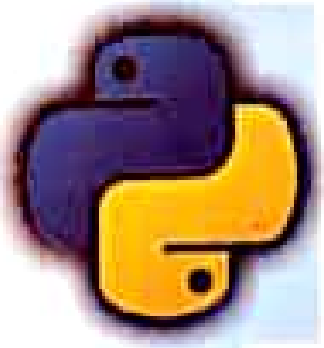
COMPUTER SCIENCE *with* python



Textbook for Class XI

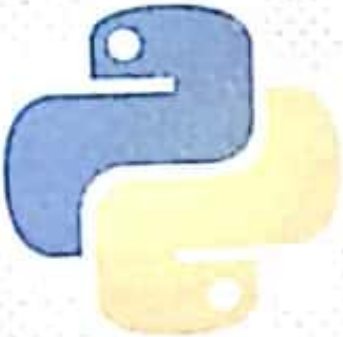
SUMITA ARORA

- Programming & Computational Thinking
- Computer System & Organization
- Data Management
- Society, Law and Ethics

DHANPAT RAI & Co.

COMPUTER SCIENCE with PYTHON

[Textbook XI]



Sumita Arora

DHANPAT RAI & CO. (Pvt.) Ltd.
EDUCATIONAL & TECHNICAL PUBLISHERS

Preface

Quality of education depends a lot on the curriculum imparted. Since *Computer Science* is a rapidly evolving field, CBSE constituted a committee to look into existing Computer Science curriculum and recommend a curriculum that is modern, relatively light, teaches timeless concepts like computational thinking, is upto international standards and focuses on problem solving skills than just the syntax.

The new syllabus of Computer Science (083) is the outcome of the recommendations by the curriculum review committee. The objectives of this new syllabus are laudable — focus on clear understanding of concepts, applications of concepts, problem solving skills, develop computational thinking etc.

This book has been written keeping all this in mind. This book adheres to the CBSE curriculumn for Computer Science (083) for Class XI. Based of the syllabus, the book has been divided into four units.

Unit 1 : Programming and Computational Thinking (PCT-1)

Programming and Computational Thinking unit has been divided in 11 chapters, **Chapters 1 to 11**. This unit covers *basics of Python, fundamentals, conditional and iterative constructs, strings, lists, tuples, dictionaries, sorting* and *program debugging techniques*. The aim of this unit is to lay foundation of problem solving skills through Python programming language.

Unit 2 : Computer Systems and Organisation (CSO)

Computer Systems and Organisation unit has been divided in four chapters, **Chapters 12 to 15**. This unit coves *basic computer organisation, Boolean logic, data representation* and *insight into program execution*. The aim of this unit is to give an idea of how the information is represented, organised and processed inside a computer.

Unit 3 : Data Management (DM-1)

Data Management unit has been divided into five chapters, **Chapters 16 to 20**. This unit talks about *relational database concepts, SQL basics, table creation and manipulation, querying, table joins and indexes in SQL*. Then this unit talks about NoSQL database and covers basics of a NoSQL database *MongoDB*. The aim of this unit is to provide idea of how data is processed in databases of modern age.

Unit 4 : Society, Law and Ethics (SLE-1)

Society, Law and Ethics unit has been divided into two chapters, **Chapters 21 to 22**. This unit covers *Cyber safety basics* and *guidelines, guidelines and usage rules for using social networks, threats* while accessing websites and how to *prevent/counter these* along with how to securely communicate data. The aim of this unit is to know the challenges in safe and secure use of cyber facilities and technologies to counter these.

Apart from the text book, we have also provided a practical book *'Progress In'* that contains additional practice exercises. Although the text book contains sufficient number of practice questions and exercises, yet the additional practice of exercises given in practical book will make the foundation of programming and other concepts even stronger.

Contents

1

Getting Started with Python

In This Chapter

1.1 INTRODUCTION

The word *Python* – isn't it scary ? Does it bring the image of big reptile that we prefer to see either in jungles or zoo ? Well, it's time to change the image. Now on, you'll remember word *Python* for its playfulness and pleasant productivity. Confused ? Well, Don't be – because, now on you'll get introduced to a new programming language namely 'Python', which promises to make you a big programming fan :-).

Python programming language was developed by Guido Van Rossum in February 1991. Python is based on or influenced with *two* programming languages :

- *ABC language*, a teaching language created as a replacement of BASIC, and
- *Modula*-3

Python is an easy-to-learn yet powerful object oriented programming language. It is a very high level programming language yet as powerful as many other middle-level not so high-level languages like C, C++, Java etc.

In this chapter, we shall introduce you to playful world of *Piquant Python* [Word 'Piquant' means pleasantly stimulating or exciting to the mind]. So, are we ready ? And... here we go.

NOTE

Do you know Python, the programming language, was named after famous *BBC* comedy show namely *Monty Python's Flying Circus.*

1.2 PYTHON – PLUSES

Pluses of Python

- Easy to Use OO Language
- Expressive Language
- Interpreted Language
- Its Completeness
- Cross-platform Language
- Free and Open Source
- Variety of Usage / Applications

Though Python language came into being in early 1990's, yet it is competing with ever-popular languages such as C, C++, Java etc. in popularity index. Although, it is not perfect for every type of application, yet it has many strengths that make it a good choice for many situations. Let's see what are these *pluses of Python*.

1. Easy to Use

Python is compact and very easy to use *object oriented language* with very simple syntax rules. **It is a very high level language and thus very-very programmer-friendly.**

2. Expressive Language

Python's expressiveness means it is more capable to expressing the code's purpose than many other languages. Reason being – fewer lines of code, simpler syntax.

For example, consider following *two* sets of codes :

```
// In C++ : Swap Values
    int a = 2, b = 3, tmp ;
    tmp = a ;
    a = b ;
    b = tmp ;
```

```
# In Python : Swap values
    a, b = 2, 3
    a, b = b, a
```

which one is compact and easier to understand ? Need I say more ? :). This is the simplest example, you'll find many more such examples of Python's expressiveness in the due course.

3. Interpreted Language

Python is an interpreted language, not a compiled language. This means that the Python installation interprets and executes the code line by line at a time. It makes Python an easy-to-debug language and thus suitable for beginners to advanced users.

4. Its Completeness

When you install Python, you get everything you need to do real work. You do not need to download and install additional libraries ; all types of required functionality is available through various modules of Python standard library[1]. For example, for diverse functionality such as emails, web-pages, databases, GUI development, network connections and many more, everything is available in Python standard library. Thus, it is also called – Python follows "Batteries Included" philosophy.

5. Cross-platform Language

Python can run equally well on variety of platforms – Windows, Linux/UNIX, Macintosh, supercomputers, smart phones etc.[2] Isn't that amazing ? And that makes Python a true cross-platform language. Or in other words, Python is a **portable language.**

1. If you install Python through *Anaconda Python Distribution* it loads most libraries and packages with Python.
2. Python even has versions that run on different languages such as Java (Jython), .NET (IronPython) etc.

6. Free and Open Source

Python language is freely available *i.e.*, without any cost (from www.python.org). And not only is it free, its source-code (*i.e.*, complete program instructions) is also available, *i.e.*, it is open-source also. Do you know, you can modify, improve/extend an open-source software !

7. Variety of Usage/Applications

Python has evolved into a powerful, complete and useful language over these years. These days Python is being used in many diverse fields/applications, some of which are :

- Scripting
- Web Applications
- Game development
- System Administrations
- Rapid Prototyping
- GUI Programs
- Database Applications

1.3 PYTHON – SOME MINUSES (SO HUMAN LIKE)

Minuses of Python

- Not the Fastest Language
- Lesser Libraries than C, Java, Perl
- Not Strong on Type-binding
- Not Easily Convertible

Although Python is very powerful yet simple language with so many advantages, it is *not the Perfect Programming language*. There are some areas where Python does not offer much or is not that capable. Let's see what these are :

1. Not the Fastest Language

Python is an interpreted language not a fully compiled one. Python is first semi-compiled into an internal byte-code, which is then exerted by a Python interpreter. Fully compiled languages are faster than their interpreted counterparts. So, here Python is little weaker though it offers faster development times but execution-times are not that fast compared to some compiled languages.

2. Lesser Libraries than C, Java, Perl

Python offers library support for almost all computing programs, but its library is still not competent with languages like C, Java, Perl as they have larger collections available. Sometimes in some cases, these languages offer better and multiple solutions than Python.

3. Not Strong on Type-binding

Python interpreter is not very strong on catching 'Type-mismatch' issues. For example, if you declare a variable as integer but later store a string value in it, Python won't complain or pin-point it.

4. Not Easily Convertible

Because of its lack of syntax, Python is an easy language to program in. But this advantage has a flip-side too : it becomes a disadvantage when it comes to translating a program into another programming language. This is because most other languages have structured defined syntax.

Since most other programming languages have strong-syntax, the translation from Python to another language would require the user to carefully examine the Python code and its structure and then implement the same structure into other programming language's syntax.

So, now you are familiar with what all Python offers. As a free and open-source language, its users are growing by leaps and bounds.

As per February 2013 popularity index, Python was 4th most popular programming language[3] after – *Java, PHP* and *C#*. That is the reason, it's part of your syllabus. Together we'll make it playful Python ;).

Since Python is an interpreted language and not a compiled language, it would be a good idea to know the difference between working of an interpreter and a compiler (as explained in following information box).

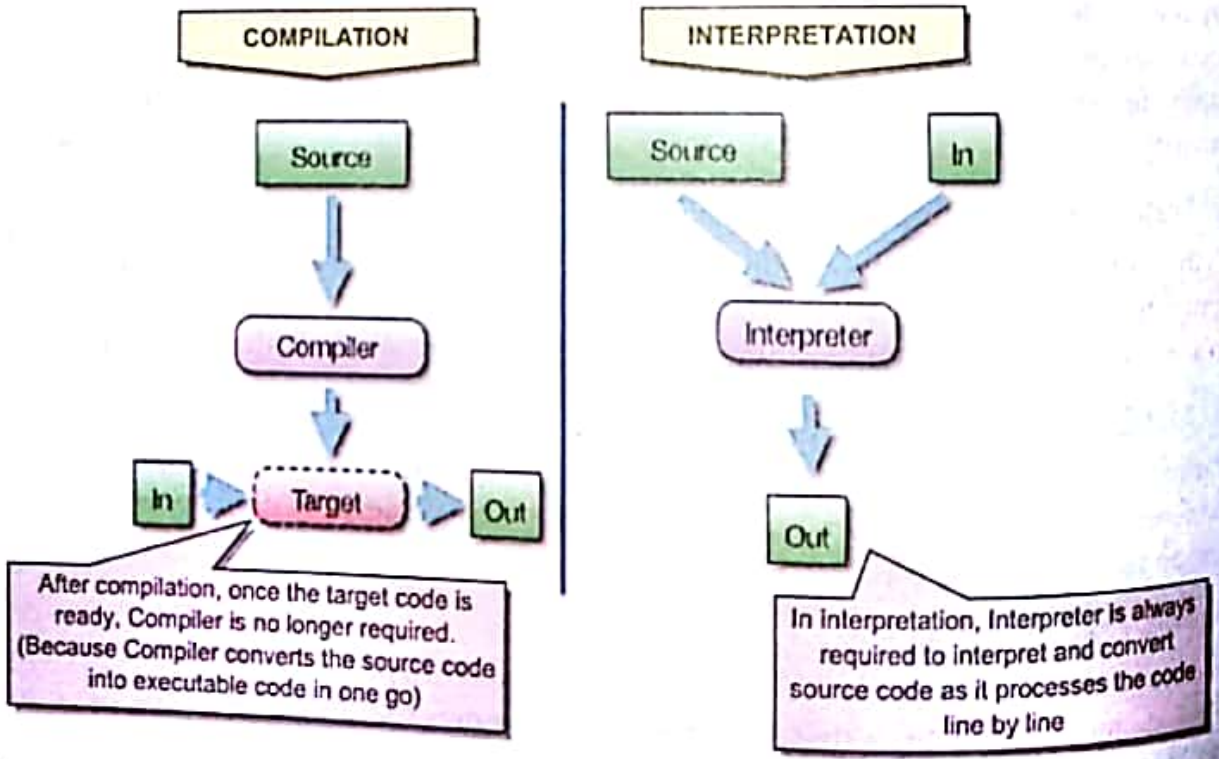
Interpretation vs Compilation

Interpreter

This language processor converts a HLL(High Level Language) program into machine language by *converting and executing it line by line*. If there is any error in any line, it reports it at the same time and program execution cannot resume until the error is rectified. Interpreter must always be present in the memory every time the program is executed as every time the program is run, it is first interpreted and then executed. For error debugging, interpreter is very much useful as it reports the error(s) at the same time. But once errors are removed, unnecessary usage of memory takes place as it has to be present in the memory always.

Compiler

It also converts the HLL program into machine language but the conversion manner is different. It converts the *entire HLL program in one go*, and reports all the errors of the program along with the line numbers. After all the errors are removed, the program is recompiled, and after that the compiler is not needed in the memory as the object program is available.



Python is an interpreted language, that is, all the commands you write are interpreted and executed one by one.

3. Source : https://sites.google.com/site/pydatalog/pypl/PyPL-PopularitY-of-Programming-Language.

1.4 WORKING IN PYTHON

Before you start working in Python, you need to install Python on your computers. There are multiple Python distributions available today.

- Default installation available from www.python.org is called **CPython installation** and comes with *Python interpreter, Python IDLE (Python GUI)* and *Pip (package installer)*.
- There are many other Python distributions available these days. **Anaconda Python distribution** is one such highly recommended distribution that comes preloaded with many packages and libraries (*e.g.*, NumPy, SciPy, Panda libraries etc.).
- Many popular IDEs are also available *e.g.*, Spyder IDE, PytCharm IDE etc. Of these, Spyder IDE is already available as a part of Anaconda Python distribution.

To install any of these distributions, PLEASE REFER TO APPENDIX A. We shall learn to work with both these distribution types [but my personal favourite is *Anaconda* ;) – not the reptile, the Python distribution :)]

Once you have Python installed on your computers, you are ready to work on it. You can work in Python in following different ways :

(*i*) in Interactive mode (*also called Immediate Mode*) (*ii*) in Script mode

1.4.1 Working in Default CPython Distribution

The default distribution, CPython, comes with **Python interpreter, Python IDLE (GUI based)** and **pip** (package installer). To work in *interactive* as well as *script* mode, you need to open **Python IDLE.**

1.4.1A Working in Interactive Mode (Python IDLE)

Interactive mode of working means you type the command – one command at a time, and the Python executes the given command there and then and gives you output. In interactive mode, you type the command in front of Python command prompt >>> . *For example,* if you type **2 + 5** in front of Python prompt, it will give you result as 7 :

```
>>> 2 + 5   <-- command/expression given here
7            <-- Result returned by Python
```

To work in interactive mode, follow the process given below :

(*i*) Click **Start button → All Programs → Python 3.6.x → IDLE (Python GUI)** [see Fig. 1.1(*a*)]

Or

Click **Start button → all Programs → Python 3.6.x → Python** (command line)

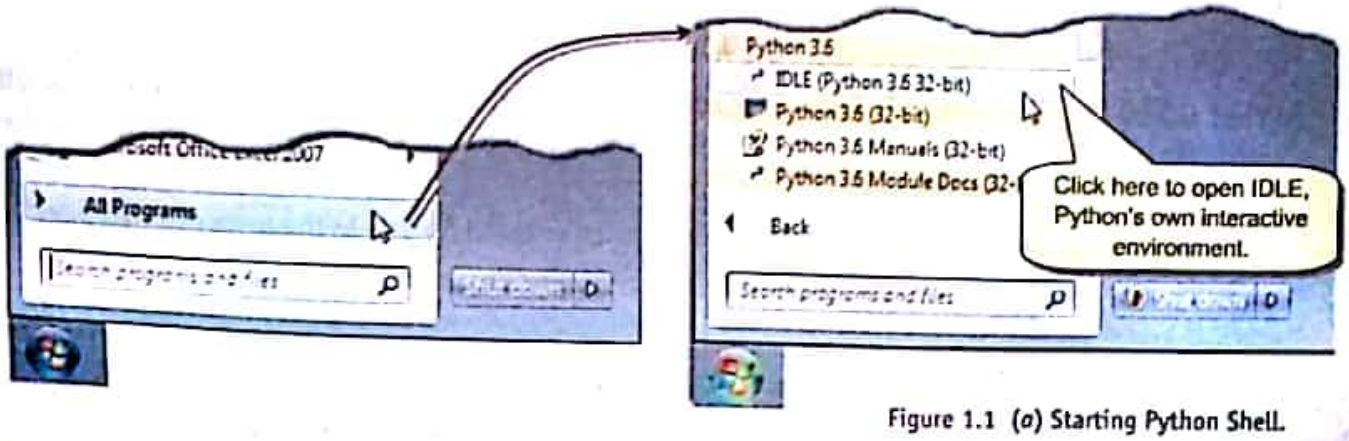


Figure 1.1 (*a*) Starting Python Shell.

(*ii*) It will open Python Shell [see Fig. 1.1(*b*)] where you'll see the Python prompt (three '>' signs *i.e.*, >>>).

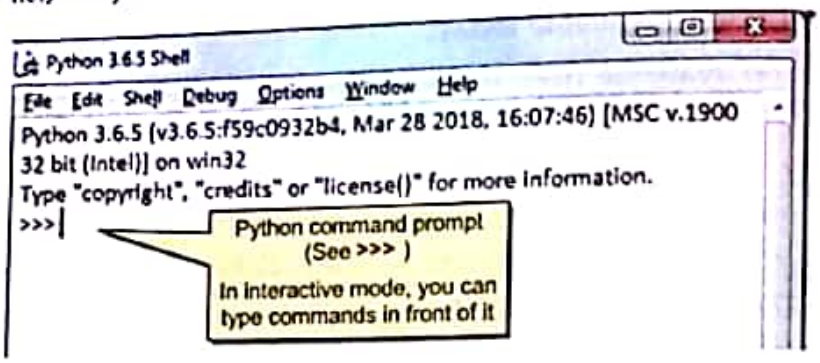


Figure 1.1 (*b*) Python's interactive interpreter – Python Shell.

NOTE

The interactive *interpreter of Python* is also called **Python Shell.**

(*iii*) Type commands in front of this Python prompt and Python will immediately give you the result. [*see* Fig. 1.1(*c*)]

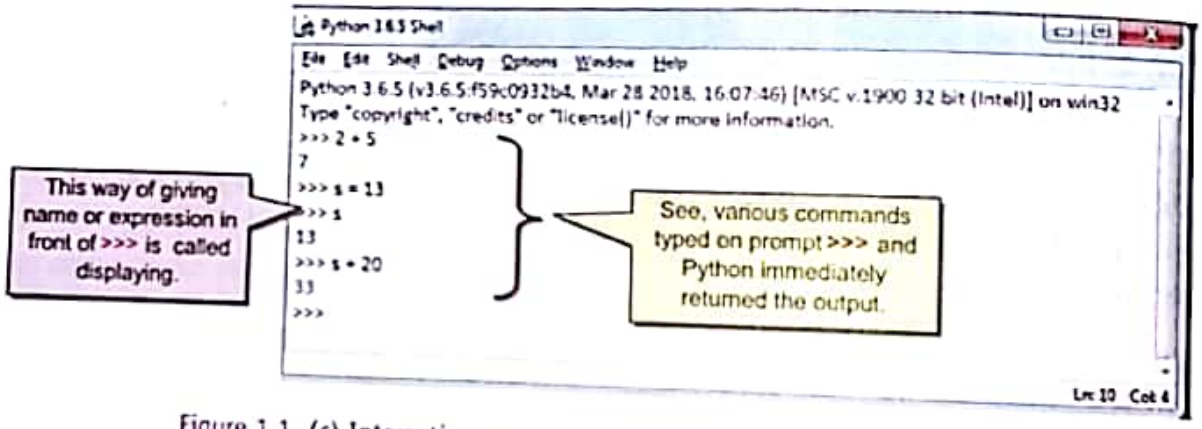


Figure 1.1 (*c*) Interactive commands and their output in Python Shell.

For example, to print string "Hello" on the screen, you need to type the following in front of Python prompt (>>>)

```
>>> print ("Hello") ↵
```

And Python interpreter will immediately display string Hello below the command. To display, you just need to mention name or expression [Fig. 1.1(*c*)] in front of the prompt.

NOTE

Interactive mode proves very useful for testing code; you type the commands one by one and get the result or error one by one.

Figure 1.1(*c*) shows you some sample commands that we typed in Python shell and the output returned by Python interpreter.

1.4.1B Working in Script Mode (Python IDLE)

What if you want to save all the commands in the form of program file and want to see all output lines together rather than sandwiched between successive commands ? With interactive mode, you cannot do so, for :

- Interactive mode does not save the commands entered by you in the form of a program[4].
- The output is sandwiched between the command lines [see Fig. 1.1(*c*)].

4. Python GUI Shell IDLE lets you save the entire session (commands followed by their results – as it appears on screen) but that is not the Python program/script containing only the instructions.

The solution to above problems is the **Script mode**. To work in a script mode, you need to do the following :

Step 1 : Create Module / Script / Program File

Firstly, you have to create and save a module / Script / Program file.

To do so, follow these instructions :

(*i*) Click **Start button → All Programs → Python 3.6.x → IDLE**. [Fig. 1.2(*a*)]

(*ii*) Click **File → New** in IDLE Python Shell. [Fig. 1.2(*a*)]

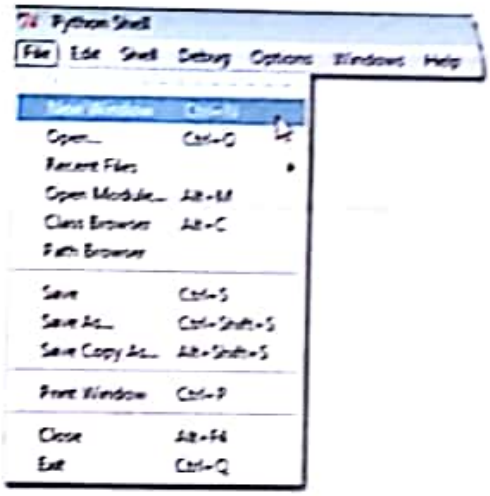
(*iii*) In the New window that opens, type the commands you want to save in the form of a program (or script). [Fig. 1.2(*b*)]

For instance, for the simple Hello World program, you'll need to type following line :

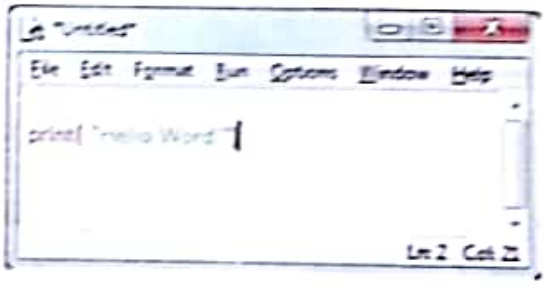
```
print ("Hello World !")
```

NOTE

You can display as well as print values in interactive mode, but for script mode, print( ) command is preferably used to print results.



(*a*) File → New command in Python Shell



(*b*) Type commands in new blank file (script mode)

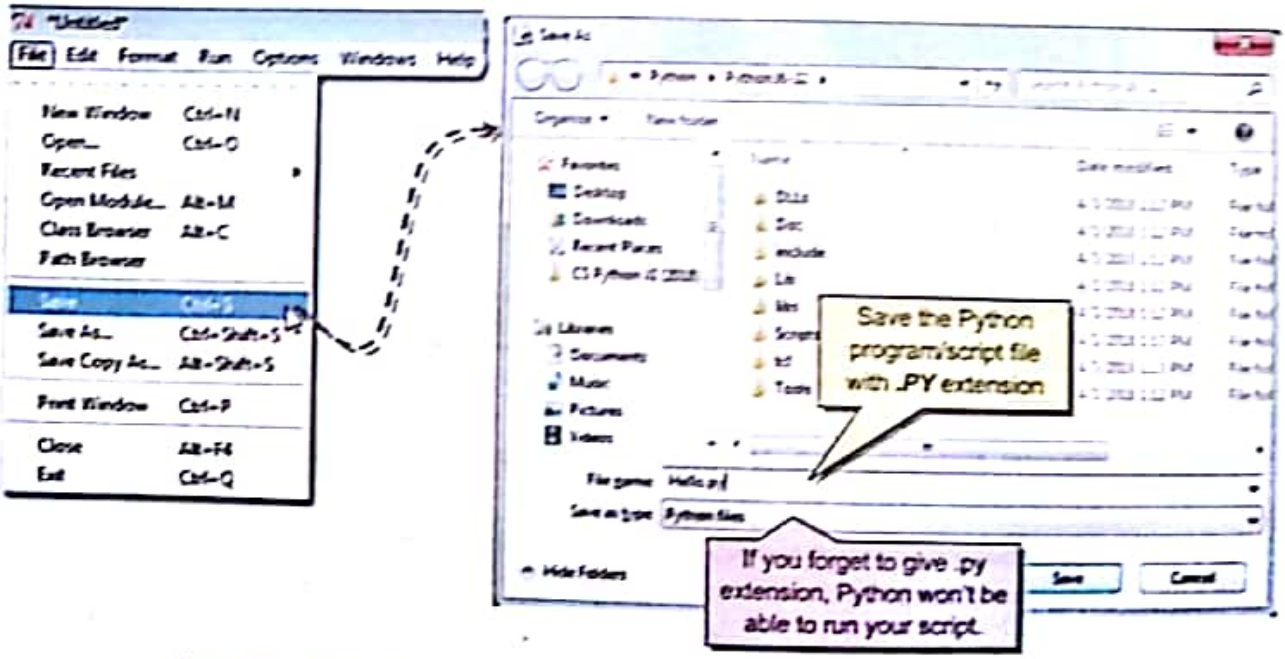


Figure 1.2 (*c*) Save file with .py extension with File → Save command (Script mode).

(*iv*) Click **File → Save** and then save the file with an extension **.py**. The Python programs has .py extension [Fig. 1.2(*c*)]. For instance, we gave the name to our program as **Hello.py**.

Now your program would be saved on the disk and the saved file will have **.py** extension.

Step 2 : Run Module / Script / Program File

After the program/script file is created, you can run it by following the given instructions :

(*i*) Open the desired program/script file that you created in previous Step 1 by using IDLE's **File → Open** command.
If the program / script file is already open, you can directly move to next instruction.

(*ii*) Click **Run → Run Module** command [Fig. 1.3(*a*)] in the open program / script file's window.
You may also press **F5 key**.

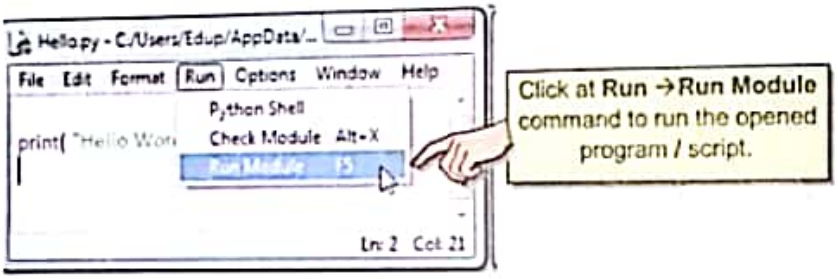


Figure 1.3 (*a*) Run → Run Module command (Script mode)

(*iii*) And it will execute all the commands stored in module / program / script that you had opened and show you the complete output in a separate Python Shell window. [Fig. 1.3(*b*)]

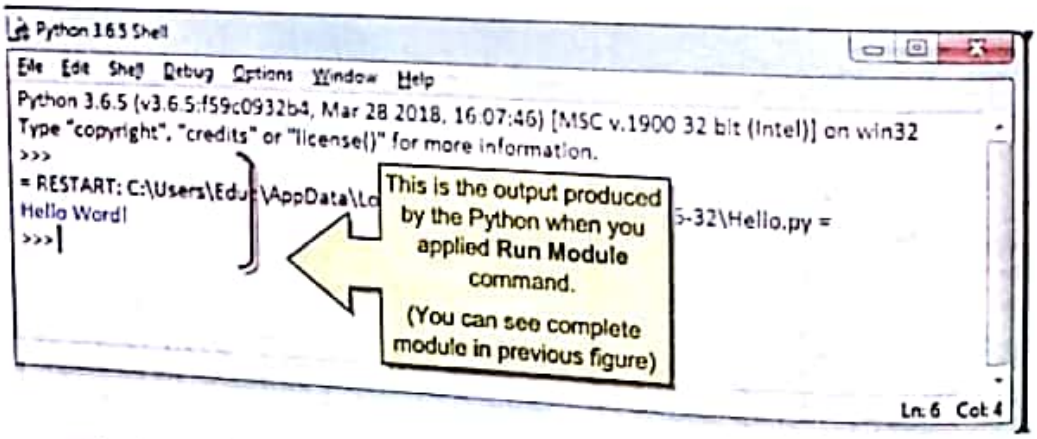


Figure 1.3 (*b*) Output of a module-run is shown in the shell window.

As you can see that with script mode, you can store all commands together in the form of a module / program / script and can get all output lines together. (No more command-output sandwiching :).

1.4.2 Working in Anaconda Distribution

Anaconda distribution comes with many preloaded packages and libraries. You can work in it in both interactive and script modes. Anaconda distribution provides the following tools that you can use to work in Python.

- **Jupyter notebook.** It is a web based, interactive computing environment.
- **Spyder.** It is a powerful Python IDE with many useful editing, interactive testing and debugging features.

Let us learn to work with both.

1.4.2A Working in Jupyter Notebook

In order to work in *jupyter notebook*, you need to first launch it using **Anaconda Navigator**[5] as it has come preloaded with Anaconda distribution.

1. Launch Anaconda Navigator

2. From the Navigator window, click on Launch below **jupyter notebook** tile.

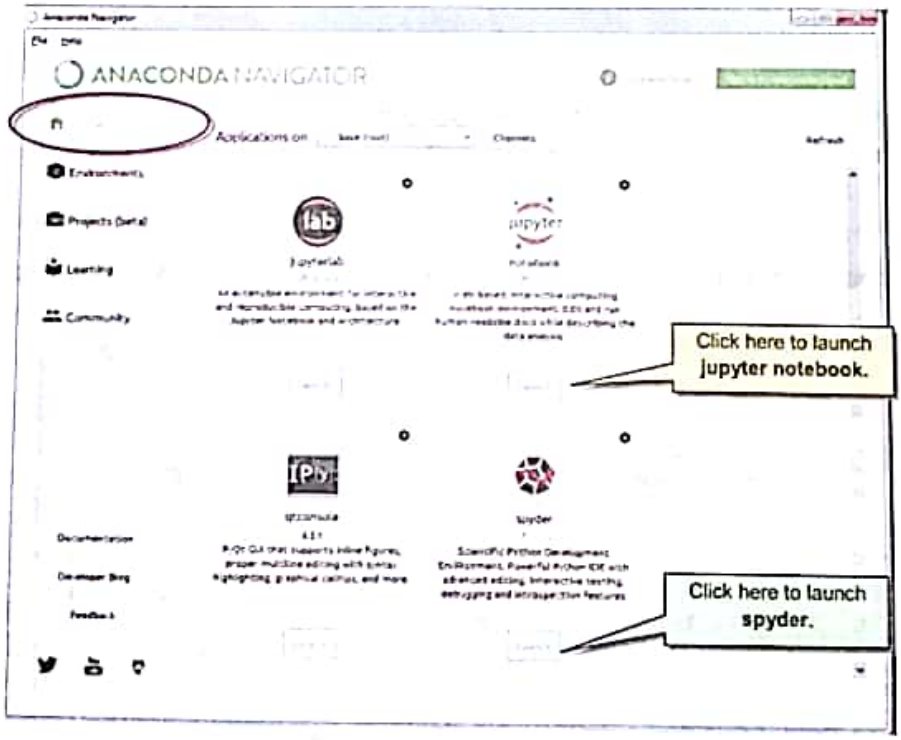


Figure 1.4 Launching applications from Anaconda Navigator.

5. Please note that jupyter notebook can also be installed separately, without Anaconda also.

3. Since jupyter notebook is a web based computing environment, it will be launched in a web browser.
Your web browser will now show you notebook dashboard (see below)

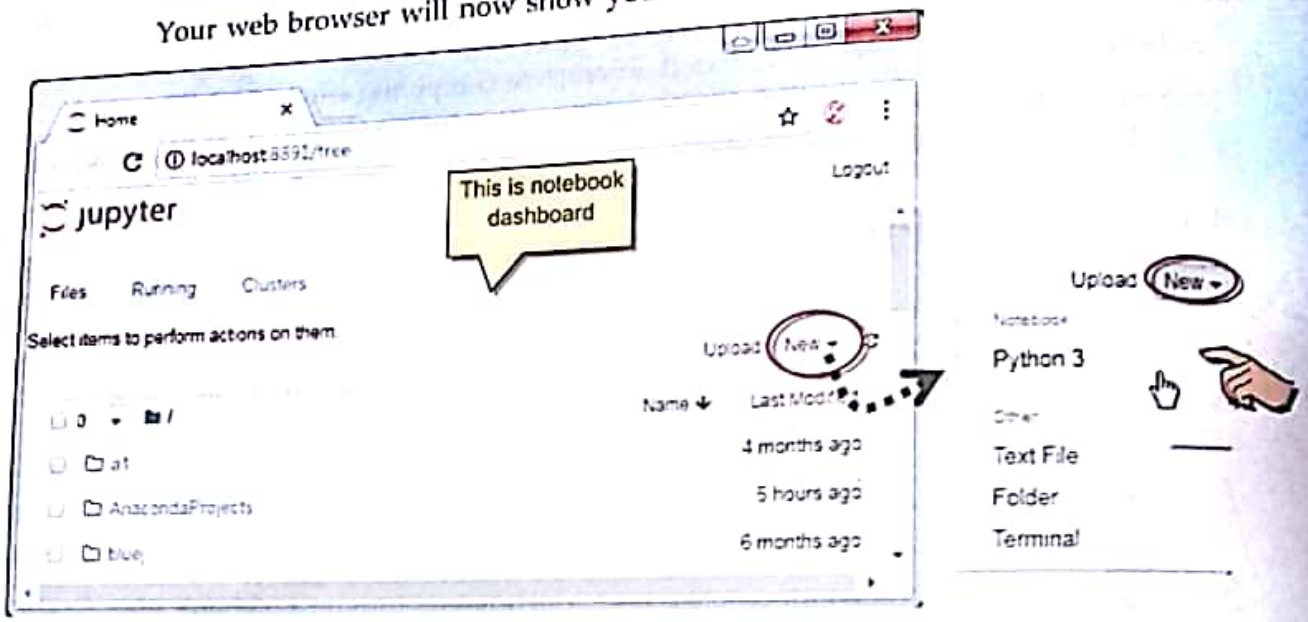


Figure 1.5

4. On the notebook dashboard, click at down-arrow next to New button and select Python 3 to create a notebook for executing Python 3.x code. (see figure above).

5. In a new tab, it will open a new notebook where you can write and run your code.

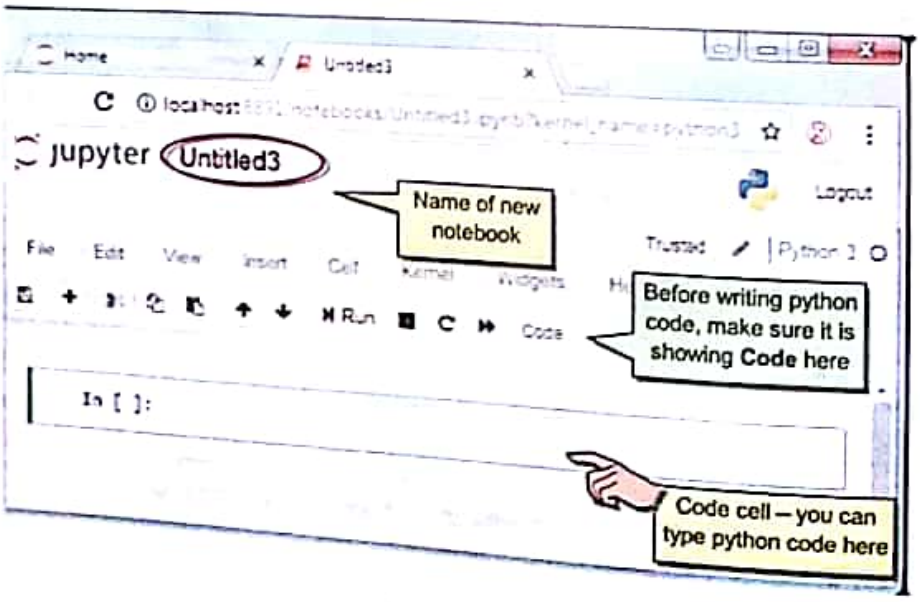


Figure 1.6

Interactive Mode

6. To run Python code in interactive mode, type code in code cell and click **Run.** (see below)

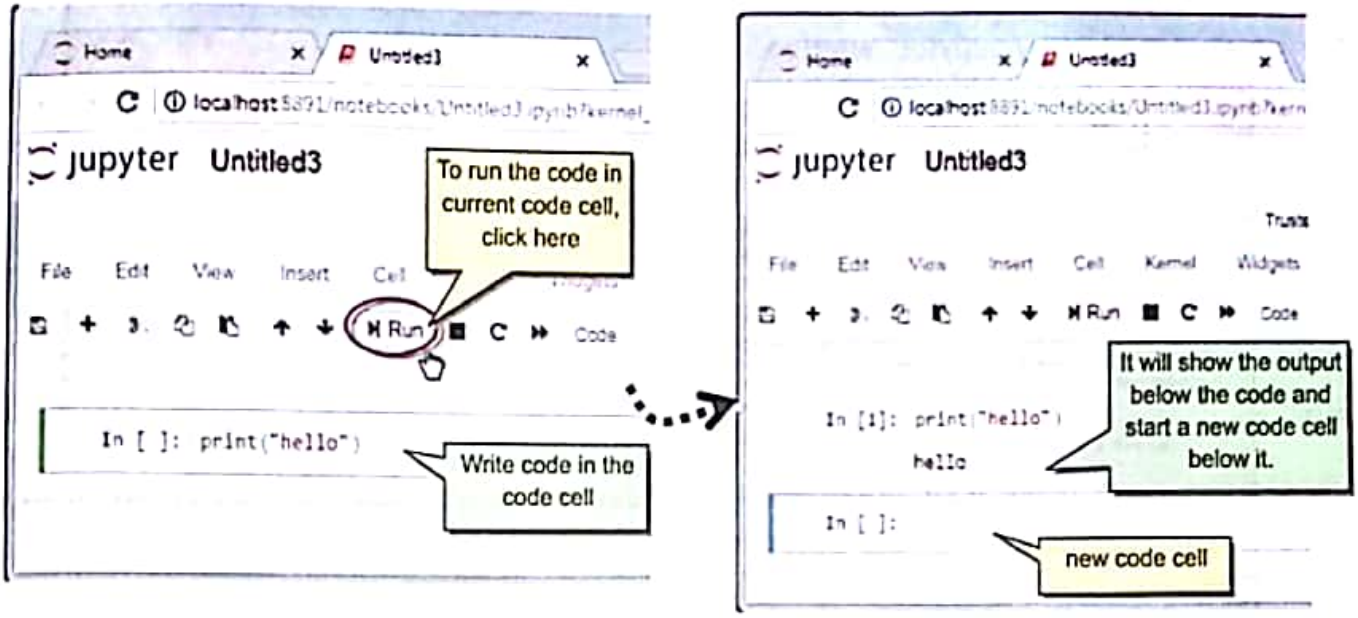


Figure 1.7

Script Mode

7. To run a script :

(*a*) Write the code of a Python script – a group of Python commands – in a code cell.

(*b*) Now click on its name to give it a new name (see below).

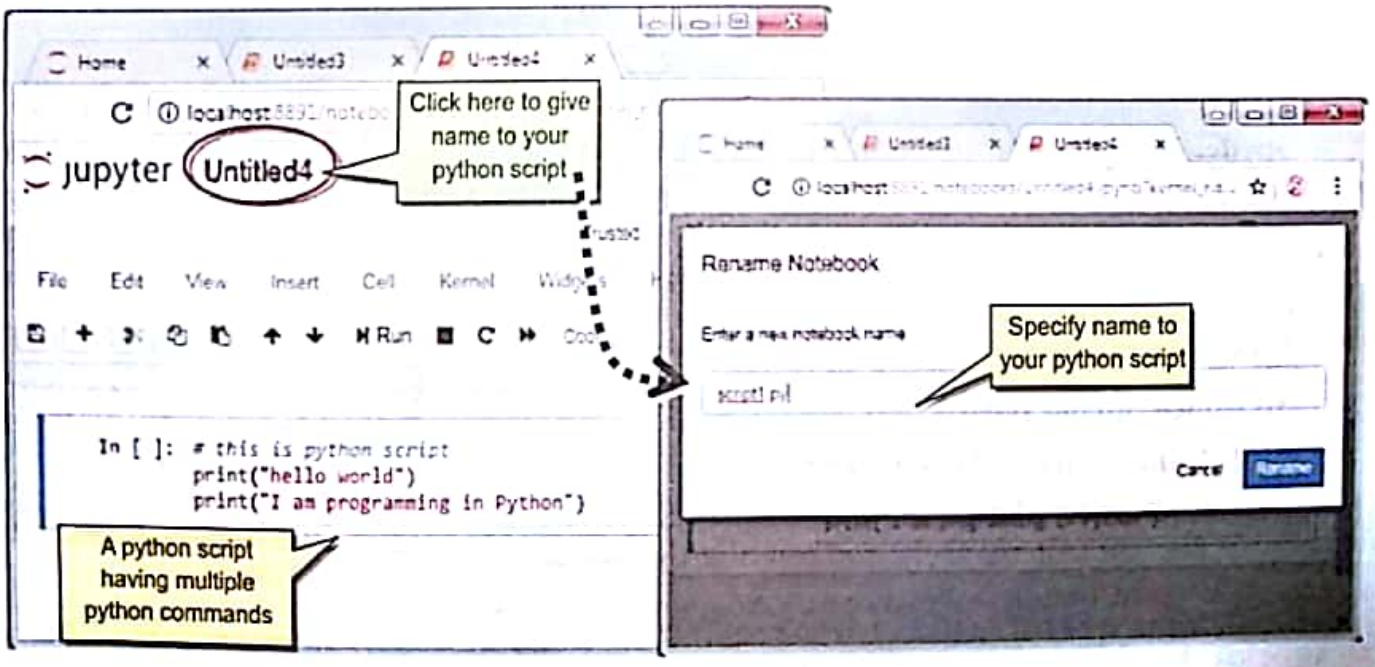


Figure 1.8

(*c*) Click on save icon first to save it.

