# PYTHON PROGRAMMING

FOR BEGINNERS

THE COMPLETE BEGINNER'S GUIDE TO PYTHON PROGRAMMING

B R U C E   B E R K E

# PYTHON PROGRAMMING

FOR BEGINNERS

THE COMPLETE BEGINNER'S GUIDE TO PYTHON PROGRAMMING

B R U C E  B E R K E

# *Python Programming for Beginners*

## *The Complete Beginner's Guide To Python Programming*

**Bruce Berke**

## Table of Contents

# Introduction

Hello, guys! I hope all of you are doing well. Today, I am bringing you a book that you can use to learn Python. If you are a beginner with basic knowledge about computers and computer programing, then this is the book for you. You should own this book if you want to become an intermediate level programmer of Python, as it contains all the right ingredients. Python is a high-level programming language. It was introduced in 1985. The creator of Python is Guido van Rossum. Python is very famous among developers and researchers. Some use this programming language for web development, and some use it for writing class libraries. Class libraries are further used by other programmers since they are written in a way that makes them re-usable. Researchers use Python as it is widely known among the research community and is powerful enough to be used for simulations and experiments. I will start from scratch and teach you how to install Python; if you are a reader with some experience in Python, I must tell you that this book might be a bit slow paced for you. So, let’s start then.

# Chapter 1: Environment Setup

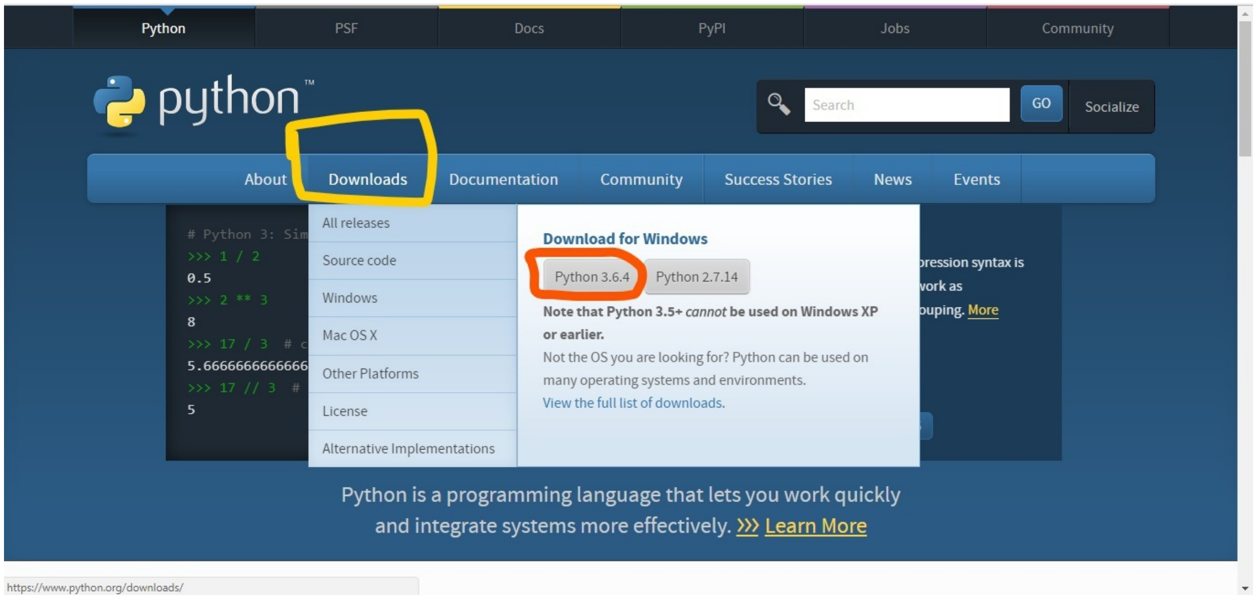
Before we start writing programs in Python, we need to install necessary tools that would allow us to do the aforementioned task. You will need to install a Python interpreter and an Integrated Development Environment (IDE). Multiple Python Interpreters are available, and the following list names them all:

- CPython
- IronPython
- Anaconda
- PyPy
- Jython

Each interpreter is somehow different from the other, and the differences between each does not concern you as a beginner. In this book, we will use CPython, which is a standard Python interpreter. To download CPython, we will use the official Python website. To make things simpler, I am pasting a link below that will lead you to the exact page to which you need to go in order to download the latest version of CPython.

https://www.python.org/

Once you click the link above, a webpage will open in your web browser. This is the main page that contains all the news, documentation, and download links for different releases of Python. Hover on the Download button present in the menu bar at the top. I am adding a screenshot too, just to make things clearer.



```
# Python 3: Sim
>>> 1 / 2
0.5
>>> 2 ** 3
8
>>> 17 / 3  # c
5.6666666666666
>>> 17 // 3  #
5
```

**Download for Windows**

**Note that Python 3.5+ *cannot* be used on Windows XP or earlier.**

Not the OS you are looking for? Python can be used on many operating systems and environments.

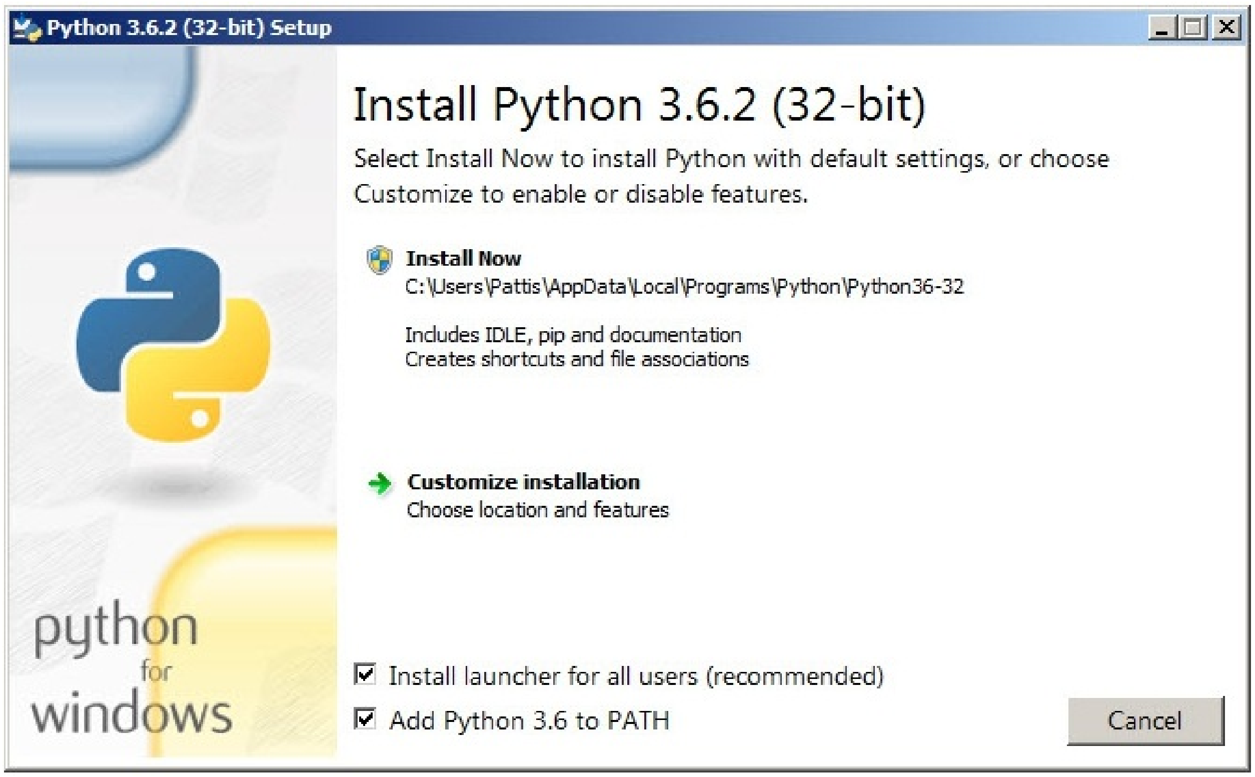
View the full list of downloads.

ression syntax is
work as
ouping.

