Internet



of

Things

Contents

|  |  |  |
| --- | --- | --- |
| **7.** | **Introduction to Python Programming** | **..............................................................163** |
|  | 7.1 Introduction | 163 |
|  | 7.2 Input-Output Statements | 163 |

7.3. [Operators 164](#_TOC_250038)

* 1. [Data Types 166](#_TOC_250037)

7.5. [Decision Making 169](#_TOC_250036)

7.6. [Loops 171](#_TOC_250035)

7.7. [Control Statements 174](#_TOC_250034)

7.8. [Functions 175](#_TOC_250033)

7.9. [Modules 178](#_TOC_250032)

7.10. [Exception Handling 180](#_TOC_250031)

7.11. [File Handling 182](#_TOC_250030)

7.12. [Reading Images 185](#_TOC_250029)

7.13. [Networking 186](#_TOC_250028)

7.14.[Conclusion 189](#_TOC_250027)

**7**

**INTRODUCTION TO PYTHON PROGRAMMING**

##### 7.1. Introduction

Python is a general-purpose interpreted, interactive, object-oriented, and high-level programming language. Python source code is available under the GNU General Public License (GPL) and it is now maintained by a core development team at the National Research Institute. Python is a versatile language which is easy to script and easy to read. It doesn’t have strict rules for syntax. Python is open source and it forms a strong backbone to build large applications. We can run Python from a Graphical User Interface (GUI) environment if we have a GUI application on our system that supports Python. IDLE is the Integrated Development Environment (IDE) for Python. Its installation comes with integrated development environment for programming. Other examples for Python IDE are Spyder, PyCharm etc. All the programs in this book follow Python 3.4.3 for Windows.

##### 7.2. Input-Output Statements

The function used to print output on a screen is the *print* statement where we can pass zero or more expressions separated by commas. The *print* function converts the expressions we pass into a string and writes the result to standard output.

**Example 1**

>>>print(”Learning Python is fun and enjoy it.”)

This will produce the following output on the screen.

Learning Python is fun and enjoy it.

In Python, we need not declare the type of the variable as in C or C++. The type of the variable depends on the value assigned to it. Hence we can simply assign a value to a variable.

**Example 2**

>>>a=2

>>>print(“The value of a is”,a)

**164** | Internet of Things

This will produce the following output on the screen. The value of a is 2

The input([prompt]) function assumes the input is a valid Python expression and returns the evaluated result to us.

**Example**

n = input(“Enter a number “) print(“The number is: “, n)

**Output**

Enter a number: 5 The number is: 5

##### 7.3. Operators

Python supports various operators in like arithmetic operators, comparison (relational) operators, assignment operators, logical operators, bitwise operators, membership operators and identity operators. Table 7.1 shows the arithmetic operators in Python.

**Table 7.1: Arithmetic Operators in Python**

|  |  |  |
| --- | --- | --- |
| **Operator** | **Operation** | **Description** |
| + | Addition | Adds values on either side of the operator. |
| - | Subtraction | Subtracts right hand operand from left hand operand. |
| \* | Multiplication | Multiplies values on either side of the operator. |
| / | Division | Divides left hand operand by right hand operand. |
| % | Modulus | Divides left hand operand by right hand operand and returns remainder. |
| \*\* | Exponent | Performs exponential (power) calculation on operators. |
| // | Floor Division | The division of operands where the result is the quotient in which the digits after the decimal point are removed. |

Comparison operators compare the values on either sides of them and decide the relation among them. They are also called relational operators. Python supports the following relational operators. Table 7.2 shows the comparison or relational operators in Python.

**Table 7.2: Comparison Operators in Python**

|  |  |
| --- | --- |
| **Operator** | **Description** |
| == | If the values of two operands are equal, then the condition becomes true. |
| != | If values of two operands are not equal, then condition becomes true. |
| > | If the value of left operand is greater than the value of right operand, then condition becomes true. |

Introduction to Python Programming | **165**

|  |  |
| --- | --- |
| < | If the value of left operand is less than the value of right operand, then condition becomes true. |
| >= | If the value of left operand is greater than or equal to the value of right operand, then condition becomes true. |
| <= | If the value of left operand is less than or equal to the value of right operand, then condition becomes true. |

Python provides various assignment operators. Various shorthand operators for addition, subtraction, multiplication, division, modulus, exponent and floor division are also supported by Python. Table 7.3 provides the various assignment operators.

