', 9, 'yellow')
print("The name of the leopard is " + leopard2.lname.title() + ".")
print("The age of the leopard is " + str(leopard2.lage) + ".")
print("The color of the leopard is " + leopard2.lcolor.title() + ".")
= RESTART: C:/Users/saifia computers/Desktop/sample.py
The name of the leopard is Tame.
The age of the leopard is 9.
The color of the leopard is Yellow.
```

The name of the leopard is Tame.

The age of the leopard is 9.

The color of the leopard is Yellow.

>>>

The Fish Class

```
class Fish():
    """This class will build the model of a leopard."""

    def __init__(self, fname, fage, fcolor):
        """here I will initialize the name, age and color attributes of the class."""
        self.fname = fname
        self.fage = fage
        self.fcolor = fcolor
    def swim(self):
        print(self.fname.title() + " is swimming at a fast pace against the current.")
    def hunt(self):
        print(self.fname.title() + " is hunting smaller fish to feed itself.")
fish1 = Fish('Tuna', 2, 'yellow')
print("The name of the fish is " + fish1.fname.title() + ".")
print("The age of the fish is " + str(fish1.fage) + ".")
print("The color of the fish is " + fish1.fcolor.title() + ".")
fish1.swim()
fish1.hunt()
fish2 = Fish('whale', 50, 'blue & white')
print("The name of the fish is " + fish2.fname.title() + ".")
```

```
print("The age of the fish is " + str(fish2.fage) + ".")
print("The color of the fish is " + fish2.fcolor.title() + ".")
fish2.swim()
fish2.hunt()
= RESTART: C:/Users/saifia computers/Desktop/sample.py
The name of the fish is Tuna.
The age of the fish is 2.
The color of the fish is Yellow.
Tuna is swimming at a fast pace against the current.
Tuna is hunting smaller fish to feed itself.
The name of the fish is Whale.
The age of the fish is 50.
The color of the fish is Blue & White.
Whale is swimming at a fast pace against the current.
Whale is hunting smaller fish to feed itself.
>>>
```

The Bike Class

In this code sample, I will create bike class. I will add the model name, make, color and year of manufacturing to the class and display the information in a neatly formatted form.

```
class Bike():
    """This class will build the model of a bike."""

    def __init__(self, bmodel, bmake, bcolor, byear):

        self.bmodel = bmodel
        self.bmake = bmake
        self.bcolor = bcolor
```

```
        self.byear = byear
    def fullname(self):
        fullbikename = str(self.byear) + ' ' + self.bmodel + ' ' + self.bmake + ' '
+ self.bcolor
        return fullbikename.title()

bike1 = Bike('CG-125', 'Honda', 'blue', 2012)
print(bike1.fullname())
= RESTART: C:/Users/saifia computers/Desktop/sample.py
2012 Cg-125 Honda Blue
>>>
```

The process is similar to that of the creation of a Leopard class. I have defined the __init__ () method and the self-parameters. Four parameters will define the make, model, color and year of making of the bike. When you are creating a new instance to the bike class, you will have to define the make, model, year and color of the bike. If you miss one of the parameters while creating an instance, you will see an interpreter error just like the following.

```
class Bike():
    """This class will build the model of a bike."""

    def __init__(self, bmodel, bmake, bcolor, byear):

        self.bmodel = bmodel
        self.bmake = bmake
       self.bcolor = bcolor
        self.byear = byear
    def fullname(self):
        fullbikename = str(self.byear) + ' ' + self.bmodel + ' ' + self.bmake + ' '
+ self.bcolor
        return fullbikename.title()
```

```
bike1 = Bike('CG-125', 'blue', 2012)
print(bike1.fullname())
>>>= RESTART: C:/Users/saifia computers/Desktop/sample.py
Traceback (most recent call last):
  File "C:/Users/saifia computers/Desktop/sample.py", line 16, in <module>
    bike1 = Bike('CG-125', 'blue', 2012)
TypeError: __init__() missing 1 required positional argument: 'byear'
>>>
```

Just like the Leopard class, you can create as many instances for the Bike class as you need. This program is helpful if you are looking forward to owning a bike showroom. You can fill in the bike class with the latest information whenever a new bike gets registered with the showroom for sale. This is how you allow your customers to view each bike and its specifications in a fast and efficient way. I will now add more instances to the Bike class to show how you can store more information to the database through a working Bike class.

