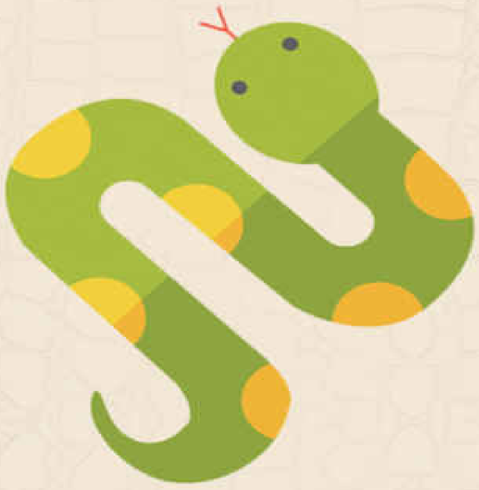

THE COMPLETE Python QUICKSTART GUIDE

FOR BEGINNER'S

python™

Python

The Complete Python Quickstart Guide For Beginner ' s

Table of Contents

Introduction

I want to thank you and congratulate you for downloading the book, “ *PYTHON: The Complete Python Quickstart Guide For Beginner ’s* ” .

This book contains proven steps and strategies on learning Python Programming quickly.

Unlike other programming language, Python emphasizes on the readability of codes. It uses a precise syntax that enables programmers, beginners and experts alike, to write fewer codes in coming up with the functions for a desired program. The best thing about Python is its universality: you can code an application using one platform but still be able to run it using other platforms. This is unlike other programming languages that promise platform independence but still manages to fall short of expectations.

The Complete Python Quickstart Guide for Beginners is all about providing you with all the necessary basics of Python. Everything that you must know in order to become productive in your line of work is in this book. Examples provided in this book are specific and relevant that you may encounter them in the ordinary course of performing your jobs.

A crash course on Python will only give you the framework of the language but will never make you appreciate the beauty of the language as much as *Python Programming for Beginners* will.

If you are still thinking twice about learning Python, perhaps what will convince you is a list of organizations using Python as part of their programming: (a) Alice Education Software, (b) Fermilab, (c) Go.com, (d) Google, (e) National Space and Aeronautics Administration, (f) New York Stock Exchange, (g) ObjectDomain, (h) Redhat, (i) Yahoo! and (j) Youtube.

This book will be your guide in anything and everything Python such that by the time you put down this book, you will be confident in performing tasks that require Python programming!

Thanks again for downloading this book. I hope you enjoy it!

Chapter 1. Understanding Python

There are many programming languages being used today, so much so that it would be near-impossible to learn about every single one. It's easy to think that programmers would be contented with all the languages available to communicate with the computer but the truth is, new languages are born every year.

Creating a new programming language isn't just out of whim. Programmers create a new one for a good reason. Each programming language specializes on specific functions. Also as computer hardware evolves, so does computer software and thus, codes and programming language. In order to keep up with this continuous evolution, programmers create a new language that will streamline the communication between hardware and software. For instance, in achieving a more appropriately-designed interface, some programmers build a different computer language after acknowledging the limitations of the ones that are available.

There are certain functions that Python excels in so it is important for you to know what these functions are. Part of getting acquainted with a programming language is learning about its strengths and weaknesses.

Reasons for Using Python

Programming languages are constructed by programmers for specific reasons. The goals they have in mind help shape the language and determine the flexibility and usability of the language. Python was created with the objective to create a language that can be used by programmers in creating codes that are productive and efficient.

Here are the reasons you will want to use Python when coding an application:

(a) Less Development Time

Compared to other similar programming language like Java and C/C++, codes used in Python are 5 to 10 times shorter. This makes Python less complicated but more efficient. It also means that you spend less time and effort in developing your application and more time toying with it.

(b) Less Learning Time

The main goal of the makers of Python is to create a language that has fewer unusual rules which make it hard for beginners to learn a language. This will help programmers create more usable applications that they can't do with more complicated and obscure language.

(c) Ease of reading

When checking for errors or bugs, it is important that the language is easier to read and understand. It is hard to code and check a program code, if language is too complicated. Python codes are easier to write and read than any other languages available today. This will make it easier for you to interpret the code and make necessary changes.

Also, here are suggested specific uses for Python:

(a) Scripting browser-based applications

(b) Creating better user interfaces

(c) Creating rough application examples

(d) Working with XML

(e) Designing scientific, mathematic and engineering applications

(f) Interacting with databases

Python versus Other Programming Language

Sometimes, comparing a programming language to other languages can be very subjective. The difference is usually a matter of personal preference but there are times that the comparison can be supported by quantifiable scientific data.

Before you read this section, it is important to note that there is no best all-encompassing language in the world. All you have to do is find a language that works best for a specific function or goal in mind.

(1) C#

There are claims that the people behind Microsoft simply replicated Java in creating C#. Be that as it may, C# has upsides and downsides compared to Java. According to Microsoft, their main goal in creating C# is to create a better version of C/C++. When pitted against C#, Python enjoys the following advantages:

(a) Better engineering and scientific applications

(b) More extendable when using C/C++ and Java

(c) Better multiplatform support

(d) More concise (and thus, smaller) code

(e) Easier to learn and understand

(f) Allows the usage of different computer environments.

(2) Java

Like Python, Java has been a one-stop shop for programmers. For many years, programmers have been in search for a language that can be written and run almost anywhere. Java is designed to perform well in various platform and computer environment. Despite this similarity, Python's most important advantages are:

(a) More concise (and thus, smaller) code

(b) Easier to learn and understand

(c) Improved variables or databoxes that can hold various types of data that can be used when running different applications

(d) Several times faster in development

(3) PERL

Otherwise known as Practical Extraction and Report Language, PERL is a programming language that is good at obtaining and processing data from a different database. Here are the reasons Python is better:

(a) Easier to read and implement

(b) Better Java integration

(c) Better data protection

(d) Easier to learn and understand

(e) Less platform-specific biases

Chapter 2. Interacting with Python

Python is built to provide users with several ways to improve interaction with the computer. Take the Integrated Development Environment (IDLE) for example. IDLE makes it easier for Python users to develop better applications. There are times, however, when you simply want to double-check an existing program for bugs or errors. If this is the case, it is recommended that you use the command-line version of Python. This version offers better user interface than other platforms because it requires less resources, fewer command-line switches and depends on minimal system requirements to run the code.

Starting Python

Depending on the platform you have installed in your work computer, there are multiple ways to open Python's command line. Here is a list of methods that you can use:

(a) Open a terminal or a command prompt. Simply key in Python , then press Enter. Programmers often use this option when they require better flexibility in configuring their Python environment.

(b) Locate the Python folder in your file manager and then select Python (command-line) option. This option makes you use a command-line version of Python configured in its default settings.

(c) If you are using Windows, you still have to install Python. Once installed, go to the Python folder. This is usually found in this address: C:\Python33 . Double-click the Python.exe filename. This method opens a command-line configured in its default settings. Using this method, however, gives you the option to open the command-line using credentials that have upgraded privileges. This allows applications that utilize secured resources to be run by system.

