2021 Beginners Guide to PYTHON PROGRAMMING LANGUAGE

A Crash Course to Mastering Python Programming Language in One Hour

GARY ELMER

2021 Beginner

PYTH

PROGRAM

LANGU

A Crash Course to Ma
Programming Langua

2021 Beginners Guide to Python Programming Language

A Crash Course to Mastering Python Programming Language in One Hour

Gary Elmer

Copyright

Printed on acid-free paper.

Printed in the United States of America

Table of Contents

CHAPTER ONE

INTRODUCTION TO PYTHON

You are probably wondering 'How do I learn Python easily?' Well, look no more, help is here. The main goal of this book is to take you from being an absolute beginner to a pro in coding in Python 3. Python gives a simple and straightforward approach thereby making it an impeccable programming language. The Python interface gives you an avenue to go on coding adventures as you get instant feedback on your ideas with the interactive mode.

Whatever your reason for venturing into programming is, whether building games or user interface, Python can give it all to you in simple steps and even short codes. Since Programming is all about making your computer follow your instructions and bringing your ideas to reality, then with Python. Even if you are not new to programming and have learnt other languages before, you are likely going to need Python to step up your coding game.

Throughout this book, I will walk you through writing codes with Python in simple and fun steps. I will be dishing out ideas and skills you will need as a beginner and walk you through the terms and basics you need to know about coding. In order to get the most out of this book, you need to be willing to learn. Take notes and write the lessons out in your journal, this helps in retaining knowledge for a longer time and helps the lesson stick. Consistency is the key to being an expert programmer so I recommend that you ensure you code every day.

Python techniques discussed in this book are concise so if you are interested in learning more about any of the concepts discussed you can progress to advanced textbooks. The first two chapters introduce you to Python, its features and rudiments. Going further, we explore how to write programs and implement them, the various sets used and writing. The last chapter discusses useful tips while writing programs. Furthermore, errors and bugs that might occur during coding are also discussed.

To get started you need to install Python. Go to www.python.org to download the latest version and install. Some computers come with the program pre-

installed while some don't. If you use a Mac or Linux, find out if its installed on it and ensure it is the latest version, if it's not you would need to upgrade to the latest version. Now that you've installed Python on your PC, let's start the journey of programming.

PROGRAMMING LOGIC AND TECHNIQUES

When we talk about programming logic and techniques, we are referring to the instructions in a program arranged in an order so as to solve a problem. While working on a problem, different program logic can be developed as a solution, what matters most is that the logic works for the problem and the given task is performed. In programming, you have to be conversant with the characters used in giving instructions in order to perform a given task. These characters include:

1. Variable
2. Constant
3. Keywords
4. Data types
5. Identifiers
6. Repetition/Loops
7. Decisions/Selections
8. Arrays

INPUT, PROCESS AND OUTPUT (IPO)

In programming, data processing has a laid out structure or model that it follows while a specific task is being carried out. This model is known as the IPO. Basically, data fed into a computer through the keyboard, touchscreen or scanner is known as the **input** . Data in its raw form is of lesser value until it is processed and output is generated. **Process** is the activity carried out on the input which converts it into something meaningful. **Output** is the result of the processed input and it can be in the form of pictures, documents, audios and in programming, output is usually the

execution of the task for which a program as written and processed. So in Python, the input is the written code or instruction while output is the task performed.

PROGRAMS

A program is a set of commands in the form of codes written to instruct a computer to execute a given task. Codes are embedded in a program so they are written as the input and processed as instructions to perform a specific task. An example of program is

Z = X + Y,

The mathematical expression given above is an example of a program where X, Y and Z are variables. When we instruct the computer to calculate the value of X and Y, the result which is Z is the output.

HOW LOGIC OF PROGRAMS WITH CONDITIONS ARE REPRESENTED

When conditions are introduced into a program, the output of the program will be according to the condition introduced. In Python, there are 'true' or 'false' values you can use as conditions on the program.

The 'if' condition contains a set of codes which only runs when the condition embedded in the code is true. If the condition is not met and the code is false, 'else' code with a different set of instructions run. So 'if' code is used when a condition is met (true) while 'else' is used when the condition is false and the 'if' condition is not met. The 'elif' condition simply gives an option to try another condition if previous conditions are not true. An example is:

```
a = 350; b=56
if b  > a :
print ("a is greater than" a)
elif a == b :
print ("a and b are equal")
else :
print ("a is greater than b")
```

VARIABLES AND CONSTANTS

When codes are written, values need to be stored in a memory bank where they can be retrieved, these memory banks are known as variables and constants. Variables are datasets that can be changed while constants' values remain unchanged throughout a program's execution. A constant value is usually named and assigned initial values while the variable value can be assigned in the process of program execution.

DATA TYPES

In python, each dataset given belongs to a category which shows the kind of tasks a program can be expected to implement on the dataset, this category is known as the data type. The standard types of data in Python are:

1. Numeric
2. Sequence type
3. Boolean
4. Set
5. Dictionary

NAMING VARIABLES

In one of the previous sections, we defined variables as values that can be modified in a program. When you change the values of variables in Python, you need to follow certain guidelines so you do not create a wrong code that will not deliver the required output. Python has specific variable naming guidelines:

1. Variable naming is case sensitive. For example, uppercase letter O can be mistaken for number zero thereby causing confusion in the code so you have to be careful when using letters like this or avoid them when writing codes.
2. Keep variable names short and clear. Do not just use a letter to represent a certain word as the program can be interpreted as something as, e.g. do not use **p** for **pillow** to avoid confusion.

3. Letters, underscores and numbers are the only characters that can be used in naming variables. When naming variables, bear in mind that you cannot start with a number. Using underscores in variables also makes codes clearer, for instance, **a_variable** is easier to understand than **avariable** .
4. Try as much as possible not to use spaces in variables naming, use underscores instead.

When naming variables, bear in mind that you are probably not going to be the only one to work on the program, therefore use names that describe the information explicitly and specifically to avoid confusion. Do not try to save time by rushing on variable naming, this might actually slow you down in the long run when you have issues with deciphering the program. As you write more programs, you get better at naming variables, you might run into errors and dead ends at first but do not worry, practice makes perfect. An example of a clear and complete program is

```
house price = price_per_room * rooms +
price_per_floor_squared * ( floors ** 2 )
house_price = house_price + expected_mean_house_price
```

The above program shows the variable names in an easy to understand format and anyone can decipher what the program means.

USING OPERATORS

When writing programs in Python, certain elements are used to execute tasks on given variables and values to give results. These elements are represented by symbols known as operators. For example, if we need to perform subtraction operation on given values, we use the '-' operator to give the instruction, i.e;

```
print ( 15 - 11)
```

There are various categories of operators in Python:

1. Arithmetic operators: they are used to execute mathematical tasks on numeric values. Examples of arithmetic operators are addition (+); subtraction (-); multiplication (*); division (/); modulus (%); exponentiation (**) and floor division (//).

