1 Introduction to Computer and Python Programming

CHAPTER OUTLINE

LEARNING OUTCOMES

After completing this chapter, students will be able to:

- Identify the functionalities of modern computer systems and various programming languages
- Explain the importance of Python and describe its need as a programming language
- Install Python in various operating systems and write and execute programs in Python

1.1 INTRODUCTION

Nowadays computers have become an integral part of human lives. They are used in diverse sectors to execute a range of everyday tasks such as reservation of tickets, payment of electricity bills, virtual transfer of money, forecasting the weather, diagnosis of diseases and so on. In short, each one of us—directly or indirectly—makes use of computers. So, before learning python programming language, this chapter explains the basics of computers and different types of programming

languages for ease of beginners and then introduces Python in detail, covering installation and execution of Python and Python programs.

1.2 WHAT IS A COMPUTER?

The word computer is derived from 'compute', which means 'to calculate'. A computer is an electronic device which accepts data from a user, processes the data for calculations specified by the user and generates an output. A computer performs these operations with speed and accuracy using certain hardware and software. Hardware is visible physical element of a computer and software consist of a written set of instructions used to control the hardware. Figure 1.1. shows the various components of a modern computer system.

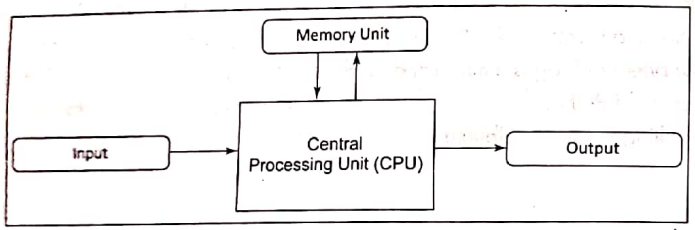


Figure 1.1 Block diagram of a modern computer system

The hardware of a computer system consists of three main components, viz. input/output (I/O) unit, central processing unit and memory unit.

1.2.1 Input/Output (I/O) Unit

Users interact with a computer using various I/O units. Inputs can be given to a computer using input devices, such as a keyboard. The input unit of a computer converts the data that it accepts from a user into a form that is understandable by it. As soon as the computer receives the input, it is processed and sent to its output device. Monitors, printers, etc., are examples of output devices of a computer.

1.2.2 Central Processing Unit (CPU)

The CPU is one of the most important parts of a computer. It handles processing of data and consists of an arithmetic logic unit (ALU) and a control unit. The ALU performs all operations on the input data and the control unit directs the computer memory and input and output devices response to the instructions received from a program.

1.2.3 Memory Unit

The function of the memory unit is to store programs and data. The unit is a compilation of numerous storage cells and each cell can store one bit of information. These cells are processed

in a group of fixed sizes of units called words and they never read or write as individual cells. A computer's memory system can be divided into the following three groups:

1. *Internal memory:* It refers to the set of registers confined to the CPU. These registers hold temporary results when a computation is in progress.
2. *Primary memory:* It is a storage area in which all the programs are executed. All programs and data must be stored in the primary memory for speedy execution.
3. *Secondary memory:* It is known as **external memory** or **storage memory**. Programs and data are stored here for the long term. Hard disk, floppy disk, CDs, DVDs and magnetic tapes are different forms of secondary memory.

1.3 OVERVIEW OF PROGRAMMING LANGUAGES

A computer program is a set of instructions, which performs a specific task when executed by a computer. Computer programs are commonly known as **software**. The instructions in a program tell a computer what to do and these instructions can be written in three types of programming languages described next.

1.3.1 Machine Language

A computer is an electronic machine which can understand any instruction written in binary form, i.e. using only 0s and 1s. A program written in 0s and 1s is called machine language. While a computer easily understands this language, it is difficult for humans to write an instruction in terms of 0's and 1's. Consider the following example.

Example

A series of numbers, such as 0011, 1000, 1010 is an instruction written in machine language. The instruction implies addition of a number stored at location 8 (1000) and another number stored at location 10 (1010) and storing the result at location 8 (1000). Here, the binary code 0011 stands for addition.

1.3.2 Assembly Language

From the above example, we know that it is difficult to write, read, communicate or change a program written in machine language for humans. Hence, the need to create another more convenient language arose. In assembly language, which was developed subsequently, machine operations are represented by mnemonic codes (such as ADD and MUL) and symbolic names that specify the memory address. Consider the following example.

Example

```
MOV X, 10
MOV Y, 20
ADD X, Y
```

Here the mnemonic MOV indicates an operation to store the value of variable X as 10. The mnemonic ADD implies addition of the contents of variable X, Y and finally storing the result in variable X itself.

Since computers cannot understand the assembly language, a program called **assembler** is used to translate assembly language programs into equivalent machine language programs.

1.3.3 High-level Language

High-level languages are much easier to write than low-level languages because programs written in these are similar to instructions written in the English language. Here 'high' does not imply that the language is complicated. It means that the language is more problem oriented. Generally, high-level languages are platform independent. This means that one can write a program in a high-level language and run it on different types of machines. Instructions written in high-level languages are called **statements**.

For example, a statement to calculate the square of a number can be written in a high-level language as

Square = number * number

There are many high-level languages and the selection of a language is based on the purpose it is expected to fulfill. A program written in a high-level language is called *source code* or *source program*. The process of executing programs written in high-level languages is given below.

- **STEP 1:** An **interpreter** or **compiler** is used to translate a program written in a high-level language into its equivalent machine code for execution.
- **STEP 2:** A **linker** is used to combine the object code and the code stored in libraries into machine language.
- **STEP 3:** Finally, the machine language code generated in Step 2 is executed.

Figure 1.2 depicts the steps on how to execute a program written in a high-level language.

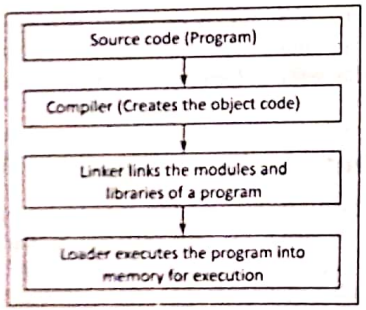


Figure 1.2 Steps to execute a high-level language program

The next section describes compiler, interpreter, linker and loader in detail.

Compiler

It is a software that translates a program written in a high-level language into machine language. This compiled program is called object code. The object code is an executable code which can run as a standalone code, i.e. it does not need the compiler to be present during execution. Every programming language, such as C, C++ and Java has its own compiler.

Interpreter

While a compiler converts the whole source code into an equivalent object code or machine code, the interpreter reads the source code line by line and converts it into object code (i.e. a code understandable to the machine.

Linker

It is a program that links different program modules and libraries to form a single executable program. A source code of a program is very large. It can consist of hundreds of lines of code. Before the execution of a program, all the modules of the program and the required libraries are linked together using a software called a **linker**. The compiled and linked program is called the **executable code**.

Loader

This software is used to load and relocate an executable program in the main memory during execution. The loader assigns a storage space to a program in the main memory for execution.

1.4 HISTORY OF PYTHON

Python was developed by Guido van Rossum at National Research Institute for Mathematics and Computer Science in Netherlands in 1990. Rossum wanted the name of his new language to be short, unique and mysterious. Inspired by *Monty Python's Flying Circus*, a BBC comedy series, he named the language Python.

Python became a popular programming language, widely used in both industry and academia because of its simple, concise and extensive support of libraries. It is a general purpose, interpreted and object-oriented programming language. Python source code is available under General Public License (GPL) and maintained by a core development team at the same institute.

1.4.1 Why Python?

COBOL, C#, C, C++ and Java are a few of the many programming languages available in information and technology today. One common question that beginners in programming often ask is, *'Why use Python when there are so many programming languages?'* While on one hand it may just be a matter of personal preference, there are some very well-known advantages of Python which make it a popular programming language. These are given below.

1. *Readability:* Developer's **readability** of code is one of the most crucial factors in programming. The longest part of any software's life cycle is its maintenance. Therefore, if a software has a highly readable code, then it is easier to maintain. Readability also helps a programmer to reuse the existing code with ease to maintain and update a software. Python offers more readability of code when compared to other programming languages.
2. *Portability:* Python is platform independent, i.e. its programs run on all platforms. The language is designed for portability.
3. *Vast support of libraries:* Python has a large collection of in-built functionalities known as standard *library functions*. Python also supports various third-party software like NumPy. **NumPy** is an extension, i.e. it provides support for large, multidimensional arrays and matrices.

4. ***Software integration:*** An important aspect of Python is that it can easily extend, communicate and integrate with several other languages. For example, Python code can easily invoke libraries of C and C++ programming languages. It can also be used to communicate with **Java** and **.net** components. Python can sometimes act as an intermediary or agent between two applications.
5. ***Developer productivity:*** Compared to other programming languages, Python is a dynamically typed language, which means there is no need to declare variables explicitly. Again, there are various other features of Python due to which the size of code written is typically smaller or half of the code written in some other languages, such as C, C++ or Java.

As the size of code is reduced quite a bit, there is less to type and debug. The amount of time needed to compile and execute is also very less as compared to other programming languages. Python programs run immediately, i.e. without taking much time to link and compile.

These benefits offered by Python make it the topmost choice for programmers to develop application software or projects with Python.

1.4.2 Installing Python in Windows

Python is available for almost all operating systems such as Windows, Mac, Linux/Unix, etc. The complete list of different versions of Python can be found at http://www.Python.org/downloads. Step-wise details for installing Python in Windows are given below.

⊙ **STEP 1:** Open an Internet browser like Internet Browser, Mozilla Firefox or Chrome. Type http://www.Python.org/ in the address bar and press Enter. Immediately, the following page will appear (Figure 1.3).

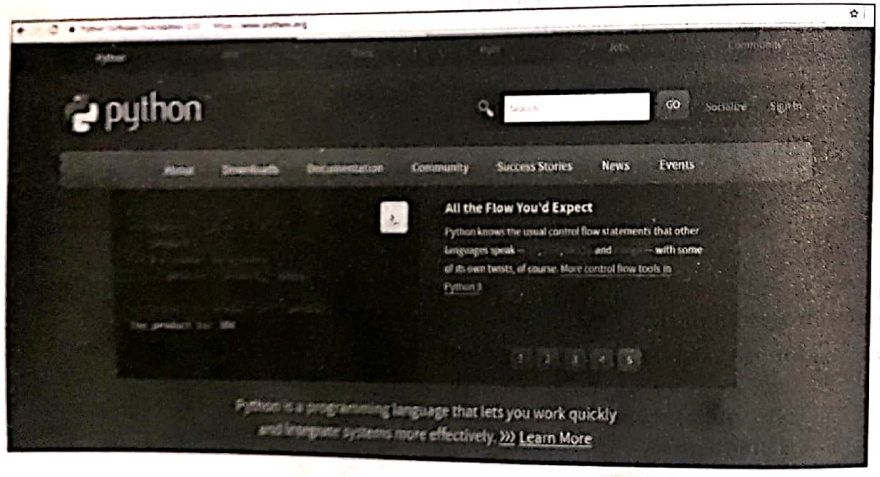


Figure 1.3 Python home page

⊙ **STEP 2:** Click on Downloads and you will see the latest version of Python. Since all programs in this book are written and executed on **Python 3.4**, download **Python 3.4** version by clicking on **All Releases** under Downloads as shown in Figure 1.4.

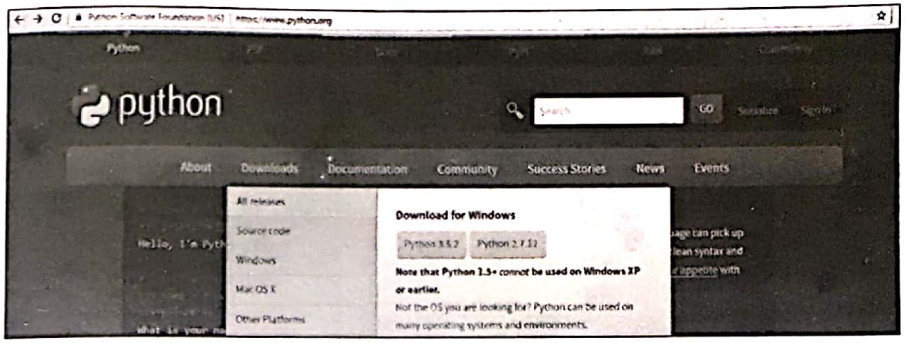


Figure 1.4 Python download page

⊙ **STEP 3:** After clicking on All Releases under Downloads browse through the page to the bottom. You will see a list of Python releases as shown in Figure 1.5.

Looking for a specific release?

Python releases by version number:

Release version	Release date		Click for more
Python 2.7.9	2014-12-10	Download	Release Notes
Python 3.4.2	2014-10-13	Download	Release Notes
Python 3.3.6	2014-10-12	Download	Release Notes
Python 3.2.6	2014-10-12	Download	Release Notes
Python 2.7.8	2014-07-02	Download	Release Notes
Python 2.7.7	2014-06-01	Download	Release Notes
Python 3.4.1	2014-05-19	Download	Release Notes

Figure 1.5 Python release versions

⊙ **STEP 4:** Click on Python 3.4.2 and download it.

⊙ **STEP 5:** Open the folder where you have downloaded the Python 3.4 version pack and double click on it to start the installation (Figure 1.6).

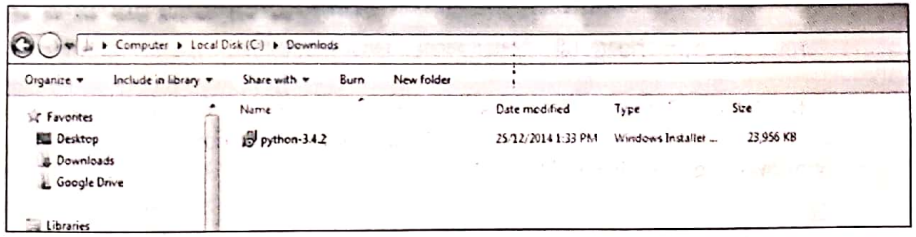


Figure 1.6 Python software

⊙ **STEP 6:** After clicking on it you will see the first window to set up Python 3.4.2 (Figure 1.7).

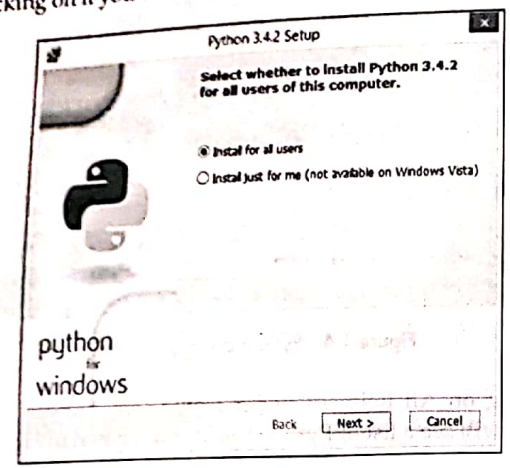


Figure 1.7 Python first setup window

⊙ **STEP 7:** Click on Next and you will see a second window which tells you to specify the location where you want to install Python (Figure 1.8).

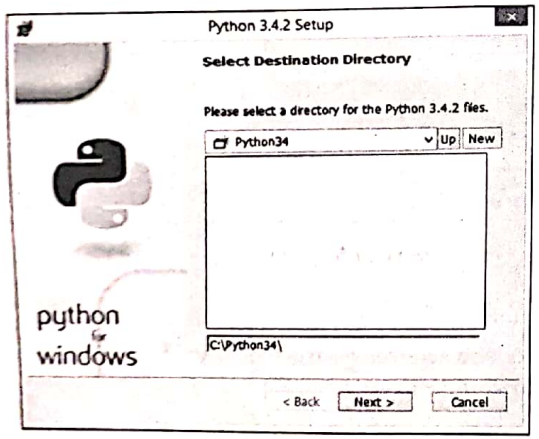
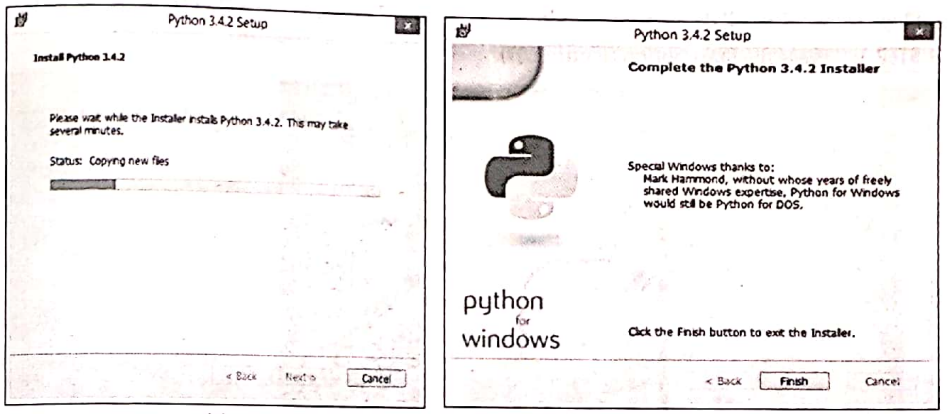


Figure 1.8 Python second setup window

By default, Python will be installed in C:\. Then click on Next to continue the installation. Just before completing the installation, it will show you the following two windows (Figures 1.9 a and b).



(a) (b)

Figures 1.9 a and b Python final setup window

⊙ **STEP 8:** Click on Finish to complete the installation.

⊙ **STEP 9:** To check if Python is installed successfully just press windows key on Windows 7 or Windows 8 and then in the search bar type Python as shown in Figure 1.10.

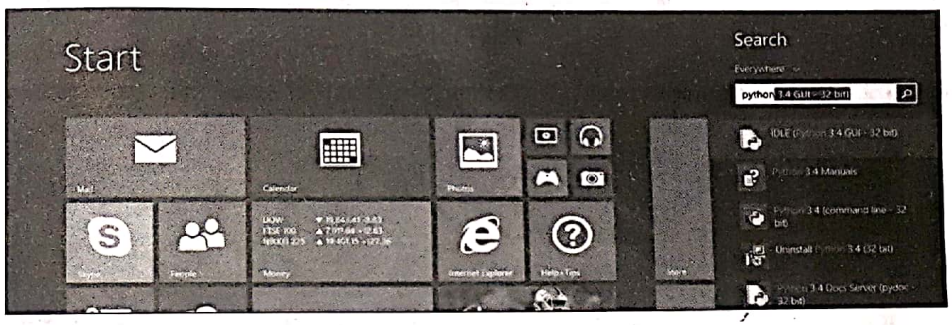


Figure 1.10 Windows 8 showing successful installation of Python

1.4.3 Starting Python in Different Execution Modes

After installing Python in Windows, you can start Python in two different modes, viz. Python (Command Line) and Python (IDLE).

Starting Python (Command Line)

Python is an interpreted language. You can directly write the code into the Python interpreter or you can write a sequence of instructions into a file and then run the file.

When you execute Python expressions or statements from the command line then you are in **interactive mode** or **interactive prompt**.

Steps for writing a Python command line in Windows 7 are given as follows:

⊙ **STEP 1:** Press the Start button (Figure 1.11).

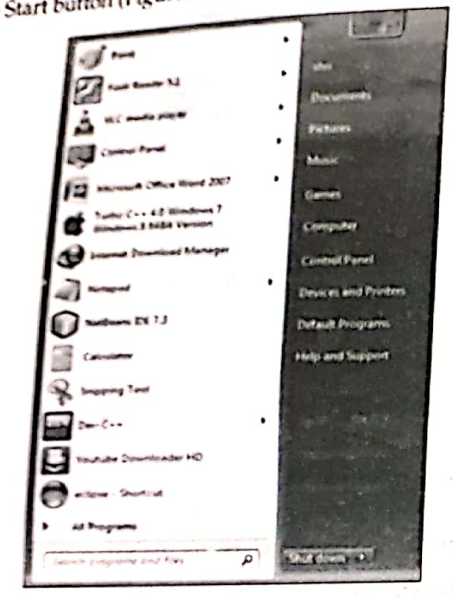


Figure 1.11

⊙ **STEP 2:** Click on All programs and then **Python 3.4**. After clicking on **Python 3.4** you will see a list of options as shown in Figure 1.12.

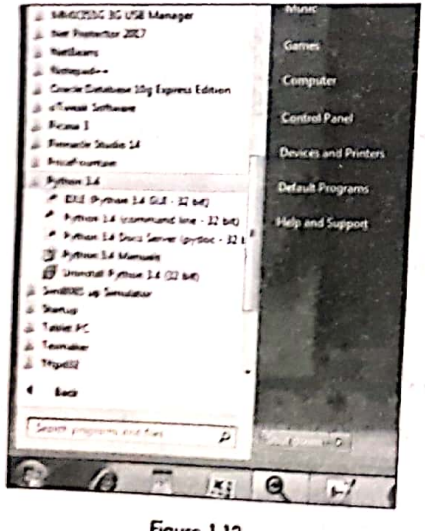


Figure 1.12

⊙ **STEP 3:** In this list click on **Python (Command Line—32 bit).** After clicking on it, you will see the Python **interactive prompt** in Python command line as shown in Figure 1.13.

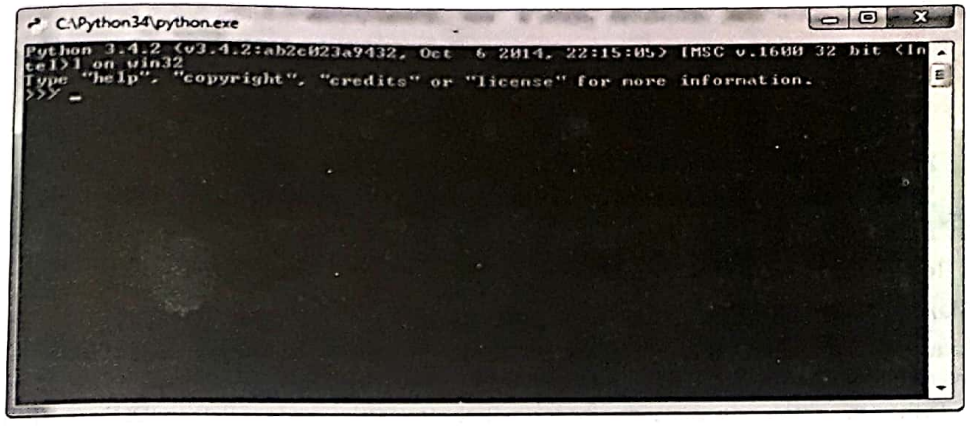


Figure 1.13 Python interactive mode as Python command line window

In Figure 1.13, the Python command prompt contains an opening message >>>, called **command prompt.** The cursor at the command prompt waits for you to enter a Python command. A complete command is called a **statement**. Simple commands executed in the interactive mode of Python command line are shown in Figure 1.14.

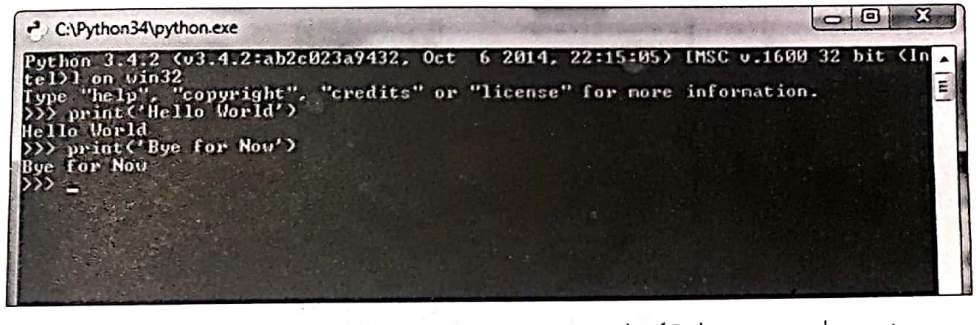


Figure 1.14 Simple commands executed in interactive mode of Python command prompt

We have written two simple commands or statements. The first statement, i.e. **print('Hello World')** when executed in the interactive mode of Python command prompt gives the output as the entered command, i.e. **'Hello World'** for this message. More details about print and its syntax are explained in Chapter 2.

Precautions to be taken while executing commands in the interactive mode of Python with command line are given as follows.

If you try to put an extra space between Python prompt, i.e. >>> and the command, then it will produce an error called **Indentation Error: Unexpected Indent**. A simple example to demonstrate this error is given below.

Example:

```
>>> print('Hello World')
 File "<stdin>", line 1
print('Hello World')
^
IndentationError: unexpected indent
```

Thus, due to an extra space between >>> and command, i.e. **print('Hello world')**, the Python interpreter raises an error.

To exit from the command line of **Python 3.4**, press **Ctrl+Z** followed by **Enter.**

Starting Python IDLE

Launching Python **IDLE** is another way to start executing Python statements or commands in the interactive mode of Python IDLE. It is a graphical integrated development environment for Python.

Python statements or commands which run in the interactive mode of Python IDLE are called **shell**. IDLE is downloaded by default while installing Python. Launching Python IDLE is the simplest way to open a Python shell. The steps to launch Python IDLE are similar to those used to start a Python command line and are detailed below.

⊙ **STEP 1:** Press the Start button.

⊙ **STEP 2:** Click on All Programs and then **Python 3.4**. After clicking on **Python 3.4** you will see a list of options as shown in Figure 1.15.

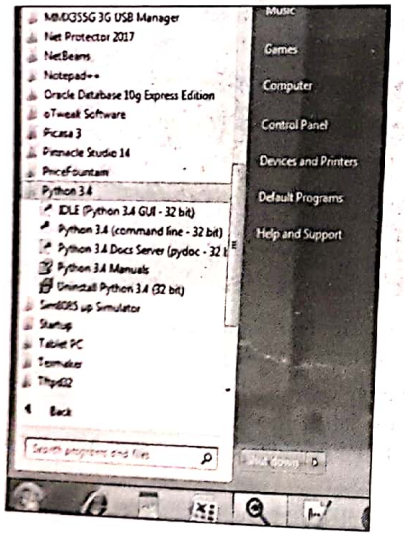


Figure 1.15

⊙ **STEP 3:** Click on **IDLE (Python 3.4 GUI—32 bit)** and you will see the Python **interactive prompt**, i.e. an **interactive shell** as shown in Figure 1.16.

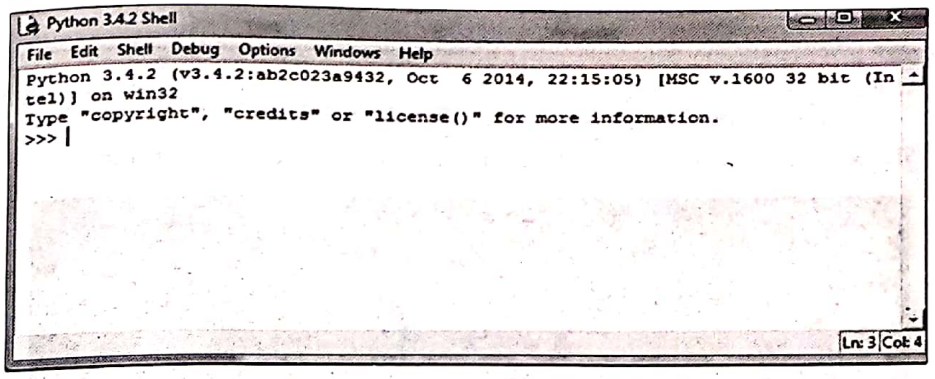


Figure 1.16 Python IDLE—Interactive shell

In Figure 1.16, a Python interactive shell prompt contains an opening message >>>, called **'shell prompt'**. The cursor at the shell prompt waits for you to enter a Python command. A complete command is called a **statement**. As soon as you write a command and press Enter, the **Python interpreter** will immediately display the result.

Figure 1.17 shows simple commands which are executed in the interactive mode, i.e. the interactive shell of Python IDLE.

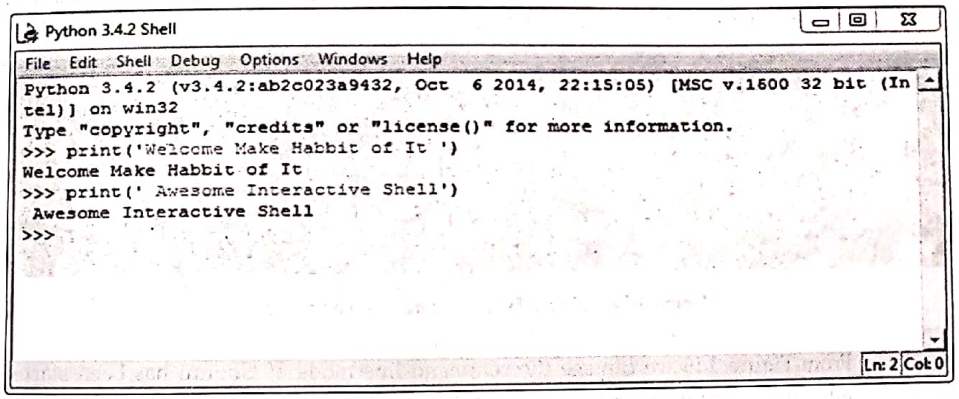


Figure 1.17 Running commands in Python IDLE's interactive shell

Note: Hereafter all commands given as examples in the forthcoming chapters of this book are executed in Python 3.4 IDLE's interactive mode, i.e. the interactive shell prompt.

1.5 INSTALLING PYTHON IN UBUNTU

Python 2.7 and Python 3.4 are installed by default on Ubuntu 15.0. The following steps can be used to check their presence.

- **STEP 1:** Open Ubuntu 15.0.
- **STEP 2:** Press the Windows button on the keyboard and type 'terminal' or press the shortcut Ctrl+Alt+T to open the terminal.
- **STEP 3:** Once the terminal is open, type **Python3—version** to check if it is installed.

Figure 1.18 Check default installation of Python

- **STEP 4:** From Figure 1.18 we can know that default Python3.X version has been installed successfully.
- **STEP 5:** To launch the **command line mode** or **interactive mode** of Python 3.X version in Ubuntu, type Python3 on the terminal.

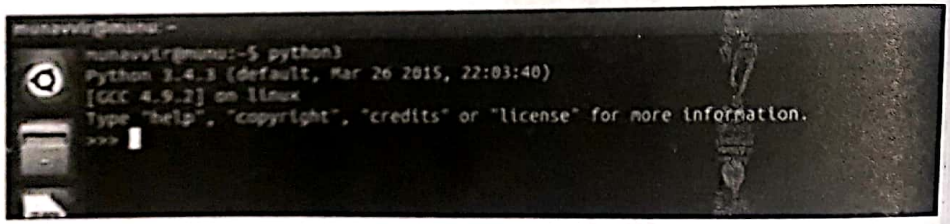


Figure 1.19 Ubuntu Python3 command line mode

From Figure 1.19 we can see the command line mode of Ubuntu has been started and the programmer is ready to give instructions to Python. The following figure (Figure 1.20) illustrates an example of printing 'hello' in the command line mode of Ubuntu.

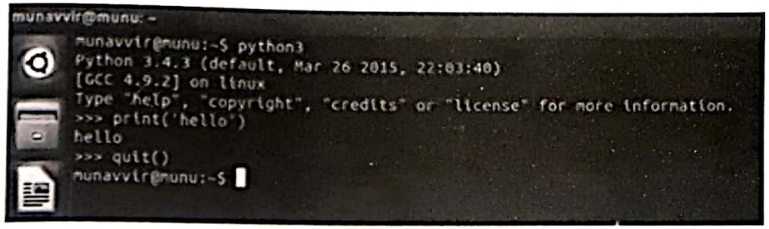


Figure 1.20 Executing instructions of Python3 in Ubuntu command line mode

- **STEP 6:** A programmer can launch **Python IDLE mode** in Ubuntu in the same manner. To launch the IDLE mode of Python, type the command given below on the terminal.