Python is a programming language that lets you work quickly and integrate systems more effectively. >>> Learn More

Once you hover on the Downloads option, a tab will open up. Refer to the screenshot mentioned before this paragraph. It shows how the actual web page will look when the tab opens. Click on the link that has been highlighted with a red-orange marker. This will download an executable version of Python's latest version, which in this case is 3.6.4.

Double click the .exe file or just press the enter key after selecting the .exe file so that you can start the setup application. The setup window will look similar to the one shown below:



Install Python 3.6.2 (32-bit)

Select Install Now to install Python with default settings, or choose Customize to enable or disable features.

**Install Now**
C:\Users\Pattis\AppData\Local\Programs\Python\Python36-32

Includes IDLE, pip and documentation
Creates shortcuts and file associations

**Customize installation**
Choose location and features

- [✓] Install launcher for all users (recommended)
- [✓] Add Python 3.6 to PATH

The reason why I added the image above is that, both options (`Install launcher for all users` and `Add Python 3.6 to PATH` ) must be selected. You don't need to customize the installation at this stage; you might need it once you acquire a great amount of knowledge regarding Python. Adding Python to the Path will allow you to use Python from the command prompt. This comes in handy when you want to execute Python commands from `window's cmd` ; the active directory of the `window's cmd` doesn't matter this way.

After you install Python, open the command prompt in your Windows and write `python` and press enter.

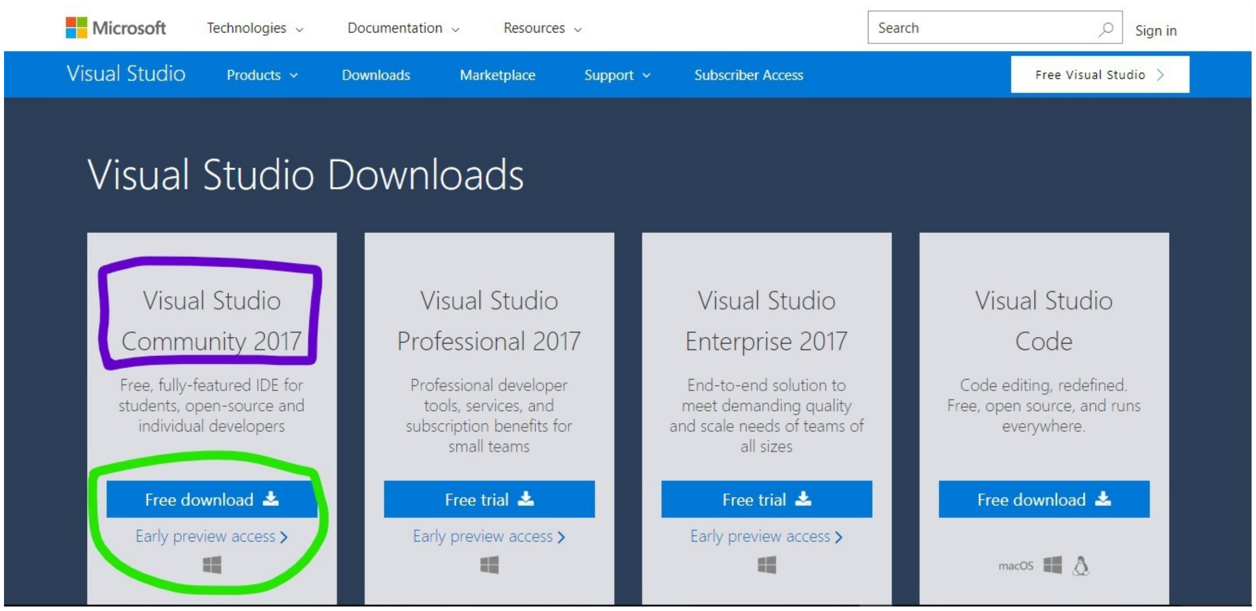
***Output:***
```
C:\Users\bruce>python
```

```
Python 3.6.3 (v3.6.3:2c5fed8, Oct 3 2017, 17:26:49) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

I have installed `3.6.3` version of Python on my system, so it’s showing that. You will see the version that you have installed in your system. If the query that I mentioned earlier works, then it means that Python has been installed correctly in your system and Python has also been added in the environment variables of Windows.

Now that we have installed Python, it’s time now to install an IDE. Numerous ways are available for using Python,: Text editors like notepad++, Visual Studio Code, Sublime text or even Window’s good old Notepad application can be used to write Python scripts. You can simply use Window’s cmd to execute that file. I am only discussing Window’s cmd here because in this book the examples only use the Window’s cmd. Extensive support for Python in Linux and its distributions is also available. Feel free to check them out after you complete this book. The procedure of using cmd to execute files often add complexity to a simpler task which is not the goal of this book. Avoiding the complexities, this book focuses on the easy and basic stuff related to Python. I used Visual Studio to prepare the examples in this book. To download Visual Studio, use the following link:

https://www.visualstudio.com/downloads/


Microsoft
Technologies
Documentation
Resources
Search
Sign in
Visual Studio
Products
Downloads
Marketplace
Support
Subscriber Access
Free Visual Studio
Visual Studio Downloads
Visual Studio Community 2017
Free, fully-featured IDE for students, open-source and individual developers
Free download
Early preview access
Visual Studio Professional 2017
Professional developer tools, services, and subscription benefits for small teams
Free trial
Early preview access
Visual Studio Enterprise 2017
End-to-end solution to meet demanding quality and scale needs of teams of all sizes
Free trial
Early preview access
Visual Studio Code
Code editing, redefined. Free, open source, and runs everywhere.
Free download
macOS

In the section: `Visual Studio Community 2017`, click on the `Free Download` button to download the setup of Visual Studio Community. The setup of Visual Studio is fairly easy. At the start, the setup will ask you to acknowledge their terms and conditions. Once you do that, the setup will move ahead and do its thing. Then after some time, you will be asked to choose the workload. A workload is basically the type of environment that should be configured. In our case, we only need to configure our Visual Studio for Python Development. So, select Python development and install. Click finish to end the installation process. A launch button will be present there too. Use it to start Visual Studio.

Everything related to the installation is complete. Now, it’s time to use the tools that we have configured so far in this chapter for writing a program in Python.

# Chapter 2: Python's Basics

In the last chapter, I showed you how to run a Python command in command prompt. The command that I used in the last chapter showed me the version of Python installed in my computer. I want you to rerun that command again. A cursor keeps blinking in front of the line that looks like:

```
>>>
```

The cursor is waiting for you to write a programming instruction in Python and execute it. For example, `print ("Hello World!")` will generate:

***Output:***

```
C:\Users\x>python
Python 3.6.3 (v3.6.3:2c5fed8, Oct  3 2017, 17:26:49) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print ("Hello World!")
Hello World!
>>>
```

The statement in red is basically the output of the print command. In the terminal, the cursor keeps on blinking after the execution of each Python script. This means that the Python interpreter expects another Python command. You can close the Python interpreter by pressing CTRL+Z and pressing the enter key subsequently.

The information above explains a way that can be used to write programs in Python. Some developers prefer using the terminal for writing programs, whilst others prefer using an IDE. As we are going to use an IDE in this book, go ahead and launch Visual Studio. Click on the `New Project` option. In the new project window, look for an option called `Templates` . Expand that option and then select `Python` . Now, after selecting the programming language that you want to use for the new project, it's time for you to select the type of Python project that you want to create. Select `Python Application` , give it a name, and create the project.

The method described above will be used throughout this book to create new projects. All Python code files have a `.py` extension. So, when you create a new project, Visual Studio is smart enough to know which file should be opened right after the project creation. The name of my project is `FirstPythonApplication` . So, it will contain a file named exactly the same as the name of the project. This file will have a `.py` extension.

Inside the file, write the following programming statement and click on the start button.