**Table 7.3: Assignment Operators in Python**

|  |  |
| --- | --- |
| **Operator** | **Description** |
| = | Assigns values from right side operands to left side operand. |
| += | It adds right operand to the left operand and assign the result to left operand. |
| -= | It subtracts right operand from the left operand and assign the result to left operand. |
| \*= | It multiplies right operand with the left operand and assign the result to left operand. |
| /= | It divides left operand with the right operand and assign the result to left operand. |
| %= | It takes modulus using two operands and assign the result to left operand. |
| \*\*= | Performs exponential (power) calculation on operators and assign value to the left operand. |
| //= | It performs floor division on operators and assign value to the left operand. |

Bitwise operator works on bits and performs bit by bit operation. The following are the bitwise operators supported by Python. Table 7.4 gives a description of bitwise operators in Python.

**Table 7.4: Bitwise Operators in Python**

|  |  |  |
| --- | --- | --- |
| **Operator** | **Operation** | **Description** |
| & | Binary AND | Operator copies a bit to the result if it exists in both operands. |
| | | Binary OR | It copies a bit if it exists in either operand. |
| ^ | Binary XOR | It copies the bit if it is set in one operand but not both. |
| ~ | Binary Ones Complement | It is unary and has the effect of ‘flipping’ bits. |
| << | Binary Left Shift | The left operands value is moved left by the number of  bits specified by the right operand. |
| >> | Binary Right Shift | The left operands value is moved right by the number  of bits specified by the right operand. |

**166** | Internet of Things

Table 7.5 shows the various logical operators supported by Python language.

**Table 7.5: Logical Operators in Python**

|  |  |  |
| --- | --- | --- |
| **Operator** | **Operation** | **Description** |
| and | Logical AND | If both the operands are true then condition becomes true. |
| or | Logical OR | If any of the operands are true then condition becomes true. |
| not | Logical NOT | Used to reverse the logical state of its operand. |

Python’s membership operators test for membership in a sequence, such as strings, lists, or tuples. There are two membership operators as explained below in Table 7.6. Strings, lists and tuples will be covered in the next section.

**Table 7.6: Membership Operators in Python**

|  |  |
| --- | --- |
| **Operator** | **Description** |
| in | Evaluates to true if the variables on either side of the operator point to the same object and false otherwise. |
| not in | Evaluates to true if it does not finds a variable in the specified sequence and  false otherwise. |

Identity operators compare whether two elements points to the same memory locations.

There are two Identity operators as shown in below Table 7.7.

**Table 7.7: Identity Operators in Python**

|  |  |
| --- | --- |
| **Operator** | **Description** |
| is | Evaluates to true if the variables on either side of the operator point to the same object and false otherwise. |
| is not | Evaluates to false if the variables on either side of the operator point to the same object and true otherwise. |

##### 7.4. Data Types

There are 6 data types in Python. They are numbers, string, list, tuple, set and dictionary.

***Numbers***

Number data types store numeric values. Number objects are created when you assign a value to them. For example a = 1, b = 20. Python supports four different numerical types.

* + - int (signed integers)
    - long (long integers, they can also be represented in octal and hexadecimal)

Introduction to Python Programming | **167**

* + - float (floating point real values)
    - complex (complex numbers)

###### String

Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows for either pairs of single or double quotes. Subsets of strings can be taken using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the string and ending at -1.

**Example Program**

str = ‘Welcome to Python Programming ‘ print(str) # Prints complete string

print(str[0]) # Prints first character of the string print(str[11:17]) # Prints characters starting from 11th to 17th print(str[11:]) # Prints string starting from 11th character

***Output***

Welcome to Python Programming W

Python

Python Programming

There are several built-in functions for string. Here we will see one of the commonly used built-in function split().

**split(str=””, num=string.count(str)) -** Returns a list of all the words in the string, using str as the separator (splits on all whitespace if left unspecified), optionally limiting the number of splits to num.

**Example Program**

#Demo of split(str=””,num=string.count(str)) s=”Python programming is fun” print(s.split(‘ ‘)) s=”Python\*programming\*is\*fun” print(s.split(‘\*’)) s=”Python\*programming\*is\*fun” print(s.split(‘\*’,2))

***Output***

[‘Python’, ‘programming’, ‘is’, ‘fun’]

[‘Python’, ‘programming’, ‘is’, ‘fun’]

[‘Python’, ‘programming’, ‘is\*fun’]

**168** | Internet of Things

###### List

List is an ordered sequence of items. It is one of the most used data type in Python and is very flexible. All the items in a list do not need to be of the same type. Items separated by commas are enclosed within brackets [ ]. The values stored in a list can be accessed using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the list and ending with -1.