Python -m idlelib

> **Note:** If IDLE is not installed then the programmer can install it by typing the command given below on the terminal.
>
> sudo apt-get install idle3

1.6 EXECUTING PYTHON PROGRAMS

The previous section explained the installation of Python3 in Windows and Ubuntu. This section describes how to execute Python programs in script mode on Windows. All the programs written in this book are written and executed on Windows. Once IDLE is launched in Ubuntu, a programmer can write programs in script mode in the same manner as done in Windows.

Running Python programs from a script file is known as running Python in **script mode**. You can write a sequence of instructions in one file and execute them. The steps required to write Python programs in Python IDLE's script mode are given as follows.

- **STEP 1:** In Python IDLE's - Shell window, click on **File** and then on **New File** or just click **CTRL+N** (Figure 1.21).

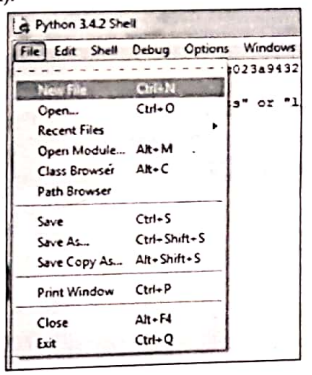


Figure 1.21 Python IDLE file menu bar

As soon as you click on New File, the window shown below will open (Figure 1.22).

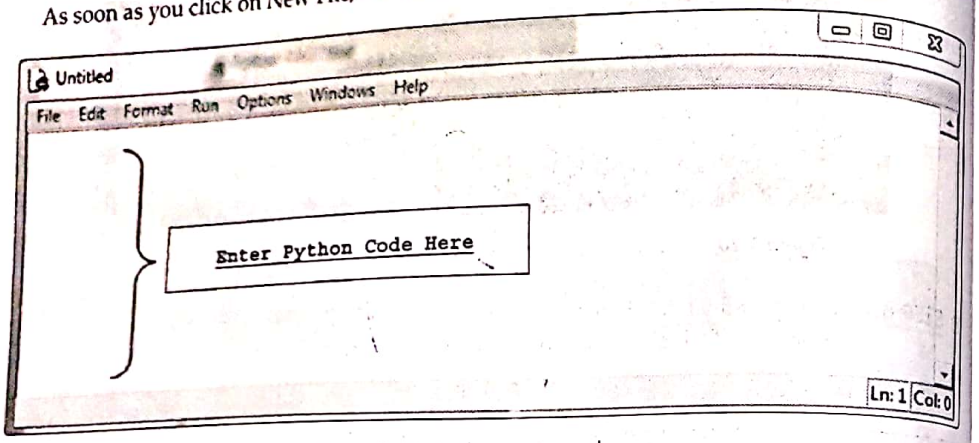


Figure 1.22 Python script mode

You can write a series of instructions in this window and then run it to view the output.

1.6.1 Writing the First Python Program in Script Mode

Use the following steps to create and run your first Python program.

⊙ **STEP 1:** Writing Python code in script mode

Let us consider a simple program to print the messages **"Hello Welcome to Python"**, **"Awesome Python!"** and **"Bye"** on the console. The statements needed to print these are

```
print('Hello Welcome to Python')
print('Awesome Python!")
```

Once you write the above statements in Python script mode, they will look like as given in **Figure 1.23**.

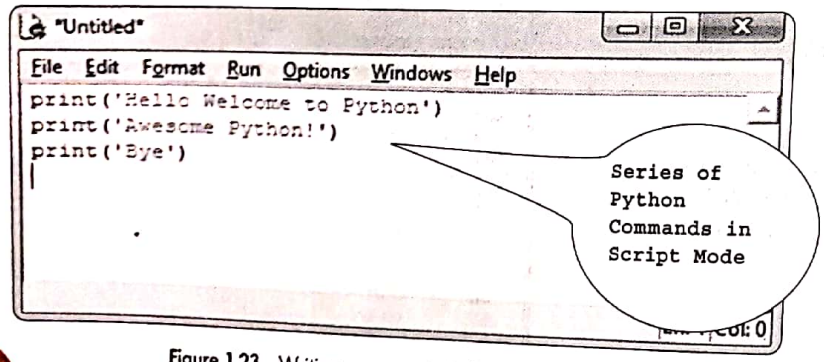


Figure 1.23 Writing program in Python script mode

⊙ **STEP 2:** Save the above code written in script mode by some name.

In Figure 1.23 we can see the name ***Untitled**. If you don't save the above code by some specific name, then by default the Python interpreter will save it using the name **Untitled.py**. In this name, **py** indicates that the code is written in Python. The * in front of Untitled indicates that the program has not been saved. To identify the purpose of a program, you should give it a proper name. Follow the steps given below to save the above program.

⊙ **STEP 1:** Click on File and then click on Save or press Ctrl+S. Then you will see the default installation folder (Python34) to save the file (Figure 1.24).

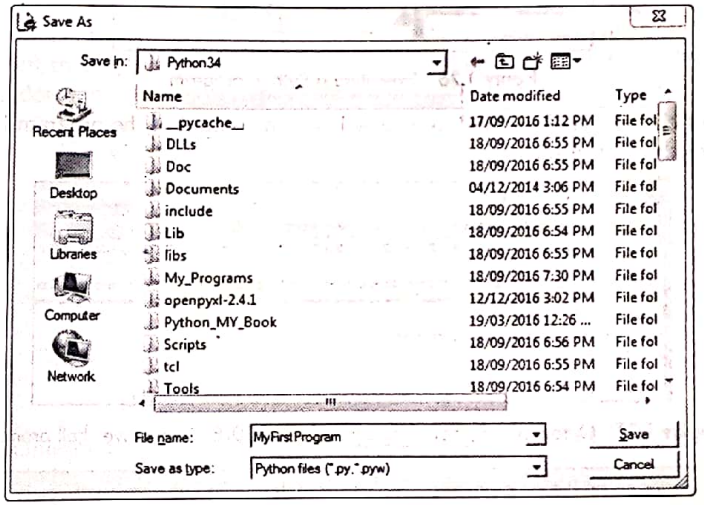


Figure 1.24 Saving a Python program

⊙ **STEP 2:** Write the name of your Python program. As it is your first Python program, you can save it as **MyFirstProgram.** Once you write the name of the file, click on Save. After the name is saved, it will get displayed on title bar of the Python script window as shown in Figure 1.25.

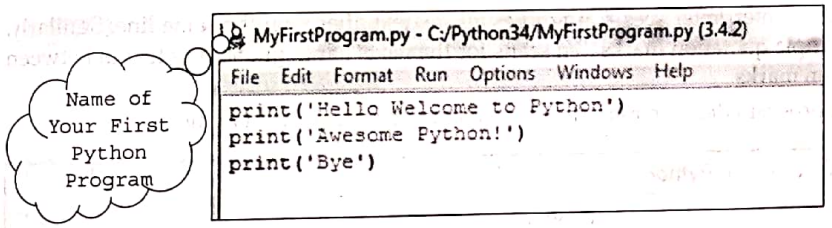


Figure 1.25 File name appearing on the title bar

⊙ **STEP 3:** Executing a Python program: A Python program is executed only after it is saved with a specific file name. Thus, to run the above Python program, click on **Run** and then **Run Module** as shown in Figure 1.26. Alternatively, you can also press **Ctrl+F5** to run the program.

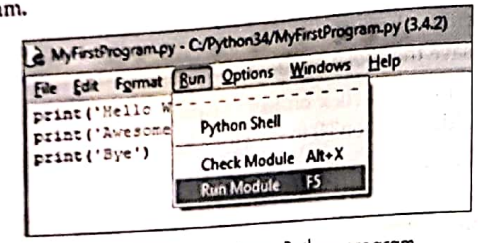


Figure 1.26 Executing a Python program

After clicking on **Run Module** you will see the output of the program if it is written correctly (**Figure 1.27**).

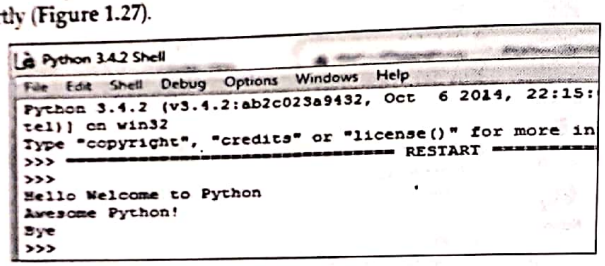


Figure 1.27 Output of a Python program in Python IDLE's interactive shell prompt

> **Note:** Hereafter all the Python programs given as examples in the forthcoming chapters of this book are executed in Python 3.4 IDLE's script mode.

1.7 COMMENTING IN PYTHON

Comments in Python are preceded by a hash symbol (#) on a line and called a **line comment**. Three consecutive single quotation marks ''' are used to give multiple comments or comments on several lines at once and called **paragraph comment**.

When the Python interpreter sees #, it ignores all the text after # on the same line. Similarly, when it sees the triple quotation marks ''' it scans for the next ''' and ignores any text in between the triple quotation marks.

The following program demonstrates the use of comment statements in Python.

#Learn How to Comment in Python

```
print('I Learnt How to Comment in Python')
''' Amazing tool
in Python called Comment'''
print('Bye')
```


Output

```
I Learnt How to Comment in Python
Bye
```

Explanation As explained above, Python ignores all the text in a statement if it is preceded by the # symbol. When the above program is executed, it ignores all the text followed by the # symbol and triple quotation marks.

1.8 INTERNAL WORKING OF PYTHON

When a programmer tries to run a Python code as a script or instructions in an interactive manner in a Python shell, then Python performs various operations internally. All such internal operations can be broken down into a series of steps as shown in **Figure 1.28**.

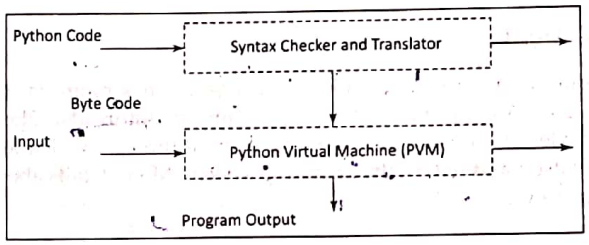


Figure 1.28 Internal working of Python

The Python interpreter performs the following steps to execute a Python program or run a set of instructions in interactive mode.

⊙ **STEP 1:** The interpreter reads a Python code or instruction. Then it verifies that the instruction is well formatted, i.e. it checks the syntax of each line. If it encounters any error, it immediately halts the translation and shows an error message.

⊙ **STEP 2:** If there is no error, i.e. if the Python instruction or code is well formatted then the interpreter translates it into its equivalent form in low level language called **"Byte Code"**. Thus, after successful execution of Python script or code, it is completely translated into byte code.

⊙ **STEP 3:** Byte code is sent to the **Python Virtual Machine (PVM).** Here again the byte code is executed on **PVM.** If an error occurs during this execution then the execution is halted with an error message.

1.9 PYTHON IMPLEMENTATIONS

The standard implementation of Python is usually called **"CPython"**. It is the default and widely used implementation of the Python programming language. It is written in C. Besides C, there are different implementation alternatives of Python, such as **Jython, IronPython, Stackless** and **PypY.** All these Python implementations have specific purposes and roles. All of them make use of simple

Python language but execute programs in different ways. Different Python implementations are briefly explained ahead.

1.9.1 Jython

Originally, Jython was known as "JPython". JPython takes Python programming language syntax and enables it to run on the Java platform. In short, it is used to run Python programs on Java platforms. More details about JPython can be found at http://jPython.org.

1.9.2 IronPython

IronPython is an open source implementation of Python for the .NET framework. It uses dynamic language runtime (DLR), which is a framework for writing dynamic languages for **.net**. A major use of **IronPython** is to embed .net applications. More details about IronPython can be found at http://ironpythonPython.net.

1.9.3 Stackless Python

It is a Python programming language interpreter. If you run a program on Stackless Python environment then the running program is split into multithreads. The best thing about multithreads in Stackless Python is the handling of multithreads, which are managed by the language interpreter itself and not by the operating system. More details about Stackless Python can be found at http://www.stackless.com.

1.9.4 PyPy

The **PyPy** is a reimplementation of Python in Python. In short, the Python interpreter is itself written in Python. It focuses on speed, efficiency and compatibility. It makes use of **Just-in-Time compiler (JIT)** to run the code more quickly as compared to running the same code in regular Python language. More details about **PyPy** Python can be found at http://pypy.org.

SUMMARY

- A computer is an electronic device which accepts data from a user, processes it for calculations specified by the user and generates an output.
- The hardware of a computer system consists of three main components, viz. **input/output (I/O) unit, central processing unit (CPU)** and **memory unit**.
- A program written in 1s and 0s is called machine language.
- In assembly languages, machine operations are represented by **mnemonic codes** such as ADD, MUL, etc. and symbolic names that specify the memory address.
- Programs written in high-level languages are similar to instructions written in English language.
- An **assembler** is used to translate an **assembly language program** into an equivalent **machine language program**.
- An **interpreter** or **compiler** is used to translate a program written in a high-level language into an equivalent machine code for execution.
- An **interpreter** reads the source code line by line and converts it into object code.

- A **compiler** is a software which translates an entire program written in a high-level language into machine language at one go.
- A **loader** is a software used to load and relocate the executable program in the main memory during execution.
- Python is a general purpose, interpreted and objects oriented programming language.
- You can enter Python statements interactively from the Python prompt **>>>**.
- Python source programs are case sensitive.
- The **#** symbol is used to comment a single line in Python.
- Triple single quotation **'''** marks are used to comment multiple lines in Python.
- Python programs can be executed on any operating system like Windows, Linux or Ubntu.

KEY TERMS

- **Assembly Language:** Machine operations are represented by mnemonic code
- **Byte Code:** The Python interpreter translates Python codes/instructions into their equivalent low level language
- **Central Processing Unit (CPU):** It consists of Arithmetic Logical Unit and Control Unit
- **High-level Language:** Programs are written in a manner similar to writing instructions in English language
- **Jython, IronPython, Stackless, PyPy:** Different implementation alternatives of Python
- **Machine Language:** Instructions are written in binary form, i.e. 0s and 1s
- **Python Virtual Machine (PVM):** Used to check if the object code contains errors

REVIEW QUESTIONS

A. Multiple Choice Questions

1. Which of the following memory is used to store temporary results in registers when the computation is in progress?
   a. Primary Memory b. Secondary Memory
   c. Internal Memory d. None of the above
2. Secondary memory is also called _____.
   a. Storage Memory b. External Memory
   c. Only b d. Both a and b
3. _____ is used to translate a program written in a high-level language into its equivalent machine code.
   a. Compiler b. Linker
   c. Loader d. Both a and b
4. _____ is used to relocate executable programs to the main memory during execution.
   a. Linker b. Compiler
   c. Interpreter d. Loader
5. What is the correct syntax for the print statement in Python 3.0?
   a. `print()` b. print
   c. `print()` d. None of the above

Basics of Python Programming

2

CHAPTER OUTLINE

LEARNING OUTCOMES

After completing this chapter, students will be able to:

- Describe keywords, delimiters, literals, operators and identifiers supported by Python
- Read data from the console using input function
- Assign value or data to a variable and multiple values to multiple variables at a time
- Use `ord` function to obtain a numeric code for a given character value, `chr` function to convert numeric value to a character and `str` function to convert numbers to a string
- Format strings and numbers using the `format` function
- Identify and use various in-built math functions supported by Python

2.1 INTRODUCTION

Computer programming languages are designed to process different kinds of data, viz. numbers, characters and strings in the form of digits, alphabets, symbols etc. to get a meaningful output known as result or information. Thus, program written in any programming language consists of a set of instructions or statements which executes a specific task in a sequential form. Whereas all these instructions are written using specific words and symbols according to syntax rules or the grammar of a language. Hence, every programmer should know the rules, i.e. syntax supported by language for the proper execution of a program.

This chapter describes basics of python programming, i.e. syntax, data types, identifiers, tokens, and how to read values from the user using input function, etc.

2.2 PYTHON CHARACTER SET

Any program written in Python contains words or statements which follow a sequence of characters. When these characters are submitted to the Python interpreter, they are interpreted or uniquely identified in various contexts, such as characters, identifiers, names or constants. Python uses the following character set:

- **Letters:** Upper case and lower case letters
- **Digits:** 0,1,2,3,4,5,6,7,8,9
- **Special Symbols:** Underscore (_), (,), [,], {,}, +, -, *, &, ^, %, $, #, !, Single quote('), Double quotes("), Back slash(\), Colon(:), and Semi Colon (;)
- **White Spaces:** ('\t\n\x0b\x0c\r'), Space, Tab.

2.3 TOKEN

A program in Python contains a sequence of instructions. Python breaks each statement into a sequence of lexical components known as **tokens.** Each token corresponds to a substring of a statement. Python contains various types of tokens. Figure 2.1 shows the list of tokens supported by Python.

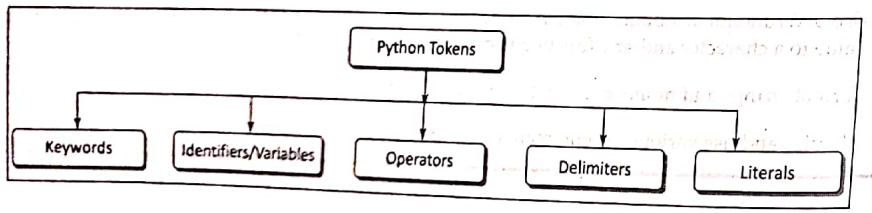


Figure 2.1 Tokens in Python

Details for all the tokens are given next.

2.3.1 Literal

Literals are numbers or strings or characters that appear directly in a program. A list of some literals in Python is as follows:

Example

```
78              #Integer Literal
21.98           #Floating Point Literal
'Q'             #Character Literal
"Hello"         #String Literal
```

Python also contains other literals, such as lists, tuple and dictionary. Details of all such literals are given in the forthcoming chapters.

Display Literals in Interactive Mode

Let us consider a simple example. Print the message **"Hello World"** as a string literal in Python interactive mode.

Example

```
>>> 'Hello World'
'Hello World'
```

As shown above, type **Hello World** in interactive mode and press enter. Immediately after pressing enter you will see the required message.

2.3.2 Value and Type on Literals

Programming languages contain data in terms of input and output and any kind of data can be presented in terms of **value**. Here **value** can be of any form like literals containing numbers, characters and strings.

You may have noticed that in the previous example we wrote 'Hello World' in single quotes. However, we don't know the type of value in it. To know the exact type of any value, Python offers an in-built method called **type.**

The syntax to know the type of any value is **type (value)**

Example

```
>>> type('Hello World')
<class 'str'>
>>> type(123)
<class 'int'>
```

Thus, when the above examples are executed in Python interactive mode, return type of value is passed to the in-built function `type()`.

2.3.3 Keywords

Keywords are reserved words with fixed meanings assigned to them. Keywords cannot be used as identifiers or variables. Table 2.1. shows the complete list of keywords supported by Python.

Table 2.1 List of Python keywords for Python version 3.0

and	del	from	None	True
as	elif	global	nonlocal	try
assert	else	if	not	while
break	except	import	or	with
class	False	in	pass	yield
continue	finally	is	raise	
def	for	lambda	return	

2.3.4 Operator

Python contains various operators, viz. arithmetic, relational, logical and bitwise operators, as shown in Table 2.2.

Table 2.2 Operators in Python

Operator Type	*Operators*
+ - * / // % **	Arithmetic Operator
== != <> <= >=	Relational Operator
and not or	Logical Operator
& | ~ ^ << >>	Bitwise Operator

Details about Python operators like operators and expressions are given in Chapter 3.

2.3.5 Delimiter

Delimiters are symbols that perform a special role in Python like grouping, punctuation and assignment. Python uses the following symbols and symbol combinations as delimiters.

```
( ) [ ] { }
, : . ` = ;
+= -= *= /= //= %=
&= |= ^= >>= <<= **=
```

2.3.6 Identifier/Variable

Identifier is the name used to find a variable, function, class or other objects. All identifiers must obey the following rules.

An identifier:

- Is a sequence of characters that consists of letters, digits and underscore

- Can be of any length
- Starts with a letter which can be either lower or upper case
- Can start with an underscore '_'
- Cannot start with a digit
- Cannot be a keyword.

Some examples of valid identifiers are Name, Roll_NO, A1, _Address etc.

Python gives a **syntax error** if a programmer writes an invalid identifier. Some examples of invalid identifiers are First Name, 12Name, for, Salary@

If we type the invalid identifiers given above in Python interactive shell, it will show an error as these are invalid.

Example

```
>>> First Name
SyntaxError: invalid syntax
>>> 12Name
SyntaxError: invalid syntax
>>> for
SyntaxError: invalid syntax
```

2.4 PYTHON CORE DATA TYPE

All features in Python are associated with an **object.** It is one of the primitive elements of Python. Further, all kinds of objects are classified into **types.** One of the easiest types to work with is numbers, and the native data types supported by Python are string, integer, floating point numbers and complex numbers.

The following section details the basic data types supported by Python.

2.4.1 Integer

From simple Mathematics, we know that an integer is a combination of positive and negative numbers including (zero) 0. In a program, integer literals are written without commas and a leading minus sign to indicate a negative value. Following is an example of simple integer literals displayed in Python interactive mode.

Example

```
>>> 10
10
>>> 1220303
1220303
>>> -87
-87
```

Integer literals can be octal or hexadecimal in format. All the above examples are of decimal type integers. Decimal integers or literals are represented by a sequence of digits in which the first digit is non-zero. To represent an **octal, 0o**, i.e. a zero and a lower or upper case letter **O** followed by a sequence of digits from **0 to 7** is used. An example of octal literals is given as follows.

Example

```
>>> 0o12
10
>>> 0o100
64
```

Note: In Python version 2.6 or earlier, octal literals were represented by the leading letter **O**, followed by a sequence of digits. In Python 3.0, octal literals have to be accompanied by a leading **0o**, i.e. a zero and a lower or upper case letter O.

In the previous section, we have learnt about representation of numbers as default **decimal (base 10)** notation and octal **(base 8)** notation. Similarly, numbers can also be represented as hexadecimal **(base 16)** notation using **0x** (zero and the letter X) followed by a sequence of digits. Simple examples of hexadecimal literals displayed in Python interactive mode are given as follows:

Example

```
>>> 0x20
32
>>> 0x33
51
```

Note: Integer in Python 2.6 (`int` and `long`)—In Python 2.6 there are two types of integers. One of 32 bits and another having unlimited precession. Python 2.6 automatically converts integers to long integers if the value of the integer overflows 32 bits.

Integers in Python 3.0 (Only `int` type)—In Python 3.0 the normal `int` and `long` integer have been merged. Hence, there is only one type called integer.

The int Function

The `int` function converts a string or a number into a whole number to integer. The `int` function removes everything after the decimal point. Consider the following example.

Example

```
>>> int(12.456)
12
```

The following example converts a string to an integer.

Example

```
>>> int('123')
123
```


2.4.2 Floating Point Number

The value of π (3.14) is an example of a real number in mathematics. It consists of a whole number, decimal point and fractional part. The length of real numbers has infinite precession, i.e. the digits in the fractional part can continue forever. Thus, Python uses floating point numbers to represent real numbers. A floating-point number can be written using a **decimal notation** or **scientific notation**. Some examples of floating point numbers displayed in Python interactive mode are given as follows:

Example

```
>>> 3.7e1
37.0
>>> 3.7
3.7
>>> 3.7*10
37.0
```

The above example shows the representation of floating point number **37.0** in both decimal and scientific manner. Scientific notations are very helpful because they help programmers to represent very large numbers. Table 2.3 shows decimal notations in scientific notation format.

Table 2.3 Example of floating point numbers

Decimal Notation	*Scientific Notation*	*Meaning*
2.34	2.34e0	$2.34 * 10^0$
23.4	2.34e1	$2.34 * 10^1$
234.0	2.34e2	$2.34 * 10^2$

The float Function

The `float` function converts a string into a floating-point number. A programmer can make use of float to convert string into float. Consider the following example.

Example

```
>>>float('10.23')
10.23
```

2.4.3 Complex Number

A complex number is a number that can be expressed in the form **a+bj**, where **a** and **b** are real numbers and **j** is an imaginary unit. Simple example of complex numbers displayed in Python interactive mode is given as follows:

Example

```
>>> 2+4j
(2+4j)
```

```
>>> type(2+4j)
<class 'complex'>
>>> 9j
9j
>>> type(9j)
<class 'complex'>
```

Here we have written a simple kind of complex number and using **type** we have checked the type of the number.

2.4.4 Boolean Type

The Boolean data type is represented in Python as type **bool**. It is a primitive data type having one of the two values, viz. **True** or **False**. Internally, the **True** value is represented as 1 and False as 0. In the following example check the type of **True** and **False** value in Python interactive mode.

```
>>> type(True)
<class 'bool'>
>>> False
False
>>> type(False)
<class 'bool'>
```

The Boolean type is used to compare the two values. For example, when relational operators, such as ==, !=, <=, >= are used in between two operands then it returns the value as **True** or **False**.

Example

```
>>> 5 == 4
False
>>> 5 == 5
True
>>> 4 < 6
True
>>> 6> 3
True
```

2.4.5 String Type

A string literal or string in Python can be created using single, double and triple quotes. A simple example of type as string is given as follows:

Example

```
>>> D = 'Hello World'
>>> D
'Hello World'
>>> D="Good Bye"
```



```
>>> D
'Good Bye'
>>> Sentence
'Hello, How are you? Welcome to the world of Python Programming. It is just the beginning. Let us move on to the next topic.'
>>> Sentence
'Hello, How are you? Welcome to the world of Python Programming. It is just the beginning. Let us move on to the next topic.'
```

In the previous examples, we presented string literals in three different formats, viz. single quote, double quote and triple single quotes. The triple single quotes are used to write a multiline string.

The str Function

The `str` function is used to convert a number into a string. The following example illustrates the same.

```
>>> 12.5          #Floating Point Number
12.5
>>> type(12.5)
<class 'float'>
>>> str(12.5)     #Convert floating point number to string
'12.5'
```

The String Concatenation (+) Operator In both mathematics and programming, we make use of '+' operator to add two numbers. Similarly, '+' operator is used to concatenate two different strings. The following example illustrates the use of + operator on strings.

```
>>> "Woooow" + "Python Programming"
'WoooowPython Programming'      #Concatenates two different strings
```

2.5 THE `print()` FUNCTION

In Python, a function is a group of statements that are put together to perform a specific task. The task of **`print`** function is to display the contents on the screen. The syntax of **`print`** function is:

Syntax of **`print()`** function:

```
print(argument)
```

The argument of the print function can be a value of any type `int`, `str`, `float` etc. It can also be a value stored in a variable. Simple examples of **`print()`** function executed in interactive mode of Python are given as follows:

Example

Display messages using `print()`

```
>>> print('Hello Welcome to Python Programming')
Hello Welcome to Python Programming
```

```
>>> print(10000)
10000
>>>print("Display String Demo")
Display String Demo
```

Suppose you want to print a message with quotation marks in the output as

print("The flight attendant asked, "May I see your boarding pass?")

If you try to run the above statement as is, Python will show an error. For Python, the second quotation mark is the end of the string and hence it does not know what to do with the rest of the characters. To overcome this problem Python has a special notation to represent a special character. This special notation consists of a **backslash (\)** followed by a letter or a combination of digits and is called an **escape sequence**. Using **backslash**, the special characters within **print** can be written as shown below.

Example

```
>>> print("The flight attendant asked,\"May I see your boarding pass?\" ")
The flight attendant asked, "May I see your boarding pass?"
```

In the above example, we have used backslash before the quotation marks to display the quotation marks in the output.

Table 2.4 illustrates a list of escape sequences used in Python.

Table 2.4 Python escape sequences

Character Escape Sequence	*Name*
\'	Single Quote
\"	Double Quote
\n	Linefeed
\f	Formfeed
\r	Carriage return
\t	Tab
\\	Backslash
\b	Backspace

Note: The syntax of **print** function is different in Python 2.X. It is

print arguments

Python 2.X does not use an additional parenthesis. If you try to execute the print statement without parenthesis, unlike Python 3, it will raise a syntax error.

Example:

```
>>> print 'Hello World'
```

Syntax Error: Missing parentheses in call to 'print'

(Contd.)

Python programs are case sensitive. Python raises an error if a programmer tries to replace **print** by Print.

Example:

```
>>> Print('hi')
Traceback (most recent call last):
File "<pyshell#3>", line 1, in <module>
Print('hi')
NameError: name 'Print' is not defined
```

2.5.1 The `print()` Function with end Argument

Consider a simple program of a print statement.

PROGRAM 2.1 Write a program to display the messages **"Hello"**, **"World"** and **"Good Bye"**. Each of the three messages should get displayed on a different line.

```
print('Hello')
print('World')
print('Good Bye')
```

Output

```
Hello
World
Good Bye
```

In the above program, we have displayed each message in a different line. In short, the print function automatically prints a **linefeed** (\n) to cause the output to advance to the next line. However, if you want to display the messages **"Hello" "World"** and **"Good Bye"** in one line without using a single print statement, then you can invoke the **print** function by passing a special argument named end=' '. The following program illustrates the use of the end argument within the print function.

PROGRAM 2.2 Write a basic program to make use of the end key and display the messages "Hello" "World" and "Good Bye" in one line.

```
print(' Hello',end=' ')
print(' World',end=' ')
print(' Good Bye')
```

Output

```
Hello World Good Bye
```

2.6 ASSIGNING VALUE TO A VARIABLE

In Python, the equal sign (=) is used as the assignment operator. The statement for assigning a value to a variable is called an **assignment statement**. The syntax used to assign value to a variable or identifier is:

```
Variable = expression
```

In the above syntax, expression may contain information in terms of values, even some time expression may contain operands with operators which evaluates to a value.

Let us consider the following example of assigning and displaying the value of a variable in Python interactive mode.

Example

```
>>> Z = 1                          # Assign value 1 to variable Z
>>> Z                              # Display value of Z
1
>>> radius = 5                     #Assign value 5 to the variable radius
>>> radius                         #Display value of variable radius
5
>>> R = radius + Z                 #Assign the addition of radius and Z to R
>>> R                              #Display value of Variable R
6
>>> E =(5 + 10 * (10 + 5))         #Assign the value of the expression to E
>>> E
155
```

This example explains how a variable can be used to assign a value and how a variable can be used on both the sides of = operator. As given in the above example:

R = radius + Z

In the above assignment statement, the result of **radius+Z** is assigned to R. Initially the value assigned to Z is 1. Once Python executes the above statement, it adds the most recent value of Z and assigns the final value to a variable R.

> **Note:** To assign a value to a variable, you must place the variable name to the left of the assignment operator. If you write in the following manner, Python will display an error.
>
> ```
> >>> 10 = X
> Syntax Error: can't assign to literal
> ```
>
> In Mathematics, E = (5 + 10 * (10 + 5)), denotes an equation, but in Python E = (5 + 10 * (10 + 5)) is an assignment statement that evaluates the expression and assigns the result to E.

2.6.1 More on Assigning Values to Variables

Consider the following example where a value has been assigned to multiple variables.