```
print ("Hello, Python from Visual Studio!")
```

The start button will have a green play button-like icon. Starting the program will trigger two very basic operations. The first operation will check for any syntactical errors present in the code file. If errors are present, Visual Studio will alert you about them. There are two scenarios here. The first scenario requires you to rectify the errors present in your script before trying to run Python code again. The other scenario is to ignore the warning and force Visual Studio to execute the code. Program will not work as it is supposed to in the later scenario.

The second operation is executing the programming statements. Now, to explain this phenomenon in simpler terms, I can say that, after checking the errors, the programming statements are interpreted by the interpreter, installed in our computer which in our case is `CPython` . It translates the instructions and sends them to the operating system which shows us the output. Upon successful execution of the program, a terminal window will open. On the terminal window, you will see the output similar to the one shown here:

**Example:**

```
print ("Hello, Python from Visual Studio!")
```

***Output:***

```
Hello, Python from Visual Studio!
Press any key to continue . . .
```

Writing a Python program using the shell is known as interactive programing, and using an IDE or simply executing a `.py` file is called scripting.

Python contains a list of reserved keywords. These keywords cannot be used as identifiers in Python. Identifiers are used to identify anything in Python. Things that could need identifying are variables, objects, classes, methods, etc. in Python. Another thing to know regarding Python is that it is another case delicate programming language. This means that two identifiers such as `PLAY` and `play` will be considered different by Python. The list of keywords in python is:

- and
- exec
- not
- assert
- finally
- break

- or
- for
- pass
- except
- yield
- in
- elif
- del
- import
- from
- class
- is
- else
- def
- if
- global
- continue
- print
- raise
- return
- with
- try
- while

You cannot use the keywords above as identifiers. Python's syntax is different from many high-level languages like Java, C#, PHP, etc. In Python, usually there are no semicolons used to terminate a programming statement. Semicolons in Python are only used if you want to write two programming statements into one line. Another thing in Python, which is different from other high-level languages, is that there are no parenthesis available. That means in order to tell Python that a certain set of statements are a part of a block of code, you need to indent each statement correctly. If your indentation isn't correct, you might encounter some errors. For example, this is how you can differentiate between `if` and `else` statements in Python:

**Example:**

```
if True:
   print "The Statement is true"
else:
   print "The Statement is false"
```

The above code block will work fine, as it has been indented correctly. However, if we remove spaces before each print statement, you will encounter an error, as the interpreter will not be able to differentiate between different code statements.

**Example:**

```
if True:
print "The Statement is true"
else:
print "The Statement is false"
```

In short, all the programming statements inside a block must have an equal number of whitespaces for them to constitute a block of code. The concept of indentation is not something one should worry about. Visual Studio is a very powerful tool, and one of the reasons behind its popularity is understanding the code and indenting it. So, it is going to be there for you whilst you write indented code. There is no need to pull your hair out over it.

As I suggested earlier, in other languages, each programming statement is terminated using the semi colon. But in Python, each line is terminated by a new line. A programing statement can span over different lines with the use of a continuation operator (\). A continuation operator joins programming statements spanned over multiple rows for the purpose of improved visibility.

**Example:**

```
Total_Items = item_1 + \
              item_2 + \
              item_3
```

The statement above is equal to the programming statement below:

```
Total_Items = item_1 + item_2 + item_3
```

The statements enclosed in round, square brackets or parenthesis can be written in multiple lines without using the continuation operator.

**Example:**

```
Total_Items = ['item_1', 'item_2', 'item_3', 'item_4', 'item_5']
```

Python also supports the use of single (‘), double (“) and triple (‘’ or “””) quotation marks. The `#` sign is used at the start of each comment in Python.

**Example:**

```
# With a hash character at the start, this is what a comment looks like in Python.
# This is another comment.
# Again! A comment in Python.
# The number of comments allowed in Python is unlimited. Just put a # sign at the start and you # are good to go.
```

# CHAPTER 3: VARIABLE TYPES

Now that we have gone through the basic syntax of Python, let’s talk about variables and their types. A variable can be defined as something that holds some kind of data. That data could be a number, a name, an address, or something else. Different variable types are available in Python for the developers to use.

Unlike other programming languages, in Python you don’t need to declare a variable before assigning a value to it. When a value is assigned to a variable, Python’s interpreter assumes that this is a new variable, and its type is also chosen based on the type of value assigned to it. Some common datatypes are integers, decimals, strings, floating point values, etc.

**Example:**

```
ID     = 1987         # A variable "ID" is holding an integer value of 1987.
Salary = 1000.00       # A Variable called Salary that contains a value of type Float
Name   = "Bruce Wayne" # A string typed variable called "Name"


print (ID)
print (Salary)
print (Name)
```

***Output:***

```
1987
1000.0
Bruce Wayne
Press any key to continue . . .
```

You can assign a value to multiple variables in Python. A single print statement can be used to display all of them in the output screen. In the example below, the value 3 has been assigned to three different variables, and then the value is being printed using a single print statement.

**Example:**

```
variable_1 = variable_2 = variable_3 = 3
print (variable_1,variable_2,variable_3)
```

***Output:***

```
3 3 3
Press any key to continue . . .
```

There are five standard data types available in Python. These types are numbers, strings, list, dictionary, and tuple.

The number typed variables can be integers, long, float, and complex numbers.

There is no difference in how these variables are declared. The examples above show how you can define a number-based variable in Python.

Strings in Python are similar to strings in any other programming language. String typed variables hold a single or set of words. A string can hold sentences too.

**Example:**

```
stringVariable = ('Say my Name!!!')

print (stringVariable)
print (stringVariable [2:5]) # The statement prints the characters present at position 2 and 5
print (stringVariable [0]) # This line of code will only print out the 1st character present # # # # in the
variable
print (stringVariable [2:]) # Using this line you can print the # value from the 3rd character
print (stringVariable + "TEST") # The value of the stringVariable # will be joined with the value "TEST"
print (stringVariable * 2) # This will print the value inside the #stringVariable twice
```

***Output:***

```
Say my Name!!!
y m
S
y my Name!!!
Say my Name!!!TEST
Say my Name!!!Say my Name!!!
Press any key to continue . . .
```

List is another datatype. List is a variable type that can hold values of different data types. For example, you can have multiple values inside a list, and each value could be either integer or string, etc.

**Example:**

```
firstListVariable = [ 'mnop', 123 , 8.09, 'bruce', 60.0 ]
secondListVariable = [0.007, 'wayne']

print (firstListVariable) # This statement will print out the entire list.
print (firstListVariable[0]) # This print statement will only print out the first value present in the list
print (firstListVariable[1:3]) #This print statement will show the values presnt in the 2nd and 3rd index.
print (firstListVariable[2:]) # This print statement will print out values from the second index and so on.
print (secondListVariable * 3) # This print statement will print out the list three times.
print (firstListVariable + secondListVariable) # This is how you can join the values of two lists and print   #
them.
```

***Output:***