2. Assignment operators: these operators are used to assign values to variables. They assign values of right operand to the left operand. Their examples are equal to (=); add AND (+=); subtract AND (-=); multiply AND (*=); divide AND (/=); modulus AND (%=); exponent AND (**=); floor division AND (//=).
3. Comparison operators: they are used to compare two given values on both sides of the operand and deduce the relationship that exists between them. The common comparison operators are equal (==); not equal (!=); greater than (>); less than (<); greater than or equal to (>=); less than or equal to (<=). Another name for comparison operators is relational operators
4. Logical operators are used in combining conditional statements. 'And' (true if both statements are true); 'or' (used if one of the statements are true) and 'not' (false if both statements are false) are all examples of logical operators. They are referred to as Boolean operators.
5. Membership operators: are used to identify the relationship of a particular sequence in an object. Operators used to indicate membership are 'in' (true if a sequence with the specified value is found within an object); 'not in' (true if a sequence with the specified value is not found within an object).
6. Bitwise Operators: are used in comparison of binary numbers. They work on bits and execute tasks bit by bit. AND (&); OR (|); XOR (^); NOT (~); Zero fill left shift (<<) and Signed right shift (>>) are examples of bitwise operators used in python.
7. Identity operators: they compare the memory locations of two different objects. The two identity operators are 'is' (true if the variables on either sides of the operator point to the same object and false if not); 'is not' (false if the variables on either side of the operator point to the same object and false if not).

There is operator precedence in python. If there are more than one operations to be performed, one of the operators needs to be evaluated before others. For example, multiplication has a higher precedence than addition.

ITERATION

Sometimes, when writing programs in Python, you might need to repeat a task. When repeating similar tasks, you are bound to make errors so Python makes this less tasking by having features to aid repetition of tasks without errors. The execution of a code over and over again is known as **iteration** . With iteration, you can repeat the same task a number of times. A program that utilizes iteration in its codes is referred to as a **loop** . There are two types of iteration; definite and indefinite. While carrying out an indefinite iteration, the number of times the loop has to be repeated is not stated but with definite iteration, it is stated when the loop starts. In Python, definite iteration is performed with the for loop while indefinite iteration is performed with while loop.

When definite iteration loops are used, for is the keyword used to integrate them into a program in Python. Python implements the collection-based or iterator based loop which loops a collection of items rather than individual values. It is the most widely used loop and uses the keyword for each instead of for . An example of for loop is

```
for a = 10 to 15
< loop body >
```

The program above means for every time a is between 10 and 15, the program should be repeated.

The while loop is used to indicate in-definite iterations where the number of times the loop has to be iterated is not specified. It is used to iterate a set of codes with a 'true' condition attached to it. When the 'true' condition is no longer met and becomes false, the while loop stops iterating and the next loop in the program is performed. The next loop which becomes executed is the else statement which has been programmed to run if the while loop is broken. In conclusion, the while loop is used when we do not know the number of times a program has to be executed, if it is known, then for loop should be used. An example of while loop is

```
num = 65
while num<81
print ( num )
```

this program will continue to print 65 until the loop is broken and the condition of the number being less than 81 is no longer met. It will then

proceed to execute the next program.

CHAPTER TWO

FEATURES OF PYTHON

As a programming language, Python offers many useful and sophisticated features thereby making it a first choice in the programming industry. It is versatile as it offers procedural oriented programming as well as object-oriented programming. Python offers many features to its users, some significant ones are:

1. Free and open source

As a widely used programming language, Python is free to use for everyone and is available to download free of charge on its official website . Its source code which is freely available to all is updated regularly to bring new features to the users. With a growing community, users worldwide can contribute and share ideas amongst themselves.

2. Ease of coding

With Python, anyone can learn to code. Although it is a high-level programming language, it is easy to master and very beginner-friendly when compared to other programming languages.

3. Extensibility

With Python, programs can be written into other languages like C++ programming language and vice versa.

4. Portability

Python can be used across various OS without having to rewrite the code. If a program was written on a Mac platform, it can be run on other platforms like Windows and Linux without changing the code.

5. Object-oriented Language

Object-oriented language is one of the key features of Python. It recognizes concepts of classes and objects encapsulation which helps in writing

programs with lesser codes and reusable codes.

6. High level language

When coding with Python, no need to worry about how the codes are structured or built. A line is enough to execute some complex programs in Python as against other programming languages that require long lines of codes for simple tasks.

BUILDING BLOCKS IN PYTHON

Programs can be developed in Python by making use of some building blocks. Some of these building blocks are:

1. **Variables:** As was discussed earlier, these are values that can be changed or manipulated when writing a program. Python allocates value to variables based on the references given.
2. **Data types:** Since Python gives value of variables based on the references assigned, the data types that are compatible with Python are Boolean, integer, string, list, object, dictionaries and float.
3. **Operators:** Are symbols used to represent mathematical tasks that are executed on given variables and values to give results. Operators used in Python are arithmetic, assignment, bitwise, comparison, logical, membership and identity operators.
4. **Expressions:** Are combinations of values with certain operators like multiplication or division. An example is x + y = z where x and y are variables.
5. **Identifiers:** As stated earlier variables and functions have to be named for proper identification and to avoid confusion. An identifier should be clear, case sensitive, not start with a digit and can only contain letters, digits and underscores.
6. **Functions:** Are also statements that depict the IPO (input-process-output). There are built-in functions and user-defined functions in Python.
7. **Classes:** One of the features that make Python stand out is being an object-oriented language. Classes are the basis on which objects are created in Python.

8. **Comments:** These are started with the hash (#) character in a code.
9. **Statements:** These display the tasks that should be executed. An example is x = 31 + 45.

RULES OR GUIDES WHEN NAMING A PYTHON CLASS

Being an object-oriented programming language, Python is full of objects which are variables and functions. Classes are prototypes used to create objects. When naming classes, some basic guidelines have to be followed:

- Class names have a different naming pattern in Python known as CapWords convention and should be used when naming them.
- The 'camel_case' naming convention where underscores are not used between words.
- Built-in classes' names come in lowercase.
- Exception classes should have *Error* words.

NAMING CONVENTIONS FOR PYTHON READABILITY

A programmer does not write a code just for him/herself, other people and even the writer will read the code again and again. Proper documentation when writing the codes enhance readability for other users not just the writer of the code. In order to write readable codes naming conventions for each building block is required and certain naming styles have to be followed e.g.

- Function: lowercase words only; underscores are used to separate words.
- Variable: lowercase letter, word or groups of words; underscores are also used.
- Class: uppercase is used to start each word, camel_case style is used.
- Method: same as function.
- Constant: uppercase letter, word or groups of words; underscores are also used.
- Module: short lowercase word or words; underscores are used.

- Package: short lowercase word or words; no underscores are used.

KEYWORDS/RESERVED WORDS IN PYTHON

In Python, some keywords have been reserved with specific functions attached to them so they can't be used outside these functions and when naming classes, variables etc. some of these keywords with their functions are:

- break: break away from a loop.
- and: logical operator
- def: define
- del: delete
- elif: conditional statement if else
- false: comparison operator
- from: import certain parts
- if: conditional statement
- lambda: create anonymous function

These and many more are examples of reserved keywords in Python.

DATA TYPES IDENTIFICATION

Data is one of the pillars of code writing, tasks are executed as a result of the data processed into the program. So every value is seen as a data-type in object programming. However, there are different data types according to their functions in Python. Some of them are:

1. Integers: also known as numbers. They do not have limits and keep going as long as desired if there is sufficient amount of memory to contain all. Prefixes can be added to integers to indicate number bases apart from 10 like binary, octal and hexadecimal.
2. Floating numbers: floating point numbers come after an integer and are represented by decimal point or e.
3. Complex numbers: are statements written with the combination of real and imaginary numbers. An example is a + bj where 'a' is the real number while 'b' is the imaginary number.