(*d*) Click Run to run your script.

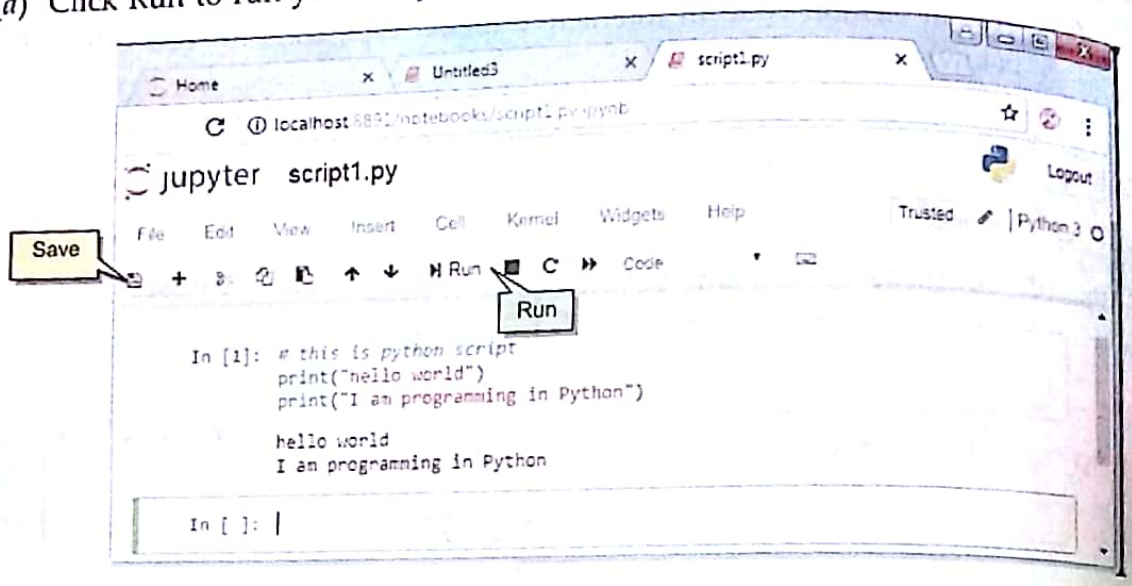


Figure 1.9

(*e*) You can check in the notebook dashboard, which is open in another tab ; it will show you the name of your currently saved script.

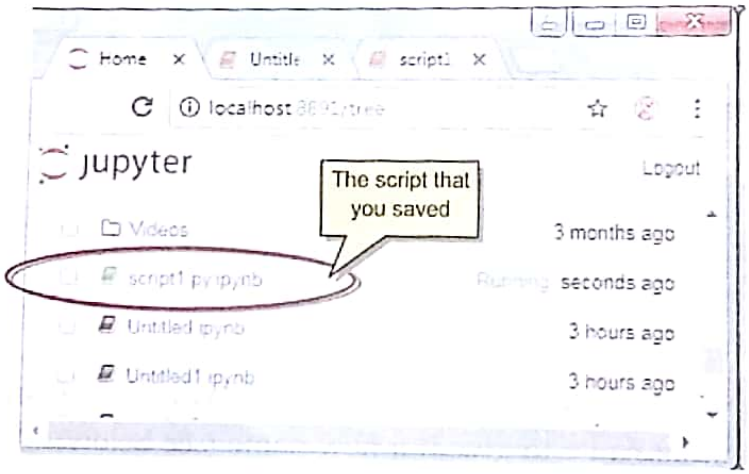


Figure 1.10

NOTE

To create a new notebook, use command **File → New notebook.**

1.4.2B Working in Spyder IDE

Spyder is a powerful interactive development environment for the Python language with advanced editing, interactive testing, debugging and introspection features. In fact, DataCamp community recommends Spyder IDE as the most preferred choice[6] out of top 5 Python development environments recommended by it. **Spyder IDE** comes preloaded with Anaconda distribution. It is my personal favourite for working on Python.

To launch Spyder IDE :

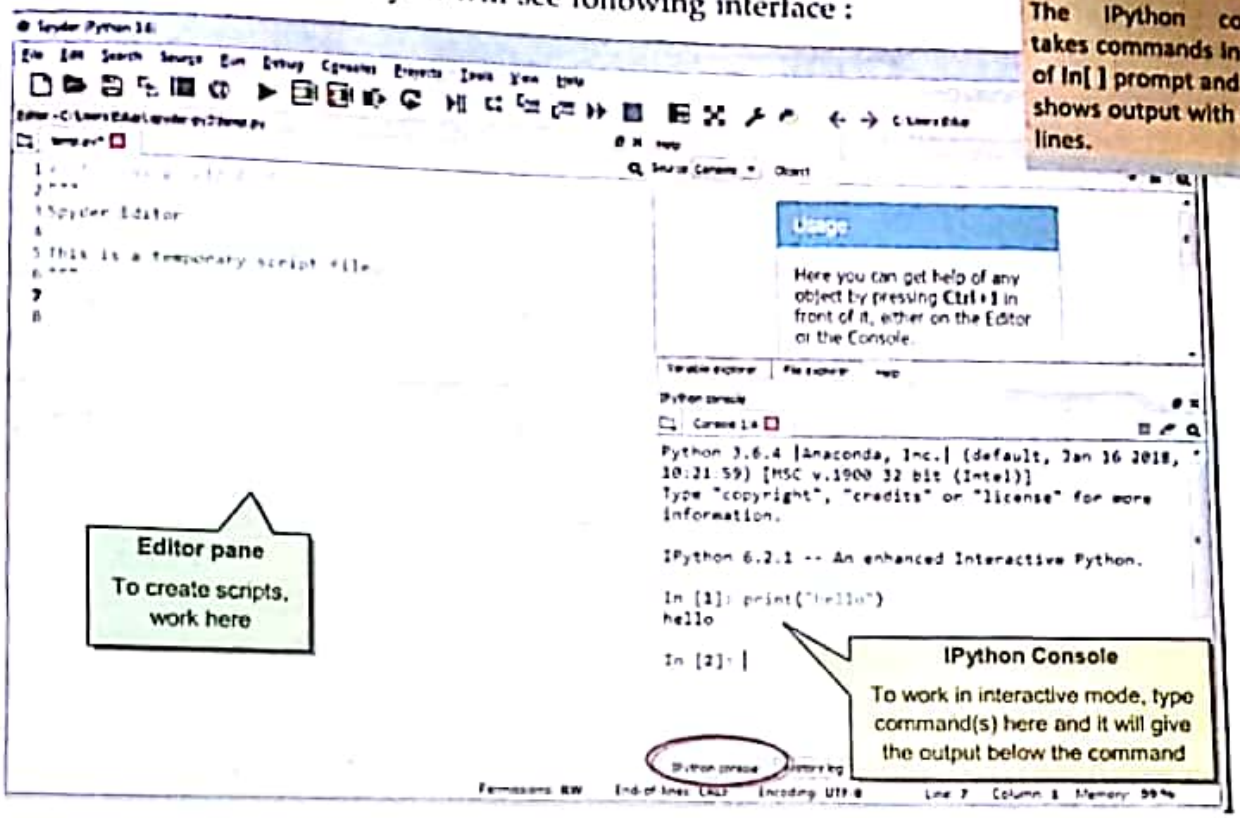
(*i*) Launch **Anaconda Navigator**

(*ii*) Click on **Spyder** tile in Anaconda navigator (refer to Fig. 1.4) [Alternatively, you can directly click at **Start button → Programs → Anaconda** (*folder name*) **→ Spyder**

6. **As per URL : https://www.datacamp.com/community/tutorials/data-science-python-ide.**

Spyder Interface

Once the spyder is loaded, you will see following interface :



NOTE

The IPython console takes commands in front of In[ ] prompt and often shows output with Out( ) lines.

Figure 1.11

Interactive Mode

To work in interactive mode in **Spyder IDE**, type your command(s) in the **IPython console** pane of spyder window. It will give you the output of the command there itself. See figure above where we typed command :

```
print("Hello")
```

in the IPython console pane and it gave us the result below it.

NOTE

We favour ANACONDA PYTHON DISTRIBUTION for these simple reasons :

(*i*) it is free and open source ;

(*ii*) it comes preloaded with many packages and libraries,

(*iii*) it offers **jupyter notebook** (preferred choice for bloggers) and **spyder IDE** like toolkits to write and Python scripts and code interactively.

I personally use Anaconda Navigator and work with Spyder IDE mostly and also on Jupyter notebook sometimes.

Script Mode

To work in script mode in **Spyder IDE**, type your script's commands in the editor pane.

- To start a new script file, click **File → New File...**
- To save current script, use command **File → Save** or **File → SaveAs.**
- Python scripts have File extension as **.py.**

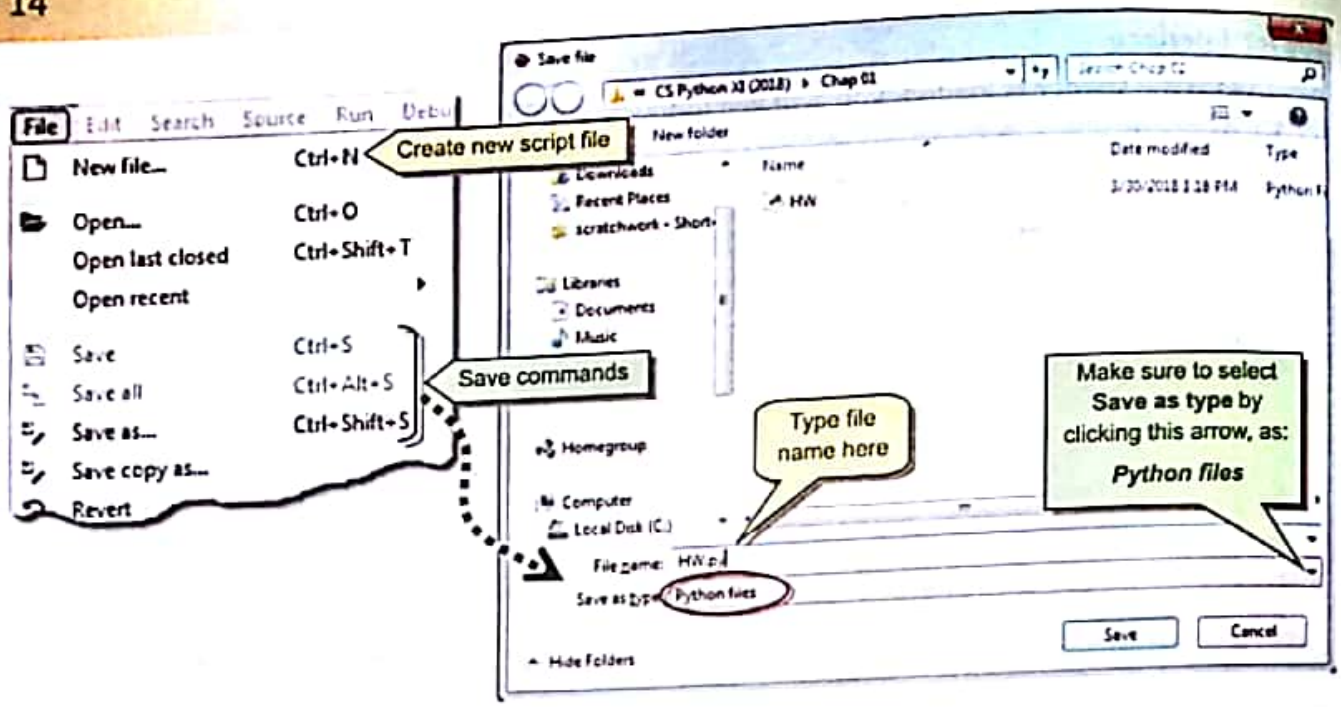


Figure 1.12

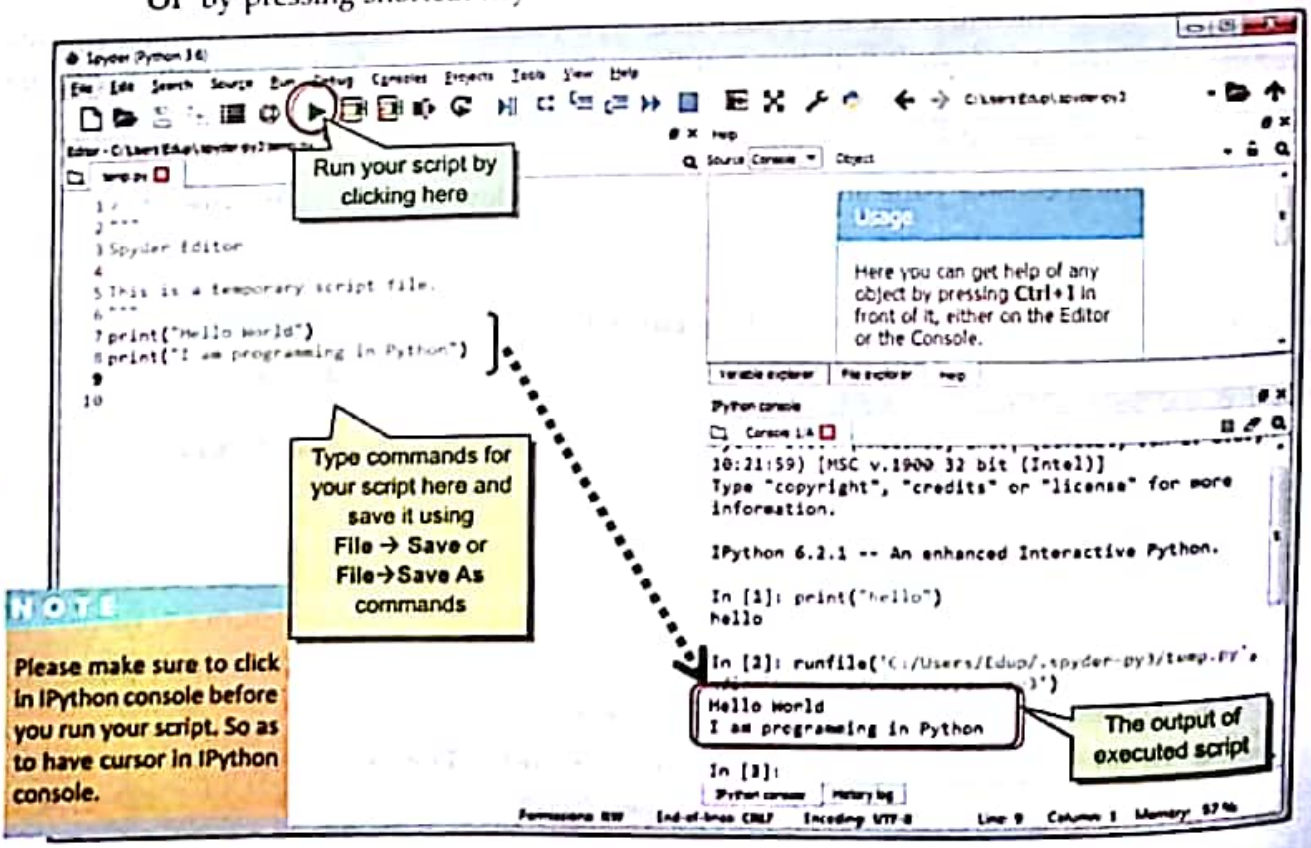
- Please make sure to select file type as **Python Files.**

After you have typed your script (and saved it), you can run your script by :

- Clicking at **Run** icon (see below) on the toolbar

  Or by clicking at **Run → Run** command

  Or by pressing shortcut key **F5.**



PyScripter IDE – another Popular IDE

There is another free and open source IDE by the name **PyScripter** IDE available. You can also use it for writing and executing Python programs if you want. Its interface looks like as shown below. For ***interactive mode***, you can type commands directly in console window and for ***script mode***, you can write script in the editor pane and then run it by pressing **Run icon** or through **Run menu**.

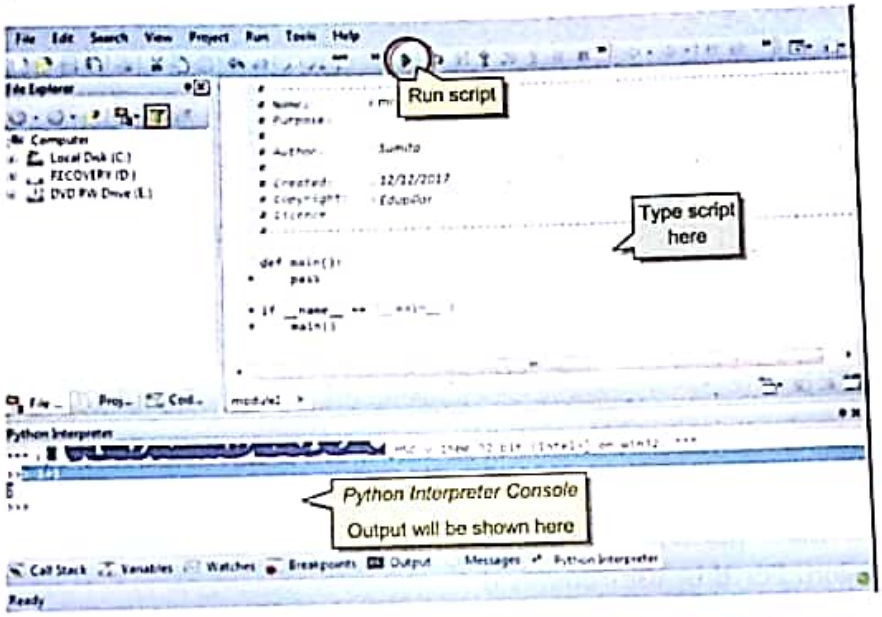


Figure 1.13

Now that you are familiar with different ways of working in Python, we can proceed with our discussion of Python. Although you can work on any of these, my personal favourite is **Anaconda Distribution** and **Spyder IDE**.

1.4.3 Writing and Compiling Python Program with Command Line in Linux

To create and run a Python program on Linux platform, you need to do this :

(*i*) Write your program code in a text editor.

(*ii*) Save your program with .py extension.

(*iii*) Compile and run on the command prompt by giving appropriate command.

Let us see how.

Typing Python program in a Text Editor

There is a variety of choices for text editors when it comes to Linux. There are command line editors like *nano, vim, emacs* etc[7]. ; Or you can work with the GUI editors like *Sublime Text, atom, brackets* etc. This really is the choice of the user.

But here, we will be working with *Sublime Text* 3.

Let us see, how we can write a simple ***Hello World*** **program in *Sublime Text*** and compile and run it using the terminal. **(Please note that *Sublime Text* editor must be installed on the machine before you can use it).**

7. *Vim* and *emacs* have their GUI version as well.

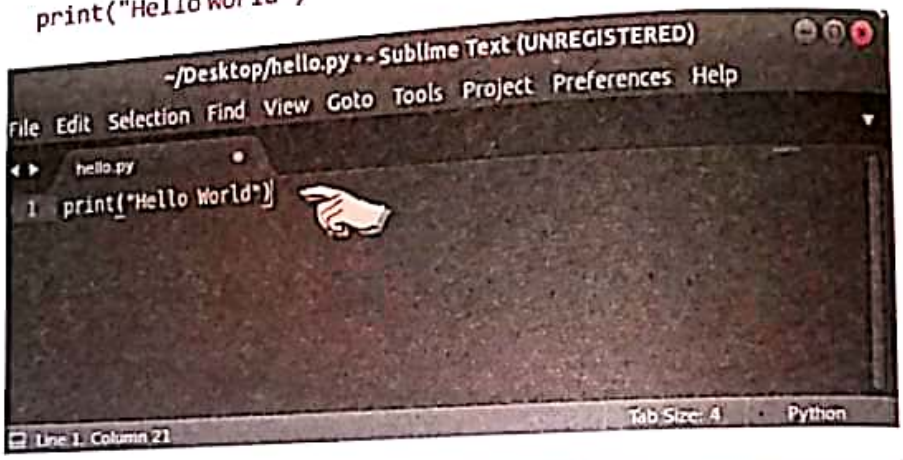
Coding in Text Editor (Sublime Text)

(*i*) **Open Sublime Text.** Press the super key (windows key) and search for Sublime Text. You will find an option to open sublime text. Click on the icon and it will appear.

(*ii*) **Create a New File.** Either use command **File → New** or press **Ctrl + N** on the keyboard.

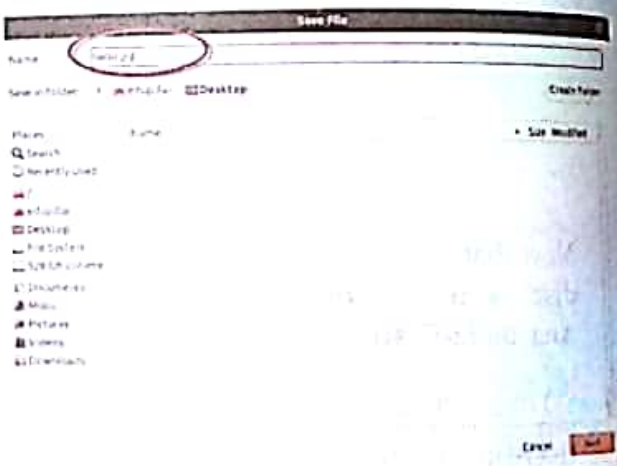
(*iii*) A new untitled file will open. Write the hello world code :

```
print("Hello World")
```



(*iv*) Save the file with proper extension. For this, use command **File → Save** or press **Ctrl + S** from the keyboard.

In the *Save dialog* that appears, write the name of the file and give an extension .py. We have named the file as **hello.py**.



Compiling and Running the Code using the Terminal

(*i*) Open the terminal. Press **Ctrl + Alt + T** from the keyboard. A command prompt will appear.

(*ii*) Go the folder where you saved the file[8].

(*iii*) Type the following command in front of the prompt and press enter :

```
python <filename>.py
```

That is we should type following command to compile and run our program *hello.py*[9] :

```
python hello.py
```

8. You can use the command *cd*. Alternatively, open files (press Windows key and search for files.) Go to the correct folder. Right click in the empty space and click "**Open in Terminal**".
9. There is no concept of extensions in Linux however, sublime text uses these extensions to identify the format and highlights the syntax accordingly. You can externally set the syntax format as well. But we are not covering that.

(iv) It will compile and run your program. You will see the output of the program in the terminal window.

1.5 UNDERSTANDING FIRST PROGRAM/SCRIPT

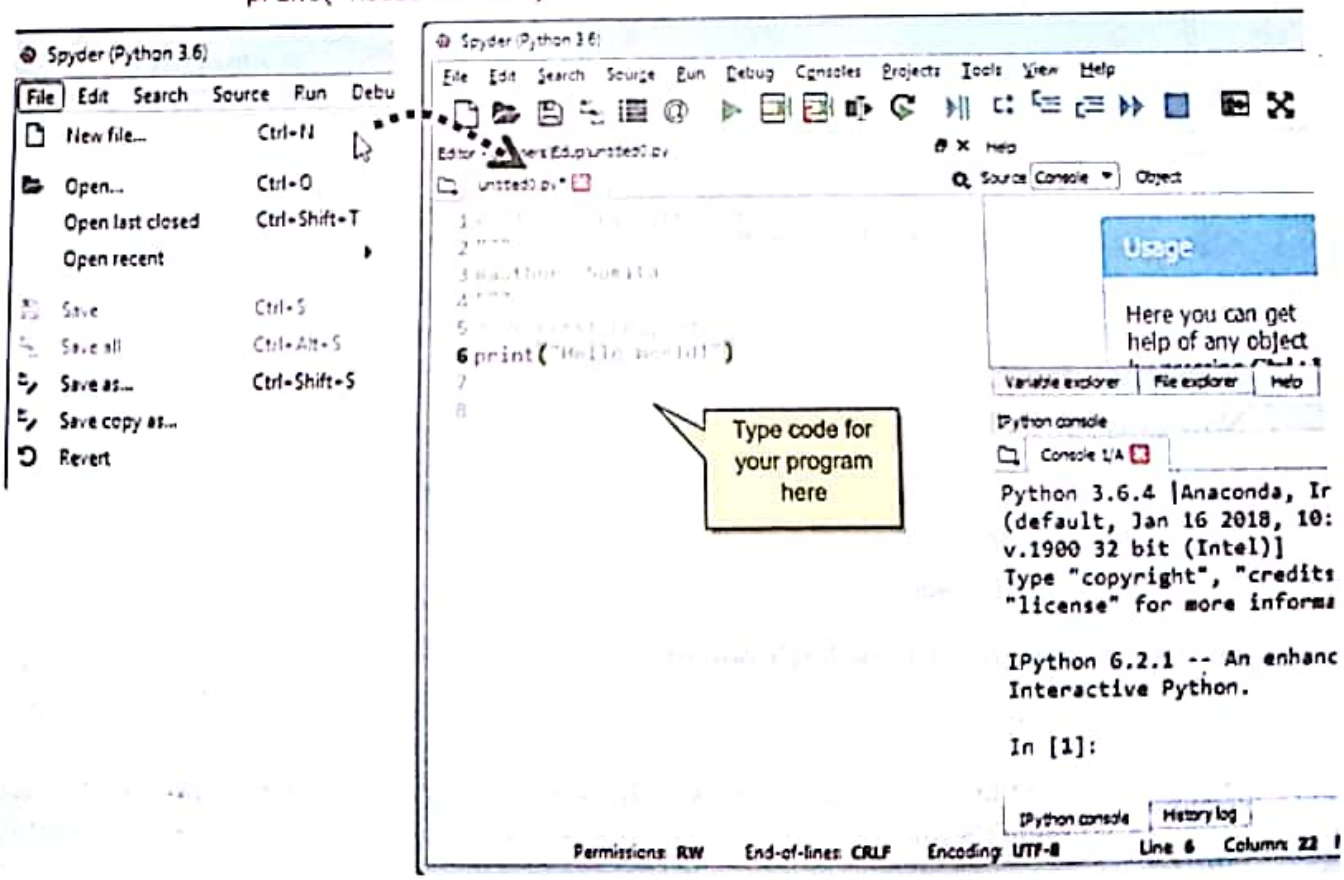
Let us create our first program – a simple "Hello World" program. (We are using Spyder IDE for this. You can use any of the above mentioned options.)

To create this,

1. Start Spyder IDE or any other editor of your choice.

2. Start new file (File → New File) and type the following text in the editor window :

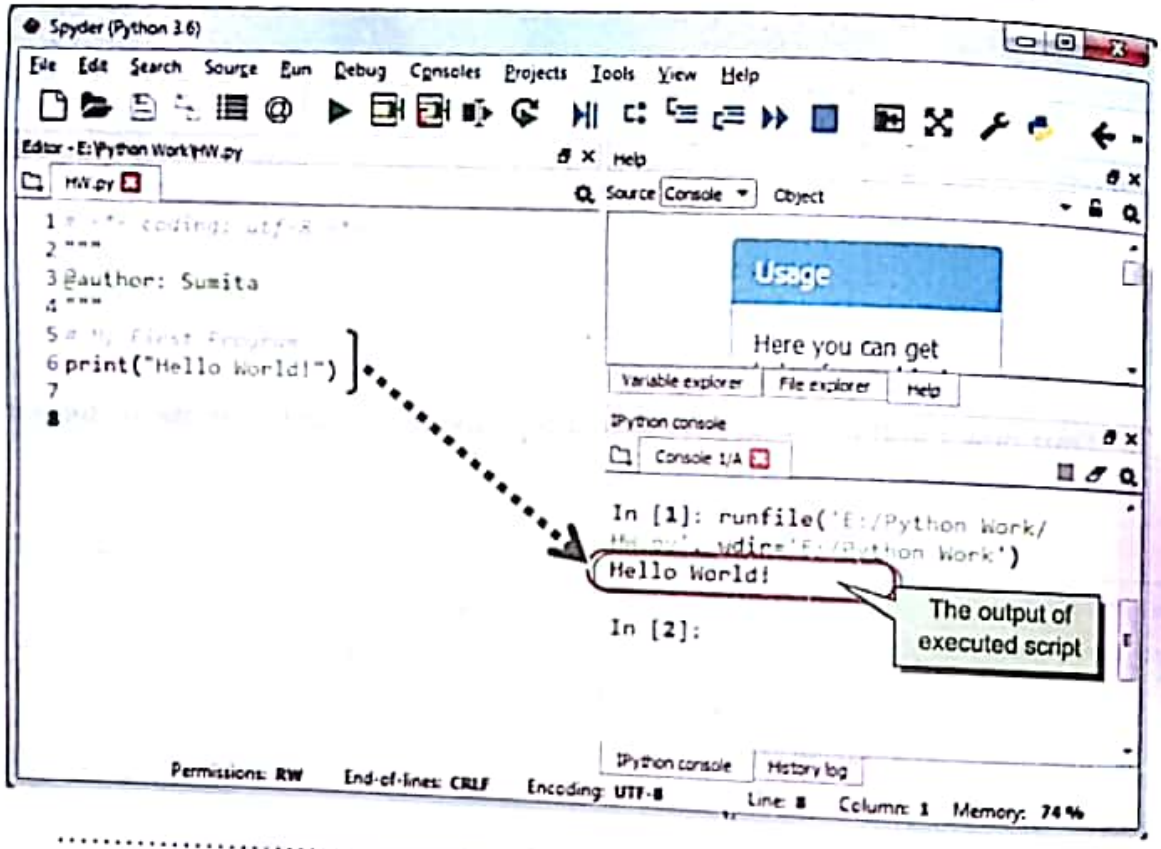
```
#My First Program
print("Hello World!")
```



3. Now save your script with a desired name.
   - Make sure to select **Save as type** as **Python Files.**
   - Give .py extension to your python program file.

4. Now run your script by clicking **Run icon** [▶] or by clicking **Run → Run command** or by pressing F5. (Firstly, click in IPython console window)

5. It will show you the output in the console window pane (see below).



Analysing Script and Output

Now carefully look at the script that you typed.

You typed two lines :

```
# My First Program
print("Hello World")
```

But Python gave you just one line's output as :

```
Hello World!
```

The reason being that any line that begins with a # symbol is a comment in Python. That is, it is for the programmer's information only ; Python will completely ignore all the lines starting with a #.

Understanding print( )

To print or display output, Python 3.x provides **print( ) function**[10]. You can use **print( )** as

```
print(<objects to be printed>...)
```

e.g., when you wrote

```
print( )
statement 1     print("Hello World!")
```

String "Hello World!" is the object to be printed

it printed :

```
Hello World!
```

Similarly, to print other strings you may give something like.

```
print( )
statement 2     print('My name is Misha")
```

String object 'My name is Misha' to be printed.

it will print

```
My name is Misha
```

Carefully look at both the **print( )** statements given above. Could you notice anything other than two different string values ? No ?

Carefully notice that the first string is enclosed in **double quotes** and the second string is enclosed in **single quotes.**

```
print("Hello World!")
```

String enclosed in double quotes

```
print('My name is Misha')
```

String enclosed in single quotes

Both these strings are valid in Python. You can enclose your strings in either double quotes or in single quotes, but just ensure that opening and closing quotation mark should be of same type. You cannot have a string like

```
'Hey there"
```

Error! opening and closing Quotation marks do not match

Following are some valid **print( )** statements.

```
print('The Golden Ratio')
print("has same letters as")
print('The God Relation')
```

With this we have come to the end of this chapter. Let us quickly revise what we have learnt so far.

10. In Python 2.x, print is a statement, not a function.

PRIP PYTHON INTERACTIVE MODE — Progress In Python 1.1

Start Python (Anaconda Navigator and then Spyder IDE or any other IDE of your choice)

On the IPython console perform this session

Python prompt as calculator

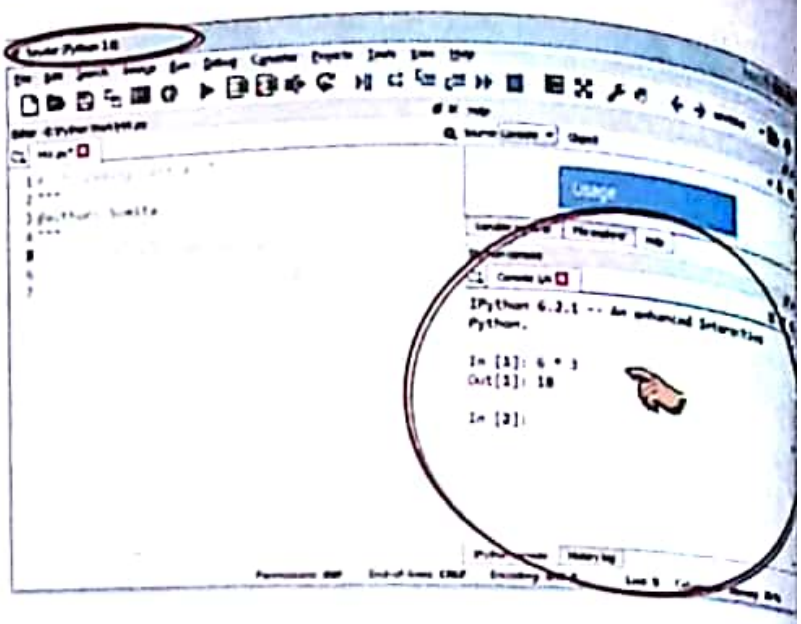
1. *In front of Python prompt, i.e.,* **In[ ]:**, *type following expressions one by one, by pressing return key after every expression, e.g.,*

```
In[1] : 6 * 3
```

It will show output lines starting with **Out[ ]:**, *e.g.*:

```
Out[2] : 18
```

⋮



Please check the practical component-book – Progress in Computer Science with Python and fill it there in PriP 1.1 under Chapter 1 after practically doing it on the computer.

>>>❖<<<

PRIP WORKING IN SCRIPT MODE — Progress In Python 1.2

Start Anaconda Navigator and then Spyder IDE or any other IDE of your choice.

Open new File window

Click **File → New File** in **Spyder.**

1. *Type following code (as it is) in the new window opened by IDLE*

```
print ("Hello Nathan", "Hello Lalit")
greet1 = "Hello Nathan"
greet2 = "Hello Lalit"
name1 = " Nathan "
name2 = " Lalit "
print (greet1, greet2)
print ("Hello", name1, ",", name2)
```

⋮

Please check the practical component-book – Progress in Computer Science with Python and fill it there in PriP 1.2 under Chapter 1 after practically doing it on the computer.

>>>❖<<<

LET US REVISE

- *Python was developed by Guido Van Rossum in February 1991.*
- *Python offers following advantages*
  - easy to use
  - expressive
  - complete
  - cross-platform
  - free and open source
- *Python also has these limitations :*
  - Not the fastest language
  - Lesser libraries than C, Java, Perl
  - Not strong on Type-binding.
- *In Python, one can work in two different ways : (i) Interactive mode, (ii) Script mode.*
- *Interactive mode does not save commands in form of a program and also, output is sandwiched between commands.*
- *Interactive mode is suitable for testing code.*
- *Script mode is useful for creating programs and then run the programs later and get complete output*
- *Python is an interpreted language.*
- *Python's interactive interpreter is also called Python Shell.*

Solved Problems

1. *Who developed Python Programming Language ?*

   Solution. Guido Van Rossum in 1990s developed Python programming language.

2. *Is Python an Object Oriented language ?*

   Solution. Yes, Python is an Object Oriented language.

3. *'Python is an interpreted high level language'. What does it mean to you ?*

   Solution. **'Python is a high level language'** means it is programmer-friendly *i.e.*, easy to program and comprehend.

   'Python is an interpreted language' means it requires an interpreter (not compiler) to execute its code line by line – one statement at a time.

4. *Python programming language got its name from which show.*

   Solution. Python programming language was named after a British TV show namely 'Monty Python's Flying Circus'.

5. *What does a cross platform language mean ?*

   Solution. A cross platform language means it can run well on variety of platforms like Windows, Linux/Unix, Macintosh etc. etc.

6. *Python is a Free and Open Source language. What do you understand by this feature ?*

   Solution. It means – to download Python, one needs not pay anything, because it is **Free.** And its source-code is also available, which can be modified/improved etc., because it is **open-source.**

7. *What is the difference between interactive mode and script mode in Python ?*

Solution. In *interactive mode*, instructions are given in front of Python prompt (*e.g.*, >>> or In[ ]: prompts) in Python Shell. Python carries out the given instruction and shows the result there itself. In *script mode*, Python instructions are stored in a file generally with .py extension and are executed together in one go as a unit. The saved instructions are known as *Python script* or *Python program*.

8. *What will be the output of following code :*

```
#This is a sample program
#to output simple statements
#print ("Such as")
print("Take every chance.")
print("Drop every fear.")
```

Pick the correct output from the following choices and give reason.

(*a*) *This is a sample program to output simple statements such as*

```
Take every chance.
Drop every fear.
```

(*b*) *Such as*

```
Take every chance.
Drop every fear.
```

(*c*)

```
Take every chance.
Drop every fear.
```

Solution. The correct output is (*c*).

Reason being : the code lines beginning with a # sign are comments. They are just for information and ignored by the Python interpreter. Hence, the third line **#print ("Such as")** will also be ignored by Python interpreter. Thus, the Python interpreter will give output of only **print()** statements.

9. *Which of the following are not valid strings in Python ?*

(*a*) "Hello" (*b*) 'Hello' (*c*) "Hello' (*d*) 'Hello" (*e*) {Hello}

Solution. Strings (*c*), (*d*) and (*e*) are not valid strings in Python.

GLOSSARY

Cross-platform	Something that is compatible across various platforms.
Open -Source	Computer software for which source-code is freely available.
Python Shell	Interactive interpreter of Python.
Source Code	Complete program instructions.

Assignments

Type A : Short Answer Questions/Conceptual Questions

1. When was Python released ?
2. Who was Python's developer and which two languages contributed to Python as a programming language ?
3. What is a cross-platform software ?
4. What are the advantages of Python programming language ?
5. What are some limitations of Python programming language ?
6. In how many different ways, can you work in Python ?
7. What are advantages/disadvantages of working in Interactive mode in Python ?
8. What are the advantages/disadvantages of working in script mode in Python ?

Type B : Application Based Question

1. Write instructions to get the following result :

```
Math is Fun so don't be resistant
Just learn the rules, the rules are consistent
And most important, you must be persistent !

Adding fractions, get common denominators.
Multiply by missing factors to get the denominators.
Add numerators only, NOT denominators.
```

Do it in both interactive mode and script mode.

2 Python Fundamentals

In This Chapter

2.1 INTRODUCTION

You must have heard the term IPO – Input, Process, Output. Most (in fact, nearly all) daily life and computer actions are governed by IPO cycle. That is, there is certain Input, certain kind of Processing and an Output.

Do you know that *programs* make IPO cycle happen ?

Anywhere and everywhere, where you want to transform some kind of input to certain output, you have some kind of input to certain output, you have to have a *program*. A program is a set of instructions that govern the processing. In other words, a program forms the base for processing.

In this chapter, we shall be talking about all basic elements that a Python program can contain. You'll be learning about Python's basics like *character set, tokens, expressions, statements, simple input and output* etc. So, are we all ready to take our first sincere step towards Python programming ? And, here we go :-).

2.2 PYTHON CHARACTER SET

Character set is a set of valid characters that a language can recognize. A character represents any letter, digit or any other symbol. Python supports Unicode encoding standard. That means, Python has the following character set :

- Letters — A-Z, a-z
- Digits — 0-9
- Special symbols — space + - * / ** \ ( ) [ ] { } // = != == < , > . ' " "" , ; : % ! & # <= >= @ _ (underscore)
- Whitespaces — Blank space, tabs (→), carriage return (↵), newline, formfeed.
- Other characters — Python can process all ASCII and Unicode characters as part of data or literals.

2.3 TOKENS

In a passage of text, individual words and punctuation marks are called *tokens* or *lexical units* or lexical elements. The smallest individual unit in a program is known as a *Token* or a *lexical unit*.

Python has following tokens :

(*i*) Keywords (*ii*) Identifiers (Names) (*iii*) Literals
(*iv*) Operators (*v*) Punctuators

TOKENS

The smallest individual unit in a program is known as a ***Token*** or a ***lexical unit***.

Let us talk about these one by one.

2.3.1 Keywords

Keywords are the words that convey a special meaning to the language compiler/interpreter. These are reserved for special purpose and must not be used as normal identifier names.

Python programming language contains the following keywords :

KEYWORD

A *keyword* is a word having special meaning reserved by programming language.

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

2.3.2 Identifiers (Names)

Identifiers are fundamental building blocks of a program and are used as the general terminology for the names given to different parts of the program *viz.* variables, objects, classes, functions, lists, dictionaries etc.

Identifier forming rules of Python are being specified below :

- An identifier is an arbitrarily long sequence of letters and digits.
- The first character must be a letter; the underscore (_) counts as a letter.

- Upper and lower-case letters are different. All characters are significant.
- The digits 0 through 9 can be part of the identifier except for the first character.
- Identifiers are unlimited in length. *Case* is significant *i.e.*, Python is case sensitive as it treats upper and lower-case characters differently.
- An identifier must not be a keyword of Python.
- An identifier cannot contain any special character except for underscore (_).

The following are some *valid* identifiers :

```
Myfile      DATE9_7_77     Z2T0Z9
MYFILE      _DS            _HJI3_JK
_CHK        FILE13
```

NOTE

Python is case sensitive as it treats upper and lower-case characters differently.

The following are some *invalid* identifiers :

`DATA-REC`	contains special character - (hyphen) (other than A - Z, a - z and _ (underscore))
`29CLCT`	Starting with a digit
`break`	reserved keyword
`My.file`	contains special character dot (.)

2.3.3 Literals / Values

Literals (often referred to as constant-Values) are data items that have a fixed value. Python allows several kinds of literals :

(*i*) String literals (*ii*) Numeric literals (*iii*) Boolean literals

(*iv*) Special Literal *None* (*v*) Literal Collections

2.3.3A String Literals

The text enclosed in quotes forms a string literal in Python. *For example, 'a', 'abc', "abc"* are all string literals in Python. Unlike many other languages, both single character enclosed in quotes such as *"a"* or *'x'* or multiple characters enclosed in quotes such as *"abc"* or *'xyz'* are treated as String literals.

As you can notice, one can form string literals by enclosing text in both forms of quotes – single quotes or double quotes. Following are some valid string literals in Python :

```
'Astha'          "Rizwan"
'Hello World'    "Amy's"
"129045"
'1-x-0-w-25'
"112FBD291"
```

NOTE

In Python, one can form string literals by enclosing text in both forms of quotes – single quotes or double quotes.