```
['mnop', 123, 8.09, 'bruce', 60.0]
mnop
[123, 8.09]
[8.09, 'bruce', 60.0]
[0.007, 'wayne', 0.007, 'wayne', 0.007, 'wayne']
```

```
['mnop', 123, 8.09, 'bruce', 60.0, 0.007, 'wayne']
Press any key to continue . . .
```

Python also contains basic operators of different types. You can do pretty much all the arithmetic, logical, comparison, assignment, bitwise, and many more kinds of operations in Python. The use of these operators is very similar to how they are used in mathematics, so we don’t need to go into those details. Summation, subtraction, multiplication, and division work just like they do in actual arithmetic operations. You can do comparisons with two digits, strings, or variables for that matter in Python too. We have seen the assignment operator in the examples above, where I used an equal sign (=) to assign some value to a variable. You could also benefit from operations like shifting bits and taking compliments.

# Chapter 4: Decision Makers in Python

Programming languages need decision makers. All the languages, whether a high-level language such as Python or something low level like C, has decision makers. To define decision makers in simpler terms, you can think of them as locks that open when a certain key is used to open them. Conditions are the key thing in this scenario. If the right key is used, the condition satisfies, and hence you are allowed to open the lock; on the other hand, if you use an incorrect key, the lock will not open, as the condition didn’t satisfy. In programing, conditional operators work very similarly to the analogy of locks and keys. If a condition or set of conditions inside a block are true, then some corresponding programming statements are executed. If that condition is false, then a certain set of programming statements present inside the conditional blocks are ignored.

There are three types of decision makers available in Python. An `If statement` , `If-Else statement` and a `nested If or If-Else statement` . If you want to learn about each conditional statement and its use along with examples, you should carefully read this chapter. Let’s start with the simplest decision maker of all: the `if` statement. The `if` statement is very simple. If a condition is satisfied, a certain set of programming statements get executed by the interpreter. On the contrary, `if` the condition is false, then the programming statements inside the conditional `if` block are ignored.

## Example:

```
Variable_One = 100
if Variable_One:
   print ("The value of Variable_One is 100")
   print (Variable_One)

Variable_Two = 0
if Variable_Two:
   print ("The value of Variable_Two is 100")
   print (Variable_Two)
print ("Bye bye!")
```

***Output:***

```
The value of Variable_One is 100
100
Bye bye!
Press any key to continue . . .
```

The example above contains two `if` statements; each if statement is dealing with a different variable, and both these variables hold a different value. If the first `If` statement had returned false, the output would have been different, but in the example above, both the conditions are returning true. The reason behind such behavior is that we aren't actually using a condition in the if block , we are just passing it a value, and it will consider the value as true. The example below contains some conditions that are being applied on the variables. Let's check them out. In the following example, we are checking whether the value of Variable_One is smaller than 101 or not. The second if statement checks whether the value in Variable_Two is equal to zero or not. Both the conditions are true, so as a result, the print statements in each conditional block will be executed.

## Example:

```
Variable_One = 100
if Variable_One < 101:
   print ("The value of Variable_One is 100")
   print (Variable_One)


Variable_Two = 0
if Variable_Two == 0:
   print ("The value of Variable_Two is 0")
   print (Variable_Two)
print ("Bye bye!")
```

***Output:***

```
The value of Variable_One is 100
100
The value of Variable_Two is 0
0
Bye bye!
Press any key to continue . . .
```

In the following code example, none of the conditions mentioned inside the if statements are true, so as a result, only the print statement outside both conditional blocks will execute.

## Example:

```
Variable_One = 100
if Variable_One == 101:
   print ("The value of Variable_One is 100")
   print (Variable_One)


Variable_Two = 0
if Variable_Two > 0:
   print ("The value of Variable_Two is 0")
   print (Variable_Two)
print ("Bye bye!")
```

***Output:***

```
Bye bye!
```

```
Press any key to continue . . .
```

I hope that the examples shown above have taught you everything that you should know about the If conditional statement. Let’s now talk about the `If-Else` statement. In the `If-Else` block, if the condition isn’t true, then whatever code is present in the else block gets executed. The example below is checking whether the value in `Variable_One` is equal to `101` . The answer is no, so the value of `Variable_One` is not equal to `101` . So, the if block will be skipped, and the IDE will go to the else block, and it will execute the print statement inside the `else` block.

## Example:

```
Variable_One = 100
if Variable_One == 101:
   print ("The value of Variable_One is 100")
   print (Variable_One)
else:
    print("Hello from the else block")
```

***Output:***

```
Hello from the else block
Press any key to continue . . .
```

With a slight change in condition, the example below will skip the code in the else block and will only execute the print statements present in the If block.

## Example:

```
Variable_One = 100
if Variable_One == 100:
   print ("The value of Variable_One is 100")
   print (Variable_One)
else:
    print("Hello from the else block")
```

***Output:***

```
The value of Variable_One is 100
100
Press any key to continue . . .
```

The final type of conditional blocks available in Python are the nested decision makers. The nested decision makers are just blocks of code that have conditional blocks inside another conditional block. For instance, in the example mentioned below, there are two conditions; the first condition (parent condition) is comparing the value of `Variable_One` with `100` . The result of that condition is obviously true, so the cursor moves on to the second if statement (child condition) which checks whether the `Varible_One’s` value is greater than seventy or not. As both conditions are true, the print statement from both if blocks will be executed.

**Example:**

```
Variable_One = 100
if Variable_One == 100:
    if Variable_One > 70:
        print ("Hello from the nest conditional block")
    print (Variable_One)
else:
    print("Hello from the else block")
```

***Output:***

```
Hello from the nest conditional block
100
Press any key to continue . . .
```

But what if only the parent condition is true? Well in that case, the programming statements present in only the parent condition’s block will be executed, and everything else will be ignored.

**Example:**

```
Variable_One = 100
if Variable_One == 100:
    if Variable_One < 70:
        print ("Hello from the nest conditional block")
    print (Variable_One)
else:
    print("Hello from the else block")
```

***Output:***

```
100
Press any key to continue . . .
```

# Chapter 5: Loops in Python

There are three kinds of loops present in Python: `for` loop, `while` loop, and `nested` loop. The kinds of loops present in Python are no different to the kinds available in other programming languages. The concepts behind the working of each loop is also similar. The only difference is the syntax, and that is because of the difference of syntaxes present between various programming languages. Python doesn't provide support for a do-while loop by convention, but you can write a loop yourself that can work just like a `do-while` loop. `For` loop will be discussed first. It is a very commonly used loop. The basic Idea of a `for` loop is that it keeps executing the programming statements present in its block until the condition being checked turns false. The condition is checked at the start of each iteration of the loop.

**Example:**

```
for character in 'Python for Beginners':
    print ('Current character in the iteration :', character)
```

***Output:***