4. List: is an important data type in Python and used extensively for many functions. It represents a sequence of data which could be the same or different written in order. Commas are used to separate variables in a list, then enclosed in a bracket [], ideally, a list contains variables which can be manipulated or changed. Operations can be executed on a single item or range of items within a list.
5. Tuple: is similar to a list, the difference between them is that values in a tuple are constant i.e. they cannot be changed. The items in a tuple are also separated with a comma but the tuple is enclosed in a parenthesis (). As with a list, operations can also be executed on a single item or range of items within a list.
6. Strings: is a list containing Unicode characters represented with single (‘’) or double quotes (“”).
7. Set: is also a form of list but the items are not arranged in an orderly manner. The items in a set are separated using commas and enclosed in braces {}.
8. Dictionary: is a data-type whose function is to store key-value pairs and be available when retrieval of the values is required. They are also enclosed in braces {} with each item in pairs.

CLASS MEMBERS IDENTIFICATION

In previous sections, classes have been discussed and defined as a blueprint for object creation in Python where all entities are regarded as objects. Object-oriented programming is an approach used to model situations in order to solve problems. When a real life task needs to be performed in programming, entities involved in the problem like cars, structures, relationships and other entities are represented as objects which are then modified in a bid to execute the task. Attributes of a particular object can be associated with its class. Classes are represented with the ‘class’ keyword with a colon (:) after it. There are class members which are the attributes allocated to each object. The basic class members are:

1. Class variable: is modifiable data value that can be shared by all objects within the class.

2. Method: this is a function allocated and distinct to a particular class definition.
3. Object: is the basic block of Python. Everything found in Python is seen as an object. All forms of variables and methods can be found in an object. An object is defined by the prototype given by its class.
4. Instance: is an object belonging to a certain class.
5. Instance variable: is a variable allocated to a certain instance and distinct to a particular method.
6. Data member: is a variable whether instance or class in which a class' data is stored.
7. Inheritance: this implies a class allocating characters to its sub-classes.
8. Instantiation: creation of an object for a particular class only.

WAYS TO ASSIGN VALUES TO VARIABLES

Variables are memory banks for storing values in Python. As soon as it is created, it holds its location in the memory bank. Various data types can be reserved in a variable depending on the preference of the programmer. Numeric, Boolean or set are examples of the types of data that can be stored in variables. To assign values to variables in Python, the equal sign (=) is used. The operand to which value is assigned i.e. the variable is on the left, then the equal to sign and lastly the value is on right like so

```
decade = 10
```

where decade is the variable, = is the equal to sign which assigns variable and 10 is the value assigned to the variable 'decade'. A single value can also be allocated to more than one variable at the same time.

RULES OF NAMING VARIABLES IN PYTHON

1. Variable naming is case sensitive i.e. uppercase O is different from lowercase o so attention should be paid when naming to

avoid mistakes and confusion.

2. Keep variable names short and clear. Do not just use a letter to represent a certain word as the program can be interpreted as something as, e.g. do not use p for pillow to avoid confusion.
3. Letters, underscores and numbers are the only characters that can be used in naming variables. A number cannot be used to start a variable name but letters and underscores can be used.
4. Avoid spaces in variables naming, use underscores instead.

DELETING VARIABLE IN PYTHON

When variables and functions become obsolete and are not usable anymore, Python deletes them automatically so as to free up memory space. They can also be removed by a programmer if they are no longer needed in a program. The del function is used to delete a variable manually. If a group of variables needs to be deleted, the action can be iterated until all that needs to be removed are deleted.

CONCEPT OF LITERALS

Simply put, literals are unprocessed information given in variable or constant. They are basically constant and do not change throughout a program's runtime. In Python, there are five types of literals:

1. String literals: generated by writing characters in single, double or triple quotes. The various forms of string literals are single line string literal and multi-line string literal.
2. Boolean literals: are conditional literals and represented by either true or false.
3. Numeric literals: represented by python numbers which are integers (positive or negative), floating points (fractions and decimals) and complex numbers (real and imaginary).
4. Special literals: is used to describe literals with no data input. It is represented by none . When no input is given, output is also blank.
5. Literal collections: this encompasses four different literals namely lust literals, tuple literals, set literals and dict literals. They are

all collections of data within Python, with differences in data sets, functions and how they are represented.

METHODS IN PYTHON

Methods are instructions that can be embedded into an object in order to execute a task. In other words, they are like action codes called on objects or classes. They are easy to integrate with a code, once written with the class or object to work on, output is given without delay. Methods should not be confused with functions; methods need an object to be able to perform while objects are not required in functions. As a result of this, methods can change an object's behavior and value since they are associated. There are various built-in methods used in Python namely:

1. **Methods for strings:** Used when datasets have the same value but different formats and the data has to be sorted. Methods used for this situation in strings are lower ( ), upper ( ), strip ( ), split ( ) etc.
2. **Methods for lists:** If a list needs to be modified, then methods are used. append (arg), remove (arg), count ( ) and clear ( ) are examples of methods used to modify a list.
3. **Methods for dictionaries:** Also used to modify the dictionary data-type. dict.keys ( ), dict.clear ( ) and dict.values ( ) are methods used when working on the dictionary data-type.

ELEMENTS TO DECLARE METHODS IN PYTHON

When using methods and functions, there are three basic categories. There is the built-in category that is already programmed with Python. Next category is user-defined methods which are created by Python users to assign certain methods to keywords and lastly anonymous functions. When declaring user defined methods, certain guidelines should be bore in mind:

1. The def keyword should be used to define the method and its name.

2. Assign tasks to be executed to the method and provide descriptions so the method can be clear.
3. Parameters should be added to a method. Enclose them in the method's parenthesis and end the line with a colon.
4. Add a return statement to the defined method so an output will be given. This helps to avoid getting none feedback.

To represent methods and functions, some elements have to be included so they can perform as desired. These elements are:

- **Docstrings:** These are used to describe what a function does. It comes immediately after the function header and is represented with triple quotes.
- **Arguments:** Are bits of information sent to a function in its call time. They come immediately after the function name within the parenthesis. There are four types of arguments namely keyword arguments, required arguments, default arguments and variable number of arguments.
- **Parameter:** Is a variable written inside the parenthesis in a function's definition.
- **Recursion:** Happens when a function is given the ability to call itself i.e. data can be looped through or iterated. Care has to be taken to prevent a situation where the loop keeps going on and eats into the processor's memory.

METHOD NAMING CONVENTION

In the previous section where naming of various concepts in Python was discussed, it has been asserted that proper naming is important for readability of codes. It also improves the quality and reusability of codes, other users will be able to interpret the codes easily if naming conventions are followed. In naming methods, some guidelines should be paid attention to:

- When naming methods, only lowercase words can be used.
- If more than a word is to be used, then an underscore should be used in between to enhance readability.
- Non-public methods names should be started with an underscore.
- When a method name needs to be mangled, two leading underscores should be used so Python mangling rules will be applied. If this is not

done, Python can mangle the method name subclasses and this causes confusion.

OBJECTS IN PYTHON

One of the most important features of Python is being an object-oriented programming language; this means almost all entities in Python are objects. It is the basic block of Python. It is the raw data that is processed to give out an output. It is a form of dataset with variables and methods embedded in it. An object is defined by the prototype given by its class. When a real world problem needs to be solved with Python, the entities like buildings, cars, interactions between employer and employee, teacher and students and all forms of entities can be represented as objects with their data inputted to give out an output. A class is the building block of an object; it contains all details about the object. An object can also be seen as an instance of a class i.e. an object belonging only to a particular class. Objects cannot stand alone; they need a class to define their variables and functions. Usually, the basic constituents of an object are:

- State: this shows the attributes of a particular object like age, breed, color etc.
- Behavior: this reflects the methods or functions embedded in an object e.g. go, come, bark, sleep etc. It also shows the response of an object to these functions.
- Identity: assigns a name to an object so interactions with other objects can be possible e.g. Frank, Dog, Car etc.