**Example Program**

first\_list = [‘abcd’, 147, 2.43, ‘Tom’, 74.9] print(first\_list) # Prints complete list print(first\_list[0]) # Prints first element of the list

print(first\_list[1:3]) # Prints elements starting from 2nd till 3rd print(first\_list[2:]) # Prints elements starting from 3rd element

***Output***

[‘abcd’, 147, 2.43, ‘Tom’, 74.9]

abcd

[147, 2.43]

[2.43, ‹Tom›, 74.9]

###### Tuple

A tuple is another sequence data type that is similar to the list. A tuple consists of a number of values separated by commas. The main differences between lists and tuples are lists are enclosed in square brackets ( [ ] ) and their elements and size can be changed, while tuples are enclosed in parentheses ( ( ) ) and cannot be updated. Tuples can be considered as read-only lists.

**Example Program**

first\_tuple = (‘abcd’, 147, 2.43, ‘Tom’, 74.9) print(first\_tuple) # Prints complete tuple print(first\_tuple[0]) # Prints first element of the tuple

print(first\_tuple[1:3]) # Prints elements starting from 2nd till 3rd print(first\_tuple[2:]) # Prints elements starting from 3rd element

***Output***

(‘abcd’, 147, 2.43, ‘Tom’, 74.9)

abcd

(147, 2.43)

(2.43, ‘Tom’, 74.9)

***Set***

Set is an unordered collection of unique items. Set is defined by values separated by comma inside braces { }. It can have any number of items and they may be of different types (integer,

Introduction to Python Programming | **169**

float, tuple, string etc.). Items in a set are not ordered. Since they are unordered we cannot access or change an element of set using indexing or slicing. We can perform set operations like union, intersection, difference on two sets. Set have unique values. They eliminate duplicates. The slicing operator [] does not work with sets. An empty set is created by the function set().

**Example Program**

# Demo of Set Creation

s1={1,2,3} #set of integer numbers print(s1)

s2={1,2,3,2,1,2}#output contains only unique values print(s2)

s3={1, 2.4, ‘apple’, ‘Tom’, 3}#set of mixed data types print(s3)

***Output***

{1, 2, 3}

{1, 2, 3}

{1, 3, 2.4, ‘apple’, ‘Tom’}

###### Dictionary

Dictionary is an unordered collection of key-value pairs. It is generally used when we have a huge amount of data. We must know the key to retrieve the value. In Python, dictionaries are defined within braces {} with each item being a pair in the form key:value. Key and value can be of any type. Keys are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object. Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([]).

**Example Program** dict={1:”one”,2:”two”} print(dict)

***Output***

{1:”one”,2:”two”}

##### 7.5. Decision Making

Decision making is required when we want to execute a code only if a certain condition is satisfied. The if…elif…else statement is used in Python for decision making.

***if* statement**

***Syntax***

if test expression: statement(s)

**170** | Internet of Things

Here, the program evaluates the test expression and will execute statement(s) only if the text expression is True. If the text expression is False, the statement(s) is not executed. In Python, the body of the if statement is indicated by the indentation. Body starts with an indentation and ends with the first unindented line. Python interprets non-zero values as True. None and 0 are interpreted as False.

**Example Program**

num =int(input(“Enter a number: “)) if num == 0:

print(“Zero”)

print(“This is always printed”)

***Output***

Enter a number: 0 Zero

This is always printed

***if….else* statement**

***Syntax***

if test expression:

Body of if else:

Body of else

**Example Program**

num =int(input(“Enter a number: “)) if num >= 0:

print(“Positive Number or Zero”) else:

print(“Negative Number”)

***Output***

Enter a number: 5 Positive Number or Zero ***if...elif...else* statement**

***Syntax***

if test expression:

Body of if

elif test expression:

Body of elif else:

Body of else

Introduction to Python Programming | **171**

The elif is short for else if. It allows us to check for multiple expressions. If the condition for if is False, it checks the condition of the next elif block and so on. If all the conditions are False, body of else is executed. Only one block among the several if...elif...else blocks is executed according to the condition.

**Example Program**

num =int(input(“Enter a number: “)) if num > 0:

print(“Positive number”) elif num == 0:

print(“Zero”) else:

print(“Negative number”)

***Output***

Enter a number: 5 Positive Number

##### 7.6. Loops

Python provides various control structures that allow for repeated execution. A loop statement allows us to execute a statement or group of statements multiple times.

***for* loop**

The for loop in Python is used to iterate over a sequence (list, tuple, string) or other objects that can be iterated.

***Syntax***

for item in sequence: Body of for

Here, item is the variable that takes the value of the item inside the sequence of each iteration. The sequence can be list, tuple, string, set etc. Loop continues until we reach the last item in the sequence. The body of for loop is separated from the rest of the code using indentation.