When running Python using a command prompt, you can use various options that increase its usability. Access these options by using the following syntax:

Python <options>

Replace <options> with any of the following case-sensitive options:

-b	This command includes red flags or warning signals to the output when the program being run includes restricted Python features
-c cmd	This option uses the data and information taken from the cmd when starting the application. This also tells Python when to stop processing other data options.

-d	Locates errors from the program by starting the debugger option
-h	Begins the program by displaying basic environment variables and helpful options in the screen.
-i	This opens a Python interface that allows you to inspect the code being run.
-m mod	This enables Python to run the library system specifically designated by the command mod as part of the script.
-q	This stops Python from printing sensitive information like version and copyright data in an interactive startup.

Typing a Command

After opening the command-line version of Python, the precautionary first step is to check whether the options you need to run your application checks out. Commands are coded instructions that the computer uses in carrying out specific tasks or evaluate ideas and information.

Unlike the IDLE which hides the less fancy details of Python programming, command-line interface gives you first-hand experience and understanding on how Python works.

Commands tell the computer what to do. They are a series of steps in a coded procedure. This is done by typing commands that Python is familiar with. In response, Python translates these commands into instructions that can be understood by the computer.

For instance, the command print() displays the result of the program on the screen.

To give you a quick example of how commands work, execute the following steps using your command-line interface:

1. Open the command-line version using any of the enumerated options in the previous section of this chapter.
2. At the command line, type the following:

```
print("This is an example of how command works.”)
```

3. Press Enter to signify the end of the command. The command-line will then display:

```
This is an example of how command works.
```

Exiting Python

If you want to close the command-line after several hours of playing with the programming language, there are two standard means for you to do this. There are other methods to close Python but it is recommended that you use the standard method to make sure that Python will behave normally the next time you use it. These methods are as follows:

quit() and exit()

These commands will shut down the current Python session. The good thing about said commands is that they can accept an optional instruction. For instance,you can use quit(5) or exit(5) to shut down Python when the ERRORLEVEL variable reaches a numeric value of 5.

To illustrate how to Exit Python properly, perform the following procedure:

1. Open a terminal or command prompt.
2. Key in Python then press enter.
3. Type quit().

Chapter 3. Coding Your First Application

Beginners often see application development as either some form of rocket science or magic that only computer nerds can understand. What they don't understand is that they, too, can harness the power of computer programming.

Building applications follows specific stages of development. It's more than just a series of interrelated steps but it's not too complicated for beginners like you to understand. After you finish this chapter, you will be able to develop simple applications that don ’ t require a magic wand.

As with other tasks, people use tools to build complicated applications. Fortunately, Python does not require tools to code applications. Using IDLE, however, makes it easier for Python programmers to simplify the coding process and process data and information faster.

Using IDLE

While you can use any text editor to code a Python application, IDLE provides a more efficient platform to write a Python code. The Interactive Development Environment is the most widely-used platform used by experienced programmers. IDLE can perform the following Python-related tasks:

(a) Write Python code,

(b) Save and run Python data and files,

(c) Perform simple editing like copy, cut and paste,

(d) Browse through and find specific Python files,

(e) Remove errors by performing debugging procedures, and

(f) Emphasize and recognize important codes and keywords from special text information.

Most of the above functions can be accomplished using a command-line interface but it takes time and effort. For instance, you can perform debugging procedures using command line interface but this is done in a manual and error-prone process. This makes using IDLE several times easier.

Creating the Application

Now that you are acquainted with the Python system, it is time to create your first Python application. When creating one, you can start by opening a new window to enter your commands. After typing the commands, you can save the file in a storage system that will serve as a database for running your application.

1. Opening a new window

The Python Shell window is an interactive window that provides immediate feedback on every command entered in the system. Python is a static environment that allows you to key in the commands and save them for future use.

In order to open a new window, click on:

Choose File ➪ New File

Using the Edit window is very similar to text editors. It also has access to basic commands such as Cut, Copy and Paste. Instead of executing the command, pressing Enter brings you to a different command line.

2. Typing the command

Using the operative commands in your arsenal, simply key in the commands to the space provided by the platform.

3. Saving the file

Before you are able to run the file, you are required to save the file. This is the system's way of freezing your code should you make an error in the process. This ensures the safety and the integrity of the code. In determining what went wrong, spotting errors, making necessary corrections, the usual first step is to save the file. This makes it easier for the system to run the application and spot possible error points.

When saving a file, follow this process:

Choose File ➪ Save

This opens the Save As dialog box. By default, the Edit window chooses the destination folder connected to the installation folder of the application. Since this folder houses the Python codes, experienced programmers create folders where they will save application codes.

In the File Name portion, fill it in withdesired name for your application. For instance, if you want to name it as: MyFirstProject , Python will save it as MyFirstProject.py .

Running the Application

In order to determine the success of the coded application, you must be able to run them. Python will attempt to translate the codes and execute several computer functions from the translation.

One of the upsides of Python is that it provides many methods to run an application. In order to run your application, click on:

Run ➩ Run Module

A new Python Shell window opens to display the output.

Chapter 4. Python Data Types

In order to process information efficiently, Python sorts input data into several categories. These categories serve as areas for data storage that can be accessed every time the Python platform is running. Also, this provides structure to the rules when modifying specific bits of information. Otherwise, the program just won't work.

Variables

Similar to algebra, variable contain specific values assigned to it. When working on writing more complicated application codes, it is necessary to store information inside variables.

In Python, variables are a kind of storage box that can be accessed anytime you wish to work or process on the information stored in it. For instance, if you require your users to input their surname, you can create a variable that stores the input that corresponds to their surname.

To make it easier for them to do their jobs, programmers use variables to make sure that information gathered from end users are safe. Any kind of information, ranging from dates and times, numbers and letters can be stored in variables.

In order to put information inside variables, you create a commandthat uses assignment operator = . For example, if you want to assign the number 3 in the variable defVar . Type defVar = 3 then press Enter.

Numeric Types

If in Mathematics, the numbers 5 and 5.0 are the same, Python views them differently. In order for Python to process the number, it understands that 5 and 5.0 undergo different data processing methods. Here are the kinds of Numeric types that you must be familiar with:

1. Integers

An integer is any whole number. For instance, 5 is a whole number but 5.0 isn't a whole number because of the decimal point. In this case, Python understands that 5 is an integer and can thus be characterized by the int data type.

Just like variables, numeric types, Integers especially, have limits in terms of capacity. Error will occur if a candidate will try to process a value beyond the normal limits of Python.

On most platforms, integers can process any numbers between -9,223,372,036.854 and its positive counterpart. This number is the maximum value of the 64-bit.

The int variable has interesting features that varies on the types of bases like: (a) Base 2 which only uses 1 and 0, (b) Base 8 which uses numbers from 0 to 7, (c) Base 10 which is similar to the decimal system, (d) Base 16 which uses numbers 0 t0 9 and letters A to F as digits.