```
Current character in the iteration : P
Current character in the iteration : y
Current character in the iteration : t
Current character in the iteration : h
Current character in the iteration : o
Current character in the iteration : n
Current character in the iteration :
Current character in the iteration : f
Current character in the iteration : o
Current character in the iteration : r
Current character in the iteration :
Current character in the iteration : B
Current character in the iteration : e
Current character in the iteration : g
Current character in the iteration : i
Current character in the iteration : n
Current character in the iteration : n
Current character in the iteration : e
Current character in the iteration : r
Current character in the iteration : s
Press any key to continue . . .
```

I love Python for its simplicity and minimalistic design, to be honest. The amount of work that I did, using just two statements of Python in the example above, is awesome. If I had to replicate the above-mentioned output using some other high-level programming language, it would have been a little bit lengthier.

The example above is very simple; it's taking a string `Python for Beginners` and it's counting its characters. In each iteration, the loop takes one character and prints it out. Number of characters in the string: `Python for Beginners` is equal to 20. 20 is also the number of times the loops will execute. For loop is also used to traverse elements of a list or any other data structure available in Python.

The second type of loop that is available in Python is the while loop. There is no conceptual difference in how a `while` and a `for` loop work. It keeps on iterating until the condition specified to it is true. The loop shown below will keep on executing until the `counter` reaches 25. By the end of each iteration, a value will be printed on the screen. This value happens to be a counter. Before exiting the iteration, one will be added to whatever value is stored in the `counter` variable. When the `counter` will not be less than 25, the condition inside while loop will turn false which will terminate the loop execution. See the following example:

## Example:

```
counter = 0
while (counter < 25):
   print ('The value inside the counter is: ', counter)
   counter = counter + 1
```

***Output:***

```
The value inside the counter is:  0
The value inside the counter is:  1
The value inside the counter is:  2
The value inside the counter is:  3
The value inside the counter is:  4
The value inside the counter is:  5
The value inside the counter is:  6
The value inside the counter is:  7
The value inside the counter is:  8
The value inside the counter is:  9
The value inside the counter is:  10
The value inside the counter is:  11
The value inside the counter is:  12
The value inside the counter is:  13
The value inside the counter is:  14
The value inside the counter is:  15
The value inside the counter is:  16
The value inside the counter is:  17
The value inside the counter is:  18
The value inside the counter is:  19
The value inside the counter is:  20
The value inside the counter is:  21
The value inside the counter is:  22
The value inside the counter is:  23
The value inside the counter is:  24
```

```
Press any key to continue . . .
```

You could also decrement the `counter` instead of incrementing it. I will make some changes in the above example, and it will print values from `25 to 1` .

**Example:**

```
counter = 25
while (counter > 0):
  print ('The value inside the counter is: ', counter)
  counter = counter - 1
```

***Output:***

```
The value inside the counter is:  25
The value inside the counter is:  24
The value inside the counter is:  23
The value inside the counter is:  22
The value inside the counter is:  21
The value inside the counter is:  20
The value inside the counter is:  19
The value inside the counter is:  18
The value inside the counter is:  17
The value inside the counter is:  16
The value inside the counter is:  15
The value inside the counter is:  14
The value inside the counter is:  13
The value inside the counter is:  12
The value inside the counter is:  11
The value inside the counter is:  10
The value inside the counter is:  9
The value inside the counter is:  8
The value inside the counter is:  7
The value inside the counter is:  6
The value inside the counter is:  5
The value inside the counter is:  4
The value inside the counter is:  3
The value inside the counter is:  2
The value inside the counter is:  1
Press any key to continue . . .
```

So, with minor changes, I was successful in printing the numbers in reverse. These kinds of tweaks are very good practice for you as a beginner, as they give you an insight into how these concepts work. I would encourage you to go ahead and play with the code above. Try printing out something else with the help of the loops above.

Just like nested decision makers, we have nested loops. A `nested` loop is basically a loop inside another loop. For example, a `while` loop inside a `for` loop is an example of a nested loop. A `for` loop inside a `while` loop is also an example of a nested loop. In the following examples, I will show you how both the aforementioned cases could be written as loops in Python. The first example contains two `while` loops. The end product of the code below is a list of prime numbers that are between one and one hundred.

**Example:**

```
number_One = 2
while(number_One < 100):
   number_Two = 2
   while(number_Two <= (number_One/number_Two)):
      if not(number_One%number_Two): break
      number_Two = number_Two + 1
   if (number_Two > number_One/number_Two) : print (number_One, " is prime number between one and
hundred!")
   number_One = number_One + 1
```

***Output:***

```
2  is prime number between one and hundred!
3  is prime number between one and hundred!
5  is prime number between one and hundred!
7  is prime number between one and hundred!
11  is prime number between one and hundred!
13  is prime number between one and hundred!
17  is prime number between one and hundred!
19  is prime number between one and hundred!
23  is prime number between one and hundred!
29  is prime number between one and hundred!
31  is prime number between one and hundred!
37  is prime number between one and hundred!
41  is prime number between one and hundred!
43  is prime number between one and hundred!
47  is prime number between one and hundred!
53  is prime number between one and hundred!
59  is prime number between one and hundred!
61  is prime number between one and hundred!
67  is prime number between one and hundred!
71  is prime number between one and hundred!
73  is prime number between one and hundred!
79  is prime number between one and hundred!
83  is prime number between one and hundred!
89  is prime number between one and hundred!
97  is prime number between one and hundred!
Press any key to continue . . .
```

In simpler terms, you can say that the first `while` loop checks whether the number passed to it is a prime or not. The inner `while` loop makes sure that if the number is not a prime and it returns a remainder, then the loop should skip that iteration after incrementing the value of variable: `number_One` . The increment statements are available in both loops so that no matter what condition satisfies, each iteration gets to add one in the value of one. There are some keywords that you are seeing for the first time. I will explain each keyword used in the loops with a certain set of examples. For now, let's see another example in which the outer loop is `while` and the inner loop is `for` . The `for` loop will go through the list of four numbers and will print them one by one in each iteration. After the inner loop ends iterating, an iteration of `while` loop will also end. At the end of each iteration of `while` loop, an iteration number will be displayed on the output stream. After that, the next iteration of the `while` loop will start, and the `for` loop will run again and will print all the numbers from one to four. This process will continue until the value inside the variable name `number_One` is either equal to or greater than the number five.

## Example:

```
number_One = 1
while (number_One < 5):
    for x in [1,2,3,4]:
        print ("The value of x is:", x)
    print ("This is the iteration number",number_One ,"of the While Loop!")
    number_One = number_One + 1
```

***Output:***

```
The value of x is: 1
The value of x is: 2
The value of x is: 3
The value of x is: 4
This is the iteration number 1 of the While Loop!
The value of x is: 1
The value of x is: 2
The value of x is: 3
The value of x is: 4
This is the iteration number 2 of the While Loop!
The value of x is: 1
The value of x is: 2
The value of x is: 3
The value of x is: 4
This is the iteration number 3 of the While Loop!
The value of x is: 1
The value of x is: 2
The value of x is: 3
The value of x is: 4
This is the iteration number 4 of the While Loop!
Press any key to continue . . .
```

There are three keywords related to loops, which you should know about. They are `break` , `continue` and `pass` . As the name suggests, the `break` keyword breaks the loop in which it is used. Now, breaking a loop means that whenever the IDE hits a `break` statement, it skips everything below it in that code block. In case of a loop, the break statement will make the code skip everything written in the code block of a loop and will exit it. But when `break` is used in anything else, it will skip anything that comes after it and will take the cursor at the end the block. Let’s see an example, which will clarify things.

**Example:**

```
for character in 'Python for Beginners':
    if character == "n":
        break
    print ('Current character in the iteration :', character)
```

***Output:***

```
Current character in the iteration : P
Current character in the iteration : y
Current character in the iteration : t
Current character in the iteration : h
Current character in the iteration : o
Press any key to continue . . .
```

When the loop got to the character `n` , it hit the break statement, and after that, the loop was over. We could get a similar output with a while loop. The example below shows how the loop was interrupted when the value of the `variable` became five.

**Example:**

```
variable = 1
while variable > 0:
   print (“Value is:”, variable)
   variable = variable + 1
   if variable == 5:
      break
```

***Output:***