**Example Program 1**

#Program to find the sum of all numbers stored in a list # List of numbers

numbers = [2,4,6,8,10]

# variable to store the sum sum = 0

# iterate over the list for item in numbers:

**172** | Internet of Things

sum = sum+item # print the sum

print(“The sum is”,sum)

***Output 1***

The sum is 30

**Example Program 2**

flowers = [‘rose’, ‘lotus’, ‘jasmine’] for flower in flowers:

print(‘Current flower:’, flower)

***Output 2***

Current flower: rose Current flower: lotus Current flower: jasmine

***for* loop with *else* statement**

Python supports to have an else statement associated with a loop statement. If the ***else*** statement is used with a ***for*** loop, the **else** statement is executed when the loop has finished iterating the list. A break statement can be used to stop a *for* loop. In this case, the else part is ignored. Hence, a *for* loop’s *else* part runs if no break occurs. Break statement will be discussed in the next section.

**Example Program**

#Program to find whether an item is present in the list list=[10,12,13,34,27,98]

num = int(input(“Enter the number to be searched in the list: “)) for item in range(len(list)):

if list[item] == num:

print(“Item found at: “,(item+1)) break

else:

print(“Item not found in the list”)

***Output***

Enter the number to be searched in the list:34 Item found at:4

***while loop***

The while loop in Python is used to iterate over a block of code as long as the test expression (condition) is true. We generally use this loop when we don’t know the number of times to iterate in advance.

Introduction to Python Programming | **173**

***Syntax***

while test\_expression: Body of while

In while loop, test\_ expression is checked first. The body of the loop is entered only if the test\_expression evaluates to True. After one iteration, the test expression is checked again. This process is continued until the test\_expression evaluates to False. In Python, the body of the while loop is determined through indentation. Body starts with indentation and the first unindented line shows the end. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

**Example Program**

# Program to find the sum of first N natural numbers n =int(input(“Enter the limit: “))

sum=0 i=1

while i<=n:

sum=sum+i i=i+1

print(“Sum of first”, n, “natural numbers is “, sum)

***Output***

Enter the limit: 5

Sum of first 5 natural numbers is 15

***while* loop with *else* statement**

If the else statement is used with a while loop, the else statement is executed when the condition becomes false. while loop can be terminated with a break statement. In such case, the else part is ignored. Hence, a while loop’s else part runs if no break occurs and the condition is false.

**Example Program**

# Demo of While with else count=1

while count<=3:

print(“Python Programming”) count=count+1

else:

print(“Exit”) print(“End of Program”)

***Output***

Python Programming

**174** | Internet of Things

Python Programming Python Programming Exit

End of Program

##### 7.7. Control Statements

Control statements change the execution from normal sequence. Loops iterate over a block of code until test expression is false, but sometimes we wish to terminate the current iteration or even the whole loop without checking test expression. The break and continue statements are used in these cases. Python supports the following three control statements.

1. break
2. continue
3. pass

***break* statement**

The break statement terminates the loop containing it. Control of the program flows to the statement immediately after the body of the loop. If it is inside a nested loop (loop inside another loop), break will terminate the innermost loop. It can be used with both for and while loops.

**Example Program**

#Demo of break statement in Python for i in range(2,10,2):

if i==6: break print(i)

print(“End of Program”)

***Output***

2

4

End of Program

The continue statement is used to skip the rest of the code inside a loop for the current iteration only. Loop does not terminate but continues on with the next iteration. Continue returns the control to the beginning of the loop. The continue statement rejects all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop. The **continue** statement can be used in both *while* and *for* loops.

**Example Program**

# Demo of continue in Python for letter in ‘abcd’:

if letter ==’c’: continue; print(letter)

Introduction to Python Programming | **175**

***Output***

a b d

***pass* statement**

In Python programming, pass is a null statement. The difference between a comment and pass statement in Python is that, while the interpreter ignores a comment entirely, pass is not ignored. But nothing happens when it is executed. It results in no operation.

It is used as a placeholder. Suppose we have a loop or a function that is not implemented yet, but we want to implement it in the future. The function or loop cannot have an empty body. The interpreter will not allow this. So, we use the pass statement to construct a body that does nothing.

**Example**

for val in sequence: pass

##### 7.8. Functions

The following specifies simple rules for defining a function.

* + - Function block or Function header begins with the keyword def followed by the function name and parentheses ( ( ) ).
    - Any input parameters or arguments should be placed within these parentheses. We can also define parameters inside these parentheses.
    - The first string after the function header is called the docstring and is short for documentation string. It is used to explain in brief, what a function does. Although optional, documentation is a good programming practice.
    - The code block within every function starts with a colon (:) and is indented.
    - The return statement [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None. We can also write a function without return statement.