Check Point

2.1

1. What is meant by token ? Name the tokens available in Python.
2. What are keywords ? Can keywords be used as identifiers ?
3. What is an identifier ? What are the identifier forming rules of Python ?
4. Is Python case sensitive ? What is meant by the term 'case sensitive' ?
5. Which of the following are valid identifiers and why/why not :

   Data_rec, _data, 1 data, data1, my.file, elif, switch, lambda, break ?

Python allows you to have certain *nongraphic-characters* in String values. ***Nongraphic characters*** are those characters that cannot be typed directly from keyboard *e.g.*, backspace, tabs, carriage return etc. (No character is typed when these keys are pressed, only some action takes place). These *nongraphic-characters* can be represented by using escape sequences. An escape sequence is represented by a backslash (\) followed by one or more characters.[1]

STRING LITERALS

A *string literal* is a sequence of characters surrounded by quotes (single or double or triple quotes).

Following table (Table 2.1) gives a listing of escape sequences.

Table 2.1 *Escape Sequences in Python*

Escape sequence	What it does [Non-graphic character]	Escape sequence	What it does [Non-graphic character]
\\	Backslash (\)	\r	Carriage Return (CR)
\'	Single quote (')	\t	Horizontal Tab (TAB)
\"	Double quote (")	\uxxxx	Character with 16-bit hex value xxxx (Unicode only)
\a	ASCII Bell (BEL)	\Uxxxxxxxx	Character with 32-bit hex value xxxxxxxx (Unicode only)
\b	ASCII Backspace (BS)	\v	ASCII Vertical Tab (VT)
\f	ASCII Formfeed (FF)	\ooo	Character with octal value ooo
\n	New line character	\xhh	Character with hex value hh
\N{name}	Character named name in the Unicode[1] database (Unicode only)		

In the above table, you see sequences representing \, ', ". Though these characters can be typed from the keyboard but when used without escape sequence, these carry a special meaning and have a special purpose, however, if these are to be typed *as it is*, then escape sequences should be used. (In Python, you can also directly type a double-quote inside a single-quoted string and vice-versa. *e.g.*, "anu's" is a valid string in Python)

NOTE

An *escape sequence* represents a single character and hence consumes one byte in ASCII representation.

String Types in Python

Python allows you to have *two* string types :

(*i*) Single-line Strings

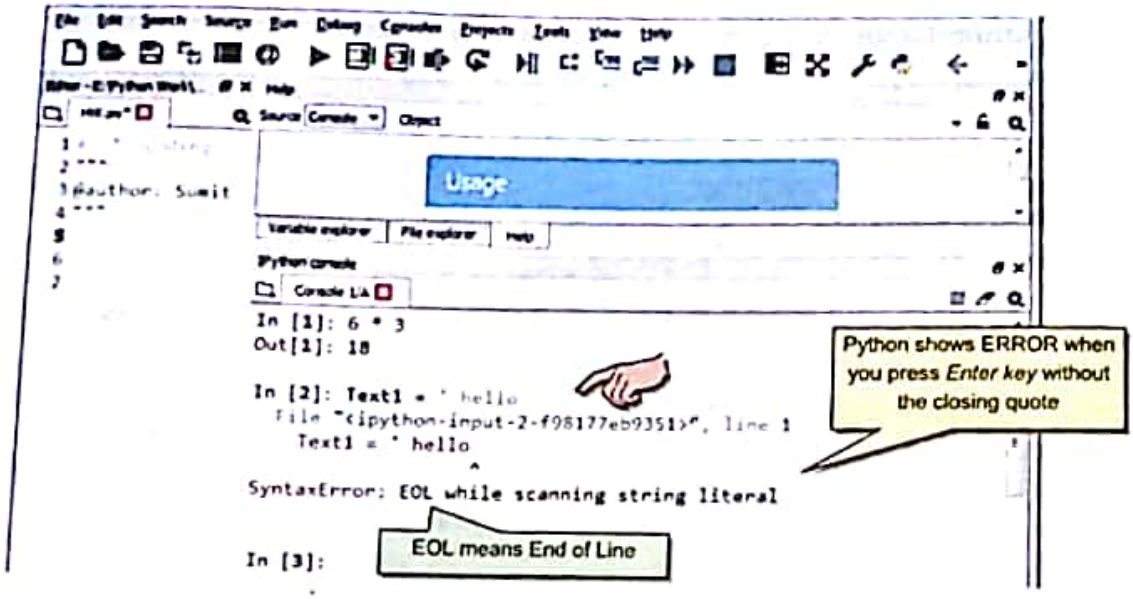
(*ii*) Multiline Strings

(*i*) ***Single-line Strings (Basic strings).*** The strings that you create by enclosing text in single quotes (' ') or double quotes (" ") are normally single-line strings, *i.e.*, they must terminate in one line. To understand this, try typing the following in IDLE window and see yourselves :

```
Text1= 'hello ↵
there'
```

1. Unicode and ASCII are two character encodings discussed later in Chapter 13 (Data Representation).

Python will show you an error the moment you press Enter key after *hello* (see below)



The reason for the above error is quite clear – Python by default creates single-line strings with both single or double quotes. So, if at the end of a line, there is no closing quotation mark for an opened quotation mark, Python shows an error.

(*ii*) ***Multiline Strings.*** Sometimes you need to store some text spread across multiple lines as one single string. For that Python offers multiline strings.

Multiline strings can be created in *two* ways :

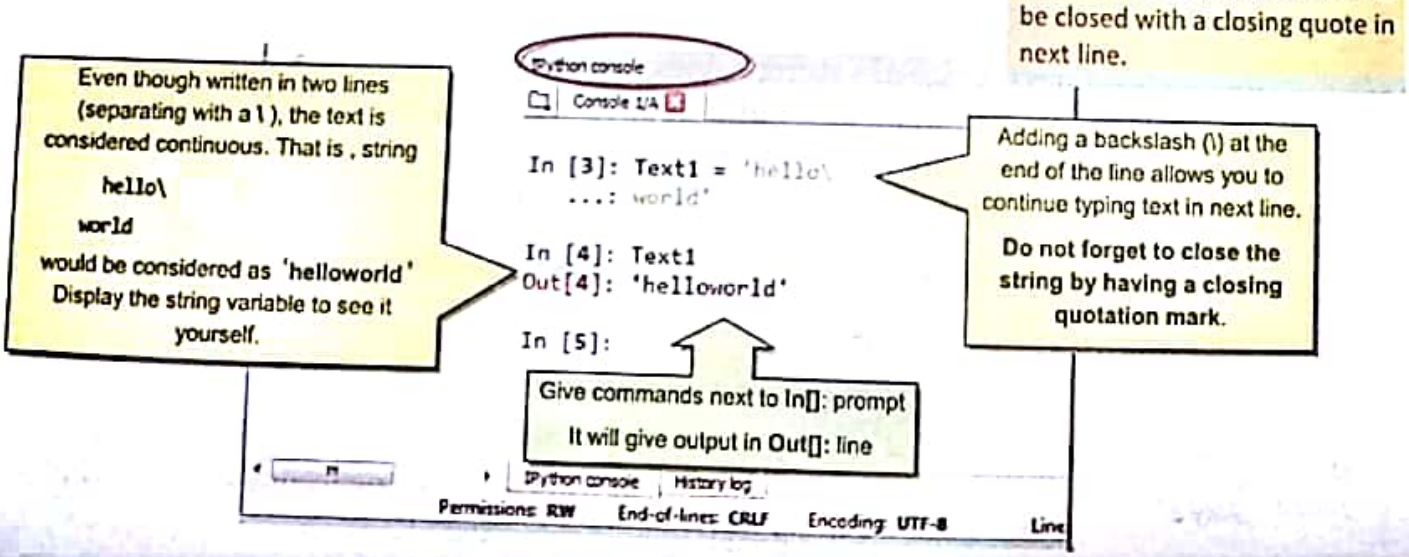
(*a*) *By adding a backslash at the end of normal single-quote / double-quote strings.* In normal strings, just add a backslash in the end before pressing Enter to continue typing text on the next line. *For instance,*

```
Text1 = 'hello\
world'
```

Do not indent when continuing typing in next line after '\'.

A basic string must be completed on a single line, or continued with a backslash (\) as the very last character of a line if it is to be closed with a closing quote in next line.

Following figure shows this :

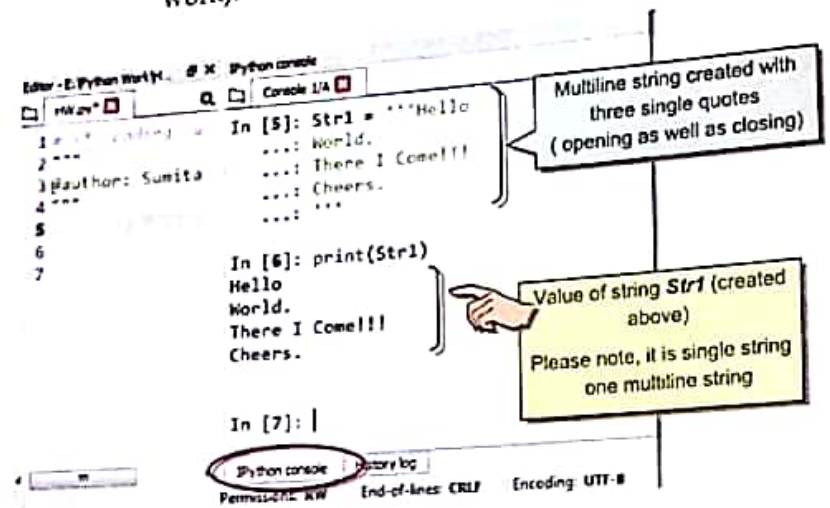


(b) *By typing the text in triple quotation marks. (No backslash needed at the end of line).* Python allows to type multiline text string by enclosing them in triple quotation marks (both triple apostrophe or triple quotation marks will work).

NOTE

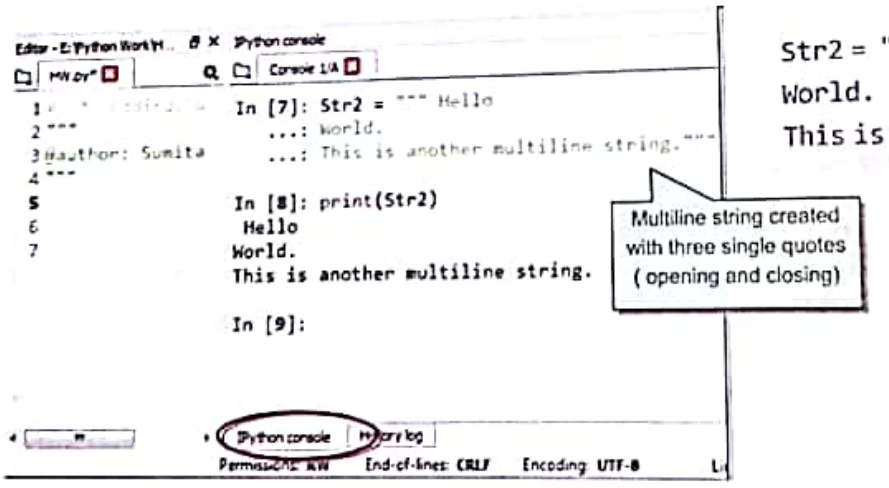
A multiline string continues on until the concluding triple-quote or triple-apostrophe.

For example,




```
Str1 = '''Hello
World.
There I Come !!!
Cheers.
'''
```

Or




```
Str2 = """Hello
World.
This is another multiline string."""
```

Size of Strings

Python determines the size of a string as the count of characters in the string. *For example,* size of string *"abc"* is 3 and of *'hello'* is 5. But if your string literal has an escape sequence contained within it, then make sure to count the escape sequence as one character. Consider some examples given below :

`'\\'`	size is 1	(\\ is an escape sequence to represent backslash)
`'abc'`	size is 3	
`"\ab"`	size is 2	(\a is an escape sequence, thus one character).
`"Seema\'s pen"`	size is 11	(For typing apostrophe (') sign, escape sequence \' has been used.).
`"Amy's"`	size is 4	Python allows a single quote (without escape sequence) in double-quoted string and vice-versa.

For multiline strings created with triple quotes, while calculating size, the EOL (end-of-line) character at the end of the line is also counted in the size. For example, if you have created a string Str3 as :

```
Str3 = '''a ↵
b ↵
c'''
```

These enter keys are considered as EOL (End-of-Line) characters and counted in the length of multiline string

```
nstr = '''he\
11\
o'''
```

But backslashes (\ ) at the end of intermediate lines are not counted in the size of the multiline strings.

then size of the string Str3 is 5 (three characters *a*, *b*, *c* and two *EOL* characters that follow characters *a* and *b* respectively).

For multiline strings created with single/double quotes and backslash character at end of the line, while calculating size, the backslashes are not counted in the size of the string ; also you cannot put EOLs using return key in single/double quoted multiline strings *e.g.*,

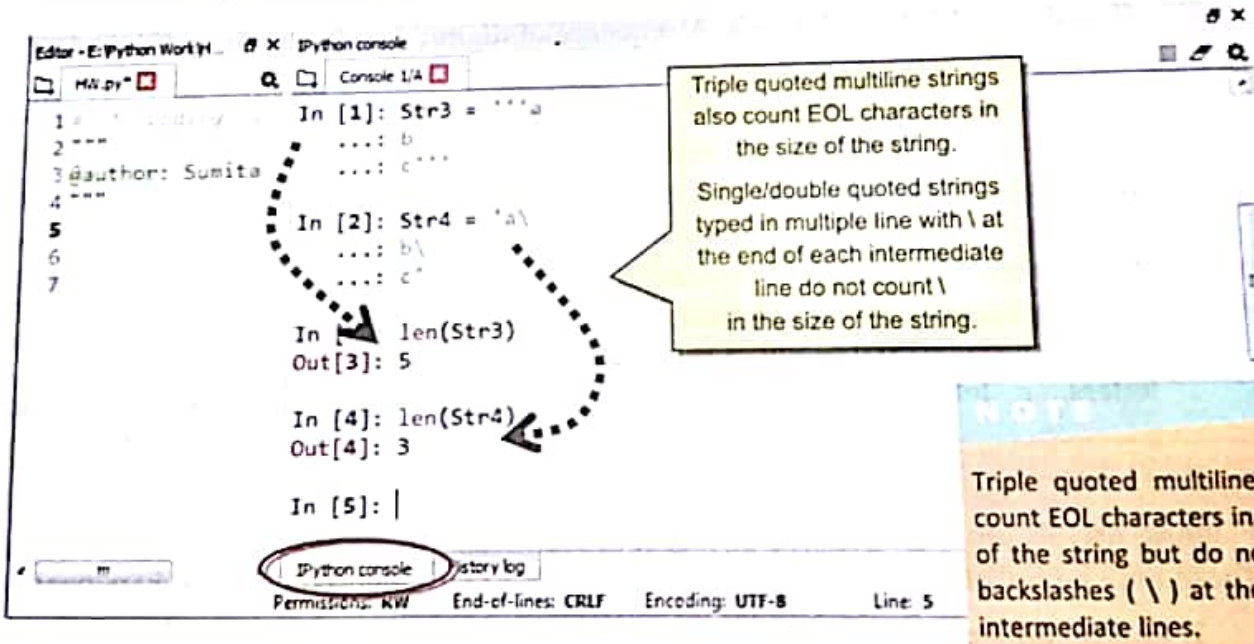
```
Str4 = 'a\
b\
c'
```

The size of string Str4 is 3 (only 3 characters, no backslash counted.)

To check the size of a string, you may also type len(<stringname>) command on the Python prompt in console window shell as shown in the following figure :

NOTE

Use len(<object name>) to get the size or length of an object.



NOTE

Triple quoted multiline strings count EOL characters in the size of the string but do not count backslashes (\ ) at the end of intermediate lines.

2.3.3B Numeric Literals

The numeric literals in Python can belong to any of the following *four* different numerical types :

int (signed integers)	often called just integers or ints, are positive or negative whole numbers with no decimal point.
float (floating point real values)	floats represent real numbers and are written with a decimal point dividing the integer and fractional parts.
complex (complex numbers)	are of the form $a + bJ$, where a and b are *floats* and J (or j) represents $\sqrt{-1}$, which is an imaginary number). a is the real part of the number, and b is the imaginary part.

Let us talk about these literal types one by one.

Integer Literals

Integer literals are whole numbers without any fractional part. The method of writing integer constants has been specified in the following rule :

An integer constant must have at least one digit and must not contain any decimal point. It may contain either (+) or (–) sign. A number with no sign is assumed to be positive. Commas cannot appear in an integer constant.

Python allows *three* types of integer literals :

(*i*) ***Decimal Integer Literals.*** An integer literal consisting of a sequence of digits is taken to be decimal integer literal unless it begins with 0 (digit zero).

For instance, 1234, 41, +97, –17 are decimal integer literals.

(*ii*) ***Octal Integer Literals.*** A sequence of digits starting with 0o (digit zero followed by letter o) is taken to be an octal integer.

For instance, decimal integer 8 will be written as 0o10 as octal integer. ($8_{10} = 10_8$) and decimal integer 12 will be written as 0o14 as octal integer ($12_{10} = 14_8$).

An octal value can contain only digits 0-7 ; 8 and 9 are invalid digits in an octal number i.e., 0o28, 0o19, 0o987 etc., are examples of invalid octal numbers as they contain digits 8 and 9 in them.

(*iii*) ***Hexadecimal Integer Literals.*** A sequence of digits preceded by *0x* or *0X* is taken to be an hexadecimal integer.

For instance, decimal 12 will be written as 0XC as hexadecimal integer.

Thus, number 12 will be written either as 12 (as decimal), 0o14 (as octal) and 0XC (as hexadecimal).

A hexadecimal value can contain digits 0-9 and letters A-F only *i.e., 0XBK9, oxPQR, 0x19AZ* etc., are examples of invalid hexadecimal numbers as they contain invalid letters, *i.e.*, letters other than A-F.

NOTE

Many programming languages such as C, C++, and even Python 2.x too have two types for integers: int (for small integers) and long (for big integers). But in Python 3.x, there is only one integer type <class 'int'> that works like long integers and can support all small and big integers.

Floating Point Literals

Floating literals are also called real literals. Real literals are numbers having fractional parts. These may be written in one of the *two* forms called *Fractional Form* or the **Exponent Form.**

1. ***Fractional form.*** A real literal in Fractional Form consists of signed or unsigned digits including a decimal point between digits.

The rule for writing a real literal in fractional form is :

A real constant in fractional form must have at least one digit with the decimal point, either before or after. It may also have either + or – sign preceding it. A real constant with no sign is assumed to be positive.

The following are valid real literals in fractional form :

2.0, 17.5, –13.0, –0.00625, .3 (*will represent 0.3*), 7. (*will represent 7.0*)

The following are invalid real literals :

7	(*No decimal point*)
+17 / 2	(*/-illegal symbol*)
17,250.26.2	(*Two decimal points*)
17,250.262	(*comma not allowed*)

2. *Exponent form.* A real literal in *Exponent form* consists of *two* parts : ***mantissa*** and ***exponent***. For instance, 5.8 can be written as $0.58 \times 10^1 = 0.58$ **E01**, where *mantissa* part is **0.58** (the part appearing before E) and *exponent* part is **1** (the part appearing after E). E01 represents 10^1 . The rule for writing a real literal in exponent form is :

> *A real constant in exponent form has two parts : a* ***mantissa*** *and an* ***exponent****. The mantissa must be either an integer or a proper real constant. The mantissa is followed by a letter E or e and the exponent. The exponent must be an integer.*

The following are the valid real literals in exponent form : 152E05, 1.52E07, 0.152E08, 152.0E08, 152E+8, 1520E04, –0.172E–3, 172.E3, .25E–4, *3.E3* (equivalent to 3.0E3)

(Even if there is no preceding or following digit of a decimal point, Python 3.x will consider it right)

The following are invalid real literals in exponent form :

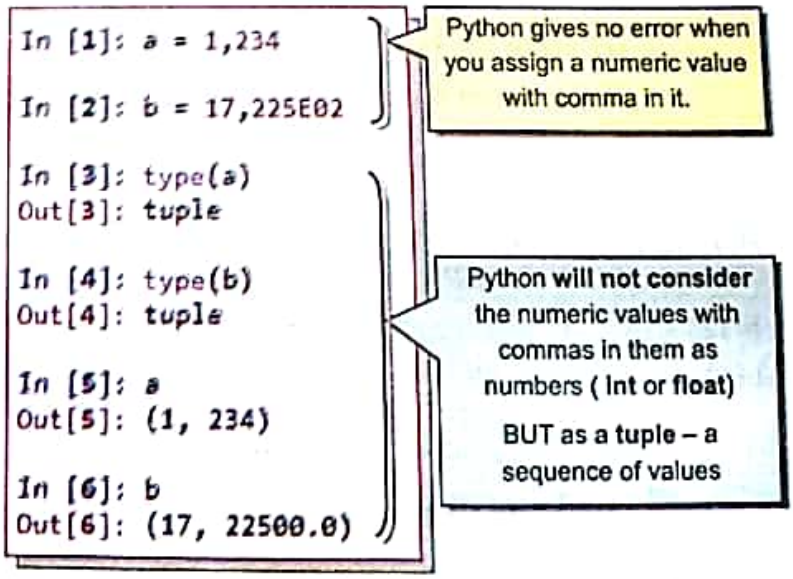
1.7E	(*No digit specified for exponent*)
0.17E2.3	(*Exponent cannot have fractional part*)
17,225E02	(*No comma allowed*) [Do read following discussion after it.]

Numeric values with commas are not considered **int** or **float** value, rather Python treats them as a tuple. A tuples is a special type in Python that stores a *sequence of values*. (You will learn about tuples in coming chapters – for now just understand a tuple as *a sequence of values only*.)

The last invalid example value given above (17,225e02) asks for a special mention here.

Any numeric value with a comma in its mantissa will not be considered a legal *floating point number*, BUT if **you assign this value, Python won't give you an error.** The reason being is that Python will not consider that as a *floating point value* rather a **tuple.** Carefully have a look at the adjacent figure that illustrates it.

We are not talking about Complex numbers here. These would be discussed later when the need arises.



2.3.3C Boolean Literals

A Boolean literal in Python is used to represent one of the two Boolean values *i.e.*, **True** (Boolean true) or **False** (Boolean false). A Boolean literal can either have value as *True* or as *False*.

NOTE

True and False are the only two Boolean literal values in Python. None literal represents absence of a value.

2.3.3D Special Literal None

Python has one special literal, which is **None**. The **None** literal is used to indicate absence of value. It is also used to indicate the end of lists in Python.

The **None** value in Python means *"There is no useful information"* or *"There's nothing here."* Python doesn't display anything when asked to display the value of a variable containing value as **None**. Printing with print statement, on the other hand, shows that the variable contains **None** (see figure here).

```
In [13]: Value1 = 10

In [14]: Value2 = None

In [15]: Value1
Out[15]: 10

In [16]: Value2

In [17]: print(Value2)
None

In [18]:
```

Displaying a variable containing **None** does not show anything. However, with *print()*, it shows the value contained as None.

Python supports literal collections also such as tuples and lists etc. But covering these here would make the discussion too complex for the beginning. So, we'll take them at a later time.

NOTE

Boolean literals True, False and special literal None are some built-in constants/literals of Python.

PriP BASICS ABOUT TOKENS — Progress In Python 2.1

Start Spyder IDE through Anaconda Navigator or any other IDE of your choice.

1. *In front of the Python prompt In[ ]: in IPythonConsole, type the following statements one by one, in the same order.*

   (*a*) Write the expected result and then write the actual result that Python returned.

   (*b*) Do write the reason(s) behind the result returned by Python.

Solved Sample

Statement to be typed	Expected result	Actual result	Reason
abc123 = 25 abc123 :	nothing 25 :	nothing 25 :	Python internally assigns the value to abc123 but shows nothing. Value of abc123 is displayed on screen. :

Please check the practical component-book – Progress in Computer Science with Python and fill it there in PriP 2.1 under Chapter 2 after practically doing it on the computer.

>>>❖<<<

Check Point

2.2

1. What are literals ? How many types of literals are available in Python ?
2. How many types of integer literals are allowed in Python ? How are they written ?
3. Why are characters \, ', " and tab typed using escape sequences ?
4. Which escape sequences represent the newline character and backspace character ? An escape sequence represents how many characters ?
5. What are string-literals in Python ? How many ways, can you create String literals in Python ? Are there any differences in them ?
6. What is meant by a floating-point literal in Python ? How many ways can a floating literal be represented into ?
7. Write the following real constants into exponent form :

   23.197, 7.214, 0.00005, 0.319
8. Write the following real constants into fractional form :

   0.13E04, 0.417E-04, 0.4E-5,

   .12E02, 12.E02
9. What are the two Boolean literals in Python ?
10. Name some built-in literals of Python.
11. Out of the following literals, determine their type whether decimal / octal / hexadecimal integer literal or a floating point literal in fractional or exponent form or string literal or other ?

    123, 0o124, 0xABC, 'abc', "ABC",

    12.34, 0.3E-01, '''ftghjkjl''',

    None, True, False
12. What kind of program elements are the following ?

    'a', 4.38925, "a", "main" ?
13. What will **var1** and **var2** store with statements : **var1** = 2,121E2 and **var2** = 0.2,121E2 ? What are the types of values stored in **var1** and **var2** ?

2.3.4 Operators

Operators are tokens that trigger some computation when applied to variables and other objects in an expression. Variables and objects to which the computation is applied, are called **operands**. So, an operator requires some *operands* to work upon.

The following list gives a brief description of the operators and their functions / operators, in details, will be covered in next chapter – *Data Handling*.

OPERATORS

Operators are tokens that trigger some computation / action when applied to variables and other objects in an expression.

Unary Operators

Unary operators are those operators that require one operand to operate upon. Following are some unary operators :

+	Unary plus
–	Unary minus
~	Bitwise complement
not	logical negation

Binary Operators

Binary operators are those operators that require two operands to operate upon. Following are some binary operators :

Arithmetic operators

+	Addition
–	Subtraction
*	Multiplication
/	Division
%	Remainder/ Modulus
**	exponent (raise to power)
//	Floor division

Bitwise operators

&	Bitwise AND
^	Bitwise exclusive OR (XOR)
|	Bitwise OR

Shift operators

<<	shift left
>>	shift right

Identity operators

is	is the identity same ?
is not	is the identity not same ?

Relational operators

<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not equal to

Assignment operators

=	Assignment
/=	Assign quotient
+=	Assign sum
*=	Assign product
%=	Assign remainder
-=	Assign difference
**=	Assign Exponent
//=	Assign Floor division

Logical operators

and	Logical AND
or	Logical OR

Membership operators

in	whether variable in sequence
not in	whether variable not in sequence

More about these operators you will learn in the due course. Giving descriptions and examples is not feasible and possible right here at the moment.

2.3.5 Punctuators

Punctuators are symbols that are used in programming languages to organize sentence structures, and indicate the rhythm and emphasis of expressions, statements, and program structure.

Most common punctuators of Python programming language are :

' " # \ ( ) [ ] { } @ , : . ` =

The usage of these punctuators will be discussed when the need arises along with normal topic discussions.

PUNCTUATORS

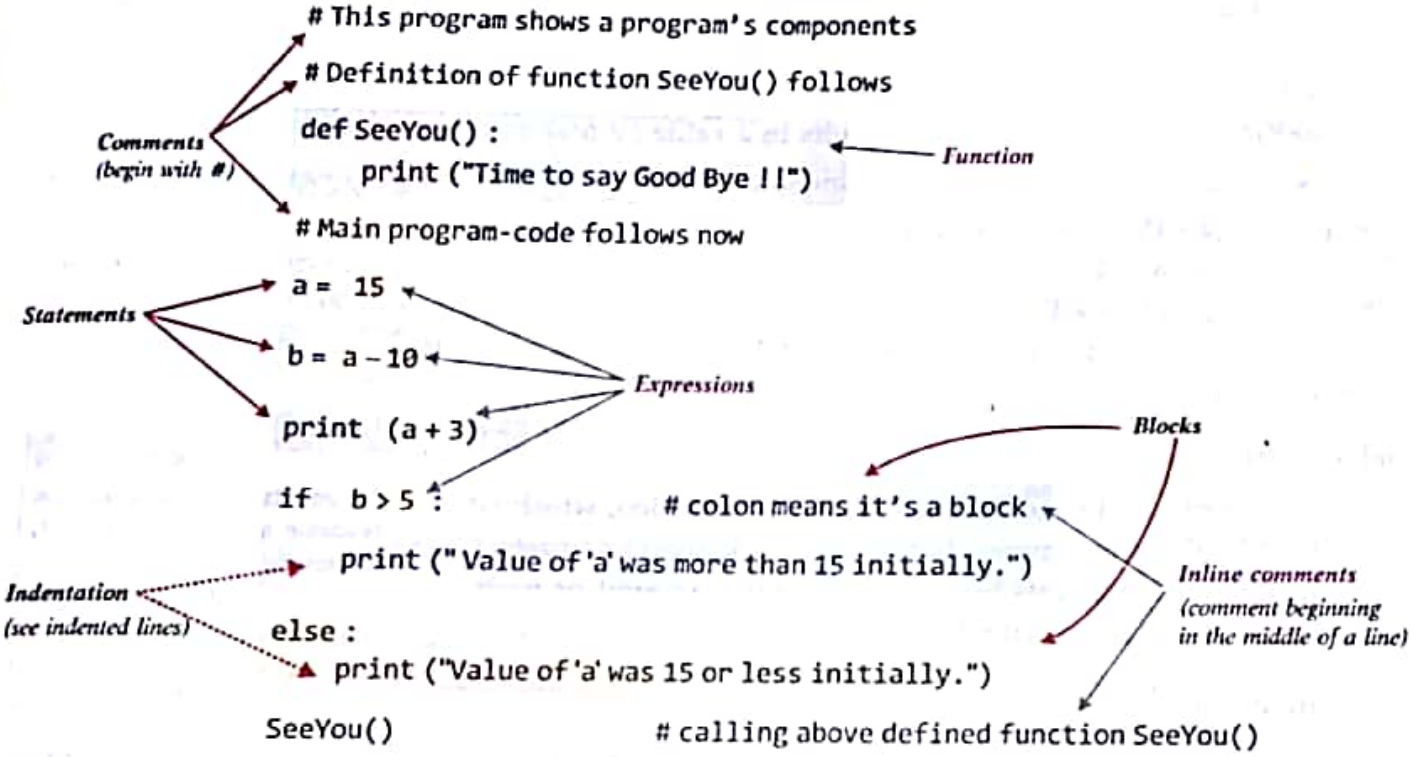
Punctuators are symbols that are used in programming languages to organize programming-sentence structures, and indicate the rhythm and emphasis of expressions, statements, and program structure.

LET US REVISE

- *A token is the smallest individual unit in a program.*
- *Python provides following tokens : keywords, identifiers (names), Values (literals), punctuators, operators and comments.*
- *A keyword is a reserved word carrying special meaning and purpose.*
- *Identifiers are the user-defined names for different parts of the program.*
- *In Python, an identifier may contain letters (a-z, A-Z), digits (0-9) and a symbol underscore (_). However, an identifier must begin with a letter or underscore ; all letters/digits in an identifier are significant.*
- *Literals are the fixed values.*
- *Python allows following literals : string literal, numeric (integer, floating-point literals, Boolean literals, special literal None and literal collections).*
- *Operators are tokens that trigger some computation / action when applied to variables and other objects in an expression.*
- *Punctuators are symbols used to organize programing- sentence structures and indicate the rhythm and emphasis of expressions, statements and program-structure.*

2.4 BAREBONES OF A PYTHON PROGRAM

Let us take our discussion further. Now we are going to talk about the basic structure of a Python program – what all it can contain. Before we proceed, have a look at following sample code. Look at the code and then proceed to the discussion that follows. Don't worry if the things are not clear to you right now. They'll become clear when the discussion proceeds.



As you can see that the above sample program contains various components like :

- expressions
- statements
- comments
- function
- blocks and indentation

Let us now discuss various components shown in above sample code.

(i) Expressions

An *expression* is any legal combination of symbols that *represents a value*. An expression represents something, which **Python evaluates** and which then produces a *value*.

Some examples of expressions are

```
15
2.9
```
} *expressions that are values only*

```
a + 5
(3 + 5) / 4
```
} *complex expressions that produce a value when evaluated.*

EXPRESSIONS

An ***expression*** is any legal combination of symbols that ***represents a value.***

Now from the above sample code, can you pick out all expressions ?

These are : 15, $a+10$, $a+3$, $b>5$

(ii) Statement

While an expression represents something, a statement is a programming instruction that does something *i.e.*, some action takes place.

Following are some examples of statements :

```
print ("Hello")   # this statement calls print function
if b > 5 :
    ⋮
```

STATEMENT

A statement is a programming instruction that does something *i.e.*, some action takes place.

While an expression is evaluated, a statement is executed *i.e.*, some action takes place. And it is **not necessary that a statement results in a value** ; it may or may not yield a value.

Some statements from the above sample code are :

```
a = 15
b = a - 10
print (a + 3)
if b < 5 :
    ⋮
```

NOTE

A statement executes and may or may not yield a value.

(iii) Comments

Comments are the additional readable information, which is read by the programmers but ignored by Python interpreter. In Python, comments begin with symbol # (Pound or hash character) and end with the end of physical line.

COMMENTS

Comments are the additional readable information to clarify the source code.

Comments in Python begin with symbol # and generally end with end of the physical line.

In the above code, you can see *four* comments :

(*i*) The physical lines beginning with # are the full line comments. There are *three* full line comments in the above program are :

```
# This program shows a program's components
# Definition of function SeeYou( ) follows
# Main program code follows now
```

(*ii*) The fourth comment is an *inline comment* as it starts in the middle of a physical line, after Python code (see below)

```
if b < 5 : # colon means it requires a block
```

NOTE

A *Physical line* is the one complete line that you see on a computer whereas a *logical line* is the one that Python sees as one full statement.

Multi-line Comments

What if you want to enter a *multi-line comment* or a *block comment* ?. You can enter a multi-line comment in Python code in *two* ways :

(*i*) Add a # symbol in the beginning of every physical line part of the multi-line comments, *e.g.*,

```
# Multi-line comments are useful for detailed additional information.
# Related to the program in question.
# It helps clarify certain important things.
```

(ii) Type comment as a triple-quoted multi-line string *e.g.*,

```
'''  Multi-line comments are useful for detailed additional
     information related to the program in question.
     It helps clarify certain important things
'''
```

This type of multi-line comment is also known as *docstring*. You can either use triple-apostrophe (''') or triple quotes (""") to write *docstrings*. The docstrings are very useful in documentation – and you'll learn about their usage later.

NOTE

Comments enclosed in triple quotes (" " ") or triple apostrophe (' ' ') are called docstrings.

(iv) Functions

A function is a code that has a name and it can be reused (executed again) by specifying its name in the program, where needed.

In the above sample program, there is one function namely **SeeYou( )**. The statements indented below its **def** statement are part of the function. [All statements indented at the same level below *def SeeYou( )* are part of *SeeYou( )*.] This function is executed in main code through following statement (Refer to sample program code given above)

FUNCTIONS

A function is a code that has a name and it can be reused (executed again) by specifying its name in the program, where needed .

```
SeeYou()      # function-call statement
```

Calling of a function becomes a statement *e.g.*, **print** is a function but when you call **print( )** to print something, then that function call becomes a statement.

For now, only this much introduction of functions is sufficient. You will learn about functions in details in Class 12.

(v) Blocks and Indentation

Sometimes a group of statements is part of another statement or function. Such a group of one or more statements is called block or code-block or suite. *For example,*

Four spaces together mark the next indent-level

```
if b < 5 :
    print ("Value of 'b' is less than 5.")
    print ("Thank you.")
```

This is a block with all its statements at same indentation level.

Many languages such as C, C++, Java etc., use symbols like curly brackets to show blocks but Python does not use any symbol for it, rather it uses indentation.

Consider the following example :

```
if b < a :
    tmp = a
    a = b
    b = tmp
print ("Thank you")
```

This is a block, part of if statement. Notice, all statements in same block have same indentation level.

This statement is not part of if's block as it is at different indentation level.

BLOCK OR CODE-BLOCK OR SUITE

A group of statements which are part of another statement or a function are called *block* or *code-block* or *suite* in Python.

A group of individual statements which make a single *code-block* is also called a suite in Python. Consider some more examples showing indentation to create blocks :

```
def check() :
    c = a + b
    if c < 50 :
        print ('Less than 50')
        b = b * 2
        a = a + 10
    else :
        print ('>= 50')
        a = a * 2
        b = b + 10
```

Block inside function check( )

Two different indentation-levels inside this code.

Block / suite inside if statement

Block / suite inside else statement

NOTE

Python uses indentation to create blocks of code. Statements at same indentation level are part of same block/suite.

Statements requiring suite/ code-block have a colon (:) at their end.

You cannot unnecessarily indent a statement ; Python will raise error for that.

Python Style Rules and Conventions

While working in Python, one should keep in mind certain style rules and conventions. In the following lines, we are giving some very elementary and basic style rules :

Statement Termination Python does not use any symbol to terminate a statement. When you end a physical code-line by pressing Enter key, the statement is considered terminated by default.

Maximum Line Length Line length should be maximum 79 characters.

Lines and Indentation Blocks of code are denoted by line indentation, which is enforced through *4 spaces* (not tabs) per indentation level.

Blank Lines Use *two* blank lines between top-level definitions, *one* blank line between method/function definitions. Functions and methods should be separated with *two* blank lines and Class definitions with *three* blank lines.

Avoid multiple statements on one line Although you can combine more than one statements in one line using symbol semicolon (;) between two statements, but it is not recommended.

Whitespace You should always have whitespace around operators and after punctuation but not with parentheses. Python considers these 6 characters as whitespace : ' ' (space), '\n' (newline), '\t' (horizontal tab), '\v' (vertical tab), '\f' (formfeed) and '\r' (carriage return)

Case Sensitive Python is case sensitive, so case of statements is very important. Be careful while typing code and identifier-names.

Docstring Convention Conventionally triple double quotes (""") are used for docstrings.

Identifier Naming You may use underscores to separate words in an identifier *e.g.*, **loan_amount** or use *CamelCase* by capitalizing first letter of the each word *e.g.*, LoanAmount or loanAmount

Check Point 2.3

1. **What is an expression in Python ?**
2. **What is a statement in Python ? How is a statement different from expression ?**
3. **What is a comment ? In how many ways can you create comments in Python ?**
4. **How would you create multi-line comment in Python ?**
5. **What is the difference between full-line comment and inline comment ?**
6. **What is a block or suite in Python ? How is indentation related to it ?**

⋮

This is another program with different components

⋮

```
def First5Multiples( ) :
    ⋮
# main code
    ⋮
```

Fill the appropriate components of program from the above code.

⋮

Please check the practical component-book – Progress in Computer Science with Python and fill it there in PriP 2.2 under Chapter 2 after practically doing it on the computer.

>>>❖<<<

2.5 VARIABLES AND ASSIGNMENTS

A variable in Python represents named location that refers to a value and whose values can be used and processed during program run. For instance, to store name of a student and marks of a student during a program run, we require some labels to refer to these *marks* so that these can be distinguished easily. *Variables*, called as *symbolic variables*, serve the purpose. The variables are called symbolic variables because these are named labels. *For instance*, the following statement creates a variable namely *marks* of Numeric type :

```
marks = 70
```

VARIABLES

Named labels, whose values can be used and processed during program run, are called *Variables*.

2.5.1 Creating a Variable

Recall the statement we used just now to create the variable marks :

```
marks = 70
```

As you can see, that we just assigned the value of numeric type to an identifier name and Python created the variable of the type similar to the type of value assigned. In short, after the above statement, we can say that *marks* is a *numeric* variable.

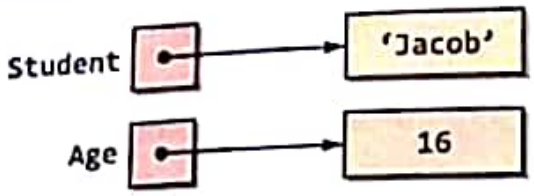
So, creating variables was just that simple only ? Yes, you are right. In Python, to create a variable, just assign to its name the value of appropriate type. *For example*, to create a variable namely ***Student*** to hold student's name and variable *age* to hold student's age, you just need to write somewhat similar to what is shown below :

```
Student = 'Jacob'
Age = 16
```

NOTE

Python variables are created by assigning value of desired type to them, *e.g.*, to create a numeric variable, assign a numeric value to *variable_name* ; to create a string variable , assign a string value to *variable_name*, and so on.

Python will internally create labels referring to these values as shown below :



Isn't it simple?? ☺ Okay, let's now create some more variables.

```
TrainNo = 'T#1234'      # variable created of String type
balance = 23456.75      # variable created of Numeric(floating point) type
rollNo = 105            # variable create of Numeric(integer) type
```

Same way, you can create as many variables as you need for your program.