When telling Python to use a different base system, you just add 0 then a special corresponding letter to the number. Here are the letters that Python use to denote the shift in base systems:

b – base 2

o – base 8

x – base 16

For example 0b1100 is in base 2, 0o741 is in base 8 and 0x123aa is in base 16. Also, to convert their values to other bases, you can use the following commands:

bin() – base 2

oct() – base 8

hex() – base 16

2. Floating-point Values

A number that has decimal, regardless if it's just .0, is considered by Python as a floating-point value. As in the example earlier, 5.0 is not an integer; instead, it is a floating point number. There's no reason to be confused because the moment a number has a decimal point, then it is a floating point. It also follows that this data is stored in the float data type. The main advantage floating-point values have over integers is its bigger storage space enabling it to store incredibly large or small values. This does not mean that it has unlimited storage space. In fact, it can contain as much as $1.7976931348623157 \times 10^{308}$and as low as $\pm 2.2250738585072014 \times 10^{-308}$.

There are two ways to assign values using floating-point values. The first one is to assign them directly and the second one is to use the scientific notation like the ones provided above. Remember that a negative exponent gives a fraction equivalent.

3. Complex Number

Perhaps to recap from basic Mathematics, complex numbers are numbers that consists of an imaginary number and a real number paired together. This is often used in the following fields: (a) Dynamic systems, (b) Computer graphics, (c) Quantum Mechanics, (d) Fluid dynamics and (e) Electrical engineering.

Complex numbers have other various uses. Since Python is one of the few programming language that can process complex numbers, people who are in this fields are familiar with the programming language.

Boolean Values

Python has a way to churn out a straight answer that can either be True or False . In fact, computers can be instructed to follow a branch of Mathematics known as Boolean Algebra. While there are still no computers during the time of George Boole (where Boolean Algebra derived its name), several computer language today rely on said mathematics in writing an algorithm to make decisions or responses. In fact, Boolean Algebra has been in existence since 1854, long before the first modern computer ever existed.

If programmers require Boolean Algebra, they refer to the bool data type. The variables can only contain two response values: True or False . This data type allows programmers to assign values using the above response. For instance:

```
myBool = 15 < 3.2
```

will give a response represented by the value False because 15 is not less than 2.

Strings

This data type is similar to words representing an exact meaning making it easier for humans to use this on a daily basis. A string is a group of characters, not necessarily intelligible words, strung together by double quotation marks. For example:

```
thisString = “I love to learn the Python language!”
```

This command assigns the phrase, "I love to learn the Python language!" to the variable thisString . In order for you to understand how the computer understands the command, you should know that the computer does not see the letters. Instead, each letter inside the double quote is represented by a unique number. For instance, the letter A is represented by 65 such that when you type the command:

```
ord(“A”)
```

The command will give 65 as the output. You can try other letters to check their numerical equivalent. Since the computer can't process letters, you will have to convert strings to numbers. In this case, you can use the commands float() or int(). For instance:

```
thisInt = int(“6766”)
```

assigns the integer 6766 and not the string 6766 to the variable thisInt .

Python also allows users to convert numbers to string equivalents by using the command str(). For example:

```
thisString = str(67.6766)
```

creates a string that contains the value 67.6766 before assigning it to the variable thisString . This allows programmers to go back and forth in converting letters to numbers or vice versa before storing the value in a variable.

Date and Time

Almost every application requires date and time for it to work effectively. It paces which tasks should be done at which particular date and time. Since the computer only understands numbers, the concept of date and time does not exist as far as the computers are concerned. The only way for them to be able to interact with inputs of dates and time is if they are converted into readable values that can be stored in variables.

The command import datetime allows programmers to access date and time. In order to get the current time from the clocks inside the computer, you can use the command datetime.datetime.now() .

For example, if the date is June 12, 2015 and the time is 4:00 in the morning and you typed in:

```
datetime.datetime.now()
```

The said command will respond with the following output:

```
datetime.datetime.now(2015, 12, 6, 4, 0, 0, 0)
```

Since this output can be hard to read and interpret, you can arrange the output by using:

```
str(dateime.datetime.now().date())
```

One of the upsides of Python is that the language has a time() command which can be used to get the exact time. The following reserved words can also be used to evaluate their corresponding exact values: microsecond , second , minute , hour , year , month and day .

Chapter 5. Performing Repetitive Tasks

This book has given concise examples on how to perform a single loop task. There are instances however when an application might require constant and continuous evaluation until it reaches a certain point where a desired value is reached and therefore, evaluated. Python allows the performance of repetitive tasks using the for statement.

For Statement

For Python users, the for statement is the first command that beginners need to learn. Almost all known conventional computer languagesuse for statements. For statements determine how many times loops will be executed in the program. Unlike the succeeding loop commands, the number of repetitions in the for loop is already determined.

Here ’ s an example of a For Statement:

```
for Alpha in “Good Morning, Python!”:
```

The command is introduced by the keyword for. The next word is the variable which will contain an assigned value. In the example, the name of the variable is Alpha . The word in signals the start of the string, "Good Morning, Python!" The colon caps the whole command.

The next line must be indented to signify that all indented commands under the for statement will serve as the instruction loop that the computer will follow as part of the for loop.

In order to create a basic for loop, consider the following:

1. Open the Python Edit window
2. Key in the example code block below:

```
AlphaNum = 1

For Alpha in “Good Morning!”:
        Print(“Letter “, AlphaNum, “ is “, Alpha)
        AlphaNum+=1
```

This example uses the variable AlphaNum given a beginning value of 1 which is adjusted in the loop by adding 1 in every cycle. Each letter is then stored in the variable Alpha.

3. Run the module.

The Python Shell window pops out and displays the output:

```
Letter  1  is  G
Letter 2 is o
Letter 3 is o
Letter 4 is d
Letter 5 is 
Letter 6 is M
Letter 7 is o
Letter 8 is r
Letter 9 is n
Letter 10 is i
Letter 11 is n
Letter 12 is g
Letter 13 is !
```

Break Statement

Now if you need to create a program that discriminates an exception in order to break the loop, you will need to learn how to use the break statement. For example, in manufacturing mobile phones, a specific part ran out of supply, the whole process must stop and wait for the depleted part to restock.

The break clause allows the statement to break out of the loop. For example:

1. Open a Python Edit window.
2. Key in the following code:

```
Standard = input("Input a word with less than ten (10) characters: ")
AlphaNum = 1

for Word in Standard:
        print("Letter ", AlphaNum, " is ", Word)
        AlphaNum+=1
        if AlphaNum > 10:
                print("The string is too long!")
                break
```

This example creates the variable Standard where the users are allowed to input at mostten characters. If the program discovers that it has performed the loop more than ten times, it stops and displays an error information saying that, "The string is too long!"

The if statement provides the condition that needs to be fulfilled by the conditional code. When AlphaNum is assigned with a value greater than 10, the whole program will stop processing the rest of the input data.