***Syntax***

def functionname( parameters ): “function\_docstring” function\_suite

return [expression]

**Example Program with Return Statement**

def sum\_of\_two\_numbers(a,b): #Function Header with 2 parameters a and b

**176** | Internet of Things

“This Function is to find the sum of two numbers” #Docstring sum=a+b

return sum

**Example Program without Return Statement**

def sum\_of\_two\_numbers(a,b): #Function Header with 2 parameters a and b “This Function is to find the sum of two numbers” #Docstring

sum=a+b print(“sum=”,sum)

After defining a function, we can call the function from another function or directly from the Python prompt. The order of the parameters specified in the function definition should be preserved in function call also.

**Example Program for Function Calling with Return Statement**

#Main Program Code

a = int(input(“Enter the first number:”)) b = int(input(“Enter the second number:”))

s = sum\_of\_two\_numbers(a,b)#Function calling print(“Sum of”,a,”and”,b,”is”, s)

***Output***

Enter the first number:4 Enter the second number:3 Sum of 4 and 3 is 7

**Example Program for Function Calling without Return Statement**

#Main Program Code

a = int(input(“Enter the first number:”)) b = int(input(“Enter the second number:”))

s = sum\_of\_two\_numbers(a,b)#Function calling

***Output***

Enter the first number:4 Enter the second number:3 Sum of 4 and 3 is 7

##### Variable Scope in Python

Global variables are those variables declared out of any function, but can be accessed inside as well as outside the function. Parameters and variables defined inside a function is not visible from outside. Hence, they have a local scope. Lifetime of a variable is the period throughout which the variable exists in the memory. The lifetime of variables inside a function is as long

Introduction to Python Programming | **177**

as the function executes. All variables in a program may not be accessible at all locations in that program. This depends on where we have declared a variable. The scope of a variable determines the portion of the program where we can access a particular identifier. They are destroyed once we return from the function. Hence, a function does not remember the value of a variable from its previous calls.

**Example Program**

a=10

def value\_change(a): #Function Header a=100

print(“Value inside function=”,a) #prints the value of a inside function return

#Main Program Code value\_change(a)#Function calling

print(“Value outside function=”,a) #prints value of a outside function

***Output***

Value inside function= 100 Value outside function= 10

**Function with more than one return value**

Python has a strong mechanism of returning more than one value at a time. This is very flexible when the function needs to return more than one value. Instead of writing separate functions for returning individual values, we can return all the values within same function. The following shows an example for function returning more than one value.

**Example Program**

#Python Function Returning more than One value def calc(a,b):

sum=a+b diff=a-b prod=a\*b quotient=a/b

return sum, diff, prod, quotient a=int(input(“Enter first number:”)) b=int(input(“Enter second number:”)) s,d,p,q=calc(a,b)

print(“Sum=”,s) print(“Difference=”,d) print(“Product=”,p) print(“Quotient=”,q)

**178** | Internet of Things

**Output**

Enter first number:10 Enter second number:5 Sum= 15

Difference= 5

Product= 50

Quotient= 2

##### 7.9. Modules

Modules refer to a file containing Python statements and definitions. We use modules to break down large programs into small manageable and organized files. Further, modules provide reusability of code. A module can define functions, classes and variables. A module can also include runnable code. Python has a lot of standard modules (built-in modules) available.

##### Creating Modules

We can define our most used functions in a module and import it, instead of copying their definitions into different programs. A file containing Python code, for e.g.: prime.py, is called a module and its module name would be prime. Let us create a module for finding the sum of two numbers. The following code creates a module in Python. Type the code and save it as test.py.

**Example**

# Python Module example “””This program adds two

numbers and return the result””” def sum(a, b):

result = a + b return result

Here we have a function sum() inside a module named test. The function takes in two numbers and returns their sum.

***import* Statement**

We can use any Python source file as a module by executing an import statement in some other Python source file. User-defined modules can be imported the same way as we import built-in modules.

We use the import keyword to do this. The import has the following syntax.

import module1[, module2[,... moduleN]

When the interpreter encounters an import statement, it imports the module if the module is present in the search path. A search path is a list of directories that the interpreter

Introduction to Python Programming | **179**

searches before importing a module. For example, to import the module test.py, you need to put the following command at the top of the script.

**Example**

import test print(test.sum(2,3))

***Output***

5

A module is loaded only once, regardless of the number of times it is imported. This prevents the module execution from happening over and over again if multiple imports occur.

***import* with renaming**

We can import built-in or user-defined modules with an alias name. The following code shows how the built-in module math can be imported using an alias name.

**Example Program**

import math as m

print(“The value of pi is”, m.pi)

**Output**

The value of pi is 3.141592653589793

We have renamed the math module as m. This can save typing time in some cases. It is important to note that the name math is not recognized in our scope. Hence, math.pi is invalid, m.pi is the correct implementation.