PACKAGE DEFINITION

Packages in Python are collections of multiple modules. Just as there are folders and subfolders in computer memories and files are arranged on the basis of certain criteria so they can be accessed easily, packages perform the same function in Python. It is used to bundle modules which is an array of multiple objects with classes and functions. It created a hierarchical arrangement of the modules to avoid collisions. It is a directory containing Python files and the package special file _init_py . Modules can be imported into a package file as long as the special file is present in the package file. Creating a package is similar to creating folders and subfolders in a computer. To create a package:

- Create a new folder and name it.
- Create a subfolder in the first folder and give it a desired package name.
- Create a _init_py file in the subfolder.
- Create or import modules and functions in the package.

MEANS OF ACCESSING CLASS MEMBERS

A class is made up of objects with variables and methods. Each of them is an attribute and can be tweaked to get a desired result. To access the attributes of a class, some in-built methods in python can be used e.g.

- getattr(): used when the attribute of a class needs to be checked.
- hasattr(): used to check if a class has a certain attribute.
- setattr(): used when a class needs more attributes.

A class' method can also be accessed by allocating an object to a particular class i.e. instantiating a class into an object. This enables the programmer to access the method of the particular class. Accessing methods of a class is done using the self parameter. This parameter is also useful when instance methods need to access the attributes and other methods and functions performed on a particular object. A class' method can also be accessed from another class by passing an instance of the called class to the calling class i.e. the latter class instance is passed to the former class instance.

USING OBJECTS IN PYTHON

As it has been established earlier, objects are basic data types in Python. They are used to represent real life concepts and consist of variables and methods. When a real world problem needs to be solved with Python, the entities like buildings, cars, interactions between employer and employee, teacher and students and all forms of entities can be represented as objects with their data inputted to give out an output. An object is an instance of a class; this means classes are made up of objects. Examples of objects and how they are used in codes are:

```
scoobie = dog ()
```

The object here is scoobie. Another example is

man = harry

dog = animal

dog = mammal

dog = canine

The last three examples assign three different attributes to the same object which is dog. This shows that objects can have different attributes. At the same time, an object can be instantiated into a class. An example is

door = door 1

door = door 2

door = door 3

The class here is the door with three object instances.

ACCESS SPECIFIERS IN PYTHON

When programs are written, there is always a possibility of unauthorized access to it. Access specifiers are responsible for denying unauthorized personnel access to data and prevent its use for functions other than what it was developed for. In other words, they specify the class variables that can be accessed in a program. There are three forms of access specifiers in Python:

- Public: these variables can be accessed from outside the class.
- Protected: these variables can be accessed from outside by a class derived from it. This can be enforced by using a single underscore before the variable name i.e. _variable-name.
- Private: this means the class variables cannot be accessed from outside, it can only be accessed from within the class. This specifier can be enforced by using double underscore for the variable name i.e. __variable-name.

ACCESS MODIFIERS IN PYTHON

These are similar to access specifiers but they are used to modify access to variables and functions of a class in object-oriented programming. They also limit access to variables from outside members. When access modifiers are put in place, data cannot be accessed by unauthorized bodies thereby

preventing it from being manipulated by these unauthorized bodies.as with specifiers, underscores are also used in modifiers to specify the type of access the programmer is willing to attach to a program. Granting of unauthorized access to data can occur as a result of inheritance. This is a very important concept in object-oriented programming and is considered one of its pillars.

Inheritance in programming is similar to inheritance in real life. A parent can transfer his/her genetic characteristics to their offspring and the offspring inherits these characteristics even though he/she possesses their unique characteristics as well. When a class inherits the attributes and variables of another class, inheritance can be said to have occurred. This leads to code reusability within and outside the program. The class from which attributes are transferred to another can be referred to as the base class while the class to whom attributes are transferred can be referred to as the child class. A base class can have several child classes and at the same time, a child class can inherit its attributes from several base classes. In the process of transferring data from base class to child class, data can get intercepted by unauthorized sources which could lead to its destruction or manipulation. Therefore, care has to be taken during inheritance to prevent data from getting into wrong hands. A demonstration of inheritance in Python is:

```
# parent class
    Class Car
    def __init__( self )
    print ( “Car is ready” )
    def whatisThis ( self ):
    print ( “Car” )
    def move ( self )
    print ( “move faster” )
    # child class
    Class Toyota ( Car ):
    def __init__( self ):
    # call super ( ) function
```

```
super ( ) . __init__( )
print ( “Toyota is ready” )
  def whatisThis ( self ):
print ( “Car” )
def move ( self )
print ( “accelerate faster” )
car = Toyota
car .whatisThis ( )
car .move ( )
car .accelerate ( )
```

Output

```
Car is ready
Toyota is ready
Toyota
Move faster
Accelerate faster
```

The program above shows Car as the parent class and Toyota as child class. It can be seen that Toyota inherits the attributes and functions of Car as seen in the move method.

Access modifiers can be executed in three forms namely:

Public access modifiers

All variables and functions in Python possess public characteristics by default so they can be accessed from both inside and outside the class. Example of public access class is

```
class car
def __init__( self, name, model ) :
self .name = name
```

```
self .model = model
```

This program can be interpreted to mean car class's attributes (name and brand) can be accessed from within and outside the class and modified. The above program variables can be modified as follows:

```
>>> c1 = car ( "Toyota", camry)
>>>c1 . model
camry
>>> c1 . model = corolla
>>> c1 .model
corolla
```

Another example of a program with public access variables and functions in a class:

```
class  employee:
        # constructor
        def  __init__(self, name, gender):
                # public data mambers
                self.employeeName=  name
                self.employeeGender=  gender
        # public member function
        def  displayGender(self) :
                # accessing public data member
                print("Gender: ", self.employeeGender)
# creating object of the class
obj=  Employee("Frank", Male)
```

```
# accessing public data member
print("Name: ", obj.employeeName)

# calling public member function of the class

obj.displayGender()
```

Output
Name: Frank
Gender: Male
This program shows employee name and gender are public data values. These values can be accessed from in the program because of this function.

PROTECTED ACCESS MODIFIER

These class members can only be accessible to subclasses extracted from it and other members within the class. In other words, a child class will be able to access a base class if the latter is protected. A class member can attain protected status by adding a single underscore as a prefix to its name. One of the public access program examples is illustrated below as a protected class member.

```
# super class

clas s  Employee:

  # protected data members

                    _name =  None

  _gender =  None

  _role =  None

  # constructor
  de f  __init__(self, name, gender, role):

                    self._name =  name

                    self._gender =  gender
```

```
                self._role =  role

    # protected member function

    de f  _displayGenderAndRole (self):

                # accessing protected data members

                print("Gender: ", self._gender)

                print("Role: ", self._role)

# derived class

  clas s  Manager(Employee):

   # constructor

   de f  __init__(self, name, gender, role):

                    Manager.__init__(self, name, gender, role)

         # public member function

   de f  displayDetails(self):

                    # accessing protected data members of super class

        print("Name: ", self._name)

         # accessing protected member functions of super class

                self._displayGenderAndRole ()

# creating objects of the derived class
  obj =  Manager("Frank", Male, "Analyst")

# calling public member functions of the class
obj.displayDetails()
```

Output

Name: Frank

Gender: Male

Role: Analyst

This program shows employee name, gender and role as protected data values, thus they can only be accessed from within the class and subclasses.