3. Run the above program. The Python Shell window pops out to ask you for an input.

Input a word with less than ten (10) characters:

4. In order to test the program, key in a word that has less than ten characters. For example, type: Python

```
Input a word with less than ten (10) characters: Python
```

The program will then display the following output:

```
Letter  1  is  P

Letter  2  is  Y

Letter  3  is  t

Letter  4  is  h

Letter  5  is  o

Letter  6  is  n
```

5. Run the program again, and input a word that is exactly ten characters. For example, type: LovePython

```
Input a word with less than ten (10) characters: LovePython
```

The program will then display the following output:

```
Letter  1  is  L

Letter  2  is  o

Letter  3  is  v

Letter  4  is  e

Letter  5  is  P

Letter  6  is  y

Letter  7  is  t

Letter  8  is  h

Letter  9  is  o

Letter  10  is  n
```

6. Run the program again, and input a word that is more than ten characters. For example, type: ILovePython

```
Input a word with less than ten (10) characters: ILovePython
```

The program will then display the following output:

```
Letter  1  is  I

Letter  2  is  L

Letter  3  is  o

Letter  4  is  v

Letter  5  is  e

Letter  6  is  P

Letter  7  is  y

Letter  8  is  t

Letter  9  is  h

Letter  10  is  o

The string is too long!
```

Noticed how the program didn ’ t finish enumerating the whole string. It missed out on the 11th letter, n.

Continue Statement

Another way to control the execution of the loop statement is touse the continue statement. While the break clause just puts an end to the loop, the continue clause is part of the original if statement. In order to illustrate how programmers use the Continue Statement:

1. Open the Python Edit window.
2. Input the following code block:

```
AlphaNum = 1

for Word in "ILovePython!":
            if Word == "y":
            continue
                        print("Letter y cannot be processed!")
            print("Letter ", AlphaNum, " is ", Word)
                        AlphaNum+=1
```

This example assigns the string y to the variable Word such that when the loop encounters the letter y , it will stop executing the loop but will continue to the other letters.

3. In order to illustrate how this differs from the break statement, run the program. Python Shell will pop out giving this output:

```
Letter  1  is  I

Letter  2  is  L

Letter  3  is  o

Letter  4  is  v

Letter  5  is  e

Letter  6  is  P

Letter  8  is  t
```

Letter 9 is h

Letter 10 is o

Letter 11 is n

Letter 12 is !

Pass Clause

Similar to the continue clause, the pass clause displays an error prompt that informs the programmer that a loop has encountered an exception.

1. Open the Python Edit window.
2. Input the following code block:

```
AlphaNum = 1

for Word in "ILovePython!":
if Word == "o":
pass
print("Letter o cannot be processed!")
print("Letter ", AlphaNum, " is ", Word)
AlphaNum+=1
```

This difference is that it does not just skips processing o but it displays a personalized error message saying that a particular element has not been processed.

3. In order to illustrate how this differs from the continue statement, run the program. Wait for the Python Shell will pop out giving this output:

```
Letter  1  is  I
Letter  2  is  L
Letter o cannot be processed
Letter  4  is  v
Letter  5  is  e
Letter  6  is  P
Letter  7  is  Y
Letter  8  is  t
Letter  9  is  h
Letter o cannot be processed
Letter  11  is  n
Letter  12  is  !
```

Chapter 6. Operators

Python has seven main types of operators. They are constructs manipulating operand values. They are symbols that tell the interpreter to perform a certain operation. You have read about operators in Chapter 4. In this chapter, you will learn about arithmetic, assignment, comparison, bitwise, logical, identity, and membership operators.

Arithmetic Operators

Operator	*Description*
Addition (+)	It performs addition of values.
Subtraction (-)	It performs subtraction of values, subtracting the second operand from the first one.
Multiplication (*)	It performs multiplication of values.
Division (/)	It performs division of values, dividing the first operand by the second one.
Modulus (%)	It performs division of values, dividing the first operand by the second one. Then, it returns the remainder
Exponent (**)	It performs exponential calculation on all the operators involved.
Floor Division (//)	It performs division of values, dividing the operands. However, it eliminates the decimal points when the result has been obtained.

Assignment Operators

Operator	*Description*
=	Values are assigned starting from the right to the left operands.
+= add AND	The left and right operands are added. The result is then allocated to the left operand.
-= subtract AND	The right operand is subtracted from the left operand. The result is then allocated to the left operand.
*= multiply AND	The left and right operands are multiplied. The result is then allocated to the left operand.
/= divide AND	The left operand is divided with the right operand. The result is then allocated to the left operand.
%= modulus AND	The modulus of both operands is found. The result is then allocated to the left operand.
**= exponent AND	Exponential computation is done on both operators. The resulting value is then assigned to the left operand.
//= floor division	Floor division is done on the operands. The resulting value is then assigned to the left operand.

Comparison Operators or Relational Operators

Operator	*Description*
==	Condition is true for the equation if both operands have equal values.
!=	Condition is true for the equation if both operands do not have equal values.
<>	Condition is true for the equation if both operands do not have equal values.
>	Condition is true for the equation if the left operand has a bigger value than the right operand.
<	Condition is true if the left operand has a lesser value than the right operand.
>=	Condition is true if the left operand has a value that is either larger or equal to that of the right operand.
<=	Condition is true if the left operand either has a lesser or equal value to that of the right operand.

Bitwise Operators

Operator	*Description*
& binary AND	The bit is copied if it is found in both operands.
| binary OR	The bit is copied if it is found in an operand.
^ binary XOR	The bit is copied if it is found in just one operand.
~ binary ones complement	The bits are flipped.
<< binary left shift	The value of the left operand is moved to the left depending on how many bits were assigned to it by the right operand.
>> binary right shift	The value of the left operand is moved to the right depending on how many bits were assigned to it by the right operand.

Logical Operators

Operator	*Description*
And logical AND	In case the two operands are true, condition is found to be true.
Or logical OR	If one operand is a non-zero, condition is found to be true.
Not logical NOT	The operand’s logical state is reversed.

Identity Operators

Operator	*Description*
Is	Condition is true if variables on either side point to the same object. Condition is false if the case is otherwise.
Is not	Condition is false if variables on either side point to the same object. Condition is true if the case is otherwise.

Membership Operators

Operato r	*Description*
Is	Condition is true if variables on either side point to the same object. Condition is false if the case is otherwise.
Not in	Condition is true if a variable is not found in a specific sequence. Condition is false if the case is otherwise.

Chapter 7. Functions

Functions are blocks of code that serve primarily to execute a particular action. In Python, there are a variety of built-in functions you can choose from. Nevertheless, you can also make your own. These custom functions are called *user-defined functions*.

Defining a Function

Since there are built-in functions, you should feel free to use any of them. However, if you think that they do not suit your program, you can create one accordingly. In fact, writing your own functions allows you to save more time. It is even possible to use such user-defined functions for your other programs. Just don't forget the rules that come with using functions.

First of all, you must begin your block of code with *def*. Then, follow it with parentheses and the function name. See to it that you put your input parameter or argument inside the parentheses. If you want, you may even define parameters using such parentheses. At the beginning of your block of code, don't forget to include a colon (:) and use indention.