```
class Bike():
    """This class will build the model of a bike."""

    def __init__(self, bmodel, bmake, bcolor, byear):

        self.bmodel = bmodel
        self.bmake = bmake
        self.bcolor = bcolor
        self.byear = byear
    def fullname(self):
        fullbikename = "We have a bike that hit the markets in " + str(self.byear) + ". The model is " + self.bmodel + ". The bike is manufactured by " + self.bmake + ". Its color is " + self.bcolor + "."
        return fullbikename.title()
```

```
bike1 = Bike('CG-125', 'Honda', 'blue', 2012)
print(bike1.fullname())
bike2 = Bike('F 900 R', 'BMW', 'black', 2014)
print(bike2.fullname())
bike3 = Bike('F 900 XR', 'BMW', 'blue', 2014)
print(bike3.fullname())
bike4 = Bike('R 1250 RT', 'BMW', 'brown', 2016)
print(bike4.fullname())
bike5 = Bike('Heritage Classic', 'Harley Davidson', 'black', 2018)
print(bike4.fullname())
= RESTART: C:/Users/saifia computers/Desktop/sample.py
We Have A Bike That Hit The Markets In 2012. The Model Is Cg-125. The Bike Is Manufactured By Honda. Its Color Is Blue.
We Have A Bike That Hit The Markets In 2014. The Model Is F 900 R. The Bike Is Manufactured By BMW. Its Color Is Black.
We Have A Bike That Hit The Markets In 2014. The Model Is F 900 Xr. The Bike Is Manufactured By BMW. Its Color Is Blue.
We Have A Bike That Hit The Markets In 2016. The Model Is R 1250 Rt. The Bike Is Manufactured By BMW. Its Color Is Brown.
We Have A Bike That Hit The Markets In 2016. The Model Is R 1250 Rt. The Bike Is Manufactured By BMW. Its Color Is Brown.
>>>
```

Each attribute in the Bike class demands an initial value. You can set the initial value at zero. It also can be an empty string. When you are running a showroom, you need to tell your customers how many kilometers the bike has run on the road. To achieve this objective, you can integrate a method into the program. See the changes in the code. I will include an odometer reading method for the Bike class.

```
class Bike():
```

```
    """This class will build the model of a bike."""

    def __init__(self, bmodel, bmake, bcolor, byear):

        self.bmodel = bmodel

        self.bmake = bmake

        self.bcolor = bcolor

        self.byear = byear

        self.odometer_reading = 0

    def fullname(self):

        fullbikename = "We have a bike that hit the markets in " + str(self.byear) + ". The model is " + self.bmodel + ". The bike is manufactured by " + self.bmake + ". Its color is " + self.bcolor + "."

        return fullbikename.title()

    def read_odometer(self):

        print("This bike has run " + str(self.odometer_reading) + " kilometers on the road.")


bike1 = Bike('CG-125', 'Honda', 'blue', 2012)

print(bike1.fullname())

bike1.read_odometer()

bike2 = Bike('F 900 R', 'BMW', 'black', 2014)

print(bike2.fullname())

bike2.read_odometer()

bike3 = Bike('F 900 XR', 'BMW', 'blue', 2014)

print(bike3.fullname())

bike3.read_odometer()