Example

```
>>> P = Q = R = 100                #Assign 100 to P, Q and R
>>> P                              #Display value of Variable P
100
>>> Q                              #Display value of Variable Q
100
>>> R                              #Display Value of Variable R
100
```

In the above example, we have assigned value 100 to **P, Q** and **R**. The statement **P = Q = R = 100** is equivalent to

P = 100
Q = 100
R = 100

2.6.2 Scope of a Variable

Each variable has a scope. The scope of a variable is a part of the program where a variable can be referenced. More details on scope of variables are given in Chapter 6. Consider the following simple example and run it on Python interpreter.

```
>>> C = Count + 1
Traceback (most recent call last):
     File "<pyshell#9>", line 1, in <module>
          C = Count + 1
NameError: name 'Count' is not defined
```

In the above example, we have written a statement as **C = Count + 1,** but when Python tries to execute the above statement it raises an error, viz. **"Count is not defined"**. To fix the above error in Python the variable must be assigned some value before it is used in an expression. Thus, the correct version of the above code written in Python interactive mode is as given as follows:

```
>>> Count = 1
>>> C = Count + 1
>>> C
2
```

> **Note:** A variable must be assigned a value before it can be used in an expression.

2.7 MULTIPLE ASSIGNMENTS

Python supports simultaneous assignment to multiple variables. The syntax of multiple assignments is

```
Var1, Var2, Var3, ..... = Exp1, Exp2, Exp3, ..... ExpN
```

In the above syntax, Python simultaneously evaluates all the expressions on the right and assigns them to a corresponding variable on the left.

Consider the following statements to swap the values of the two variables P and Q. The common approach to swap the contents of the two variables is shown as follows:

Example

```
>>> P = 20
>>> Q = 30
>>> Temp = P          #Save value of variable P into a variable Temp
>>> P = Q             #Assign value of Q to P
>>> Q = Temp          #Assign the value of Temp to Q
#After Swapping the value of P, and Q are as follows.
>>> P
30
>>>
>>> Q
20
```

In the above code, we have used the following statements to swap the values of the two variables P and Q.

```
Temp = P
    P = Q
    Q = Temp
```

However, by using the concept of multiple assignment, you can simplify the task of swapping two numbers.

```
>>> P, Q = Q, P     #Swap P with Q & Q with P
```

Thus, the entire code to swap two numbers using multiple assignment is as follows:

```
>>> P = 20          #Initial Values of P and Q
>>> Q = 30
>>> P
20
>>> Q
30
>>> P, Q = Q, P #Swap values of P and Q
>>> P               #Display value of P
30
>>> Q               #Display Value of Q
20
```


2.8 WRITING SIMPLE PROGRAMS IN PYTHON

How can a simple program to calculate the area of a rectangle be written in Python?

We know that a program is written in a step-wise manner. Consider the initial steps given as follows:.

⊙ **STEP 1:** Design an algorithm for the given problem.

An algorithm describes how a problem is to be solved by listing all the actions that need to be taken. It also describes the order in which the series of actions need to be carried out. An algorithm helps a programmer to plan for the program before actually writing it in a programming language. Algorithms are written in simple English language along with some programming code.

⊙ **STEP 2:** Translate an algorithm to programming instructions or code.

Let us now write an algorithm to calculate the area of a rectangle.

Algorithm to Calculate the Area of a Rectangle

a. Get the length and breadth of the rectangle from the user.

b. Use the relevant formula to calculate the area

Area = Length * Breadth

c. Finally display the area of the rectangle.

This algorithm can be written as code as shown in program 2.3.

PROGRAM 2.3 | Write a program to calculate the area of a rectangle.

```
Length = 10
breadth = 20
print(' Length = ',length,' Breadth = ',breadth)
area = length * breadth
print(' Area of Rectangle is = ',area)
```

Output

```
Length = 10 Breadth = 20
Area of Rectangle is = 200
```

Explanation In the above program, two variables, viz. length and breadth are initialized with values 10 and 20, respectively. The statement **area = length x breadth** is used to compute the area of the rectangle.

Here the values of the variables are fixed. However, a user may want to calculate the area of different rectangles with different dimensions in future. In order to get the values according to the user's choice, a programmer must know how to read the input values from the console. This is described in the next section.

2.9 THE input() FUNCTION

The input() function is used to accept an input from a user. A programmer can ask a user to input a value by making use of input().

input() function is used to assign a value to a variable.

Syntax

```
Variable_Name = input()
        OR
Variable_Name = input('String')
```

2.9.1 Reading String from the Console

A simple program of input() function to read strings from the keyboard is given in Program 2.4.

PROGRAM 2.4 | Write a program to read strings from the keyboard.

```
Str1 = input('Enter String1:')
Str2 = input('Enter String2: ')
print(' String1 = ',Str1)
print(' String2 = ',Str2)
```

Output

```
Enter String1:Hello
Enter String2: Welcome to Python Programming
     String1 = Hello
     String2 = Welcome to Python Programming
```

Explanation The input() function is used to read the string from the user. The string values entered from the user are stored in two separate variables, viz. **Str1** and **Str2.** Finally all the values are printed by making use of print() function.

Let us also check what happens if by mistake the user enters digits instead of characters. Program 2.5 illustrates the same.

PROGRAM 2.5 | Write a program to enter digits instead of characters.

```
print(' Please Enter the Number:')
X = input()
print(' Entered Number is: ',X)
print(' Type of X is:')
print(type(X))
```

(Contd.)

Output

```
Please Enter the number:
60
Entered Number is: 60
Type of X is:
<class 'str'>
```

Explanation We know that Python executes statements sequentially. Hence, in the above program the first print statement is printed, i.e. 'Please Enter the Number.' But when it runs the second statement, i.e. X = input() the programming execution stops and waits for the user to type the text using the keyboard. The text that the user types is not committed until he/she presses Enter. Once the user enters some text from the keyboard, the value gets stored in an associated variable. Finally, the entered value is printed on the console. The last statement is used to check the type of value entered.

Note: The input function produces only string. Therefore, in the above program even if the user enters a numeric, i.e. integer value, Python returns the type of input value as string.

In the above program, how does a programmer read integer values using the input function?

Python has provided an alternative mechanism to convert existing string to int. A programmer can use int to convert a string of digits into an integer. Program 2.6 illustrates the use of int and input ().

PROGRAM 2.6 | Write a program to demonstrate the use of int and input function.

```
print(' Please Enter Number')
Num1 = input()                          #Get input from user
print(' Num1 = ',Num1)                  #Print value of Num1
print(type(Num1))                       #Check type of Num1
print(' Converting type of Num1 to int ')
Num1 = int(Num1)                        #Convert type of Num1 from str to int
print(Num1)                             #print the value of Num1
print(type(Num1))                       #Check type of Num1
```

Output

```
Please Enter Number
12
 Num1 = 12
<class 'str'>
 Converting type of Num1 to int
12
<class 'int'>
```

Explanation The above program asks the user for input. The user has entered the input as 12 b it is of type str. By making use of int, i.e. the statement **Num1 = int(Num1)**, it converts the existi type to int.

We can minimise the number of lines in a program directly by making use of **int** before **inpu** function. A shorter version of the above program is given in Program 2.7.

PROGRAM 2.7 | Write a program to demonstrate the use of int before input.

```
Num1 = int(input(' Please Enter Number:'))
print(' Num1 = ',Num1) #Print the value of Num1
print(type(Num1)) #Check type of Num1
```

Output

```
Please Enter Number:
20
Num1 = 20
<class 'int'>
```

PROGRAM 2.8 | Write a program to read the length and breadth of a rectangle from a user and display the area of the rectangle.

```
print(' Enter Length of Rectangle:', end=' ')
Length = int(input()) #Read Length of Rectangle
print(' Enter Breadth of Rectangle:', end=' ')
Breadth = int(input()) #Read Breadth of Rectangle
Area = Length * Breadth #Compute Area of Rectangle
print('-----Details of Rectangle------')
print(' Length = ',Length) #Display Length
print(' Breadth = ',Breadth) #Display Breadth
print(' Area of rectangle is :',Area)
```

Output

```
Enter Length of Rectangle: 10
Enter Breadth of Rectangle: 20
-----Details of Rectangle------
Length = 10
Breadth = 20
Area of rectangle is: 200
```


Note: A programmer can make use of any type to convert the string into a specific type.
Example:

```
X = int(input())      #Convert it to int
X = float(input())    #Convert it to float
```

PROGRAM 2.9 | Write a program to add one integer and floating type number.

```
print('Enter integer number: ',end='')
Num1 = int(input()) # Read Num1
print('Enter Floating type number:',end='')
Num2 = float(input()) #Read Num2
print(' Number1 = ',Num1) #Print Num1
print(' Number2 =',Num2) #Print Num2
sum = Num1 + Num2 #Calculate Sum
print(' sum = ',sum) #Display Sum
```

Output

```
Enter integer number: 2
Enter Floating type number:2.5
Number1 = 2
Number2 = 2.5
sum = 4.5
```

Note: Python 3 uses **input()** method to read the input from the user.
Python 2 uses **raw_input()** method to read the input from the user.
In subsequent programs in this chapter we are going to use **input()** method only as all programs are executed in Python 3.

2.10 THE eval() FUNCTION

The full form of **eval** function is to evaluate. It takes a string as parameter and returns it as if it is a Python expression. For example, if we try to run the statement **eval('print("Hello")')** in Python interactive mode, it will actually run the statement **print("Hello")**.

Example

```
>>> eval('print("Hello")')
Hello
```

The eval function takes a string and returns it in the type it is expected. The following example illustrates this concept.

Example

```
>>> X = eval('123')
>>> X
123
>>> print(type(X))
<class 'int'>
```

2.10.1 Apply eval() to input() Function

In the previous section we learnt about the input() function in detail. We know that the **input()** function returns every input by the user as string, including numbers. And this problem was solved by making use of **type** before **input()** function.

Example

```
X = int (input('Enter the Number'))
```

Once the above statement is executed, Python returns it into its respective type.

By making use of **eval()** function, we can avoid specifying a particular type in front of **input()** function. Thus, the above statement,

```
X = int (input('Enter the Number'))
```

can be written as:

```
X = eval(input('Enter the Number'))
```

With respective to the above statement, a programmer does not know what values a user can enter. He/she may enter a value of any type, i.e. int, float, string, complex etc. By making use of eval, Python automatically determines the type of value entered by the user. Program 2.10 demonstrates the use of **eval()**.

PROGRAM 2.10 | Write a program to display details entered by a user, i.e. name, age, gender and height.

```
Name = (input('Enter Name :'))
Age = eval(input('Enter Age :')) #eval() determine input type
Gender = (input('Enter gender:'))
Height = eval(input('Enter Height:')) #eval() determine input type
print(' User Details are as follows: ')
print(' Name: ',Name)
print(' Age: ',Age)
print(' Gender: ',Gender)
print(' Height ',Height)
```

Output

```
Enter Name: Donald Trump
```

(Contd.)


```
Enter Age: 60
Enter Gender: M
Enter Height:5.9
User details are as follows:
Name: Donald Trump
Age: 60
Gender: M
Height: 5.9
```

Explanation In the above program we have used **eval()** in front of **input()** function as:

```
Age = eval(input('Enter Age :'))
```

The above statement reads the input as a string and converts a string into a number. After the user enters a number and presses Enter, the number is read and assigned to a variable name.

2.11 FORMATTING NUMBER AND STRINGS

A formatting string helps make string look presentable to the user for printing. A programmer can make use of **format** function to return a formatted string. Consider the following example to calculate the area of a circle before using this function.

PROGRAM 2.11 | Write a program to calculate the area of a circle.

```
radius = int(input('Please Enter the Radius of Circle: '))
print(' Radius = ', radius)              #Print Radius
PI = 3.1428                              #Initialize value of PI
Area = PI * radius * radius              #Calculate Area
print(' Area of Circle is: ',Area)       #Print Area
```

Output

```
Please Enter the Radius of Circle: 4
Radius = 4
Area of Circle is: 50.2848
```

In the above program, the user entered the radius as 4. Thus, for a circle having radius 4, it has displayed the area as **50.2848**. To display only two digits after the decimal point, make use of **format()** function. The syntax of format function is

```
format(item, format-specifier)
```

item is the number or string and **format-specifier** is a string that specifies how the item is formatted. A simple example of **format()** function executed in Python interactive mode is given as follows:

Example

```
>>> x = 12.3897                    #Assign value to variable x
>>> print(x)                       # print x
12.3897
>>> format(x,'.2f')                #Return formatted string
'12.39'
```

PROGRAM 2.12 | Make use of `format()` function and display the area of a circle.

```
radius = int(input('Please Enter the Radius of Circle: '))
print(' Radius = ', radius)
PI = 3.1428
Area = PI * radius * radius
print(' Area of Circle is: ',format(Area,'.2f'))
```

Output

```
Please Enter the Radius of Circle: 4
Radius = 4
Area of Circle is: 50.28
```

In the above program, the statement, **print ('Area of Circle is: ',format(Area,'.2f'))** is used to display the area of the circle. Within format function, **Area** is an item and **'.2f'** is the format specifier which tells the Python interpreter to display only two digits after the decimal point.

2.11.1 Formatting Floating Point Numbers

If the item is a float value, we can make use of specifiers to give the **width** and **precision**. We can use **format** function in the form **width.precisionf**. Precision specifies the number of digits after the decimal point and width specifies the width of the resultant string. In **width.precisionf** 'f' is called conversion code. 'f' indicates the formatting for floating point numbers. Examples of floating point numbers are

```
print(format(10.345,"10.2f"))
print(format(10,"10.2f"))
print(format(10.32245,"10.2f"))
```

displays the output as follows:

```
10.35
10.00
10.32
```

In above example, print statement uses 10.2f as the format specifier. 10.2f is explained in detail in **Fig. 2.2.**

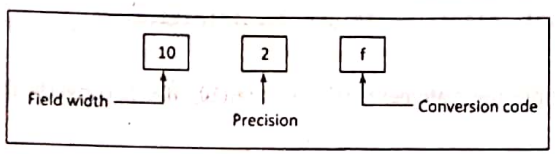
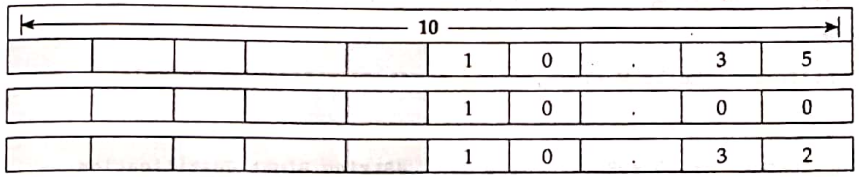


Figure 2.2 Format specifier details

The actual representation of the above output is:



The gray box denotes a blank space. The decimal point is also counted as 1.

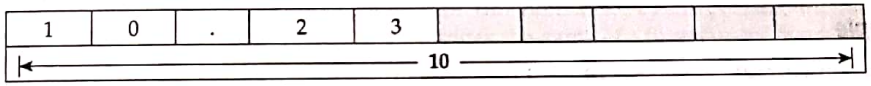
2.11.2 Justifying Format

By default, the integer number is right justified. You can insert **<** in the format specifier to specify an item to be left justified. The following example illustrates the use of right and left justification.

Example

```
>>>print(format(10.234566,"10.2f")) #Right Justification Example
 10.23
>>> print(format(10.234566,"<10.2f"))  #Left Justification Example
10.23
```

The actual representation of the above output for left justification is:



2.11.3 Integer Formatting

In case of integer formatting, you can make use of conversion code **d** and **x. d** indicates that the integer is to be formatted, whereas **x** specifies that the integer is formatted into a hexadecimal integer. The following example illustrates integer formatting.

Example

```
>>>print(format(20,"10x"))    #Integer formatted to Hexadecimal Integer
 14
>>> print(format(20,"<10x"))
14
```

```
>>> print(format(1234,"10d"))    #Right Justification
 1234
```

In the above example the statement **print(format(20,"10x"))** converts the number 20 into hexadecimal, i.e. 14.

2.11.4 Formatting String

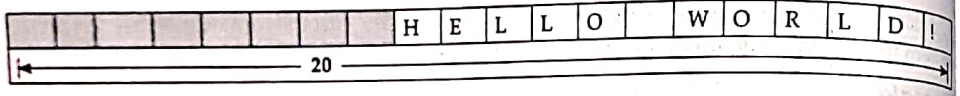
A programmer can make use of conversion code **s** to format a string with a specified width. However, by default, string is left justified. Following are some examples of string formatting.

Example

```
>>> print(format("Hello World!","25s") #Left Justification Example
Hello World!

>>>print(format("HELLO WORLD!",">20s"))    #String Right Justification
         HELLO WORLD!
```

In the above example, the statement **(format("HELLO WORLD!",">20s"))** displays the output as:



In print function 20 specifies the string to be formatted with a width of 20. In the second print statement, ">" is used for right justification of the given string.

2.11.5 Formatting as a Percentage

The conversion code **%** is used to format a number as a percentage. The following example illustrates the same.

Example

```
>>>  print(format(0.31456,"10.2%"))
     31.46%
>>>  print(format(3.1,"10.2%"))
     310.00%
>>>  print(format(1.765,"10.2%"))
     176.50%
```

In the above example, the statement **print(format(0.31456,"10.2%"))**, contains the format specifier **10.2%**. It causes the number to be multiplied by 100. The **10.2%** denotes the integer to be formatted with a width of 10. In width, % is counted as one space.

2.11.6 Formatting Scientific Notation

While formatting floating point numbers we have used the conversion code **f**. However, if we want to format a given floating point number in scientific notation then the conversion code **e** will be used. An example of formatting floating point numbers is given as follows:

Example

```
>>> print(format(31.2345,"10.2e"))
      3.12e+01
>>> print(format(131.2345,"10.2e"))
      1.31e+02
```

Most frequently used specifiers are shown in Table 2.5.

Table 2.5 Frequently used specifiers

Specifier	*Format*
10.2f	Format floating point number with precision 2 and width 10.
<10.2f	Left Justify the floating point number.
>10.2f	Right Justify the formatted item.
10X	Format integer in hexadecimal with width 10
20s	Format String with width 20
10.2%	Format the number in decimal

2.12 PYTHON INBUILT FUNCTIONS

In the previous sections of this chapter we have learnt how to use the functions **print, eval, input** and **int**. We know that a function is a group of statements that performs a specific task. Apart from the above functions, Python supports various inbuilt functions as well. A list of all inbuilt functions supported by Python is given in **Table 2.6**. It provides the name of a function, its description and examples executed in Python interactive mode.

Table 2.6 Inbuilt functions in Python

Function	*Description*
abs(x)	Returns absolute value of x

Example
`>>> abs(-2)`
`2`
`>>> abs(4)`
`4`	

(Contd.)

`max(x1, X2, X3,.....,XN)`	Returns largest value among X1, X2, X3, X4,... , XN
Example:	
`>>> max(I0,20,30,40)`	
`40`	
`max(x1, X2, X3,.....,XN)`	Returns minimum value among X1, X2, X3, X4,... , XN.
`pow(X, Y)`	Return the X^Y
Example:	
`>>> pow(2,3)`	
`8`	
`round(x)`	Returns an integer nearest to the value of x.
Example:	
`>>> round(10.34)`	
`10`	
`>>> round(10.89)`	
`11`	

Functions given in Table 2.6 are not enough to solve mathematical calculations. Thus, Python has an additional list of functions defined under Python's **math** module to solve problems related to mathematical calculations. List of functions under the **math** module is given in Table 2.7.

Table 2.7 Inbuilt mathematical functions in Python

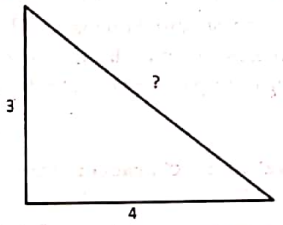
Function	*Example*	*Description*
`ceil(X)`	`>>> math.ceil(10.23)` `11`	Round X to nearest integer and returns that integer.
`floor(X)`	`>>> math.floor(18.9)` `18`	Returns the largest value not greater than X
`exp(X)`	`>>> math.exp(1)` `2.718281828459045`	Returns the exponential value for e^x
`log(X)`	`>>> math.log(2.71828)` `0.999999327347282`	Returns the natural logarithmic of x (to base e)
`log(x,base)`	`>>> math.log(8,2)` `3.0`	Returns the logarithmic of x to the given base
`sqrt(X)`	`>>>math.sqrt(9)` `3.0`	Return the square root of x
`Sin(X)`	`>>> math.sin(3.14159/2)` `0.9999999999991198`	Return the sin of X, where X is the value in radians
`asin(X)`	`>>> math.asin(1)` `1.5707963267948966`	Return the angle in radians for the inverse of sine
`cos(X)`	`>>> math.cos(0)` `1.0`	Return the sin of X, where X is the value in radians
`aCos(X)`	`>>> math.acos(1)` `0.0`	Return the angle in radians for the inverse of cosine

(Contd.)

`tan(X)`	`>>> math.tan(3.14/4)` `0.9992039901050427`	Return the tangent of X, where X is the value in radians
`degrees(X)`	`>>> math.degrees(1.57)` `89.95437383553924`	Convert angle X from to radians to degrees

`Radians(X)`	`>>> math.`
	`radians(89.99999)`
	`1.5707961522619713`
		Convert angle x from degrees to radians

PROGRAM 2.13 | Write a program to calculate the hypotenuse of the right-angled triangle given as follows:



Hypotenuse = Square_Root {(Base)^2 + (Height)^2}

= Square_Root {(3)^2 + (4)^2}

= 5

from math import *

```
import math #Import Math Module
Base = int(input('Enter the base of a right-angled triangle:'))
Height = int(input('Enter the height of a right-angled triangle:'))
print(' Triangle details are as follows: ')
print(' Base = ',Base)
print(' Height = ',Height)
Hypotenuse = math.sqrt(Base * Base + Height * Height )
print(' Hypotenuse =',Hypotenuse)
```

Output

```
Enter the base of a right-angled triangle:3
Enter the height of a right-angled triangle:4
Triangle details are as follows:
Base = 3
Height = 4
Hypotenuse = 5.0
```

Explanation In the above program, the first line '**import math**' is used to include all in-built functions supported by Python under the math module. The **input** function is used to read the

base and height of the right-angled triangle. The statement **math.sqrt** is executed to find the square root of the number. Finally, print for the hypotenuse of the right-angled triangle is given.

2.12.1 The ord and chr Functions

As we know, a string is a sequence of characters. It can include both text and numbers. All these characters are stored in a computer as a sequence of 0s and 1s. Therefore, a process of mapping a character to its binary representation is called **character encoding**.

There are different ways to encode a character. The **encoding scheme** decides the manner in which characters are encoded. The American Standard Code for Information Interchangeable **(ASCII)** is one of the most popular encoding schemes. It is a 7-bit encoding scheme for representation of all lower and upper case letters, digits and punctuation marks. The ASCII uses numbers from 0 to 127 to represent all characters. Python uses the in-built function **ord(ch)** to return the ASCII value for a given character. The following example demonstrates the use of the in-built function ord().

Example

```
>>> ord('A') #Returns ASCII value of Character 'A'
65
>>> ord('Z') #Returns ASCII Value of Character of 'Z'
90
>>> ord('a') #Returns ASCII Value of Character of 'a'
97
>>> ord('z') #Returns ASCII Value of Character of 'z'
122
```

The **chr(Code)** returns the character corresponding to its code, i.e. the ASCII value. The following example demonstrates the use of in-built function chr().

Example

```
>>> chr(90)
'Z'
>>> chr(65)
'A'
>>> chr(97)
'a'
>>> chr(122)
'z'
```

PROGRAM 2.14 Write a program to find the difference between the ASCII code of any lower case letter and its corresponding upper case letter.

```
Char1 = 'b'
Char2 = 'B'
print('Letter\tASCII Value')
```

(Contd.)


```
print(Char1,'\t',ord(Char1))
print(Char2,'\t',ord(Char2))
print(' Difference between ASCII value of two Letters:')
print(ord(Char1),'-',ord(Char2),'=', end=' ')
print(ord(Char1)-ord(Char2))
```

Output

```
Letter ASCII Value
b    98
B    66
Difference between ASCII value of two Letters:
98 - 66 = 32
```

Explanation In the above program, the letter 'b' is stored in variable **Char1** and the letter 'B' is stored in variable **Char2.** The **ord()** function is used to find the ASCII value of the letters. Finally, the statement **ord(Char1)- ord(Char2)** is used to find the difference between the ASCII values of the two letters **Char1** and **Char2.**

SUMMARY

- Python breaks each statement into a sequence of lexical components called tokens.
- Literals are numbers, strings or characters that appear directly in a program.
- Python offers an inbuilt method called type to know the exact type of any value.
- Keywords are reserved words.
- Keywords cannot be used as identifiers or variables.
- An identifier is a name used to identify a variable, function, class or other objects.
- Everything in Python is an object.
- The `int` function converts a string or a number into a whole number or integer.
- The float function converts a string into a floating-point number.
- The Boolean data type is represented in Python as of type bool.
- print function is used to display contents on the screen.
- `input()` function is used to accept input from the user.
- `format()` function can be used to return a formatted string.

KEY TERMS

- **chr():** Returns a character for a given ASCII value
- **end():** Used as argument with print() function
- **format():** Formats string and integer
- **Identifier:** Name to identify a variable

3 Operators and Expressions

CHAPTER OUTLINE

LEARNING OUTCOMES

After completing this chapter, students will be able to:

- Perform simple arithmetic operations
- Explain the difference between division and floor division operators
- Use unary, binary and bitwise operators, and perform multiplication and division operations using bitwise left and right shift operators
- Evaluate numeric expressions and translate mathematical formulae into expressions
- Recognise the importance of associativity and operator precedence in programming languages

3.1 INTRODUCTION

An operator indicates an operation to be performed on data to yield a result. In our day to day life, we use various kinds of operators to perform diverse data operations. Python supports different

operators which can be used to link variables and constants. These include arithmetic operators, Boolean operators, bitwise operators, relational operators and simple assignment and compound assignment operators.

Table 3.1 lists basic operators in Python with their symbolic representation

Table 3.1 Types of operators

Type of Operator	*Symbolic Representation*
Arithmetic Operators	+, - , / ,// *, %, %%
Boolean Operators	and, or, not
Relational Operators	>, <, <=, >= ,!=
Bitwise Operators	&, | , ^, >> , << , ~
Simple Assignment and Compound Assignment Operators	= , +=,*=,/=,%=,**=

3.2 OPERATORS AND EXPRESSIONS

Most statements contain expressions. An expression in Python is a block of code that produces a result or value upon evaluation. A simple example of an expression is **6 + 3**. An expression can be broken down into operators and operands. Operators are symbols which help the user or command computer to perform mathematical or logical operations. In the expression **6 + 3**, the '+' acts as the operator. An operator requires data to operate and this data is called **operand**. In this example, **6** and **3** are the operands.

The following sections describe the various kinds of operators and their usage. The expressions given in the examples are executed in Python interactive mode.

3.3 ARITHMETIC OPERATORS

There are two types of arithmetic operators in Python, viz. binary and unary (as shown in Fig. 3.1).

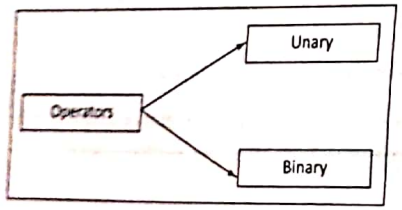


Figure 3.1 Types of arithmetic operators

3.3.1 Unary Operators

Unary arithmetic operators perform mathematical operations on one operand only. The '+' and '-' are two unary operators. The unary operator minus (-) produces the negation of its numeric operand. The unary operator plus (+) returns the numeric operand without change. Table 3.2 gives the details of unary operators.

Table 3.2 Unary operators

Unary Operator	*Example*	*Description*
+	+X (+X returns the same value, i.e. X)	Returns the same value that is provided as input
-	-X (-x returns the negation of x)	Negates the original value so that the positive value becomes negative and vice versa

Examples of Unary Operators

```
>>> x=-5 #Negates the value of X
>>> x
-5
>>> x=+6 #Returns the numeric operand, i.e. 6, without any change
>>> x
6
```

Some More Complex Examples of Unary Operators

```
>>> +-5
-5
```

In the above expression **+-5**, the first '+' operator represents the unary plus operation and the second '-' operator represents the unary minus operation. The expression **+-5** is equivalent to **+(-(5))**, which is equal to **-5**.

```
>>> 1--3  #Equivalent to 1-(-3)
4
>>> 2---3 #Equivalent to 2-(-(-3))
-1
>>> 3+--2 #Equivalent to 3+(-(-2))
5
```

3.3.2 Binary Operators

Binary operators are operators which require two operands. They are written in **infix** form, i.e. the operator is written in between two operands.

The Addition (+) Operator

The '+' operator in Python can be used with binary and unary form. If the addition operator is applied in between two operands, it returns the result as the arithmetic sum of the operands. Some examples of addition operators executed in Python interactive mode are given as follows:

Example

```
>>> 4+7      #Addition
11

>>>5+5       Addition
10
```

Table 3.3 explains the syntax and semantics of the addition operator in Python, using its three numeric types, viz. **int, float and complex.**

Table 3.3 Addition operator

Syntax	*Example*
(int, int)-> int	2+4 returns 6
(float, float)->float	1.0+4.0 returns 5.0
(int, float)->float	1+2.0 returns 3.0
(float, int)->float	2.0+1 returns 3.0
(complex, complex)->complex	3j+2j returns 5j

The Subtraction (-) Operator

The '-' operator in Python can be used with binary and unary form. If the subtraction operator is applied in between two operands, the result is returned as the arithmetic difference of the operands. Some examples of subtraction operators executed in Python interactive mode are given as follows:

Example

```
>>> 7-4      #Subtraction
3

>>>5-2       #Subtraction
3
```

Table 3.4 explains the syntax and semantics of the subtraction operator in Python, using its three numeric types, viz. **int, float and complex.**

Table 3.4 Subtraction operator

Syntax	*Example*
(int, int)-> int	4-2 returns 2
(float, float)->float	3.5-1.5 returns 2.0
(int, float)->float	4-1.5 returns 2.5
(float, int)->float	4.0-2 returns 2.0
(complex, complex)->complex	3j-2j returns 1j

PROGRAM 3.1 Read the cost and selling price of an object and write a program to find the profit earned by a seller (in rupees). The selling price is greater than the cost price.

```
SP=eval(input('Enter the Selling Price of an Object:'))
CP=eval(input('Enter the Cost Price of an Object:'))
print('-----------------------------------------')
print(' Selling Price = ',SP)
print(' Cost Price =',CP)
print('-----------------------------------------')
Profit=SP - CP            #Formula to Calculate Profit
print(' Profit Earned by Selling  = ',Profit)
```

Output

```
Enter the Selling Price of an Object: 45
Enter the Cost Price of an Object: 20
-----------------------------------------
 Selling Price =  45
 Cost Price = 20
-----------------------------------------
 Profit Earned by Selling  =  25
```

Explanation At the start of the program, the selling price and cost price of the object is read using eval. The statement, **Profit = SP - CP** is executed to calculate the profit earned by the seller.