The initial statement in a function is optional. So, it is perfectly alright if you do not want to include one. Nevertheless, you can always use it as a documental string or *docstring* of functions. In order to exit a function, you have to use the *return* statement. You can also use *return* to pass expressions back to their caller. However, if there is no argument and you tried to use *return*, it will be pointless and comparable to *return None*.

When writing a program with functions, do not forget to include the *functionname*, which is literally your function's name. Beneath it, you should put the code in your function. See to it that it is indented; because if not, it will not be read as a program. In Python, functions are treated independently in a program. Your computer may not recognize a function, but it will recognize the value it returns.

What does this mean? For instance, you want to store the value *7* inside the variable *d*. When you run your program, your computer will not recognize the variable *d*. Instead, all it will see is the value *7* that you wish to store in *d*.

A function appears as whatever value it was given when it is run. It is practically a short program that you give parameters to. It runs itself and then returns a value. By default, a parameter has a positional behavior. It has to be called the same way it was defined. Take a look at this example:

```
def functionname ( parameters ) :
            “function_docstring”
            function_suite
            return [ expression ]
```

```
def printme ( str ):
            “The example prints out a passed string into a function.”
print str
return
```

Calling a Function

Before you can successfully define a function, you need to provide a name, structure, and parameter specifications. It is crucial for you to provide a function name and a code structure. In addition, you need to provide parameter definition. When you have finalized the basic structure of your function, then you can call it from another function to execute it. However, if you wish to call it directly from the prompt, you may do so as well.

```
# The function is defined at this point
def printme ( str ) :
            “This example prints a passed string into the function.”
print str;
return;

# You can call the printme function here
printme ( “ This is the first call to the user defined function. ” ) ;
printme ( “ This is the second call to the user defined function. ” ) ;
```

When you run the code shown above, you will obtain the following output:

```
This is the first call to the user defined function.
This is the second call to the user defined function.
```

Pass By Reference vs. Value

A reference passes all parameters and arguments in Python. This means that everything the parameter refers to within the function shows any changes you have made to it. For example:

```
# This is where you put the function definition
def changeme ( mylist ) :
            “It changes a passed list within a function”
            mylist.append ( [ 5, 6, 7, 8 ] );
            print “The values within the function are: “, mylist
            return

# You can call the changeme function here
mylist = [ 20, 30, 40 ];
changeme ( mylist );
print “The values outside of the function are: “, mylist
```

When you run the code shown above, you will get the following output:

```
The values within the function are: [ 20, 30, 40, [ 5, 6, 7, 8 ] ]
The values outside of the function are: [ 20, 30, 40, [ 5, 6, 7, 8 ] ]
```

Regarding arguments being passed by reference, take a look at this:

```
# This is where you put a function definition
def changeme ( mylist ) :
            “It changes a passed list within the function”
mylist = [ 5, 6, 7, 8 ];
print “The values within the function are: “, mylist
return

# You can call the changeme function here
mylist = [ 20, 30, 40 ];
changeme ( mylist );
print “The values outside of the function are: “, mylist
```

The parameter *mylist* is local to the function *changeme*. When you change it within the function, you will see no change. The function does not have any effect on it, and you will get the following output:

```
The values within the function are: [ 5, 6, 7, 8]
The values outside of the function are [ 20, 30, 40 ]
```

As you can see, the reference has been overwritten within the called function.

Math Functions

These functions are meant to perform mathematical operations. There is no way you can use them with complex numbers. The following are the math functions used in Python:

Number-Theoretic and Representation Functions

math.floor (x) – it returns the floor of x, which is the largest integer that is less than or equal to (≤) x. It delegates to *x . _ floor ( )* if x is not a float. This returns an integral value.

math.ceil (x) – it returns the ceiling of x, which is the smallest integer that is greater than or equal to (≥) x. It delegates to *x . _ ceil _ ( )* if x is not a float. This returns an integral value.

math.factorial (x) – it returns the x factorial. If x is negative or is not an integral, it raises the exception *ValueError*.

math.copysign (x, y) – it returns a float with the magnitude of x and the sign of y.

math.fabs (x) – it returns the absolute value of x (| x |).

math.frexp (x) – it returns the exponent and mantissa (coefficient or significand) of x as the pair *m* and *e* (m, e) such that x = m * 2 ^ e, where *m* is a float and *e* is an integer.

math.ldexp (x, i) – it returns x * (2 * * i).

math.fmod (x, y) – it returns fmod (x, y). Take note that in Python, the expression *x % y* may not yield the same output as in C. It returns a result with the sign of y. It may also not be computable for a float argument. It is more ideal to be used with integers instead of floats.

math.modf (x) – it returns the integer and fractional parts of x. The results are floats and have the sign of x.

math.fsum (iterable) – it returns an accurate floating point sum of values in the iterable and tracks multiple intermediate partial sums to avoid loss of precision.

math.isnan (x) – it verifies if the float x is a NaN (not a number).

math.isinf (x) – it verifies if the float x is either negative or positive infinite.

math.modf (x) – it returns the integer and fractional parts of x. The results are floats and have the sign of x.

math.trunc (x) – it returns the real value of x that is truncated to an integral.

Logarithmic and Power Functions

math.pow (x, y) – it returns x raised to the power of y.

math.exp (x) – it returns e * * x.

math.expm1 (x) – it returns e * * x – 1.

math.log10 (x) – it returns the base 10 logarithm of x.

math.log1p (x) – it returns the natural logarithm of 1 + x (base e).

math.log (x [, base]) – it returns the logarithm of x to the given base, and returns the natural logarithm of x if there is no specific base.

math.sqrt (x) – it returns the square root of x.

Trigonometric Functions

math.sin (x) – it returns the sine of x (sin (x)), in radians.

math.cos (x) – it returns the cosine of x (cos (x)), in radians.

math.tan (x) – it returns the tangent of x (tan (x)), in radians.

math.asin (x) – it returns the arc sine of x (arcsin (x)), in radians.

math.acos (x) – it returns the arc cosine of x (arccos (x)), in radians.

math.atan (x) – it returns the arc tangent of x (arctan (x)), in radians.

math.atan2 (y, x) – it returns *atan (y / x),* in radians.

math.hypot (x, y) – it returns *sqrt (x * x + y * y)*, which is the Euclidean norm or the magnitude.

Angular Conversion

math.radians (x) – it converts the angle x from degrees (deg) to radians (rad).

math.degrees (x) – it converts the angle x from radians (rad) to degrees (deg).

Hyperbolic Functions

math.asinh (x) – it returns the inverse hyperbolic sine of x.

math.acosh (x) – it returns the inverse hyperbolic cosine of x.

math.atanh (x) – it returns the inverse hyperbolic tangent of x.

math.sinh (x) – it returns the hyperbolic sine of x.

math.cosh (x) – it returns the hyperbolic cosine of x.

math.tanh (x) – it returns the hyperbolic tangent of x.

Constants

math.e – it is the mathematical constant *e*.

math.pi – it is the mathematical constant *pi*.

Special Functions

math.gamma (x) – it returns the gamma function at x.

math.lgamma (x) – it returns the gamma function at x.

math.erf (x) – it returns the error function at x.

math.erfc (x) – it returns the complementary error function at x.