bike4 = Bike('R 1250 RT', 'BMW', 'brown', 2016)
```

```
print(bike4.fullname())
bike4.read_odometer()
bike5 = Bike('Heritage Classic', 'Harley Davidson', 'black', 2018)
print(bike4.fullname())
bike5.read_odometer()
= RESTART: C:/Users/saifia computers/Desktop/sample.py
We Have A Bike That Hit The Markets In 2012. The Model Is Cg-125. The Bike Is Manufactured By Honda. Its Color Is Blue.
This bike has run 0 kilometers on the road.
We Have A Bike That Hit The Markets In 2014. The Model Is F 900 R. The Bike Is Manufactured By BMW. Its Color Is Black.
This bike has run 0 kilometers on the road.
We Have A Bike That Hit The Markets In 2014. The Model Is F 900 Xr. The Bike Is Manufactured By BMW. Its Color Is Blue.
This bike has run 0 kilometers on the road.
We Have A Bike That Hit The Markets In 2016. The Model Is R 1250 Rt. The Bike Is Manufactured By BMW. Its Color Is Brown.
This bike has run 0 kilometers on the road.
We Have A Bike That Hit The Markets In 2016. The Model Is R 1250 Rt. The Bike Is Manufactured By BMW. Its Color Is Brown.
This bike has run 0 kilometers on the road.
>>>
```

Python calls the __init__() method to form a new instance. It stores the values in the form of attributes just as it did for the past example. Python has now created a new attribute and adjusts its value to zero. Coupled with the attribute comes a new method, namely read_odometer(). This is how your customers can easily read the mileage of the bike. It is also helpful for you, as you can easily track how many miles your car has run.

You have the power to change the value of the attributes in different ways.

You can directly change the value of the attribute by an instance. You can set its value with the help of a method or increment the same by a method. In the following code sample, I will test how we can make the above-mentioned changes. I have also changed the read_odometer() method to reading_odometer() method to make more current and interactive.

```
class Bike():
    """This class will build the model of a bike."""

    def __init__(self, bmodel, bmake, bcolor, byear):

        self.bmodel = bmodel
        self.bmake = bmake
        self.bcolor = bcolor
        self.byear = byear
        self.odometer_reading = 0
    def fullname(self):
        fullbikename = "We have a bike that hit the markets in " + str(self.byear) + ". The model is " + self.bmodel + ". The bike is manufactured by " + self.bmake + ". Its color is " + self.bcolor + "."
        return fullbikename.title()
    def reading_odometer(self):
        print("This bike has run " + str(self.odometer_reading) + " kilometers on the road.")

bike1 = Bike('CG-125', 'Honda', 'blue', 2012)
print(bike1.fullname())
bike1.odometer_reading = 21
bike1.reading_odometer()
bike2 = Bike('F 900 R', 'BMW', 'black', 2014)
print(bike2.fullname())
```

```
bike2.odometer_reading = 27
bike2.reading_odometer()
bike3 = Bike('F 900 XR', 'BMW', 'blue', 2014)
print(bike3.fullname())
bike3.odometer_reading = 30
bike3.reading_odometer()
bike4 = Bike('R 1250 RT', 'BMW', 'brown', 2016)
print(bike4.fullname())
bike4.reading_odometer()
bike5 = Bike('Heritage Classic', 'Harley Davidson', 'black', 2018)
print(bike4.fullname())
bike5.reading_odometer()
>>>= RESTART: C:/Users/saifia computers/Desktop/sample.py
We Have A Bike That Hit The Markets In 2012. The Model Is Cg-125. The Bike Is Manufactured By Honda. Its Color Is Blue.
This bike has run 21 kilometers on the road.
We Have A Bike That Hit The Markets In 2014. The Model Is F 900 R. The Bike Is Manufactured By BMW. Its Color Is Black.
This bike has run 27 kilometers on the road.
We Have A Bike That Hit The Markets In 2014. The Model Is F 900 Xr. The Bike Is Manufactured By BMW. Its Color Is Blue.
This bike has run 30 kilometers on the road.
We Have A Bike That Hit The Markets In 2016. The Model Is R 1250 Rt. The Bike Is Manufactured By BMW. Its Color Is Brown.
This bike has run 0 kilometers on the road.
We Have A Bike That Hit The Markets In 2016. The Model Is R 1250 Rt. The Bike Is Manufactured By BMW. Its Color Is Brown.
This bike has run 0 kilometers on the road.
```

>>>

You can also change the default value at 50 kilometers to manage the difference of mileage consumed in transporting the bike from one place to another.

```
class Bike():
    """This class will build the model of a bike."""

    def __init__(self, bmodel, bmake, bcolor, byear):

        self.bmodel = bmodel

        self.bmake = bmake

        self.bcolor = bcolor

        self.byear = byear

        self.odometer_reading = 50
    def fullname(self):

        fullbikename = "We have a bike that hit the markets in " + str(self.byear) + ". The model is " + self.bmodel + ". The bike is manufactured by " + self.bmake + ". Its color is " + self.bcolor + "."

        return fullbikename.title()

    def reading_odometer(self):

        print("This bike has run " + str(self.odometer_reading) + " kilometers on the road.")