The Multiplication (*) Operator

The '*' operator in Python can be used only with binary form. If the multiplication operator is applied in between two operands, it returns the result as the arithmetic product of the operands. Some examples of multiplication operators executed in Python interactive mode are given as follows:

Example

```
>>> 7*4          #Multiplication
28
>>>5*2           #Multiplication
10
```

Table 3.5 explains the syntax and semantics of the multiplication operator in Python, using its three numeric types, viz. **int, float and complex.**

Table 3.5 Multiplication operator

Syntax	*Example*
(int, int)-> int	4*2 returns 8
(float, float)->float	1.5*3.0 returns 4.5
(int, float)->float	2*1.5 returns 3.0
(float, int)->float	1.5* 5 returns 7.5
(complex, complex)->complex	2j*2j returns -4+0j

PROGRAM 3.2 Write a program to calculate the square and cube of a number using * operator.

```
num=eval(input('Enter the number:'))          # Read Number
print('Number = ',num)
Square=num* num                               #Calculate Square
Cube = num * num * num                        #Calculate Cube
print('Square of a Number = ',num,' is ',Square)
print('Cube of a Number   = ',num,' is ',Cube)
```

Output

```
Enter the number: 5
Number =  5
Square of a Number =  5  is  25
Cube of a Number   =  5  is  125
```

The Division (/) Operator

The '/' operator in Python can be used only with binary form. If the division operator is applied in between two operands, it returns the result as the arithmetic quotient of the operands. Some examples of division operators executed in Python interactive mode are given as follows:

Example

```
>>> 4/2                              #Division
2.0

>>> 10/3
3.3333333333333335                   #Division
```

Table 3.6 explains the syntax and semantics of the division operator in Python, using its three numeric types, viz. **int, float and complex.**

Table 3.6 Division (/) operator

Syntax	*Example*
(int, int)-> float	25/5 returns 5.0
(float, float)->float	0.6/2.0 returns 0.3
(int, float)->float	4/0.2 returns 20.0
(float, int)->float	1.5/2 returns 0.75
(complex, complex)->complex	6j/2j returns 3+0j

Note: When the division (/) operator is applied on two int operands, Python returns a float result.

PROGRAM 3.3 Write a program to calculate simple interest (SI). Read the principle, rate of interest and number of years from the user.

```
P=eval(input('Enter principle Amount in Rs = ')) #Read P
ROI=eval(input('Enter Rate of Interest = ')) #Read ROI
years=eval(input('Enter the Number  of years ='))#Read years
print(' Principle = ',P)
print(' Rate of Interest = ',ROI)
print(' Number of Years = ',years)
SI = P*ROI*Years/100                  #Calculate SI
print('Simple Interest = ',SI)
```

Output

```
Enter Principle Amount in Rs = 1000
Enter Rate of Interest = 8.5
Enter the Number  of Years = 3
Principle =  1000
Rate of Interest =  8.5
Number of Years =  3
Simple Interest = 255.0
```

PROGRAM 3.4 Write a program to read a temperature in Celsius from the user and convert it into Fahrenheit.

```
Celsius =eval(input('Enter Degree is Celsius:'))#Read Celsius from User
print('Celsius = ', Celsius)   #Print Celsius
Fahrenheit = (9 / 5) * Celsius + 32 # Convert Celsius to Fahrenheit
print(' Fahrenheit = ', Fahrenheit) # Print Fahrenheit
```

(Contd.)

Output

```
Enter Degree is Celsius: 23
Celsius =  23
Fahrenheit =  73.4
```

Note: Formula to convert Celsius into Fahrenheit is:
Fahrenheit = (9/5)*Celsius + 32

The Floor Division (//) Operator

The '//' operator in Python can be used only with binary form. If the floor division operator is applied in between two operands, it returns the result as the arithmetic quotient of the operands. Some examples of floor division operators executed in Python interactive mode are given as follows:

Example

```
>>> 4//2 # Floor Division
2
>>> 10//3
3    #Floor Division
```

Table 3.7 explains the syntax and semantics of the floor division operator in Python, using its numeric types, viz. **int and float.**

Table 3.7 Floor division (//) operator

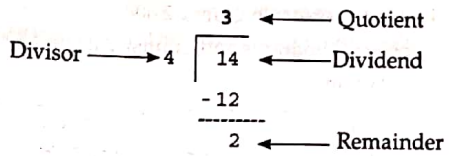
Syntax	*Example*
(int, int)-> int	25//5 returns 5
(float, float)->float	10.5//5.0 returns 2.0
(int, float)->float	11//2.5 returns 4.0
(float, int)->float	4.0//3 returns 1.0

Note: a. From the above example, it is clear that when the floor division (//) operator is applied on two **int** operands, Python returns an **int** result.
b. In the second example 10.5//5.0, the result returned is 2.0. However, if 10.5/5.0 returns 2.1, it means the floor division operator has been applied on two float operands. Hence, it returns the result in float but ignores the decimal number after the decimal point.

The Modulo (%) Operator

When the second number divides the first number, the modulo operator returns the remainder. The % modulo operator is also known as the remainder operator. If the remainder of x divided by y is zero then we can say that x is divisible by y or x is a multiple of y.

Consider the following example.



In the above example, **14 % 4** returns **3** as the remainder. Thus, the left-side operand, i.e. 14 is the dividend and the right-side operand, i.e. 4 is the divisor. Some more examples of the modulo operator executed in Python interactive mode are given below.

Example

```
>>> 10 % 4      # 10 is divided by 4 returns remainder as 2
2
>>> 13%5
3
```

Table 3.8 explains the syntax and semantics of the modulo (%) operator in Python, using its numeric types, viz. **int and float.**

Table 3.8 Modulo (%) operator

Syntax	*Example*
(int, int)-> int	25%4 returns 1
(float, float)->float	2.5 % 1.2 returns 0.10
(int, float)->float	13%2.0 returns 1.0
(float, int)->float	1.5 % 2 returns 1.5

Note: Mathematically, X%Y is equivalent to X – Y * (x//Y)

Example: 14%5 returns 4
Therefore,
14 % 5 = 14 – 5*(14//5)
= 14 – 5* (2)
= 14 – 10
= 4

Use of % Modulo Operator in Programming The modulo operator, i.e. the remainder operator is very useful in programming. It is used to check if a number is even or odd, i.e. if number % 2 returns zero then it is an even number and if number % 2 == 1 then it is an odd number.

PROGRAM 3.5 Write a program to read the weight of an object in grams and display its weight in kilograms and grams, respectively.

Example

Input: Enter the weight of the object in grams: 2,500
Output: Weight of the object (kilograms and grams): 2 kg and 500 g
Note: 1 kilogram = 1,000 grams

```
W1 = eval(input('Enter the Weight of Object in grams:')) #Input Weight
print(' Weight of Object = ',W1,' grams') # Print Weight
W2 = W1 // 1000  #Calculate No of kg
W3 = W1 % 1000   #Calculate No of g
print(' Weight of Object = ',W2,' kg and ',W3,' g')
```

Output

```
Enter the Weight of Object in g : 1250
Weight of Object =  1250  g
Weight of Object =  1  kg and  250  g
```

PROGRAM 3.6 | Write a program to reverse a four-digit number using **%** and **//** operators.

```
Num=eval(input('Enter four-digit number: '))
print('Entered number is:',num)
r1=num%10
q1=num//10
r2=q1%10
q2=q1//10
r3=q2%10
q3=q2//10
r4=q3%10
print('Reverse of ',num,'is:',r1,r2,r3,r4)
```

Output

```
Enter four-digit number: 8763
Entered number is: 8763
Reverse of 8763 is: 3 6 7 8
```

Explanation In the above program, initially the number is read from the user. For instance, the number read through the user is 8763. To reverse the contents of the number, initially the operation (8763 % 10) gives a remainder 3. To display the second digit 6, the number has to be divided by 10. Hence, (8763//10) gives 876. After obtaining the quotient as 876, the modulus operation (876%10) is performed again to obtain the digit 6. This process is continued three times to obtain the reverse of the four-digit number entered by the user.

The Exponent ** Operator

The '**' exponent operator is used to calculate the power or exponent of a number. To compute x^Y (X raised to Y), the expression is written as X**Y. The exponent operator is also called **power operator**.

Example

```
>>> 4**2    #Calculate Square of a Number 4
16
>>> 2**3    #Calculate Cube of a Number 2
8
```

Table 3.9 explains the syntax and semantics of the exponent (**) operator in Python, using its numeric types, viz. **int and float.**

Table 3.9 Exponent(**) operator

Syntax	*Example*
(int, int)-> int	2**4 returns 16
(float, float)->float	2.0**3.0 returns 8.0
(int, float)->float	5**2.0 returns 25.0
(float, int)->float	4.0 **3 returns 64.0

PROGRAM 3.7 | Write a program to calculate the distance between two points. The formula for computing distance is

$$\sqrt{(X2-X1)^2+(Y2-Y1)^2}$$

We can use Z**0.5 to compute the square root of the expression $\sqrt{Z}$. The program below prompts the user to read the coordinates of the two points and compute the distance between them.

```
print('Point1')
X1 = eval(input('Enter X1 coordinate:'))   #Read X1
Y1 = eval(input('Enter Y1 coordinate:'))   #Read Y1
print('point2')
X2 = eval(input('Enter X2 coordinate: ')) #Read X2
Y2 = eval(input('Enter Y2 coordinate: ')) #Read Y2
L1=(X2-X1)**2 + (Y2-Y1)**2   #Computer inner expression
Distance = L1**0.5 #Compute Square root.
print('Distance between two point is as follows')
print('(',X1,Y1,')','(',X2,Y2,')=', Distance)
```

Output

```
Point1
Enter X1 Coordinate :4
Enter Y1 Coordinate :6
```

(*Contd.*)

```
point2
Enter X2 Coordinate: 8
Enter Y2 Coordinate: 10
Distance between the two points is as follows
( 4 6 ) ( 8 10 )= 5.656854249492381
```

PROGRAM 3.8 | Write a program to display the following table.

X	Y	X**Y
10	2	100
10	3	1000
10	4	10000
10	5	100000

```
print('X  \t Y \t X**Y')
print('10 \t  2 \t ',10**2)
print('10 \t  3 \t ',10**3)
print('10 \t  4 \t ',10**4)
print('10 \t  5 \t ',10**5)
```

Output

```
X    Y    X**Y
10     2    100
10     3    1000
10     4    10000
10     5    100000
```

3.4 OPERATOR PRECEDENCE AND ASSOCIATIVITY

Operator precedence determines the order in which the Python interpreter evaluates the operators in an expression.

Consider the expression **4+5*3**.

Now, you may ask so how does Python know which operation to perform first? In the above example **4+5*3**, it is important to know whether **4+5*3** evaluates to 19 (where the multiplication is done first) or 27 (where the addition is done first).

The default order of precedence determines that multiplication is computed first so the result is 19. As an expression may contain a lot of operators, operations on the operands are carried out according to the priority, also called the precedence of the operator. The operator having higher priority is evaluated first.

Table 3.10 gives the list of operator precedence in the descending order. The operators on the top rows have higher precedence and the operators on the bottom rows have lower precedence. If a row contains multiple operators, it means all the operators are of equal priority or

Table 3.10 Operator precedence

Precedence	*Operator*	*Name*
↓	**	Exponential
	+,-,~	Plus, Minus, Bitwise not
	*,/,//,%	Multiplication, division, integer division, and remainder
	+, -	Binary Addition, Subtraction
	<<, >>	Left and Right Shift
	&	Bitwise AND
	^	Bitwise XOR
	|	Bitwise OR
	<,<=,>,>=	Comparison
	==, !=	Equality
	=,%=,/=,//=,-=,+=,*=,**=	Assignment Operators
	is, is not	Identity Operators
	in, not in	Membership Operator
	Not	Boolean Not
	And	Boolean and
	Or	Boolean or

3.4.1 Example of Operator Precedence

Consider arithmetic operators *, /, // and %, which have higher precedence as compared to operators + and -.

Example

4+5*3-10

As compared to + and * operators, the * operator has higher priority. Hence, the multiplication operation is performed first. Therefore, above expression becomes,

4+15-10

Now in above expression, + and – have the same priority. In such a situation, the leftmost operation is evaluated first. Hence, the above expression becomes

19 - 10

Consequentially, subtraction is performed last and the final answer of the expression will be 9.

3.4.2 Associativity

When an expression contains operators with equal precedence then the associativity property decides which operation is to be performed first. Associativity implies the direction of execution and is of two types, viz. left to right and right to left.

(i) Left to Right: In this type of expression, the evaluation is executed from the left to right.

4 + 6 – 3 + 2

In the above example, all operators have the same precedence. Therefore, associativity rule is followed (i.e. the direction of execution is from the left to right).

The evaluation of the expression 4+6-3+2 is equivalent to

=((4+6)-3)+ 2
= ((10)-3)+ 2
=(7)+2
= 9

(ii) Right to Left: In this type of expression, the evaluation is executed from the right to left.

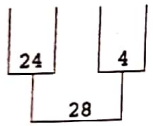
X = Y = Z = Value

In the above example, assignment operators are used. The value of Z is assigned to Y and then to X. Thus, the evaluation starts from the right.

Example of Associativity

(i) When operators of the same priority are found in an expression, precedence is given to the leftmost operator.

Z = 4 * 6 + 8 // 2



In the above expression * is evaluated first, even though * and // have the same priorities. The operator * occurs before // and hence the evaluation starts from the left. Therefore, the final answer for the above expression is 28.

The examples so far illustrated how Python uses associativity rules for evaluating expressions. Table 3.11 shows the precedence and associativity for arithmetic operators.

Table 3.11 Associativity table for arithmetic operators

Precedence	*Operators*	*Associativity*
Highest	()	Innermost to Outermost
	**	Highest
	*,/,//,%	Left to Right
Lowest	+-	Left to Right

3.5 CHANGING PRECEDENCE AND ASSOCIATIVITY OF ARITHMETIC OPERATORS

One can change the precedence and associativity of arithmetic operators by using (), i.e. the parentheses operator. The () operator has the highest precedence among all other arithmetic

operators. It can be used to force an expression to evaluate in any order. Parentheses operator () also makes an expression more readable.

Some examples of parentheses operator executed in Python interactive mode are given as follows:

Example

```
>>> z=(5+6)*10
>>> z
110
```

Explanation

In the above example, z is initialized with one expression **(5+6)*10**. The sub expression **(5 + 6)** is evaluated first, followed by the multiplication operation.

Some More Complex Examples

```
>>> A= 100 / (2*5)
>>> A
10.0

>>> B= 4 + (5 * (4/2) + (4 + 3))
>>> B
21.0
```

PROGRAM 3.9 Write a program to find the area and perimeter of a rectangle using **()**, i.e. the parenthesis operator.

```
Length  = eval(input('Enter the Length of Rectangle:'))
Breadth = eval(input('Enter the Breadth of Rectangle:'))
print('- - - - - - - - - - - - ')
print(' Length = ',Length)
print(' Breadth = ',Breadth)
print('- - - - - - - - - - - - ')
print(' Area = ', Length * Breadth)
print(' Perimeter = ',2 * (Length + Breadth))
```

Output

```
Enter the Length of Rectangle: 10
Enter the Breadth of Rectangle: 20
- - - - - - - - - - - -
 Length =  10
 Breadth =  20
- - - - - - - - - - - -
Area =  200
Perimeter =  60
```

Explanation In the above program, the values of variables length and breadth of the rectangle are initially read from the user. Then using the multiplication * operator, the area of the rectangle is computed. Finally, in order to compute the perimeter, the addition of length and breadth is performed and the result is multiplied by 2.

Note: Area of Rectangle = Length * Breadth
Perimeter of Rectangle = 2 * (Length + Breadth)

3.6 TRANSLATING MATHEMATICAL FORMULAE INTO EQUIVALENT PYTHON EXPRESSIONS

Consider the following quadratic equation written in normal arithmetic manner.

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

The steps required to convert this quadratic equation into its equivalent Python expression are given as follows:

- **STEP 1:** The numerator and denominator are computed first to find the roots of the quadratic equation. Division between the numerator and denominator is performed as the last step. Hence, we can write the above expression as:

  Numerator/Denominator

- **STEP 2:** The denominator is just 2a, so we can rewrite the formula as:

  Numerator/((2 *a))

- **STEP 3:** Now we can split the numerator into two parts, i.e. left and right as follows:

  (Left+Right)/((2 *a))

- **STEP 4:** Substitute –b for left. There is no need to put parenthesis for –b because unary operator has higher precedence than binary addition. Hence, the above equation becomes:

  (-b+Right)/((2 *a))

- **STEP 5:** The right contains the expression inside the square root. Therefore, the above equation can be rewritten as:

  (-b+sqrt(expression)/((2 *a))

- **STEP 6:** But the expression inside the square root contains two parts left and right. Hence, the above equation is further rewritten as

  (-b+sqrt(left-right)/((2 *a))

- **STEP 7:** Now the left part contains the expression b**2 and the right part contains the expression 4*a*c. There is no need to put parenthesis for b**2 because the exponent operator has higher precedence than the * operator since the expression 4*a*c is present on the right side. The above equation can be rewritten as

  (-b+sqrt(b**2-4*a*c)/((2 *a))

Thus, we have converted the mathematical expression into a Python expression. While converting an equation into a Python expression, one needs to only remember the rules of operator precedence and associativity.

PROGRAM 3.10 | Write the following numeric expression in Python and evaluate it.

$$\frac{2+8P}{2} - \frac{(P-Q)(P+Q)}{2} + 4 * \frac{(P+Q)}{2}$$

Consider the value of variables P and Q as 4 and 2, respectively.

```
P = 4
Q = 2
Z =( 2 + 8 * P) / 2 - ((P-Q)*(P+Q))/2 + 4 * ((P+Q)/2)
print('( 2 + 8 * P) / 2 - ((P-Q)*(P+Q))/2 + 4 * ((P+Q)/2)')
print(' Where P = ',P,' and Q = ', Q)
print(' Answer of above expression = ',Z)
```

Output

```
(2 + 8 * P) / 2 - ((P-Q)*(P+Q))/2 + 4 * ((P+Q)/2)
 Where P =  4  and Q =  2
 Answer of above expression =  23.0
```

Explanation In the above program, initially the equation

$$\frac{2+8P}{2} - \frac{(P-Q)(P+Q)}{2} + 4 * \frac{(P+Q)}{2}$$

is translated into a Python expression as

(2 + 8 * P) / 2 - ((P - Q) * (P + Q))/2 + 4 * ((P + Q)/2).

Once the expression is converted into a Python expression, the values of P and Q are substituted by the Python interpreter and finally the expression is evaluated considering Python precedence and associativity rules.

3.7 BITWISE OPERATOR

Python has six bitwise operators for bitwise manipulation. The bitwise operator permits a programmer to access and manipulate individual bits within a piece of data. Table 3.12. shows various bitwise operators supported by Python.

Table 3.12 Bitwise operators

Operator	*Meaning*
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
>>	Right Shift
<<	Left Shift
~	Bitwise NOT

3.7.1 The Bitwise AND (&) Operator

This operator performs **AND** operation on input bits of numbers. The Bitwise **AND** operator is represented as '**&**'. The '**&**' operator operates on two operands bit-by-bit. Table 3.13. explains the AND operator.

Table 3.13 AND operator

Input		*Output*
X	Y	X & Y
0	0	0
0	1	0
1	0	0
1	1	1

We can conclude from this table that the output is obtained by multiplying the input bits.

Example of AND Operator

```
>>> 1 & 3
1    # The bitwise & operator on 1 and 3 returns 1
>>> 5 & 4
4    # The bitwise & operator on 5 and 4 returns 4
```

Working of the bitwise operator is given as follows:

```
1 and 3 are converted into their equivalent binary format
                  0       0       0       1       (one)
            &
                  0       0       1       1       (Three)
            -------------------------------
Bitwise operation (0 & 0) (0 & 0) (0 & 1) (1 & 1)
            -------------------------------
          Result  0       0       0       1       (One)
            Decimal equivalent of (0 0 0 1) = 1
             Therefore, 1 & 3 = 1
```


PROGRAM 3.11 Write a program to read two numbers from the user. Display the result using bitwise & operator on the numbers.

```
num1 = int(input('Enter First Number: '))
num2 = int(input('Enter Second Number: '))
print(num1,' & ',num2,' = ', num1 & num2)
```

Output

```
#Test Case 1
Enter First Number: 1
Enter Second Number: 3
   1 &  3  =  1

#Test Case 2
Enter First Number: 5
Enter Second Number: 6
5  &  6  =  4
```

3.7.2 The Bitwise OR (|) Operator

This operator performs bitwise OR operation on the numbers. The bitwise OR operator is represented as '|'. It also operates on two operands and the two operands are compared bit-by-bit. Table 3.14 explains the '|' (OR) operator.

Table 3.14 Bitwise OR operator

Input		*Output*
X	Y	X | Y
0	0	0
0	1	1
1	0	1
1	1	1

We can conclude from this table that the output is obtained by adding the input bits.

Examples of Bitwise '|' (OR) Operator

```
>>> 3 | 5
7    # The bitwise | operator on 3 and 5 returns 7

>>> 1 | 5
5    # The bitwise | operator on 1 and 5 returns 5
```

Working of the bitwise OR (|) operator is given as follows:

```
Working of expression 3 | 5 is as below.
Initially 3 and 5 are converted into their equivalent binary format
                0       0       1       1       (Three)
              |
                0       1       0       1       (Five)
              ---------------------------
Bitwise operation (0 | 0) (0 | 1) (1 | 0) (1 | 1)
              ---------------------------
          Result  0       1       1       1       (seven)
            Decimal Equivalent of (0 1 1 1) = 7
            Therefore 3 | 5 = 7
```

PROGRAM 3.12 Write a program to read two numbers from the user. Display the result using bitwise | operator on the numbers.

```
num1 = int(input('Enter First Number: '))
num2 = int(input('Enter Second Number: '))
print(num1,' | ',num2,' = ', num1 | num2)
```

Output

```
#Test  Case 1
Enter First Number: 3
Enter Second Number: 5
3  |  5  =  7

#Test  Case 2
Enter First Number: 6
Enter Second Number: 1
6  |  1  =  7
```

3.7.3 The Bitwise XOR (^) Operator

This operator performs bitwise exclusive or XOR operation on the numbers. It is represented as '^'. The '^' operator also operates on two operands and these two operands are compared bit-by-bit. Table 3.15. explains the '^' (XOR) operator.

Table 3.15 The Table for Bitwise XOR Operator

Input		Output
X	Y	X ^ Y
0	0	0
0	1	1
1	0	1
1	1	0

We can conclude from this table that the output is logic one when one of the input bits is logic one.

Examples of Bitwise XOR (^) Operator

```
>>> 3 ^  5
6     # The bitwise ^ operator on 3 and 5 returns 6
>>> 1 ^ 5
4     # The bitwise ^ operator on 1 and 5 returns 4
```

Working of the bitwise XOR ('^') operator is given as follows:

```
Working of expression 3 ^ 5 is as below.
Initially 3 and 5 are converted into their equivalent binary format
                0       0       1       1       (Three)
              ^
                0       1       0       1       (Five)
              ---------------------------
Bitwise operation (0 ^ 0) (0 ^ 1) (1 ^ 0) (1 ^ 1)
              ---------------------------
          Result  0       1       1       0       (Six)
            Decimal Equivalent of (0 1 1 0) = 6
            Therefore 3 ^ 1 = 6
```

PROGRAM 3.13 Write a program to read two numbers from the user. Operate bitwise ^ operator on them and display the result.

```
num1 = int(input('Enter First Number: '))
num2 = int(input('Enter Second Number: '))
print(num1,' ^ ',num2,' = ', num1 ^ num2)
```

Output

```
#Test Case 1
Enter First Number: 3
```

(Contd.)

```
Enter Second Number: 5
3 ^ 5 = 6
#Test Case 2
Enter First Number: 1
Enter Second Number: 2
1 ^ 2 = 3
```

3.7.4 The Right Shift (>>) Operator

The right shift operator is represented as >>. It also needs two operands. It is used to shift bits to the right by *n* position. Working of the right shift operator (>>) is explained as follows:

Example

```
>>>4 >> 2  # The input data 4 is to be shifted by 2 bits towards the right side
1
>>>8>>2
2
```

Explanation

Consider the expression 4 >> 2.

Initially, the number 4 is converted into its corresponding binary format, i.e. 0 1 0 0

0 0 0 0 0 0 1 0 0 Binary 4 ←

8 7 6 5 4 3 2 1 0 Bit Index ←

The input data 4 is to be shifted by 2 bits towards the right side.

The answer in binary bits would be

0 0 0 0 0 0 0 0 1 Binary 1 ←

8 7 6 5 4 3 2 1 0 Bit Index ←

Note: Shifting the input number by N bits towards the right means the number is divided by 2^S.
In short, it means $Y = N/2^S$.
Where,

N = The Number
S = The Number of Bit Positions to Shift

Consider the above example 4 >> 2. Let us solve this using the above formula, i.e. $y = N/2^S$

$= 4 / 2^2$
$= 4 / 4$
$= 1$

Therefore, 4 >> 2 returns 1 in Python interactive mode.

PROGRAM 3.14 | Write a program to shift input data by 2 bits towards the right.

```
N = int(input('Enter  Number: '))
S = int(input('Enter Number of Bits to be shift Right: '))
print(N,' >> ',S,' = ', N >> S)
```

Output

```
Enter  Number: 8
Enter Number of Bits to be shift Right: 2
8  >>  2  =  2
```

3.7.5 The Left Shift (<<) Operator

The left shift operator is represented as <<. It also needs two operands. It is used to shift bits to the left by N position. The working of the left shift operator is given as follows:

Example

```
>>> 4 << 2  # The input data 4 is to be shifted by 2 bits towards the left side
16

>>> 8 << 2  # The input data 8 is to be shifted by 2 bits towards the left side
32
```

Explanation

Consider the expression 4 << 2.

Initially, the number 4 is converted into its corresponding binary format, i.e. 0 1 0 0

0 0 0 0 0 0 1 0 0 Binary 4 ←

8 7 6 5 4 3 2 1 0 Bit Index ←

The input data 4 is to be shifted by 2 bits towards the left side.

The answer in binary bits would be

0 0 0 0 1 0 0 0 0 Binary 16 ←

8 7 6 5 4 3 2 1 0 Bit Index ←

Note: Shifting the input number by N bits towards the left side means the number is multiplied by 2^S. In short, it means $Y = N*2^S$.
Where,
N = The Number
S = The Number of Bit Positions to Shift
Consider the above example 4 << 2. Let us solve this using the above formula, i.e.

$y = N * 2^S$
$= 4 * 2^2$
$= 4 * 4$
$= 16$

Therefore, 4 << 2 returns 16 in Python interactive mode.

PROGRAM 3.15 | Write a program to shift input data by four bits towards the left.

```
N = int(input('Enter  Number: '))
S = int(input('Enter Number of Bits to be shift Left: '))
print(N,' << ',S,' = ', N << S)
```

Output

```
Enter  Number: 4
Enter Number of Bits to be shift Left: 2
4  <<  2  =  16
```

3.8 THE COMPOUND ASSIGNMENT OPERATOR

The operators +, *, //, /, % and ** are used with the assignment operator (=) to form the compound or augmented assignment operator.

Example

Consider the following example, where the value of a variable X is increased by 1.

X = X + 1

Python allows a programmer to combine the assignment and addition operator. Thus, the above statement X = X + 1 can also be written as

X + = 1

The += operator is called the addition operator. A list of all other compound assignment operators is given in Table 3.16.

Table 3.16 Compound assignment operators

Operator	*Example*	*Equivalent*	*Explanation*
+=	Z+=X	Z=Z+X	Add the value of Z to X
-=	Z-=X	Z=Z-X	Subtract X from Z
*=	Z*=X	Z=Z*X	Multiplies the value of X, Y and stores the result in Z
/=	Z/=X	Z=Z/X	Performs floating point division operation and stores the result in Z
//=	Z//=X	Z=Z//X	Performs normal integer floor division and stores the result in Z
**=	Z**=X	Z=Z**X	The value of variable X is raised to Z and the result is stored in variable Z
%=	Z%=X	Z=Z%X	The Z modulo X operation is performed.

PROGRAM 3.16 | Write a program using compound assignment operators to calculate the area of a circle.

```
radius = eval(input('Enter the Radius of Circle: ')) #Read Radius
print(' Radius = ',radius) #Display Radius
area = 3.14
radius **=2    #Radius = Radius ** 2
area*=radius   #Area=Area*Radius
print(' Radius of Circle is  = ',area)  #Print area
```

Output

```
Enter the Radius of Circle: 2
Radius =  2
Radius of Circle is  =  12.56
```

Thus, to perform various operations in the above program we have to make use of compound assignment operators such as **=, and *=.

MINI PROJECT Goods Service Tax (GST) Calculator

What is GST?

Goods and services tax is a comprehensive tax **levied on the manufacture, sale and consumption of goods and services** at a national level. This tax has **substituted all indirect taxes** levied on goods and services earlier by the central and state governments in India.

Problem Statement

We all buy various goods from a store. Along with the price of the goods we wish to buy, we also have to pay an additional tax, which is calculated as a specific percentage on the total price of the goods. This is called GST on the products.

Model of GST Using an Example

The GST has two components, viz. one which is levied by the central government (referred to as central GST or CGST), and one levied by the state government (referred to as state GST or SGST). The **rates** for central GST and state GST are given as follows:

Type of Tax	*Tax Rate*
CGST	@9%
SGST	@9%

Example

Invoice of a product

Particulars	*GST on Particulars*
Cost of Production	5,000
Add: CGST @ 9%	450
Add: SGST @ 9%	450
Total Cost of Product:	**₹5,900**

Formula to Calculate Total Cost

(CGST Tax Rate on product) + (SGST Tax Rate on product)

> Note: Make use of proper operators to solve the above problem.

Algorithm

- **STEP 1:** Read Cost of Production
- **STEP 2:** Input the CGST tax rate
- **STEP 3:** Input the SGST tax rate
- **STEP 4:** Calculate and print the total cost of the product.

Program

```
CP = float(input('Enter the Cost of Product:'))
CGST = float(input('Enter tax % imposed by Centre, i.e. CGST:'))
SGST = float(input('Enter tax % imposed by State,  i.e. SGST:'))
total = 0
Amount_CGST = ((CGST/100) * CP)
Amount_SGST = ((SGST/100) * CP)
total = CP + Amount_CGST + Amount_SGST
print('Total Cost of Product: Rs ',total)
```

(Contd.)

Output

```
Enter the Cost of Product: 5000
Enter tax % imposed by Centre, i.e. CGST: 9
Enter tax % imposed by State, i.e. SGST: 9
Total Cost of Product: Rs.  5900.0
```

In the above example, we have calculated final cost of the product based on the tax rate.

SUMMARY

- Python supports various operators such as Arithmetic, Boolean, Relational, Bitwise and compound Assignment Operator.
- Unary Operator perform operation on one operand only whereas Binary operator requires two operands.
- The Divison(\) operator applied on two operands returns a float value.
- Modulo (%) operator return s remainder when first number is divided by the second.
- Exponent (**) operator calculates power of number.
- Operator precedence determines the order in which python evaluates the operators in an expression.
- Associativity gives direction of execution, i.e. **left to right** or **right to left**.

KEY TERMS

- **Arithmetic Operators: Binary** and **Unary** Operators
- **Bitwise Operators:** and (&), or (|), xor (^), left shift (<<) and right shift (>>)
- **Augmented Assignment Operator**: Operators used with the assignment operator
- **Operator Precedence:** Determines the order in which the Python interpreter evaluates an expression
- **Associativity**: Determines which operation is to be performed first.

REVIEW QUESTIONS

A. Multiple Choice Questions

1. What will be the output of the following expression if it is executed in Python interactive mode?

   16 % 3

   a. 5 b. 1

   c. 0 d. -1

2. What will be the output of the following program?

```
X=5
Y=5
print(X/Y)
```

Decision Statements

4

CHAPTER OUTLINE

LEARNING OUTCOMES

After completing this chapter, students will be able to:

- Describe Boolean expressions and **bool** data type
- Perform operations on numbers and strings using **Boolean** and **Relational** operators (> <,>= <= and !=)
- Write a simple decision making statement and its implementation with **if** statement, two-way decision making statements and their implementation with **if else** statement, nested statements and their implementation with if statements and multi-way decision making statements and their implementation with **if-elif-else** statements
- Explain and use **conditional expressions** to write programs
- Write non-sequential programs using Boolean expressions

4.1 INTRODUCTION

So far, we have seen programs that contain a sequence of instructions. These programs are executed by the compiler line by line, in the way the program line appears. The control flow in

such programs is sequential. Control flow refers to the order in which program statements are executed, i.e. when the execution of one statement is complete, the computer control passes to the next statement in the code. This process is similar to reading the text, figures and tables on a page of a book.