Here are some examples of codes that involve math functions:

math.sqrt (x) :

```
import math
print math.sqrt ( 25.0 )
print math.sqrt ( 5 )
try:
        print math.sqrt ( - 3 )
except ValueError, err:
        print ‘Not possible to compute for the sqrt ( - 3 ) : ‘ , err
```

Running this code gives you the following output:

```
5.0
2.2360679775
Not possible to compute for sqrt ( -3 ) : math domain error
```

math.log (x) :

```
import math
print math.log ( 12 )
print math.log ( 12, 3 )
print math.log ( 0.5, 4 )
```

Running this code gives you the following output:

```
2.48490664979
2.26185950714
-0.5
```

Referring to this example, you can see that the logarithm (log) function finds y, where x = b * * y. It calculates the natural logarithm (base e) by default. If there is a second argument, its value becomes the base. If the value of x is less than one, you acquire a negative result.

A lot of users have this misconception that functions and formula in Python are written backwards. However, this is not the case because they are actually written inside out. In order for your program to work, you need to break down the functions into separate function calls. If you want to include a floating point, it is crucial that you use float *(raw_input ( ))* rather than of *int (raw_input ( ))*.

Function Arguments

Users are allowed to call a function through:

- A keyword argument
- A required argument
- A variable-length argument
- A default argument

These are all types of formal arguments.

The Keyword Argument

This argument is related to the function call. Whenever you use it in function calls, it is identified by the parameter name. It is alright for you to skip it or place it out of order. The interpreter uses keywords in matching values with their respective parameters. If you want to make a keyword call, you can do it with *printme( )*.

```
# This is where you put the function definition
def printme ( str )
        "It prints a passed string into the function"
        print str;
        return;

# You can call the printme function here
printme ( str = "String sample" );
```

Running this code gives you the output:

```
String sample
```

As evident in the output, your result is the string that gets passed onto the function.

The Required Argument

This argument is passed onto a function in a particular order. The number of arguments you use is crucial, so make sure that you keep tabs on it. It must match the function definition that you use. In order for you to call the function *printme ( )*, an argument has to be passed. Otherwise, all you will see is a syntax error.

```
# This is where you put the function definition
def printme ( str ) :
            “It prints a passed string into the function”
            print str;
            return;

# You can call the printme function here
printme ( );
```

If you run this code, you will get the following output:

```
Traceback ( most recent call last ):
            File “test.py”, line 11, in <module>
printme ( );
TypeError: printme ( ) takes exactly 1 argument ( 0 given )
```

The Variable-Length Argument

When you define functions, it is a must that you process them more than the number of arguments you have. The variable-length argument is not like a default or required argument because it does not show up in the function definition. Functions with non-keyword variable arguments have this syntax:

```
def functionname ( [ formal_args, ] “var_args_tuple ) :
            “function_docstring”
            function_suite
            return [expression]
```

Make sure that you put an asterisk (*) prior the variable names that you use to store non-keyword variable argument values. Remember that the tuple stays empty until you declare another argument in the function call.

```
# This is where you put the function definition
def printinfo ( arg1, *vartuple ) :
“This prints a variable passed argument”
print “The output is: “
print arg1
for var in vartuple:
            print var
return;

# You can call the printinfo function here
printinfo ( 20 );
printinfo ( 30, 40, 50 );
```

If you run the code shown above, you will get the following output:

```
The output is:
20
The output is:
30
40
50
```

The Default Argument

This argument assumes default values when values are not given in the function calls.

```
# This is where you put the function definition
def printinfo ( name, age = 10 ):
            “It prints a passed info into the function”
print “What is your name?”, name;
print “How old are you?”, age;
return;
```

```
# You can call the printinfo function here
printinfo ( age = 99, name = “Wendy” );
printinfo ( name = “Wendy” );
```

If you do not pass the default age, the output includes it:

```
What is your name? Wendy
How old are you? 99
What is your name? Wendy
How old are you? 10
```

Anonymous Functions

In Python, anonymous functions are supported. They can be created during runtime using the construct *lambda*. These functions are not bound to any name. Take note that the lambda here is not the same as the lambda used in functional programming languages. Basically, Python’s lambda is meant to be a tool for creating functions, more commonly referred to as function objects. Nevertheless, it is still powerful and often used in conjunction with *filter ( ), reduce ( ), map ( ),* and other functional concepts.

Anonymous functions are not declared using *def*, but rather *lambda*. Def and lambda are actually Python’s two main tools for building functions. Keep in mind that there are certain rules that govern anonymous functions. For starters, forms of lambda can contain any number of arguments. However, they should return just one value as an expression. It is not possible to use multiple expressions or commands. Also, anonymous functions are not allowed to be called to print directly because lambda needs to call certain expressions.

Some users believe that lambda is a function, albeit a single line version. This is a common misconception. Lambda is not really similar to the inline statements in C and C++ when it comes to passing function stack allocations for invocations. Take note that lambda has its very own namespace. Therefore, it is not allowed to access variables, except for the ones that are within the global namespace and the parameter list.

The syntax of lambda is as follows:

```
lambda [ arg1 [ ,arg2 , . . . argn ] ] : expression
```

Chapter 8. Variable and Multiple Assignments

Variable Assignment

You have read about variables in Chapter 4 as well. Variables are reserved memory storage for values. Whenever you create one, you reserve a space in the memory. The interpreter then allocates the memory, depending on your variable data type, and identifies which elements are allowed to be stored in it. Users are allowed to store decimals, characters, and integers in variables if you assign them with data types.

You do not have to declare variables in an explicit manner if you want to reserve a space in the memory. Variables are declared automatically the moment you give them values. You can assign values by using the equal sign (=). The operand at its left side is the variable name. The operand at its right side is the variable value.

To help you understand variable assignment better, here is an example:

```
amount = 10                                # This is an integer assignment.
kilometers = 100.0                       # This is a floating point.
name = “Wendy”                         # This is a string.

print amount
print kilometers
print name
```

As you can see in the given example, the values 10, 100.0, and Wendy are all assigned to amount, kilometers, and name, all of which are variables. So when you run this code, you will get this result:

```
10
100.0
```

```
Wendy
```

Whatever you input as values, you get as output. It does not matter what kind of value you input. In the above given example, an integer, a float, and a string were used.

Multiple Assignment

Users are allowed to simultaneously assign values to multiple variables. Take a look at the following:

```
x = y = z = 1
```

Here, the integer object was made using a value. Variables *x*, *y*, and *z* had the same value: *1*. Because of this, they are allocated to the same memory. If you wish to allocate multiple objects to multiple variables, you may do so as well. Take a look at the following:

```
x, y, z = 2, 4, “wendy”
```

Here, two integer objects have the values of *2* and *4*, respectively. Such values were allocated to the *x* and *y* variables while the string object *“wendy”* was allocated to the *z* variable.

Chapter 9. Lists

The list is the most versatile compound data type. It contains items that are enclosed in brackets ([ ]) and separated by commas (,) in a particular order. It implements sequence and lets you remove and add objects from it.