bike1 = Bike('CG-125', 'Honda', 'blue', 2012)

print(bike1.fullname())

bike1.odometer_reading = 100

bike1.reading_odometer()

bike2 = Bike('F 900 R', 'BMW', 'black', 2014)

print(bike2.fullname())
```

bike2.odometer_reading = 500

bike2.reading_odometer()

bike3 = Bike('F 900 XR', 'BMW', 'blue', 2014)

print(bike3.fullname())

bike3.odometer_reading = 700

bike3.reading_odometer()

bike4 = Bike('R 1250 RT', 'BMW', 'brown', 2016)

print(bike4.fullname())

bike4.reading_odometer()

bike5 = Bike('Heritage Classic', 'Harley Davidson', 'black', 2018)

print(bike4.fullname())

bike5.reading_odometer()

= RESTART: C:/Users/saifia computers/Desktop/sample.py

We Have A Bike That Hit The Markets In 2012. The Model Is Cg-125. The Bike Is Manufactured By Honda. Its Color Is Blue.

This bike has run 100 kilometers on the road.

We Have A Bike That Hit The Markets In 2014. The Model Is F 900 R. The Bike Is Manufactured By BMW. Its Color Is Black.

This bike has run 500 kilometers on the road.

We Have A Bike That Hit The Markets In 2014. The Model Is F 900 Xr. The Bike Is Manufactured By BMW. Its Color Is Blue.

This bike has run 700 kilometers on the road.

We Have A Bike That Hit The Markets In 2016. The Model Is R 1250 Rt. The Bike Is Manufactured By BMW. Its Color Is Brown.

This bike has run 50 kilometers on the road.

We Have A Bike That Hit The Markets In 2016. The Model Is R 1250 Rt. The Bike Is Manufactured By BMW. Its Color Is Brown.

This bike has run 50 kilometers on the road.

>>>

Proper Modification of Values

It is quite helpful to have a bunch of methods that would update different attributes of your program. Instead of directly accessing multiple attributes, you can pass the latest value to a newly added method and let it handle updating the program. The program will do the updating internally and you do not have to worry about it anymore. The new method will be dubbed as updating_the_odometer().

```
class Bike():
    """This class will build the model of a bike."""

    def __init__(self, bmodel, bmake, bcolor, byear):

        self.bmodel = bmodel
        self.bmake = bmake
        self.bcolor = bcolor
        self.byear = byear
        self.odometer_reading = 50
    def fullname(self):
        fullbikename = "We have a bike that hit the markets in " + str(self.byear) + ". The model is " + self.bmodel + ". The bike is manufactured by " + self.bmake + ". Its color is " + self.bcolor + "."
        return fullbikename.title()
    def reading_odometer(self):
        print("This bike has run" + str(self.odometer_reading) + " kilometers on the road.")
    def updating_the_odometer(self, bmileage):
        self.odometer_reading = bmileage

bike1 = Bike('CG-125', 'Honda', 'blue', 2012)
```

```
print(bike1.fullname())
bike1.updating_the_odometer(100)
bike1.reading_odometer()
bike2 = Bike('F 900 R', 'BMW', 'black', 2014)
print(bike2.fullname())
bike2.updating_the_odometer(1000)
bike2.reading_odometer()
bike3 = Bike('F 900 XR', 'BMW', 'blue', 2014)
print(bike3.fullname())
bike3.updating_the_odometer(700)
bike3.reading_odometer()
bike4 = Bike('R 1250 RT', 'BMW', 'brown', 2016)
print(bike4.fullname())
bike4.reading_odometer()
bike5 = Bike('Heritage Classic', 'Harley Davidson', 'black', 2018)
print(bike4.fullname())
bike5.reading_odometer()
= RESTART: C:/Users/saifia computers/Desktop/sample.py
We Have A Bike That Hit The Markets In 2012. The Model Is Cg-125. The Bike Is Manufactured By Honda. Its Color Is Blue.
This bike has 100 kilometers on the road.
We Have A Bike That Hit The Markets In 2014. The Model Is F 900 R. The Bike Is Manufactured By BMW. Its Color Is Black.
This bike has 1000 kilometers on the road.
We Have A Bike That Hit The Markets In 2014. The Model Is F 900 Xr. The Bike Is Manufactured By BMW. Its Color Is Blue.
This bike has 700 kilometers on the road.
```

We Have A Bike That Hit The Markets In 2016. The Model Is R 1250 Rt. The Bike Is Manufactured By BMW. Its Color Is Brown.

This bike has 50 kilometers on the road.

We Have A Bike That Hit The Markets In 2016. The Model Is R 1250 Rt. The Bike Is Manufactured By BMW. Its Color Is Brown.

This bike has 50 kilometers on the road.

>>>

You can make further experiments with the odometer method. A major problem in a showroom is keeping tabs on who is reversing the odometer of the motorbikes. If you are a true businessman, you will not like to dupe your customers. However, sometimes it is not that you who want to dupe the customers. It is your employees who are trying to bag extra profit in addition to their commission. You must stop this practice if you want to live up to your customers' expectations and keep your reputation intact. You can add some logic to your program to ensure no rolling back of the odometer by anyone. I will integrate an if-else statement to the Bike class and do a few changes to make it possible.