In monolithic programs, instructions are executed sequentially one by one in the order in which they come into sight in the program. Of course, this is a fundamental programming concept for beginners to develop simple programs. It is not advisable to have a sequential program writing style for solving every problem. Quite often, it is advantageous in a program to alter the sequence of the flow of statements depending upon the circumstances. In real time applications, there are a number of situations where a programmer has to change the order of execution of statements based on certain conditions. Therefore, when a programmer desires the control flow to be non-sequential then he/she may use control structures or decision statements. Thus, decision making statements help a programmer in transferring the control from one statement to another in the program. In short, a **programmer decides which statement is to be executed based on a condition.** Decision making statements use conditions which are similar to **Boolean expressions**.

After reading this chapter, a programmer is expected to take up real life problems/applications and implement with Python programming containing conditional statements. Programmer may think the programming pattern for preparation of mark sheet, grade sheet, preparation of electricity bill for residential and commercial consumers, Railway tariff based on distances, simple calculations of interest on deposits for banking problems, etc. Of course, unlimited problems are existing in the nature for which a programmer is expected to give programming solution.

4.2 BOOLEAN TYPE

Python has a type called '**bool**'. The bool has only two values, viz. **true** and **false**. The term 'Boolean' comes from the name of the British mathematician, George Boole. In the 1840's, Boole showed that the classical rules of logic could be expressed in purely mathematical form using only two values, viz. true and false. The simplest Boolean expression in Python is True and False. In Python interactive shell, a programmer can check if the type of two values, viz. true and false belong to the type '**bool**' in the following manner:

```
>>> True
True
>>> False
False
>>> type(True)
<class 'bool'>      #The Value True belongs to the class type bool
>>> type(False)
<class 'bool'>      #The Value False belongs to the class type bool
```

Note: There are only two Boolean values, **True** and **False**. Capitalization of the first letter is important for these values and so **true** and **false** are not considered Boolean values in Python. As illustrated, the Python interpreter will show an error if a programmer checks the type of 'true' or 'false'.

(Contd.)

```
>>> type(true)
Traceback (most recent call last):
File "<pyshell#10>", line 1, in <module>
type(true)
NameError: name 'true' is not defined
```

4.3 BOOLEAN OPERATORS

The **and**, **or** and **not** are the only three basic Boolean operators. Boolean operators are also called **logical operators**. The not operator has the highest precedence, followed by **and** and then **or**.

4.3.1 The not Operator

The **not** operator is a unary operator. It is applied to just one value. The **not** operator takes a single operand and negates or inverts its Boolean value. If we apply the not operator on an expression having false value then it returns it as true. Similarly, if we apply the not operator on an expression having true value then it returns it as false.

Example

Use of the not operator on a simple Boolean expression in Python, i.e. true and false.

```
>>> True
True
>>> not True
False
>>> False
False
>>> not False
True
```

4.3.2 The and Operator

The **and** is a binary operator. The and operator takes two operands and performs left to right evaluation to determine whether both the operands are true. Thus, and of Boolean operand is **true** if and only if both operands are **true**. Table 4.1 explains the add operator.

Table 4.1 The **and** operator

X	*Y*	*X and Y*
True	True	True
True	False	False
False	True	False
False	False	False

Example

Evaluation of the **and** operator in Python interactive mode.

```
>>> True and True
True
>>> True and False
False
>>> False and True
False
>>> False and False
False
```

4.3.3 The or Operator

The **or** of two Boolean operands is true if at least one of the operands is true. Table 4.2 explains the or operator.

Table 4.2 The **or** operator

X	*Y*	*X or Y*
True	True	True
True	False	True
False	True	True
False	False	False

Example

Evaluation of the or operator in Python interactive mode.

```
>>> True or True
True
>>> True or False
True
>>> False or True
True
>>> False or False
False
```

4.4 USING NUMBERS WITH BOOLEAN OPERATORS

A programmer can use numbers with Boolean operators in Python. One such example is given as follows:

Example

```
>>> not 1
False
```

```
>>> 5
5
>>> not 5
False
>>> not 0
True
>>> not 0.0
True
```

Explanation Here, Python uses the Boolean operator **not** on the numbers and treats all numbers as **True**. Therefore, by writing **not 1**, Python substitutes 1 as **True** and evaluates **not True**, which returns **False**. Similarly, **not** is used before 5 and Python substitute **True** in place of 5 and it again evaluates the expression **not True**, which returns **False**. But in case of the numbers 0 and 0.0, Python treats them as **False**. Therefore, while evaluating **not 0**, it substitutes **False** in place of 0 and again evaluates the expression **not False**, which returns **True**.

4.5 USING STRING WITH BOOLEAN OPERATORS

Like numbers, a programmer can use strings with Boolean operators in Python. One such example is given as follows:

Example

```
>>> not 'hello'
False
>>> not ''
True
```

Explanation Here, Python uses the Boolean operator **not** on string. The expression **not hello** returns **True** since Python treats all strings as **True**. Therefore, it substitutes **True** in place of 'hello' and again reevaluates the expression **not True**, which returns **False**. However, if it is an empty string, Python will treat it as **False**. Therefore, it substitutes **False** in place of an empty string "and reevaluates the expression **not False**, which in turn returns **True**.

4.6 BOOLEAN EXPRESSIONS AND RELATIONAL OPERATORS

A Boolean expression is an expression that is either true or false. The following example compares the value of two operands using the == operator and produces the result true if the values of both the operands are equal.

Example

The == operator compares two values and produces a Boolean value.

```
>>> 2==2
True
```



```
>>> a=2
>>> b=2
>>> a==b
True
```

> **Note:** The comparison operator == contains two equal signs. Whereas the assignment operator = contains only one equal sign.

From the above example, it is clear how we can compare two values or two operands. Thus, == is one of the Python **relational operators**. Other relational operators supported in Python are given in Table 4.3.

Table 4.3 Relational operators

Operator	*Meaning*	*Example*	*Python Return Value*
>	Greater than	4>1	True
<	Less than	4<9	True
>=	Greater than or equal to	4>=4	True
<=	Less than or equal to	4<=3	True
!=	Not equal to	5!=4	True

PROGRAM 4.1 Write a program to prompt a user to enter the values of the three different variables and display the output of the following expressions.

a. p>q>r
b. p<q<r
c. p<q and q<z
d. p<q or q<z

```
p,q,r=eval(input('Enter Three Numbers:'))
print(' p =',p,' q = ',q,' r = ',r)
print('(p > q > r) is ', p > q >r)
print('(p < q < r) is ', p < q <r)
print(' (p < q) and (q < r ) is ', (p < q) and (q < r ))
print(' (p < q) or (q < r) is ', (p < q) or (q < r ))
```

Output

```
Enter Three Numbers:1,2,3
 p = 1  q =  2  r =  3
(p > q > r) is  False
(p < q < r) is  True
(p < q) and (q < r ) is  True
(p < q) or (q < r) is  True
```

Note: An expression always returns a value and a statement does not return any value. A statement may include one or more than one expression.

4.7 DECISION MAKING STATEMENTS

Python supports various decision-making statements. These are:

1. if statements
2. if-else statements
3. Nested if statements
4. Multi-way if-elif-else statements

4.7.1 The if Statements

The **if** statement executes a statement if a condition is true. The syntax for if statement is shown in Figure 4.1.

```
if condition:
    statement(s)
```

OR

```
if condition:
    Block
```

Figure 4.1 Syntax for **if** statement

Details of the if Statement

The keyword **if** begins the **if** statement. The condition is a Boolean expression which determines whether or not the body of **if** block will be executed. A colon (:) must always be followed by the condition. The block may contain one or more statements. The statement or statements are executed if and only if the condition within the **if** statement is true. The flow chart for **if** statement is given in Figure 4.2.

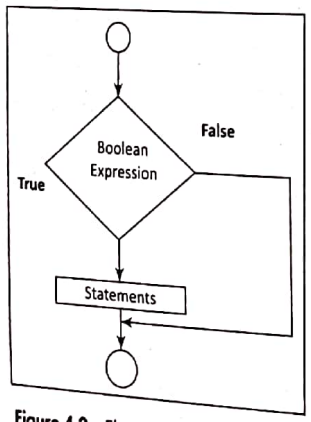


Figure 4.2 Flowchart for **if** statement

Points to Remember

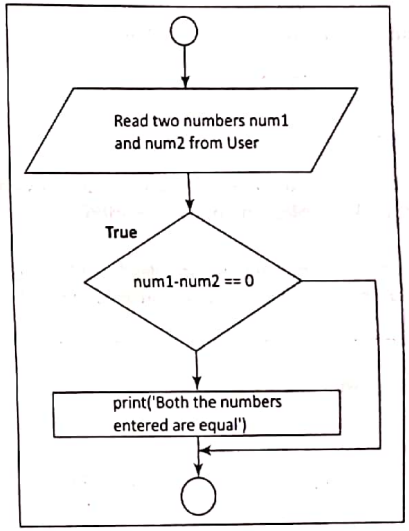
(a) The statement(s) must be indented at least one space right of the **if** statement.

(b) In case there is more than one statement after the **if** condition, then each statement must be indented using the same number of spaces to avoid indentation errors.

The statement(s) within the if block are executed if the Boolean expression evaluates to true.

PROGRAM 4.2 Write a program that prompts a user to enter two integer values. Print the message **'Equals'** if both the entered values are equal.

Flow Chart




```
num1=eval(input("Enter First Number:  "))
num2=eval(input("Enter Second Number:  "))
if  num1-num2==0:
    print("Both the numbers entered are Equal")
```

Output

```
Enter First Number:  12
Enter Second Number:  12
Both the numbers entered are Equal
```

Explanation In the above program, the two numbers are provided by a user. The statement within the if block is executed if and only if the Boolean expression num1 – num2 evaluates to True.

Precautions Sometimes a program may contain only one statement within the if block. In such case a programmer can write the block of code in two different ways.

(a) Consider the code given as:

```
Number=eval(input("Enter the Number: "))
if  Number>0:
    Number =  Number * Number
```

This code can also be written as:

```
Number=eval(input("Enter the Number:  "))
if  Number>0:Number =  Number * Number
```

(b) The above code **cannot** be written as:

```
Number=eval(input("Enter the Number:  "))
if  Number>0:
Number =  Number * Number
```

The above code does not run and displays an error called **indentation error**. Thus, Python determines which statement makes a block using indentation.

PROGRAM 4.3 Write a program which prompts a user to enter the radius of a circle. If the radius is greater than zero then calculate and print the area and circumference of the circle.

```
from math import pi
Radius=eval(input("Enter Radius of Circle:  "))
if  Radius>0:
    Area=Radius*Radius*pi
    print(" Area of Circle is  = ",format(Area,".2f"))
    Circumference=2*pi*Radius
    print("Circumference  of Circle is = ",format(Circumference,".2f"))
```

Output

```
Enter Radius of Circle:  5
 Area of Circle is  =  78.54
Circumference  of Circle is =  31.42
```

4.7.2 The if-else Statement

The execution of the **if** statement has been explained in the previous programs. We know, the **if** statement executes when the condition following **if** is true and it does nothing when the condition is false. The **if-else** statement takes care of a true as well a false condition. The syntax for **if-else** statement is given in Figure 4.3.


```
if condition:
    statement(s)
else:
    statement(s)
```

OR

```
if condition:
    if_block
else
    else_block
```

Figure 4.3 Syntax for **if-else** statement

Details of if-else Statement

The **if-else** statement takes care of both true and false conditions. It has two blocks. One block is for **if** and it may contain one or more than one statements. The block is executed when the condition is true. The other block is for **else**. The **else** block may also have one or more than one statements. It is executed when the condition is false. A colon (:) must always be followed by the condition. The keyword **else** should also be followed by a colon (:). The flow chart for **if-else** statement is given in Figure 4.4.

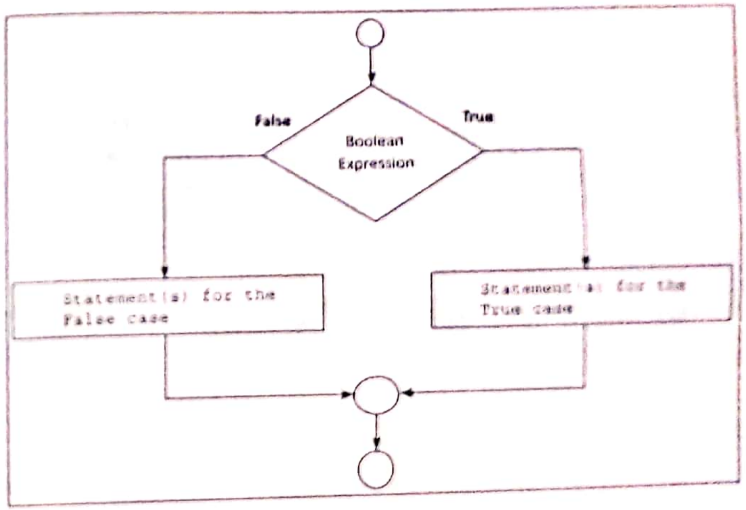


Figure 4.4 Flow chart for **if-else** statement

PROGRAM 4.4 Write a program to prompt a user to enter two numbers. Find the greater number.

```
num1=int(input("Enter the First Number:"))
num2=int(input("Enter the Second Number:"))
if num1>num2:
    print(num1,"is greater than ",num2)
else:
    print(num2,"is greater than ",num1)
```

(Contd.)

Output

```
Enter the First Number:100
Enter the Second Number:43
100 is greater than 43
```

Explanation The above program prompts a user to read any two numbers. The two numbers entered are stored in variables num1 and num2, respectively. If the value of num1 is greater than num2 is checked using the **if** condition. If the value of num1 is greater then the message 'num1 is greater than num2' is displayed. Otherwise, the message 'num2 is greater than num1' is displayed.

PROGRAM 4.5 | Write a program to calculate the salary of a medical representative considering the sales bonus and incentives offered to him are based on the total sales. If the sales exceed or equal to ₹1,00,000 follow the particulars of Column 1, else follow Column 2.

Column 1	Column 2
Basic = ₹4000	Basic = ₹4000
HRA = 20% of Basic	HRA = 10% of Basic
DA = 110 % of Basic	DA = 110 % of Basic
Conveyance = ₹500	Conveyance = ₹500
Incentive = 10% of Sales	Incentive = 4% of Sales
Bonus = ₹1000	Bonus = ₹500

```
Sales=float(input('Enter Total Sales of the Month:'))
if Sales >= 100000:
    basic = 4000
    hra = 20 * basic/100
    da  = 110 * basic/100
    incentive = Sales * 10/100
    bonus = 1000
    conveyance = 500
else:
    basic = 4000
    hra = 10 * basic/100
    da  = 110 * basic/100
    incentive = Sales * 4/100
    bonus = 500
    conveyance = 500

salary= basic+hra+da+incentive+bonus+conveyance
print('Salary Receipt of Employee ')
```

(Contd.)


```
print(' Total Sales = ',Sales)
print(' Basic = ',basic)
print(' HRA = ',hra)
print(' DA = ',da)
print(' Incentive = ',incentive)
print(' Bonus = ',bonus)
print(' Conveyance = ',conveyance)
print(' Gross Salary = ',salary)
```

Output

```
Enter Total Sales of the Month:100000
Salary Receipt of Employee
 Total Sales =  100000.0
 Basic =  4000
 HRA =  800.0
 DA =  4400.0
 Incentive =  10000.0
 Bonus =  1000
 Conveyance =  500
 Gross Salary =  20700.0
```

Explanation The program calculates the salary of a medical representative according to the total sale of products. The basic salary is the same but other allowances and incentives change according to the total sales. If the total sale is more than ₹1,00,000 the rate of allowances and incentive is calculated as per Column 1, **else** as per Column 2. The **if** condition checks the given figure of total sale. If the total sale is more than ₹1,00,000 the first block following the **if** statement is executed, otherwise the **else** block is executed.

Points to Remember

(a) Indentation is very important in Python. The **else** keyword must properly line up with the if statement.

(b) If a programmer does not line up **if** and **else** in exactly the same columns then Python will not know that **if** and **else** will go together. Consequentially, it will show an indentation error.

(c) Both statements within the if block and else block must be indented and must be indented the same amount.

PROGRAM 4.6 | Write a program to test whether a number is divisible by 5 and 10 or by 5 or 10.

```
num=int(input('Enter the number:'))
print('Entered Number is: ',num)
```

(Contd.)

```
if(num % 5 == 0 and num % 10==0):
    print(num,' is divisible by both 5 and 10')
if(num % 5 == 0 or num % 10 == 0):
    print(num,'is divisible by 5 or 10')
else:
    print(num,' is not divisible either by 5 or 10')
```

Output

```
#Test Case 1:
 Enter the number:45
Entered Number is:  45
45 is divisible by 5 or 10
#Test Case 2:
Enter the number:100
Entered Number is:  100
100  is divisible by both 5 and 10
100 is divisible by 5 or 10
```

Explanation In the above program, the number is read from the user. The Boolean expression **num % 5 == 0 and num % 10==0** checks whether the number is divisible by both 5 and 10. Again the Boolean expression **num % 5 == 0 or num % 10 == 0** is used to check if the number entered is divisible either by 5 or by 10.

> **Note: Conditional or Short Circuit AND Operator:** If one of the operands of an AND operator is false, the expression is false. Consider two operands OP1 and OP2. When evaluating OP1 and OP2, Python first evaluates OP1 and if OP1 is True then Python evaluates the second operand OP2. Python improves the performance of the AND operator, i.e. if the operand OP1 is False, it does not evaluate the value of the second operand OP2. The AND operator is also referred to as conditional or short circuit AND operator.
>
> **Conditional or Short Circuit OR Operator:** We have seen in Table 4.2 that even if one of the operands of an OR operator is True, the expression is True. Python improves the performance of the OR operator. Consider two operands OP1 and OP2 and the expression OP1 or OP2. While evaluating the expression OP1 or OP2, Python first evaluates OP1. If OP1 is False, it evaluates OP2. If OP1 is True, it does not evaluate OP2. The OR operator is also referred to as conditional or short circuit OR operator.

4.7.3 Nested if Statements

When a programmer writes one if statement inside another if statement then it is called a **nested if** statement. A general syntax for **nested if** statements is given as follows:

```
if Boolean-expression1:
    if Boolean-expression2:
        statement1
    else:
        statement2
else:
statement3
```


In the above syntax, if the Boolean-expression1 and Boolean-expression2 are correct then statement1 will execute. If the Boolean-expression1 is correct and Boolean-expression2 is incorrect then statement2 will execute. And if both Boolean-expression1 and Boolean-expression2 are incorrect then statement3 will execute.

A program to demonstrate the use of **nested if** statements is given as follows:

PROGRAM 4.7 | Write a program to read three numbers from a user and check if the first number is greater or less than the other two numbers.

```
num1=int(input("Enter the number:"))
num2=int(input("Enter the number:"))
num3=int(input("Enter the number:"))
if num1>num2:
    if num2>num3:
        print(num1,"is greater than ",num2,"and ",num3)
else:
    print(num1," is less than ",num2,"and",num3)
print("End of Nested if")
```

Output

```
Enter the number:12
Enter the number:34
Enter the number:56
12 is less than 34 and 56
End of Nested if
```

Explanation In the above program, three numbers—num1, num2 and num3—are provided from the user through a keyboard. Initially, the **if** condition with Boolean expression **num1>num2** is checked if it is true and the then other nested **if** condition with Boolean expression **num2>num3** is checked. If both the **if** conditions are true then the statements following the second if statement are executed.

4.7.4 Multi-way if-elif-else Statements

The syntax for **if-elif-else** statements is given as follows:

```
If Boolean-expression1:
        statement1
    elif Boolean-expression2 :
```

```
        statement2
    elif Boolean-expression3 :
        statement3
.   . . . . . .
.   . . . . . .
elif Boolean-expression n :
    statement N
else :
    Statement(s)
```

In this kind of statements, the number of conditions, i.e. Boolean expressions are checked from top to bottom. When a true condition is found, the statement associated with it is executed and the rest of the conditional statements are skipped. If none of the conditions are found true then the last else statement is executed. If all other conditions are false and if the final else statement is not present then no action takes place.

PROGRAM 4.8 Write a program to prompt a user to read the marks of five different subjects. Calculate the total marks and percentage of the marks and display the message according to the range of percentage given in table.

Percentage	Message
per > = 90	Distinction
per > = 80 && per < 90	First Class
per > = 70 && per < 80	Second Class
per > = 60 && per < 70	First Class
per <60	Fail

```
Subject1=float(input("Enter the Marks of Data-Structure:"))
Subject2=float(input("Enter the Marks of Python:"))
Subject3=float(input("Enter the Marks of Java:"))
Subject4=float(input("Enter the Marks of C Programming:"))
Subject5=float(input("Enter the Marks of HTML:"))
sum=Subject1+Subject1+Subject3+Subject4+Subject5
per=sum/5
print("Total Marks Obtained", sum, "Out of 500")
print("Percentage = ",per)

if per>=90:
    print("Distinction")
else:
    if per>=80:
        print(" First Class")
```

(Contd.)


```
    else:
        if  per>=70:
            print("Second Class")
    else:
        if per>=60:
            print("Pass")
        else:
            print("Fail")
```

Output

```
Enter the Marks of Data-Structure: 60
Enter the Marks of Python: 70
Enter the Marks of Java: 80
Enter the Marks of C Programming: 90
Enter the Marks of HTML: 95
Total Marks Obtained 385.0 out of 500
Percentage = 77.0
Second Class
```

Explanation In the above program, the marks of five subjects are entered through a keyboard. Their sum and average is calculated. The percentage obtained is stored in the variable **'per'**. The obtained percentages are checked with different conditions using if-else blocks and the statements are executed according to the conditions.

Note: The above program consists of **if-else-if** statements. It can also be written in **if-elif-else** form as shown in Figure 4.5(b).

```
if per>=90:
    print("Distinction")
else:
    if per>=80:
        print(" First Class")
    else:
        if  per>=70:
            print("Second Class")
        else:
            if per>=60:
                print("Pass")
            else:
                print("Fail")
```

(a)

Equivalent

```
if per>=90:
    print("Distinction")
elif per>=80:
    print(" First Class")
elif per>=70:
    print("Second Class")
elif per>=60:
    print("Pass")
else:
    print("Fail")
```

(b)

Figure 4.5 (a) if-else-if-else (b) if-elif-else

The flowchart for multi-way **if-else-if** statements for the above program is given in Figure 4.6

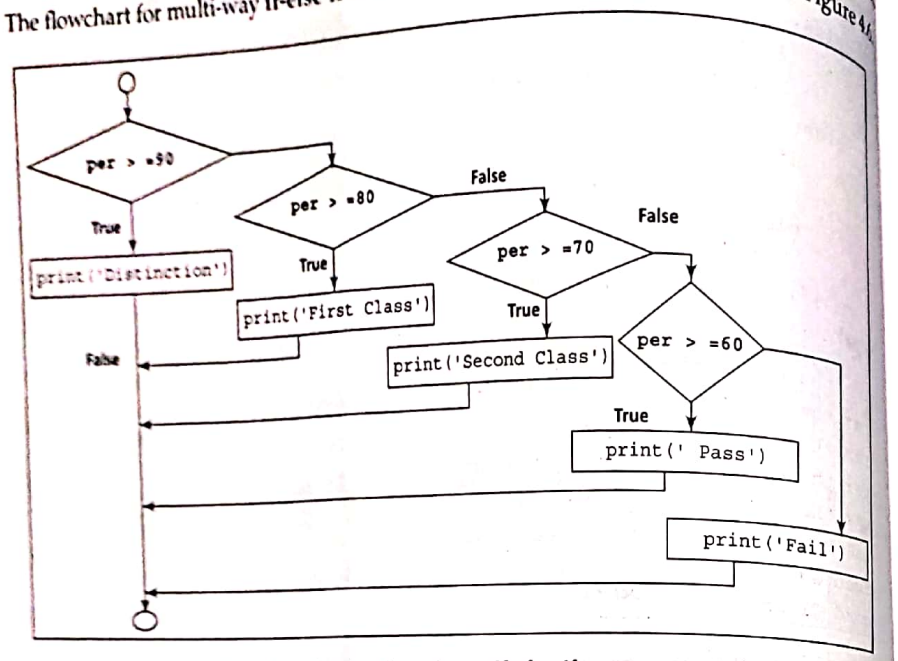


Figure 4.6 Flowchart for multi-way **if-else-if** statements

PROGRAM 4.9 Write a program to prompt a user to enter a day of the week. If the entered day of the week is between 1 and 7 then display the respective name of the day.

```
Day=int(input("Enter the day of week:"))
if day==1:
    print(" Its Monday")
elif day==2:
    print("Its Tuesday")
elif day==3:
    print("Its Wednesday")
elif day==4:
    print("Its Thursday")
elif day==5:
    print("Its Friday")
elif day==6:
    print("Its Saturday")
elif day==7:
    print(" Its Sunday")
else:
    print("Sorry!!! Week contains only 7 days")
```

(Contd.)

Output

```
Enter the day of week: 7
Its Sunday
```

PROGRAM 4.10 Write a program that prompts a user to enter two different numbers. Perform basic arithmetic operations based on the choices.

```
num1=float(input("Enter the first number:"))
num2=float(input("Enter the Second number:"))

print("1) Addition ")
print("2) Subtraction ")
print("3) Multiplication ")
print("4) Division ")

choice = int(input("Please Enter the Choice:"))
if choice==1:
    print(" Addition of ",num1,"and",num2,"is:",num1+num2)
elif choice==2:
    print(" Subtraction of ",num1,"and",num2,"is:",num1-num2)
elif choice==3:
    print(" Multiplication of ",num1,"and",num2,"is:",num1*num2)
elif choice==4:
    print(" Division of ",num1,"and",num2,"is:",num1/num2)
else:
    print("Sorry!!! Invalid Choice")
```

Output

```
Enter the first number:15
Enter the Second number:10
1) Addition
2) Subtraction
3) Multiplication
4) Division
Please Enter the Choice:3
 Multiplication of 15.0 and 10.0 is: 150.0
```

4.8 CONDITIONAL EXPRESSIONS

Consider the following piece of code.

```
if x%2==0:
    x = x*x
```

```
else:
    x = x*x*x
```

In the above code, initially, x is divided by 2. If x is divisible by 2 then the square of the number is assigned to variable x, else the cube of the number is assigned. To improve the performance of simple if-else statements, Python provides a conditional expression. Using this conditional expression, the code above can be rewritten as:

```
x=x*x if x % 2 == 0 else x*x*x
```

Therefore, the general form of conditional expression is:

$Expression_1$ `if condition else` $Expression_2$

$Expression_1$ is the value of the conditional expression if the condition is true.
Condition is a normal Boolean expression that generally appears in front of an if statement.
$Expression_2$ is the value of the conditional expression if the condition is false.
Consider the program without conditional expression given as follows:

PROGRAM 4.11 | Write a program to find the smaller number among the two numbers.

```
num1=int(input('Enter two Numbers:'))
num2=int(input('Enter two Numbers:'))
if num1 < num2:
    min=num1
    print('min = ',min)
else:
    min=num2
    print('min = ',min)
```

Output

```
Enter two Numbers: 20
Enter two Numbers: 30
min = 20
```

The same program can be written using conditional expression as follows:

```
num1=int(input('Enter two Numbers:'))
num2=int(input('Enter two Numbers:'))
min = print('min = ',num1) if num1 < num2 else print('min = ',num2)
```

Output

```
Enter two Numbers: 45
Enter two Numbers: 60
min = 45
```


Note: Many programming languages, such as Java, C++ have a '?:', i.e. ternary operator. This is a conditional operator. The syntax for the '?:' ternary operator is:

```
Boolean expression? if_true_return_value1: if_false_return_value2
```

The ternary operator works like **if-else**. If the Boolean expression is true, it returns value1 and if the Boolean expression is false, it returns the second value.
Python does not have a ternary operator. It uses a **conditional expression**.

MINI PROJECT Finding the Number of Days in a Month

This mini project will make use of programming features such as **if statement** and **elif** statements. It will help a programmer to know the number of days in a month.

Hint: If entered the month is 2 then read the corresponding year. To know the number of days in month 2 check if the entered year is a leap year. If leap then **num_days = 29** or not leap then **num_days = 28** for month 2, respectively.

Leap year: A leap year is divisible by **4** but not by **100** or divisible by **400**.

Algorithm

- **STEP 1:** Prompt the month from the user.
- **STEP 2:** Check if the entered month is 2, i.e. February. If so then go to **Step 3**, else go to **Step 4.**
- **STEP 3:** If the entered month is **2** then check if the year is a **leap** year. If it is a leap year then store **num_days** = 29, else **num_days** = 28.
- **STEP 4:** If the entered month is one of the following from the list (1, 3, 5, 7, 8, 12) then store **num_days** = 31. Or if the entered month is from the list (4, 6, 9, 11) then store **num_days** = 29. If the entered month is different from the range (1 to 12) then display message "Invalid Month".
- **STEP 5:** If the input is valid then display the message as "there are **N** number of days in the month **M**".

Program

```
#Number of Days in a Month

print('Program will print number of days in a given month')
#init

flag  = 1  # Assumes user enters valid input

#Get month from the user
month =  (int(input('Enter the month(1-12):')))
```

(Contd.)

```
# Check if entered month = 2 i.e. February
if month == 2:
        year = int(input('Enter year:'))

        if (year % 4  == 0) and (not(year % 100 == 0)) or (year % 400 == 0):
                num_days = 29
        else:
                num_days = 28
# if entered month is one from (jan, march, may, july, august, october, or
december)
elif month in (1,3,5,7,8,10,12):
        num_days = 31
# if entered month is one from (April, June, September November,)
elif month in (4, 6, 9, 11):
        num_days = 30
else:
        print('Please Enter Valid Month')
        flag = 0

#Finally print num_days
if flag == 1:
        print('There are ',num_days, 'days in', month,' month')
```

Output (Case 1)

```
Program will print number of days in a given month
Enter the month(1-12):2
Enter year: 2020
There are 29  days in 2 month
```

Output (Case 2)

```
Program will print number of days in a given month
Enter the month(1-12):4
There are 30  days in 4 month
```

Thus, the above case study helps the user to know the number of days for the entered year.

SUMMARY

- *A Boolean expression contains two values, viz. True and False.*
- *True and False are of type 'bool'.*
- *The and, or and not are the three basic Boolean operators.*

- *The not operator has highest precedence, followed by and and then or.*
- *A programmer can use strings with Boolean operators.*
- *The == operator compares two values and produces a Boolean value.*
- *Python supports various relational Operators such as, >, <, >=, <= and !=.*
- *Applying relational operators on numbers and characters yields a Boolean value.*
- *Python Supports various decision statements, such as if, if-else and multi-way if-elif-else statements.*
- *Python does not have a ternary operator. It uses a a conditional expression instead.*

KEY TERMS

- **Boolean Expressions:** An expression whose value is either **True** or **False.**
- **Logical Operators:** Comprise the **and, or** and **not** operators.
- **Relational Operators:** Comparison of two values with relational operators, such as <, <=, >, >=, != and == operators. One of the operators among them is used while comparing two operands.
- **Conditional Expression:** Evaluates expression based on condition.
- **Conditional or Short Circuit AND Operator:** Improves performance. Python avoids executing the second operand in case the first operand is false.
- **Conditional or Short circuit OR Operator:** Improves performance. Python avoids executing the second operand in case the first operand is true.