PRIVATE ACCESS MODIFIERS

Class members with private status can only be accessed from within the class. It is the most secure means of protecting data from unauthorized access and manipulation. A class member can attain protected status by adding double underscores as a prefix to its name. the program below illustrates our employee program with private accessed variables.

```
clas s  Employee:

  # private members

        __name =  None

       __gender =  None

        __role =  None

                  # constructor

      de f  __init__(self, name, gender, role):

                    self.__name =  name

               self.__gender =  gender

               self.__role =  role

          # private member function

de f  __displayDetails(self):

                   # accessing private data members

               print("Name: ", self.__name)

               print("Gender: ", self.__gender)
```

```
                print("Role: ", self.__role)

    # public member function

        de f  accessPrivateFunction(self):

                # accesing private member function

                self.__displayDetails()
# creating object

  obj =  Employee("Frank", Male, "Analyst")

# calling public member function of the class

obj.accessPrivateFunction()
```

Output

Name: Frank

Gender: Male

Role: Analyst

This program illustrates __name, __gender and __role as private class members and can only be accessed from within the class. The accessPrivateFunction ( ) calls for the accessibility of private members of the class.

An illustration showing a combination of the three forms of access modifiers is given below

```
# super class

  clas s  Parent:

    # public data member

    var1 =  None

    # protected data member

    _var2 =  None
```

```
# private data member
__var3 =  None
# constructor
de f  __init__(self, var1, var2, var3):
                    self.var1 =  var1
                    self._var2 =  var2
                    self.__var3 =  var3
                   # public member function
de f  displayPublicMembers(self):
                # accessing public data members
                print("Public Data Member: ", self.var1)
# protected member function
de f  _displayProtectedMembers(self):
                # accessing protected data members
                print("Protected Data Member: ", self._var2)
      # private member function
de f  __displayPrivateMembers(self):
                # accessing private data members
                print("Private Data Member: ", self.__var3)
# public member function
de f  accessPrivateMembers(self):
                # accessing private member function
```

```
                    self.__displayPrivateMembers()

# derived class

  clas s  Child(Parent):

    # constructor

          de f  __init__(self, var1, var2, var3):

         Parent.__init__(self, var1, var2, var3)

    # public member function

          de f  accessProtectedMembers(self):

         # accessing protected member functions of parent class
         self._displayProtectedMembers()

# creating objects of the derived class

  obj =  Child("Employees", Male, "Employees !")

# calling public member functions of the class

obj.displayPublicMembers()

obj.accessProtectedMemebers()

obj.accessPrivateMembers()

# Object can access protected member

print("Object is accessing protected member:", obj._var2)

 # object can not access private member, so it will generate
 Attribute error

#print(obj.__var3)
```

Output

Public Data Member: Employees

Protected Data Member: Male

Private Data Member: Employees !

The program shows a combination of public, protected and private access modifiers in a single program.

Summary

Python is a high level programming language with several attractive features making it desirable for programmers nowadays. The building blocks which act as the foundation of any program written were identified along with naming conventions for some of them and guidelines for naming them have been reviewed in detail in this chapter. These naming conventions are important for readability and ease of interpretation when multiple users are involved with a program. It has been established that variables are memory banks in which data is stored in Python. The importance of object-oriented programming and its peculiarity in Python cannot be overemphasized with every entity seen as objects in the programming language.

CHAPTER THREE

IMPLEMENTING OPERATORS IN PYTHON

Operators are certain symbols used to execute tasks on given variables and values to give results when writing programs in Python. They are used to manipulate values of operands. Without operators, certain tasks can't be carried out in Python. When using operators as functions, additional functions can be merged with the primary function of the operator. This concept is known as operator overloading. When this happens, it is observed that the same built in function of an operator performs differently with objects of different classes. An example is a situation where the operator + is expected to perform addition functions ordinarily. When the operator is overloaded, it will be able to merge lists or concatenate two strings depending on the objects in the classes. When functions are overloaded in a program, the codes can be reused without having to rewrite multiple codes with slight variations. It also improves code clarity and eliminates complexity. However, overloading should not be applied too much when writing programs to avoid confusion and inability to interpret codes properly. An example of a program where overloading is used is on the len() function on Buy class is:

```
class Buy:

def __init__(self, pencil, pad ) :

self .pencil = list(pencil)

self .buyer = buyer

def __len__( self ) :

return len (self .pencil)

buy = Buy ( [ 'pen', 'notepad', 'book' ], 'Python' )

print (len (buy))
```

Output

3

COMPARISON OPERATORS

They are used to compare two given values on both sides of the operand and deduce the relationship that exists between them. They are also known as relational operators. When operands are compared with these operators, the result given will be True or False depending on whether the condition is attached to the statement. There are six comparison operators used in Python:

- Less than (<): gives true result if the operand on the left is less than the operand on the right. If this condition is not met then the result is false. An example is

  4<10

  The output is true since 4 is less than 10.

- Greater than (>): gives true result if the operand on the left is greater than the operand on the right. If this condition is not met then the result is false. With the example above

  4>10

  The output is false since 4 is not greater than 10.

- Less than or equal to (<=): gives true result if the operand on the left is less than or equal to the operand on the right. If this condition is not met then the result is false.

  4 <= 10

  Output is true since 4 is less than 10, one of the conditions of the operator has been fulfilled.

- Greater than or equal to (>=): gives true result if the operand on the left is greater than or equal to the operand on the right. If this condition is not met then the result is false.

  4>=10

  Output is false since 4 is not greater than 10 or equal to 10.

- Equal to (===): gives true result if the operand on the left is equal to the operand on the right. If this condition is not met then the result is false.

4===10

Output is false as 4 is not equal to 10.

- Not equal to (!==): gives true result if the operand on the left is not equal to the operand on the right. If this condition is not met then the result is false.

4 !== 10

Output is true as 4 is not equal to 10.

Logical Operators

Logical operators are used in combining conditional statements. 'And' (true if both statements are true); 'or' (used if one of the statements are true) and 'not' (false if both statements are false) are all examples of logical operators. They are referred to as Boolean operators.

Identity operators

Identity operators compare the memory locations of two different objects. The two identity operators are 'is' (true if the variables on either sides of the operator point to the same object and false if not); 'is not' (false if the variables on either side of the operator point to the same object and false if not).

Membership operators

Membership operators are used to identify the relationship of a particular sequence in an object. Operators used to indicate membership are 'in' (true if a sequence with the specified value is found within an object); 'not in' (true if a sequence with the specified value is not found within an object).

Bitwise operators

Bitwise operators are used in comparison of binary numbers. They work on bits and execute tasks bit by bit. AND (&); OR (|); XOR (^); NOT (~); Zero fill left shift (<<) and Signed right shift (>>) are examples of bitwise operators used in python.

OPERATOR PRECEDENCE IN PYTHON

When using Python, there could be several operators in a program that needs to be executed. When this happens, there is a rule of precedence which determines the order in which each of the operators is carried out. The rule of precedence in Python is listed below with level of precedence in descending order i.e. from highest to lowest.

- Parentheses ()
- Exponent **
- Bitwise NOT +x, -x,~x
- Multiplication, division, floor division, modulus *, /, //, %
- Addition, subtraction +,-
- Bitwise shift operators <<,>>
- Bitwise AND &
- Bitwise XOR ^
- Bitwise OR |
- Comparison, identity and membership operators ==,!=,>,>=,<,<=,is, is not, in, not in
- Logical NOT not
- Logical AND and
- Logical OR or

Summary

Operators are an integral part of programming with Python and it has been examined critically in this chapter. The various types are evaluated along with the operators distinct to each type, their examples and applications are also discussed. Overloading of operators is good but should not be over-exploited to prevent a program from becoming cumbersome. Also, the guide of operator precedence should be properly followed so the task assigned to a program will be executed properly to get the desired result.