```
class Bike():
    """This class will build the model of a bike."""

    def __init__(self, bmodel, bmake, bcolor, byear):

        self.bmodel = bmodel
        self.bmake = bmake
        self.bcolor = bcolor
        self.byear = byear
        self.odometer_reading = 0
    def fullname(self):
        fullbikename = "We have a bike that hit the markets in " + str(self.byear) + ". The model is " + self.bmodel + ". The bike is manufactured by " + self.bmake + ". Its color is " + self.bcolor + "."
```

```
        return fullbikename.title()
    def reading_odometer(self):
        print("This bike has run " + str(self.odometer_reading) + " kilometers on the road.")
    def updating_the_odometer(self, bmileage):
        self.odometer_reading = bmileage
        if bmileage >= self.odometer_reading:
            self.odometer_reading = bmileage
        else:
            print("You are not authorized to roll back the reading of the odometer.")

bike1 = Bike('CG-125', 'Honda', 'blue', 2012)
print(bike1.fullname())
bike1.updating_the_odometer(100)
bike1.reading_odometer()
bike2 = Bike('F 900 R', 'BMW', 'black', 2014)
print(bike2.fullname())
bike2.updating_the_odometer(1000)
bike2.reading_odometer()
bike3 = Bike('F 900 XR', 'BMW', 'blue', 2014)
print(bike3.fullname())
bike3.updating_the_odometer(700)
bike3.reading_odometer()
bike4 = Bike('R 1250 RT', 'BMW', 'brown', 2016)
print(bike4.fullname())
bike4.updating_the_odometer(40)
```

bike4.reading_odometer()

bike5 = Bike('Heritage Classic', 'Harley Davidson', 'black', 2018)

print(bike5.fullname())

bike5.updating_the_odometer(0)

bike5.reading_odometer()

= RESTART: C:/Users/saifia computers/Desktop/sample.py

We Have A Bike That Hit The Markets In 2012. The Model Is Cg-125. The Bike Is Manufactured By Honda. Its Color Is Blue.

This bike has run 100 kilometers on the road.

We Have A Bike That Hit The Markets In 2014. The Model Is F 900 R. The Bike Is Manufactured By BMW. Its Color Is Black.

This bike has run 1000 kilometers on the road.

We Have A Bike That Hit The Markets In 2014. The Model Is F 900 Xr. The Bike Is Manufactured By BMW. Its Color Is Blue.

This bike has run 700 kilometers on the road.

We Have A Bike That Hit The Markets In 2016. The Model Is R 1250 Rt. The Bike Is Manufactured By BMW. Its Color Is Brown.

This bike has run 40 kilometers on the road.

We Have A Bike That Hit The Markets In 2018. The Model Is Heritage Classic. The Bike Is Manufactured By Harley Davidson. Its Color Is Black.

This bike has run 0 kilometers on the road.

>>>

You can increase the value of an attribute by introducing a simple method to the program. I will another method to the class to make it work. I will add incremental values to each of the five instances I have created. The method will tell Python to add up the incremental value to the existing value and run the program. The incremented value will be displayed in a separate print statement in the code.

class Bike():

```
    """This class will build the model of a bike."""