```
Value is: 1
Value is: 2
Value is: 3
Value is: 4
Press any key to continue . . .
```

The `continue` statement is used to skip an iteration in a loop. It skips whatever code comes after it and takes the cursor at the starting point of the loop. I will use the examples above and will show you the difference after making a minor change. The code below will skip the iteration when the value is 5. After skipping value:

five, the rest of the numbers will be displayed on the screen.

### Example:

```
variable = 11
while variable > 0:


  variable = variable - 1
  if variable == 5:
    continue
  print ('Value is :', variable)
```

***Output:***

```
Value is : 10
Value is : 9
Value is: 8
Value is: 7
Value is: 6
Value is: 4
Value is: 3
Value is: 2
Value is: 1
Value is: 0
Press any key to continue . . .
```

Now in the following example, I am using a `for` loop and a `continue` keyword. Whenever the loop will encounter the letter `n` , it will skip the iteration. The output of the following code doesn’t contain the letter `n`.

### Example:

```
for character in 'Python for Beginners':
    if character == "n":
        continue
    print ('Current character in the iteration :', character)
```

***Output:***

```
Current character in the iteration : P
Current character in the iteration : y
Current character in the iteration : t
Current character in the iteration : h
Current character in the iteration : o
Current character in the iteration :
Current character in the iteration : f
Current character in the iteration : o
Current character in the iteration : r
Current character in the iteration :
Current character in the iteration : B
Current character in the iteration : e
Current character in the iteration : g
Current character in the iteration : i
Current character in the iteration : e
Current character in the iteration : r
Current character in the iteration : s
Press any key to continue . . .
```

You might encounter a scenario when you have to write a programming statement just to support the syntax without changing any values or calling any functions. Python supports a keyword called `pass` , designed specifically for something like this. It is more like a null operation, and it doesn't mean anything. It doesn't change any value or affect any condition. It is used just to support the syntax. Use `pass` like this:

## Example:

```
for character in 'Python for Beginners':
    if character == "n":
        pass
        print("pass statement has been hit!!")
    print ('Current character in the iteration :', character)
```

***Output:***

```
Current character in the iteration : P
Current character in the iteration : y
Current character in the iteration : t
Current character in the iteration : h
Current character in the iteration : o
pass statement has been hit!!
Current character in the iteration : n
Current character in the iteration :
Current character in the iteration : f
Current character in the iteration : o
Current character in the iteration : r
Current character in the iteration :
Current character in the iteration : B
Current character in the iteration : e
Current character in the iteration : g
Current character in the iteration : i
pass statement has been hit!!
Current character in the iteration : n
pass statement has been hit!!
Current character in the iteration : n
Current character in the iteration : e
Current character in the iteration : r
Current character in the iteration : s
Press any key to continue . . .
```

As you can see in the example mentioned above, when loop hits the character `n` , the `pass` block gets executed. Infinite loop is also a concept that you should know before I conclude this chapter. An `infinite` loop is a `while` loop that never terminates and keeps on executing.

**Example:**

```
variable = 1
while variable == 1:
   print ("Infinte loop")
```

**Output:**

```
Infinte loop
Infinte loop
Infinte loop
Infinte loop
Infinte loop
Infinte loop
Infinte loop
Infinte loop
And so on. . . . .  (This is just a very small part of the output)
```

You can use `else` keyword with `for` or `while` loops too. The `else` keyword works differently for a `while` and `for` loop. For `while` loop, when the condition turns false, a signal is sent to the `else` block. That signal or instructions execute the `else` block

**Example:**

```
variable = 11
while variable > 0:


  variable = variable - 1
  if variable == 5:
     continue
  print ('Current variable value :', variable)
else:
   print("The While loop has stopped iterating")
```

**Output:**

```
Current variable value : 10
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Current variable value : 4
Current variable value : 3
Current variable value : 2
Current variable value : 1
Current variable value : 0
The While loop has stopped iterating
Press any key to continue . . .
```

In the `for` loop, the `else` block executes when the `for` loop finishes processing and executing all the programing statements in it.

## Example:

```
for character in 'Python for Beginners':
    if character == "n":
        pass
        print("pass statement has been hit!!")
    #print ('Current character in the iteration :', character
else:
    print("For loop has stopped the iteration process!!")
```

***Output:***

```
pass statement has been hit!!
pass statement has been hit!!
pass statement has been hit!!
For loop has stopped the iteration process!!
Press any key to continue . . .
```

# Chapter 6: Numbers in Python

So far, we only looked at the generic definition of data types; in this chapter and subsequent chapters, I will teach you details about the main data types present in Python. Numbers represent anything that is either a digit or a floating point. Python also supports complex numbers. Specifically, `integers` , `floating points` , `long` and `complex` are the kind of data types that fall under the umbrella of `number` class.

`Integers` are just numbers like `1,2,3,4` etc. `Long` data type also contains integers, but the range of numbers it supports is wider as compared to simple `integer` data type. `Floating point` number is any number that has a decimal point in it. For example, `1.1, 4.5, 8.9, 23.4589` etc. belong to the `floating points` . `Floating points` are also called `floats` . Although `complex` numbers are seldom used in Python programming, the support for them in Python is present. A `complex` number is a number which has an imaginary number I (Square root of -1) with a real co- efficient.

Several built-in conversion mechanisms are available in Python. You can use these methods to convert various types of inputs into your desired data type. The table below is explaining the meaning of each method.

| **Method** | **Meaning** |
|---|---|
| int (variable) | This method will convert the variable’s datatype to an `integer` |
| long (variable) | This method will convert the variable’s datatype to `long` . |
| complex (variable) | When you pass a variable to this function, it will change its datatype to complex. Since there is only one value being passed to the function, the variable will become the real part, and the imaginary part will be zero. |
| complex (variable_One, variable_Two) | In this case, `variable_One` will become the real part, and `variable_Two` will become the imaginary part of the `complex` number. |

Many other mathematical functions are available in Python too. I will not talk about all of them, but I will discuss a few. You can google the rest of them on your own.

| **Method** | **Meaning** |
|---|---|
| sqrt (variable) | This method will perform a square root operation on the variable. |
| pow (variable_One, Variable_Two) | variable_One will act as the base, and variable_Two will act as the exponent. The method is used to take a power of the base value passed to it. |
| ciel (variable) | This function works exactly how ceil works in mathematics |
| floor (variable) | Use this method to take the floor of a given input. |
| log (variable) | You can take log of a variable using this built-in method. |
| max (var1, var2, var3,…..) | This is how you can calculate the maximum number present among a series of input variables or values. |
| min (var1, var2, var3,…..) | This is how you can calculate the minimum number present among a series of input variables or values. |

A number of methods that perform trigonometric operations are also present in Python. Here’s a list:

- acos
- asin
- atan
- atan2
- hypot
- cos
- sin
- tan
- degrees
- radian

Mathematical constants like π (pi) and e are also available in Python.

# Chapter 7: Strings in Python

You have seen the use of strings in the previous chapters; in this chapter, we will see everything important related to strings. As defined earlier, a string is basically a sequence of characters. For example, the most common and trivial string example in Python is "Hello World ". Another example is: "This is a string. "

Sub string is a portion of another string. In Python, you can access a substring from a string like this:

## Example:

```
string1 = 'Hello World!'
string2 = "This book teaches Python programming to beginners"


print ("string1 [1]: ", string1 [1] )
print ("string2 [1 : 15]: ", string2 [1 : 15] )
```

***Output:***