CHAPTER FOUR
CONDITIONAL AND LOOP CONSTRUCTS

As discussed earlier, conditions can be introduced into programs to direct the program to give different outcomes when variations exist in the conditions outlined. Boolean operators are used to denote True or False conditions in programs. Comparison and arithmetic operators are applied on variables which then give outputs as either True or False. So the condition attached to a certain action determines the action that will be performed, these programmed actions are known as Conditional constructs or statements. The loop construct suggests the execution of a code over and over again. With loops, you can repeat the same task a number of times. When loop construct is implemented, a sequence of steps keeps getting executed as long as the condition attached is being met. Once the loop breaks or the condition is no longer met, the sequence is not followed again and a new task is executed.

THE CONDITIONAL CONSTRUCT

In conditional constructs, there are usually two tasks to be executed with a condition attached. if the condition is met, only one of the tasks will be executed but if otherwise the other task will be performed. The if statement is used to symbolize conditional construct. A program where conditional construct is used takes the following form:

```
if condition_1:
statement_block_1
elif condition_2:
statement_block_2
else:
statement_block_ 3
```

The program above implies that if condition_1 is True, then statement_block_1 should be executed. If it comes False, then statement_block_2 will be executed. If condition_2 is True, then

statement_block_2 will be executed, if it is False, statement_block_3 will be implemented. The elif condition means 'if else' i.e. if condition_2 is True, statement_block_2 should be executed, else statement_block_3 should be executed.

In order to identify false objects in Python, the following are interpreted as False:

- Numerical zero values
- Boolean value False
- Empty lists and empty tuples
- Empty strings
- Empty dictionaries
- Special value None

THE LOOP CONSTRUCT

Loops are used when programs need to be iterated. There are two types of loops; definite and indefinite. While utilizing indefinite loops, the number of times the loop has to be repeated is not stated but with definite loops, it is stated when the loop starts. In Python, a definite loop is represented by for loop repeats a sequence for a given number of times while an indefinite loop is represented by while which continues repeating a sequence as long as the condition attached remains True.

Loop constructs have control statements which determine the next line of action in a sequence. The control statements used in Python are:

- Break statement: as the name implies, the loop breaks when the condition is no longer met or it has iterated over the given number of times. This causes the program to execute the next statement.
- Continue statement: this control statement instructs the loop to skip the rest of the code and at the same time check if the condition continues to be met before reiterating.
- Pass statement: pass statement instructs the loop not to execute and code on a particular statement.

The for loop is used when data types (dictionaries, lists, strings, tuples) needs to be iterated. Examples of for loops in action are:

Exit loop when x is 'pencil':

```
stationeries = [“pen”, “pencil”, “ruler”]
for x in stationeries:
print (x)
if x == “pencil”:
break
```

Do not print pencil

```
stationeries = [“pen”, “pencil”, “ruler”]
for x in stationeries:
if x == “pencil”:
continue
print (x)
```

To print each adjective for every stationeries

```
adj = [“yellow”, “2.B”, “sharpened”]
 stationeries = [“pen”, “pencil”, “ruler”]
for x in adj:
for y in stationeries:
print(x, y)
```

To repeat a program when it ranges between 10 and 15

```
 for a = 10 to 15
print (a)
```

To give the code to be executed when a loop has ended

```
for x in range (15):
print (x)
else:
print (“done!”)
```

The while loop continues to execute a sequence in as much as the condition attached remains True. A simple example is

```
 x = 65
while x<81
print ( x )
```

This means as long as x is less than 81, the loop will continue iterating.

CHAPTER FIVE

ARRAYS, ENUMS AND STRINGS IN PYTHON

In Python, data sometimes comes in the form of collections and each collection consists of different types of data, this chapter examines some of these data collections in detail. An array is a representation of a sequence of large data which are similar. An enum gives datasets in an enumeration type with iteration and comparison capabilities. Lastly, a string is also a dataset used to represent text rather than numbers.

ARRAYS

Although Python does not have an in-built function for arrays, Python Lists are modified to give arrays. An array can be illustrated as a classroom with each seat numbered with a value, where each student is seated and can be easily identified by knowing the number of their seat. Arrays are useful when large data of similar type needs to be represented by a single variable, this ability makes it a special variable. Multiple values are held in an array under a single variable and individual values can be accessed through an index number. When working with arrays in Python, a NumPy library has to be imported to be able to use lists as arrays. Since an array is a variable, its elements can be manipulated and modified as with a list.

ARRAY CREATION

To create an array, the array(data type,value_list) module has to be imported. This specifies the data type and list of each value.

```
# Python program to demonstrate creation of array
# importing “array” for array creations
Import array as arr
# creating an array with integer type
A = arr.array(‘i’, [1, 2, 3])
#printing original array
```

```
print (“The new created array is : ”, end = “ ”)
for I in range (0, 3):
print (a[],end = “ ” )
print ()
# creating an array with float type
b = arr.array(‘d’, [2.5, 3.2, 3.3])
# printing original array
print (“The new created array is : end = “ ”)
for I in range (0, 3):
print (b [i], end = “ ”)
```

Output

```
The new created array is : 1 2 3
The new created array is : 2.5, 3.2, 3.3
```

Enums

In Python, an enum is used in creating an enumeration of a set of symbolic members bound to unique constant values. It is a simple data collection type for representing names, enums names begin with uppercase letters and values used are singular. To create an enum, the enum module is used. An enum’s name is displayed using ‘name,’, type() is used to check the enum type and ‘repr()’ represents the object in the string.

```
import enum
#using enum class create enumerations
Class Months (enum.Enum)
Jan = 1
Feb = 2
Mar = 3
# print the enum member as a string
print (“The enum member as a string is : “, end=””)
print (Months.Jan)
# print the enum member as a repr
print (“the enum member as a repr is : “,end =””)
print (repr(Months.Feb))
#check type of enum member
print (“The type of enum member is : “,end =””)
```

```
print (type(Months.Feb))
#print name of enum member
print ("The name of enum member is : ",end ="")
print (Months.Feb.name)
```

Output

```
The enum member as a string is : Months.Jan
The enum member as a repr is :
The type of enum member is :
The name of enum member is : Feb
```

Strings

A string is a list containing Unicode characters represented with single (‘’) or double quotes (“”).

Unicode characters are symbols from all languages used in coding. Creating strings can be likened to assigning values to variables in Python. To assign a string to a variable, the ‘equal to’ sign is used as displayed below

```
b = "Hello"
print(b)
```

To access the elements in a string, square brackets are used

```
b = "Hello, World!"
print([b1])
```

Strings can also be looped through with the for function. For example, to loop through the letters in the word ‘pencil’:

```
for a in "pencil":
print(a)
```

These functions and many others can be performed on strings.

INHERITANCE AND POLYMORPHISM IN PYTHON

In chapter two, inheritance was briefly discussed with access modifiers. Inheritance in programming is similar to inheritance in real life as a new class is created from an existing class which transfers its characteristics to the new class. However, polymorphism is a concept which displays the

execution of a task in various forms. It gives the object the luxury of choosing the kind of function to be performed on it. Both concepts are examined in detail in this section.