IMPORTANT – Variables are Not Storage Containers in Python

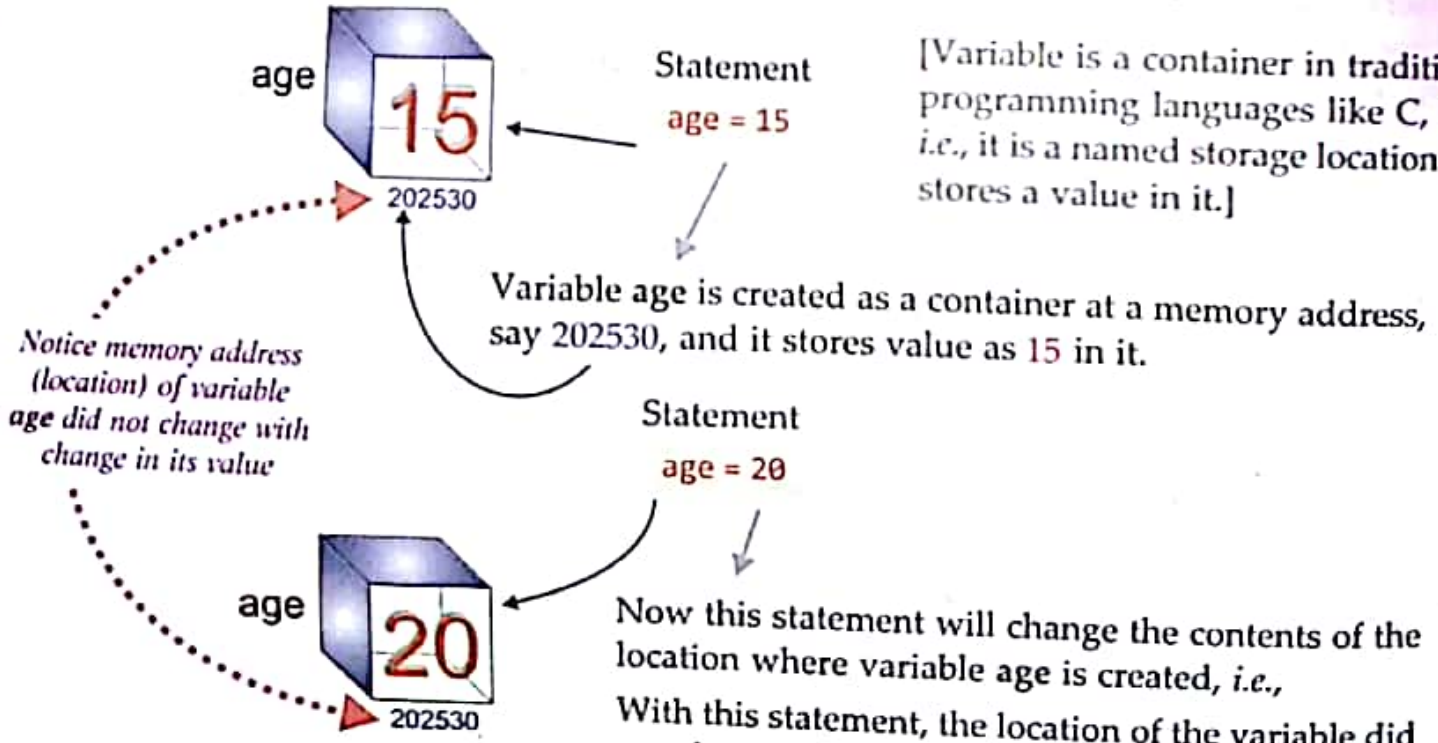
If you have an earlier exposure to programming, you must be having an idea of variables. BUT PYTHON VARIABLES ARE NOT CREATED IN THE FORM MOST OTHER PROGRAMMING LANGUAGES DO. Most programming languages create variables as storage containers *e.g.*,

Consider this :

```
age = 15
age = 20
```

Firstly value 15 is assigned to variable age and then value 20 is assigned to it.

Traditional Programming Languages' Variables



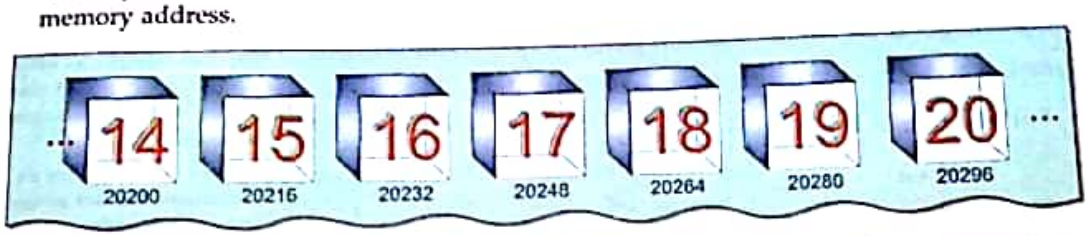
This is how traditionally variables were created in programming languages like C, C++, Java etc.

Python's Handling of Variables

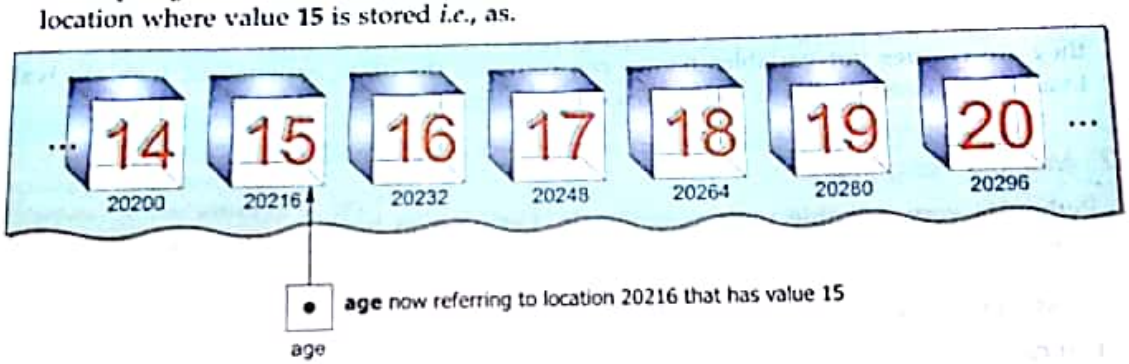
BUT PYTHON DOES THIS DIFFERENTLY

Let us see how Python will do it.

Memory has literals/values at defined memory locations, and each memory location has a memory address.



When you give statement **age = 15**, variable **age** will be created as a label pointing to memory location where value **15** is stored *i.e.*, as.



And when you give statement age = 20, the label age will not be having the same location as earlier. It will now refer to value 20, which is at different location *i.e.*,

So this time memory location of variable age's value is changed.

Thus variables in Python do not have fixed locations unlike other programming languages. The location they refer to changes everytime their values change (**This rule is not for all types of variables, though**). It will become clear to you in the next chapter, where we talk about *Mutable and Immutable types*.

Lvalues and Rvalues

Broadly *lvalue* and *rvalue* can be thought of as :

lvalue : expressions that can come on the lhs (left hand side) of an assignment.

rvalue : expressions that can come on the rhs (right hand side) of an assignment.

e.g., you can say that

```
a = 20
b = 10
```

But you cannot say

```
20 = a
or  10 = b
or  a * 2 = b
```

LVALUES AND RVALUES

Lvalues are the objects to which you can assign a value or expression. Lvalues can come on lhs or rhs of an assignment statement.

Rvalues are the literals and expressions that are assigned to lvalues. Rvalues can come on rhs of an assignment statement.

The literals or the expressions that evaluate a value cannot come on lhs of an assignment hence they are **rvalues** but variables names can come on lhs of an assignment, they are **lvalues**. Lvalues can come on lhs as well as rhs of an assignment.

NOTE

In Python, assigning a value to a variable means, variable's label is referring to that value.

2.5.2 Multiple Assignments

Python is very versatile with assignments. Let's see in how many different ways, you can use assignments in Python :

1. Assigning same value to multiple variables

You can assign same value to multiple variables in a single statement, *e.g.*,

```
a = b = c = 10
```

It will assign value 10 to all three variables *a*, *b*, *c*. That is, all three labels *a*, *b*, *c* will refer to same location with value 10.

2. Assigning multiple values to multiple variables

You can even assign multiple values to multiple variables in single statement, *e.g.*,

```
x, y, z = 10, 20, 30
```

It will assign the values **order wise**, *i.e.*, first variable is given first value, second variable the second value and so on. That means, above statement will assign value 10 to *x*, 20 to *y* and 30 to *z*.

This style of assigning values is very useful and compact. *For example*, consider the code given below :

```
x, y = 25, 50
print (x, y)
```

It will print result as

```
25  50
```

Because x is having value 25 and y is having 50. Now, if you want to swap values of x and y, you just need to write :

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

Now the result will be

```
50  25
```

Because this time the values have been swapped and x is having value 25 and y is having 50.

While assigning values through multiple assignments, please remember that Python first evaluates the RHS (right hand side) expression(s) and then assigns them to LHS, *e.g.*,

```
a, b, c = 5, 10, 7                  # statement1
b, c, a = a + 1, b + 2, c - 1       # statement2
print (a, b, c)
```

- Statement1 assigns 5, 10 and 7 to *a*, *b* and *c* respectively.
- Statement2 will first evaluate RHS *i.e.*, $a + 1$, $b + 2$, $c - 1$ which will yield

```
5 + 1, 10 + 2, 7 - 1 = 6, 12, 6
```

Then it will make the statement (by replacing the evaluated result of RHS) as :

```
b, c, a = 6, 12, 6
```

Thus, $b = 6$, $c = 12$ and $a = 6$

- The third statement **print (a, b, c)** will print

```
6  6  12
```

Isn't this easy ? Now can you guess the output of following code fragment[2]

```
p, q = 3, 5
q, r = p - 2, p + 2
print (p, q, r)
```

Please note the expressions separated with commas are **evaluated from left to right and assigned in same order** *e.g.*,

```
x = 10
y, y = x + 2, x + 5
```

will evaluate to following (after evaluating expressions on rhs of = operator)

```
y, y = 12, 15
```

i.e., firstly it will assign *first RHS value* to *first LHS variable i.e.*,

```
y = 12
```

then it will assign second RHS value to second LHS variable *i.e.*,

```
y = 15
```

So if you print *y* after this statement *y* will contain 15.

2. **The output should be : 3 1 5**

Now, consider following code and guess the output :

```
x, x = 20, 30
y, y = x + 10, x + 20
print (x, y)
```

Well, it will print the output as

```
30  50
```

Now you know why ?☺

2.5.3 Variable Definition

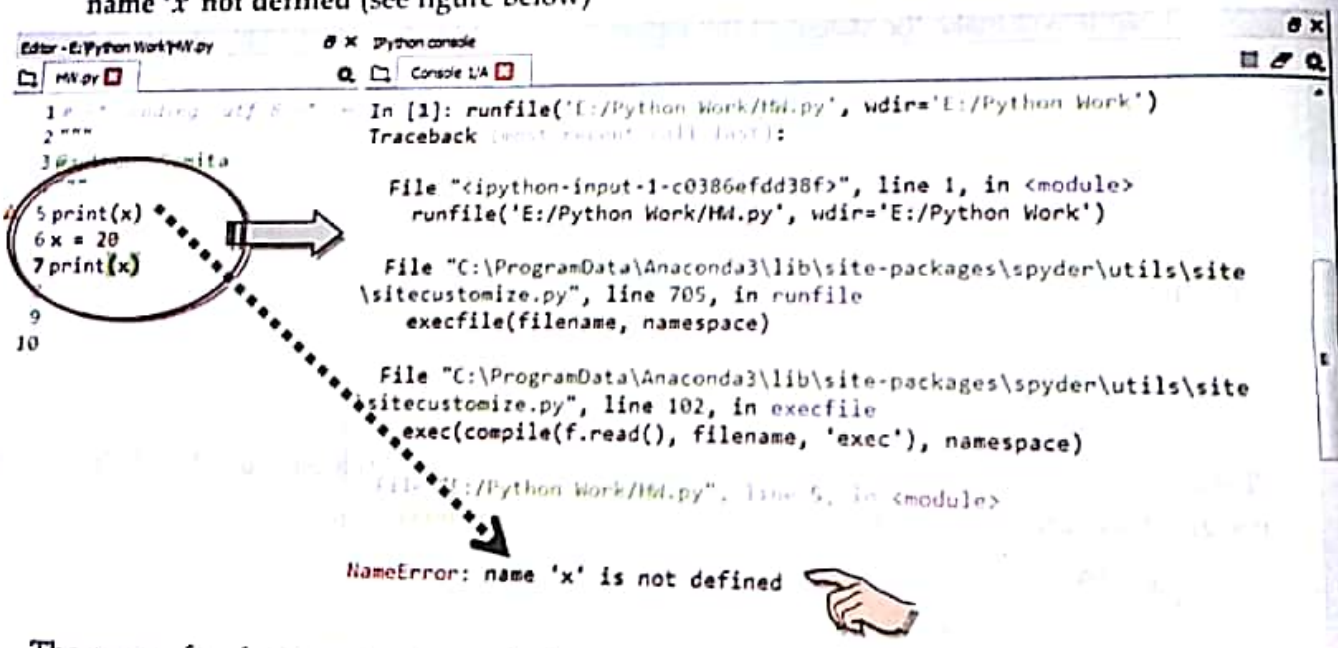
So, you see that in Python, a variable is *created* when you first assign a value to it.

It also means that a **variable is not created until some value is assigned to it.**

To understand it, consider the following code fragment. Try running it in script mode :

```
print (x)
x = 20
print (x)
```

when you run the above code, it will produce an error for the first statement (line 1) only – name '*x*' not defined (see figure below)



The reason for above error is implicit. As you know that a *variable is not created until some value is assigned to it*. So, variable *x* is not created and yet being printed in line 1. Printing/using an uncreated (undefined) variable results into error.

So, to correct the above code, you need to first assign something to *x* before using it in a statement, somewhat like :

```
x = 0                    # variable x created now
print (x)
x = 20
print (x)
```

Now the above code will execute without any error.

IMPORTANT

A variable is defined only when you assign some value to it. Using an undefined variable in an expression/statement causes an error called Name Error.

2.5.4 Dynamic Typing

In Python, as you have learnt, a variable is defined by assigning to it some value (of a particular type such as numeric, string etc.)

For instance, after the statement :

```
X = 10
```

We can say that variable *x* is referring to a value of integer type.

Later in your program, if you reassign a value of some other type to variable *x*, Python will not complain (no error will be raised), *e.g.,*

```
X = 10
print (X)
X = "Hello World"
print (X)
```

DYNAMIC TYPING

A variable pointing to a value of a certain type, can be made to point to a value/object of different type. This is called *Dynamic Typing*.

Above code will yield the output as :

```
10
Hello World
```

So, you can think of a Python variable as ***labels*** associated with objects (literal values in our case here) ; with dynamic typing, Python makes the label refer to new value (Fig. 2.1). Following figure illustrates it.

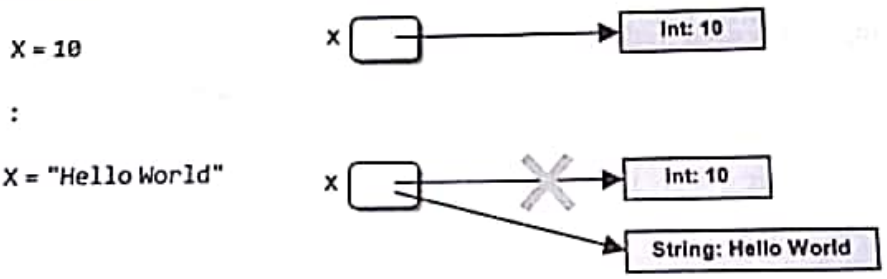


Figure 2.1 Dynamic typing in Python variables.

As you can see in Fig. 2.1, variable *X* is first pointing to/referring to an integer value **10** and then to a string value "**Hello world**".

Please note here that variable *X* does not have a type but the value it points to does have a type. So you can make a variable point to a value of different type by reassigning a value of that type ; Python will not raise any error. This is called **Dynamic Typing** feature of Python.

Caution with Dynamic Typing

Although Python is comfortable with changing types of a variable, the programmer is responsible for ensuring right types for certain type of operations. *For example,*

```
X = 10
Y = 0
Y = X/2
X = 'Day'
Y = X/2
```

legal, because two integers can be used in divide operation

Python is comfortable with dynamic typing

ERROR!! a string cannot be divided.

So as programmer, you need to ensure that variables with right type of values should be used in expressions.

If you want to determine the type of a variable *i.e.*, what type of value does it point to ?, you can use **type( )** in following manner :

```
type (<variable name>)
```

For instance, consider the following sequence of commands that uses **type( )** *three* times :

```
>>> a = 10
>>> type(a)
<class 'int'>
>>> a = 20.5
>>> type(a)
<class 'float'>
>>> a = "hello"
>>> type(a)
<class 'str'>
>>>
```

The type returned as int (integer)

The type returned as float (floating point number)

The type returned as str (string)

NOTE

Dynamic typing is different from Static Typing In Static Typing, a data type is attached with a variable when it is defined first and it is fixed. That is, data type of a variable cannot be changed in static typing whereas there is no such restriction in dynamic typing.

Programming languages like C, C++ support static typing.

NOTE

Use type (<object name>) function to determine the type of an object. Object can be a variable or a literal etc.

2.6 SIMPLE INPUT AND OUTPUT

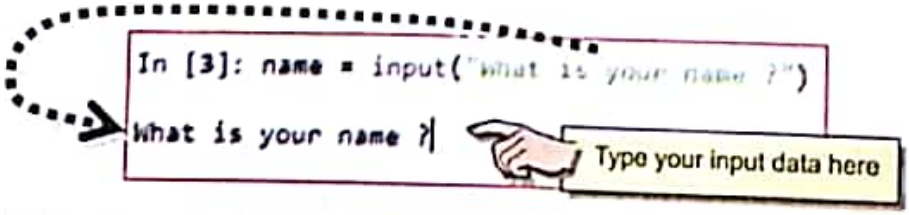
In Python 3.x, to get input from user interactively, you can use built-in function **input( )**. The function **input( )** is used in the following manner :

```
variable_to_hold_the_value = input (<prompt to be displayed>)
```

For example,

```
name = input ('What is your name ?')
```

The above statement will display the prompt as :



in front of which you can type the name. The value that you type in front of the displayed prompt will be assigned to given variable, **name** in above case. Now consider the following command sequence :

```
In[ ] : name = input("Enter Your name :" )
Enter Your name : Simar
In[ ] : age = input( "Enter Your age :" )
Enter Your age : 16
In[ ] : name
Out[ ] : 'Simar'
In[ ] : age
Out[ ] : '16'
```

```
In [5]: name
Out[5]: ' Simar'

In [6]: age
Out[6]: '16'
```

NOTE

Please note the **In[ ] :** is the prompt of IPython shell (used with Spyder IDE) and **>>>** is shown as general prompt of a Python shell.

You can use any of the available IDEs of Python.

In above code, we used **input( )** function to input *two* values *name* and *age*.

But **input( )** has a property, which you must be aware of. The **input( ) function always returns a value of *String* type.** Notice carefully in above code – while displaying both values name and age, Python has enclosed both the values in quotes *i.e.*, as **'Simar'** and **'16'**. This is because whatever value you enter through **input( )** function is treated as a String.

NOTE

The Input( ) function always returns a value of String type

Now what are its consequences, especially when you are entering a number? In order to understand this, consider the following code :

```
In [8]: age = input("What is your age ? ")

What is your age ? 16

In [9]: age + 1
Traceback (most recent call last):

  File "<ipython-input-9-eac256a954eb>", line 1, in <module>
    age + 1

TypeError: must be str, not int
```

Notice the error raised when you try to add a number to variable **age** which was given a value using ***input( )*** function.

See, Python raised an error when you tried to add 1 to *age* whose value you entered as 16. The reason is obvious and clear – **Python cannot add on integer to a string.** Since variable ***age*** received value 16 through **input( )**, it actually had **'16'** in it *i.e.*, **string value '16'** ; thus you cannot add an integer to it.

You can check yourselves the type of variable age whose value you entered through **input( )** function :

```
In [10]: age = input("What is your age ? ")

What is your age ? 16

In [11]: age
Out[11]: '16'

In [12]: type(age)
Out[12]: str
```

See it shows the type of variable *age* as str

NOTE

When you try to perform an operation on a data type not suitable for it (e.g., dividing or multiplying a string), Python raises an error called TypeError.

2.6.1 Reading Numbers

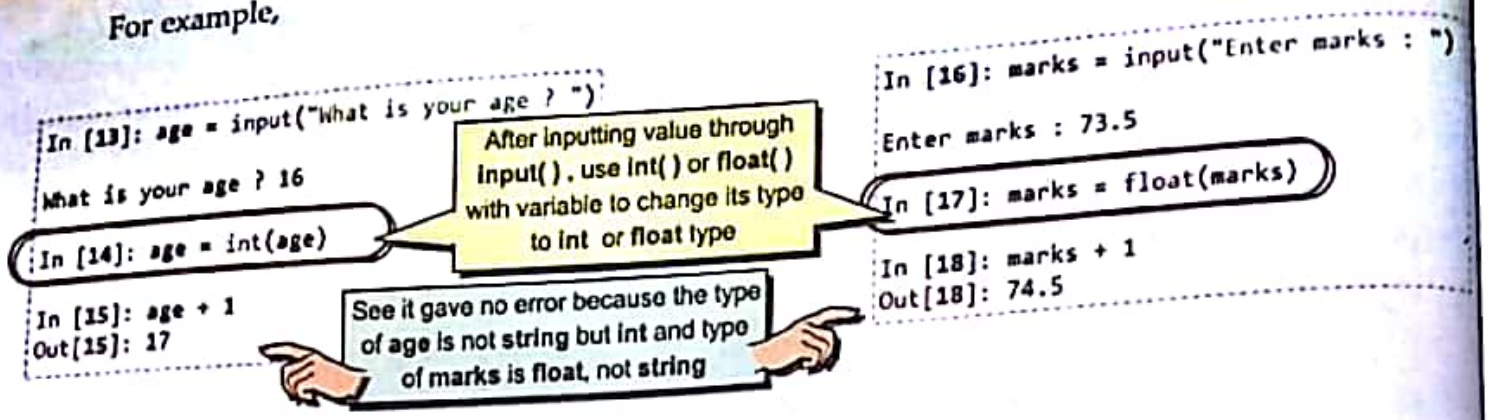
String values cannot be used for arithmetic or other numeric operations. For these operations, you need to have values of numeric types (integer or float).

But what you would you do if you have to read numbers (**int** or **float**) ? The function **input( )** returns the entered value in ***string*** type only. Then what is the way out ?

Don't worry. Python offers two functions **int( )** and **float( )** to be used with **input( )** to convert the values received through **input( )** into **int** and **float** types. You can :

- Read in the value using **input( )** function.
- And then use **int( )** or **float( )** function with the *read value* to change the type of input value to **int** or **float** respectively.

For example,

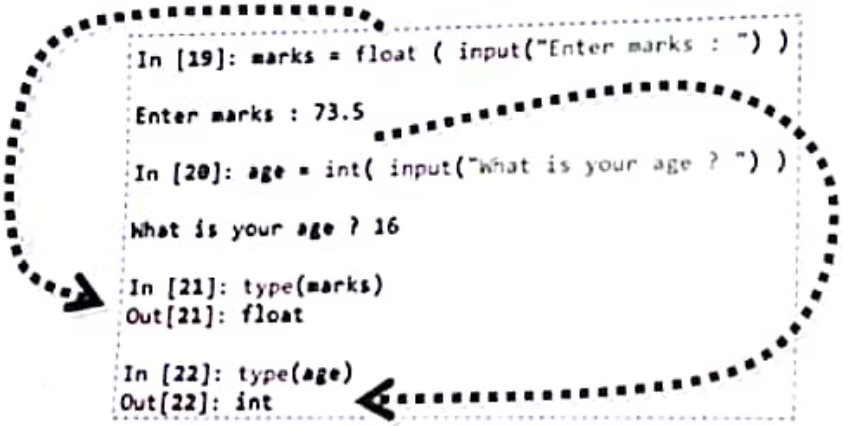


You can also combine these two steps in a single step too, *i.e.*, as :

```
<variable_name> = int( input( <prompt string>) )
```

Or

```
<variable_name> = float( input( <prompt string>) )
```



Function **int( )** around **input( )** converts the read value into **int** type and function **float( )** around **input( )** function converts the read value into **float** type.

After using **int( )** and **float( )** functions with **input( )**, you can check yourselves using **type( )** function that the type of read value is now **int** or **float** now (see above)

NOTE

There is another function **eval( )**, which can also be used with **input( )**, but covering it here would derail our basic discussion. Hence, we have talked about **eval( )** in a later chapter.

2.6.1A Possible Errors When Reading Numeric Values

If you are planning to input an integer or floating-point number using **input( )** inside **int( )** or **float( )** such as shown below :

```
age = int( input('Enter Your age :') )
```

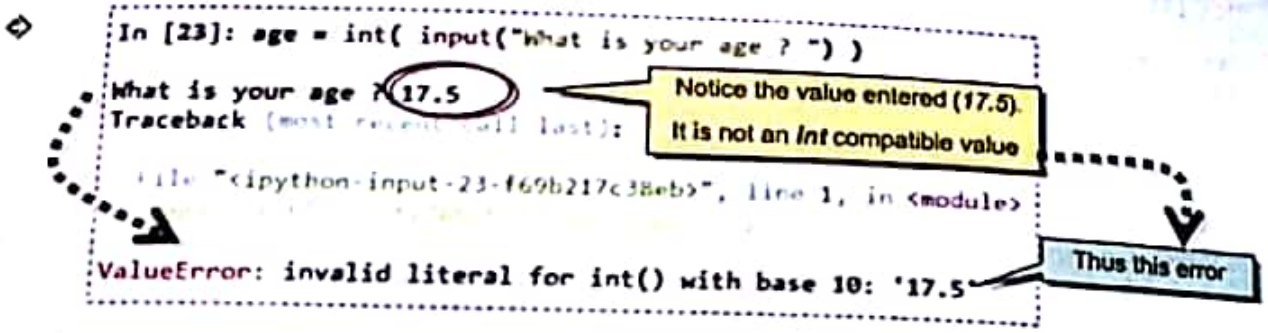
or

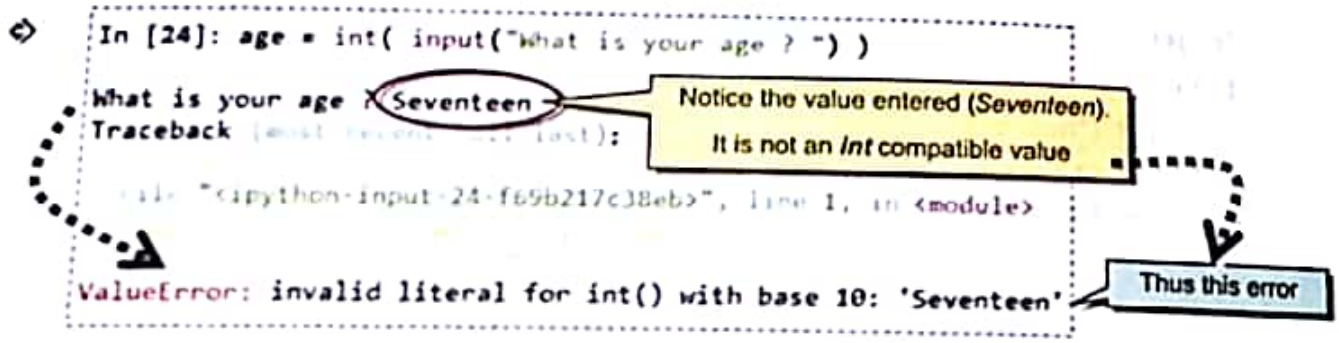
```
percentage = float( input('Enter your percentage :' )
```

Then you must ensure that the value you are entering must be in a form that is easily convertible to the target type.

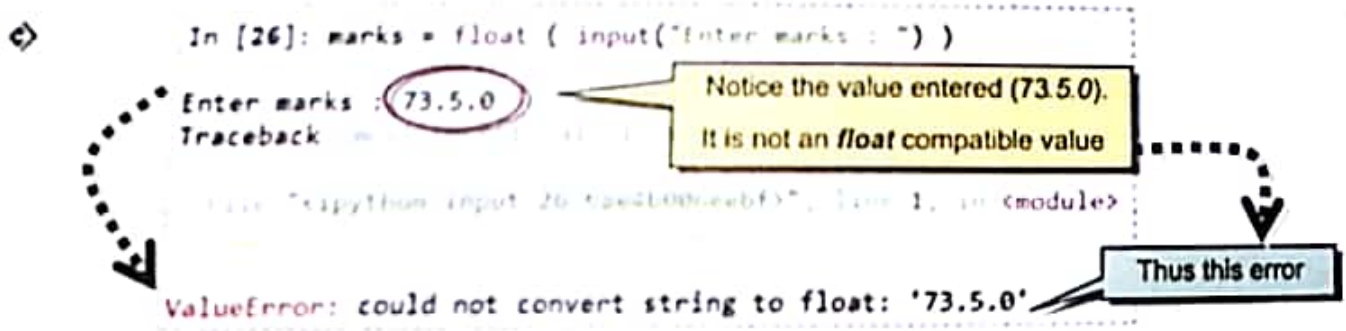
In other words :

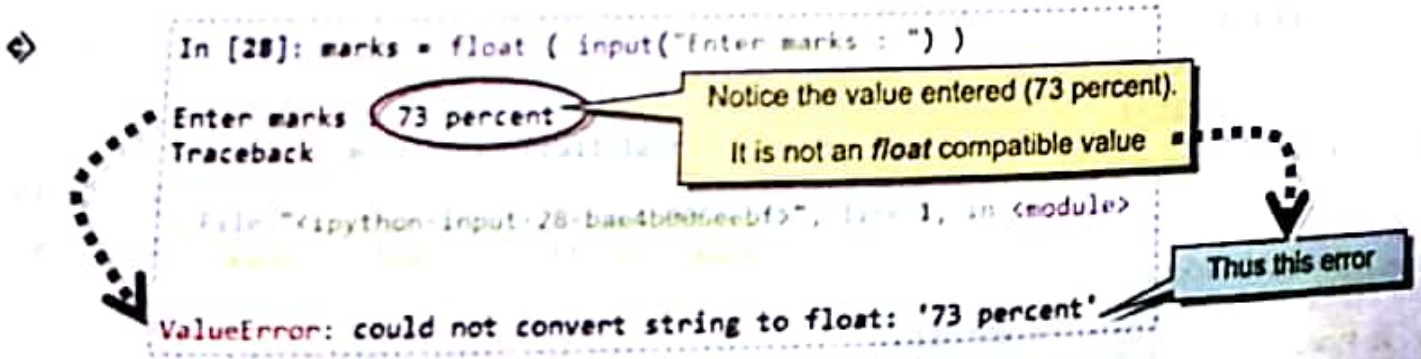
(i) while inputting integer values using **int( )** with **input( )**, make sure that the value being entered must be **int** type compatible. Carefully have a look at example code given below to understand it :





(ii) while inputting floating point values using **float( )** with **input( )**, make sure that the value being entered must be **float** type compatible. Carefully have a look at example code given below to understand it :





Please note that values like 73 or 73. or .73 are all float convertible, hence Python will be able to convert them to float and no error shall be reported if you enter such values.

```
In [25]: marks = float ( input("Enter marks : ") )
Enter marks : 73
In [27]: marks = float ( input("Enter marks : ") )
Enter marks : 73.
```

Values like .73 or 73. or 73 etc. can be easily converted into float, hence Python report no error if you enter such values with float( ) used with Input( ) .

TIP

While entering numeric values through Input( ) along with Int( ) /float( ), make sure that you enter values that are convertible to the target type otherwise Python will raise an error.

2.6.2 Output Through print Statement

The print( ) function of Python 3.x is a way to send output to standard output device, which is normally a monitor. The simplified syntax[3] to use print( ) function is as follows :

```
print(*objects, [ sep = ' ' or <separator-string> end = '\n' or <end-string> ])
```

*objects means it can be one or multiple comma separated *objects* to be printed.

Let us consider some simple examples first :

```
print ("hello")                          # a string
print (17.5)                             # a number
print (3.14159*(r*r))                    # the result of a calculation, which will
                                         # be performed by Python and then printed
                                         # out (assuming that some number has been
                                         # assigned to the variable r)
print ("I\'m", 12 + 5, "years old.")     # multiple comma separated expressions
```

Consider some examples with outputs :

Example statement 1 :

```
print ("Python is wonderful.")
```

will print the output as:

```
Python is wonderful.
```

Example statement 2 :

```
print ("Sum of 2 and 3 is", 2 + 3)
```

will print the output as follows (2+3 is evaluated and its result is printed as 5):

```
Sum of 2 and 3 is 5
```

Example statement 3 :

```
a = 25
print ("Double of", a, "is ", a*2)
```

will print the output as follows (a*2 is evaluated and its result is printed) :

```
Double of 25 is 50
```

Now consider some more print statement examples :

```
print (obj)
print (obj1, obj2, obj3)
print( )
print ('Object1 has more value than Object2')
print (obj1, 'is lesser than', obj2)
```

The output of these print( ) functions, you'll be able to determine if the values of variables *obj*, *obj1*, *obj2* and *obj3* are known to you. (A print( ) without any value or name or expression prints a blank line.)

3. In syntax, elements in square brackets mean that they are optional.

Features of print statement

The print statement has a number of features :

- it auto-converts the items to strings *i.e.*, if you are printing a numeric value, it will automatically convert it into equivalent string and print it ; for numeric expressions, it first evaluates them and then converts the result to string, before printing (as it did in example statement 2 above)

IMPORTANT : With print( ), the objects/items that you give, must be convertible to string type. Covering it here with examples may derail our discussion so we have given examples of using *print( )* with different types of arguments in **Appendix B**. Please do refer to Appendix B.

- it inserts spaces between items automatically because the default value of *sep* argument is space(' '). The *sep* argument specifies the separator character. The print( ) automatically adds the *sep* character between the items/objects being printed in a line. If you do not give any value for *sep*, then by default the print( ) will add a space in between the items when printing. Consider this code :

```
print ("My", "name", "is", "Amit.")
```

Four different string objects with no space in them are being printed.

will print

```
My name is Amit.
```

But the output line has automatically spaces inserted in between them because default sep character is a space.

You can change the value of separator character with sep argument of print( ) as per this :

The code :

```
print ("My", "name", "is", "Amit.", sep = '...' )
```

will print

```
My...name...is...Amit.
```

This time the print( ) separated the items with given sep character, which is '...'

- it appends a newline character at the end of the line unless you give your own end argument. Consider the code given below :

```
print ("My name is Amit.")
print("I am 16 years old")
```

It will produce output as :

```
My name is Amit.
I am 16 years old
```

NOTE

A print( ) function without any value or name or expression prints a blank line.

So, a *print( )* statement appended a newline at the end of objects it printed, *i.e.*, in above code :

```
My name is Amit. ↵ or \n
I am 16 years old.
```

Python automatically added a newline character in the end of a line printed so that the next print( ) prints from the next line

The print( ) works this way only when you have not specified any *end* argument with it because by default print( ) takes value for end argument as '\n' – the *newline character*[4].

4. A newline is a character used to represent the end of a line of text and the beginning of a new line.

If you explicitly give an end argument with a print( ) function then the print( ) will print the line and end it with the string specified with the end argument, *e.g.*, the code

```
print("My name is Amit. ", end = '$')
print("I am 16 years old. ")
```

will print output as :

```
My name is Amit. $I am 16 years old.
```

This time the print( ) ended the line with given end character, which is '$' here

So the *end* argument determines the *end* character that will be printed at the end of print line.

Code fragment 1

```
a, b = 20, 30
print ("a =", a, end = ' ')
print ("b =", b)
```

Notice first print statement has end set as a space

NOTE

In print( ) function, the default value of *end* argument is newline character ('\n') and of *sep* argument, it is space (' ').

Now the output produced will be like :

```
a = 20 b = 30
```

This space is because of end = ' ' in print( )

The reason for above output is quite clear. Since there is end character given as a space (*i.e.*, end = ' ') in first print statement, the newline ('\n') character is not appended at the end of output generated by first print statement. Thus the output-position-cursor stays on the same line. Hence the output of second print statement appears in the same line. Now can you predict the output of following code fragment ?

```
Name = 'Enthusiast'
print ("Hello", end = ' ')
print (Name)
print ("How do you find Python ?")
```

Well, you guessed it right. ☺ It is :

```
Hello Enthusiast
How do you find Python ?
```

In Python you can break any statement by putting a \ is the end and pressing Enter key, then completing the statement in next line. For example, following statement is perfectly right.

```
print ("Hello", \
  end = ' ')
```

The backslash at the end means that the statement is still continuing in next line

Now consider following sample programs

Check Point 2.4

1. What is a variable ?
2. Why is a variable called symbolic variable ?
3. Create variables for the following
   (*i*) to hold a train number
   (*ii*) to hold the name of a subject
   (*iii*) to hold balance amount in a bank account
   (*iv*) to hold a phone number
4. What do you mean by dynamic typing of a variable ? What is the caution you must take care of ?
5. What happens when you try to access the value of an undefined variable ?
6. What is wrong with the following statement ?

```
Number = input("Number")
Sqr = Number*Number
```

7. Write Python code to obtain the balance amount.
8. Write code to obtain fee amount and then calculate fee hike as 10% of fees (i.e., fees × 0.10).

Program 2.1 *Program to obtain three numbers and print their sum.*

```
# to input 3 numbers and print their sum
num1 = int( input("Enter number 1 : "))
num2 = int( input("Enter number 2 : "))
num3 = int( input("Enter number 3 : "))
Sum = num1 + num2 + num3
print("Three numbers are : ", num1, num2, num3)
print("Sum is : ", Sum)
```

The output produced by above program is as shown below :

```
Enter number 1 : 7
Enter number 2 : 3
Enter number 3 : 13
Three numbers are :  7 3 13
Sum is :  23
```

Program 2.2 *Program to obtain length and breadth of a rectangle and calculate its area.*

```
# to input length and breadth of a rectangle and calculate its area
length = float( input("Enter length of the rectangle : "))
breadth = float( input("Enter breadth of the rectangle : "))

area = length * breadth

print ("Rectangle specifications ")
print ("Length = ", length, end = ' ')
print ("breadth = ", breadth)
print ("Area = ", area)
```

The output produced by above program is as shown below :

```
Enter length of the rectangle : 8.75
Enter breadth of the rectangle : 35.0

Rectangle specifications

Length =  8.75 Breadth =  35.0
Area =  306.25
```

Program 2.3 *Program to calculate BMI (Body Mass Index) of a person.*

Body Mass Index is a simple calculation using a person's height and weight.
The formula is **BMI = kg/m²** where *kg* is a person's weight in kilograms and m^2 is their height in *metres squared.*

```
# to calcu;ate BMI = kg /m square
weight_in_kg = float(input("Enter weight in kg : "))
```

```
height_in_meter = float(input("Enter height in meters : "))
bmi = weight_in_kg / ( height_in_meter * height_in_meter)
print("BMI is : ", bmi)
```

The output produced by above program is as shown below :

```
Enter weight in kg : 66
Enter height in meters : 1.6
BMI is :  25.781249999999996
```

VARIABLES, SIMPLE I/O — Progress In Python 2.3

Start a Python IDE of your choice

1. *Click File → New File... and type the following code into it :*

```
myNumber = 10
print (myNumber + 1)          # output statement 1
print (myNumber)              # output statement 2
```

⋮

Please check the practical component-book – Progress in Computer Science with Python and fill it there in PriP 2.3 under Chapter 2 after practically doing it on the computer.

>>>❖<<<

LET US REVISE

- *A Python program can contain various components like expressions, statements, comments, functions, blocks and indentation.*
- *An expression is a legal combination of symbols that represents a value.*
- *A statement is a programing instruction.*
- *Comments are non-executable, additional information added in program for readability.*
- *In Python, comments begin with a # character.*
- *Comments can be single-line comment, multi-line comments and inline comments.*
- *Function is a named code that can be reused with a program.*
- *A block/suite/code-block is a group of statements that are part of another statement.*
- *Blocks are represented through indentation.*
- *A variable in Python is defined only when some value is assigned to it.*
- *Python supports dynamic typing i.e., a variable can hold values of different types at different times.*
- *The input( ) is used to obtain input from user ; it always returns a string type of value.*
- *Output is generated through print( ) (by calling print function) statement.*

Solved Problems

1. *What is the difference between a keyword and an identifier ?*

Solution. **Keyword** is a special word that has a special meaning and purpose. Keywords are reserved and are a few. *For example,* if, elif, else etc. are keywords.

Identifier is a user-defined name given to a part of a program *viz.* variable, object, function etc. Identifiers are not reserved. These are defined by the user but they can have letters, digits and a symbol underscore. They must begin with either a letter or underscore. For instance, *_chk, chess, trial* etc. are identifiers in Python.

2. *What are literals in Python ? How many types of literals are allowed in Python ?*

Solution. Literals mean constants *i.e.,* the data items that never change their value during a program run. Python allows *five* types of literals :

(*i*) String literals
(*ii*) Numeric literals
(*iii*) Boolean literals
(*iv*) Special Literal *None*
(*v*) Literal Collections like tuples, lists etc.

3. *How many ways are there in Python to represent an integer literal ?*

Solution. Python allows *three* types of integer literals :

(*a*) Decimal (base 10) integer literals
(*b*) Octal (base 8) integer literals
(*c*) Hexadecimal (base 16) integer literals

(*a*) **Decimal Integer Literals.** An integer literal consisting of a sequence of digits is taken to be decimal integer literal unless it begins with 0 (digit zero).

For instance, 1234, 41, +97, −17 are decimal integer literals.

(*b*) **Octal Integer Literals.** A sequence of digits starting with 0o (digit zero followed by letter o) is taken to be an octal integer. For instance, decimal integer 8 will be written as 0o10 as octal integer. ($8_{10} = 10_8$) and decimal integer 12 will be written as 0o14 as octal integer ($12_{10} = 14_8$).

(*c*) **Hexadecimal Integer Literals.** A sequence of digits preceded by *0x* or *0X* is taken to be an hexadecimal integer.

For instance, decimal 12 will be written as 0XC as hexadecimal integer.

Thus number 12 will be written either as 12 (as decimal), 0o14 (as octal) and 0XC (as hexadecimal).

4. *What will be the sizes of following constants :*

```
'\a',  "\a",  "Reema\'s",  '\"',  "it's",  "XY \
                                            YZ" ,  """XY⏎
                                                     YZ""" ?
```

Solution.

`'\a'` Size is 1 as there is 1 character (escape sequence) and it is a string literal enclosed in single quotes.

`"\a"` Size is 1 as there is 1 character enclosed in double quotes.

`"Reema\'s"` Size is 7 because it is a string having 7 characters enclosed in double quotes. (\' is escape sequence for apostrophe and is considered a single character.)

`'\"'` Size is 1. It is a character constant and is containing just one character \".

`"it's"` Size is 4. Python allows single quote without escape sequence in a double quoted string and a double quote without escape sequence in a single quoted string *e.g.*, 'no."1tag'.

```
"xy\
yz"
```
Size is 4. It is a multi-line string create with \ in the basic string.

```
"""xy↵
  yz"""
```
Size is 5. Triple quoted multi-line string, EOL (↵) character is also counted in size.

5. *How many types of strings are supported in Python ?*

Solution. Python allows *two* string types :

(*i*) *Single-line Strings* — Strings that are terminated in single line

(*ii*) *Multi-line Strings* — Strings storing multiple lines of text.