REVIEW QUESTIONS

A. Multiple Choice Questions

1. What will be the output of following program after the execution of the following code?

```
x = 0
y = 0
if x > 0:
    y = y + 1
else:
    if x < 0 :
       y = y + 2
    else:
       y = y + 5
print(' Y =',y)
```

a. 1 b. 0
c. 2 d. 5

2. What will be stored in num after the execution of the following code?

```
i=10
j=20
k=30
```

Loop Control Statements

5

CHAPTER OUTLINE

LEARNING OUTCOMES

After completing this chapter, students will be able to:

- Write programs using **for** and **while** loop to repeat a sequence of instructions
- Write a program and perform a task until a condition is satisfied
- Use **loops** to traverse the sequence of characters in string or traverse the sequence of integers
- Apply the syntax and working of **`range()`** function
- Control the execution of programs using **break** or **continue** statement

5.1 INTRODUCTION

In our day-to-day life, we perform certain tasks repeatedly. It can be tedious to perform such tasks using pen and paper. For instance, teaching multiplication tables to multiple classes can become easier if the teacher uses a simple computer program with loop instructions instead of pen and paper.

Let us try to understand the concept of control statements in this context. Suppose a programmer wants to display the message, "I Love Python" 50 times. It would be tedious for him/her to write the statement 50 times on a computer screen or even on paper. This task can become very easy, quick and accurate if the programmer completes it using loop instructions in a computer programming language. Almost all computer programming languages facilitate the use of control loop statements to repeatedly execute a block of code until a condition is satisfied.

Consider the example to print the statement, "I Love Python" 50 times. Assume that the programmer doesn't know the concept of control statements and writes the code in the following manner.

Example

```
print("I Love Python")
print("I Love Python")
print("I Love Python")
.
.
.
print("I Love Python")
```

```
print "I Love Python" for 50 times
```

In the above example, the print statement is written for displaying the message 50 times. This can be done more easily using loop in Python. Loops are used to repeat the same code multiple times. Python provides two types of loop statements, viz. **while** and **for** loops. The **while** loop is a **condition controlled** loop. It is controlled by true or false conditions. The **for** loop is a **count controlled** loop which repeats for a specific number of times.

After understanding the concept of loop, a programmer can take up any challenging application in which statements/actions are to be repeated several times.

5.2 THE while LOOP

The **while** loop is a loop control statement in Python and frequently used in programming for repeated execution of statement(s) in a loop. It executes a sequence of statements repeatedly as long as a condition remains true. The syntax for while loop is given as follows:

```
while test-condition:
    #Loop Body
    statement(s)
```

5.2.1 Details of while Loop

The reserved keyword **while** begins with the **while** statement. The test condition is a Boolean expression. The colon (:) must follow the test condition, i.e. the **while** statement be terminated with a colon (:). The statement(s) within the while loop will be executed till the condition is true, i.e. the condition is evaluated and if the condition is true then the body of the loop is executed. When the condition is false, the execution will be completed out of the loop or in other words, the control goes out of the loop. The flowchart in Fig. 5.2 shows the execution of the while loop.

5.2.2 Flowchart for while Loop

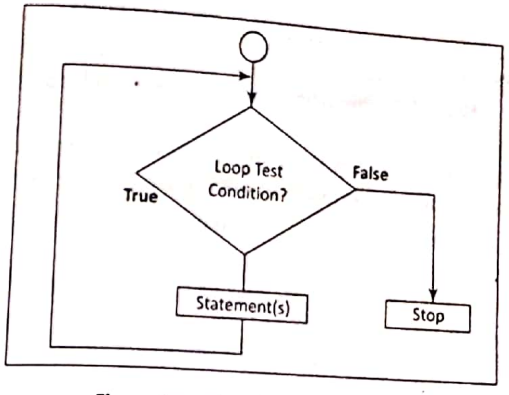


Figure 5.1 Flowchart of while loop

PROGRAM 5.1 | Write a program to print the numbers from one to five using the while loop.

```
count=0  #initialize the counter
while count<=5:                          # Test condition
    print("Count = ",count)              # print the value of count
    count=count+1                        # Increment the value of count by 1
```

Output

```
Count =  0
Count =  1
Count =  2
Count =  3
Count =  4
Count =  5
```

Explanation In the above program, initially the value of a variable **count** is initialised to 0. The loop checks whether the value of the count is less than 5 (**count<=5**). If the condition is **true**, it executes the part of the loop that contains the statements to be repeated in order to display the value of **count** and it increments the value of **count** by 1. It repeatedly executes the statements within the loop until **count<=5**. The loop terminates when the value of **count** reaches 6.

Note: Precaution is to be taken while writing statements within the while loop.

Consider the program as shown in Figure 5.2.

```
count=0
while count<=5:
    print("Count = ",count)
count=count+1
```

(a) Good Code

```
count=0
while count<=5:
    print("Count = ",count)
    count=count+1
```

(b) Bad Code

Figure 5.2 Precautions regarding the while loop

In Figure 5.2 (a) the value of count is initially set to 0. Then it increments to 2, 3, 4 and 5. When the value of count becomes 6, the condition **count<=5** is false and the loop exits.

Consider the Figure 5.2 (b) where the loop is mistakenly written as:

```
count=0
while count<=5:
    print("Count = ",count)
count=count+1
```

The above code is called **bad code** because the entire loop body must be indented inside the loop. Since the statement **count=count+1** is not in the loop body, the loop executes for infinite number of times. And because the value of count is always 0, the condition **count <=5** is always true.

> Note: All statements within the while block must be indented with the same number of spaces.

PROGRAM 5.2 | Write a program to add 10 consecutive numbers starting from 1 using the while loop.

```
count=0  #initialize the counter
sum=0    #initialize  sum to zero
while count<=10:                    #test condition if true
    sum= sum +count                 #add sum + count
    count=count+1                   #increase the value of count by
1
print("Sum of First 10 Numbers = ",sum)      #print sum
```

Output

```
Sum of First 10 Numbers =  55
```

PROGRAM 5.3 | Write a program to find the sum of the digits of a given number.

For example, if a user enters 123. The program should return (3+2+1), i.e. 6 as the sum of all the digits in a number.

```
num=int(input("Please Enter the number:"))#Read Number from User
x=num                                    #Assign value of num to x
```

(Contd.)


```
sum=0
rem=0
while num>0:
    rem=num % 10
    num=num // 10
    sum=sum + rem
print("Sum of the digits of an entered number ",x," is = ",sum)
```

Output

```
please Enter the number: 12345
Sum of the digits of an entered number 12345 is = 15
```

Explanation The integer number is read from the user through the keyboard and it is stored in variable num. Initially, the value of sum and rem are initialised to 0. Unless and until the value of num>0 the statements within the loop continue to be executed. The modulus operator, i.e. **num%10** and the division operator, i.e. **num//10** are used frequently to obtain the sum of the numbers entered.

5.2.3 Some More Programs on while Loop

PROGRAM 5.4 | Write a program to display the reverse of the number entered.

For example, if a user enters 12345. The program should return (54321), i.e. the reverse of the number entered.

```
num =int(input("Please Enter the number: "))
x=num
rev=0
while num>0:
    rem=num % 10
    num=num // 10
    rev=rev*10+rem
print("Reverse of a entered number ",x," is = ",rev)
```

Output

```
Please Enter the number: 8759
Reverse of a entered number 8759 is = 9578
```

PROGRAM 5.5 | Write a program to print the sum of the numbers from 1 to 20 (1 and 20 are included) that are divisible by 5 using the while loop.

```
count=1
sum=0
```

(Contd.)

```
while count<=20:
    if count%5 == 0:
        sum=sum+count
    count=count+1
print("The Sum of Numbers from 1 to 20 divisible by 5 is: ",sum)
```

Output

```
The Sum of Numbers from 1 to 20 divisible by 5 is:  50
```

PROGRAM 5.6 | Write a program using the while loop to print the factorial of a number.

Factorial of 6 = 6*5*4*3*2*1 = 720

```
Num=int(input("Enter the number:"))
fact=1
ans=1
while fact<=num:
    ans=ans*fact
    fact=fact+1
print("Factorial of",num," is: ",ans)
```

Output

```
Enter the number:6
Factorial of 6 is:  720
```

Note: The factorial of a number is defined as the product of all the numbers from 1 to n.

PROGRAM 5.7 | Write a program to check whether the number entered is an Armstrong number or not.

$153 = 1^3 + 5^3 + 3^3 = 153$

```
num=int(input("Please enter the number: "))
sum=0
x=num
while num>0:
    d=num%10
    num=num // 10
    sum=sum+(d*d*d)
```

(Contd.)


```
if(x==sum):
    print("The number ",  x ,"is Armstrong Number")
else:
    print(" The number ", x ,"is not Armstrong Number")
```

Output

```
Please enter the number: 153
The number 153 is Armstrong Number
```

Note: An Armstrong number is a number which is equal to the sum of the cube of its digits

5.3 THE `range()` FUNCTION

There is a inbuilt function in Python called **`range()`**, which is used to generate a list of integers. The range function has one, two or three parameters. The last two parameters in **`range()`** are optional.

The general form of the range function is:

```
range(begin, end, step)
```

The 'begin' is the first beginning number in the sequence at which the list starts.
The 'end' is the limit, i.e. the last number in the sequence.
The 'step' is the difference between each number in the sequence.

5.3.1 Examples of `range()` Function

Example 1

Create a list of integers from 1 to 5.

```
>>> list(range(1,6))
    [1,2,3,4,5]
```

range(1,6) function is used in the above example. It generates a list of integers starting from 1 to 5. Note that the second number, i.e. 6 is not included in the elements of this list. By default, the difference between the two successive numbers is one.

Note: The above range (1,6) is equivalent to range(6). The output of both the range functions will be the same.

Example 2

Create a list of integers from 1 to 20 with a difference of 2 between two successive integers.

```
>>> list(range(1,20,2))
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

range(1,20,2) function is used in the above example. It generates a list of integers starting from 1 with a difference of two between two successive integers up to 20.

Table 5.1 shows different examples of the **range()** function with relevant outputs.

Table 5.1 Examples of range() function

Example of Range Function	*Output*
range(5)	[0, 1, 2, 3, 4]
range(1,5)	[1, 2, 3, 4]
range(1,10,2)	[1, 3, 5, 7, 9]
range(5,0,-1)	[5, 4, 3, 2, 1]
range(5,0,-2)	[5, 3, 1]
range (-4,4)	[-4, -3, -2, -1, 0, 1, 2, 3]
range (-4,4,2)	[-4, -2, 0, 2]
range(0,1)	[0]
range(1,1)	Empty
range(0)	Empty

5.4 THE for LOOP

The **for** loops in Python are slightly different from the **for** loops in other programming languages. The Python for loop iterates through a sequence of objects, i.e. it iterates through each value in a sequence, where the sequence of object holds multiple items of data stored one after another.

In the forthcoming chapters, we will study various sequence type objects of Python, such as string, list and tuples. The syntax of for loop is given as follows:

```
for var in sequence:
    statement(s)
    ____________
    ____________
        ____________
```

5.4.1 Details of for Loop

The **for** loop is a Python statement which repeats a group of statements for a specified number of times. As described in the syntax, the keywords **for** and **in** are essential keywords to iterate the sequence of values. The variable **var** takes on each consecutive value in the sequence and the statements in the body of the loop are executed once for each value. A simple example of for loop is:

```
for var in range(m,n):
    print var
```

As discussed in Section 5.3, the function **range(m, n)** returns the sequence of integers starting from m, m+1, m+2, m+3..... n-1.

PROGRAM 5.8 | Use for loop to print numbers from 1 to 5.

```
for i in range(1,6):
    print(i)
print("End of The Program")
```

Output

```
1
2
3
4
5
End of The Program
```

Explanation In the above program, the sequence of numbers from 1 to 5 is printed. These numbers are generated using the inbuilt **range()** function. The expression **range(1, 6)** creates an object known as an iterable. This allows the for loop to assign the values 1, 2, 3, 4 and 5 to the iteration variable *i*. During the first iteration of the loop, the value of i is 1 within the block. During the second iteration, the value of *i* is 2 and so on.

PROGRAM 5.9 | Display capital letters from A to Z.

```
print(" The Capital Letters A to Z are as follows:")
for i in range(65,91,1):
    print(chr(i),end=" ")
```

Output

```
The Capital Letters A to Z are as follows:
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

Explanation The **range()** function contains three different parameters, viz. (**begin, end, step_size**). As in the above program, the range function contains the values 65, 90 and 1. It indicates to print the characters whose ASCII value starts from 65 and ends at 90. Therefore, the statement print(chr(i),end=" ") is used to print equivalent character value of ASCII value.

5.4.2 Some More Programs on for Loop

PROGRAM 5.10 | Use for loop to print numbers from 1 to 10 in the reverse order.

```
print("Numbers from 1 to 10 in Reverse Order: ")
for i in range(10,0,-1):
    print(i,end=" ")
print("\n End of Program")
```

(Contd.)

Output

```
Numbers from 1 to 10 in Reverse Order:
10 9 8 7 6 5 4 3 2 1
End of Program
```

PROGRAM 5.11 | Write a program to print squares of the first five numbers.

```
for i in range(1,6):
    square=i*i
    print("Square of ",i," is: ",square)
print("End of Program")
```

Output

```
Square of 1 is:  1
Square of 2 is:  4
Square of 3 is:  9
Square of 4 is:  16
Square of 5 is:  25
End of Program
```

PROGRAM 5.12 | Write a program to print even numbers from 0 to 10 and find their sum.

```
sum=0
print("Even numbers from 0 to 10 are as follows")
for i in range(0,11,1):
    if i%2==0:
        print(i)
        sum=sum+i
print("Sum of Even numbers from 0 to 10 is = ",sum)
```

Output

```
Even numbers from 0 to 10 are as follows
0
2
4
6
8
10
Sum of Even numbers from 0 to 10 is = 30
```


PROGRAM 5.13 | Write a program to calculate the sum of numbers from 1 to 20 which are not divisible 2, 3 or 5.

```
Sum=0
print("Numbers from 1 to 20 which are not divisible by 2,3,or 5")
for i in range(1,20):
    if i%2==0 or i%3==0 or i%5==0:
        print("")
    else:
        print(i)
        sum=sum+i
print("Sum of Even numbers from 1 to 10 is = ",sum)
```

Output

```
Numbers from 1 to 20 which are not divisible by 2, 3, and 5
1
7
11
13
17
19
Sum of Even numbers from 1 to 10 is = 68
```

PROGRAM 5.14 | Write a program that prompts a user to enter four numbers and find the greatest number among the four numbers entered.

```
Num1=int(input("Enter the first Number:"))
num2=int(input("Enter the first Number:"))
num3=int(input("Enter the first Number:"))
num4=int(input("Enter the first Number:"))
sum=num1+num2+num3+num4
print("The sum of Entered 5 Numbers is = ",sum)
for i in range(sum):
    if i==num1 or i==num2 or i==num3 or i==num4:
        Large=i
print(" Largest Number  = ",Large)
print("End of Program")
```

Output

```
Enter the first Number: 4
Enter the first Number: 3
Enter the first Number: 12
```

(Contd.)

```
Enter the first Number: 2
The sum of Entered 5 Numbers is = 21
Largest Number = 12
End of Program
```

PROGRAM 5.15 | Write a program to generate a triangular number.

If the number entered is 5, its triangular number would be (1+2+3+4+5) = 15.

```
Num=int(input("Please enter the Number: "))
Triangular_Num=0
for i in range(Num,0,-1):
    Triangular_Num=Triangular_Num+i
print(" Triangular Number of ",Num," is = ",Triangular_Num)
```

Output

```
Please enter the Number: 10
Triangular Number of 10 is = 55
```

Note: A triangular number is nothing but the summation of 1 to the given number.

PROGRAM 5.16 | Write a program to print Fibonacci series up to 8.

First_Number = 0

Second_Number = 1

Fibonacci Series = 0 1 1 2 3 5 8 13 21 34 55

```
First_Number=int(input("Please enter First Number:"))
Second_Number=int(input("Please enter First Number:"))
Limit=int(input(" Number of Fibonacci Numbers to be Print:  "))
print(First_Number,end=" ")
print(Second_Number,end=" ")
for i in range(Limit+1):
    sum=First_Number+Second_Number
    First_Number=Second_Number
    Second_Number=sum
    print(sum,end=" ")
```

(Contd.)

Output

```
please enter First Number:0
please enter First Number:1
 Number of Fibonacci Numbers to be Print:  8
0 1 1 2 3 5 8 13 21 34 55
```

5.5 NESTED LOOPS

The **for** and **while** loop statements can be nested in the same manner in which the if statements are nested. Loops within the loops or when one loop is inserted completely within another loop, then it is called **nested loop**.

PROGRAM 5.17 | Write a program to demonstrate the use of the nested for loop.

```
for i in range(1,4,1):                  #Outer Loop
    for j in range(1,4,1):              #Inner Loop
        print("i = ",i," j = ",j," i + j =",i + j)
print("End of Program")
```

Output

```
i = 1 j = 1 i + j = 2
i = 1 j = 2 i + j = 3
i = 1 j = 3 i + j = 4
i = 2 j = 1 i + j = 3
i = 2 j = 2 i + j = 4
i = 2 j = 3 i + j = 5
i = 3 j = 1 i + j = 4
i = 3 j = 2 i + j = 5
i = 3 j = 3 i + j = 6
End of Program
```

Explanation In the above program, we have used two loops. One is the outer loop and the other is the inner loop. The inner loop 'j' terminates when the value of j exceeds 3. Whereas, outer loop 'i' terminates when the value of i exceeds 3.

PROGRAM 5.18 | Write a program to display multiplication tables from 1 to 5.

```
Print("Multiplication Table from 1 to 5 ")
for i in range(1,11,1):                 #Outer Loop
    for j in range(1,6,1):              #Inner Loop
```

(Contd.)

```
        print(format(i * j,"4d"),end=" ")
    print()
print("End of Program")
```

Output

```
Multiplication Table from 1 to 5
   1    2    3    4    5
   2    4    6    8   10
   3    6    9   12   15
   4    8   12   16   20
   5   10   15   20   25
   6   12   18   24   30
   7   14   21   28   35
   8   16   24   32   40
   9   18   27   36   45
  10   20   30   40   50
End of Program
```

Explanation The program contains two for loops. The 'j' for loop is the innermost for loop and the 'i' for loop is the outermost for loop. The outermost loop 'i' executes for 10 times. For each value of 'i', the innermost loop 'j' executes 5 times. At the same time for each value of 'i', the product i*j is carried out. To align the numbers properly, the program formats the product of i*j using **format(i*j,"4d")**. The digit **4d** within **format()** specifies a decimal integer format with width 4.

5.5.1 Some More Programs on Nested Loops

PROGRAM 5.19 | Write a program to display the pattern of stars given as follows:

```
*   *   *   *   *
*   *   *   *
*   *   *
*   *
*
```

```
print(" Star Pattern Display")
num=7
x=num
for i in range(1,6,1):
    num=num-1;
    for j in range(1,num,1):
        print(" * ",end=" ")
    x=num-1
```

(Contd.)


```
    print()
print("End of Program")
```

Output

```
Star Pattern Display
*   *   *   *   *
*   *   *   *
*   *   *
*   *
*
End of Program
```

PROGRAM 5.20 | Write a program to display the pattern of stars given as follows:

```
*
*   *
*   *   *
*   *   *   *
*   *   *   *   *
```

```
print(" Star Pattern Display")
num=1
x=num
for i in range(1,6,1):
    num=num+1;
    for j in range(1,num,1):
        print(" * ",end=" ")
        x=num+1
    print()
print("End of Program")
```

Output

```
Star Pattern Display
*
*   *
*   *   *
*   *   *   *
*   *   *   *   *
End of Program
```

PROGRAM 5.21 | Write a program to display the pattern of numbers given as follows:

```
1
1 2
```

```
1 2 3
1 2 3 4
1 2 3 4 5
```

```
print(" Number Pattern Display")
num=1
x=num
for i in range(1,6,1):
    num=num+1;
    for j in range(1,num,1):
        print(j, end=" ")
        x=num+1
    print()
print("End of Program")
```

Output

```
Number Pattern Display
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
End of Program
```

PROGRAM 5.22 | Write a program to display the pattern of numbers given as follows:

```
1
1 2
1 2 3
1 2 3 4
1 2 3
1 2
1
```

```
print(" Number Pattern Display")
num=1
x=num
for i in range(1,5,1):
    num=num+1;
    for j in range(1,num,1):
        print(j, end=" ")
        x=num+1
    print()
```

(Contd.)


```
num=5
x=num
for i in range(1,5,1):
    num=num-1;
    for j in range(1,num,1):
        print(j, end=" ")
        x=num-1
    print()
```

Output

```
Number Pattern Display
1
1 2
1 2 3
1 2 3 4
1 2 3
1 2
1
```

5.6 THE break STATEMENT

The keyword **break** allows a programmer to terminate a loop. When the break statement is encountered inside a loop, the loop is immediately terminated and the program control automatically goes to the first statement following the loop. The flowchart for break is shown in Figure 5.3.

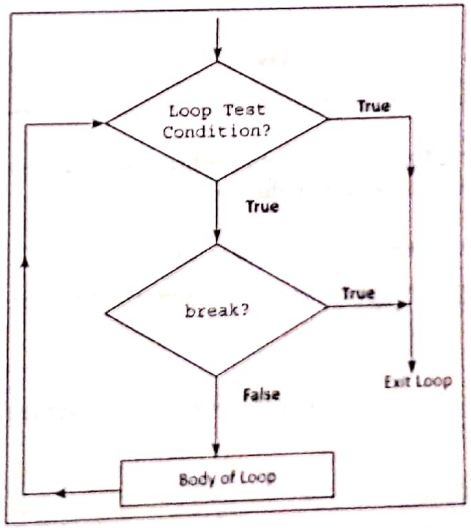


Figure 5.3 Flowchart for **break** statement

The working of break in while and for loop is shown as follows:

Working of break in while loop:

```
while test-Boolean-expression:
    body of while
    if condition:
        break
    body of while
statement(s)
```

Working of break in for loop:

```
for var in sequence:
    body of for
    if condition:
        break
    body of for
statement(s)
```

PROGRAM 5.23 | Write a program to demonstrate the use of the break statement.

```
print("The Numbers from 1 to 10 are as follows:")
for i in range(1,100,1):
    if(i==11):
        break
    else:
        print(i, end=" ")
```

Output

```
The Numbers from 1 to 10 are as follows:
1 2 3 4 5 6 7 8 9 10
```

Explanation The above program prints the numbers from 0 to 10 on the screen. The loop terminates because 'break' causes immediate exit from the loop.

PROGRAM 5.24 | Check if the number entered is prime or not.

```
num=int(input("Enter the Number:"))
x=num
for i in range(2,num):
        if num%i==0:  #Check if entered number is divisible by i
            flag=0
            break
      else:
```

(Contd.)


```
            flag=1
if(flag==1):
    print(num," is Prime ")
else:
    print(num," is not prime ")
```

Output

```
#Test case 1:
Enter the Number:23
23 is Prime
#Test case 2:
Enter the Number:12
12 is not prime
```

Explanation The number is read from the user through the keyboard. A prime number should be divisible by 1 and itself. Therefore, the variable 'i' is iterated from 2 to one less than the number entered. Each value of 'i' is used to check if 'i' can divide the number entered.

5.7 THE continue STATEMENT

The **continue** statement is exactly opposite of the **break** statement. When **continue** is encountered within a loop, the remaining statements within the body are skipped but the loop condition is checked to see if the loop should continue or exit. Flowchart for continue statements is shown in Figure 5.4.

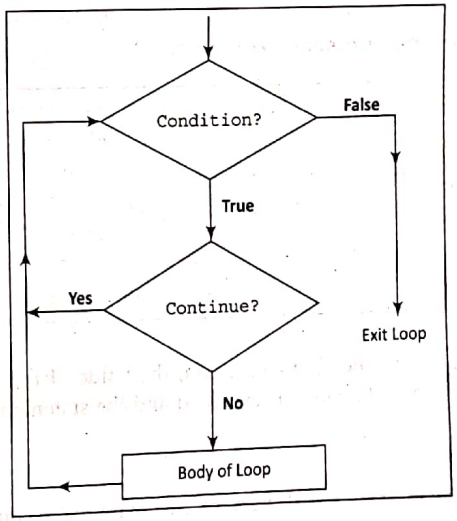


Figure 5.4 Flowchart for **continue** statement

The working of **continue** in while loop is shown as follows:

```
while test-boolean-expression:
    body of while
    if condition:
        continue
    body of while
    statement(s)
```

Alternatively, the working of **continue** in for loop is shown as follows:

```
for var in sequence:
    body of for
    if condition:
        continue
    body of for
        statement(s)
```

The difference between **break** and **continue** is given in Table 5.2.

Table 5.2 Difference between **break** and **continue** functions

Break	*Continue*
Exits from current block or loop.	Skips the current iteration and also skips the remaining statements within the body.
Control passes to the next statement.	Control passes at the beginning of the loop.
Terminates the loop.	Never terminates the loop.

PROGRAM 5.25 | Demonstrate the use of **continue** keyword.

```
for i in range(1,11,1):
    if i == 5:
        continue
    print(i, end=" ")
```

Output

```
1 2 3 4 6 7 8 9 10
```

Explanation In each iteration in the above program, the value of the variable 'i' is checked. If the value of 'i' is 5 then continue statement is executed and the statements following the continue statement are skipped.

PROGRAM 5.26 | Read the string "Hello World" from the user. Make use of **continue** keyword and remove space.


```
str1=str(input("Please Enter the String: "))
print(" Entered String is : ", str1)
print(" After Removing Spaces, the String becomes:")
for i in str1:
    if i==" ":
        continue
    print(i, end="")
```

Output

```
Please Enter the String: Hello World
Entered String is :  Hello World
After Removing Spaces, the String becomes:
HelloWorld
```

Explanation The string str1 is read from the user. Each character of entered string is iterated through the variable 'i'. The statement **if i ==" ":** is used to check if the entered string contains any space. If it contains space, the **continue** statement is executed and the rest of the statements following the **continue** statement are skipped. Finally, we obtain the string without spaces.

MINI PROJECT Generate Prime Numbers using Charles Babbage Function

Charles Babbage discovered the first calculating machine to print prime numbers for a given equation. This mini project will make use of **if**, **if–else**, **if-elif** and for loop concepts of programming.

Let us consider the formula used by Charles Babbage:

$$T = X^2 + X + 41$$

The above formula generates a sequence of values for *T*, which happen to be prime numbers. Thus, calculate the sequence of prime numbers *T* for the values of x starting from 0 to 5. The following table contains prime numbers generated by the Charles Babbage function.

Table 5.3 Evaluation of Charles Babbage function

D2	*D1*	$T = X^2 + X + 41$	*(Value of X)*
		41	0
		43	1
2	2	47	2
2	4	53	3
2	6	61	4
2	8	71	5
2	10		

In Table 5.3, we have calculated prime numbers for all the values of x, i.e. from 0 to 5 using the Charles Babbage function. The D1 is the first difference column and D^2 is the second difference column.

Program Statement

Write a program to generate prime number using the Charles Babbage formula, $(T = X^2 + X + 41)$. The output should be as shown in Table 5.3

Algorithm

- ⊙ **STEP 1:** Since we want 5 values of x, i.e. from 0 to 5. Iterate x as i 5 times.
- ⊙ **STEP 2:** For each value of i, assign the value to x.
- ⊙ **STEP 3:** Calculate the value of T for the value of x
- ⊙ **STEP 4:** If the value of i is equal to 0 then print the values of T and i.
- ⊙ **STEP 5:** If the value of i is greater than 0 and less than 2 then print the values of D, T and i. Else go to Step 6
- ⊙ **STEP 6:** Print value of D2, D, T and i.

Program

```
##########  x2 + x + 41 = T ###############
######  Charles Babbage Function #########
########### for second order #############
x = 0;
print('{}\t{}\t{}\t{}'.format('D2','D1','T','X'))
print('-------------------------')
for i in range(0,5):
    x = i
    T =  (x*x) + x + 41
    if(i == 0):
        print('\t\t{}\t{}'.format(T,i))
    elif( i > 0 and i < 2):
        a =   ((x-1)*(x-1) + (x-1) + 41)
        print('\t{}\t{}\t{}'.format(T -(a),T,i))
    else:
        a  = ((x-1)*(x-1) + (x-1) + 41)
        b  = ((x-2)*(x-2) + (x-2) + 41)
        c  = (T - a)-(a - b)
        print('{}\t{}\t{}\t{}'.format(c,(T - a),T,i))
```

Thus, the above program generates all the prime numbers for all the values of x, i.e. from 0 to 5 for the given equation T = x2 + x + 41. The `format()` method is used to print data in a well-formatted manner.

SUMMARY

- Loop is the process of executing a set of statements for fixed number of times.
- Iteration refers to one time execution of a statements within a loop.
- Python supports two types of loop control statements, i.e. for loop and while loop.
- While loop is condition controlled loop.
- for loop is count controlled loop and it execute statements within the body of loop for fixed number of times.
- The break and continue keywords can be in the loops.
- The break statement exits from the current block or loop and control passes to the next statement.
- The continue statement skips the current iteration and also skips the remaining statements within the body of a loop.

KEY TERMS

- ⇨ **while Loop:** Condition controlled loop
- ⇨ **for Loop:** Count controlled loop
- ⇨ **range():** Generates a list of integers
- ⇨ **nested Loop:** Loop within a loop
- ⇨ **break Statement:** The break statement within a loop helps a programmer to terminate the loop immediately
- ⇨ **continue Statement:** Skips the current iteration and also skips the remaining statement within the body.

REVIEW QUESTIONS

A. Multiple Choice Questions

1. How many times will a loop with header for count in range(5): execute statements in its body?
   a. 5 times b. 4 times
   c. 6 times d. 3 times
2. What will be the output of the following program?

```
count = 35
for x in range(0,10):
    count = count - 1
    if x == 2:
        break
print(count)
```

   a. 35 b. 32
   c. 35, 34 , 33 d. 34, 33, 32

Functions

6

CHAPTER OUTLINE

LEARNING OUTCOMES

After completing this chapter, students will be able to:

- Describe the necessity and importance of functions in programming languages
- Invoke functions with actual parameters and write a program by invoking a function using keyword or positional arguments
- Use local and global scope of a variable appropriately
- Define recursive function and its implementation with programs
- Write functions that return multiple values with programs

6.1 INTRODUCTION

It is difficult to prepare and maintain a large-scale program and the identification of the flow of data subsequently gets harder to understand. The best way to create a programming application is to divide a big program into small modules and repeatedly call these modules.

With the help of functions, an entire program can be divided into small independent modules (each small module is called a function). This improves the code's readability as well as the flow of execution as small modules can be managed easily.

6.2 SYNTAX AND BASICS OF A FUNCTION

A function is a self-contained block of one or more statements that performs a special task when called. The syntax for function is given as follows:

```
def name_of_function(Parameters):      <-- Function Header
        statement1
        statement2
        statement3                     <-- Function Body
        .....
        .....
        statementN
```

The syntax for the Python function contains a **header** and **body**. The function header begins with the **'def'** keyword. The def keyword signifies the beginning of the function's definition. The name of the function is followed by the def keyword. The function header may contain zero or more number of parameters. These parameters are called **formal parameters**. If a function contains more than one parameter then all the parameters are separated by commas. A function's body is a block of statements. The statements within the function's body define the actions that the function needs to perform.

A simple example for creating a function is explained in the following program.