***from…import* statement**

We can import specific names form a module without importing the module as a whole. The module math contains a lot of built-in functions. But if we want to import only the pi function, the from…import statement can be used. The following example illustrates importing pi from math module.

**Example Program**

from math import pi

print(“The value of pi is :”, pi)

**Output**

The value of pi is : 3.14159265359

We have imported only the pi function from the math module. Hence no need to use the dot operator. We can also import multiple attributes from the module using from…import statement. The following example illustrates the import of pi as well as sqrt() from the math module using from…import statement.

**180** | Internet of Things

**Example Program**

from math import pi,sqrt print(“The value of pi is :”, pi)

print(“The square root of 4 is:”,sqrt(4))

***Output***

The value of pi is: 3.14159265359 The square root of 4 is: 2.0

***import* all names**

We can import all names (definitions) from a module using the following construct. The following example shows how all the definitions from the math module can be imported. This makes all names except those beginning with an underscore, visible in our scope.

**Example Program**

from math import \*

print(“The value of pi is :”, pi ) print(“The square root of 4 is:”,sqrt(4))

***Output***

The value of pi is: 3.14159265359 The square root of 4 is: 2.0

Here all functions in the module math are imported. But we have used only pi and sqrt. Importing everything with the asterisk (\*) symbol is not a good programming practice. This can lead to duplicate definitions for an identifier.

##### 7.10. Exception Handling

An exception is an abnormal condition that is caused by a runtime error in the program. It disturbs the normal flow of the program. An example for an exception is division by 0. When a Python script encounters a situation that it cannot deal with, it raises an exception. An exception is a Python object that represents an error.

If the exception object is not caught and handled properly, the interpreter will display an error message. If we want the program to continue with the execution of the remaining code, then we should try to catch the exception object thrown by the error condition and then display appropriate messages for taking corrective actions. This task is known as exception handling.

Python has a collection of built-in Exception classes. If the runtime error belongs to any of the pre-defined built-in exception, it will throw the object of the appropriate exception. Python uses a keyword try to prepare a block of code that is likely to cause an error and throw an exception. An except block is defined which catches the exception thrown by the

Introduction to Python Programming | **181**

try block and handles it. The try block can have one or more statements that could generate an exception. If anyone statement generates an exception, then the remaining statements in the block are skipped and execution jumps to the except block that is placed next to the try block.

The except block can have more than one statement and if the except parameter matches with the type of the Exception object, then the exception is caught and statements in the except block will be executed. Every try block should be followed by atleast one except statement.

**try….except**

The following gives the syntax of try….except statement.

**Syntax**

try:

suite

except *Exception1*:

Exception1\_suite#Executes when Exception1 occurs. except *Exception2*:

Exception2\_suite#Executes when Exception2 occurs. else:

else\_suite #Executes if there is no exception in the try block.

When an exception occurs inside a try block, it will go to the corresponding except block. If no exception is raised inside a try block then after the try block, the statements in the else block is executed. The else block can also be considered a place to put the code that does not raise any exception.

**Example Program** #Exception Handling try:

a=int(input(“First Number:”)) b=int(input(“Second Number:”)) result=a/b print(“Result=”,result)

except ZeroDivisionError: print(“Division by Zero”)

else:

print(“Successful Division”)

***Output***

First Number:10 Second Number:0 Division by Zero

**182** | Internet of Things

In the above example, the second number is 0. Since division by zero is not possible, an exception is thrown and the execution goes to the except block. All the rest of the statements is bypassed in the try block. Hence the output is displayed as Division by Zero.

##### 7.11. File Handling

File is a named location on disk to store related information. When we want to read from or write to a file we need to open it first. When we are done, it needs to be closed, so that resources that are tied with the file are freed. Hence, in Python, a file operation takes place in the following order.

1. Open a file
2. Read or write (perform operation)
3. Close the file

##### Opening a File

Before we can read or write a file, we have to open it using Python’s built-in open() function. This function returns a file object, also called a handle, as it is used to read or modify the file accordingly.

We can specify the mode while opening a file. In mode, we specify whether we want to read ‘r’, write ‘w’ or append ‘a’ to the file. We also specify if we want to open the file in text mode or binary mode.

The following shows the syntax for opening a file.

***Syntax***

file object = open(filename,accessmode)

filename: The filename argument is a string value that contains the name of the file that we want to access.

accessmode: The accessmode determines the mode in which the file has to be opened, i.e., read, write, append, etc. A complete list of possible values is given below. This is optional parameter and the default file access mode is read (r).

##### Modes for Opening a File

1. ‘r’ - Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
2. ‘r+’- Opens a file for both reading and writing. The file pointer placed at the beginning of the file.
3. ‘w’- Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
4. ‘w+’ - Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.