6. *How can you create multi-line strings in Python ?*

Solution. Multi-line strings can be created in *two* ways :

(*a*) By adding a backslash at the end of normal single-quote or double-quote strings *e.g.*,

```
Text = "Welcome \
       To\
       Python"
```

(*b*) By typing the text in triple quotation marks. (No backslash needed at the end of line) *e.g.*,

```
Str1 = """Welcome
To
Python
"""
```

7. *What factors guide the choice of identifiers in programs ?*

Solution.

(*i*) An identifier must start with a letter or underscore followed by any number of digits and/or letters.

(*ii*) No reserved word or standard identifier should be used.

(*iii*) No special character (other than underscore) should be included in the identifier.

8. *Write the following real constants in exponent form : 17.251, 151.02, 0.00031, 0.452.*

Solution.

(*i*) $17.251 = 0.17251 \times 10^2$ = 0.17251E02

(*ii*) $151.02 = 0.15102 \times 10^3$ = 0.15102E03

(*iii*) $0.00031 = 0.31 \times 10^{-3}$ = 0.31E–3

(*iv*) $0.452 = 0.0452 \times 10^1$ = 0.0452E01

9. *What is None literal in Python ?*

Solution. Python has one special literal called **None**.

The **None** literal is used to indicate something that has not yet been created in simple words, or absence of value. It is also used to indicate the end of lists in Python.

10. *Identify the types of following literals ?*

23.789	23789	True	'True'	"True"
False	'"False"'	0XFACE	0o213	0o789

Solution.

23.789	Floating point
23789	integer
True	Boolean
'True'	String
"True"	String
False	Boolean
'"False"'	String
0XFACE	Integer (Hexadecimal)
0o213	Integer(Octal)
0o789	Invalid token (beginning with 0 means it is octal number but digits 8 and 9 are invalid digits in octal numbers)
None	None

11. *What is the difference between an expression and a statement in Python ?*

Solution.

Expression	Statement
Legal combination of symbols	Programming instruction as per Python syntax
Represents something	Does something
Python evaluates it	Python executes it
End result is a value	Need not result in a value
Example :	*Examples :*
`2.3`	`print ("Hello")`
`(3 + 5) / 4`	`if a > 0 :`

12. *Which of the following are syntactically correct strings? State reasons.*

(*a*) `"This course is great!"`

(*b*) `'She shouted "Hello!" very loudly.'`

(*c*) `"Goodbye'`

(*d*) `'This course is great!'`

(*e*) `"Hello`

(*f*) `"I liked the movie 'Bruce Almighty' very much."`

Solution. Strings (*a*), (*b*), (*d*) and (*f*) are syntactically correct. (Strings (*b*) and (*f*) are also valid as single quote strings can use double–quotes inside them and vice versa.)

String (*c*) ("Goodbye') is incorrect because opening and closing quotes don't match.

String (*e*) ("Hello) is invalid because it has no closing quotes.

13. *What is the error in following Python program with one statement ?*

```
print ("My name is", name)
```

Suggest a solution.

Solution. The above statement is trying to print the value of an undefined variable name. The solution to above problem is to define the variable name before using it, *i.e.*,

```
name = 'Tanya'
print ("My name is", name)
```

14. *The following code is not giving desired output. We want to input value as 20 and obtain output as 40. Could you pinpoint the problem ?*

```
Number = input( "Enter Number" )
DoubleTheNumber = Number * 2
Print (DoubleTheNumber)
```

Solution. The problem is that input( ) returns value as a string, so the input value 20 is returned as string '20' and not as integer 20. So the output is not 40.

Also **Print( )** is not legal function of Python ; it should be **print( )**.

15. *What would be the correction for problem of previous question ?*

Solution. By using int( ) with input( ), we can convert the string into integer value, *i.e.*, as :

```
Number = int(input ("Enter Number" ) )
DoubleTheNumber = Number * 2
print (DoubleTheNumber)
```

Now the program will print desired output as 40 if 20 is typed as input.

16. *Why is following code giving errors ?*

```
name = "Rehman"
print ("Greetings !!!")
    print ("Hello", name)
    print ("How do you do ?")
```

Solution. The problem with above code is inconsistent indentation. In Python, we cannot indent a statement unless it is inside a suite and we can indent only as much is required.

Thus, corrected code will be :

```
name = "Rehman"
print ("Greetings !!!")
print ("Hello", name)
print ("How do you do ?")
```

17. *Write a program to obtain temperature in Celsius and convert it into Fahrenheit using formula*

$$°C \times 9/5 + 32 = °F$$

Solution.

```
# Celsius to Fahrenheit
Cels = float(input( "Enter temp in Celsius :") )
print ("Temperature in Celsius is :", Cels)
Fahr = Cels * 9/5 + 32
print ("Temperature in Fahrenheit is :", Fahr)
```

18. *What will be the output produced by following code ?*

```
value = 'Simar'
age = 17
print (name, ", you are ", 17, "now but", end = '')
print ("you will be ", age + 1, "next year")
```

Solution.

```
Simar, you are 17 now but you will be 18 next year.
```

19. *What will be the output of following code ?*

```
x, y = 2, 6
x, y = y, x + 2
print (x, y)
```

Solution.

```
6 4
```

Explanation. First line of code assigns values 2 and 6 to variables x and y respectively. Next line (second line) of code first evaluates right hand side *i.e.*, y, $x+2$ which is 6, 2 +2 *i.e.*, 6, 4 ; and then assigns this to x, y *i.e.*, x, $y = 6, 4$; so x gets 6 and y gets 4. Third line prints x and y so the output is 6 4.

20. *Predict the output of following :*

```
x, y = 7, 2
x, y, x = x + 1, y + 3, x + 10
print (x, y)
```

Solution.

```
17 5
```

GLOSSARY

Constant	A data item that never changes its value during a program run.
Identifier	Name given by user for a part of the program.
Keyword	Reserved word having special meaning and purpose.
Lexical Unit	Other name of token.
Literal	Constant.
Token	The smallest individual unit in a program.
String literal	Sequence of characters enclosed in any type of quotes.
Variable	Named stored location whose value can be manipulated during program run.
Expression	Legal combination of symbols that represents a value.

Assignments

Type A : Short Answer Questions/Conceptual Questions

1. What are tokens in Python ? How many types of tokens are allowed in Python ? Examplify your answer.
2. How are keywords different from identifiers ?
3. What are literals in Python ? How many types of literals are allowed in Python ?
4. Can nongraphic characters be used in Python ? How ? Give examples to support your answer.
5. How are floating constants represented in Python ? Give examples to support your answer.
6. How are string-literals represented and implemented in Python ?
7. Which of these is not a legal numeric type in Python ? (*a*) int (*b*) float (*c*) decimal.
8. Which argument of print( ) would you set for :
   (*i*) changing the default separator (space) ? (*ii*) printing the following line in current line ?
9. What are operators ? What is their function ? Give examples of some unary and binary operators.
10. What is an expression and a statement ?
11. What all components can a Python program contain ?
12. What do you understand by block/code block/suite in Python ?
13. What is the role of indentation in Python ?
14. What are variables ? How are they important for a program ?
15. What do you understand by undefined variable in Python ?
16. What is Dynamic Typing feature of Python ?
17. What would the following code do : $X = Y = 7$?
18. What is the error in following code : $X, Y = 7$?
19. Following variable definition is creating problem $X = 0281$, find reasons.
20. "Comments are useful and easy way to enhance readability and understandability of a program." Elaborate with examples.

Type B : Application Based Questions

1. From the following, find out which assignment statement will produce an error. State reason(s) too.

   (*a*) $x = 55$ (*b*) $y = 037$ (*c*) $z = 0o98$ (*d*) 56thnumber = 3300
   (*e*) length = 450.17 (*f*) !Taylor = 'Instant' (*g*) this variable = 87.E02
   (*h*) float = .17E – 03 (*i*) FLOAT = 0.17E – 03

2. Find out the error(s) in following code fragments :

   (*i*)
   ```
   temperature = 90
   print temprature
   ```

   (*ii*)
   ```
   a = 30
   b = a + b
   print (a And b)
   ```

   (*iii*)
   ```
   a, b, c = 2, 8, 9
   print (a, b, c)
   c, b, a = a, b, c
   print (a ; b ; c)
   ```

   (*iv*)
   ```
   X = 24
   4 = X
   ```

   (*v*)
   ```
   print ("X =" X)
   ```

   (*vi*)
   ```
   else = 21 - 5
   ```

3. What will be the output produced by following code fragment (s) ?

(i)
```
X = 10
X = X + 10
X = X - 5
print (X)
X, Y = X - 2, 22
print (X, Y)
```

(ii)
```
first = 2
second = 3
third = first * second
print (first, second, third)
first = first + second + third
third = second * first
print (first, second, third)
```

(iii)
```
side = int(input('side') )     #side given as 7
area = side * side
print (side, area)
```

4. What is the problem with the following code fragments ?

(i)
```
a = 3
    print (a)
b = 4
    print (b)
s = a + b
    print (s)
```

(ii)
```
name = "Prejith"
age = 26
print ("Your name & age are ", name + age)
```

(iii)
```
a = 3
s = a + 10
a = "New"
q = a / 10
```

5. Predict the output :

```
x = 40
y = x + 1
x = 20, y + x
print (x, y)
```

6. Predict the output

```
x, y = 20, 60
y, x , y = x, y - 10, x + 10
print (x, y)
```

7. Predict the output

(a)
```
a, b = 12, 13
c, b = a*2, a/2
print (a, b, c)
```

(b)
```
a, b = 12, 13
print (print(a + b))
```

8. Predict the output

```
a, b, c = 10, 20, 30
p, q, r = c - 5, a + 3, b - 4
print ('a, b , c :', a, b, c, end = '')
print ('p, q, r :', p, q, r)
```

9. Find the errors in following code fragment

(a)
```
y = x + 5
print (x, Y)
```

(b)
```
print (x = y = 5)
```

(c)
```
a = input("value")
b = a/2
print (a, b)
```

10. Find the errors in following code fragment : (The input entered is XI)

```
c = int(input( "Enter your class") )
print ("Your class is", c)
```

11. Consider the following code :

```
name = input("What is your name?")
print ('Hi', name, ',')
print ("How are you doing?")
```

was intended to print output as

```
Hi <name>, How are you doing ?
```

But it is printing the output as :

```
Hi <name>,
How are you doing?
```

What could be the problem ? Can you suggest the solution for the same ?

12. Find the errors in following code fragment :

```
c = input( "Enter your class" )
print ("Last year you were in class") c – 1
```

13. What will be returned by Python as result of following statements?

(a) `>>> type(0)` (b) `>>> type(int(0))` (c) `>>>.type(int('0'))`

(d) `>>> type('0')` (e) `>>> type(1.0)` (f) `>>> type(int(1.0))`

(g) `>>>type(float(0))` (h) `>>> type(float(1.0))` (i) `>>> type( 3/2)`

Match your result after executing above statements.

14. What will be the output produced by following code ?

(a) `>>> str(print())+"One"`

(b) `>>> str(print("hello"))+"One"`

Match your result after executing above statements. Can you explain why this result came?

15. What will be the output produced by following code ?

(a) `>>> print(print("Hola"))`

(b) `>>> print(print("Hola", end = " "))`

Match your result after executing above statements. Can you explain why this result came?

16. Carefully look at the following code and its execution on Python shell. Why is the last assignment giving error ?

```
>>> a = 0o12
>>> print(a)
10
>>> b = 0o13
>>> c = 0o78
    File "<python-input-41-27fbe2fd265f>", line 1
    c = 0o78
        ^
SyntaxError : invalid syntax
```

17. Predict the output

```
a, b, c = 2, 3, 4
a, b, c = a*a, a*b, a*c
print(a, b, c)
```

18. The id( ) can be used to get the memory address of a variable. Consider the following code and tell if the id( ) functions will return the same value or not (as the value to be printed via print( )) ? Why ?
[There are four print( ) function statements that are printing id of variable *num* below)

```
num = 13
print( id(num) )
num = num + 3
print( id(num) )
num = num - 3
print( id(num) )
num = "Hello"
print( id(num) )
```

19. Consider below given two sets of codes, which are nearly identical, along with their execution in Python shell. Notice that first code-fragment after taking input gives error, while second code-fragment does not produce error. Can you tell why ?

(a)

```
>>> print(num = float(input("value1:")) )
value1:67          <-- Input taken as per execution of input()
Traceback (most recent call last):
    File "<python-input-56-78b83d911bc6>", line 1, in <module>
        print(num = float(input("value1:")) )
TypeError: 'num' is an invalid keyword argument for this function
```

(b)

```
>>> print(float(input("value1:")) )
value1:67
67.0          <-- Code successfully executed. No error reported.
```

20. Predict the output of the following code :

```
days = int(input("Input days: ")) * 3600 * 24
hours = int(input("Input hours: ")) * 3600
minutes = int(input("Input minutes: ")) * 60
seconds = int(input("Input seconds: "))
time = days + hours + minutes + seconds
print("The amounts of seconds", time)
```

If the input given is in this order : 1, 2, 3, 4

Type C : Programming Practice/Knowledge based Questions

1. Write a program that displays a joke. But display the punchline only when the user presses enter key. (**Hint.** You may use input( ))
2. Write a program to read today's date (only del part) from user. Then display how many days are left in the current month.
3. Write a program that generates the following output :

```
5
10
9
```

Assign value 5 to a variable using assignment operator (=) Multiply it with 2 to generate 10 and subtract 1 to generate 9.

4. Modify above program so as to print output as 5@10@9.
5. Write the program with maximum three lines of code and that assigns first 5 multiples of a number to 5 variables and then print them.
6. Write a Python program that accepts radius of a circle and prints its area.
7. Write Python program that accepts marks in 5 subjects and outputs average marks.
8. Write a short program that asks for your height in centimetres and then converts your height to feet and inches. (1 foot = 12 inches, 1 inch = 2.54 cm).
9. Write a program to read a number n and print n^2, n^3 and n^4.
10. Write a program to find area of a triangle.
11. Write a program to compute simple interest and compound interest.
12. Write a program to input a number and print its first five multiples.
13. Write a program to read details like *name, class, age* of a student and then print the details firstly in same line and then in separate lines.

    Make sure to have two blank lines in these two different types of prints.
14. Write a program to input a single digit(n) and print a 3 digit number created as <n(n+1)(n+2)> e.g., if you input 7, then it should print 789. Assume that the input digit is in range 1-7.
15. Write a program to read three numbers in three variables and swap first two variables with the sums of first and second, second and third numbers respectively.

3 Data Handling

In This Chapter

3.1 INTRODUCTION

In any language, there are some fundamentals you need to know before you can write even the most elementary programs. This chapter introduces some such fundamentals : *data types, variables, operators* and *expressions* in Python.

Python provides a predefined set of data types for handling the data it uses. Data can be stored in any of these data types. This chapter is going to discuss various types of data that you can store in Python. Of course, a program also needs a means to identify stored data.

So, this chapter shall also talk about mutable and immutable variables in Python.

IMPORTANT : In this chapter and onwards, we have shown screenshots of Python shell used in Spyder IDE (IPython shell) which uses In[ ] : prompt to take commands. But in text section, we have given commands with >>> prompt, which is a standard way of telling that this command is given on a Python shell prompt.

All Python shells execute the code in the same way, though IPython shell is an enhanced shell, used in many universities (*e.g.*, Toronto University). You are free to use any Python IDE/shell of your choice, but we have installed Python using **Anaconda distribution** which installs maximum libraries (such as NumPy, SciPy, Panda and many more) along with Python. As **Spyder IDE** and **IPython shell** is available as part of *Anaconda Python* distribution and is very easy to use, we have given screenshots of *Spyder IDE* and *IPython shell* in this book at times.

3.2 DATA TYPES

Data can be of many types *e.g., character, integer, real, string* etc. Anything enclosed in quotes represents string data in Python. Numbers without fractions represent integer data. Numbers with fractions represent real data and *True* and *False* represent Boolean data. Since the data to be dealt with are of many types, a programming language must provide ways and facilities to handle all types of data.

Built-in Core Data Types
- Numbers (*int, float, complex*)
- String
- List
- Tuple
- Dictionary

Before you learn how you can process different types of data in Python, let us discuss various data-types supported in Python. In this discussion of data types, you'll be able to know Python's capabilities to handle a specific type of data, such as the memory space it allocates to hold a certain type of data and the range of values supported for a data type etc.

Python offers following built-in core data types : (*i*) *Numbers* (*ii*) *String* (*iii*) *List* (*iv*) *Tuple* (*v*) *Dictionary*.

3.2.1 Numbers

As it is clear by the name the Number data types are used to store numeric values in Python. The Numbers in Python have following core data types :

(*i*) Integers
- Integers (signed)
- Booleans

(*ii*) Floating-Point Numbers

(*iii*) Complex Numbers

3.2.1A Integers

Integers are whole numbers such as 5, 39, 1917, 0 etc. They have no fractional parts. Integers are represented in Python by numeric values with no decimal point. Integers can be positive or negative, *e.g.*, + 12, – 15, 3000 (missing + or – symbol means it is positive number).

Types of Integers in Python
- Integers (signed)
- Booleans

There are *two* types of integers in Python :

(*i*) **Integers (signed).** It is the normal integer[1] representation of whole numbers. Integers in *Python 3.x* can be of any length, it is only limited by the memory available. Unlike other languages, *Python 3.x* provides single data type (*int*) to store any integer, whether *big* or *small*. It is signed representation, *i.e.*, the integers can be positive as well as negative.

(*ii*) **Booleans.** These represent the truth values *False* and *True*. The Boolean type is a subtype of plain integers, and Boolean values *False* and *True* behave like the values 0 and 1, respectively. To get the Boolean equivalent of 0 or 1, you can type *bool(0)* or *bool (1)*, Python will return *False* or *True* respectively.

[1] In some cases, the exception *OverflowError* is raised instead if the given number cannot be represented through available number of bytes.

See figure below (left side).

```
In [5]: bool(0)
Out[5]: False

In [6]: bool(1)
Out[6]: True
```

```
In [9]: str(False)
Out[9]: 'False'

In [10]: str(True)
Out[10]: 'True'
```

NOTE

The str( ) function converts a value to string.

However, when you convert Boolean values *False* and *True* to a string, the strings **'False'** or **'True'** are returned, respectively. The str( ) function converts a value to string. See figure above (right side)

NOTE

The two objects representing the values *False* and *True* (not false or true) are the only Boolean objects in Python.

3.2.1B Flooting Point Numbers

A number having fractional part is a floating-point number. *For example*, 3.14159 is a floating-point number. The decimal point signals that it is a floating-point number, not an integer. The number 12 is an integer, but 12.0 is a floating-point number.

Recall (from Literals/Values' discussion in chapter 2) that fractional numbers can be written in *two* forms :

(*i*) **Fractional Form** (*Normal Decimal Notation*) *e.g.*, 3500.75, 0.00005, 147.9101 etc.

(*ii*) **Exponent Notation** *e.g.*, 3.50075E03, 0.5E-04, 1.479101E02 etc.

Floating point variables represent real numbers, which are used for measurable quantities like distance, area, temperature etc. and typically have a fractional part.

Floating-point numbers have *two* **advantages** over integers :

- They can represent values between the integers.
- They can represent a much greater range of values.

But floating-point numbers suffer from one disadvantage also :

- Floating-point operations are usually slower than integer operations.

In Python, floating point numbers represent machine-level **double precision floating point numbers**[2] **(15 digit precision)**. The range of these numbers is limited by underlying machine architecture subject to available (virtual) memory.

NOTE

In Python, the Floating point numbers have precision of 15 digits (double-precision).

Check Point 3.1

1. What are the built-in core data types of Python ?
2. What do you mean by Numeric types ? How many numeric data types does Python provide ?
3. What will be the data types of following two variables ?

   A = 2147483647

   B = A + 1

   (Hint. Carefully look the values they are storing. You can refer to range of Python number table.)
4. What are Boolean numbers ? Why are they considered as a type of integers in Python ?

2. As per Python documentation, "Python does not support single-precision floating point numbers ; the savings in processor and memory usage that are usually the reason for using these is dwarfed by the overhead of using objects in Python, so there is no reason to complicate the language with two kinds of floating point numbers".

3.2.1C Complex Numbers

Python is a versatile language that offers you a numeric type to represent *Complex Numbers* also. Complex Numbers ? Hey, don't you know about *Complex numbers* ? Uhh, I see. You are going to study about *Complex numbers* in class XI Mathematics book. Well, if you don't know anything about complex numbers, then for you to get started, I am giving below brief introduction of *Complex numbers* and then we shall talk about *Python's representation of Complex numbers.*

Mathematically, a complex number is a number of the form $A+Bi$ where i is the imaginary number, *equal to the square root of* -1 *i.e.*, $\sqrt{-1}$.

A complex number is made up of both **real and imaginary components.** In complex number $A+Bi$, A and B are real numbers and i is imaginary. If we have a complex number z, where $z=a+bi$ then a would be the *real component* and b would represent the *imaginary component* of z, *e.g.*, real component of $z=4+3i$ is 4 and the imaginary component would be 3.

NOTE

A complex number is in the form $A+Bi$ where i is the imaginary number, equal to the square root of -1 *i.e.*, $\sqrt{-1}$, that is $i^2=-1$.

Complex Numbers in Python

Python represents complex numbers in the form $A+Bj$. That is, to represent imaginary number, Python uses j (or J) in place of traditional i. So in Python $j=\sqrt{-1}$. Consider the following examples where a and b are storing two complex numbers in Python :

```
a = 0 + 3.1j
b = 1.5 + 2j
```

The above complex number a has real component as 0 and imaginary component as 3.1 ; in complex number b, the real part is 1.5 and imaginary part is 2. When you display complex numbers, Python displays complex numbers in parentheses when they have a nonzero real part as shown in following examples.

NOTE

Python represents complex numbers as a pair of floating point numbers.

```
>>> c = 0 + 4.5j
>>> d = 1.1 + 3.4j
>>> c
4.5j
>>> d
(1.1+3.4j)
>>> print (c)
4.5j
>>> print (d)
(1.1+3.4j)
```

See, a complex number with non-zero real part is displayed with parentheses around it.

But no parentheses around complex number with real part as zero(0).

NOTE

Complex numbers are quite commonly used in *Electrical Engineering*. In the field of electricity, however, because the symbol i is used to represent current, they use the symbol j for the square root of -1. Python adheres to this convention.

Unlike Python's other numeric types, complex numbers are a composite quantity made of two parts : *the real part* and the *imaginary part*, both of which are represented internally as ***float*** values (floating point numbers).

You can retrieve the two components using attribute references. For a complex number z :

- z.real gives the *real part*.
- z.imag gives the *imaginary part* as a float, not as a complex value.

For example,

```
>>> z = (1 + 2.56j) + (-4 -3.56j)
>>> a
(-3 -1j)
>>> z.real        <-- It will display real part of complex number z
-3.0
>>> z.imag        <-- It will display imaginary part of complex number z
-1.0
```

TIP

The real and imaginary parts of a complex number z can be retrieved through the read-only attributes *z.real* and *z.imag*.

The range of numbers represented through Python's numeric data types is given below.

Table 3.1 *The Range of Python Numbers*

Data type	Range
Integers	an unlimited range, subject to available (virtual) memory only
Booleans	*two* values True (1), False (0)
Floating point numbers	an unlimited range, subject to available (virtual) memory on underlying machine architecture.
Complex numbers	Same as floating point numbers because the real and imaginary parts are represented as **floats**.

Check Point

3.2

1. What are floating point numbers ? When are they preferred over integers ?
2. What are complex numbers ? How would Python represent a complex number with real part as 3 and imaginary part as – 2.5 ?
3. What will be the output of following code ?

```
p = 3j
q = p + (1 + 1.5j)
print (p)
print (q)
```

4. What will be the output of following code ?

```
r = 2.5 + 3.9j
print (r.real)
print (r.imag)
```

5. Why does Python uses symbol *j* to represent imaginary part of a complex number instead of the conventional *i* ?

Hint. Refer note above.

3.2.2 Strings

You already know about strings (as data) in Python. In this section, we shall be talking about Python's data type string. A string data type lets you hold string data, *i.e.*, any number of valid characters into a set of quotation marks.

In Python 3.x, each character stored in a string[3] is a Unicode character. Or in other words, all strings in Python 3.x are sequences of *pure Unicode characters*. *Unicode* is a system designed to represent every character from every language. A string can hold any type of known characters *i.e.*, *letters*, *numbers*, and *special characters*, of any known scripted language.

NOTE

All Python (3.x) strings store Unicode characters.

Following are all legal strings in Python :

"abcd" , "1234", '$%^&', '????', "ŠÆËá" , "??????", '????', "????", '???', "??"

3. Python has no separate character datatype, which most other programming languages have – that can hold a single character. In Python, a character is a string type only, with single character.

String as a Sequence of Characters

A Python string is a sequence of characters and each character can be individually accessed using its index. Let us understand this.

Let us first study the internal structure or composition of Python strings as it will form the basis of all the learning of various string manipulation concepts. Strings in Python are stored as individual characters in contiguous location, with two-way index for each location.

The individual elements of a string are the characters contained in it (stored in contiguous memory locations) and as mentioned the characters of a string are given two-way index for each location.

Let us understand this with the help of an illustration as given in Fig. 3.1.

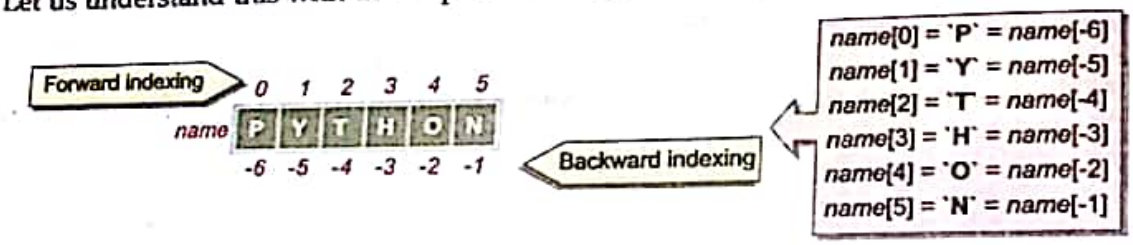


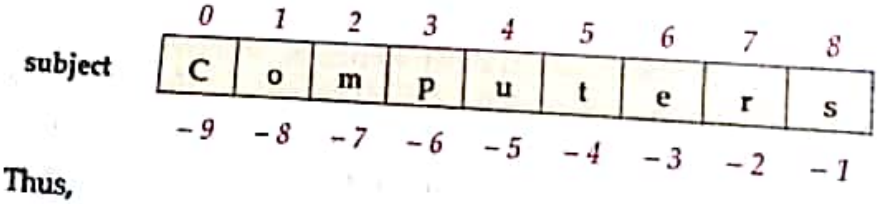
Figure 3.1 Structure of a Python String.

From Fig. 3.1 you can infer that :

- Strings in Python are stored by storing each character separately in contiguous locations.
- The characters of the strings are given two-way indices :
  - 0, 1, 2, ... in the *forward direction* and
  - – 1, – 2, – 3, ... in the *backward direction.*

Thus, you can access any character as *<stringname>[<index]* e.g., to access the first character of string *name* shown in Fig. 3.1, you'll write name[0], because the index of first character is 0. You may also write name [–6] for the above example *i.e.*, when string name is storing "PYTHON".

Let us consider another string, say subject = 'Computers'. It will be stored as :



Thus,

```
subject[0] = 'C'      subject[2] = 'm'      subject [6]= 'e'
subject [-1] = 's'    subject[-7]= 'm'      subject[-9] = 'C'
```

Since length of string variable can be determined using function len(<string>), we can say that :

- first character of the string is *at index 0* or *–length*
- second character of the string is *at index 1* or *–(length-1)*

  :

- second last character of the string is *at index (length -2)* or *-2*
- last character of the string is *at index (length -1)* or *-1*

In a string, say *name*, of length *ln*, the valid indices are 0, 1, 2, ... *ln*-1. That means, if you try to give something like :

```
>>> name[ln]
```

Python will return an error like :

```
Traceback (most recent call last):
  File "<pyshell#40>", line 1, in <module>
    name[ln]
IndexError: string index out of range
```

NOTE

The index (also called subscript sometimes) is the numbered position of a letter in the string. In Python, indices begin 0 onwards in the forward direction and –1, –2, ... in the backward direction.

The reason is obvious that in string there is no index equal to the length of the string, thus accessing an element like this causes an error.

Also, another thing that you must know is that you cannot change the individual letters of a string in place by assignment because **strings are immutable and hence item assignment is not supported**, *i.e.*,

```
name = 'hello'
name[0] = 'p'
```

individual letter assignment not allowed in Python

will cause an error like :

```
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    name[0] = 'p'
TypeError: 'str' object does not support item assignment
```

However, you can assign to a string another string or an expression that returns a string using assignment, *e.g.*, following statement is valid :

```
name = 'hello'
name = "new"
```

Strings can be assigned expressions that give strings.

NOTE

Valid string indices are *0, 1, 2* ... upto *length-1* in forward direction and *–1, –2, –3...* *– length* in backward direction.

3.2.3 Lists and Tuples

The lists and tuples are Python's compound data types. We have taken them together in one section because they are basically the same types with one difference. Lists can be changed / modified (*i.e.*, mutable) but tuples cannot be changed or modified (*i.e.*, immutable). Let us talk about these two Python types one by one.

3.2.3A Lists

A List in Python represents a list of comma-separated values of any datatype between square brackets *e.g.*, following are some lists :

```
[1, 2, 3, 4, 5]
['a', 'e', 'i', 'o', 'u']
['Neha', 102, 79.5]
```

Like any other value, you can assign a list to a variable *e.g.*,

```
>>> a = [1, 2, 3, 4, 5]        # Statement 1
>>> a
[1, 2, 3, 4, 5]
>>> print (a)
[1, 2, 3, 4, 5]
```

To change first value in a list namely *a* (given above), you may write

```
>>> a [0] = 10                  # change 1st item – consider statement 1 above
>>> a
[10, 2, 3, 4, 5]
```

To change 3rd item, you may write

```
>>> a[2] = 30                   # change 3rd item
>>> a
[10, 2, 30, 4, 5]
```

You guessed it right ; the values internally are numbered from 0 (zero) onwards *i.e.*, first item of the list is internally numbered as 0, second item of the list as 1, 3rd item as 2 and so on.

We are not going further in list discussion here. Lists shall be discussed in details in a later chapter.

3.2.3B Tuples

You can think of Tuples (pronounced as tu-pp-le, rhyming with couple) as those lists which cannot be changed *i.e.*, are not modifiable. Tuples are represented as list of comma-separated values of any date type within parentheses, *e.g.*, following are some tuples :

```
p = (1, 2, 3, 4, 5)
q = (2, 4, 6, 8)
r = ('a', 'e', 'i', 'o', 'u')
h = (7, 8, 9, 'A', 'B', 'C')
```

Tuples shall be discussed in details in a later chapter.

NOTE

Values of type list are *mutable i.e.*, changeable – one can change / add / delete a list's elements. But the values of type **tuple** are *immutable i.e.*, non-changable ; one cannot make changes to a tuple.

2.4 Dictionary

Dictionary data type is another feature in Python's hat. The *dictionary* is an unordered set of comma-separated key : value pairs, within { }, with the requirement that within a dictionary, no two keys can be the same (*i.e.*, there are unique keys within a dictionary). For instance, following are some dictionaries :

```
{'a' : 1 , 'e' : 2, 'i' : 3, 'o' : 4, 'u' : 5}
```

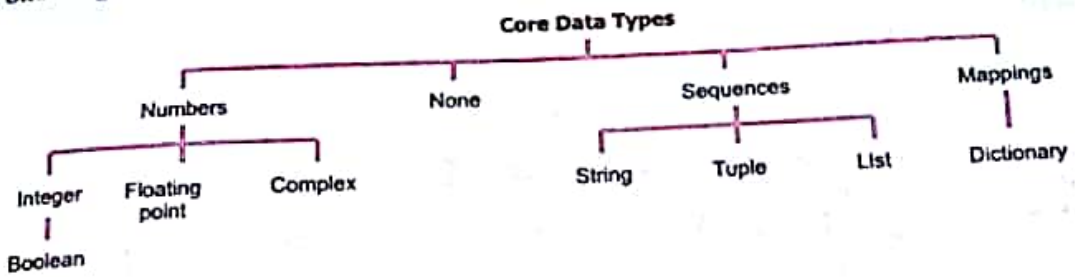
```
>>> vowels = {'a' : 1, 'e' : 2, 'i' : 3, 'o' : 4, 'u' : 5}
>>> vowels['a']
1
>>> vowels['u']
5
```

Here 'a', 'e', 'i', 'o' and 'u' are the keys of dictionary vowels; 1, 2, 3, 4, 5 are values for these keys respectively.

*Specifying **key** inside [ ] after dictionary name gives the corresponding **value** from the **key : value** pair inside dictionary.*

Dictionaries shall be covered in details in a later chapter.

Following figure summarizes the core data types of Python.



3.3 MUTABLE AND IMMUTABLE TYPES

The Python data objects can be broadly categorized into *two – mutable* and *immutable* types, in simple words changeable or modifiable and non-modifiable types.

1. Immutable types

The immutable types are those that can never change their value in place. In Python, the following types are immutable : *integers, floating point numbers, Booleans, strings, tuples.*

Let us understand the concept of immutable types. In order to understand this, consider the code below :

Immutable Types
- integers
- floating point numbers
- booleans
- strings
- tuples

Sample code 3.1

```
P = 5
q = P
r = 5
                :          # will give 5, 5, 5
P = 10
r = 7
q = r
```

After reading the above code, you can say that values of integer variables *p, q, r* could be changed effortlessly. Since *p, q, r* are integer types, you may think that integer types can change values.

But hold : It is not the case. Let's see how.

You already know that in Python, variable-names are just the references to value-objects *i.e.*, data values. The variable-names do not store values themselves *i.e.*, they are not storage containers. Recall section 2.5.1 where we briefly talked about it.

Now consider the **Sample code 3.1** given above. Internally, how Python processes these assignments is explained in Fig. 3.2. Carefully go through figure 3.2 on the next page and then read the following lines.

So although it appears that the value of variable *p / q / r* is changing ; values are *not changing "in place"* the fact is that the variable-names are instead made to refer to new immutable integer object. (Changing **in place** means modifying the same value in same memory location).

- Initially these three statements are executed :

```
p = 5
q = p
r = 5
```

All variables having same value reference the same value object i.e., p, q, r will all reference same integer objects.

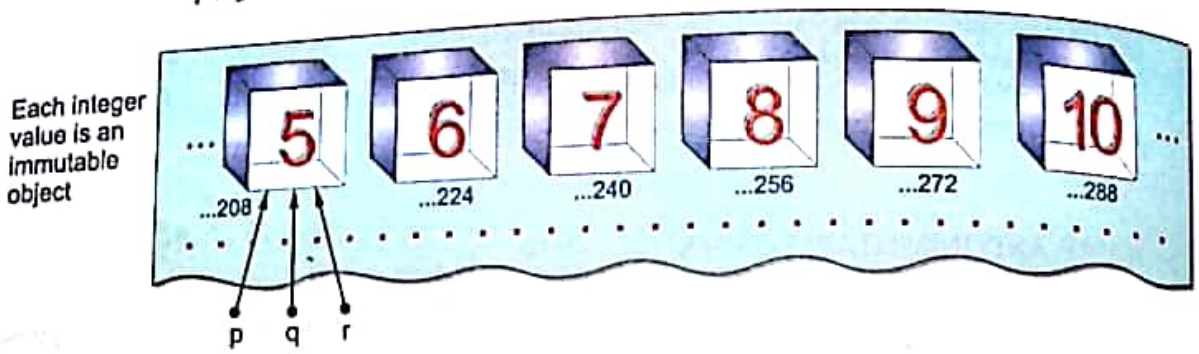


Figure 3.2

You can check/confirm it yourself using id( ). The id( ) returns the memory address to which a variable is referencing.

```
In [40]: p = 5

In [41]: q = p

In [42]: r = 5

In [43]: id(5)
Out[43]: 1457662208

In [44]: id(p)
Out[44]: 1457662208

In [45]: id(q)
Out[45]: 1457662208

In [46]: id(r)
Out[46]: 1457662208
```

Notice the id( ) is returning same memory address for *value 5*, *p*, *q*, *r* - which means all these are referencing the same object.

NOTE

Please note that both In [ ] : and >>> are valid prompts of different Python shells. Where In [ ] : is a prompt of *IPython shell* used with Spyder IDE, >>> is a prompt of Python shell used by IDLE IDE, PyCcharmIDE and others. You will find In [ ] : are in our captured screenshots but in text component we have used >>> , a standard way of depicting any Python prompt.

Please note, memory addresses depend on your operating system and will vary in different sessions.

- When the next set of statements execute, *i.e.*,

```
p = 10
r = 7
q = r
```

then these variable names are made to point to different integer objects. That is, now their memory addresses that they reference will change. The original memory address of p that was having value 5 will be the same with the same value *i.e.*, 5 but p will no longer reference it. Same is for other variables.

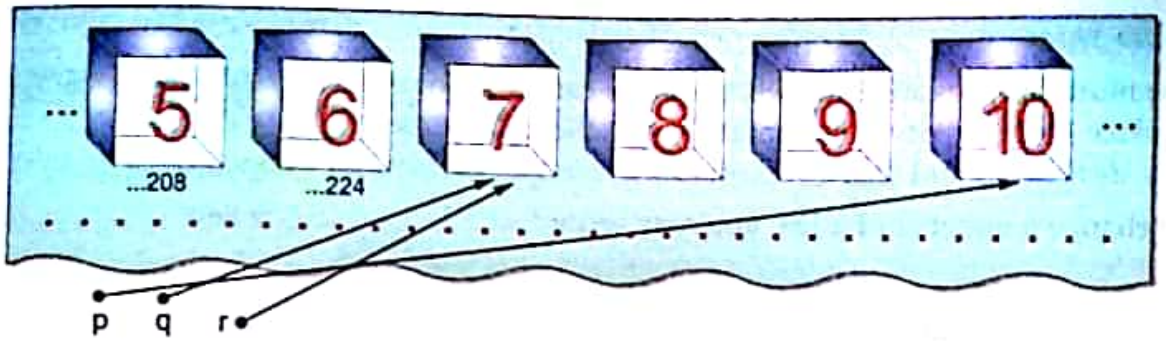


Figure 3.3

Let us check their ids

```
In [47]: p=10

In [48]: r=7

In [49]: q=r

In [50]: id(10)
Out[50]: 1457662288

In [51]: id(p)
Out[51]: 1457662288

In [52]: id(7)
Out[52]: 1457662240

In [53]: id(q)
Out[53]: 1457662240

In [54]: id(r)
Out[54]: 1457662240

In [55]: id(5)
Out[55]: 1457662208
```

Notice, this time with change in value, the reference memory address of variables *p* , *q* and *r* have changed.

The value 5 is at the same address

- Now if you assign 5 to any other variable. Let us see what happens.

```
In [56]: t = 5

In [57]: id(t)
Out[57]: 1457662208
```

Now variable *t* has reference memory address same as initial reference memory address of variable *p* when it has value 5. Compare listings given above

Thus, it is clear that **variable names are stored as references to a value-object. Each time you change the value, the variable's reference memory address changes.**

Variables (of certain types) are NOT LIKE storage containers *i.e.*, with fixed memory address where value changes every time. Hence they are IMMUTABLE.

The objects of following value types are immutable in Python :

- Integer
- Booleans
- tuples
- floating point number
- strings

NOTE

Mutability means that in the same memory address, new value can be stored as and when you want. The types that do not support this property are immutable types.

2. Mutable types

The mutable types are those whose values can be changed in place. Only three types are mutable in Python. These are : *lists, dictionaries* and *sets*.

Mutable Types
- Lists
- Dictionaries
- Sets

To change a member of a list, you may write :

```
Chk = [2, 4, 6]
Chk[1] = 40
```

It will make the list namely Chk as [2, 40, 6].

```
In [60]: Chk = [2, 4, 6 ]

In [61]: id(Chk)
Out[61]: 150195536

In [62]: Chk[1] = 40

In [63]: id(Chk)
Out[63]: 150195536
```

See, even after changing a value in the list Chk, its reference memory address has remained same. That means the change has taken in place - the lists are mutable

NOTE

Mutable objects are : *list, dictionary, set*

Immutable objects: *int, float, complex, string, tuple*

Lists and Dictionaries shall be covered later in this book.

3.3.1 Variable Internals

Python is an object oriented language. Python calls every entity that stores any values or any type of data as an **object**.

An **object** is an entity that has certain properties and that exhibit a certain type of behavior, *e.g.*, integer values are objects – they hold whole numbers only and they have infinite precision (**properties**); they support all arithmetic operations (**behavior**).

So all data or values are referred to as **object** in Python. Similarly, we can say that a variable is also an object that refers to a value.

Python object's Key Attributes
- a type
- a value
- an id

Every Python object has *three* key attributes associated to it :

(*i*) The type of an object.

The type of an object determines the operations that can be performed on the object. Built-in function **type( )** returns the type of an object.

Consider this :

```
>>> a = 4
>>> type (4)
<class 'int'>
>>> type (a)
<class 'int'>
```

Type of integer value 4 is returned int i.e., integer

Type of variable a is also int i.e., integer because a is currently referring to an integer value.

(ii) The value of an object

It is the data-item contained in the object. For a literal, the value is the literal itself and for a variable the value is the data-item it (the variable) is currently referencing. Using **print** statement you can display value of an object. For example,

```
>>> a = 4
>>> print (4)
4
>>> print (a)
4
```

value of integer literal 4 is 4

value of variable a is 4 as it is currently referencing integer value 4.

(iii) The id of an object

The id of an object is generally the memory location of the object. Although **id** is implementation dependent but in most implementations it returns the memory location of the object. Built-in function **id( )** returns the id of an object, *e.g.*,

```
>>> id(4)
30899132
>>> a = 4
>>> id (a)
30899132
```

Object 4 is internally stored at location 30899132

*Variable **a** is current referencing location 30899132 (Notice same as id(4). Recall that variable is not a storage location in Python, rather a label pointing to a value object).*

Check Point 3.3

1. What is String data type in Python ?
2. What are two internal subtypes of String data in Python ?
3. How are str type strings different from Unicode strings ?
4. What are List and Tuple data types of Python ?
5. How is a list type different from tuple data type of Python ?
6. What are Dictionaries in Python ?
7. Identify the types of data from the following set of data

```
'Roshan', u'Roshan', False,
'False', ['R', 'o', 's', 'h', 'a', 'n',],
('R', 'o', 's', 'h', 'a', 'n'),
{'R' : 1, ' o' : 2, 's' : 3, 'h' : 4 'a'
: 5, 'n' : 6}, (2.0-j),
12, 12.0, 0.0, 0, 3j, 6+2.3j,
True, "True"
```

8. What do you understand by mutable and immutable objects ?

The **id( )** of a variable is same as the **id( )** of value it is storing.