PROGRAM 6.1 Write a program to create a function having a name, display. Print the message, "Welcome to Python Programming" inside the function.

```
def Display():
    print("Welcome  to Python Programming ")
Display()                #call function
```

Output

```
Welcome to Python Programming
```

Explanation In the above program, a function having the name `display()` is created. This function takes no parameters. The body of the function contains only one statement. Finally, function `display()` is called to print the message **"Welcome to Python Programming"** within the block of the function.

PROGRAM 6.2 Write a program to prompt the name of a user and print the welcome message, **"Dear Name_of_user Welcome to Python Programming!!!"**

```
def print_msg():
    str1=input("Please Enter Your Name:")
    print("Dear ",str1," Welcome  to Python Programming ")
print_msg()  #call function
```

Output

```
Please Enter Your Name: Virat
Dear Virat  Welcome  to Python Programming
```

Explanation The function named `print _ msg()` is created. Initially, the function `print _ msg()` is called and the control of the program passes to the called function `print _ msg()`. The function reads the name of the user by making use of the **input** reserved keyword and finally the welcome message is printed.

6.3 USE OF A FUNCTION

A programmer wants to find the sum of numbers starting from 1 to 25, 50 to 75 and 90 to 100. Without functions, he/she will write the code in the following manner.

PROGRAM 6.3 | Write a program to add the sum of digits from 1 to 25, 50 to 76 and 90 to 101 using three different for loops.

```
sum=0
for i in range(1,26):
    sum=sum+i
print('Sum of integers from 1 to 25 is:',sum)

sum=0
for i in range(50,76):
    sum=sum+i
print('Sum of integer from 50 to 76 is:',sum)


sum=0
for i in range(90,101):
    sum=sum+i
print('Sum of integer from 90 to 100 is:',sum)
```

Output

```
Sum of integers from 1 to 25 is: 325
Sum of integer from 50 to 76 is: 1625
Sum of integer from 90 to 100 is: 1045
```


The programmer has created the above code. Observe that the code to compute the sum of numbers is conventional. However, there is a slight difference in the range of numbers, i.e. starting integers and ending integers. Here, all the three for loops contain a different range, i.e. from 1 to 26, 50 to 76 and 90 to 101. Thus, by observing the above code, we can say that it would be better if we could simply write the common code once and then use it repeatedly. A programmer can accomplish this by defining function and using it repeatedly. The code above can be simplified and written using functions as shown in Program 6.4.

PROGRAM 6.4 | Write a program to illustrate the use of functions.

```
def  sum(x,y):
    s=0;
    for i in range(x,y+1):
        s=s+i
    print('Sum of integers from ',x,' to ',y,' is ',s)
sum(1,25)
sum(50,75)
sum(90,100)
```

Explanation The function named sum is created with two parameters 'x' and 'y'. Initially, the function invokes the first function call, i.e. **sum(1, 25)** to compute the sum of numbers from 1 to 25. After computing the sum of numbers from 1 to 25 the control passes to the next function call, i.e. **sum(50, 75)**. After computing the sum of integers from 50 to 75, the third function is finally called, i.e. **sum(90,100)**.

Thus, a programmer can effectively make use of functions to write this program.

a. If a programmer wants to perform a task repetitively, then it is not necessary to re-write the particular block of the program repeatedly. A particular block of statements can be shifted in a user-defined function. The function defined can be then called any number of times to perform a task.
b. Large programs can be reduced to smaller ones using functions. It is easy to debug, i.e. find out the errors in it and hence, it also increases readability.

6.4 PARAMETERS AND ARGUMENTS IN A FUNCTION

Parameters are used to give inputs to a function. They are specified with a pair of parenthesis in the function's definition. When a programmer calls a function, the values are also passed to the function.

While parameters are defined by names that appear in the function's definition, arguments are values actually passed to a function when calling it. Thus, parameters define what types of arguments a function can accept.

Let us consider the example of passing parameters to a function given as follows and use it to differentiate between argument and parameter.

Example

```
def printMax(num1,num2):
        Statemen1
        Statemen2
        ----------
        ----------
        StatementN                    #Define a Function

printMax(10,20)  <-------- Call a function(Invoke)
```

In the above example, printMax(num1, num2) has two parameters, viz. **num1** and **num2**. The parameters **num1** and **num2** are also called formal parameters. A function is invoked by calling the name of the function, i.e. printMax(10,20), where 10, 20 are the actual parameters. Actual parameters are also called arguments. **num1** and **num2** are the parameters of a function. Program 6.5 demonstrates the use of parameters and arguments in a function.

PROGRAM 6.5 | Write a program to find the maximum of two numbers.

```
def printMax(num1,num2):  #Function Definition
    print(" num1 = ",num1)
    print(" num2 = ",num2)

    if num1>num2:
        print("The Number ",num1," is Greater than ",num2)
    elif num2>num1:
        print("The Number ",num2," is Greater than ",num1)
    else:
        print(" Both  Numbers ",num1,",and",num2,"are equal")
printMax(20,10)    #call to function printMax
```

Output

```
num1 =  20
num2 =  10
The Number 20 is Greater than 10
```

Explanation In the above program we have defined a function printMax(). The function contains two parameters, viz. **num1** and **num2**. The function **printMax()** is called by passing the values as arguments to the function. The statement **printMax(10, 20)** causes the value of 10 and 20 to be assigned to parameters **num1** and **num2**, respectively. Finally, based on the values of **num1** and **num2** within the function, greatest of the two numbers is calculated and displayed.

PROGRAM 6.6 | Write a program to find the maximum of two numbers.

```
def calc_factorial(num):
    fact=1
    print(" Entered Number is: ",num)
    for i in range(1,num+1):
        fact=fact*i
    print("Factorial of Number ",num," is  = ",fact)
number=int(input("Enter the Number:"))
calc_factorial(number)
```

Output

```
Enter the Number:5
Entered Number is:  5
Factorial of Number 5  is  =  120
```

6.4.1 Positional Arguments

Consider the question—If there are more than one parameters, how does Python know which argument in the call statement has to be assigned to which parameter?

The answer is quite simple. The parameters are assigned by default according to their position, i.e. the first argument in the call statement is assigned to the first parameter listed in the function definition. Similarly, the second argument in the call statement is assigned to the second parameter listed in the function's definition and so on.

Consider a simple example to demonstrate the use of positional arguments.

Example

```
def Display(Name,age):
    print("Name = ",Name,"age = ",age)
Display("John",25)
Display(40,"Sachin")
```

In the above example, the evaluation of statement **Display("John",25)** prints the result as Name = John and age = 25. However, the statement **Display(40,"Sachin")** has a different meaning. It passes 40 to name and Sachin to age. It means the first argument binds to the first parameter and the second argument binds to the second parameter. This style of matching up arguments and parameter is called **positional argument style** or **positional parameter style**.

In the above example, the function definition Display(Name, age) contains two parameters. Thus, the call is made to function `Display()` by passing exactly two parameters.

What will be the output of the following program?

```
def Display(Name,age):
    print("Name = ",Name,"age =  ",age)
Display("John")
```

Output

```
Prints the error message

Traceback (most recent call last):
File "C:\Python34\keyword_1.py", line 3, in <module>
Display("John")TypeError: Display() missing 1 required positional
argument: 'age'
```

Explanation In the above program, there is no output due to an error. The third line of the program contains the statement Display("John"), i.e. the statement has made a call to function Display(name, age). As the function call contains lesser number of arguments as compared to the function definition, Python will report a missing argument error.

> Note: Python will show an error when an incorrect number of arguments are passed to the function call. The arguments must match the parameters in order, number and type as defined in the function.

6.4.2 Keyword Arguments

An alternative to positional argument is keyword argument. If a programmer knows the parameter name used within the function then he/she can explicitly use the parameter name while calling the function. A programmer can pass a keyword argument to a function by using its corresponding parameter name rather than its position. This can be done by simply typing **Parameter_name = value** in the function call.

Syntax to call a function using keyword argument is:

```
Name_of_Function(pos_args,keyword1=value,keyword2=value2.....)
```

PROGRAM 6.7 | Write a simple program on keyword argument.

```
def Display(Name,age):
    print("Name = ",Name,"age =  ",age)
Display(age=25,Name="John")  #Call function using keyword
                             arguments
```

Output

```
Name = John age =  25
```


Explanation Thus, in the above program, the statement **Display(age=25,Name="John")** passes the value 25 to the parameter 'age' and 'John' to the parameter 'Name'. It means arguments can appear in any order using keyword arguments.

Precautions for Using Keyword Arguments

1. A positional argument cannot follow a keyword argument.
   Example: Consider the function definition,

   ```
   def  Display(num1,num2):
   ```

   Thus, a programmer can invoke the above `Display()` function as:

   ```
   Display(40,num2=10)
   ```

   But, he/she cannot invoke the function as:

   ```
   Display(num2=10,40)
   ```

   because the positional argument 40 appears after the keyword argument **num2=10**.
2. A programmer cannot duplicate an argument by specifying it as both, a positional argument and a keyword argument.
   Example: Consider the function definition,

   ```
   def  Display(num1,num2):
   ```

   Thus, a programmer cannot invoke the above `Display()` function as

   ```
   Display(40,num1=40)    #Error
   ```

   because he/she has specified multiple values for parameter num1.

6.4.3 Parameter with Default Values

Parameters within a function's definition can have default values. We can provide default value to a parameter by using the assignment (=) operator.

PROGRAM 6.8 | Write a program to illustrate the use of default values in a function's definition.

```
def greet(name,msg="Welcome to Python!!"):
    print(" Hello ",name,msg)
greet("Sachin")
```

Output

```
Hello Sachin Welcome to Python!!
```

In the above example, the function `greet()` has the parameter **name.** The parameter **name** does not have any default value and is mandatory during a function call. On the other hand, the parameter **msg** has a default value as "**Welcome to Python!!**". Hence, it is optional during

a function call. If a value is provided, it will overwrite the default value. Here are some valid function calls to this function.

```
#Test case 1
>>> greet("Amit")
```

Output

```
Hello Amit Welcome to Python!!
```

```
#Test case 2
>>> greet("Bill Gates","How are You?")
```

Output

```
Hello Bill Gates How are You?
```

The above example has two test cases. In the first test case, only one argument is passed to the function `greet()` during the function call. And the second parameter is not passed. In such a case, Python uses the default value of a parameter specified during function definition. But in case of test case 2, both the parameters **greet("Bill Gates","How are You?")** are passed during the function call. In such a situation, the new argument value overwrites the default parameter value.

> **Note:** During a function's definition, any number of parameters in a function can have default values. But once we have a default value to a parameter, all the parameters to its right must also have default values. For example, if we define a function's definition as:
>
> `def greet(msg="Welcome to Python!!", name):  #Error`

Python will give the error as:

Syntax Error: Non-default argument follows default argument

PROGRAM 6.9 | Write a program to calculate the area of a circle using the formula:

Area of Circle = pi*(r)^2

Declare the default parameter value of pi as 3.14 and radius as 1.

```
def area_circle(pi=3.14,radius=1):
        area=pi*radius*radius
        print("radius=",radius)
        print(" The area of Circle  = ",area)
area_circle()
area_circle(radius=5)
```

Output

```
radius= 1
The area of Circle = 3.14
radius= 5
The area of Circle = 78.5
```

(Contd.)

What will be the output of the following program?

```
def  disp_values(a,b=10,c=20):
    print(" a = ",a," b = ",b,"c= ",c)
disp_values(15)
disp_values(50,b=30)
disp_values(c=80,a=25,b=35)
```

Output

```
a =  15  b =  10 c=  20
a =  50  b =  30 c=  20
a =  25  b =  35 c=  80
```

Explanation In the above program, the function named **disp_values** has one parameter without a default argument value, followed by two parameters with default argument values.

During the first function call **disp_values(15)**, parameter a gets the value 15 and parameters b and c get the default values 10 and 20, respectively.

During the function call disp_values(50,b=30), parameter a gets the value 50, parameter b gets the value 30, i.e. the value of b is overwritten and parameter c gets the default value 20.

During the function call disp_values(c=80,a=25,b=35), the default values of parameters b and c are replaced by the new values 35 and 80, respectively.

6.5 THE LOCAL AND GLOBAL SCOPE OF A VARIABLE

Variables and parameters that are initialised within a function including parameters, are said to exist in that function's local scope. Variables that exist in local scope are called **local variables**. Variables that are assigned outside functions are said to exist in global scope. Therefore, variables that exist in global scope are called **global variables**.

PROGRAM 6.10 | Write a program to show local scope vs global scope.

```
p = 20          #global variable p
def Demo():
    q = 10      #Local variable q
    print('The value of Local variable q:',q)
    #Access global variable p within this function
    print('The value of Global Variable p:',p)
Demo()
```

(Contd.)

```
#Access global variable p outside the function Demo()
print('The value of global variable  p:',p)
```

Output

```
The value of Local variable q: 10
The value of Global Variable p: 20
The value of global variable p: 20
```

Explanation In the above example, we have created one **local variable 'q'** and one **global variable 'p'**. As global variables are created outside all functions and are accessible to all functions in their scope, in the above example as well the global variable **'p'** is accessed from the function **Demo()** and it is also accessed outside the function.

Local Variables Cannot be Used in Global Scope

PROGRAM 6.11 | Write a program to access a local variable outside a function.

```
def Demo():
    q = 10      #Local variable q
    print('The value of Local variable q:',q)
Demo()
#Access local variable q outside the function Demo()
print('The value of local variable  q:',q)    #Error
```

Output

```
The value of Local variable q: 10
Traceback (most recent call last):
  File "C:/Python34/loc1.py", line 6, in <module>
    print('The value of local variable  q:',q)    #Error
NameError: name 'q' is not defined
```

Explanation The local variable 'q' is defined within the function `Demo()`. The variable 'q' is accessed from the function `Demo()`. The scope of a local variable lies within the block of the function, i.e. it starts from its creation and continues up to the end of the function. Therefore, any attempt to access the variable from outside of the function causes an error.

Note: Accessing a local variable outside the scope will cause an error.

6.5.1 Reading Global Variables from a Local Scope

Consider the following program where global variables are read from a local scope.

PROGRAM 6.12 | Write a program where global variables are read from a local scope.

```
def Demo():
    print(S)
S='I Love Python'
Demo()
```

Output

```
I Love Python
```

Explanation Before calling the function **Demo()**, the variable 's' is defined as a string, "I Love Python". However, the body of the function `Demo()` contains only one statement **print(s)** statement. As there is no local variable 's' defined within the function `Demo()`, the **print(s)** statement uses the value from the global variable. Hence, the output of the above program will be 'I Love Python'.

6.5.2 Local and Global Variables with the Same Name

What will be the output of the above program if we change the value of 's' inside the function `Demo()`? Will it affect the value of the global variable? Program 6.13 demonstrates the change in value 's' within the function `Demo()`.

PROGRAM 6.13 | Write a program to change the value 's' within the function.

```
def Demo():
    S='I Love Programming'
    print(S)
S='I Love Python'
Demo()
print(S)
```

Output

```
I Love Programming
I Love Python
```

Explanation As we know, the scope of a local variable lies within the block of a function. Initially, the value of 's' is assigned as 'I Love Python'. But after calling the function `Demo()`, the value of 's' is changed to 'I Love Programming'. Therefore, the print statement within the function `Demo()` will print the value of the local variable 's', i.e. 'I Love Programming'. Whereas the print statement after the `Demo()` statement, will print the old value of the variable 's', i.e. 'I Love Python'.

6.5.3 The Global Statement

Consider a situation where a programmer needs to modify the value of a global variable withi a function. In such a situation, he/she has to make use of the **global** statement. The followir program demonstrates the use of the **global** statement.

PROGRAM 6.14 | Write a program without using the global statement.

```
a = 20
def Display():
    a = 30
    print(' The value of a in function:',a)
Display()
print('The value of an outside function:',a)
```

Output

```
The value of a in function: 30
The value of an outside function: 20
```

Explanation In the above program, we have assigned the value of an outside function as 20. By chance, a programmer uses the same name, i.e. 'a' inside the function. But in this case the variable 'a' within the function is local to the function. Therefore, any changes to the value associated with the name inside the function will change the value of the local variable itself and not the value of the global variable 'a'.

PROGRAM 6.15 | Write a program using the global statement.

```
a = 20
def Display():
    global a
    a = 30
    print(' The value of a in function:',a)
Display()
print('The value of an outside function:',a)
```

Output

```
The value of a in function: 30
The value of an outside function: 30
```

Explanation The program demonstrates the use of the **global** keyword. The **global** keyword has been used before the name of the variable to change the value of the local variable. Since the value of the global variable is changed within the function, the value of 'a' outside the function will be the most recent value of 'a'.

6.6 THE `return` STATEMENT

The `return` statement is used to **return a value from the function.** It is also used to return from a function, i.e. break out of the function.

PROGRAM 6.16 | Write a program to return the minimum of two numbers.

```
def minimum(a,b):
    if a<b:
        return a
    elif b<a:
        return b
    else:
        return "Both the numbers are equal"
print(minimum(100,85))
```

Output

```
8 is minimum
```

Explanation The minimum function returns the minimum of the two numbers supplied as parameters to a function minimum. It uses simple if..elif..else statement to find the minimum value and then returns that value.

PROGRAM 6.17 | Write a function calc_Distance(x1, y1, x2, y2) to calculate the distance between two points represented by Point1(x1, y1) and Point2 (x2, y2). The formula for calculating distance is:

$$\text{Distance} = \sqrt{(x2 - x1)^2 + (y2 - y1)^2}$$

```
import math
def calc_Distance (x1, y1, x2, y2):
    print(" x1 = ",x1)
    print(" x2 = ",x2)
    print(" y1 = ",y1)
    print(" y2 = ",y2)
    dx=x2-x1
    dx=math.pow(dx,2)
    dy=y2-y1
    dy=math.pow(dy,2)
    z = math.pow((dx + dy), 0.5)
    return z
print("Distance = ",(format(calc_Distance(4,4,2,2),".2f")))
```

Output

```
 x1 =  4
 x2 =  2
 y1 =  4
 y2 =  2
Distance =  2.83
```

PROGRAM 6.18 For a quadratic equation in the form of ax^2+bx+c, the discriminant D, is $b^2 - 4ac$. Write a function to compute the discriminant D, that returns the following output depending on the discriminant D.

if D > 0: The Equation has two Real Roots
if D = 0: The Equation has one Real Root
if D < 0: The Equation has two Complex Roots

```
def quad_D(a,b,c):
    d=b*b-4*a*c
    print("a = ",a)
    print("a = ",b)
    print("a = ",c)
    print("D = ",d)
    if d>0:
        return "The Equation has two Real Roots"
    elif d<0:
        return "The Equation has two Complex Roots"
    else:
        return "The Equation has one Real Root"
print(quad_D(1,2,5))
```

Output

```
a =  1
a =  2
a =  3
D = -8
The Equation has two Complex Roots
```

Note: The return statement without a value is equivalent to return 'None'. Where, 'None' is a special type in Python that represents nothingness.

PROGRAM 6.19 Write a program to pass the radius of a circle as a parameter to a function **area_of_circle()**. Return the value **none** if the value of the radius is negative or return the area of the circle.

```
def area_of_Circle(radius):
    if radius<0:
        print(" Try Again, Radius of circle cannot be Negative ")
        return
    else:
        print("Radius = ",radius)
```

(Contd.)


```
        return 3.1459*radius**radius
print("Area of Circle =",area_of_Circle(2))
```

Output

```
Radius = 2
Area of Circle = 12.5836
```

Explanation In the above program, the user has to pass the radius of the circle as a parameter to the function area_of_circle(). If the radius of the circle is positive then it calculates and returns the area of the circle. Whereas, if the entered radius of the circle is negative, it returns a none value, i.e. it returns nothing.

What will be the output of the above program?

```
def calc_abs(x):
    if x<0:
        return -x
    elif x>0:
        return x
print(calc_abs(0))
```

Output

```
None
```

Explanation The above piece of code is incorrect because when the user has passed the value 0 as a parameter to the function calc_abs(), the value of x happened to be 0. Then neither condition is true and the function ends without executing any return statement. In such a situation, the function returns a special value called **None**.

6.6.1 Returning Multiple Values

It is possible to return multiple values in Python.

PROGRAM 6.20 Write a function calc_arith_op(num1, num2) to calculate and return at once the result of arithmetic operations such as addition and subtraction.

```
def calc_arith_op(num1, num2):
    return num1+num2, num1-num2    #Return multiple values
print(" ",calc_arith_op(10,20))
```

Output

```
 (30, -10)
```

Explanation In the above program, two parameters, viz. num1 and num2 are passed to a function **calc_arith_op()**. Within the body of the function, the **single return** statement computes the addition and subtraction of the two numbers. Finally, the single return statement returns the result of both the arithmetic operations, viz. addition and subtraction.

6.6.2 Assign Returned Multiple Values to Variable(s)

It is also possible for a function to perform certain operations, return multiple values and assign the returned multiple values to a multiple variable.

PROGRAM 6.21 | Write a program to return multiple values from a function.

```
def compute(num1):
    print("Number = ",num1)
    return num1*num1, num1*num1*num1
square,cube=compute(4)
print("Square = ",square,"Cube = ",cube)
```

Output

```
Number = 4
Square = 16 Cube = 64
```

Explanation The number is passed to the function `compute()`. The return statement calculates the square and cube of a passed number. After computation, it returns both the values simultaneously. The returned square of a number is assigned to a variable square and the returned cube of a number is assigned to a variable cube.

6.7 RECURSIVE FUNCTIONS

So far, we have seen that it is legal for one function to call another function. In programming, there might be a situation where a function needs to invoke itself. Python also supports the recursive feature, which means that a function is repetitively called by itself. Thus, a function is said to be recursive **if a statement within the body of the function calls itself.**

Let us consider a simple example of recursion. Suppose we want to calculate the factorial value of an integer. We know that the factorial of a number is the product of all the integers between 1 and that number, i.e. n! is defined as n * (n-1)!.

Consider the following example.

Formula to calculate the factorial of a number (n)! = n*(n-1)!

5!= 5*(4)!
= 5*4*(3)!
= 5*4*3*(2)!
= 5*4*3*2*(1)
= 120

PROGRAM 6.22 | Calculate the factorial of a number using recursion.

```
def factorial(n):
    if n==0:
        return 1
    return n*factorial(n-1)
print(factorial(5))
```

Output

```
120
```

Explanation In the above program, `factorial()` is a recursive function. The number is passed to function `factorial()`. When the function factorial is executed, it is repeatedly invoked by itself. Every time a function is invoked, the value of 'n' is reduced by one and multiplication is carried out. The recursion function produces the number 5, 4, 3, 2 and 1. The multiplication of these numbers is carried out and returned. Finally, the print statement prints the factorial of the number.

PROGRAM 6.23 | Write a recursive function which computes the n^{th} Fibonacci number. Fibonacci numbers are defined as:

Fib(0)= 1,
Fib(1) = 1
Fib(n)= Fib(n-1)+Fib(n-2).

Write this as a Python code and then find the 8^{th} Fibonacci number.

```
def fib(n):
    if n==0:
        return 1
    if n==1:
        return 1
    return fib(n-1)+fib(n-2)
print(" The Value of 8th Fibonacci number = ",fib(8))
```

Output

```
The Value of 8th Fibonacci number = 34
```

6.8 THE LAMBDA FUNCTION

Lambda functions are named after the Greek letter λ (lambda). These are also known as **anonymous functions**. Such kind of functions are not bound to a name. They only have a code to execute that which is associated with them. The basic syntax for a lambda function is:

Name = lambda(variables): Code

Let us consider a simple example which calculates the cube of a number using simple concepts of a function.

```
>>> def func(x):
    return x*x*x

>>> print(func(3))
27
```

Without the lambda function Now we will calculate the cube of a number using the lambda function.

```
>>> cube = lambda x: x*x*x     #Define lambda function
>>> print(cube(2))             #Call lambda function
8
```

Using the lambda function Thus, in the above example, both the functions `func()` and `cube()` do exactly the same thing. The statement **cube = lambda x: x*x*x** creates a **lambda** function called cube, which takes a single argument and returns the cube of a number.

> **Note:** (a) A lambda function does not contain a return statement.
> (b) It contains a single expression as a body and not a block of statements as a body.

MINI PROJECT Calculation of Compound Interest and Yearly Analysis of Interest and Principal Amount

This mini project will use programming features, such as **decision, control statements** and **functions** to calculate the interest deposited for a principal amount for some period of time **'n'** at some interest **'r'**.

Explanation and Calculation of Compound Interest

Compound interest is the addition of interest to the initial principal amount and also to the accumulated interest over preceding periods of a deposit or loan.

Compound interest is different from simple interest. In simple interest, there is no interest on interest. Simply interest is added to the principal amount.

The formula to calculate annual compound interest including principal amount is

$$CI = P \cdot \left(1 + \frac{r}{t}\right)^{nt} - P$$

where,

P = Principal investment amount

r = Annual interest rate

n = Number of years the money is invested

t = Number of times the interest is compounded per year

The formula to calculate interest if it is compounded once per year is

$$I = P * (1 + r)^n \text{ ------ (A)}$$

Thus, 'I' gives future values of an investment or loan which is compound interest plus the principal. So, we are going to use formula 'A'.

Example

Let principal (P) amount = ₹10,000

Rate (R) of interest = 5

Number of Years = 7

Value of compound interest per year (t) = 1

We will use the above formula 'A' to calculate the interest accumulated each year.

Year	Starting Balance	Interest	Ending Balance
1	10000.00	500.00	10500.00
2	10500.00	525.00	11025.00
3	11025.00	551.25	11576.25
4	11576.25	578.81	12155.06
5	12155.06	607.75	12762.82
6	12762.82	638.14	13400.96
7	13400.96	670.05	14071.00

Algorithm to Calculate Compound Interest

- **STEP 1:** Read the principal amount, rate of interest and number of years the amount is to be deposited. (Assuming interest is compounded once per year).
- **STEP 2:** Pass the principal, rate of interest and the number of years to the function named **Calculate_Compund_Interest()**.
- **STEP 3:** Iterate **for loop** for 'n' number of times to calculate interest generated per year by using the formula for compound interest as discussed above.
- **STEP 4:** Display the final compound interest.

PROGRAM STATEMENT Write a program to calculate compound interest for principal amount as ₹10,000, at rate of interest as 5% and number of years the amount is deposited as 7 years.

```
def Calculate_Compund_Interest(p,n,r):
    print('StartBalance\t','\tInterest\t','Ending Balance')
    total = 0
```

(Contd.)

```
    x= r/100
    tot = 0
    for i in range(1,n+1):
        z_new = p*(1 + x) **i - p
        z_old = p*(1 + x)**(i-1) - p
        tot = tot + (z_new - z_old)
        if(i == 1):
            print('{0:.2f}\t'.format(p),end='')
            print('\t{0:.2f}\t'.format(z_new - z_old),end='')
            print('\t\t{0:.2f}\t'.format(z_new+p))
        else:
            print('{0:.2f}\t'.format(p+z_old),end='')
            print('\t{0:.2f}\t'.format(z_new - z_old ),end='')
            print('\t\t{0:.2f}\t'.format(z_new+p))
    print('Total Interest Deposited:Rs{0:.2f}'.format(tot))
p  = int(input('Enter the Principal amount:'))
r  = int(input('Enter the rate of interest:'))
n  = int(input('Enter number of year:'))
Calculate_Compund_Interest(p,n,r)
```

Output

```
Enter the Principal amount:10000
Enter the rate of interest:5
Enter number of year:7

Start Balance      Interest    Ending Balance
10000.00     500.00        10500.00
10500.00     525.00        11025.00
11025.00     551.25        11576.25
11576.25     578.81        12155.06
12155.06     607.75        12762.82
12762.82     638.14        13400.96
13400.96     670.05        14071.00

Total Interest Deposited: Rs 4071.00
```

In the above program, initially principal amount, rate of interest and number of years are read from the user. The same values are passed as a parameter to the function **Calculate_Compund_Interest()**. The for loop is iterated for **n** number of times to calculate the annual interest generated per year. The difference between **Z_new** and **Z_old** in above program gives the interest generated per year. At last, the compound interest is displayed.

SUMMARY

- A function is a self-contained block of one or more statements that perform a special task when called.
- A function's definition in Python begins with the def keyword followed by the function's name, parameter and body.
- The function header may contain zero or more number of parameters.
- Parameters are the names that appear in a function's definition.
- Arguments are the values actually passed to a function while calling a function.
- Arguments to a function can be passed as positional or keyword arguments.
- The arguments must match the parameters in order, number and type as defined in the function.
- A variable must be created before it is used.
- Variables defined within the scope of a function are said to be local variables.
- Variables that are assigned outside of functions are said to be global variables.
- The return statement is used to return a value from a function.
- Functions in Python can return multiple values.
- Python also supports a recursive feature, i.e. a function can be called repetitively by itself.

KEY TERMS

- **The def Keyword:** Reserved word to define a function
- **Positional Arguments:** By default, parameters are assigned according to their position
- **Keyword Arguments:** Use syntax keyword = Value to call a function with keyword arguments
- **Local and Global Scope of a Variable:** Describes two different scopes of a variable
- **The Return Keyword:** Used to return single or multiple values
- **Lambda:** An anonymous function

REVIEW QUESTIONS

A. Multiple Choice Questions

1. A variable defined outside a function is referred to as
   a. Local variable
   b. Only variable
   c. Global variable
   d. None of the above
2. Which of the following function headers is correct?
   a. def Demo(P, Q = 10):
   b. def Demo(P=10,Q = 20):
   c. def Demo(P=10,Q)
   d. Both a and c
3. What will be the output of the following program?

```
x = 10
def f():
    x= x + 10
    print(x)
f()
```

Strings

7

CHAPTER OUTLINE

LEARNING OUTCOMES

After completing this chapter, students will be able to:

- Create and use string in programming
- Write programs to access characters within a string using index operators, including accessing characters via negative index
- Use str[start : end] slicing operator to get a substring from larger strings
- Use various inbuilt functions of strings, such as `len()`, `min()` and `max()` functions
- Apply inbuilt operators on strings +, * and compare two different strings using >,>=, <, <=, ==,!= operators
- Use various methods of strings such as `capitalise()`, `upper()`, `lower()`, `swapcase()`, and `replace()` to convert string from one form to another
- Search substrings from a given string using various methods of string such as `find()`, `rfind()`, `endswith()`, `startwith()`
- Format strings by using `ljust()`, `rjust()`, `centre()`,`format()` functions

7.1 INTRODUCTION

Characters are building blocks of Python. A program is composed of a sequence of characters. When a sequence of characters is grouped together, a meaningful string is created. Thus, a string is a sequence of characters treated as a single unit.

In many languages, strings are treated as arrays of characters but in Python a string is an object of the **str** class. This string class has many constructors.

The next section describes constructors and how to access strings.

7.2 THE str CLASS

Strings are objects of the str class. We can create a string using the constructor of str class as:

```
S1=str() #Creates an Empty string Object
S2=str("Hello") #Creates a String Object for Hello
```

An alternative way to create a string object is by assigning a string value to a variable.

Example

```
S1 = ""  # Creates a Empty String
S2= "Hello" # Equivalent to S2=str("Hello")
```

All the characters of a string can be accessed at one time using the index operator. This has been explained in Section 7.4.

7.3 BASIC INBUILT PYTHON FUNCTIONS FOR STRING

Python has several basic inbuilt functions that can be used with strings. A programmer can make use of **`min()`** and **`max()`** functions to return the largest and smallest character in a string. We can also use **`len()`** function to return the number of characters in a string.

The following example illustrates the use of the basic function on strings.

```
>>> a = "PYTHON"
>>> len(a) #Return length i.e. number of characters in string a
6
>>> min(a) #Return smallest character present in a string
'H'
>>> max(a) #Return largest character present in a string
'Y'
```

7.4 THE `index[]` OPERATOR

As a string is a sequence of characters, the characters in a string can be accessed one at a time through the index operator. The characters in a string are **zero based**, i.e. the first character of the string is stored at the 0^{th} position and the last character of the string is stored at a position one less than that of the length of the string. Figure 7.1 illustrates how a string can be stored.