Introduction to Python Programming | **183**

1. ‘wb+’ - Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
2. ‘a’ - Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
3. ‘a+’ - Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.

##### Reading from a File

To read the content of a file, we must open the file in reading mode. The read() method reads a string from an open file. It is important to note that Python strings can have binary data, apart from text data. The following gives the syntax for reading from a file.

**Example Program**

# Open a file in read mode fo = open(“test.txt”, “r”)

str = fo.read(11) #Reads first 11 characters from the file print(“String Read is : “, str)

# Close opened file fo.close()

1. file.read() – Here, the passed parameter size is the number of bytes to be read from

the opened file. This method starts reading from the beginning of the file and if size

is missing, then it tries to read as much as possible, maybe until the end of file.

1. file.readline()-The method readline()reads one entire line from the file. A trailing newline character is kept in the string. An empty string is returned only when EOF is encountered immediately.
2. file.readlines([sizeint]) - The method readlines() reads until EOF and returns a list containing the lines. sizeint is an optional argument and if it is present, instead of reading up to EOF, whole lines approximate to sizeint bytes are read. An empty string is returned only when EOF is encountered immediately.

##### Writing to a File

In order to write into a file we need to open it in write ‘w’, append ‘a’ or exclusive creation ‘x’ mode. We need to be careful with the ‘w’ mode as it will overwrite into the file if it already exists. All previous data will be erased.

Writing a string or sequence of bytes (for binary files) is done using write() method. This method returns the number of characters written to the file. The write() method does

**184** | Internet of Things

not add a newline character (‘\n’) to the end of the string. The following shows the syntax of

write() method.

***Syntax***

fileObject.write(string);

Here, passed parameter string is the content to be written into the opened file.

**Example Program**

# Open a file in writing mode fo = open(“test.txt”, “w”)

fo.write(“Programming with Python is Fun.\nLet’s try Python!\n”); # Close opened file

fo.close()

print(“File”, fo.name, “closed.”)

***with* Statement to Open a File**

Python’s *with* statement is handy when we have two related operations which we would like to execute as a pair, with a block of code in between. The classic example is opening a file, manipulating the file, then closing it.

**Example Program**

with open(‘output.txt’, ‘w’) as f: f.write(‘Hello Python!’)

The above with statement will automatically close the file after the nested block of code. The advantage of using a *with* statement is that it is guaranteed to close the file no matter how the nested block exits. If an exception occurs before the end of the block, it will close the file before the exception is caught by an outer exception handler.

##### CSV Files

Python has a Comma Separated Values (CSV) module for supporting CSV files. Following example illustrates how to write to a CSV file.

**Example Program**

import csv data=[“1,2,3,4,5,6,7,8,9”.split(“,”)]

file=”output.csv”

with open(file,”w”)as csv\_file: writer=csv.writer(csv\_file,delimiter=”,”) for line in data:

writer.writerow(line) csv\_file.close()

Introduction to Python Programming | **185**

***Output***

A CSV file named “output.csv” with contents 1,2,3,4,5,6,7,8,9 is created. Below example shows how to read from a CSV file.

**Example Program** import csv file=”output.csv”

with open(file,”r”)as csv\_file: reader=csv.reader(csv\_file) for row in reader:

print(“ “.join(row)) csv\_file.close()

***Output***

1 2 3 4 5 6 7 8 9

##### 7.12. Reading Images

Python supports Pillow library for image related operations. For this we need to install PIL through PIP. Python Image library is used to work with image files.

from PIL import Image

* + - Open an image file

image=Image.open(image\_name)

* + - Display the image

image.show()

* + - resize(): Resizes the image to a specified size.

image.resize(255,255)

* + - rotate(): Rotates the image to the specified degrees counter clockwise.

image.rotate(90)

* + - format – Gives the format of the image.
    - size- Gives a tuple with 2 values as width and height of the image in pixels.
    - mode-Gives the band of the image, “L” for grey scale, “RGB” for true color image.
    - print(image.format,image.size, image.mode)
    - Converting image to different mode – Any image can be converted from one mode to “L” or “RGB” mode. Conversion between modes other than “L” and “RGB” needs conversion into any of these 2 intermediate modes.

conv\_image=image.convert(“L”)

**Following program converts a sample image into grey scale.** from PIL import Image im=Image.open(“C:/Python34/Lib/Jeeva.jpg”)

**186** | Internet of Things

im.show() grey\_image=im.convert(“L”) grey\_image.show() grey\_image.save(“Greyscale.jpg”)

##### 7.13. Networking

Python supports network services for client-server model. Socket support in the operating system allows implementing clients and servers for both connection-oriented and connectionless protocols. Python has libraries that provide higher-level access to specific application level network protocols.