Now consider this :

Sample code 3.2

```
>>> id(4)
30899132
>>> a = 4
>>> id(a)
30899132
>>> b = 5
>>> id(5)
30899120
>>> id(b)
30899120
>>> b = b-1
>>> id(b)
30899132
>>>
```

*The id's of **value 4** and **variable a** are the same since the memory-location of **4** is same as the location to which **variable a** is referring to.*

*Variable **b** is currently having value **5**, i.e., referring to integer value 5*

*Variable **b** will now refer to value **4***

*Now notice that the **id** of **variable b** is same as **id** of **integer 4**.*

Thus internal change in value of variable *b* (from 5 to 4) of sample code 3.2 will be represented as shown in Fig. 3.4.

```
a = 4
:
b = 5
:
b = b-1
```

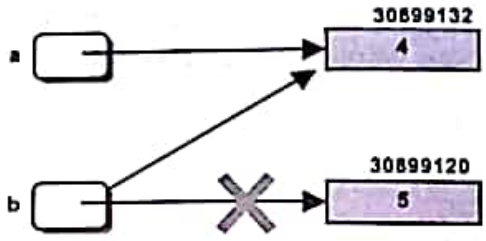


Figure 3.4 Memory representation of sample code 3.2.

Please note that while storing complex numbers, id's are created differently, so a complex literal say 2.4*j* and a complex variable say *x* having value 2.4*j* may have different id's.

DATA TYPES IN PYTHON, MUTABILITY, INTERNALS — Progress In Python 3.1

In the Python Shell IDLE or IPython shell of Spyder IDE, type the statements as instructed.

1. Using value 12, create data-item (that contains 12 in it) of following types. Give at least two examples for each type of data. Check type of each of your examples using type( ) function, e.g., to check the type of 3.0, you can type on Python prompt type(3.0).

(*a*)	Integer	
(*b*)	Floating point number	
(*c*)	Complex number	

Please check the practical component-book – Progress In Computer Science with Python and fill it there in PriP 3.1 under Chapter 3 after practically doing it on the computer.

>>>❖<<<

3.4 OPERATORS

The operations being carried out on data, are represented by operators. The symbols that trigger the operation / action on data, are called operators. The operations(specific tasks) are represented by *Operators* and the objects of the operation(s) are referred to as *Operands*.

Python's rich set of operators comprises of these types of operators : (*i*) Arithmetic operators (*ii*) Relational operators (*iii*) Identity operators (*iv*) Logical operators (*v*) Bitwise operators (*vi*) Membership operators.

Out of these, we shall talk about membership operators later when we talk about strings, lists, tuples and dictionaries. (Chapter 5 onwards)

Let us discuss these operators in detail.

OPERATORS

The symbols[4] that trigger the operation / action on data, are called **Operators**, and the data on which operation is being carried out, *i.e.*, the objects of the operation(s) are referred to as **Operands**.

3.4.1 Arithmetic Operators

To do arithmetic, Python uses arithmetic operators. Python provides operators for basic calculations, as given below :

+	addition	//	floor division
–	subtraction	%	remainder
*	multiplication	**	exponentiation
/	division		

Each of these operators is a binary operator *i.e.*, it requires two values (operands) to calculate a final answer. Apart from these binary operators, Python provides two unary arithmetic operators (that require one operand) also, which are unary +, and unary –.

3.4.1A Unary Operators

Unary +

The operators unary '+' precedes an operand. The operand (the value on which the operator operates) of the unary + operator must have arithmetic type and the result is the value of the argument. *For example,*

if a = 5 then + *a* means 5.
if a = 0 then + *a* means 0.
if a = – 4 then + *a* means – 4.

UNARY OPERATORS

The operators that act on one operand are referred to as *Unary Operators*.

Unary –

The operator unary – precedes an operand. The operand of the unary – operator must have arithmetic type and the result is the negation of its operand's value. *For example,*

if a = 5 then – *a* means – 5.
if a = 0 then – *a* means 0 (there is no quantity known as – 0)
if a = – 4 then – *a* means 4.

This operator reverses the sign of the operand's value.

4. Sometimes these can be words too, *e.g.*, in Python ***is*** and ***is not*** are also operators.

3.4.1B Binary Operators

Operators that act upon two operands are referred to as Binary Operators. The operands of a binary operator are distinguished as the left or right operand. Together, the operator and its operands constitute an expression.

BINARY OPERATORS

Operators that act upon two operands are referred to as *Binary Operators.*

1. Addition operator +

The arithmetic binary operator + adds values of its operands and the result is the sum of the values of its two operands. *For example,*

```
4 + 20                  results in  24
a + 5 (where a = 2)     results in  7
a + b (where a = 4, b = 6)  results in  10
```

For Addition operator + operands may be of *number* types[5].

Python also offers + as a concatenation operator when used with strings, lists and tuples. This functionality for strings will be covered in Chapter 5 – String Manipulation ; for lists, it will be covered in *Chapter 7 – List Manipulation.*

2. Subtraction operator –

The – operator subtracts the second operand from the first. *For example,*

```
14 – 3                  evaluates to 11
a – b (where a = 7, b = 5)  evaluates to 2
x – 3 (where x = –1)    evaluates to –4
```

The operands may be of number types.

3. Multiplication operator *

The * operator multiplies the values of its operands. *For example,*

```
3 * 4                   evaluates to 12
b * 4 (where b = 6)     evaluates to 24
p * 2 (where p = –5)    evaluates to –10
a * c (where a = 3, c = 5)  evaluates to 15
```

The operands may be of *integer* or *floating point number* types.

*Python also offers * as a replication operator when used with strings. This functionality will be covered in Chapter 5 – String Manipulation.*

4. Division operator /

The / operator in *Python 3.x* divides its *first operand* by the *second operand* and always returns the result as a **float** value, *e.g.,*

```
4/2         evaluates to  2.0
100/10      evaluates to  10.0
7/2.5       evaluates to  2.8
100/32      evaluates to  3.125
13.5/1.5    evaluates to  9.0
```

Please note that in older version of Python (2.x) , the / operator worked differently.

5. For Boolean values *True* and *False*, recall that Python internally takes values 1 and 0 (zero) respectively so *True + 1* will give you 2 :).

5. Floor Division operator //

Python also offers another division operator //, which performs the floor division. The floor division is the division in which only the whole part of the result is given in the output and the fractional part is truncated.

To understand this, consider the third example of division given in division operator / , *i.e.,*

`a = 15.9 , b = 3,`
`a/b` evaluates to `5.3`.

Now if you change the division operator /, with floor division operator // in above expression, *i.e.,*

`If a = 15.9 , b = 3,`
`a//b will evaluate to 5.0`

See, the Fractional part 0.3 is discarded from the actual result 5.3

Consider some more examples :

```
100//32   evaluates to   3.0
7//3      evaluates to   2
6.5//2    evaluates to   3.0
```

The operands may be of number types.

NOTE

Floor division (//) truncates fractional remainders and gives only the whole part as the result.

6. Modulus operator %

The % operator finds the modulus (*i.e.,* remainder but pronounced as *mo-du-lo*) of its first operand relative to the second. That is, it produces the remainder of dividing the first operand by the second operand.

For example,

`19 % 6` evaluates to `1`, since 6 goes into 19 three times with a remainder `1`.

Similarly,

`7.2 % 3` will yield `1.2`
`6 % 2.5` will yield `1.0`

The operands may be of *number* types.

7. Exponentiation operator **

The exponentiation operator ** performs exponentiation (power) calculation, *i.e.,* it returns the result of a number raised to a power (exponent). *For example,*

`4 ** 3` evaluates to 64 (4^3)
`a ** b (a = 7, b = 4)`
evaluates to `2401` (a^b i.e., 7^4).
`x ** 0.5 ( x = 49.0)`
evaluates to `7.0`. ($x^{0.5}$, i.e., $\sqrt{x}$, i.e., $\sqrt{49}$)
`27.009 ** 0.3`
evaluates to `2.68814413570761`. ($27.009^{0.3}$)

The operands may be of number types.

Example 3.1 *What will be the output produced by the following code ?*

```
A, B, C, D = 9.2, 2.0, 4, 21
print (A/4)
print (A // 4)
print (B ** C)
print (D // B)
print (A % C)
```

Solution.

```
2.3
2.0
16.0
10.0
1.2
```

Example 3.2 *Print the area of a circle of radius 3.75 metres.*

Solution.

```
Radius = 3.75
Area = 3.14159 * Radius ** 2
print (Area, 'sq. metre')
```

Table 3.2 *Binary Arithmetic Operators*

Symbol	Name	Example	Result	Comment
+	addition	6 + 5 5 + 6	11 11	adds values of its two operands.
–	subtraction	6 – 5 5 – 6	1 – 1	subtracts the value of right operand from left operand.
*	multiplication	5 * 6 6 * 5	30 30	multiplies the values of its two operands.
/	division	60/5	12	divides the value of left operand with the value of right operand and returns the result as a **float** value.
%	Modulus (pronounced mo-du-lo) or Remainder	60%5 6%5	0 1	divides the two operands and gives the remainder resulting.
//	Floor division	7.2 // 2	3.0	divides and truncates the fractional part from the result.
**	Exponentiation (Power)	2.5 ** 3	15.625	returns base raised to power exponent. (2.5^3 here)

Negative Number Arithmetic in Python

Arithmetic operations are straightforward even with negative numbers, especially with non-division operators *i.e.*,

```
-5 + 3      will give you    2
-5 - 3      will give you    -8
-5 * 3      will give you    -15
-5 ** 3     will give you    -125
```

But when it comes to division and related operators (/, //, %), mostly people get confused. Let us see how Python evaluates these. To understand this, we recommend that you look at the operation shown in the adjacent screenshot and then look for its working explained below, where the result is shown shaded

(*a*) $-3\overline{)\,5\,(}\,-2$ with $\frac{6}{-1}$

(*b*) $3\overline{)\,-5\,(}\,-2$ with $\frac{-6}{+1}$

(*c*) $4\overline{)\,-7\,(}\,-1.75$ with $\frac{-4}{-3}$, $\frac{-3}{0}$

(*d*) $4\overline{)\,-7\,(}\,-2$ with $\frac{-8}{1}$

(*e*) $4\overline{)\,-7\,(}\,-2$ with $\frac{-8}{+1}$

(*f*) $-4\overline{)\,7\,(}\,-2$ with $\frac{8}{-1}$

(*d*) $-4\overline{)\,7\,(}\,-2$ with $\frac{8}{-1}$

IPython console

Console 1/A

```
In [67]: 5// -3
Out[67]: -2

In [68]: -5 //3
Out[68]: -2

In [69]: -7 / 4
Out[69]: -1.75

In [70]: -7 // 4
Out[70]: -2

In [71]: -7 % 4
Out[71]: 1

In [72]: 7 % -4
Out[72]: -1

In [73]: 7 // -4
Out[73]: -2
```

3.4.1C Augmented Assignment Operators[6]

You have learnt that Python has an assignment operator = which assigns the value specified on RHS to the variable/object on the LHS of = . Python also offers augmented assignment arithmetic operators, which combine the impact of an arithmetic operator with an assignment operator, *e.g.*, if you want to add value of *b* to value of *a* and assign the result to *a*, then instead of writing

```
a = a + b
```

you may write

```
a += b
```

To add value of *a* to value of *b* add assign the result to *b*, you may write

```
b += a                    # instead of b = b + a
```

Operation	Description	Comment
$x += y$	$x = x + y$	Value of y added to the value of x and then result assigned to x
$x -= y$	$x = x - y$	Value of y subtracted from the value of x and then result assigned to x
$x *= y$	$x = x * y$	Value of y multiplied to value of x and then result assigned to x
$x /= y$	$x = x / y$	Value of y divides value of x and then result assigned to x
$x //= y$	$x = x // y$	Value of y does floor division to value of x and then result assigned to x
$x **= y$	$x = x ** y$	x^y computed and then result assigned to x
$x \%= y$	$x = x \% y$	Value of y divides value of x and then remainder assigned to x

These operators can be used anywhere that ordinary assignment is used. Augmented assignment doesn't violate *mutability*. Therefore, writing $x += y$ creates an entirely new object x with the value $x + y$.

3.4.2 Relational Operators

In the term *relational operator*, relational refers to the relationships that values (or operands) can have with one another. Thus, the relational operators determine the relation among different operands. Python provides six relational operators for comparing values (thus also called *comparison operators*). If the comparison is true, the relational expression results into the Boolean value *True* and to Boolean value *False*, if the comparison is false.

The *six* relational operators are :

<	less than,	<=	less than or equal to,	==	equal to
>	greater than,	>=	greater than or equal to,	!=	not equal to,

Relational operators work with nearly all types of data in Python, such as numbers, strings, lists, tuples etc.

Relational operators work on following principles :

- For *numeric types*, the values are compared after removing trailing zeros after decimal point from a floating point number. *For example*, 4 and 4.0 will be treated as equal (after removing trailing zeros from 4.0, it becomes equal to 4 only).

6. Please note this style of combining assignment with other operator also works with bitwise operators.

- Strings are compared on the basis of lexicographical ordering (ordering in dictionary).
  - Capital letters are considered lesser than small letters, *e.g.*, 'A' is less than 'a' ; 'Python' is not equal to 'python' ; 'book' is not equal to 'books'.

    Lexicographical ordering is implemented via the corresponding codes or **ordinal values** (*e.g.*, ASCII code or Unicode code) of the characters being compared. That is the reason 'A' is less than 'a' because ASCII value of letter 'A' (65) is less than 'a'(97). You can check for ordinal code of a character yourself using **ord(<character>)** function (*e.g.*, **ord('A')**)
  - For the same reason, you need to be careful about nonprinting characters like spaces. Spaces are real characters and they have a specific code (ASCII code 32) assigned to them. If you are comparing two strings that appear same to you but they might produce a different result – if they have some spaces in the beginning or end of the string.
    See screenshot on the right.

```
In [14]: 'Apple' > ' Apple'
Out[14]: True

In [15]: 'Apple' == ' Apple'
Out[15]: False
```

Results different from anticipated results because the second string contains a space in the beginning and space's ordinal value (32) is less than 'A' 's ordinal value (65)

- Two lists and similarly two tuples are equal if they have same elements in the same order.
- Boolean *True* is equivalent to 1 (numeric one) and Boolean *False* to 0 (numeric zero) for comparison purposes.

For instance, consider the following relational operations.

Given $a = 3,\ b = 13,\ p = 3.0$

$c = 'n',\ d = 'g',\ e = 'N',$

$f = 'god',\ g = 'God',\ h = 'god',\ j = 'God',\ k = "Godhouse",$

$L = [1, 2, 3],\ M = [2, 4, 6],\ N = [1, 2, 3]$

$O = (1, 2, 3),\ P = (2, 4, 6),\ Q = (1, 2, 3)$

Check Point

3.4

1. What is the function of operators ? What are arithmetic operators ? Give their examples.
2. How is 'unary +' operator different from '+' operator ? How is 'unary –' operator different from '–' operator ?
3. What are binary operators ? Give examples of arithmetic binary operators.
4. What does the modulus operator % do ? What will be the result of 7.2 % 2.1 and 8 % 3 ?
5. What will be the result of 5.0/3 and 5.0//3 ?
6. How will you calculate 4.5^5 in Python ?
7. Write an expression that uses a relational operator to return *True* if the variable *total* is greater than or equal to variable *final*.

`a < b`	`will return True`	
`c < d`	`will return False`	
`f < h`	`will return False`	
`f == h`	`will return True`	
`c == e`	`will return False`	
`g == j`	`will return True`	
`"God" < "Godhouse"`	`will return True`	*Both match up to the letter 'd ' but 'God 'is shorter than 'Godhouse' so it comes first in the dictionary.*
`"god" < "Godhouse"`	`Will return False`	
`a == p`	`will return True`	*The length of the strings does not matter here. Lower case g is greater than upper case G*
`L == M`	`will return False`	
`L == N`	`will return True`	
`O == P`	`will return False`	
`O == Q`	`will return True`	
`a == True`	`will return False`	
`0 == False`	`will return True`	
`1 == True`	`will return True`	

Table 3.3 summarizes the action of these relational operators.

Table 3.3 *Relational Operators in Python*

p	q	p<q	p<=q	p==q	p>q	p>=q	p!=q
3	3.0	False	True	True	False	True	False
6	4	True	False	False	True	True	True
'A'	'A'	False	True	True	False	True	False
'a'	'A'	False	False	False	True	True	True

IMPORTANT

While using floating-point numbers with relational operators, you should keep in mind that floating point numbers are approximately presented in memory in binary form up to the allowed precision (15 digit precision in case of Python). This approximation may yield unexpected results if you are comparing floating-point numbers especially for equality (==). Numbers such as 1/3 etc., cannot be fully represented as binary as it yields 0.3333... etc. and to represent it in binary some approximation is done internally.

Consider the following code for to understand it :

```
In [18]: 0.1+0.1+0.1 == 0.3
Out[18]: False

In [19]: print(0.1+0.1+0.1)
0.30000000000000004

In [20]: print(0.3)
0.3
```

Notice , Python returns False when you compare **0.1+ 0.1 +0.1** with **0.3.**

See, it does not give you **0.3** when you print the result of expression **0.1+ 0.1 +0.1** ***(because of floating pt approximation)*** and this is the reason the result is False for quality comparison of **0.1+ 0.1 +0.1** and **0.3**

Thus, you should avoid floating point equality comparisons as much as you can.

Relational Operators with Arithmetic Operators

The relational operators have a lower precedence than the arithmetic operators.

That means the expression

```
a + 5 > c - 2        ...    expression 1
```

corresponds to

```
(a + 5) > (c - 2)    ...    expression 2
```

and not the following

```
a + (5 > c) - 2      ...    expression 3
```

Expression 1 means the *expression* 2 and not the *expression* 3.

Though relational operators are easy to work with, yet while working with them, sometimes you get unexpected results and behaviour from your program. To avoid so, I would like you to know certain tips regarding relational operators.

Check Point

3.5

1. Given that $i = 4$, $j = 5$, $k = 4$, what will be the result of following expressions ?
   (i) $i < k$ (ii) $i < j$ (iii) $i <= k$ (iv) $i == j$
   (v) $i == k$ (vi) $j > k$ (vii) $j >= i$
   (viii) $j != i$ (ix) $j <= k$
2. What will be the order of evaluation for following expressions ?
   (i) $i + 5 >= j - 6$ (ii) $s * 10 < p ** 2$
   (iii) $i < j + k > 1 - n$
3. How are following two expressions different ? (i) ans = 8 (ii) ans == 8

A very common mistake is to use the assignment operator = in place of the relational operator == . Do not confuse testing the operator == with the assignment operator (=). For instance, the expression :

```
value1 == 3
```

tests whether *value1* is equal to 3 ? The expression has the value *True* if the comparison is true otherwise it is *False*. But the expression

```
value1 = 3
```

assigns 3 to *value1* ; no comparison takes place.

NOTE

In Python, arithmetic operators have higher precedence over relational operators *i.e.*, $y + x > y * 2$ would be evaluated internally as $(y + x) > (y * 2)$.

TIP

Do not confuse the = and the == operators.

3.4.3 Identity Operators

There are two identity operators in Python *is* and *is not*. The identity operators are used to check if both the operands reference the same object memory *i.e.*, the identity operators compare the memory locations of two objects and return *True* or *False* accordingly.

Operator	Usage	Description
is	a is b	returns *True* if both its operands are pointing to same object (*i.e.*, both referring to same memory location), returns *False* otherwise.
is not	a *is not* b	returns *True* if both its operands are pointing to different objects (*i.e.*, both referring to different memory location), returns *False* otherwise.

Consider the following examples :

```
A = 10
B = 10
A is B
```

will return True because both are referencing the memory address of value 10

You can use id( ) to confirm that both are referencing same memory address.

```
In [34]: a = 235

In [35]: b = 240

In [36]: c = 235

In [37]: a is b
Out[37]: False

In [38]: a is c
Out[38]: True

In [39]: print( id(a), id(b), id(c) )
492124000 492124080 492124000
```

a is b returns *False* because a and b are referring to different objects (235 and 240)

a is c returns *True* because both a and c are referring to same object (235)

The ids (*id( )*) of a, b and c tell that a and c are referring to same object (their memory addresses are same) but b is referring to a different object as its memory address is different from the other two

Now if you change the value of *b* so that it is not referring to same integer object, then expression a is b will return *True* :

```
In [40]: b = b - 5

In [41]: a is b
Out[41]: True

In [42]: print( id(a), id(b), id(c) )
492124000 492124000 492124000
```

Now b is also pointing to same integer object(235) thus a is b is giving True this time.

Their *ids* also reflect the same I.e., all **a** , **b** and **c** are referring to same memory location

The **is not** operator is opposite of the **is** operator. It returns *True* when both its operands are not referring to same memory address.

3.4.3A Equality (==) and Identity (is) – Important Relation

You have seen in above given examples that when two variables are referring to same value, the **is** operator returns **True**. When the **is** operator returns **True** for two variables, it implicitly means that the equality operator will also return **True**. That is, expression a is b as **True** means that a == b will also be **True**, always. See below :

```
In [58]:  print (a, b )
235 235

In [59]: a is b
Out[59]: True

In [60]: a == b
Out[60]: True
```

But it is not always true other way round. That means there are some cases where you will find that the two objects are having just the same value *i.e.*, == operator returns **True** for them but the **is** operator returns False.

See in the screenshots shown here.

```
In [45]: s1 = 'abc'

In [46]: s2 = input("Enter a string:")

Enter a string:abc

In [47]: s1 == s2
Out[47]: True

In [48]: s1 is s2
Out[48]: False

In [49]: s3 = 'abc'

In [50]: s1 is s3
Out[50]: True
```

- The strings **s1** and **s2** although have the same value **'abc'** in them
- The == operator also returns ***True*** for **s1 == s2**
- But the **is** operator returns ***False*** for **s1 is s2**

Similarly,

```
In [51]: i = 2+3.5j

In [52]: j = 2+3.5j

In [53]: i is j
Out[53]: False
```

- Objects *I* and *J* store the same complex number value 2+3.5 J in them
- But Is operator returns *False* for I Is J

Also,

```
In [54]: k = 3.5

In [55]: l = float(input("Enter a value:"))

Enter a value:3.5

In [56]: k == l
Out[56]: True

In [57]: k is l
Out[57]: False
```

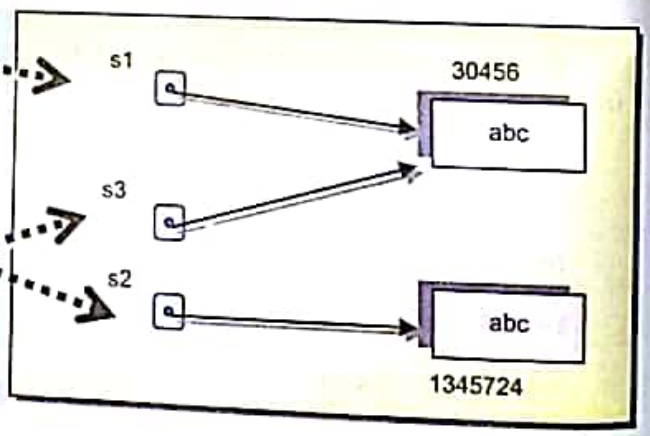
- The variables *k* and *l* both store float value 3.5 (k has been assigned 3.5 and l has taken this value through input() and float())
- But k == l returns True and k is l returns False

The reason behind this behaviour is that there are a few cases where **Python creates two different objects that both store the same value**. These are :

- input of strings from the console;
- writing integers literals with many digits (very big integers);
- writing floating-point and complex literals.

Following figure illustrates one of the above given screenshots.

```
In [45]: s1 = 'abc'

In [46]: s2 = input("Enter a string:")

Enter a string:abc

In [47]: s1 == s2
Out[47]: True

In [48]: s1 is s2
Out[48]: False

In [49]: s3 = 'abc'

In [50]: s1 is s3
Out[50]: True
```



Most of the times we just need to check whether the two objects refer to the same value or not – in this case the equality operator (==) is sufficient for this test. However, in advanced programs or in your projects, you may need to check whether they refer to same memory address or not – in this case, you can use the is operator.

NOTE

If the Is operator returns True for two objects then the == operator must return True for the same objects, *i.e.*, If two references refer to the same memory address, then their values are always same.

3.4.4 Logical Operators

An earlier section discussed about relational operators that establish relationships among the values. This section talks about logical operators, the Boolean logical operators (**or, and, not**) that refer to the ways these relationships (among values) can be connected. Python provides *three* logical operators to combine existing expressions. These are **or, and,** and **not.**

Before we proceed to the discussion of logical operators, it is important for you to know about **Truth Value Testing,** because in some cases logical operators base their results on truth value testing.

3.4.4A Truth Value Testing

Python associates with every value type, some truth value (the *truthiness*), *i.e.*, Python internally categorizes them as *true* or *false*. Any object can be tested for truth value. Python considers following values *false*, (*i.e.*, with truth-value as *false*) and *true* :

Values with truth value as false	*Values with truth value as true*
None	All other values are considered *true*.
False (Boolean value False)	
zero of any numeric type, for example, 0, 0.0, 0j	
any empty sequence, for example, '', ( ), [ ] (please note, '' is empty string ; ( ) is empty tuple ; and [ ] is empty list)]	
any empty mapping, for example, { }	
The result of a relational expression can be *True* or *False* depending upon the values of its operands and the comparison taking place.	

Do not confuse between Boolean values *True, False* and truth values (truthiness values) *true, false*. Simply put truth-value tests for zero-ness or emptiness of a value. Boolean values belong to just one data type, *i.e.*, Boolean type, whereas we can test truthiness for every value object in Python. But to avoid any confusion, we shall be giving truth values *true* and *false* in small letters and with a subscript *tval*, *i.e.*, now on in this chapter $true_{tval}$ and $false_{tval}$ will be referring to truth-values of an object.

The utility of Truth Value testing will be clear to you as we are discussing the functioning of logical operators.

3.4.4B The or Operator

The **or** operator combines *two* expressions, which make its *operands*. The **or** operator works in these ways :

(*i*) relational expressions as *operands* (*ii*) numbers or strings or lists as *operands*

(i) Relational expressions as operands

When **or** operator has its operands as relational expressions (*e.g.*, $p > q$, $j != k$, etc.) then the **or** operator performs as per following principle :

The or operator evaluates to True if either of its (relational) operands evaluates to True ; False if both operands evaluate to False.

That is :

x	y	x or y
False	False	False
False	True	True
True	False	True
True	True	True

Following are some examples of this *or* operation :

(4 == 4) or (5 == 8) results into **True** because first expression (4 == 4) is *True*.

5 > 8 or 5 < 2 results into **False** because both expressions 5 > 8 and 5 < 2 are *False*.

(*ii*) Numbers / strings / lists as operands[7]

When or operator has its operands as numbers or strings or lists (*e.g.*, **'a' or '', 3 or 0**, etc.) then the or operator performs as per following principle :

In an expression **x or y**, *if first operand, (i.e., expression* **x**) *has* ***false***$_{tval}$, *then return second operand* **y** *as result, otherwise return* **x**.

That is :

x	y	x or y
$false_{tval}$	$false_{tval}$	*y*
$false_{tval}$	$true_{tval}$	*y*
$true_{tval}$	$false_{tval}$	*x*
$true_{tval}$	$true_{tval}$	*x*

NOTE

In general, ***True*** and ***False*** represent the Boolean values and ***true*** and ***false*** represent truth values.

Examples

Operation	Results into	Reason
0 or 0	0	first expression (0) has $false_{tval}$, hence second expression 0 is returned.
0 or 8	8	first expression (0) has $false_{tval}$, hence second expression 8 is returned.
5 or 0.0	5	first expression (5) has $true_{tval}$, hence first expression 5 is returned.
'hello' or ''	'hello'	first expression ('hello') has $true_{tval}$, hence first expression '*hello*' is returned
'' or 'a'	'a'	first expression ('') has $false_{tval}$, hence second expression '*a*' is returned.
'' or ''	''	first expression ('') has $false_{tval}$, hence second expression '' is returned.
'a' or 'j'	'a'	first expression ('a') has $true_{tval}$, hence first expression '*a*' is returned.

How the truth value is determined ? – refer to section 3.4.4A above.

7. **This type of or functioning applies to any value which is not a relational expression but whose truth-ness can be determined by Python.**

The or operator will test the second operand **only if** the first operand is *false*, otherwise ignore it ; even if the second operand is logically wrong *e.g.*,

```
20 > 10 or "a" + 1 > 1
```

will give you result as

```
True
```

without checking the second operand of or *i.e.*, "*a*" + 1 > 1, which is syntactically wrong – you cannot add an integer to a string.

NOTE

The or operator will test the second operand only if the first operand is *false*, otherwise ignore it.

3.4.4C The and Operator

The and operator combines *two* expressions, which make its operands. The and operator works in these ways :

(*i*) relational expressions as operands

(*ii*) numbers or strings or lists as operands

(i) Relational expressions as operands

When and operator has its operands as relational expressions (*e.g.*, $p > q$, $j != k$, etc.) then the and operator performs as per following principle :

The and operator evaluates to True if both of its (relational) operands evaluate to True ; False if either or both operands evaluate to False.

That is :

x	*y*	*x and y*
False	False	False
False	True	False
True	False	False
True	True	True

Following are some examples of the and operation :

(4 == 4) and (5 == 8)	results into False because first expression (4 == 4) is *True* but second expression (5 == 8) evaluates to *False*. Both operands have to result into *True* in order to have the final results as *True*
5 > 8 and 5 < 2	results into False because first expression : 5 > 8 evaluates to *False*.
8 > 5 and 2 < 5	results into True because both operands : 8 > 5 and 2 < 5 evaluate to *True*

(ii) Numbers / strings / lists as operands[8]

When and operator has its operands as numbers or strings or lists (*e.g.*, *'a' or ''*, 3 or 0, etc.) then the and operator performs as per following principle :

In an expression x and y, if first operand, (i.e., expression x) has $false_{tval}$*, then return first operand x as result, otherwise return y.*

8. This type of or functioning applies to any value which is not a relational expression but whose truth-ness can be determined by Python.

That is :

x	y	x and y
$false_{trul}$	$false_{trul}$	x
$false_{trul}$	$true_{trul}$	x
$true_{trul}$	$false_{trul}$	y
$true_{trul}$	$true_{trul}$	y

Examples

Operation	Results into	Reason
0 and 0	0	first expression (0) has $false_{trul}$, *hence first expression 0* is returned.
0 and 8	0	first expression (0) has $false_{trul}$, hence first expression *0* is returned.
5 and 0.0	0.0	first expression (5) has $true_{trul}$, hence second expression *0.0* is returned.
'hello' and ''	''	first expression ('hello') has $true_{trul}$, hence second expression '' is returned
'' and 'a'	''	first expression ('') has $false_{trul}$, hence first expression '' is returned.
'' and ''	''	first expression ('') has $false_{trul}$, hence first expression '' is returned.
'a' and 'j'	'j'	first expression ('a') has $true_{trul}$, hence second expression *'j'* is returned.

How the truth value is determined ? – refer to section 3.4.4A above.

IMPORTANT

The **and** operator will test the second operand **only** if the first operand is *true*, otherwise ignore it ; even if the second operand is logically wrong *e.g.*,

```
10 > 20 and "a" + 10 < 5
```

will give you result as

```
False
```

ignoring the second operand completely, even if it is wrong – you cannot add an integer to a string in Python.

NOTE

The and operator will test the second operand only if the first operand is *true*, otherwise ignore it.

3.4.4D The not Operator

The Boolean/logical **not** operator, works on single expression or operand *i.e.*, it is a unary operator. The logical **not** operator negates or reverses the truth value of the expression following it *i.e.*, if the expression is *True* or $true_{trul}$, then *not expression* is *False*, and vice versa. Unlike **'and'** and **'or'** operators that can return number or a string or a list etc. as result, the **'not'** operator returns always a Boolean value *True* or *False*.

Consider some examples below :

not 5	results into *False* because 5 is non-zero (*i.e.*, $true_{trul}$)
not 0	results into *True* because 0 is zero (*i.e.*, $false_{trul}$)
not –4	results into *False* because – 4 is non zero thus $true_{trul}$.
not (5 > 2)	results into *False* because the expression 5 > 2 is *True*.
not (5 > 9)	results into *True* because the expression 5 > 9 is *False*.

NOTE

Operator not has a lower priority than non-Boolean operators, so not a == b is interpreted as not (a == b), and a == not b is a syntax error.

Following table summarizes the logical operators.

Table 3.4 *The Logical Operators*

Operation	Result	Notes
x or y	if x is $false_{bool}$, then return y as result, else x	It (or) only evaluates the second argument if the first one is $false_{bool}$
x and y	if x is $false_{bool}$, then x as result, else y	It (and) only evaluates the second argument if the first one is $true_{bool}$
not x	if x is $false_{bool}$, then return *True* as result, else *False*	not has a lower priority than non-Boolean operators.

Chained Comparison Operators

While discussing Logical operators, Python has something interesting to offer. You can chain multiple comparisons which are like shortened version of larger Boolean expressions. Let us see how. Rather than writing 1 < 2 and 2 < 3, you can even write 1 < 2 < 3, which the chained version of earlier Boolean expression.

The above statement will check if *1 was less than* 2 and *if* 2 *was less than 3.*

Let's look at a few examples of using chains :

```
>>> 1 < 2 < 3          is equivalent to      >>> 1 < 2 and 2 < 3
True                                          True
```

As per the property of and, the expression 1 < 3 will be first evaluated and ***if only it is True***, then only the next chained expression 2 < 3 will be evaluated.

Similarly consider some more examples :

```
>>> 11 < 13 > 12
True
```

The above expression checks if *13* is larger than both the other numbers ; it is the shortened version of **11 < 13** and 13 > 12.

3.4.5 Bitwise Operators

Python also provides another category of operators - *bitwise operators, which* are similar to the logical operators, except that they work on a smaller scale – on binary representations of data. Bitwise operators are used to change individual bits in an operand.

Python provides following Bitwise operators.

Table 3.5 *Bitwise Operators*

Operator	Operation	Use	Description
&	bitwise and	op1 & op2	The AND operator compares two bits and generates a result of 1 if both bits are 1; otherwise, it returns 0.
|	bitwise or	op1 | op2	The OR operator compares two bits and generates a result of 1 if the bits are complementary; otherwise, it returns 0.
^	bitwise xor	op1 ^ op2	The EXCLUSIVE-OR (*XOR*) operator compares two bits and returns 1 if either of the bits are 1 and it gives 0 if both bits are 0 or 1.
~	bitwise complement	~op1	The COMPLEMENT operator is used to invert all of the bits of the operand

Let us examine them one by one.

3.4.5A The AND operator &

When its operands are numbers, the & operation performs the bitwise AND function on each parallel pair of bits in each operand. The AND function sets the resulting bit to 1 if the corresponding bit in both operands is 1, as shown in the following Table 3.6.

Table 3.6 *The Bitwise AND (&) Operation*

op1	op2	Result
0	0	0
0	1	0
1	0	0
1	1	1

For AND operations, 1 AND 1 produces 1.
Any other combination produces 0.

Suppose that you were to AND the values 13 and 12, like this : 13 & 12. The result of this operation is 12 because the binary representation of 12 is 1100, and the binary representation of 13 is 1101. You can use **bin( )** to get binary representation of a number.

```
13 & 12    1101
           1100
           ----
           1100
```

```
In [76]: bin(13)
Out[76]: '0b1101'

In [77]: bin(12)
Out[77]: '0b1100'

In [78]: 13 & 12
Out[78]: 12

In [79]: bin(13 & 12)
Out[79]: '0b1100'
```

If both operand bits are 1, the AND function sets the resulting bit to 1 ; otherwise, the resulting bit is 0. So, when you line up the two operands and perform the AND function, you can see that the two high-order bits (the two bits farthest to the left of each number) of each operand are 1. Thus, the resulting bit in the result is also 1. The low-order bits evaluate to 0 because either one or both bits in the operands are 0.

3.4.5B The inclusive OR operator |

When both of its operands are numbers, the | operator performs the inclusive OR operation. Inclusive OR means that if either of the two bits is 1, the result is 1. The following Table 3.7 shows the results of inclusive OR operations.

Table 3.7 *The Inclusive OR (|) Operation*

op1	op2	Result
0	0	0
0	1	1
1	0	1
1	1	1

For OR operations, 0 OR 0 produces 0.
Any other combination produces 1.

```
13 | 12   0000 1101
          0000 1100
          ---------
          0000 1101
```

```
In [80]: bin(13)
Out[80]: '0b1101'

In [81]: bin(12)
Out[81]: '0b1100'

In [82]: bin(13 | 12)
Out[82]: '0b1101'

In [83]: 13 | 12
Out[83]: 13
```

3.4.5C The eXclusive OR (XOR) operator ^

Exclusive OR means that if the two operand bits are different, the result is 1 ; otherwise the result is 0. The following Table 3.8 shows the results of an eXclusive OR operation.

Table 3.8 *The eXclusive OR (^) Operation*

op1	op2	Result
0	0	0
0	1	1
1	0	1
1	1	0

```
13 ^ 12     0000 1101
            0000 1100
            ---------
            0000 0001
```

```
In [84]: bin(13)
Out[84]: '0b1101'

In [85]: bin(12)
Out[85]: '0b1100'

In [86]: 13 ^ 12
Out[86]: 1

In [87]: bin(13 ^ 12)
Out[87]: '0b1'
```

For XOR operations, 1 XOR 0 produces 1, as does 0 XOR 1. (All these operations are commutative.) Any other combination produces 0.

3.4.5D The Complement Operator ~

The complement operator inverts the value of each bit of the operand : if the operand bit is 1 the result is 0 and if the operand bit is 0 the result is 1.

Table 3.9 *The Complement (~) Operation*

op1	Result
0	1
1	0

This is binary code of –13 in 2's complement form.[9]

```
~12     0000 1100
        ---------
        1111 0011
        = - (0000 1101)
```

```
In [91]: bin(12)
Out[91]: '0b1100'

In [92]: ~12
Out[92]: -13

In [93]: bin(13)
Out[93]: '0b1101'

In [94]: bin(~12)
Out[94]: '-0b1101'
```

3.4.6 Operator Precedence

When an expression or statement involves multiple operators, Python resolves the order of execution through Operator Precedence. The chart of operator precedence from highest to lowest for the operators covered in this chapter is given below.

Operator	Description
()	Parentheses (grouping)
**	Exponentiation
~x	Bitwise nor
+x, -x	Positive, negative (unary + , –)
*, /, //, %	Multiplication, division, floor division, remainder
+, –	Addition, subtraction
&	Bitwise and
^	Bitwise XOR
|	Bitwise OR
<, <=, >, >=, <>, !=, ==, is, is not	Comparisons (Relational operators), identity operators
not x	Boolean NOT
and	Boolean AND
or	Boolean OR

Highest ↓ Lowest

[9] To obtain the number whose 2's complement is given, you can calculate its 2's complement again by following this rule – starting from right to left , copy all the bits as it is UNTIL you find first 1, then invert all other bits. As per this rule, we can calculate 2's complement of 11110011 as 00001101, which is 13.

Operator Associativity

Python allows multiple operators in a single expression as you have learnt above, *e.g.*, a < b + 2 < c or p < q > r etc. If the operators used in an expression have different precedence, there is not any problem as Python will evaluate the operator with higher precedence first. BUT what if the expression contains two operators that have the same precedence?

In that case, **associativity** helps determine the order of operations.

Associativity is the order in which an expression (having multiple operators of same precedence) is evaluated. Almost all the operators have **left-to-right associativity except exponentiation (**), which has right-to left associativity.** That means, in case of multiple operators with same precedence, other than **, in same expression – the operator on the left is evaluated first and then the operator on its right and so on.

For example, multiplication operator (*), division operator (/) and floor division operator (//) have the same precedence. So, if we have an expression having these operators simultaneously, then the same-precedence-operators will be evaluated in left-to-right order.

For example,

```
In[1] : 7 * 8 / 5 // 2
Out[1]: 5.0

In[2] : (((7 * 8) / 5 ) // 2 )
Out[2]: 5.0

In[3] : 7 * ( (8 / 5) // 2)
Out[3]: 0.0

In[4]: 7 * (8 / (5 // 2) )
Out[4]: 28.0
```

This first expression is evaluated in left to right order of operators as evident from the 2nd expression's evaluation that clearly marks this order of evaluation

An expression having multiple ** operators is evaluated from right to left, *i.e.*,

2 ** 3 ** 4 will be evaluated as 2 ** (3 **4) and NOT AS (2**3)**4

Consider following example :

```
In [11]: 3 ** 3 ** 2
Out[11]: 19683

In [12]: 3 ** (3 ** 2)
Out[12]: 19683

In [13]: (3 ** 3) ** 2
Out[13]: 729
```

*See the default order of evaluation (**first expression**) **MATCHES** with the **second expression** where parentheses are added as per right-to-left associativity order and **NOT LIKE third expression** that has parentheses from left-to-right order, because exponentiation (**) has right-to-left associativity*

NOTE

Associativity is the order in which an expression having multiple operators of same precedence, is evaluated.

All operators are left associative except *exponentiation (right associative).*

OPERATORS IN PYTHON

Progress In Python 3.2

This practical session aims at strengthening operators' concepts. It involves both interactive mode and script mode. For better understanding of the concepts, it would be better if you first perform the *interactive mode practice questions* followed by *script mode practice questions.*

Please check the practical component-book – Progress in Computer Science with Python and fill it there in PriP 3.2 under Chapter 3 after practically doing it on the computer.

>>>❖<<<

Check Point

3.6

1. What is the function of logical operators ? Write an expression involving a logical operator to test if *marks* are 55 and *grade* is '*B*'.
2. What is the order of evaluation in the following expressions :
   (*i*) a > b or b <= d ?
   (*ii*) x == y and y >= m ?
   (*iii*) a > b < c > d
3. What is the result of following expression : *a* >= *b* and (*a* + *b*) > *a* if
   (*i*) *a* = 3, *b* = 0 (*ii*) *a* = 7, *b* = 7 ?
4. What is the result of following expressions (*a* to *e*) if
   (*i*) check = 3, mate = 0.0
   (*ii*) check = 0, mate = −5
   (*iii*) check = ' ', mate = '*e*'
   (*iv*) check = 'meera', mate = ' '
   (*a*) check or mate (*b*) mate or check
   (*c*) check and not mate
   (*d*) check and mate (*e*) mate and check
   Evaluate each of the above expressions for all four sets of values.
5. Identify the order of evaluation in the following expression :
   4 * 5 + 7 * 2 − 8 % 3 + 4 and
   5.7 // 2 − 1 + 4 or not 2 == 4 and
   not 2 ** 4 > 6 * 2
6. a = 15.5, b = 15.5 then why is a is b is False while a == b is True ?