Inheritance

A parent can transfer his/her genetic characteristics to their offspring and the offspring inherits these characteristics even though he/she possesses their unique characteristics as well. As an object oriented programming language, inheritance is necessary when writing programs. When a class inherits the attributes and variables of another class, inheritance can be said to have occurred. This leads to code reusability within and outside the program as well as reducing code length. The class from which attributes are transferred to another can be referred to as the base class while the class to whom attributes are transferred can be referred to as the child class. There are five types of inheritance namely:

1. Single inheritance
2. Multi-level inheritance
3. Multiple inheritance
4. Hybrid inheritance
5. Hierarchical inheritance

To create a base class and child class

```
class Person:
def__init__(self, fname,lname):
self.firstname = fname
self.lastname = lname
def printname (self):
print(self.firstname, self.lastname)
   #Use the Person class to create an object, and then execute the printname method:
a = Person(“Frank”, “Pen”)
a.printname()
class Employee(Person)
pass
```

POLYMORPHISM

Breaking down the word, “poly” means many while “morphism” means form. Therefore, polymorphism is a situation where a function name can be responsible for executing different tasks. As with inheritance where a child class inherits attributes from the base class, sometimes the attributes inherited do not actually fit into the child class so they are modified to fit and re-implemented, this process is known as method overriding or run-time polymorphism. Polymorphism provides flexibility of codes and allows objects to be used by functions without giving space for distinction across the classes. It allows different object classes to share the same method name. It is basically applied to functions and used in pattern designing. Types of polymorphism includes:

1. Compile-time polymorphism also known as method overloading
2. Run-time polymorphism also known as method overriding.

An example of method overriding program is

```
class Animal
def type(self):
print(various types of animals)
def age(self):
print(“Age of the animal.”)
class Leopard(Animal):
def age(self):
print(“Age of leopard.”)

class donkey(Animal):
def age(self):
print(“Age of donkey.”)
obj_animal = Animal()
obj_leopard = Leopard()
obj_donkey = Donkey()
obj_animal.type()
obj_animal.age()
obj_leopard.type()
obj_leopard.age()
```

```
obj_donkey.type()
obj_donkey.age()
```

Output

```
Various types of animals
Age of the animal.
Various types of animals
Age of leopard
Various types of animals
Age of donkey
```

Polymorphism differs from inheritance in various ways:

- Inheritance is applied to all classes while polymorphism is only applied to functions or methods.
- Inheritance allows code reusability while polymorphism is concerned with availing an object the opportunity of choosing the function it will execute during run-time (overrriding) and complile-time (overloading).
- A new class is derived from an existing class in inheritance while functions have multiple forms in polymorphism.

Summary

As an object oriented programming language, Python is filled with objects and some have to be represented in data sets of similar and dissimilar data types. Arrays, strings and enums are few of the many ways data sequences can be represented. Inheritance and polymorphism are important features of Python, the former is necessary for data transfer and creation of new classes while the latter is responsible for assigning different tasks to a single function.

CHAPTER SIX
EXCEPTION HANDLING

When executing a program in Python, errors are likely to occur and when this happens the program stops. Syntax error and exception error are the forms of error that can arise in Python. Syntax errors are common with beginner programmers; they are also referred to as parsing errors. Usually, a wrong syntax in the code triggers this error and when it is detected, the program terminates immediately. A simple program with a syntax error is

```
>>>while True print(“Hello world”)
File”<stdin>”,line 1
while True print(“Hello world”)
^
SyntaxError: invalid syntax
```

In this program, the arrow pointing upward shows the earliest point of error, i.e at the line before the arrow. It is caused by the absence of a colon (“:”) before the function print .

Exception error happens when the syntax of a code is correct but for some reason the code still gave an error response. The program is not terminated like in syntax error but the normal flow of the program is obstructed. Exception errors are unavoidable but they can be handled. A program resulting in exception error is

```
>>> 10* (1/0)
Traceback (most recent call last):
File”<stdin>”, line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
File”<stdin>”, line 1, in <module>
NameError: name ‘spam’ is not defined
>>>’2’+2
Traceback (most recent call last):
```

```
File"<stdin>", line 1, in <module>
  TypeError: Can't convert 'int' object to str implicitly
```

The error code displayed in the last line of the program shows the types of errors that occurred in the program. ZeroDivisionError, NameError and TypeError are found in this program but the string printed as exception type is the name of the built-in exception that occurred in the program.

To handle exceptions in Python, try and except keywords are used. To catch exceptions, try block is used while except block handles the exception. A finally block can also be added so the program it is attached to is executed without minding where the exception occurs. A simple program where exception error is handled using try and except blocks is displayed below:

```
a = [1, 2, 3]
try:
print "Second element = %d" %([1])
# Throws error since there are only 3 elements in                    array
print "Fourth element = %d" %([3])
Except IndexError:
print "An error occured"
```

Output

Second element = 2

An error occurred

In the program above, runtime error occurs when the print "Fourth element" command was given.

When handling exceptions, the except block can be executed more than one in a program with several exceptions but at most, only one handler will be executed. This situation is displayed below:

```
try :
a = 3
if a < 4:
# throws ZeroDivisonError for a = 3
b = a/(a-3)
# throws NameError if a >= 4
print "Value of b = " , b
   # note that braces ()are necessary here for multiple exceptions
   Except(ZeroDivisionError, NameError):
                Print "\nError Occurred and Handled"
```

Output

Error Occurred and Handled

If the value of (a) is changed to greater than or equal to 4, the output given will be

Value of b =
Error Occurred and Handled

This is as a result of the NameError that occurs when the value of b needs to be accessed. For the finally block to be executed, the try block has to be terminated. Else block is also used when the try block does not result in an exception. A syntax featuring all these exception handling keywords is:

```
# Python program to demonstrate exception
try:
a = 7//0 # raises divide by zero exception
print (a)

# handles zerodovision exception
except ZeroDivisionError:
print("Can't divide by zero")
else:
# execute if no exception
finally:
# this block is always executed
# regardless of exception generation
print('This is always executed')
```

Output

Can't divide by zero

This is always executed

This sample program displays the exception handling keywords discussed in action

EXCEPTION DEFINITION

As mentioned earlier, exceptions are errors that arise in programs even though the codes are syntactically correct. Exceptions are unavoidable but when handled properly, they do not lead to termination of programs. They are triggered by actions that Python cannot execute or perform. Examples of standard exceptions that are raised in Python include:

- Exception: the basic form of exception.

- StopIteration: this is triggered when the next()function of an iterator has no object to perform on.
- ArithmeticError: class of errors with roots from numeric operations.
- StandardError: all types of exceptions except StopIteration and SystemExit.
- ZeroDivisionError: occurs when numbers are divided by zero.
- IndexError: oc)curs when a sequence index number for objects is not found.
- NameError: occurs when the namespace is empty.
- SyntaxError: occurs as a result of error in Python syntax.
- SystemError: occurs when the Python interpreter fails to exit a program due to an internal fault.
- RuntimeError: occurs when the error code in a program cannot be attributed to any error category.

These and many others are built-in exceptions in Python.

ASSERTION IMPLEMENTATION

The assertion function is raised from the exception. It is a concept that allows a programmer to attach a condition to a code. A True condition implies execution of the next code while False stops the program from running. The assert statement is used to execute assertions in a program. When the condition comes False, *AssertionError* Exception is implemented. So when a program has passed the testing stage, assertion can be implemented as a sanity check. Assertions are implemented when a function is called to ascertain the input validity and after the function to check for the output. This establishes assertion as a debugging aid utilized in program self-checks, however, it cannot be used in handling run-time errors.