```
string1 [1]:  e
string2 [1 : 15]:  his book teach
Press any key to continue . . .
```

In the above-mentioned example, there are two print statements and two variables that contain different strings. Each print statement will print different portions of a string. You must be wondering, why does the print statement: print ("string1 [1]: ", string1 [1] ), say string1 [1]? It means that you should take the value present in the variable named string1 and get the character present at the index number 1 . The result is showing the character "e " as an output because, in Python, the index starts from zero (0) . So, index when 1 will mean the second character. The second print statement is printing everything from index number 1 to index number 15 in the variable named string2 .

A number of escape characters are also present in Python. They are listed below:

- \cx
- \C-x
- \e
- \a
- \b
- \c
- \f
- \t
- \v
- \x

- \r
- \s
- \xnn
- \M-\C-x
- \n
- \nnn

You should look into the definition of each escape character. Try to use each one in your string and see its result for a better understating of the said concept.

Several string related operations are also available in Python. You can slice your desired sub string from a main string using the slice method. The + sign is used to concatenate/join multiple strings. The * sign can be used to repeat the string n times, where n will be a number. A string can also be formatted using % sign. These are the most basic operations related to strings. In order to get more detail, kindly look on the internet. The following example is a very simple demonstration of how a string can be formatted.

**Example:**

```
print ("I am %s and I weigh %d kg!" % ('John', 13))
```

***Output:***

```
I am John and I weigh 13 kg!
Press any key to continue . . .
```

List of escape characters available in Python:

- %c
- %s
- %i
- %d
- %u
- %o
- %x
- %X
- %e
- %E
- %f
- %g
- %G

You might have noticed that I always use double quotation marks on both ends of a string typed value. This is part of the syntax, and this tells the Python interpreter that the value being processed is a `string` . Moreover, there is a use for `triple` quotation marks as well. With the help of triple quotation marks, your string value can span over multiple lines.

## Example:

```
print ("""This is an example of triple qoutation marks,
you can see that the print statement spans more than one line.""")
```

***Output:***

```
This is an example of triple qoutation marks,
you can see that the print statement spans more than one line.
Press any key to continue . . .
```

Just like the `numbers` , strings also have a lot of built-in functions in Python. I am listing a few of them here, but it’s an assignment for you guys to go out and explore the meaning and use of each function.

- capitalize
- center
- count
- expandtabs
- find
- index
- decode
- encode
- endswith
- isalnum
- isalpha
- isspace
- istitle
- isupper
- join
- len
- isdigit
- islower
- isnumeric
- ljust

# Chapter 8: Lists and Tuples

List is another commonly used data type in Python. Lists are nothing but a collection of similar or different types of values. A list has the power to hold values that belong to the data types of integers , strings , long etc. The example below is printing the values present inside a list .

## Example:

```
list =[11, 22, 33, 44, 55, 66, 77, 88, 99, 1010, 1111, 1212, 1313, 1414, 1515, 1616, 1717, 1818, 1919,
2020]
for number in list:
    print (number)
```

**Output:**

```
11
22
33
44
55
66
77
88
99
1010
1111
1212
1313
1414
1515
1616
1717
1818
1919
2020
Press any key to continue . . .
```

## Example:

```
list =["T","h","i","s","","b","o","o","k","","t","e","a","c","h"
       ,"e","s","","P","y","t","h","o","n"]
for character in list:
    print (character)
```

**Output:**

```
T
h
i
s

b
```

```
o
o
k

t
e
a
c
h
e
s

P
y
t
h
o
n
Press any key to continue . . .
```

Now that I have shown you how to print out items present in a `list` , refer to the subsequent example to know how you can perform an `update` or `delete` operation on an element inside a list. The first example is for updating an element. I will pass an index number and the value that I want to replace with the original value. At index number `4` , I used special characters to create gibberish. Then, I replaced that value.

### Example:

```
list =["T","h","i","s","_ & * % $ # @ !","b","o","o","k","","t","e","a","c","h"
      ,"e","s","","P","y","t","h","o","n"]
#for character in list:
print (list[4])
list[4]="This is the updated list's value"
print (list[4])
```

***Output:***

```
_ & * % $ # @ !
This is the updated list's value
Press any key to continue . . .
```

You can `delete` the value present inside an index too; here's how:

### Example:

```
list =["This","_ & * % $ # @ !","book","teaches","Python"]
#
for character in list:
    print (character)
del list[1]
```

```
print("")
for character in list:
    print (character)
```

***Output:***

```
This
_ & * % $ # @ !
book
teaches
Python

This
book
teaches
Python
Press any key to continue . . .
```

Other operations that you can perform on a list are list concatenation, finding the length of a list, repeating the items of a list with respect to some number, checking if an element is a member of a list or not, and iteration. Have a look at the examples shown below. Each example demonstrates an aforementioned operation respectively.

### Example: Multiplying list’s members to some number.

```
list = ["Hello! "]
print (list[0] *5)
```

***Output:***

```
Hello! Hello! Hello! Hello! Hello!
Press any key to continue . . .
```

### Example: List concatenation.

```
list = ["Hello "]
list2 =["Mate!"]
print (list[0]+list2[0])
```

***Output:***

```
Hello Mate!
Press any key to continue . . .
```

### Example: Checking whether an element is part of a list or not.

```
list = ["Hello "]
print ("Hello " in list[0])
```

***Output:***

```
True
Press any key to continue . . .
```

### Example: Calculating length of the value, present at some index of list.

```
list = ["Hello "]
print (len (list[0]))
```

***Output:***
```
6
Press any key to continue . . .
```

Tuples are immutable lists. That means that we cannot update or delete any element inside a tuple. We can join them and perform other operations like tuple concatenation, finding the length of a tuple, repeating the items of a tuple with respect to some number, checking whether an element is part of a tuple or not and iterating a tuple. Everything is the same, and you can use the above examples to perform these operations on a tuple. The only syntactic difference between the list and the tuple is that, while declaring a list, we use square brackets [] to enclose its values. The tuple declaration involves the use of round bracket () for enclosing the elements. The following example shows how you can define a tuple. It's very simple.

## Example:
```
tuple = ("Hello! ")
print (tuple[0])
```
***Output:***
```
Hello!
Press any key to continue . . .
```

As I told you guys earlier, you can't perform update or delete operations on a tuple. I thought it would be a good example to show you what does happen when you try to do so. The first example is trying to delete an element from a tuple. I am handling the error using exception handling which is a very advanced concept, so you don't need to worry about it right now. The second example is also giving me an error stating that 'tuple' object doesn't support item Updation.

## Example:
```
tuple =("This","_ & * % $ # @ !","book","teaches","Python")
try:
   del tuple[1]
except:
   print("'tuple' object doesn't support item deletion")
```
***Output:***
```
'tuple' object doesn't support item deletion
Press any key to continue . . .
```
## Example:
```
tuple =("This","_ & * % $ # @ !","book","teaches","Python")
try:
   tuple[1]= ""
except:
   print("'tuple' object doesn't support item Updation")
```

***Output:***

```
'tuple' object doesn't support item Updation
Press any key to continue . . .
```

# Chapter 9: Dictionary

Dictionary is another data type available in Python. It is similar to List and tuple in the way that it can hold values of different types. In lists and tuples, indexes are used to keep track of values. If you want to access , update or delete a value in a list, you would use the index value along with the desired operation. Same goes with the tuple except the fact that there are no update and delete operations in tuples. In a dictionary, there are key value pairs. A value is a key, and the other is the value associated with that key. Keys work just like index numbers work. See the following example to understand how you can declare a dictionary .

**Example:**

```
dictionary = {'Name': 'John', 'Age': 27, 'Salary': '1200$'}
print ("Name:", dictionary['Name'])
print ("Age:", dictionary['Age'] )
print ("Salary:", dictionary['Salary'] )
```

***Output:***