LET US REVISE

- *Operators are the symbols (or keywords sometimes) that represent specific operations.*
- *Arithmetic operators carry out arithmetic for Python. These are unary +, unary −, +, −, *, /, //, % and **.*
- *Unary + gives the value of its operand, unary − changes the sign of its operand's value.*
- *Addition operator + gives sum of is operand's value, − subtracts the value of second operand from the value of first operand, * gives the product of its operands' value. The operator / divides the first operand by second and returns a float result // performs the floor division, % gives the remainder after dividing first operand by second and ** is the exponentiation operator, i.e., it gives base raised to power.*
- *Relational operators compare the values of their operands. These are >, <, ==, <=, >= and != i.e., less than, greater than, equal to, less than or equal to, greater than or equal to and not equal to respectively.*
- *Bitwise operators are like logical operators but they work on individual bits.*
- *Identify operators (is, is not) compare the memory two objects are referencing.*
- *Logical operators perform comparisons on the basis of truth-ness of an expression or value. These are 'or', 'and' and 'not'.*
- *Boolean or relational expressions' truth value depends on their Boolean result True or False.*
- *Values' truth value depends on their emptiness or non-emptiness. Empty numbers (such as 0, 0.0, 0j etc.) and empty sequences (such as "", [ ], ( )) and None have truth-value as false and all others (non-empty ones) have truth-value as true.*

H. P

3.5 EXPRESSIONS

ATOM

An atom is something that has a value. Identifiers, literals, strings, lists, tuples, sets, dictionaries etc. are all atoms.

EXPRESSION

An *expression* in Python is any valid combination of operators and atoms. An expression is composed of one or more *operations*.

An expression in Python is any valid combination of *operators, literals* and *variables*. An expression is composed of one or more *operations*, with *operators, literals*, and *variables* as the constituents of expressions.

Python puts it in this way : a valid combination of **atoms** and **operators** forms a Python expression. In simplest words, **an *atom* is something that has a value.** So all of these are atoms in Python : *identifiers, literals* and *values-in-enclosures* such as quotes (''), parentheses, brackets, etc. *i.e., strings, tuples, lists, dictionaries, sets* etc.

The expressions in Python can be of any type : *arithmetic expressions, string expressions, relational expressions, logical expressions, compound expressions* etc.

- The types of operators and operands used in an expression determine the expression type.
- An expression can be **compound expression** too if it involves multiple types of operators, *e.g.*, $a + b > c * * d$ or $a * b < c * d$ is a *compound expression* as it involves *arithmetic* as well as *relational* as well as *logical* operators.

Let us talk about these one by one.

1. Arithmetic Expressions

Arithmetic expressions involve numbers (integers, floating-point numbers, complex numbers) and arithmetic operators.

2. Relational Expressions

An expression having literals and/or variables of any valid type and relational operators is a relational expression. *For example,* these are valid relational expressions :

```
x > y,  y <= z,  z <> x,  z == q,  x < y > z,  x == y <> z
```

3. Logical Expressions

An expression having literals and/or variables of any valid type and logical operators is a logical expression. *For example,* these are valid logical expressions :

```
a or b ,  b and c,  a and not b ,  not c or not b
```

4. String expressions

Python also provides two string operators + and *, when combined with string operands and integers, form string expressions.

- With operator +, *the concatenation operator,* the operands should be of string type only.
- With * operator, *the replication operator,* the operands should be one string and one integer.

For instance, following are some legal string expressions :

```
"and" + "then"          # would result into 'andthen' - concatenation
"and" * 2               # would result into 'andand' - replication
```

String manipulation is being covered in a separate chapter – ***chapter 5.***

3.5.1 Evaluating Expressions

In this section, we shall be discussing how Python evaluates different types of expressions : arithmetic, relational and logical expressions. String expressions, as mentioned earlier will be discussed in a separate chapter – ***chapter 5***.

3.5.1A Evaluating Arithmetic Expressions

You all are familiar with arithmetic expressions and their basic evaluation rules, right from your primary and middle-school years. Likewise, Python also has certain set of rules that help it evaluate an expression. Let's see how Python evaluates them.

Evaluating Arithmetic Expressions

To evaluate an arithmetic expression (with operator and operands), Python follows these rules :

- Determines the order of evaluation in an expression considering the operator precedence.
- As per the evaluation order, for each of the sub-expression (generally in the form of **<value> <operator><value>** *e.g.*, 13% 3)
  - Evaluate each of its operands or arguments.
  - Performs any implicit conversions (*e.g.*, promoting **int** to **float** or **bool** to **int** for arithmetic on mixed types). For implicit conversion rules of Python, read the text given after the rules.
  - Compute its result based on the operator.
  - Replace the subexpression with the computed result and carry on the expression evaluation.
  - Repeat till the final result is obtained.

Implicit type conversion (Coercion). An implicit type conversion is a conversion performed by the compiler without programmer's intervention. An implicit conversion is applied generally whenever differing data types are intermixed in an expression (mixed mode expression), so as not to lose information.

In a mixed arithmetic expression, Python converts all operands up to the type of the largest operand (***type promotion***). In simplest form, an expression is like ***op1 operator op2*** (*e.g.*, *x/y* or *p ** *a*). Here, if both arguments are standard numeric types, the following coercions are applied :

- If either argument is a complex number, the other is converted to complex ;
- Otherwise, if either argument is a floating point number, the other is converted to floating point ;
- No conversion if both operands are integers.

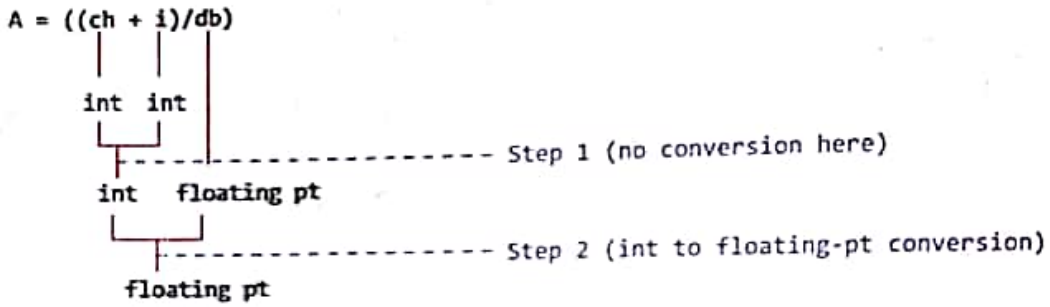
To understand this, consider the following example, which will make it clear how Python internally coerces (*i.e.*, promotes) data types in a mixed type arithmetic expression and then evaluates it.

Example 3.3 *Consider the following code containing mixed arithmetic expression. What will be the final result and the final data type ?*

```
ch = 5          # integer
i = 2           # integer
fl = 4          # integer
```

```
db = 5.0                    # floating point number
fd = 36.0                   # floating point number
A = (ch + i) / db           #expression 1
B = fd /db * ch /2          #expression 2
print (A)
print (B)
```

Solution. As per operator precedence, expression 1 will be internally evaluated as :



So overall, final datatype for **expression 1** will be **floating-point number** and the expression will be evaluated as :

((ch + i) / db)

((5 + 2)) / 5.0

((5 + 2)) / 5.0

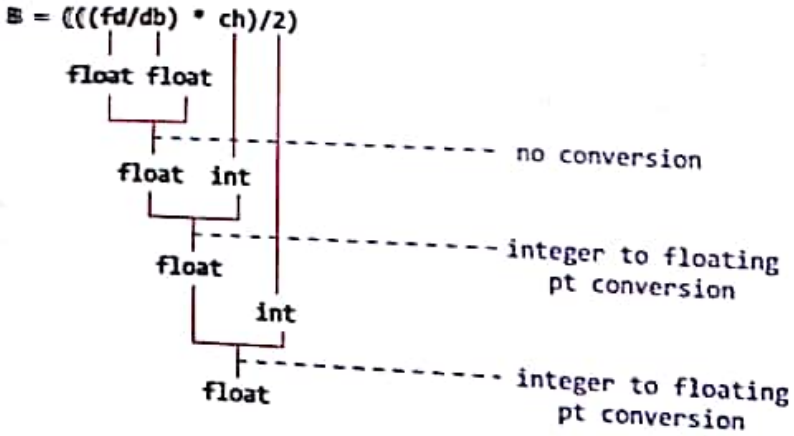
= (7) / 5.0

[*int to floating point conversion*]

= 7.0 / 5.0

A = 1.4

As per operator precedence, expression 2 will be internally evaluated as :



So, final datatype for expression 2 will be **floating point number.**

The expression, expression 2 will be evaluated as :

$$(((fd / db) * ch) / 2)$$
$$= (((36.0 / 5.0) * 5L / 2) \quad \text{[no conversion required]}$$
$$= ((7.2 * 5) / 2) \quad \text{[int to floating point conversion]}$$
$$= ((7.2 * 5.0) / 2)$$
$$= (36.0 / 2) \quad \text{[integer to floating point conversion]}$$
$$= 36.0 / 2.0$$
$$B = 18.0$$

The output will be
```
1.4
18.0
```

The final datatype of expression 1 will be **floating point number** and of expression 2, it will be floating-point number.

IMPORTANT In Python, if the operator is the division operator (/), the result will always be a floating point number, even if both the operands are of integer types (an *exception* to the rule). Consider following example that illustrates it.

Example 3.4 *Consider below given expressions what. Will be the final result and final data type ?*

(*a*)
```
a, b = 3, 6
c = b/a
```
(*b*)
```
a, b = 3, 6
c = b // a
```
(*c*)
```
a, b = 3, 6
c = b % a
```

Ans. (*a*) In expression

```
c = 6/3
c = 2.0
```

```
b   /   a
|       |
int     int
 └──┬──┘
  floating pt
```

Here, the operator is /, which always gives floating pt result.

(*b*) In expression

```
c = 6 // 3
c = 2
```

```
b   //  a
|       |
int     int
 └──┬──┘
    int
```

For other division related operations, // and %, if both operands are integers, result will be integer)

(*c*) In expression

```
c = 6 % 3
c = 0
```

```
b   %   a
|       |
int     int
 └──┬──┘
    int
```

You, yourself, can run these expressions in Python shell and then check the type of C using type (C) function.

3.5.1B Evaluating Relational Expressions (Comparisons)

All comparison operations in Python have the same priority, which is lower than that of any arithmetic operations. All relational expressions (comparisons) yield Boolean values only *i.e.*, *True* or *False*.

Further, chained expressions like $a < b < c$ have the interpretation that is conventional in mathematics *i.e.*, comparisons in Python are chained arbitrarily, *e.g.*, $a < b < c$ is internally treated as $a < b$ and $b < c$

For chained comparisons like $x < y <= z$ (which is internally equivalent to $x < y$ and $y <= z$), the common expression (the middle one, y here) is evaluated only once and the third expression (z here) is not evaluated at all when first comparison ($x < y$ here) is found to be *False*.

Example 3.5 *What will be the output of following statement when the inputs are :*

(*i*) $a = 10,\ b = 23,\ c = 23$ (*ii*) $a = 23,\ b = 10,\ c = 10$

```
print (a < b)
print (b <= c)
print (a < b <= c)
```

Solution.

For input combination (*i*), the output would be :

```
True
True
True
```

For input combination (*ii*), the output would be :

```
False
True
False
```

Example 3.6 *How would following relational expressions be internally interpreted by Python ?*

(*i*) $p > q < y$ (*ii*) $a <= N <= b$

Solution. (*i*) $(p > q)$ and $(q < y)$ (*ii*) $(a <= N)$ and $(N <= b)$

3.5.1C Evaluating Logical Expressions

Recall that the use of logical operators **and**, **or** and **not** makes a logical expression. While evaluating logical expressions, Python follows these rules :

(*i*) The precedence of logical operators is lower than the arithmetic operators, so constituent arithmetic sub-expression (if any) is evaluated first and then logical operators are applied, *e.g.*,

`25/5 or 2.0 + 20/10` will be first evaluated as : `5 or 4.0`

So, the overall result will be 5. (For logical operators' functioning, refer to section 3.4.3)

(*ii*) The precedence of logical operators among themselves is **not, and, or**. So, the expression *a* or *b* and not *c* will be evaluated as :

`(a or (b and (not c) ))` Similarly, following expression *p* and *q* or not *r* will be evaluated as : `((p and q) or (not r))`

Check Point 3.7

Evaluation of expression and type conversion

1. What is an expression ? How many different types of expressions can you have in Python ?
2. What is atom in context of expression ?
3. From the following expression, identify atoms and operators.

   `str(a + b > c + d > e + f or not g-h)`
4. What is type conversion (coercion) ? How does Python perform it ?

(*iii*) **Important.** While evaluating, Python minimizes internal work by following these rules :

(*a*) In **or** evaluation, Python only evaluates the second argument if the first one is ***false*** $_{val}$

(*b*) In **and** evaluation, Python only evaluates the second argument if the first one is ***true*** $_{val}$

For instance, consider the following examples :

- In expression **(3 < 5) or (5 < 2)**, since first argument (3 < 5) is *True*, simply its (first argument's) result is returned as overall result ; the **second argument (5 < 2) will not be evaluated at all.**

- In expression (5 < 3) or (5 < 2), since first argument (5 < 3) is *False*, it will now evaluate the second argument (5 < 2) and its (second argument's) result is returned as overall result.
- In expression (3 < 5) and (5 < 2), since first argument (3 < 5) is *True*, it will now evaluate the second argument (5 < 2) and its (second argument's) result is returned as overall result.
- In expression (5 < 3) and (5 < 2), since first argument (5 < 3) is *False*, simply its (first argument's) result is returned as overall result ; the second argument (5 < 2) will not be evaluated at all.

Example 3.7 *What will be the output of following expression ?*

```
(5 < 10) and (10 < 5) or (3 < 18) and not 8 < 18
```

Solution. False

Example 3.8 *'Divide by zero' is an undefined term. Dividing by zero causes an error in any programming language, but when following expression is evaluated in Python, Python reported no error and returned the result as True. Could you tell, why ?*

```
(5 < 10 ) or (50 < 100/0)
```

Solution. In or evaluation, firstly Python tests the first argument, *i.e.*, 5 < 10 here, which is *True*. In or evaluation, Python does not evaluate the second argument if the first argument is *True* and returns the result of first argument as the result of overall expression.

So, for the given expression, the second argument expression (50 < 100/0) is NOT EVALUATED AT ALL. That is why, Python reported no error, and simply returned *True*, the result of first argument.

Check Point

3.8

1. Are the following expressions equal ? Why/why not ?
   x or y and not z
   (x or y) and (not z))
   (x or (y and (not z)))
2. State when would only first argument be evaluated and when both first and second arguments are evaluated in following expressions if $a = 5$, $b = 10$, $c - 5$, $d = 0$.
   (i) $b > c$ and $c > d$
   (ii) $a <= b$ or $c <= d$
   (iii) $(b + c) <= a$ and not $(c < a)$
   (iv) $b < d$ and $d < a$
3. What is Type casting ?
4. How is implicit type conversion different from explicit type conversion ?
5. Write conversion function you would use for following type of conversions.
   (i) Boolean to string
   (ii) integer to float
   (iii) float to integer
   (iv) string to integer
   (v) string to float
   (vi) string to Boolean
   (vii) integer to complex

3.5.2 Type Casting

You have learnt in earlier section that in expression with mixed types, Python internally changes the data type of some operands so that all operands have same data type. This type of conversion is automatic, *i.e.*, implicit and hence known as implicit type conversion. Python, however, also supports explicit type conversion.

Explicit Type Conversion

An explicit type conversion is user-defined conversion that forces an expression to be of specific type. The explicit type conversion is also known as **Type Casting**.

Type casting in Python is performed by <type>( ) function of appropriate data type, in the following manner :

```
<datatype> (expression)
```

where <datatype> is the data type to which you want to type-cast your expression.

For example, if we have ($a = 3$ and $b = 5.0$), then

```
int(b)
```

will cast the data-type of the expression as **int**.

TYPE CASTING

The explicit conversion of an operand to a specific type is called type casting.

Similarly,

```
d = float(a)
```

will assign value 3.0 to *d* because *float(a)* cast the expression's value to *float* type and then assigned it to *d*.

Python offers some conversion functions that you can use to type cast a value in Python. These are being listed in following Table 3.10.

Table 3.10 *Python Data Conversion Functions*

S. No.	Conversion		Conversion Function	Examples
	From	To		
1.	any number-convertible type *e.g.*, a float, a string having digits	integer	int( )	int(7.8) will give 7 (floating point number to integer conversion) int('34')[10] will give 34 (string to integer conversion)
2.	any number-convertible type *e.g.*, a float, a string having digits	floating point number	float( )	float(7) will give 7.0 (integer to floating point number conversion) float('34') will give 34.0 (string to floating point number conversion)
3.	numbers	complex number	complex( )	complex(7) will give $7+0j$ (ONE ARGUMENT- integer to complex number conversion) complex(3, 2) will give $3+2j$ (TWO ARGUMENTS – integer to complex number conversion)

10. If a number (in string form) is given in any other base *e.g.*, octal or hexadecimal or binary, it can also be converted to integers using **int( )** as int (<number-in-string-form>, base). *For example*, to convert a string **'0o11'** (octal bases equivalent of) into integer, you can write **int('0o10', 8)** and it will give 8.
If you want to convert an integer to octal or hexadecimal or binary form then oct( ), hex( ) or bin( ) functions respectively are there but they produce the equivalent number in string they produce the equivalent number in string form *i.e.*, **hex(10)** will give you '0x A'. This value can be displayed or printed but cannot be used in calculations as it is not number. However, by combining hex( ), oct( ), bin( ) with int (<number string>, base) you can convert to appropriate type.

S. No.	Conversion From	Conversion To	Conversion Function	Examples
4.	number Booleans	string	str( )	**str(3)** will give '3' (integer to string conversion) **str(5.78)** will give '5.78' (floating-point number to string conversion) **str(0o17)** will give '15' (octal number to string conversion ; string converts the equivalent decimal number to string : 0o17 = 15) **str(1 + 2j)** will give '(1+2j)' (complex number to string conversion) **str(True)** will give 'True' (Boolean to string conversion)
5.	any type	Boolean	bool( )	**bool(0)** will give **False** ; **bool(0.0)** will give **False** **bool(1)** will give **True** ; **bool(3)** will give **True** **bool('')** will give **False** ; **bool('a')** will give **True** **bool('hello')** will give **True** With **bool( )**, non-zero, non-empty values of any type will give **True** and rest (zero, empty values) will give **False**.

Type Casting Issues

Assigning a value to a type with a greater range (*e.g.*, from *short* to *long*) poses no problem, however, assigning a value of larger data type to a smaller data type (*e.g.*, from *floating-point* to *integer*) may result in losing some precision.

Floating-point type to integer type conversion results in loss of fractional part. Original value may be out of range for target type, in which case result is undefined.

With this, we have come to end of our chapter. Let us quickly recap what we have learnt so far.

PriP EXPRESSION EVALUATION — Progress In Python 3.3

This is an important practical session to reinforce the concepts of expression evaluation in Python. Proper practice of this PriP Session would lay a strong foundation, which would help you solve application based questions like outputs, errors etc. This practical session would involve both interactive mode and script mode.

⋮

Question 1 : (*Solved for your reference*)

Evaluate the following expression (stepwise) on paper first. Then execute it in Python shell. Compare your result with Python's. State reasons

```
len(str(10 < 20))
```

⋮

Please check the practical component-book – Progress in Computer Science with Python and fill it there in PriP 3.3 under Chapter 3 after practically doing it on the computer.

>>>❖<<<

Working With math Module of Python

Other than built-in functions, Python makes available many more functions through modules in its standard library. Python's standard library is a collection of many modules for different functionalities, e.g., module *time* offers time related functions ; module *string* offers functions for string manipulation and so on.

Python's standard library provides a module namely **math** for math related functions that work with all number types except for complex numbers.

In order to work with functions of **math** module, you need to first *import* it to your program by giving statement as follows as the top line of your Python script :

```
import math
```

Then you can use **math** library's functions as math.<function-name> . *Conventionally (not a syntactical requirement), you should give import statements at the top of the program code.*

Following table (Table 3.11) lists some useful math functions that you can use in your programs.

Table 3.11 *Some Mathematical Functions in math Module*

S. No.	Function	Prototype (General Form)	Description	Example
1.	ceil	math.ceil(num)	The **ceil( )** function returns the smallest integer not less than *num*.	math.ceil(1.03) gives 2.0 math.ceil(–1.03) gives –1.0.
2.	sqrt	math.sqrt (num)	The **sqrt( )** function returns the square root of *num*. If num <0, domain error occurs.	math.sqrt(81.0) gives 9.0.
3.	exp	math.exp(arg)	The **exp( )** function returns the natural logarithm **e** raised to the *arg* power.	math.exp(2.0) gives the value of e^2.
4.	fabs	math.fabs (num)	The **fabs( )** function returns the absolute value of *num*.	math.fabs(1.0) gives 1.0 math.fabs(–1.0) gives 1.0.
5.	floor	math.floor (num)	The **floor( )** function returns the largest integer not greater than *num*.	math.floor(1.03) gives 1.0 math.floor(–1.03) gives –2.0.
6.	log	math.log (num, [base])	The **log( )** function returns the natural logarithm for *num*. A domain error occurs if *num* is negative and a range error occurs if the argument *num* is zero.	math.log(1.0) gives the natural logarithm for 1.0. math.log(1024, 2) will give logarithm of 1024 to the base 2.
7.	log10	math.log10 (num)	The **log10( )** function returns the base 10 logarithm for *num*. A domain error occurs if *num* is negative and a range error occurs if the argument is zero.	math.log10(1.0) gives base 10 logarithm for 1.0.
8.	pow	math.pow (base, exp)	The **pow( )** function returns *base* raised to *exp* power *i.e,. base exp*. A domain error occurs if *base* = 0 and *exp* <= 0 ; also if *base* < 0 and *exp* is not integer.	math.pow(3.0, 0) gives value of 3^0. math.pow(4.0, 2.0) gives value of 4^2.

S. No.	Function	Prototype (General Form)	Description	Example
9.	sin	math.sin(arg)	The sin() function returns the sine of *arg*. The value of *arg* must be in radians.	math.sin(val) (*val* is a number).
10.	cos	math.cos(arg)	The cos( ) function returns the cosine of *arg*. The value of *arg* must be in radians.	math.cos(val) (*val* is a number).
11.	tan	math.tan(arg)	The tan( ) function returns the tangent of *arg*. The value of *arg* must be in radians.	math.tan(val) (val is a number)
12.	degrees	math.degrees(x)	The degrees( ) converts angle *x* from radians to degrees.	math.degrees(3.14) would give 179.91
13.	radians	math.radians(x)	The radians( ) converts angle *x* from degrees to radians.	math.radians(179.91) would give 3.14

The **math** module of Python also makes available two useful constants namely **pi** and **e**, which you can use as :

math.pi gives the mathematical constant $\pi = 3.141592...$, to available precision.

math.e gives the mathematical constant $e = 2.718281...$, to available precision.

Following are examples of **valid** anthmetic expressions (after **import math** statement) :

Given : $a = 3$, $b = 4$, $c = 5$, $p = 7.0$, $q = 9.3$, $r = 10.51$, $x = 25.519$, $y = 10\text{-}24.113$, $z = 231.05$

(*i*) math.pow(*a* / *b*, 3.5) (*ii*) math.sin(*p* / *q*) + math.cos(*a* − *c*)

(*iii*) *x* / *y* * math.floor(*p* * *a* / *b*) (*iv*) (math.sqrt (*b*) * *a*) − *c*

(*v*) (math.ceil (*p*) * *a*) * *c*

Following are examples of **invalid** arithmetic expressions :

(*i*) *x* + * *r* — two operators in continuation.

(*ii*) *q*(*a* + *b* − *z* / 4) — operator missing between *q* and *a*.

(*iii*) math.pow(0, −1) — Domain error because if base = 0 then exp should not be <= 0.

(*iv*) math.log(−3) * *p* / *q* — Domain error because logarithm of a negative number is not possible.

Example. *Write the corresponding Python expressions for the following mathematical expressions :*

(*i*) $\sqrt{a^2 + b^2 + c^2}$ (*ii*) $2 - ye^{2y} + 4y$ (*iii*) $p + \frac{q}{(r+s)^4}$

(*iv*) $(\cos x / \tan x) + x$ (*v*) $\left| e^2 - x \right|$

Solution.

(*i*) math.sqrt (*a* * *a* + *b* * *b* + *c* * *c*)

(*ii*) 2 − *y* * math.exp(2 * *y*) + 4 * *y*

(*iii*) *p* + *q* / math.pow((*r* + *s*), 4)

(*iv*) (math.cos (*x*) / math.tan (*x*)) + *x*

(*v*) math.fabs (math.exp(2) − *x*)

LET US REVISE

- *An expression is composed of one or more operations. It is a valid combination of operators, literals and variables.*
- *In Python terms, an expression is a legal combination of atoms and operators.*
- *An atom in Python is something that has a value. Examples of atoms are variables, literals, strings, lists, tuples, sets etc.*
- *Expressions can be arithmetic, relational or logical, compound etc.*
- *Types of operators used in an expression determine its type. For instance, use of arithmetic operators makes it arithmetic expression.*
- *Arithmetic expressions can either be integer expressions or real expressions or complex number operations or mixed-mode expressions.*
- *An arithmetic expression always results in a number (integer or floating-point number or a complex number) ; a relational expression always results in a Boolean value i.e., either True or False ; and a logical expression results into a number or a string or a Boolean value, depending upon its operands.*
- *In a mixed-mode expression, different types of variables/constants are converted to one same type. This process is called type conversion.*
- *Type conversion can take place in two forms : implicit (that is performed by compiler without programmer's intervention) and explicit (that is defined by the user).*
- *In implicit conversion, all operands are converted up to the type of the largest operand, which is called type promotion or coercion.*
- *The explicit conversion of an operand to a specific type is called type casting and it is done using type conversion functions that is used as*

```
<type conversion function > ( <expression>)
```

e.g., to convert to float, one may write

```
float(<expression>)
```

Solved Problems

1. *What are data types ? What are Python's built-in core data types ?*

Solution. The real life data is of many types. So to represent various types of real-life data, programming languages provide ways and facilities to handle these, which are known as *data types*.

Python's built-in core data types belong to :

- Numbers (integer, floating-point, complex numbers, Booleans)
- String
- List
- Tuple
- Dictionary

2. *Which data types of Python handle Numbers ?*

Solution. Python provides following data types to handle numbers

(*i*) Integers (*ii*) Boolean

(*iii*) Floating-point numbers (*iv*) Complex numbers

3. *Why is Boolean considered a subtype of integers ?*

Solution. Boolean values *True* and *False* internally map to integers *1* and *0*. That is, internally *True* is considered equal to 1 and *False* equal to 0 (zero). When 1 and 0 are converted to Boolean through bool( ) function, they return *True* and *False*. That is why Booleans are treated as a subtype of integers.

4. *Identify the data types of the values given below :*

3, 3j, 13.0, '13', "13", 2 + 0*j*, 13, [3, 13, 2], (3, 13, 2)

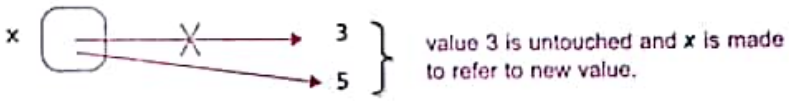
Solution.

3	integer	3*j*	complex number
13.0	Floating-point number	'13'	string
"13"	String	2+0*j*	complex number
13	integer	[3, 13, 2]	List
(3, 13, 2)	Tuple		

5. *What do you understand by term 'immutable' ?*

Solution. Immutable means unchangeable. In Python, immutable types are those whose values cannot be changed in place. Whenever one assigns a new value to a variable referring to immutable type, variable's reference is changed and the previous value is left unchanged. *e.g.,*

```
x = 3
x = 5
```



6. *What will be the output of the following ?*

```
print (len(str(17//4)) )
print (len(str(17/4)) )
```

Solution. 1
3

because

```
len(str(17//4))
   = len(str(4))
   = len('4')
   = 1
```

and

```
len(str(17/4))
= len(str(4.0))
= len('4.0')
= 3
```

7. *What will be the output produced by these ?*

(*a*) 12/4 (*b*) 14//14 (*c*) 14%4 (*d*) 14.0/4 (*e*) 14.0//4 (*f*) 14.0%4

Solution. (*a*) 3.0 (*b*) 1 (*c*) 2 (*d*) 3.5 (*e*) 3.0 (*f*) 2.0

8. *Given that variable CK is bound to string "Raman" (i.e., CK = "Raman"). What will be the output produced by following two statements if the input given in "Raman" ? Why ?*

```
DK = input("Enter name:")
Enter name : Raman
```

(*a*) DK == CK (*b*) DK is CK

Solution. The output produced will be as :

(*a*) True (*b*) False

The reason being that both DK and CK variable are bound to identical strings 'Raman'. But input strings are always bound to fresh memory even if they have value identical to some other existing string in memory.

Thus DK == CK produces True as strings are identical.

But DK is CK produces False as they are bound to different memory addresses.

9. *What will be the output of following code ? Explain reason behind output of every line :*

```
5 < 5 or 10
5 < 10 or 5
5 < (10 or 5)
5 < (5 or 10)
```

Solution.

```
10
True
True
False
```

Explanation

Line 1	`5 < 5 or 10`
	`= False or 10`
	`= 10`
		precedence of < is higher than or
		
		because or would evaluate the second argument if first argument is False or $false_{real}$

Line 2	`5 < 10 or 5`
	`= True or 5`
	`= True`
		precedence of < is higher than or
		
		or would return first argument if it is True or $true_{real}$

Line 3	`5 < (10 or 5)`
	`= 5 < 10`
	`= True`
		10 or 5 returns 10 since 10 is $true_{real}$

Line 4	`5 < (5 or 10)`
	`= 5 < 5`
	`= False`
		5 or 10 returns 5 since 5 is $true_{real}$

10. *What will be output produced by the three expressions of the following code ?*

```
a = 5
b = -3
c = 25
d = -10
a + b + c > a + c - b * d
str(a + b + c > a + c - b * d) == 'true'
len(str(a + b + c > a + c - b * d)) == len(str(bool(1)))
```

Solution.
```
True
False
True
```

11. *What would Python produce if for the following code, the input given is*
(*i*) 11 (*ii*) hello (*iii*) just return key pressed, no input given
(*iv*) 0 (*v*) 5-5

Code
```
bool(input("Input:")) and 10 < 13 - 2
```

Solution.

(*i*) Input : 11 would yield
```
bool ('11') and 10 < 11
True and 10 < 11
= True
```

(*ii*) **True** [For the same reason as in (*i*)]

(*iii*) **False** because when just return key is pressed, input is '' *i.e.*, empty string, hence expression becomes
```
bool( '' ) and 10 < 11
False and True
= False
```

(*iv*)
```
bool ( '0' ) and 10 < 11
True and True
= True
```
(Please note '0' is a non-empty string and hence has truth value as true $_{true}$)

(*v*)
```
bool ( '5-5' ) and 10 < 11
= True and 10 < 11
= True
```

12. *What would be the output of the following code ? Explain reason(s).*
```
a = 3 + 5/8
b = int(3 + 5/8)
c = 3 + float(5/8)
d = 3 + float(5)/8
e = 3 + 5.0/8
f = int(3 + 5/8.0)
print (a, b, c, d, e, f)
```

Solution. The output would be
```
3.625  3  3.625  3.625  3.625  3
```

Explanation

Line 1
```
a = 3 + 5/8
  = 3 + 0.625
```
$\therefore$ $a = 3.625$

Line 2	`b = int (3 + 5/8)` = int (3 + 0.625) = int (3.625) *b* = 3	(int( ) will drop the fractional part)
Line 3	`c = 3 + float(5/8)` *c* = 3 + float(0.625) = 3 + 0.625 *c* = 3.625	
Line 4	`d = 3 + float(5)/8` = 3 + 5.0/8 = 3 + 0.625 *d* = 3.625	5.0/8 = 0.625 because one operand is floating point, the integer operand will be internally converted to floating-pt
Line 5	`e = 3 + 5.0/8` = 3 + 0.625 *e* = 3.625	(same reason as above)
Line 6	`f = int(3 + 5/8.0)` *f* = int (3 + 0.625) *f* = int (3.0) *f* = 3	(same reason as above) (int( ) will drop the fractional part)

13. *What will be the output produced by following code statements ?*

(*a*) 87 // 5 (*b*) 87 // 5.0

(*c*) (87 // 5.0) == (87 // 5) (*d*) (87 // 5.0) == int (87 / 5.0)

(*e*) (87 // int (5.0)) == (87 // 5.0)

Solution. (*a*) 17 (*b*) 17.0 (*c*) True (*d*) True (*e*) True

14. *What will be the output produced by following code statement ? State reasons(s).*

(*a*) 17 % 5 (*b*) 17 % 5.0

(*c*) (17 % 5) == (17 % 5) (*d*) (17 % 5) is (17 % 5)

(*e*) (17 % 5.0) == (17 % 5.0) (*f*) (17 % 5.0) is (17 % 5.0)

Solution. (*a*) 2 (*b*) 2.0 (*c*) True (*d*) True (*e*) True (*f*) False

Both (*c*) and (*e*) evaluate to **True** because both the operands of == operator are same values (2 == 2 in (*c*) and 2.0 == 2.0 in (*e*).

(*d*) evaluates to **True** as both the operands of **is** operator are same integer objects (2).

Since both operand expressions evaluate to same integer value 2 and 2 is a small integer value, both are bound to same memory address, hence **is** operator returns **True**.

In (*f*), even though both operands evaluate to same floating point value 2.0, the **is** operator returns **False** because Python assigns different memory address to floating point values even if their exists a same value in the memory.

15. *What will be the output produced by the following code statements ? State reasons.*

(*a*) bool(0) (*b*) bool (1) (*c*) bool('0') (*d*) bool('1') (*e*) bool(")
(*f*) bool(0.0) (*g*) bool('0.0') (*h*) bool(0j) (*i*) bool('0j')

Solution.

(*a*) **False.** Integer value 0 has false truth value hence bool( ) converts it to **False.**

(*b*) **True.** Integer value 1 has true truth value hence bool( ) converts it to **True.**

(*c*) **True.** '0' is string value, which is a non-empty string and has a true truth value, hence bool( ) converts it to *True.*

(*d*) **True.** Same reason as above.

(*e*) **False.** " is an empty string, thus has false truth value, hence bool( ) converts it to *False.*

(*f*) **False.** 0.0 is zero floating point number and has false truth value, hence bool( ) converts it to *False.*

(*g*) **True.** '0.0' is a non-empty string hence it has true truth value and thus bool( ) converted it to *True.*

(*h*) **False.** 0j is zero complex number and has false truth value and thus bool( ) converted it to *False.*

(*i*) **True.** '0j' is non-empty string, hence it has true truth value and thus boo( ) converted it to *True.*

16. *What will be the output produced by these code statement ?*

(*a*) bool(int('0'))
(*b*) bool(str(0))
(*c*) bool(float('0.0'))
(*d*) bool(str(0.0))

Solution. (*a*) False (*b*) True (*c*) False (*d*) True

17. *What will be the output of following code ? Why ?*

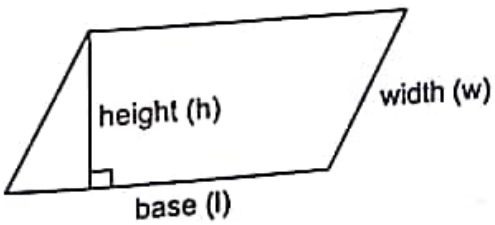
(*i*) `13 or len(13)`
(*ii*) `len(13) or 13`

Solution.

(*i*) **13** because or evaluates first argument 13's truth value, which is *true* $_{bool}$ and hence returns the result as 13 without evaluating second argument.

(*ii*) error because when or evaluates fist argument **len(13)**, Python gives error as **len( )** works on strings only.

18. *Write a program to read base/length (l), width(w) and height(h) of a parallelogram and calculate its area and perimeter.*



Solution.

```
l = float(input("Enter base/length of the parallelogram : "))
w = float(input("Enter width of the parallelogram : "))
h = float(input("Enter height of the parallelogram : "))
area = l * h
perimeter = 2*l + 2*w
print("The area of given parallelogram is :", area)
print("The perimeter of given parallelogram is :", perimeter)
```

The sample run of above program is as shown below :

```
Enter base/length of the parallelogram : 13.5
Enter width of the parallelogram : 7
Enter height of the parallelogram : 5
The area of given parallelogram is : 67.5
The perimeter of given parallelogram is : 41.0
```

GLOSSARY

Atom	Something that has a value.
Coercion	Implicit Type Conversion
Expression	Valid combination of operators and atoms.
Explicit Type Conversion	Forced data type conversion by the user.
Immutable Type	A type whose value is not changeable in place.
Implicit Type Conversion	Automatic Internal Conversion of data type (lower to higher type) by Python
Mutable Type	A type whose value is changeable in place.
Operator	Symbol/word that triggers an actron or operation.
Type Casting	Explicit Type Conversion

Assignments

Type A : Short Answer Questions/Conceptual Questions

1. What are data types ? How are they important ?
2. How many integer types are supported by Python ? Name them.
3. How are these numbers different from one another ? 33, 33.0, 33j, 33 + j
4. The complex numbers have two parts : real and imaginary. In which data type are real and imaginary parts represented ?
5. How many string types does Python support ? How are they different from one another ?
6. What will following code print ?

```
str1 = '''Hell ↵
          o'''
str2 = '''Hell\
          o'''
print(len(str1) > len(str2))
```

7. What are immutable an mutable types ? List immutable and mutable types of Python.
8. What are three internal key-attributes of a value-variable in Python ? Explain with example.
9. Is it true that if two objects return True for is operator, they will also return True for == operator ?
10. Are these values equal ? Why/why not ?
    (*i*) 20 and 20.0 (*ii*) 20 and int(20)
    (*iii*) str(20) and str(20.0) (*iv*) 'a' and "a"
11. What is an atom ? What is an expression ?
12. What is the difference between implicit type conversion and explicit type conversion ?
13. Two objects (say *a* and *b*) when compared using == , return True. But Python gives False when compared using is operator. Why ?
    (*i.e.*, *a* == b is True but why is *a is b* False ?)
14. Given str1 = "Hello", what will be the values of ?
    (*a*) str1 [0] (*b*) str1 [1] (*c*) str [–5] (*d*) str [–4] (*e*) str [5]
15. If you give the following for str1 = "Hello", why does Python report error ?
    `str1[2] = 'p'`
16. What will the result given by the following ?
    (*a*) type (6 + 3) (*b*) type (6 – 3) (*c*) type (6 * 3) (*d*) type (6/3) (*e*) type (6//3) (*f*) type (6 % 3)
17. What are augmented assignment operators ? How are they useful ?
18. Differentiate between (555/222)**2 and (555.0/222)**2.
19. Given three Boolean variables *a*, *b*, *c* as : *a* = False, *b* = True, *c* = False.
    Evaluate the following Boolean expressions :
    (*a*) *b* and *c* (*b*) *b* or *c* (*c*) not *a* and *b* (*d*) (*a* and *b*) or not *c*
    (*e*) not *b* and not (*a* or *c*) (*f*) not ((not *b* or not *a*) and *c*) or *a*
20. What would following code fragments result in ? Given $x = 3$.
    (*a*) $1 < x$ (*b*) $x >= 4$ (*c*) $x == 3$ (*d*) $x == 3.0$ (*e*) "Hello" == "Hello"
    (*f*) "Hello" > "hello" (*g*) $4/2 == 2.0$ (*h*) $4/2 == 2$ (*i*) $x < 7$ and $4 > 5$.
21. Write following expressions in Python
    (*a*) $\frac{1}{3}b^2h$ (*b*) $\pi r^2 h$ (*c*) $\frac{1}{3}\pi r^2 h$ (*d*) $d = \sqrt{(x2-x1)^2+(y2-y1)^2}$ [Hint. for sqrt( ) use math.sqrt( )]
    (*e*) $(x-h)^2+(y-k)^2 = r^2$ (*f*) $x = \frac{-b+\sqrt{b^2-4ac}}{2a}$ (*g*) $a^n \times a^m = a^{n+m}$
    (*h*) $(a^n)^m = a^{nm}$ (*i*) $\frac{a^n}{a^m} = a^{n-m}$ (*j*) $a^{-n} = \frac{1}{a^n}$
22. int('*a*') produces error. Why ?
23. int('*a*') produces error but following expression having int('*a*') in it, does not return error. Why ?
    len('*a*') + 2 or int('*a*')
24. Write expression to convert the values 17, len(ab) to
    (*i*) integer (*ii*) str (*iii*) Boolean values
25. Evaluate and Justify : (*i*) 22 / 17 = 37 / 47 + 88 / 83 (*ii*) len('375')**2.
26. Evaluate : (*i*) 22.0/7.0 – 22/7 (*ii*) 22.0/7.0 – int(22.0/7.0) (*iii*) 22/7 – int (22.0)/7 and justify.
27. Evaluate and justify : (*i*) false and None (*ii*) 0 and None (*iii*) True and None (*iv*) None and None.