To create a socket, we must import the socket module. The socket.socket() function available in the socket module, helps to create a socket. It has the following syntax.

s = socket.socket(socket\_family,socket\_type,protocol = 0)

socket\_family – The family of protocols that is used as the transport mechanism. These values are constants such as AF\_INET, PF\_INET, PF\_UNIX, PF\_X25, and so on.

socket\_type- The type of communications between the two endpoints. The values are SOCK\_STREAM for connection-oriented protocols and SOCK\_DGRAM for connectionless protocols.

**Example**

import socket s=socket.socket(socket.AF\_INET,socket.SOCK\_STREAM)

Here we made a socket instance and passed it two parameters. The first parameter is

AF\_INET and the second one is SOCK\_STREAM. AF\_INET refers to the address family ipv4. Secondly the SOCK\_STREAM means connection oriented TCP protocol. Now we could connect to a server using this socket.

***Making a Server***

A server has a bind() method which binds it to a specific IP and port so that it can listen to incoming requests on that IP and port. Next a server has a listen() method which puts the server into listen mode. This allows the server to listen to incoming connections. And lastly a server has an accept() and close() method. The accept method initiates a connection with the client and the close method closes the connection with the client. Now let’s begin making our simple server.

**Example Program**

# first of all import the socket library import socket

# next create a socket object

Introduction to Python Programming | **187**

s = socket.socket()

print(“Socket successfully created”)

# reserve a port on your computer in our # case it is 12345 but it can be anything port = 12345

# Next bind to the port

# we have not typed any ip in the ip field # instead we have inputted an empty string # this makes the server listen to requests

# coming from other computers on the network s.bind((‘’, port))

print(“socket binded to %s” %(port)) # put the socket into listening mode s.listen(5)

print(“socket is listening”)

# a forever loop until we interrupt it or # an error occurs

while True:

# Establish connection with client. c, addr = s.accept()

print(‘Got connection from’, addr)

# send a thank you message to the client. c.send(‘Thank you for connecting’)

# Close the connection with the client c.close()

***Output***

Socket successfully created socket binded to 12345 socket is listening

First of all we import socket which is necessary. Then we made a socket object and

reserved a port on our personal computer. After that we binded our server to the specified port. Passing an empty string means that the server can listen to incoming connections from other computers as well. If we would have passed 127.0.0.1 then it would have listened to only those calls made within the local computer. After that we put the server into listen mode with parameter 5. It means that 5 connections are kept waiting if the server is busy and if a 6th socket tries to connect then the connection is refused. Lastly we make a while loop and start to accept all incoming connections and close those connections after a thank you message to all connected sockets. Now we have to program a client.

###### Making a Client

Let us write a very simple client program which opens a connection to a given port 12345

**188** | Internet of Things

and a given host. It is very simple to create a socket client using the Python’s socket module function. The socket.connect(hosname,port) opens a TCP connection to hostname on the port. Once we have a socket open, you can read from it like any IO object. When done, remember to close it, as we would close a file. Type these commands in the terminal.

# start the server

$ python server.py

# keep the above terminal open

# now open another terminal and type:

$ telnet localhost 12345

After typing these commands we will get the following output in our terminal.

# in the server.py terminal you will see # this output:

Socket successfully created socket binded to 12345 socket is listening

Got connection from (‘127.0.0.1’, 52617)

# In the telnet terminal you will get this: Trying ::1...

Trying 127.0.0.1...

Connected to localhost. Escape character is ‘^]’.

Thank you for connectingConnection closed by foreign host.

This output shows that our server is working. Now let’s make our client.

**Example Program**

# Import socket module import socket

# Create a socket object s = socket.socket()

# Define the port on which you want to connect port = 12345

# connect to the server on local computer s.connect((‘127.0.0.1’, port))

# receive data from the server print(s.recv(1024))

# close the connection s.close()

***Output***

# start the server:

$ python server.py

Introduction to Python Programming | **189**

Socket successfully created socket binded to 12345 socket is listening

Got connection from (‘127.0.0.1’, 52617)

$ python client.py

Thank you for connecting

##### 7.14. Conclusion

This Chapter explains about Python programming. Various components of Python like input- output statements, operators are explained. Data types of Python namely numbers, string, list, tuple, set and dictionary are explained with example programs. Decision making statements, for loop, for loop with else statement, while loop, while loop with else statement and control statements are explained with illustrations. Function definition, function calling, global and local variables, functions returning more than one value are explained with example programs. Modules, different types of importing a module, exception handling techniques, file handling methods, reading images and networking or socket programming is illustrated.