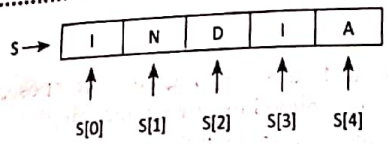


Figure 7.1 Accessing characters in a string using the index operator

Example

```
>>> S1="Python"
>>>S1[0] #Access the first element of the string.
'P'
>>>S1[5] #Access the last element of the String.
'n'
```

Note: Consider a string of length 'n', i.e. the valid indices for such string are from 0 to n-1. If you try to access the index greater than n-1, Python will raise a '**string index out of range**' error. The following example illustrates the same.

```
>>> a='IIT'
>>> a[3]
Traceback (most recent call last):
 File "<pyshell#1>", line 1, in <module>
 a[3]
IndexError: string index out of range
```

7.4.1 Accessing Characters via Negative Index

The negative index accesses characters from the end of a string by counting in backward direction. The **index** of the last character of any non-empty string is always **-1**, as shown in Figure 7.2.

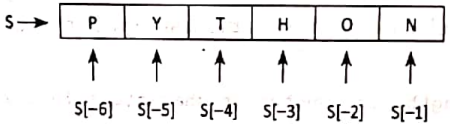


Figure 7.2 Accessing characters in a string using negative index

Example

```
>>> S="PYTHON"
>>> S[-1]#Access the last character of a String 'S'
'N'
>>> S[-2]
'O'
>>> S[-3]
'H'
>>> S[-4]
'T'
>>> S[-5]
'Y'
>>> S[-6]#Access the First character of a String 'S'
'P'
```

Note:

```
S[-n] == S[Length_of(S)-n]
```

Example:

```
S="IIT-Bombay"
>>> S[-3]
>>>'b'
```

Explanation

`S[-3]==S[Len(S)-3]=S[10-3]=S[7].`

Thus, S[-3]==S[7] prints the character stored at index 7 counting in a forward direction or we can say it prints the character stored at index -3 counting in backward direction from the string S.

7.5 TRAVERSING STRING WITH for AND while LOOP

A programmer can use the for loop to traverse all characters in a string. For example, the following code displays all the characters of a string.

PROGRAM 7.1 | Write a program to traverse all the elements of a string using the for loop.

```
S="India"
for ch in S:
    print(ch,end="")
```

Output

```
India
```

Explanation The string 'India' is assigned to the variable S. The for loop is used to print all the characters of a string S. The statement '**for ch in S:**' can read as 'for each character **ch** in **S** print **ch**'.

PROGRAM 7.2 | Write a program to traverse every second character of a string using the for loop.

```
S="ILOVEPYTHONPROGRAMMING"
for ch  in range(0,len(S),2):#Traverse each Second character
    print(S[ch],end=" ")
```

Output

```
I O E Y H N R G A M N
```

7.5.1 Traversing with a while Loop

A programmer can also use the while loop to traverse all the elements of a string. The following example illustrates the use of the while loop to traverse all the characters within a string using the while loop.

PROGRAM 7.3 | Write a program to traverse all the elements of a string using the while loop.

```
S="India"
index=0
while index<len(S):
    print(S[index],end="")
    index=index+1
```

Output

```
India
```

Explanation The while loop traverses a string and displays each character. The condition index<len(S) is checked in each iteration. When the value of an index is equal to the length of the string, the condition is false and the body of loop is not executed. The last character accessed is one less than that of that of the length of the string.

7.6 IMMUTABLE STRINGS

Character sequences fall into two categories, i.e. mutable and immutable. Mutable means changeable and immutable means unchangeable. Hence, strings are immutable sequences of characters.

Consider the following example. Let's see what happens if we try to change the contents of the string.

Example

```
Str1="I Love Python"
Str1[0]="U"
print(Str1)
```

```
ERROR:
TypeError: 'str' object does not support item assignment
```

Explanation

In the above example, we have assigned the string "I Love Python" to Str1. The `index []` operator is used to change the contents of the string. Finally, it shows an error because the strings are **immutable**, which means one cannot change the existing string.

Note: If you want to change the existing string, the best way is to create a new string that is a variation of the original string.

```
Str1="I Love Python"
Str2="U"+Str1[1:]
print(Str2)
```

Output

```
U Love Python
```

Consider the following two similar strings. "Hello" is assigned to two different variables as:

```
Str1="Hello"
Str2="Hello"
```

In the above example both the variables, str1 and str2 have the same content. Thus, Python uses one object for each string which has the same content as shown in Figure 7.3. str1 and str2 refers to the same string object, whereas str1 and str2 have the same ID number.

```
>>>str1="Hello"
>>>str2="Hello"
>>>id(Str1)
53255968
>>>id(Str2)
53255968
```

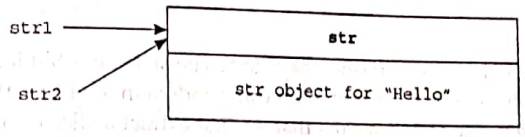


Figure 7.3 String with the same contents share the similar id

7.7 THE STRING OPERATORS

String contains the slicing operator and the slicing with step size parameter is used to obtain the subset of a string. It also has basic concatenation '+', 'in' and repetition '*' operators. The next section describes string operators in more detail.

7.7.1 The String Slicing Operator [start: end]

The slicing operator returns a subset of a string called **slice** by specifying two indices, viz. **start** and **end**. The syntax used to return a subset of a string is:

```
Name_of_Variable_of_a_String[Start_Index: End_Index]
```

Example

```
>>> S="IIT-BOMBAY"
>>> S[4:10]#Returns a Subset of a String
'BOMBAY'
```

The S[4:10] returns a subset of a string starting from start index, i.e. 4 to one index less than that of end parameter of slicing operation, i.e. 10 - 1 = 9.

7.7.2 String Slicing with Step Size

In the above section, we learnt how to select a portion of a string. But how does a programmer select every second character from a string?

This can be done using **step size**. In slicing, the first two parameters are **start index** and **end index**. We need to add a third parameter as **step size** to select the characters from a string with step size.

Syntax

```
Name_of_Variable_of_a_String[Start_Index:End_Index:Step_Size]
```

Example

```
>>>S="IIT-BOMBAY"
>>> S[0:len(S):2]
>>>'ITBMA'
```

Explanation

Initially we have assigned a string "IIT-Bombay" to S. The statement **S[0:len(S):2]** indicates us to select the portion of a string which starts at index 0 and ends at index 10, i.e. the length of the string "IIT-BOMBAY". The step size is 2. It means that we first extract a slice or a portion of the string which starts with the index 0, ends with the index 10 and selects every other second character from the string S.

Some More Complex Examples of String Slicing

```
>>> S="IIT-MADRAS"
>>> S[::]#Prints the entire String
'IIT-MADRAS'

>>> S[::-1]
'SARDAM-TII'#Display the String in Reverse Order

>>>S="IIT-MADRAS"
>>> S[-1:0:-1]#Access the characters of a string from index -1
>>>'SARDAM-TI'
>>>S="IIT-MADRAS"
>>> S[-1:0:-1]#Access the characters of a string from index -1
>>>'SARDAM-TI'
```



```
>>> S[:-1]
#start with the character stored at index 0 & exclude the last character stored
at index -1.
'IIT-MADRA'
```

7.7.3 The String +, * and in Operators

1. ***The + Operator:*** The concatenation operator '+' is used to join two strings.

   Example:

   ```
   >>> S1="IIT "       #The String "IIT" assigned to S1
   >>> S2="Delhi"#The String "Delhi" assigned to S1
   >>> S1+S2
   'IIT Delhi'
   ```

2. ***The * Operator:*** The multiplication (*) operator is used to concatenate the same string multiple times. It is also called **repetition operator.**

   Example:

   ```
   >>> S1="Hello"
   >>> S2=3*S1#Print the String "Hello" three times
   >>> S2
   'HelloHelloHello'
   ```

   Note: S2=3*S1 and S2=S1*3 gives same output

3. **The in and not in Operator:** Both Operators **in** and **not in** are used to check whether a string is present in another string.

   Example:

   ```
   >>> S1="Information Technology"
   #Check if the string "Technology" is present in S1
   >>> "Technology" in S1
   True
   #Check if the string "Technology" is present in S1
   >>> "Engineering" in S1
   False
   >>> S1="Information Technology"
   # Check if the string "Hello" is not present in S1
   >> "Hello" not in S1
   True
   ```

PROGRAM 7.4 | Write a program to print all the letters from word1 that also appear in word2.

Example: Word1 = USA North America
word2= USA South America

```
#Print the letter that appear in word1 &also appears in word2
```

Output

```
USA orth America
```

Solution

```
word1="USA North America"
word2="USA South America"
print("word1=",word1)
print("word2=",word2)
print("The words that appear in word1 also appears in word2")
for letter in word1:
    if letter in word2:
        print(letter,end="")
```

Output

```
word1= USA North America
word2= USA South America
The words that appear in word1 also appears in word2
USA orth America
```

Explanation

In the above program, the string "USA North America" is assigned to word1 and the string "USA South America" is assigned to the String word2. In the for loop, each letter of word1 is compared with all the letters of word2. If a letter of word1 appears in word2 then the particular letter is printed. A programmer can read the above for loop as **for each letter in the first word, if it appears in the second word then print that letter.**

7.8 STRING OPERATIONS

The str class provides different basic methods to perform various operations on a string. It helps to calculate the **length of a string**, to retrieve the individual characters from the given string and to compare and concatenate the two different strings.

7.8.1 String Comparison

Operators such as ==,<,>,<=,>=and != are used to compare the strings. Python compares strings by comparing their corresponding characters.

Example

```
>>> S1="abcd"
>>> S2="ABCD"
>>> S1>S2
True
```


Explanation

The string 'abcd' is assigned to the string S1 and the String 'ABCD' is assigned to S2. The statement S1 > S2 returns True because Python compares the numeric value of each character. In the above example, the numeric value, i.e. ASCII value of 'a' is 97 and ASCII numeric value of 'A' is 65. It means 97 > 65. Thus, it returns True. However, character by character comparison goes on till the end of the string.

Some More Examples of String Comparison

```
>>> S1="abc"
>>> S2="abc"
>>> S1==S2
True
>>> S1="ABC"
>>> S2="DEF"
>>> S1>S2
False

>>> S1="AAA"
>>> S2="AAB"
>>> S2>S1
True

>>> S1="ABCD"
>>> S2="abcd".upper()
>>> S2
'ABCD'
>>> S1>S2
False
>>> S1>=S2
True
```

7.8.2 The String .format() Method()

In Python 2 and 3, programmers can include %s inside a string and follow it with a list of values for each %.

Example

```
>>> "My Name is %s and I am from %s"%("JHON","USA")
'My Name is JHON and I am from USA'
```

In the above example, we have seen how to format a string using % (modulus) operator. However, for more complex formatting, Python 3 has added a new string method called **format()** method. Instead of % we can use {0}, {1} and so on. The syntax for `format()` method is:

```
template.foramt(P0,P1,.....,k0=V0,K1=V1...)
```

whereas the arguments to the .format() method are of two types. It consists of zero or more positional arguments P_i followed by zero or more keyword arguments of the form, $K_i=V_i$.

Example

```
>>> '{} plus {} equals {}'.format(4,5,'Nine')
'4 plus 5 equals Nine'
```

Explanation

The `format()` method is called on the string literal with arguments 4,5 and 'nine'. The empty {} are replaced with the arguments in order. The first {} curly bracket is replaced with the first argument and so on. By default, the index of the first argument in format always start from **zero**. One can also give a position of arguments inside the curly brackets. The following example illustrates the use of index as argument inside the curly bracket.

Example

```
>>>"My Name is {0} and I am from {1}".format("Milinda","USA")
'My Name is Milinda and I am from USA'
```

Explanation

The `format()` method contains various arguments. In the above example, the `format()` method has two arguments, viz. "Milinda" and "USA". The index of the first argument of the `format()` method always starts from 0. Therefore, {0} replaces the 0^{th} argument of the format. Similarly {1} replaces the first argument of the format.

Keyword Argument and `format()` Method

We can also insert text within curly braces along with numeric indexes. However, this text has to match keyword arguments passed to the `format()` method.

Example

```
>>> "I am {0} years old.I Love to work on {PC} Laptop".format(25,PC="APPLE")
'I am 25 years old.I Love to work on APPLE Laptop'
```

7.8.3 The `split()` Method

The `split()` method returns a list of all the words in a string. It is used to break up a string into smaller strings.

Example

Consider the following example where names of different programming languages such as C, C++, Java and Python is assigned to a variable Str1. Applying **split()** method on str1 returns the **list** of programming languages.

```
>>>Str1="C C++ JAVA Python"#Assigns names of Programming languages to Str1
>>>Str1.split()
['C,C++,JAVA,Python']
```


PROGRAM 7.5 Consider a input string that has a list of names of various multinational companies, such as TCS, INFOSYS, MICROSOFT, YAHOO and GOOGLE. Use split method and display the name of each company in a different line.

```
TOP_10_Company="TCS,INFOSYS,GOOGLE,MICROSOFT,YAHOO"
Company=TOP_10_Company.split(",")
print(Company)
for c  in Company:
   print(end="")
   print(c)
```

Output

```
['TCS', 'INFOSYS', 'GOOGLE', 'MICROSOFT', 'YAHOO']
TCS
INFOSYS
GOOGLE
MICROSOFT
YAHOO
```

Note: The `split()` method can be called without arguments. If it is called without a delimiter, then by default the space will act as a delimiter.

7.8.4 Testing String

A string may contain digits, alphabets or a combination of both of these. Thus, various methods are available to test if the entered string is a digit or alphabet or is alphanumeric. Methods to test the characters in a string are given in Table 7.1.

Table 7.1 The str class methods for testing its characters

Methods of str Class for Testing its Character	*Meaning*

`bool isalnum()`
Example:
`>>>S="Python Programming"`
`>>>S.isalnum()`
`False`
`>>> S="Python"`
`>>>S.isalnum()`
`True`
`>>> P="1Jhon"`
`>>>P.isalnum()`
`True`
	Returns True if characters in the string are alphanumeric and there is at least one character.

(Contd.)

bool isalpha()
Example:
`>>> S="Programming"`
`>>>S.isalpha()`
`True`
`>>> S="1Programming"`
`>>>S.isalpha()`
`False`
	Returns True if the characters in the string are alphabetic and there is at least one character.

bool isdigit()
Example:
`>>> Str1="1234"`
`>>> Str1.isdigit()`
`True`
`>>> Str2="123Go"`
`>>> Str2.isdigit()`
`False`
	Returns True if the characters in the string contain only digits.

bool islower()
Example:
`>>> S="hello"`
`>>>S.islower()`
`True`
	Returns True if all the characters in the string are in lowercase.

bool isupper()
Example:
`>>> S="HELLO"`
`>>>S.isupper ()`
`True`
	Returns True if all the characters in the string are in uppercase.

bool isspace()
Example:
`>>> S=" "`
`>>>S.isspace()`
`True`
`>>> Str1="Hello Welcome to Programming World"`
`>>> Str1.isspace ()`
`False`
	Returns true if the string contains only white space characters.

7.8.5 Searching Substring in a String

Table 7.2 contains methods provided by the str class to search the substring in a given string.

Table 7.2 Methods to search a substring in a given string

Methods of str Class for Searching the Substring in a Given String	*Meaning*

bool endswith(str Str1)
Example:
`>>> S="Python Programming"`
`>>>S.endswith("Programming")`
`True`
	Returns true if the string ends with the substring Str1.

bool startswith(str Str1)
Example:
`>>> S="Python Programming"`
`>>>S.startswith("Python")`
`True`
	Returns true if the string starts with the substring Str1.

int find(str Str1)
Example:
`>>> Str1="Python Programming"`
`>>> Str1.find("Prog")`
`7#Returns the index from where the string "Prog" begins`
`>>> Str1.find("Java")`
`-1#Returns -1 if the string "Java" is not found in the string str1`
	Returns the lowest index where the string Str1 starts in this string or returns -1 if the string Str1 is not found in this string.

int rfind(str Str1)
Example:
`>>> Str1="Python Programming"`
`>>> Str1.rfind("o")`
`9#Returns the index of last occurrence of string "o" in Str1`
	Returns the highest index where the string Str1 starts in this string or returns -1 if the string Str1 is not found in this string.

int count(str S1)
Example:
`>>> Str1="Good Morning"`
`>>> Str1.count("o")`
`3`
	Returns the number of occurrences of this substring.

7.8.6 Methods to Convert a String into Another String

A string may be present in lower case or upper case. The string in lower case can be converted into upper case and vice versa using various methods of the str class. Table 7.3 contains various methods to convert a string from one form to another.

Table 7.3 Methods to convert string from one form to another

Methods of Str Class to Convert a String from One Form to Another	*Meaning*

str capitalize()
Example:
`>>> Str1="hello"`
`>>> Str1.capitalize ()`
`'Hello' #Convert first alphabet of String Str1 to uppercase`
	Returns a copy of the string with only the first character capitalised.

str lower()
Example:
`>>> Str1="INDIA"`
`>>> Str1.lower ()`
`'india'`
	Returns a copy of the string with all the letters converted into lower case.

str upper()
Example:
`>>> Str1="iitbombay"`
`>>> Str1.upper()`
`'IITBOMBAY'`
	Returns a copy of the string with all the letters converted into upper case.

str title()
Example:
`>>> Str1="welcome to the world of programming"`
`>>> Str1.title()`
`'Welcome To The World Of Programming'`
	Returns a copy of the string with the first letter capitalised in each word of the string.

str swapcase()
Example:
`>>> Str1="IncreDible India"`
`>>> Str1.swapcase ()`
`'iNCREdIBLE iNDIA'`
	Returns a copy of the string which converts upper case characters into lower case characters and lower case characters into upper case characters.

str replace (str old, str new [,count])
Example:
`>>> S1="I have brought two chocolates, two cookies and two cakes"`
`#Replace the old string i.e "two" by new string i.e. "three".`
`>>> S2=S1.replace("two","three")`
`#Replace all occurrences of old string "two" by "three"`
`>>> S2`
`'I have brought three chocolates, three cookies and three cakes'`
	Returns a new string that replaces all the occurrences of the old string with a new string. The third parameter, i.e. the count is optional. It tells the number of old occurrences of the string to be replaced with new occurrences of the string.

(Contd.)

Q Replace two chocolates and two cookies by three chocolates and three cookies.

```
>>> S1="I have brought two chocolates, two cookies
and two cakes"
>>> S1.replace("two","three",2)
#Replace only first 2 occurrences of old string
"two" by "three"
'I have brought three chocolates, three cookies and
two cakes'
```

7.8.7 Stripping Unwanted Characters from a String

A common problem when parsing text is leftover characters at the beginning or end of a string. Python provides various methods to remove white space characters from the beginning, end or both the ends of a string.

> Note: Characters such as ', \f,\r, and \n are called white space characters.

Methods to strip leading and trailing white space characters are given in Table 7.4.

Table 7.4 Methods to strip leading and trailing white space characters

Methods of str Class for Stripping White Space Characters	*Meaning*

str lstrip()
Example:
`>>> Scentence1=" Hey Cool!!."`
`>>> Scentence1#Display Scentence1`
`' Hey Cool!!.'#Before Stripping left white space`
`>>> Scentence1.lstrip()#Remove left white space characters`
`'Hey Cool!!.'#After Stripping left white space characters`
Example:
`>>>Bad_Sentence=" \t\tHey Cool!!."`
`>>>Bad_Sentence#print Bad_Scentence before removing whitespace`
`' \t\tHey Cool!!.'`
`>>>Bad_Sentence.lstrip()#Print Bad_Sentence after removing`
`'Hey Cool!!.'`
	Returns a string with the leading white space characters removed.

str rstrip()
Example:
`>>> Scentence1="Welcome!!!\n\n\ "`
`>>> Scentence1.rstrip()#Remove trailing white space character`
`'Welcome!!!\n\n\\'#After Removing white space character`
	Returns a string with the trailing white space characters removed.

(Contd.)

str strip() Returns a string with the leading and trailing white space characters removed.

Example:

```
>>> Str1=" Hey,How are you!!!\t\t\t "
>>> Str1#Print string str1 before stripping
' Hey,How are you!!!\t\t\t '
>>> Str1.strip()#Print after Stripping
'Hey,How are you!!!'
Example:
>>> s1="@Cost Prize of Apple Laptop is at Rs = 20 Dollars $$$$"
>>> s1#Before removing unwanted characters @ and $
'@Cost Prize of Apple Laptop is at Rs = 20 Dollars $$$$'
>>> s1.strip('@$')
'Cost Prize of Apple Laptop is at Rs = 20 Dollars '#After Removing
```

Note: Stripping does not apply to any text in the middle of a string. It only strips the white space characters from the beginning and end of a string.

Example

```
>>> S1="Python Programming"
>>> S1#Print S1 before stripping
'Python Programming'
>>> S1.strip()
'Python Programming'#Print S1 after stripping
```

In the above example, there are multiple spaces between the two string "Python" and "Programming". Even though after applying `strip()` method on S1, The string S1 remain unchanged. The white space characters are not removed from the string S1.

7.8.8 Formatting String

Table 7.5 Methods to format a string

Methods of str Class for Formatting Characters	*Meaning*

str center(int width)
Example
`>>> S1="APPLE MACOS"`
#Place the string S1 in the center of a string with 11 characters
`>>> S1.center(15)`
`'APPLE MACOS'`
	Returns a copy of the string centered in a field of the given width.

(Contd.)

str ljust(int width) Returns a string left justified in a field of the given width.

Example:

```
>>> S1="APPLE MACOS"
#Place the string S1 at the left of a string with 15 characters.
>>> S1.ljust(15)
'APPLE MACOS'
```

str rjust(int width) Returns a string right justified in a field of the given width.

Example:

```
>>> S1="APPLE MACOS"
#Place the string S1 at the right of a string with 15 characters.
>>> S1.rjust(15)
'APPLE MACOS'
```

7.8.9 Some Programs on String

PROGRAM 7.6 Write the function countB(word) which takes a word as the argument and returns the number of 'b' in that word.

```
def countB(word):
 print(word)
 count = 0
 for b in word:
    if (b == 'b'):
        count = count + 1
 return count
print(" Number of  'b' = ",countB("abbbabbaaa"))
```

Output

```
abbbabbaaa
Number of 'b' =  5
```

PROGRAM 7.7 Write the function count_Letter(word, letter) which takes a word and a letter as arguments and returns the number of occurrences of that letter in the word.

```
def count_Letter(word,letter):
 print("Word = ",word)
 print("Letter to count = ",letter)
 print("Number of occurrences of '",letter,"' is =",end="")
 count = 0
for i in word:
```

(Contd.)

```
    if (i == letter):
        count = count + 1
 return count
x=count_Letter('INIDA','I')
print(x)
```

Output

```
Word = INIDA
Letter to count = I
Number of occurrences of ' I ' is =2
```

PROGRAM 7.8 Write the function modify_Case(word) which changes the case of all the letters in a word and returns the new word.

```
def modify_Case(word):
    print("Original String = ",word)
    print("After Swapping String  = ",end="")
    return word.swapcase()
print(modify_Case("hi Python is intresting, isn't it ? "))
```

Output

```
Original String =  hi Python is intresting, isn't it ?
After Swapping String  = HI PYTHON IS INTRESTING, ISN'T IT ?
```

PROGRAM 7.9 A string contains a sequence of characters. Elements within a string can be accessed using an index which starts from 0. Write the function getChar(word, pos) which takes a word and a number as arguments and returns the character at that position.

```
def getChar(word,pos):
    print("Word = ",word)
    print("Character at Position  ",pos," = ",end="")
    counter = 0
    for i in word:
        counter = counter + 1
        if (counter == pos):
            return i
print(getChar("Addicted to Python ",3))
```

Output

```
Word = Addicted to Python
Character at Position 3 = i
```


PROGRAM 7.10 Write a function Eliminate_Letter(word, letter) which takes a word and a letter as arguments and removes all the occurrences of that particular letter from the word. The function will return the remaining letters in the word.

```
def Eliminate_Letter(word,Letter):
    print("String = " ,word)
    print("After Removing Letter : ",Letter)
    print("String = ",end="")
    newstr = ''
    newstr = word.replace(Letter,"")
    return newstr
#Sample test
x = Eliminate_Letter(' PYTHON PROGRAMMING','P')
print(x)
```

Output

```
String =    PYTHON PROGRAMMING
After Removing Letter:  P
String = PYTHON ROGRAMMING
```

PROGRAM 7.11 Write the function countVowels(word) which takes a word as an argument and returns the vowels ('a', 'e', 'i', 'o', 'u') in that word.

```
def countVowels(word):
    print(" Word = ",word)
    word  = word.lower()
    return {v:word.count(v) for v in 'aeiou'}
print(countVowels("I Love Python Programming"))
```

Output

```
Word =  I Love Python Programming
{'u': 0, 'i': 2, 'o': 3, 'e': 1, 'a': 1}
```

PROGRAM 7.12 Write the function UpperCaseVowels(word) which returns the word with all the vowels capitalised.

```
def UpperCaseVowels(word):
    new= ''
    print("string = ",word)
    print(" After Capitializing Vowels")
    print("String = ",end="")
    for i in word:
```

(Contd.)

```
        if(i == 'a' or i == 'e' or i == 'i' or i == 'o' or i == 'u' ):
            new = new + i.upper()
        else:
            new = new + i
    return new
#Sample run
x = UpperCaseVowels('aehsdfiou')
print(x)
```

Output

```
string =  aehsdfiou
 After Capitializing Vowels
String = AEhsdfIOU
```

PROGRAM 7.13 Write the function replacevowels(word) which removes all the vowels ('a', 'e', 'i', 'o', 'u') in a word and returns the remaining letters in the word.

```
def removeVowels(word):
    new = ''
    print("String =",word)
    print("String After Removing Vowels =",end="")
    for i in word:
        if(i!= 'a' and i!= 'e' and i!= 'i' and i!= 'o' and i!= 'u'):
            new = new + i
    return new
#Sample run
x = removeVowels('abceiodeuf')
print(x)
```

Output

```
String = abceiodeuf
String After Removing Vowels =bcdf
```

PROGRAM 7.14 Write the function isReverse(word1, word2) which takes two words as arguments and returns True if the second word is the reverse of the first word.

```
def isReverse(word1,word2):
    print("First Word = ",word1)
    print("Second Word = ",word2)
    if(word1 == word2[::-1]):
        return True
```

(Contd.)


```
    else:
        return False
x = isReverse('Hello','olleH')
print(x)print(x)
```

Output

```
First Word =  Hello
Second Word =  olleH
True
```

PROGRAM 7.15 Write a function mirrorText(word1, word2) which takes two words as arguments and returns a new word in the following order: word1word2word2word1.

```
def mirrorText(word1, word2):
    print("String1 = ",word1)
    print("String2 = ",word2)
    print("Mirror String = ",end="")
    return word1+word2+word2+word1
x = mirrorText('PYTHON','STRONG')
print(x)
```

Output

```
String1 =  PYTHON
String2 =  STRONG
Mirror String = PYTHONSTRONGSTRONGPYTHON
```

MINI PROJECT Conversion of HexDecimal Number into its Equivalent Binary Number

Table 7.6. contains conversion of a hexadecimal number into its equivalent binary number.

Table 7.6 Hexadecimal into equivalent binary form

Hexadecimal Number	*Equivalent Binary Number*	*Equivalent Decimal Number*
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6

(Contd.)

7	0111	7
8	1000	8
9	1001	9
'A'	1010	10
'B'	1011	11
'C'	1100	12
'D'	1101	13
'E'	1110	14
'F'	1111	15

Program Statement

Write a program to convert a hexadecimal number entered as a string into its equivalent binary format.

Note: Use `ord()` to obtain the ASCII value of a character.

Sample Input

```
Please Enter Hexadecimal Number: 12FD
```

Output

```
Equivalent Binary Number is
0001 0010 1111 1101
```

Algorithm

- **STEP 1:** Read the hexadecimal number as string from the user.
- **STEP 2:** Pass **'h'**, i.e. the number as string to function named **'hex_to_bin(h)'**
- **STEP 3:** Inside **function hex_to_bin(h)**, traverse each character of string **'h'**.

  Check if the character inside the string contains values in between **'A'** and **'F'**. Then add 10 to the difference between ASCII values, i.e. (ord('ch') – ord('A')) + 10 and pass the obtained sum 'X' as string to function **dec_bin(X).**
- **STEP 4:** Calculate the equivalent binary number of x and print the same.

```
def dec_bin(x):    #Decimal to Binary
      k=[]
      n=x
      while (n>0):
            a=int(float(n%2))
            k.append(a)
```

(Contd.)


```
            n=(n-a)/2
      k.append(0)
      string=""
      for j in k[::-1]:
            string=string+str(j)
      if len(string)>4:
            print(string[1:],end=' ')
      elif len(string)>3:
            print(string,end=' ')
      elif len(string)>2:
            print('0'+string,end=' ')
      else:
            print('00'+string,end=' ')


def hex_to_bin(h): #Hexdeciaml Number 'h' passed to function
      print('',end='')
      for ch in range(len(h)):
            ch = h[ch]
            if 'A' <= ch <='F':
                  dn = 10 + (ord(ch)-ord('A'))
                  dec_bin(dn)
            else:
                  dn = (ord(ch)-ord('0'))
                  dec_bin(dn)

n=input('Please Enter Hexadecimal Number:')
print('Equivalent Binary Number is as follows:')
hex_to_bin(n)
```

Output

```
Please Enter Hexadecimal Number:12FD
Equivalent Binary Number is as follows:
0001 0010 1111 1101
```

In the above program, initially the number as string is read from the user and passed to the function **hex_to_bin()**. The function traverses all the characters. For each character it converts into its equivalent decimal form 'X', the equivalent decimal number 'X' is passed to the function **dec_bin(X),** to calculate the binary of a number. Thus, finally we obtain an equivalent binary number for a given hexadecimal number.

SUMMARY

- String is the object of the str class.
- String object is immutable.
- The `index[]` operator is used to access individual characters in a string.
- You can use the for loop and the while loop to traverse the contents of a string.
- Various string methods can be used to manipulate strings and perform various operations such as conversion of lower to uppercase, reversal, concatenation, comparison, search and replacement of string elements.

KEY TERMS

- **The `index[]` Operator:** Access character
- **The +, * and in Operator:** Concatenate, repetition, check characters in a string
- **Slicing str[start: end] Operation:** Obtain substring
- **Comparison Operators:** ==,!= , >=, <=
- **Immutable Strings:** Cannot change the existing string
- **The `split()` Method:** Returns a list of words
- **The `format()` Method:** Format string, i.e. left justify, right justify or center
- **Testing String:** Check if the string contains digits, numbers or alphanumeric characters.

REVIEW QUESTIONS

A. Multiple Choice Questions

1. What will be the output of the following program?

```
S1="Welcome to JAVA Programming"
S2=S1.replace("JAVA","Python")
print(S1)
print(S2)
```

   a. Welcome to JAVA Programming
   b. Welcome to Python Programming
   c. Welcome to Java Python Programming
   d. None of the above

2. What will be the output of the following program?

```
Str1 = "Hello"
Str2=Str1[:-1]
print(Str2)
```

   a. olle
   b. Hello
   c. el
   d. Hell

3. What will be the output of the following program?

```
Str1= "The Sum of {0:b} and {1:b} is {2:b}".format(2,2,4)
print(Str1)
```