    def __init__(self, bmodel, bmake, bcolor, byear):

        self.bmodel = bmodel
        self.bmake = bmake
        self.bcolor = bcolor
        self.byear = byear
        self.odometer_reading = 0
    def fullname(self):
        fullbikename = "We have a bike that hit the markets in " + str(self.byear) + ". The model is " + self.bmodel + ". The bike is manufactured by " + self.bmake + ". Its color is " + self.bcolor + "."
        return fullbikename.title()
    def reading_odometer(self):
        print("This bike has run " + str(self.odometer_reading) + " kilometers on the road.")
    def updating_the_odometer(self, bmileage):
        self.odometer_reading = bmileage
        if bmileage >= self.odometer_reading:
            self.odometer_reading = bmileage
        else:
            print("You are not authorized to roll back the reading of the odometer.")
    def incrementing_odometer(self, bmileage):
        self.odometer_reading += bmileage

bike1 = Bike('CG-125', 'Honda', 'blue', 2012)
print(bike1.fullname())
```

```
bike1.updating_the_odometer(100)
bike1.reading_odometer()
bike1.incrementing_odometer(1000)
bike1.reading_odometer()
bike2 = Bike('F 900 R', 'BMW', 'black', 2014)
print(bike2.fullname())
bike2.updating_the_odometer(1000)
bike2.reading_odometer()
bike2.incrementing_odometer(500)
bike2.reading_odometer()
bike3 = Bike('F 900 XR', 'BMW', 'blue', 2014)
print(bike3.fullname())
bike3.updating_the_odometer(700)
bike3.reading_odometer()
bike3.incrementing_odometer(1000)
bike3.reading_odometer()
bike4 = Bike('R 1250 RT', 'BMW', 'brown', 2016)
print(bike4.fullname())
bike4.updating_the_odometer(40)
bike4.reading_odometer()
bike4.incrementing_odometer(1000)
bike4.reading_odometer()
bike5 = Bike('Heritage Classic', 'Harley Davidson', 'black', 2018)
print(bike5.fullname())
bike5.updating_the_odometer(0)
bike5.reading_odometer()
```

bike5.incrementing_odometer(10000)

bike5.reading_odometer()

= RESTART: C:/Users/saifia computers/Desktop/sample.py

We Have A Bike That Hit The Markets In 2012. The Model Is Cg-125. The Bike Is Manufactured By Honda. Its Color Is Blue.

This bike has run 100 kilometers on the road.

This bike has run 1100 kilometers on the road.

We Have A Bike That Hit The Markets In 2014. The Model Is F 900 R. The Bike Is Manufactured By BMW. Its Color Is Black.

This bike has run 1000 kilometers on the road.

This bike has run 1500 kilometers on the road.

We Have A Bike That Hit The Markets In 2014. The Model Is F 900 Xr. The Bike Is Manufactured By BMW. Its Color Is Blue.

This bike has run 700 kilometers on the road.

This bike has run 1700 kilometers on the road.

We Have A Bike That Hit The Markets In 2016. The Model Is R 1250 Rt. The Bike Is Manufactured By BMW. Its Color Is Brown.

This bike has run 40 kilometers on the road.

This bike has run 1040 kilometers on the road.

We Have A Bike That Hit The Markets In 2018. The Model Is Heritage Classic. The Bike Is Manufactured By Harley Davidson. Its Color Is Black.

This bike has run 0 kilometers on the road.

This bike has run 10000 kilometers on the road.

>>>

Chapter Nine: The Inheritance Class

Now that you have learned how to write a class, it is pertinent to mention that Python classes are well known for the ease of use they offer to programmers. Once you have written a class, you can reuse it multiple times. There is a process called inheritance in which a subclass is inherited from the parent class. The inherited class is named that way because it inherits the attributes of the parent class. The inherited class is dubbed as the child class. It can use each attribute and method of the parent class. However, you also can create new attributes only for the child class.

Just as you did for the parent class, you will also have to use the __init__() method for the child class. I will create a child class of racer bikes.