The following is classic example of a list:

```
#!/usr/bin/env phyton

list = [ ‘wxyz’, 3.14159265359, ‘wendydawn’, ‘david’ ]
shortlist = [123]

print list              # This prints the entire list.
print list [ 0 ]                          # This prints the first element in the list.
                                        print list [ 3:14 ]              # This prints the fourth to
                                        the fourteenth elements in the list.
print list [ 2: ]             # This prints the third element onwards.
print shortlist * 3             # This prints the list three times.
print list + shortlist              # This prints the concatenated lists.
```

Running the above code gives you the following output:

```
[ ‘wxyz’, ‘3.1415926535900001, ‘wendydawn’, ‘david’ ]
wxyz
[ ‘david’ ]
[ ‘wendydawn’, ‘david’ ]
[ 123, 123, 123]
[ ‘wxyz’, 3.1415926535900001, ‘wendydawn’, ‘david’, 123 ]
```

Creating a List

In order for you to create a list, you have to place expressions inside square brackets, such as the following *list display*:

```
W = [ ]
W = [ expression, . . . ]
```

In Python, computed lists known as *list comprehensions* are also supported. Its syntax looks like the following:

```
W = [ expression for a variable in a sequence ]
```

in which an expression is evaluated once for each item in a sequence.

This expression can be anything. It may include any kind of objects, even other lists. It may also include multiple references to a particular object.

Users are allowed to make use of the built-in list in Python to create a list as well. For example:

```
W = list ( )                          # This list is empty.
W = list ( sequence )
W = list ( expression for a variable in a sequence)
```

This can be any kind of iterable or sequence object, including generators and tuples. In case you pass in another list, a copy is made by the list function.

Keep in mind that a new list is created each time you execute the [ ] expression. You would never be able to create a new list when you assign a variable with a list. Take a look at the following:

```
W = D = [ ]                           # The two names point to the same list.
W = [ ]
D = W                           # The two names point to the same list.
W = [ ] ; D = [ ]               # These lists are independent.
```

Accessing a List

A list implements a standard sequence interface. It responds to the * and + operators just like strings, meaning they concatenate and repeat as well, except that they result in a new list instead of a string. A list actually responds to every general sequence operation used on strings.

For instance, *len (L)* returns how many items there are in the list, *L [i : j]* returns a new list that contains whatever objects are between *i* and *j*, and *L [i]* returns the first item with an index of *0* or the item at the index. When you pass in a negative index, the list’s length is added to the index. You can use *L [-1]* to access the list’s last item.

In case the resulting index is out of the list, an *IndexError* exception is raised. The slices are regarded as boundaries, and the result contains all the items between them. Slice steps are also supported by lists. Here are a few examples of slice steps:

```
seq = L [ start : stop : step ]
seq = L [ 1 : : 2 ]                    # It gets every other item,
beginning with the second
seq = L [ : : 2 ]                      # It gets every other item,
beginning with the first
```

To help you understand lists better, here are more examples:

Input:

```
len ( [ 1, 2, 3 ] )
```

Output:

```
3
```

In the above given example, len was used to determine the length of the expression.

Input:

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

Output:

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

In the above given example, the expression performed the process of concatenation.

Input:

```
[ ‘Hello!’ ] * 5
```

Output:

```
[ ‘Hello!’ , ‘Hello!’ , ‘Hello!’ , ‘Hello!’ , ‘Hello!’ ]
```

In the above given example, the expression was used to perform the process of repetition.

Input:

```
3 in [ 1, 2, 3 ]
```

Output:

```
True
```

In the example given above, the membership of the expression was determined to be either true or false.

Input:

```
for x in [ 1, 2, 3 ] : print x
```

Output:

```
1 2 3
```

In the above given example, the expression was used to perform the process of iteration.

Looping Over Lists

To loop over items in a list, you can use the *for – in* statement. For example:

```
for item in L :
            print item
```

In case you need both the item and the index, you can use the function *enumerate*, such as in this example:

```
for index, item in enumerate ( L ) :
            print index, item
```

In case you need just the index, you can use *len* and *range*, such as in this example:

```
for index in range ( len ( L ) ) :
            print index
```

The iterator protocol is supported by the list object. In order for you to create an iterator explicitly, you can use *iter*, which is a built-in function in Python. Take a look at the following example:

```
i = iter ( L )
item = i.next ( )                        # It gets the first value
item = i.next ( )                        # It gets the second value
```

In Python, there are different shortcuts for basic list operators. For instance, when a list contains numbers, you can get their sum by using the function *sum*. When a list contains strings and you want to combine them into just one string, you can use the *join* string method. You would then be able to combine these strings into one long string. For example:

```
s = ‘ ‘ .join ( L )
```

Modifying a List

Users are allowed to assign variables to individual slices or items, and then delete them. For example:

```
L [ i ] = onj
L [ i : j ] = sequence
```

Take note that operations modifying lists also modify them in place. Hence, if you have numerous variables that point toward the same list, every variable is going to be updated all at once. For example:

```
L = [ ]
M = L
```

# This modifies the two lists

```
L.append ( obj )
```

If you want to make a new list, you can use list to create another copy quickly. You can also use slicing. For example:

```
L = [ ]
M = L [ : ]                    # This creates a copy.
```

# This only modifies L

```
L.append ( obj )
```

It is possible to add items to existing sequences in Python. For instance, *append* adds an item to the end of the list while *extend* adds an item from another sequence or list to the end. *Insert* inserts an item at an index and moves the other items to the right. It is also possible to remove items. For instance, you can use *del* to remove one or all items that a slice identifies. You can use *pop* to remove one item and then return it. You can also use *remove* to search for a particular item and then remove the first item that matches it from the list. *Del* and *pop* are very similar, except that *pop* can return the removed item.

Another thing you can do with lists is reversing their order. If you wish to do this, you can use

```
L.reverse ( )
```

It is actually very easy and very quick to reverse a list's order. In fact, reversing a list temporarily can speed things up when you need to insert and remove certain items at the list's beginning.

Keep in mind that the *for – in* statement maintains the internal index. It is incremented for every iteration in the loop. This means that in case you decide to modify a list you are looping over, your indexes will be out of sync. You may even end up skipping over certain items or processing the same ones over and over again. This is obviously not a good outcome. To fix such problem, you can loop over the list's copy. For example:

```
for object in L [ : ] :
        if not condition :
                del L [ index ]
```

Likewise, you can create another list and then append to it, such as in the following:

```
out = [ ]
        for object in L :
                if condition :
                        out.append ( object )
```

It is common practice to apply a function to each of the items in the list and then replace it with the function's return value. For example:

```
for index, object in enumerate ( L ) :
            L [ index ] = function ( object )
out = [ ]
for object in L :
            out.append ( function ( object ) )
```

If you want to rewrite the above given example and make it simpler, you can use *map*. It is a built-in function that makes your code more efficient. This happens because your function object is only fetched once. Anyway, you can rewrite the above given example as the following using *map*:

```
            out = map ( function, L )
            out = [ function ( object ) for object in L ]
```

For other constructs, such as calls or expressions to object methods, you can use *lambda* or a callback to run the program and make it more efficient as well as easier to understand.