```
Name: John
Age: 27
Salary: 1200$
Press any key to continue . . .
```

To update the value of a key , you can do something like this:

**Example:**

```
dictionary = {'Name': 'John', 'Age': 27, 'Salary': '1200$'}
print ("Name:", dictionary['Name'])
print ("Age:", dictionary['Age'] )
print ("Salary:", dictionary['Salary'] )
print("\n I am going to update the age of John to 72\n")
dictionary['Age'] = 72
print ("Updated Age:", dictionary['Age'] )
```

***Output:***

```
Name: John
Age: 27
Salary: 1200$

 I am going to update the age of John to 72

Updated Age: 72
Press any key to continue . . .
```

I used the key to first access the values, and then I used the key to update the value against it. If you want to add a new key value pair in your dictionary, you can do something like this:

## Example:

```
dictionary = {'Name': 'John', 'Age': 27, 'Salary': '1200$'}
print ("Name:", dictionary['Name'])
print ("Age:", dictionary['Age'] )
print ("Salary:", dictionary['Salary'] )
print("\n I am going to add a key value pair. The new Dictionary is: \n")
dictionary['Country'] = "United States of America (USA)"
for key,value in dictionary.items():
    print (key,":", value)
```

***Output:***

```
Name: John
Age: 27
Salary: 1200$

I am going to add a key value pair. The new Dictionary is:

Country : United States of America (USA)
Name : John
Age : 27
Salary : 1200$
Press any key to continue . . .
```

To delete a `key value` pair, you can do something similar to the following example:

## Example:

```
dictionary = {'Name': 'John', 'Age': 27, 'Salary': '1200$'}
print ("Name:", dictionary['Name'])
print ("Age:", dictionary['Age'] )
print ("Salary:", dictionary['Salary'] )
print("\nI am going to delete a key value pair. The updated Dictionary is: \n")
del dictionary['Salary']
for key,value in dictionary.items():
    print (key,":", value)
```

***Output:***

```
Name: John
Age: 27
Salary: 1200$

I am going to delete a key value pair. The updated Dictionary is:

Age : 27
Name : John
Press any key to continue . . .
```

Numerous built-in functions are also available for dictionaries. You should go ahead and check them out.

# Chapter 10: Functions

Basically, functions are a block of code. This block of code is comprised of a single or multiple programming statements. Functions make your code re-usable. To calculate the sum of a pair of numbers at five different occasions, you can write a function that could do that and can call that function any time you want. Otherwise, you could rewrite the code repeatedly to get the job done. The later approach is inefficient, dull, and not desirable at all. Functions also allow you to create different modules of a program. Think of a module as a part of an engine, which has to do its job to make the engine work. Code which has been divided into modules is more maintainable. That code is also easier to understand for anyone who uses that code. Let’s start with defining a function, and then we will do something more with it.

**Example:**

```
def sum(num1,num2):
  sum = num1 + num2
  print ("The sum of two numbers passed is:",sum)
  return
```

I have declared a function named sum . This function or method takes two numbers as inputs. Then, the sum is stored in the variable called sum . After that, this value inside the sum variable is displayed, and then the function call ends.

Now the question is what is a function call ? A function call is something that is used to call a function. For example, if I suddenly come up with this need to sum two numbers, now I know that I have a function that does exactly that, and I can call it. In order to understand how calling a function works, refer to the example shown below:

**Example:**

```
def sum(num1,num2):
  sum = num1 + num2
  print ("The sum of two numbers passed is:",sum)
  return


sum (1,2)
sum (3,4)
sum (5,6)
```

***Output:***

```
The sum of two numbers passed is: 3
The sum of two numbers passed is: 7
The sum of two numbers passed is: 11
Press any key to continue . . .
```

The statements in red are the function calls. The numbers passed to the statement call are called arguments . If you want to pass more arguments, you have to change the definition of the function to accept more arguments. There are four types of function arguments:

- Default
- Required
- Variable length
- Keyword

The required arguments are the kinds of arguments that you have to pass whilst calling a function. The order in which the arguments are being passed should match to the order of parameters allowed by the function's definition. An error occurs when you don't pass any argument whilst calling the function in the successive example:

## Example:

```
def sum(num1,num2):
  sum = num1 + num2
  print ("The sum of two numbers passed is:",sum)
  return
sum()
```

***Output:***

```
Type Error was unhandled by user code
Message: sum () missing 2 required positional arguments: 'num1' and 'num2'
```

Keyword arguments are the kind of function's arguments that are passed in a function's call by mentioning the name of the parameter in the function's definition.

## Example:

```
def sum (num1,num2):
  sum = num1 + num2
  print ("The sum of two numbers passed is:",sum)
  return
sum (num1 = 2, num2 = 3)
```

***Output:***

```
The sum of two numbers passed is: 5
Press any key to continue . . .
```

When the value of an argument isn't passed whilst its function's call, its default argument comes into play. Look at the code below; I am not passing the value to num2 in the highlighted print statement.

## Example:

```
def sum (num1, num2 = 1):
  sum = num1 + num2
  print ("The sum of two numbers passed is:",sum)
  return
sum (num1 = 2, num2 = 3)
sum (num1 = 3)
```

***Output:***

```
The sum of two numbers passed is: 5
The sum of two numbers passed is: 4
Press any key to continue . . .
```

The functions which I defined here are called `user defined` functions. A bunch of built-in functions are also available in Python. `Print` is one of those pre-built functions.

You saw the use of the `return` keyword in the examples of this chapter, but I haven't explained its meaning. The return statement basically returns the control to the module that is not part of the function. You could also use a return statement to return a value or a set of values. In the future, when you will be creating applications much bigger than the ones being shown in this book, you will often feel the need to use a function to perform some task and at the end of the task, you would want to take the value and use it somewhere else. Well, this is what I am going to show you in the next example. I will be adding two numbers together, and the sum of those numbers will be used in a multiplication operation.

**Example:**

```
sumValue= 0
def sum(num1, num2 = 1):
  sumValue = num1 + num2
  print ("The sum of two numbers passed is:",sumValue)
  return sumValue
sumValue = sum(num1 = 2, num2 = 3)


print ("The result of multiplying 5 with the sum of two numbers is:", sumValue * 5)
```

***Output:***

```
The sum of two numbers passed is: 5
The result of multiplying 5 with the sum of two numbers is: 25
Press any key to continue . . .
```

You can see that I created a variable named `sumValue` and assigned a value of `zero` to it. Then, I used the `sum` function, returned the result, and stored it in the `sumValue` . Finally, I used the `print` statement to show the result of multiplication on the screen.

A variable method has a scope. Its scope could be either `global` or `local` . The variable named `sumValue` is a global variable with regard to the function `sum` . The variables used as function’s parameters (`num1` and `num2` ) are `local` variables. The main difference is that a local variable like `num1` and `num2` will only exist if the function gets called. Otherwise, these variables will not be defined. But the `sumValue` variable will be created and will last until the program terminates.

# Conclusion

Congratulations are in order here. You guys have successfully completed the introduction part required for Python. If you have read the book and have tried examples and assignments that I suggested, then my friend, you are now ready to go to the next level of learning Python. There are many advanced topics that were not covered in this book. You have to go ahead and look into how Python is used as an object-oriented programming language. You need to also learn the object-oriented paradigm. Python is used in industry as a scripting language. Information security professionals often use it. Even web developers use Python for developing web applications. So, you should first learn all the advanced topics of Python, and then you should decide what field you want to pursue. I wish you best of luck for the future.