```
class Bike():
    """This class will build the model of a bike."""

    def __init__(self, bmodel, bmake, bcolor, byear):

        self.bmodel = bmodel
        self.bmake = bmake
        self.bcolor = bcolor
        self.byear = byear
        self.odometer_reading = 0
    def fullname(self):
        fullbikename = "We have a bike that hit the markets in " + str(self.byear) + ". The model is " + self.bmodel + ". The bike is manufactured by " + self.bmake + ". Its color is " + self.bcolor + "."
        return fullbikename.title()
    def reading_odometer(self):
        print("This bike has run " + str(self.odometer_reading) + " kilometers on the road.")
```

```
    def updating_the_odometer(self, bmileage):
        self.odometer_reading = bmileage
        if bmileage >= self.odometer_reading:
            self.odometer_reading = bmileage
        else:
            print("You are not authorized to roll back the reading of the odometer.")
    def incrementing_odometer(self, bmileage):
        self.odometer_reading += bmileage

class RacerBike(Bike):
    def __init__(self, bmodel, bmake, bcolor, byear):
        super().__init__(bmodel, bmake, bcolor, byear)
racer1 = RacerBike('URS: Gravel Riding', 'BMC', 'Grey', '2017')
print(racer1.fullname())
racer2 = RacerBike('Trackmachine', 'BMC', 'Blue', '2015')
print(racer2.fullname())
racer3 = RacerBike('Alpenchallenge', 'BMC', 'Red', '2012')
print(racer2.fullname())
= RESTART: C:/Users/saifia computers/Desktop/sample.py
We Have A Bike That Hit The Markets In 2017. The Model Is Urs: Gravel Riding. The Bike Is Manufactured By Bmc. Its Color Is Grey.
We Have A Bike That Hit The Markets In 2015. The Model Is Trackmachine. The Bike Is Manufactured By Bmc. Its Color Is Blue.
We Have A Bike That Hit The Markets In 2015. The Model Is Trackmachine. The Bike Is Manufactured By Bmc. Its Color Is Blue.
>>>
```

The most important thing to keep in mind while creating a child class is to

keep the child class inside the parent class. The name of the child class must include parenthesis that carry the name of the parent class. I have added one additional function, the super function that aids Python in forming connections between the child class and the parent class. The child class has taken all the attributes of the parent class. I have not yet added any special attribute to the racer bike.

Child Class in Python 2.7

If you are using Python 2.7, the child class will appear to be a bit different. See the following code and note the difference.

```
class Bike():
    """This class will build the model of a bike."""

    def __init__(self, bmodel, bmake, bcolor, byear):

        self.bmodel = bmodel
        self.bmake = bmake
        self.bcolor = bcolor
        self.byear = byear
        self.odometer_reading = 0
    def fullname(self):
        fullbikename = "We have a bike that hit the markets in " + str(self.byear) + ". The model is " + self.bmodel + ". The bike is manufactured by " + self.bmake + ". Its color is " + self.bcolor + "."
        return fullbikename.title()
    def reading_odometer(self):
        print("This bike has run " + str(self.odometer_reading) + " kilometers on the road.")
    def updating_the_odometer(self, bmileage):
        self.odometer_reading = bmileage
        if bmileage >= self.odometer_reading:
```

```
            self.odometer_reading = bmileage
        else:
            print("You are not authorized to roll back the reading of the odometer.")
    def incrementing_odometer(self, bmileage):
        self.odometer_reading += bmileage

class RacerBike(Bike):
    def __init__(self, bmodel, bmake, bcolor, byear):
        super(RacerBike,self).__init__(bmodel, bmake, bcolor, byear)
racer1 = RacerBike('URS: Gravel Riding', 'BMC', 'Grey', '2017')
print(racer1.fullname())
racer2 = RacerBike('Trackmachine', 'BMC', 'Blue', '2015')
print(racer2.fullname())
racer3 = RacerBike('Alpenchallenge', 'BMC', 'Red', '2012')
print(racer2.fullname())
```

The only change I made was in the super function. I filled in the parenthesis with the name of the child class and self-parameter. An interesting thing is that this change does not affect the result of the program, no matter if you do it in Python 3 or 2.7. However, the former technique will not work in 2.7. You can see the result of the program as under:

```
= RESTART: C:/Users/saifia computers/Desktop/sample.py
We Have A Bike That Hit The Markets In 2017. The Model Is Urs: Gravel Riding. The Bike Is Manufactured By Bmc. Its Color Is Grey.
We Have A Bike That Hit The Markets In 2015. The Model Is Trackmachine. The Bike Is Manufactured By Bmc. Its Color Is Blue.
We Have A Bike That Hit The Markets In 2015. The Model Is Trackmachine. The Bike Is Manufactured By Bmc. Its Color Is Blue.
>>>
```