In case you need both the index and the item, you can use *enumerate*, such as in the following example:

```
            out = [ function ( index, object ) for index, object in enumerate ( L
) ]
```

List is also ideal for implementing simple data structures, including queues and stacks. For bigger structures, on the other hand, you can use *collections.deque*, which is a specialized data structure. A least-recently-used (LRU) container is another data structure that you can use. However, it is only applicable for small structures.

Searching a List

You can use the operator *in* to check whether or not a particular item is included in a list and *index* to perform a linear search and stop at the first item that matches it. In case no matching item is found, a *ValueError* exception is raised. If you want to obtain the index for matching items, you can create a loop and then pass in a start index.

Count is used for counting matching items, such as in the following example:

```
n = L.count ( value )
```

Take note that *count* loops over the whole list. Hence, if you wish to verify if a certain value is found in it, you can use *index* or *in*. In order for you to obtain the largest or smallest item in a list, you can use *max* or *min*, which are built-in functions in Python. For example:

```
hi = max ( L )
lo = min ( L )
```

To pass in a *key* that maps the items in a list before they get compared, you can use *sort*. For example:

```
hi = max ( L, key = int )
lo = min ( L, key = int )
```

Sorting a List

If you want to sort a list in place, you can use *sort*. Its syntax is as follows:

```
L.sort ( )
```

In order for you to obtain a sorted copy, you can use sorted, which is another built-in function in Python. For example:

```
out = sorted ( L )
```

Keep in mind that Python does not need to allocate new lists just to hold results. Hence, an in-place sort is more efficient to be used. By default, the sort algorithm identifies the order through means of comparison. It compares all the objects in a list from one another. If you wish to override this, you can pass in a callable object, taking two items and returning *-1* for *less than*, *1* for *greater than*, and *0* for *equal*. You can use *cmp* to perform this operation. Take a look at the following example:

```
def compare ( x, y ) :
        return cmp ( int ( x ) , int ( y ) )        # This compares the values
as integers
L.sort ( compare )
                                        def compare_columns ( x , y ) :#
                                        This sorts on
                                        ascending index 0 and descending
                                        index 2
                                        return cmp ( x [ 0 ] , y [ 0 ] ) or
                                        cmp ( y [ 2 ] , x [ 2 ] )
                                        out    =    sorted    (    L,
                                        compare_columns )
```

Conversely, you can use mapping between search keys and list items. When you do this, you will be able to build key arrays by letting the sort algorithm make a pass over your data. Both the list and the key array will then be sorted based on your keys. However, if the list is too big or the transform is complex, you can use a compare function to make things easier. After all, the items only need to be transformed one time.

Printing a List

List does a *repr* on every item, and adds commas and brackets as necessary. Take a look at the following example:

```
print [ 1, 2, 3 ]        # This prints out [ 1, 2, 3 ]
```

In order for you to control the formatting, you can use *join* and combine it with either a generator expression or a list comprehension. You can also use *map*. If you wish to print string fragments to a certain file, you can skip using *write* and go for *writelines* instead. You can use the following example if all the items in the list are strings:

```
sys.stdout.writelines(L)
```

Chapter 10. Tuples

Tuples are basically sequences of immutable objects that are quite similar to lists. Their primary purpose is to prevent things from changing and keep them constant. However, unlike lists, they are not allowed to be changed and they involve parentheses rather than brackets. There is no way for you to add, delete, or change elements in a tuple.

But what if you made a mistake and you changed a variable you are not supposed to change? Well, you can fix this mistake by converting a tuple into a list or vice versa. Also, unlike lists, tuples are not homogenous sequences. They are heterogeneous data structures, so there are different meanings to their entities. Keep in mind that tuples have structure and lists have order.

How can you create a tuple? It’s easy: just separate your values using commas or put them in between parentheses. For example:

```
tup1 = ( ‘algebra’, ‘physics’, 1900 , 2015 );
tup2 = ( 5, 6, 7, 8 );
tup3 = “w”, “x”, “y”, “z”;
```

It is also possible for you to create an empty tuple that does not contain any value:

```
tup1 = ( );
```

Take note that you still need to put a comma even if the tuple has a single value. For example:

```
tup1 = (28, );
```

Tuple indices begin at 0, and can both be concatenated and sliced.

Slicing a Tuple

You can reference multiple values by slicing the tuple. For example:

```
tup [ 1 : 5 ]
```

Tuple Assignment

Tuples of variables on the left side of assignments can be assigned to values from tuples on the right side of assignments. Consider the following example:

```
( name, middle_name, surname, ID_number, profession ) = Chuck Norris
```

This single line performs an operation equivalent to five assignment statements. Then again, your program will not work if the number of elements in your tuple does not match the number of variables that you use on the left side.

You can think of a tuple assignment as tuple packing and/or tuple unpacking. For example:

```
w = ( “Wendy, 99, “WD” )          # This is an example of tuple packing
```

When it comes to tuple unpacking, you unpack the values into variables or names. For example:

```
w = ( “Wendy”, 99, “WD” )
```

```
(name, level, initials ) = w          # This is an example of tuple unpacking
```

At times, it is ideal to swap values. With a standard assignment statement, you need to use a temporary variable. So if you want to swap x and y, such as in the following example, you have to use a tuple assignment:

```
temp = x
x = y
y = temp
```

To solve such issue, you can write:

```
( x, y ) = ( y, x )
```

On the left side, you can see a tuple of variables. On the right side, you can see a tuple of values. Every value is allocated to a specific variable. Every expression you see at the right is evaluated before the assignments. This is what makes tuple assignments versatile. Remember that the number of values at the right and the number of variables at the left must be equal. Otherwise, you will get an error like this:

```
( w, x, y, z ) = ( 8, 9, 10 )
```

If you use this, you will get a *ValueError* because you need more than three values to unpack.

Tuple As A Return Value

Note that a function can only return one value. However, you can return more than one value if you turn that value into a tuple. Oftentimes, users wish to know the exact date of the year, the lowest and highest scores, or even the standard deviation. It is also a common math problem to find the circumference and area of a circle with a given radius. So using this example, you can write:

```
def f ( r ) :
            " " " Return ( area, circumference ) of a circle with radius r " " "
            a = math.pi * r * r
            c = 2 * math.pi * r
            return ( a, c )
```

Conclusion

Thank you again for downloading this book!

I hope this book was able to fulfill its promise to equip you with the necessary skills to harness the power of Python Programming. Hopefully by the time that you put down this book, you are now able to wield the power of Python in the fulfillment of your everyday duties at work.

Python language is one of the most widely-used programming language in the world. It is also important to remember that this book only provided you with the basics of what you must know to appreciate the beauty of Python. What you do with this appreciation and where to go from here is now up to you.

The next step is to practice coding using the Python programming language.

Finally, if you enjoyed this book, then I ’ d like to ask you for a favor, would you be kind enough to leave a review for this book on Amazon? It ’ d be greatly appreciated!

Thank you and good luck!