IMPORTANT NOTES IN ASSERTION IN PYTHON

Assertions are used to express confidence in a program. For example, when a division function is used in a program and you are sure the divisor cannot be zero, assertion condition is attached to the program to this effect. Boolean operators which evaluate whether a condition is true or false are also assertions. So if the program comes true, it continues to run smoothly but when it comes false the program stops and an *AssertionError* is triggered.
An example of a program with assert function is:

```
num = int(input('Enter a number: '))
assert num>= 0
print('You entered: ', num)
```

Output

```
Enter a number: 100
You entered 100
Enter a number: -10
Traceback (most recent call last):
    File "C:/python36/xyz.py", line 2, in <module>
    Assert num>=0
    AssertionErro
```

Another example of AssertionError using exception handling keywords is:

```
try:
num=int(input('Enter a number : '))
assert(num >=0), "Only positive numbers are accepted."
print(num)
except AssertionError as msg:
print(msg)
```

Output

Enter a number: -19

Only positive numbers accepted.

As a result of the except block in the program above, this program will not stop. The except block will pass the assert statement as an exception argument.

Overall/general principles of Testing and Writing Programs

After writing a program in Python, it is subjected to tests of various types to determine its usability. These tests form a crucial part of activities done after writing a program. However, there are certain guidelines to performing testing on programs. They are called principles and are briefly explained below:

1. Testing detects presence of defects not their absence, i.e. when testing is done, it lowers the possibility of finding defects later on although no defects do not prove correctness of a program.
2. Exhaustive testing is not possible. Everything cannot be tested so be prudent when planning for testing and make optimal use of time and resources so the test can cover the most important areas.
3. Early testing is important. Testing a program earlier helps save time, money and other resources. This ensures defects are discovered as soon as possible before bugs eat into a program and also prevents the need to rewrite and rerun codes. Therefore, it should be started as soon as the requirements of a defect are defined.
4. Defects cluster together in a small portion of a program according to Pareto Principle which states that 80% of defects are found in 20% of modules. This means once a bug is found in a module, it is likely to find more bugs in it. Once the vulnerable areas are identified, tests should be focused there so other defects can be identified although this might lead to resistance of the module to testing later on.
5. The pesticide paradox sets in, this implies that when the sane tests are run regularly on a program, it grows resistant to the tests and defects are not found even though they are present. This is similar to the resistance pests grow to pesticide when it has been applied several times; the pest develops immunity against the pesticide and renders it useless after some time. Therefore, tests should be reviewed and modified regularly to avoid pesticide paradox.

6. Absence of errors in a program is a fallacy. Even if a program is 99% free of bugs, it still might not be usable if it does not meet the need for which it was written.
7. Testing is context dependent, this means the use of a program determines the kind of testing that will be done on it.

When writing codes, certain guidelines or principles also have to followed so as to yield great codes:

1. A code is read a million times but written once.
2. Follow a coding standard.
3. Independence of code.
4. Easily testable.
5. Keep it Simple
6. Resilience of code
7. Manage system dependencies.

HOW DOES ONE DESIGN A PROGRAM BEFORE CODING?

When a program needs to be written, an outline showing the expectations and required input should be developed in order to yield great codes and avoid wasting of resources. Developing a plan helps to avoid getting stuck midway and wasting time on the program. To design a program before coding is started, the following steps need to be followed:

- Ensure your code is solving a problem, this is the foundation of building a program. Have a certain goal of what you want the program to address.
- With the problem that needs to be solved figured out, outline the features the program needs to have to be able to address the problem.
- Develop a step by step guide of building the sub-components of the program.
- With the blueprint developed in the earlier steps, develop the optimal means of performing the task and avoid repeating codes.
- Design your interface or prototype with the blueprint developed; Adobe XD can be used for this purpose. With this code writing can be started.

PROGRAMMING BEING STRUCTURED

In program writing, there are three main structures which serve as building blocks in programming. They are:

- Sequence: is a series of events which are completed in a specific order. An action leads to a next action is a predetermined order in sequence structure. The actions are completed in the set order without skipping any when the sequence is set.
- Selection: is a bit different from sequence as a condition is attached to the program. The outcome of the condition determines the next action in the program.
- Loop: keeps repeating a code until the attached condition is no more fulfilled. The type of loop in the program determines when it stops repeating.

PROGRAM COMPILATION AND COMPILATION ERROR

In Python, source code (.py file) is compiled into a simpler form known as bytecode. Source codes might be difficult to interpret at times so they are compiled into lower-level and platform-independent bytecode forms. The bytecode(.pyc files) undergoes reloading when the source file is also updated. Program compilation is useful when a program needs to be distributed to other python users; all that is needed is to send the .py files or pyc files. Compiling bytecode is an automatic process with the code used to interpret (Python Virtual Machine) is already installed with Python software. Compilation errors occur when Python cannot run a program as a result of instruction malfunction.

Program compiler errors

These errors arise as a result of not writing a syntax properly thereby hindering the program from starting. When a compiler error has been detected, the program should check for other omissions and typos in the program. Situations which might lead to compiler error include:

- Not declaring the value of a variable before printing.
- Using keywords as variable name.

- Skipping required semicolons.
- Skipping parenthesis

An example of compiler error is

```
// program to illustrate
// syntax error
#include<stdio.h>
  void main()
  {
  int a = 5
  int b = 10
// semicolon missed
printf("%d", (a, b))
}
```

Output:

error: expected ';' before '}' token

The compiler error in this program is as a result of not adding parentheses before the closing brace.

Another example of compiler error arising as from syntax error is

```
a = int(input('Enter a number:'))
if a%4==0
print('You have entered an even number.')
else:
print('You have entered an odd number.')
Output
C: Python34Scripts>python error .py
File "error .py", line 3
If a%4==0
^
SyntaxError: invalid syntax
```

The program above gave an error due to the missing colon (:) at the end of the if statement. This hinders the program from running.

RUNTIME ERRORS

These are errors encountered during a program's runtime after code compilation. They are hard to detect because the program does not distinguish the line causing the error. they can be found by proofreading the code line by line or testing the program with a debugger. When runtime error occurs, the program misbehaves, displays an exception dialog box, gives a wrong output, freezes the application or shutdown totally. They are caused by variable name misspelling and other errors of commission. Inputting wrong information into the application or computer hardware problem can also raise runtime error. In essence, error of commission and error of omission causes runtime error. Common examples of runtime error are division of numbers by zero, deducing square roots of negative integers etc.

DEBUGGING IN PROGRAMS

After writing a program, debugging is done to detect and deal with the various errors discussed earlier. So, a debugger serves as a tool used to search through the nooks and crannies of a program, the variables, source codes and every core area of the program for defects, errors and every other problem in a program and also solve them. It acts by performing all actions in a program, executing the tasks line by line and repeating these actions to detect abnormalities and bugs in the program through the development process.

There is a built-in debugging tool in Python called pdb . It comes with the Python program and has all functions needed to detect errors and solve them. Its features can also be extended using ipdb from IPython. To use pdb , it has to be called into the code to be debugged, i.e.

```
Import pdb; pdb.set_trace()
```

Some programmers also use the print function to show a program's behavior and spot bugs rather than default debuggers. To stop debugging, the quit function is called by pressing "q" followed by ENTER key.

Summary

Various types of errors that might arise during code writing and execution are discussed in this chapter. Exception handling is done with the try, except, else and finally functions.

Assertions are implemented when a function is called to ascertain the input validity and after the function for check for the output, it is raised from

exception. In program testing, pesticide paradox should be avoided and also, testing should be program-context dependent. These and other principles of testing are necessary for detecting defects. Code writing principles like keeping it simple, resilient and of good standard needs to be followed to turn out great codes. Errors should be detected early in programs to avoid rewriting and save time and resources. Each program should be developed towards solving a specific program so proper planning should be implemented.

About the Author

Gary Elmer is a writer dedicated to the world of technology and programming software innovations. He has written a number of books that proved to be very helpful in breaking down the intricacies of the world of technology to readers. This guide to Python programming language is another entry in the series and has gathered numerous positive reviews