28. Evaluate and justify :
    (a) `0 or None and "or"`
    (b) `1 or None and 'a' or 'b'`
    (c) `False and 23`
    (d) `23 and False`
    (e) `not (1 == 1 and 0 != 1)`
    (f) `"abc" == "Abc" and not (2 == 3 or 3 == 4)`
    (g) `False and 1 == 1 or not True or 1 == 1 and False or 0 == 0`
29. Evaluate the following for each expression that is successfully evaluated, determine its value and type for unsuccessful expression, state the reason.
    (a) `len("hello") == 25/5 or 20/10`
    (b) `3 < 5 or 50/(5 - (3 + 2))`
    (c) `50/(5 - (3 + 2)) or 3 < 5`
    (d) `2 * (2 * (len("01")))`
30. Write an expression that uses exactly 3 arithmetic operators with integer literals and produces result as 99.
31. Add parentheses to the following expression to make the order of evaluation more clear.
    `y % 4 == 0 and y % 100 != 0 or y % 400 == 0`

Type B : Application Based Questions

1. What is the result produced by (*i*) bool (0) (*ii*) bool (str(0)) ? Justify the outcome.
2. What will be the output, if input for both the statements is 5 + 4 / 2.

```
6 == input ("Value 1:")
6 == int(input ("value 2:"))
```

3. Following code has an expression with all integer values. Why is the result in floating point form ?

```
a, b, c = 2, 3, 6
d = a + b * c/b
print(d)
```

4. What will following code print ?

(a)
```
a = va = 3
b = va = 3
print (a, b)
```

(b)
```
a = 3
b = 3.0
print (a == b)
print (a is b)
```

5. What will be output produced by following code ? State reason for this output.

```
a, b, c = 1, 1, 2
d = a + b
e = 1.0
f = 1.0
g = 2.0
h = e + f
print(c == d)
print(c is d)
print(g == h)
print(g is h)
```

6. What will be output produced by following code ? State reason.

```
a = 5 - 4 - 3
b = 3**2**3
print(a)
print(b)
```

7. What would be the output produced by following code ? Why ?

```
a, b, c = 0.1
d = 0.3
e = a + b + c - d
f = a + b + c == d
print(e)
print(f)
```

8. What will be the output of following Python code ?

```
a = 12
b = 7.4
c = 1
a - = b
print(a, b)
a *= 2 + c
print(a)
b += a * c
print(b)
```

9. What will be the output of following code ?

```
x, y = 4, 8
z = x/y*y
print(z)
```

10. Make change in the expression for z of previous question so that the output produced is zero. You cannot change the operators and order of variables.

(Hint. Use a function around a sub-expression)

11. Following expression does not report an error even if it has a sub-expression with 'divide by zero' problem :

```
3  or  10/0
```

What changes can you make to above expression so that Python reports this error ?

12. What is the output produced by following code ?

```
a, b = bool (0), bool (0.0)
c, d = str (0), str (0.0)
print (len(a), len(b))
print (len(c), len(d))
```

13. Given a string s = "12345". Can you write an expression that gives sum of all the digits shown inside the string s *i.e.*, the program should be able to produce the result as 15 (1+2+3+4+5).

[Hint. Uses indexes and convert to integer]

14. Predict the output if e is given input as 'True'

```
a = True
b = 0 < 5
print (a == b)
print (a is b)
```

```
c = str (a)
d = str (b)
print (c == d)
print (c is d)
e = input ("Enter :")
print (c == e)
print (c is e)
```

15. Find the errors(s).

(a)
```
name = "HariT"
print (name)
name[2] = 'R'
print (name)
```

(b)
```
a = bool (0)
b = bool (1)
print (a == false)
print (b == true)
```

(c)
```
print (type (int("123")))
print (type (int("Hello")))
print (type (str("123.0")))
```

(d)
```
pi = 3.14
print (type (pi))
print (type ("3.14"))
print (type (float ("3.14")))
print (type (float("three point fourteen")))
```

(e)
```
print ("Hello" + 2)
print ("Hello" + "2")
print ("Hello" * 2)
```

(f)
```
print ("Hello"/2)
print ("Hello" / 2)
```

Type C : Programming Practice/Knowledge based Questions

1. Write a program to obtain principal amount, rate of interest and time from user and compute simple interest.
2. Write a program to obtain temperatures of 7 days (Monday, Tuesday ... Sunday) and then display average temperature of the week.
3. Write a program to obtain x, y, z from user and calculate expression : $4x^4 + 3y^3 + 9z + 6\pi$.
4. Write a program that reads a number of seconds and prints it in form : mins and seconds, *e.g.*, 200 seconds are printed as 3 mins and 20 seconds.
   [Hint. use // and % to get minutes and seconds]
5. Write a program that reads from user – (*i*) an hour between 1 to 12 and (*ii*) number of hours ahead. The program should then print the time after those many hours, *e.g.*,
   ```
   Enter hour between 1-12 : 9
   How many hours ahead : 4
   Time at that time would be : 1 0clock
   ```
6. Write a program to take year as input and check if it is a leap year or not.
7. Write a program to take two numbers and print if the first number is fully divisible by second number or not.
8. Write a program to take a 2-digit number and then print the reversed number. That is, if the input given is 25, the program should print 52.
9. Try writing program (similar to previous one) for three digit number *i.e.*, if you input 123, the program should print 321.
10. Write a program to take two inputs for day, month and then calculate which day of the year, the given date is. For simplicity, take 30 days for all months. For example, if you give input as : Day3, Month2 then it should print "Day of the year : 33".

4

Conditional and Iterative Statements

In This Chapter

4.1 INTRODUCTION

Generally a program executes its statements from beginning to end. But not many programs execute all their statements in strict order from beginning to end. Programs, depending upon the need, can choose to execute one of the available alternatives or even repeat a set of statements. To perform their manipulative miracles, programs need tools for performing repetitive actions and for making decisions. Python, of course, provides such tools by providing statements to attain so. Such statements are called *program control statements*. This chapter discusses such statements in details. Firstly, selection statement *if* and later iteration statements *for* and *while* are discussed.

This chapter also discusses some jump statements of Python which are *break* and *continue*.

4.2 TYPES OF STATEMENTS IN PYTHON

Statements are the instructions given to the computer to perform any kind of action, be it *data movements*, and be it *making decisions* or be it *repeating actions*. Statements form the smallest executable unit within a Python program. Python statements can belong to one of the following *three* types :

- Empty Statement
- Compound Statement
- Simple Statement (Single statement)

1. Empty Statement

The simplest statement is the *empty* statement *i.e.*, a statement which does nothing. In Python an empty statement is pass statement. It takes the following form :

```
pass
```

Wherever Python encounters a pass statement, Python does nothing and moves to next statement in the flow of control.

A pass statement is useful in those instances where the syntax of the language requires the presence of a statement but where the logic of the program does not. We will see it in loops and their bodies.

NOTE

The pass statement of Python is a *do nothing* statement *i.e.*, empty statement or null operation statement.

2. Simple Statement

Any single executable statement is a simple statement in Python. For example, following is a simple statement in Python :

```
name = input ( "Your name" )
```

Another example of simple statement is :

```
print (name)          # print function called
```

As you can make out that simple statements are single line statements.

3. Compound Statement

A compound statement represents a group of statements executed as a unit. The compound statements of Python are written in a specific pattern as shown below :

```
<compound statement header> :
    <indented body containing multiple simple and/or compound statements>
```

That is, a compound statement has :

- a **header line** which begins with a keyword and ends with a colon.
- a **body** consisting of one or more Python statements, each indented[1] inside the header line. All statements in the body are at the same level of indentation.

You'll learn about some other compound statements (*if, for, while*) in this chapter.

NOTE

A *compound statement* in Python has a *header* ending with a colon (:) and a *body* containing a sequence of statements at the same level of indentation.

1. Conventionally, Python uses four spaces to move to next level of indentation.

4.3 STATEMENT FLOW CONTROL

In a program, statements may be executed sequentially, selectively or iteratively. Every programming language provides constructs to support sequence, selection or iteration.

Let us discuss what is meant by sequence, selection or iteration constructs.

Sequence

The sequence construct means the statements are being executed sequentially. This represents the default flow of statement (see Fig. 4.1).

Every Python program begins with the first statement of program. Each statement in turn is executed (sequence construct). When the final statement of program is executed, the program is done. Sequence refers to the normal flow of control in a program and is the simplest one.

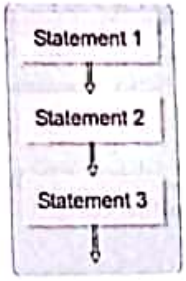


Figure 4.1 The sequence construct.

Selection

The selection construct means the execution of statement(s) depending upon a condition-test. If a condition evaluates to *True*, a course-of-action (a set of statements) is followed otherwise another course-of-action (a different set of statements) if followed. This construct (*selection construct*) is also called *decision construct* because it helps in making decision about which set-of-statements is to be executed.

Following figure (Fig. 4.2) explains selection construct.

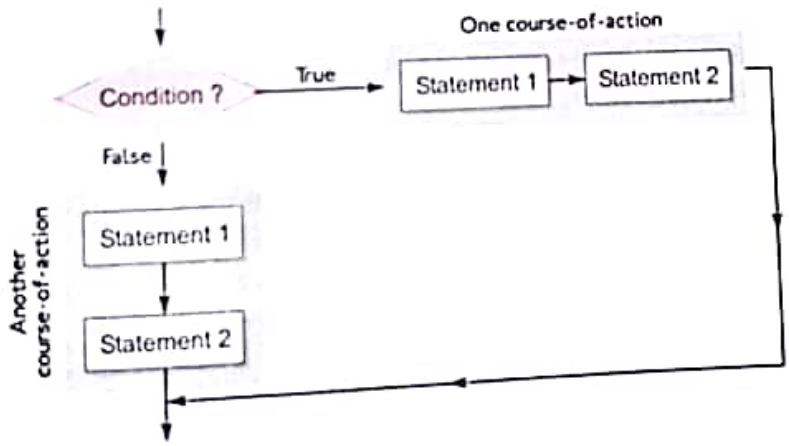


Figure 4.2 The selection construct.

You apply decision-making or selection in your real life so many times *e.g.*, if the traffic signal light is **red**, then **stop**; if the traffic signal light is **yellow** then **wait** ; and if the signal light is **green** then **go**.

You can think of many such real life examples of selection/decision-making.

Iteration (Looping)

The iteration constructs mean *repetition of a set-of-statements* depending upon a *condition-test*. Till the time a condition is *True* (or *False* depending upon the loop), a set-of-statements are repeated again and again. As soon as the condition becomes *False* (or *True*), the repetition stops. The iteration construct is also called *looping construct*.

The adjacent figure (Fig. 4.3) illustrates an iteration construct.

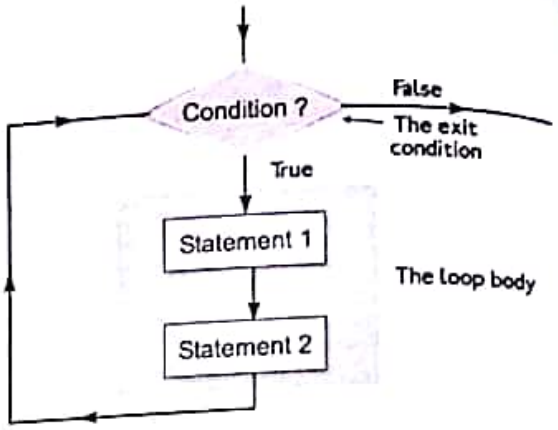


Figure 4.3 The iteration construct.

The set-of-statements that are repeated again and again is called the *body of the loop*. The condition on which the execution or exit of the loop depends is called *the exit condition* or *test-condition*.

You can find many examples of iteration or looping around you. For instance, you often see your mother cook *chapatis* or *dosas* or *appams* for you.

Let's see, what she does for it :

(*i*) put rolled chapati or dosa batter on flat pan or tawa

(*ii*) turn it 2-3 times

(*iii*) once done take it off.

Repeat the above process (steps (*i*), (*ii*), (*iii*)) for next chapati/dosa/appam. This is looping or iteration.

You can find numerous other examples of repetitive work in your real life *e.g.*, washing clothes; colouring in colouring book etc. etc.

Every programming language must support these three types of constructs as the sequential program execution (the default mode) is inadequate to the problems we must solve. Python also provides statements that support these constructs. Coming sections discuss these Python statements – *if* that supports selection and *for* and *while* that support iteration. Using these statements you can create programs as per your need.

But before we talk about these statements, you should know basic tools that will help you decide about the logic to solve a given problem *i.e.*, the algorithm. For this, there are multiple tools available. In the following section, we are going to talk about three such tools – *Flowcharts, decision trees and pseudocode.*

Check Point 4.1

1. What is a statement ? How many types of statements are there in Python ?
2. What is the significance of a pass statement?
3. What is a compound statement ? Give example of a compound statement.
4. What are the three constructs that govern statement flow ?
5. What is the need for selection and looping constructs ?

4.4 PROGRAM LOGIC DEVELOPMENT TOOLS

Before developing the solution of a problem in terms of a program, you should read and analyze the given problem and decide about basic sub-tasks needed to solve a problem and the order of these subtasks. In other words, you figure out the algorithm for the solution of a given problem and represent it through various logic development tools.

An **algorithm** is a step-by-step procedure (well-defined instructions) to solve a given problem. For instance, the algorithm to find the remainder of given two numbers is :

ALGORITHM

An **algorithm** is a step-by-step procedure (well-defined instructions) to solve a given problem.

1. **Input** *first number.*
2. **Input** *second number.*
3. **Divide** *first number* with *second number* and store the **remainder** as *third number.*
4. **Display** the result (the *third number*).

An algorithm is a set of ordered and finite steps (the subtasks) to solve a given problem. Consider another example that extends the above problem – *using the same logic determine if the first number is divisible by second number or not.*

1. **Input** *first number.*
2. **Input** *second number.*
3. **Divide** *first number* with *second number* and store the **remainder** in *third number.*
4. **Check** if the *third number is 0.*
   (*a*) If Yes, Display *'the first number IS divisible by second number'.*
   (*b*) If No, Display *'the first number IS NOT divisible by second number'.*

As you can see that the subtasks in above algorithms are shown in bold letters and there is a proper order of carrying out these steps/subtasks. Algorithms are commonly written out with tools like *pseudocode, flow charts,* or *decision trees* and *tables.* Let us talk about these tools one by one.

4.4.1 Flowcharts

A flowchart is a graphical representation of an algorithm. A flowchart shows different subtasks with different symbols. Following figure (4.4) shows commonly used flowchart symbols.

FLOWCHART

A **flowchart** is a graphical representation of steps an algorithm to solve a given problem.

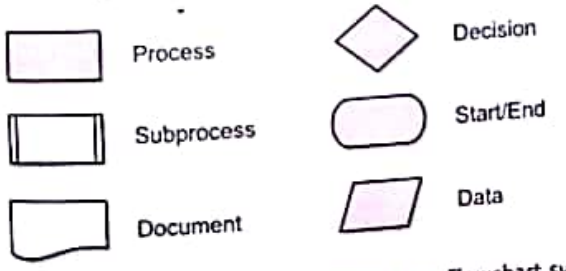
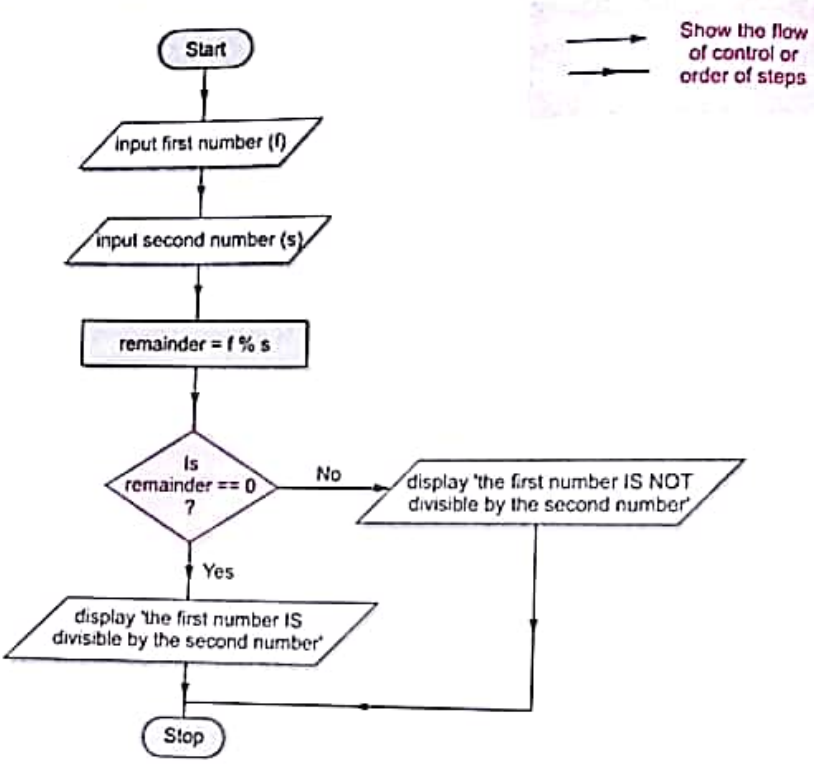


Figure 4.4 Flowchart symbols.

- Use **Data** symbol for Input/Output (I/O) operation (taking input and showing output).
- Use **Process** symbol for any type of computation and internal operations like initialization, calculation etc.
- Use **Subprocess** symbol to invoke a procedure written already.

The flowchart for the above algorithm (*determine if the first number is divisible by second number or not*) will be :



With flowcharts, a programmer can pictorially view the subtasks of an algorithm and the order their execution.

4.4.2 Pseudocode

Pseudocode is an informal language that helps programmers describe steps of a program's solution without using any programming language syntax. Pseudocode is a "text-based" detail (algorithmic) design tool. A pseudocode creates an outline or a rough draft of a program that gives the idea of how the algorithm works and how the control flows from one step to another.

NOTE

Pseudocode is an informal way of describing the steps of a program's solution without using any strict programming language syntax or underlying technology considerations.

For example, consider the following pseudocode :

```
If student's marks is greater than or equal to 50
    display "passed"
else
    display "failed"
```

The above pseudocode gives you an idea how a program determines whether a student has *passed* or *failed* comparing the marks of the student.

Consider the same algorithm that we talked above (*determine if the first number is divisible by second number or not*). The pseudocode of this algorithm can be like :

```
Input first number in variable firstnum.
Input second number in variable secondnum
remainder = firstnum modulus secondnum                    # modulus to get remainder
if the remainder is 0
    display 'the first number IS divisible by second number'.
else
    display 'the first number IS NOT divisible by second number'.
```

Please note there are no standard rules to write a pseudocode as it is totally informal way of representing an algorithm.

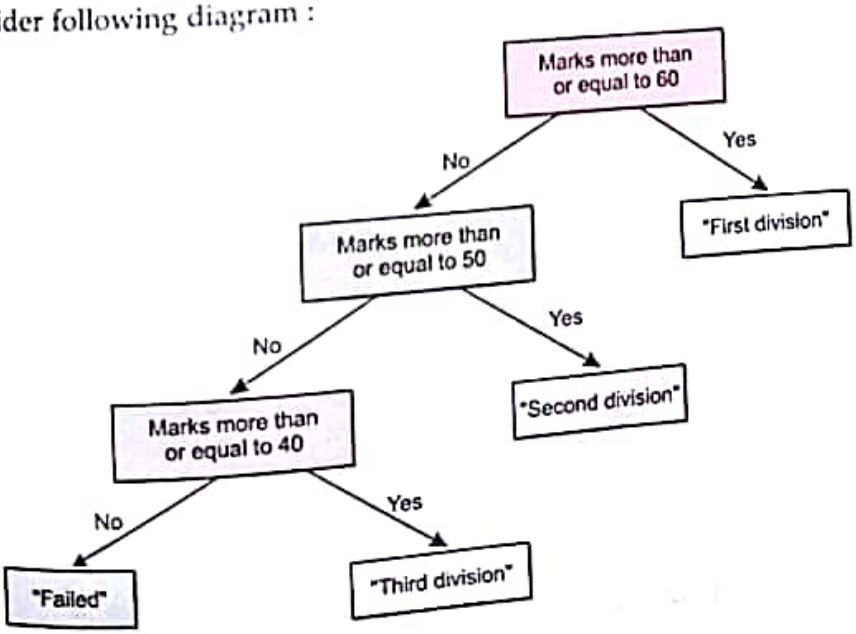
4.4.3 Decision Trees

Decision trees are a way of presenting rules in a hierarchical and sequential structure, where based on the hierarchy of *rules*, certain *outcomes* are predicted.

Let us consider a familiar example. You all know how divisions are determined based on marks, *e.g.*,

Marks	*Division*
≥ 60	First
50 – 60	Second
40 – 50	Third
< 40	Failed

Now consider following diagram :



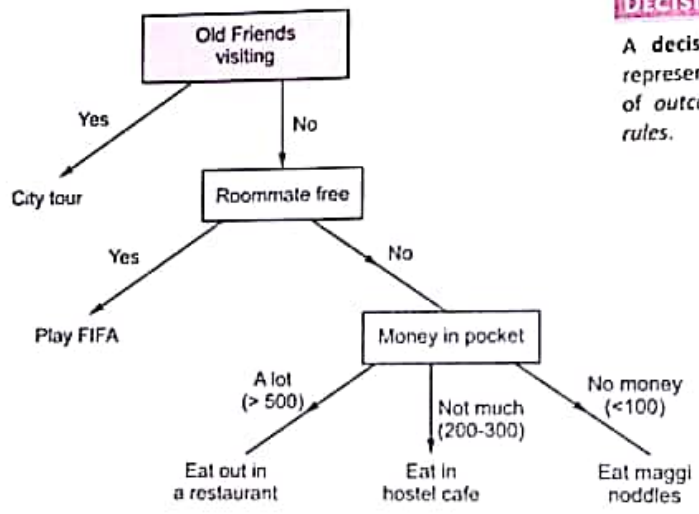
Does it make sense ? Isn't it more readable ?

A Decision tree is an excellent tool to represent a hierarchical structure of outcomes based on certain rules. In other words, a decision uses a tree like structure that can represent a number of possible rules based *decision paths* and an outcome for each path.

Let us imagine another hypothetical situation. Aryan has taken admission in a new school in another city and is staying in the school hostel. This Sunday Aryan's old friends may visit him. So, Aryan has decided to spend his Sunday as per this plan :

> If his friends visit him, he will go with them for *city tour*. If they don't come, he will *play FIFA* with his roommate if his roommate is free, otherwise, depending upon money in his pocket, he may *eat out in a restaurant* or from small *Cafe in hostel premises* or will simply *eat maggi noodles*.

Now this situation can also be depicted through a decision tree :



DECISION TREE

A **decision tree** is a tool to represent a hierarchical structure of *outcomes* based on certain *rules*.

So the rules for spending Sunday as per this decision tree can be written as :

If old friends are visiting, go for **city tour**

Or

If old friends are not visiting and roommate is free then **play FIFA**

Or

If old friends are not visiting and roommate is not free and has a lot of money (say more than ₹ 500) then **eat out in a restaurant**

Or

If old friends are not visiting and roommate is not free and has not much money (roughly ₹ 200-300) then **eat in the hostel cafe**

Or

If old friends are not visiting and roommate is not free and has no money (less than ₹ 100) then **eat maggi noodles.**

Decision trees are majorly used in learning algorithms of modern programming languages.

After learning about these logic development tools, let us learn about Python's selection and iterative statements.

4.5 THE if STATEMENTS OF PYTHON

The if statements are the conditional statements in Python and these implement selection constructs (decision constructs).

An *if* statement tests a particular condition ; if the condition evaluates to *true*, a course-of-action is followed *i.e.*, a statement or set-of-statements is executed. Otherwise (if the condition evaluates to *false*), the course-of-action is ignored.

Before we proceed, it is important that you recall what you learnt in Chapter 3 in relational and logical operators as all that will help you form conditionals in this chapter. Also, please note in all forms of *if* statement, we are using *true* to refer to Boolean value ***True as well as truth value true*** $_{true}$ collectively ; similarly, *false* here will refer to Boolean value ***False and truth value false*** $_{true}$ collectively.

4.5.1 The if Statement

The simplest form of *if* statement tests a condition and if the condition evaluates to *true*, it carries out some instructions and does nothing in case condition evaluates to *false*.

The *if* statement is a compound statement and its syntax (general form) is as shown below :

```
if <conditional expression> :
    statement
    [statements]
```

The statements inside an if are indented at same level.

where a statement may consist of a single statement, a compound statement, or just the ***pass*** statement (in case of empty statement).

Carefully look at the *if statement* ; it is also a *compound statement* having a *header* and a *body* containing indented statements.

For instance, consider the following code fragment :

```
if ch == ' ' :
    spaces += 1
    chars += 1
```

Conditional expression

*The **header** of if statement, notice colon (:) at the end*

Body of if . Notice that all the statements in the if-body are indented at same level

In an ***if statement***, if the *conditional expression* evaluates to *true*, the statements in the *body-of-if* are executed, otherwise ignored.

The above code will check whether the character variable ch stores a space or not ; if it does (*i.e.*, the condition ch==' ' evaluates to *true*) , the number of spaces are incremented by *1* and number of characters (stored in variable chars) are also incremented by value *1*.

If, however, variable ch does not store a space *i.e.*, the condition ch==' ' evaluates to *false*, then nothing will happen ; no statement from the *body-of-if* will be executed.

NOTE

Indentation in Python is very important when you are using a compound statement. Make sure to indent all statements in one block at the same level.

Consider another example illustrating the use of *if* statement :

```
ch = input ( "Enter a single character :" )
if ch >= '0' and ch <= '9' :
    print ("You entered a digit.")
```

The above code, after getting input in ch, compares its value ; if value of ch falls between characters '0' to '9' *i.e.*, the condition evaluates to *true*, and thus it will execute the statements in the *if-body*; that is, it will print a message saying 'You entered a digit.'

Now consider another example that uses two if statements one after the other :

```
ch = input ( 'Enter a single character : ' )
if ch == ' ' :
    print ("You entered a space")
if ch >= '0' and ch <= '9' :
    print ("You entered a digit.")
```

If this condition is false then the control will directly reach to following if statement, ignoring the body-of this-if statement

The above code example reads a single character in variable ch. If the character input is a space, it flashes a message specifying it. If the character input is a digit, it flashes a message specifying it.

The following example also makes use of an *if* statement :

```
A = int ( input ( "Enter first integer :" ) )
B = int ( input ( "Enter second integer :" ) )
    if A > 10 and B < 15 :
    C = (A - B) * (A + B)
    print ("The result is", C)
print ("Program over")
```

This statement is not part of if statement as it not indented at the same level as that of body of if statements.

The above program has an *if* statement with two statements in its body ; last *print* statement is not part of *if* statement.

Have a look at some more examples of conditional expressions in if statements :

(*a*)
```
if grade == 'A' :                    # comparison with a literal
    print ("You did well")
```

(*b*)
```
if a > b :                           # comparing two variables
    print ("A has more than B has")
```

(*c*)
```
if x :                               # testing truth value of a variable
    print ("x has truth value as true")
    print ("Hence you can see this message.")
```

You have already learnt about truth values and ***truth value testing*** in ***Chapter 3 under section 3.4.3A.*** I'll advise you to have a look at section 3.4.3A once again as it will prove very helpful in understanding conditional expressions.

Now consider one more test condition :

```
if not x :                    # not x will return True when x has a truth value as false
    print ("x has truth value as false this time")
```

The value of **not** *x* will be *True* only when *x* is *false* $_{final}$ and *False* when *x* is *true* $_{final}$.

The above discussed plain if statement will do nothing if the condition evaluates to *false* and moves to the next statement that follows if-statement.

4.5.2 The if – else Statement

This form of *if* statement tests a condition and if the condition evaluates to *true*, it carries out statements indented below if and in case condition evaluates to *false*, it carries out statements indented below else.

The syntax (general form) of the *if-else* statement is as shown below :

```
if <conditional expression> :
    statement
    [statements]
else :
    statement
    [statements]
```

colons after both if and else

The block below if gets executed if the condition evaluates to *true* and the block below else gets executed if the condition evaluates to *false*.

For instance, consider the following code fragment :

```
if a >= 0 :
    print (a, "is zero or a positive number")
else :
    print (a, "is a negative number")
```

For any value more than zero (say 7) of variable *a*, the above code will print a message like :

```
7 is zero or a positive number
```

And for a value less than zero (say -5) of variable *a*, the above code will print a message like :

```
-5 is a negative number
```

Unlike previously discussed plain-if statement, which does nothing when the condition results into *false*, the if-else statement performs some action in both cases whether condition is *true* or *false*. Consider one more example :

```
if sales >= 10000 :
    discount = sales * 0.10
else :
    discount = sales * 0.05
```

The above statement calculates *discount* as 10% of *sales amount* if it is 10000 or more otherwise it calculates discount as 5% of *sales amount*.

NOTE

An if-else in comparison to two successive if statements has less number of condition-checks i.e., with if-else, the condition will be tested once in contrast to two comparisons if the same code is implemented with two successive ifs.

Following figure (Fig. 4.5) illustrates *if* and *if-else* constructs of Python.

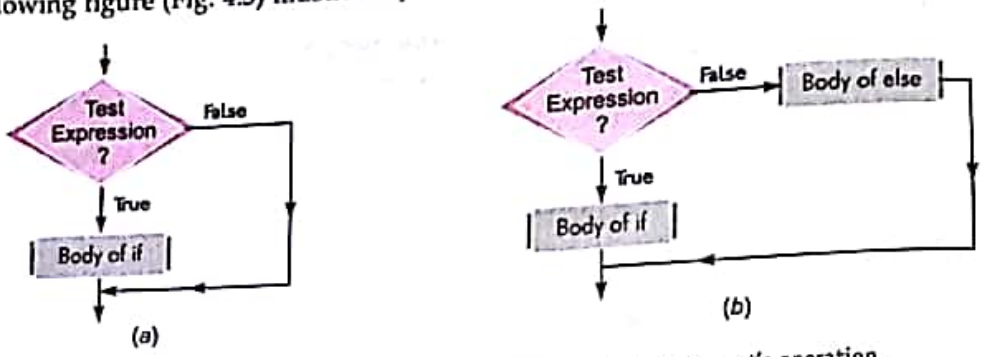
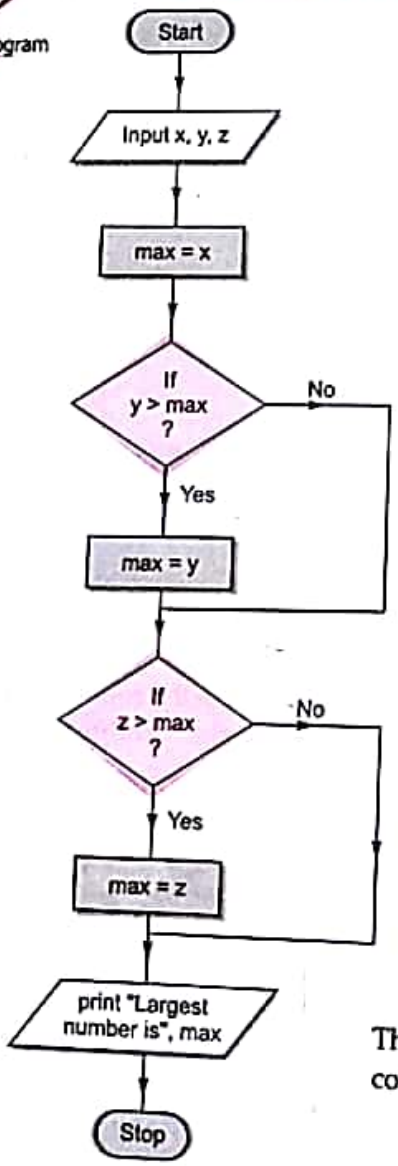


Figure 4.5 (*a*) The if statement's operation (*b*) The if-else statement's operation.

Let us now apply this knowledge of *if* and *if-else* in form of some programs.

Program 4.1 *Program to accept three integers and print the largest of the three. Make use of only if statement.*




```
Input three numbers x, y, z
max = x          #first number
If second number y is more than max then
max = y
If third number z is more than max then
max = z
Display max as the largest number
```

Pseudocode

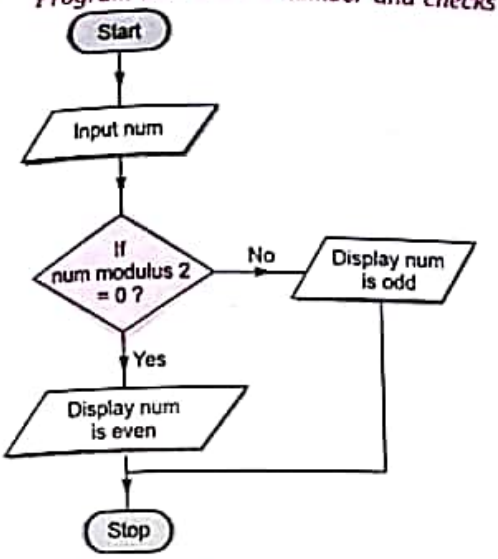
Code in Python

```
x = y = z = 0
x = float (input( "Enter first number :" ) )
y = float (input( "Enter second number :" ) )
z = float (input( "Enter third number :" ) )
max = x
if y > max :
    max = y
if z > max :
    max = z
print ("Largest number is", max)
```

The sample run of above code is as shown here.

```
Enter first number : 7.235
Enter second number : 6.99
Enter third number : 7.533
Largest number is  7.533
```

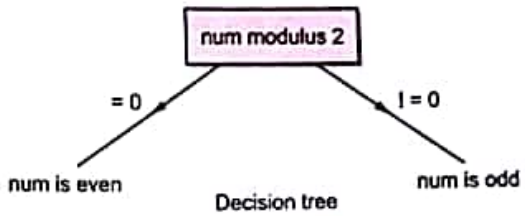
Program 4.2 *Program that takes a number and checks whether the given number is odd or even.*



Flowchart

```
Input num
    If num modulus 2 is equal to 0 then
        display num is even
    else
        display num is odd
```

Pseudocode



Decision Tree

Code in Python

```
num = int ( input( "Enter an integer :" ))
if num % 2 == 0 :
    print (num, "is EVEN number.")
else :
    print (num, "is ODD number.")
```

Sample run of above program is as shown below :

```
Enter an integer : 8
8  is EVEN number.
```

Program 4.3 *Program that inputs three numbers and calculates two sums as per this :*

Sum1 as the sum of all input numbers

Sum2 as the sum of non-duplicate numbers; if there are duplicate numbers in the input, ignores them

e.g., Input of numbers 2, 3, 4 will give two sums as 9 and 9

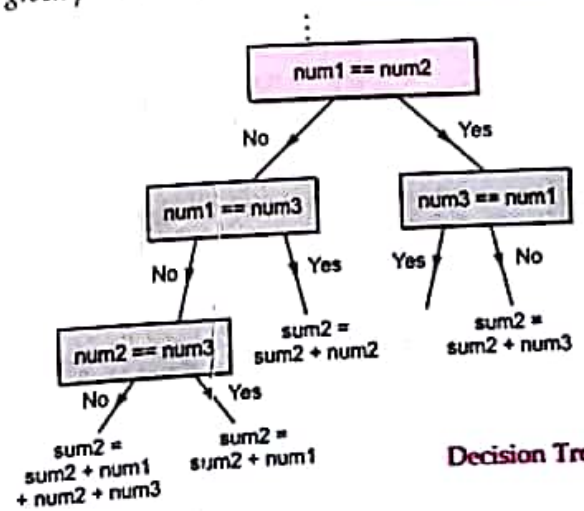
Input of numbers 3,2, 3, will give two sums as 8 and 2 (both 3's ignored for second sum)

Input of numbers 4, 4, 4 will give two sums as 12 and 0 (all 4's ignored for second sum)

Here we are giving two different ways of solving the given problem. Ist alternative's pseudocode is given and 2nd alternative's decision tree is given :

```
sum 1 = 0, sum 2 = 0
input three numbers as num1, num2, num3
sum 1 = num1 + num2 + num3
if num1 not equal to num2 or num3
    sum2 = sum2 + num1
if num2 not equal to num1 or num3
    sum2 = sum2 + num2
if num3 not equal to num1 or num2
    sum2 = sum2 + num3
display sum1 and sum2
```

Pseudocode



Decision Tree

Alternative 1

```
sum1 = sum2 = 0
num1 = int(input("Enter number 1 : "))
num2 = int(input("Enter number 2 : "))
num3 = int(input("Enter number 3 : "))

sum1 = num1 + num2 + num3
if num1 != num2 and num1 != num3 :
  sum2 += num1
if num2 != num1 and num2 != num3 :
  sum2 += num2
if num3 != num1 and num3 != num2 :
  sum2 += num3
print("Numbers are", num1, num2, num3)
print("Sum of three given numbers is", sum1)
print("Sum of non-duplicate numbers is", sum2)
```

Alternative 2

```
sum1 = sum2 = 0
num1 = int(input("Enter number 1 : "))
num2 = int(input("Enter number 2 : "))
num3 = int(input("Enter number 3 : "))

sum1 = num1 + num2 + num3

if num1 == num2 :
    if num3 != num1 :
        sum2 += num3
else :
    if num1 == num3 :
        sum2 += num2
    else :
        if num2 == num3 :
            sum2 += num1
        else :
            sum2 += num1 + num2 + num3

print("Numbers are", num1, num2, num3)
print("Sum of three given numbers is", sum1)
print("Sum of non-duplicate numbers is", sum2)
```

NOTE

Please note that the pseudocode and decision trees are given here just for understanding purpose, to make you understand that how easily you get the logic behind these solutions using these tools.

Sample run of above program is as shown below :

```
Enter number 1 : 2
Enter number 2 : 3
Enter number 3 : 4
Numbers are 2 3 4
Sum of three given numbers is 9
Sum of non-duplicate numbers is 9
==========
Enter number 1 : 3
Enter number 2 : 2
Enter number 3 : 3
Numbers are 3 2 3
Sum of three given numbers is 8
Sum of non-duplicate numbers is 2
==========
Enter number 1 : 4
Enter number 2 : 4
Enter number 3 : 4
Numbers are 4 4 4
Sum of three given numbers is 12
Sum of non-duplicate numbers is 0
```

Program 4.4 *Program to test the divisibility of a number with another number (i.e., if a number is divisible by another number)*

```
number1 = int(input("Enter first number :"))
number2 = int(input("Enter second number :"))
remainder = number1 % number2
if remainder == 0 :
    print(number1,"is divisible by", number2)
else :
    print(number1,"is not divisible by", number2)
```

Sample run of the program is as shown below :

```
Enter first number : 119
Enter second number : 3
119.0 is not divisible by 3.0
==========
Enter first number : 119
Enter second number : 17
119.0 is divisible by 17.0
==========
Enter first number : 1234.30
Enter second number : 5.5
1234.3 is not divisible by 5.5
```

Program 4.5 Program to find the multiples of a number (the divisor) out of given five numbers.

```
print("Enter five numbers below")
num1 = float(input("First number :"))
num2 = float(input("Second number :"))
num3 = float(input("Third number :"))
num4 = float(input("Fourth number :"))
num5 = float(input("Fifth number :"))
divisor = float(input("Enter divisor number :"))
count = 0
print("Multiples of", divisor, "are :")
remainder = num1 % divisor
if remainder == 0 :
    print(num1, sep =" ")
    count += 1
remainder = num2 % divisor
if remainder == 0 :
    print(num2, sep =" ")
    count += 1
remainder = num3 % divisor
if remainder == 0 :
    print(num3, sep =" ")
    count += 1
remainder = num4 % divisor
if remainder == 0 :
    print(num4, sep =" ")
    count += 1
remainder = num5 % divisor
if remainder == 0 :
    print(num5, sep =" ")
    count += 1
print()
print(count, "multiples of", divisor, "found")
```

Sample run of the program is as shown below :

```
Enter five numbers below
First number : 185
Second number : 3450
Third number : 1235
Fourth number : 1100
Fifth number : 905
Enter divisor number : 15
Multiples of 15.0 are :
3450.0
1 multiples of 15.0 found
```

Program 4.6 *Program to display a menu for calculating area of a circle or perimeter of a circle.*

```
radius = float ( input ( "Enter radius of the circle :" ))
print ("1. Calculate Area")
print ("2. Calculate Perimeter")
choice = int (input( "Enter your choice (1 or 2 ) :" ))
if choice == 1 :
    area = 3.14159 * radius * radius
    print ("Area of circle with radius", radius, 'is', area)
else :
    perm = 2 * 3.14159 * radius
    print ("Perimeter of circle with radius", radius, 'is', perm)
```

Two sample runs of above program is as shown below :

```
Enter radius of the circle : 2.5
     1. Calculate Area
     2. Calculate Perimeter
Enter your choice (1 or 2 ) : 1
Area of circle with radius  2.5  is 19.6349375
========== RESTART ==========
Enter radius of the circle : 2.5
     1. Calculate Area
     2. Calculate Perimeter
Enter your choice (1 or 2 ) : 2
Perimeter of circle with radius 2.5 is 15.70795
```

Notice, the test condition of *if* statement can be any relational expression or a logical statement (*i.e.*, a statement that results into either *true* or *false*). The *if* statement is a compound statement, hence its both **if** and **else** lines must end in a colon and statements part of it must be indented below it.

4.5.3 The if - elif Statement

Sometimes, you need to check another condition in case the test-condition of *if* evaluates to *false*. That is, you want to check a condition when control reaches **else** part, *i.e.*, condition test in the form of **else if**. To understand this, consider this example :

```
if runs are more than 100
    then it is a century
else if runs are more than 50
    then it is a fifty
else
    batsman has neither scored a century nor fifty
```

Refer to program 4.3 given earlier, where we have used **if** inside another **if/else**.

To serve conditions as above *i.e.*, in **else if** form (or **if** inside an **else**), Python provides *if-elif* and *if-elif-else* statements.

The general form of these statements is :

```
if <conditional expression> :
    statement
    [statements]
elif <conditional expression> :
    statement
    [statements]
```

NOTE

if, elif and else all are block or compound statements.

and

```
if <conditional expression> :
    statement
    [statements]
elif <conditional expression> :
    statement
    [statements]
else :
    statement
    [statements]
```

Now the above mentioned example can be coded in Python as :

```
if runs >= 100 :
    print ("Batsman scored a century")
elif runs >= 50 :
    print ("Batsman scored a fifty")
else :
    print ("Batsman has neither scored a century nor fifty")
```

Python will test this condition in case previous condition (runs >= 100) is false.

This block will be executed when both the if's condition (i.e., runs >=100) and elif condition (i.e., runs>=50) are false.

Let us have a look at some more code examples :

```
if num < 0 :
    print (num, "is a negative number.")
elif num == 0 :
    print (num, "is equal to zero.")
else :
    print (num, "is a positive number.")
```

Can you make out what the above code is doing ? Hey, don't get angry. I was just kidding. I know you know that it is testing a number whether it is negative (< 0), or zero (= 0) or positive (> 0). Consider another code example :

```
if sales >= 30000 :
    discount = sales * 0.18
elif sales >= 20000 :
    discount = sales * 0.15
elif sales >= 10000 :
    discount = sales * 0.10
else :
    discount = sales * 0.05
```

*There can be as many **elif** blocks as you need. Here, the code is having two **elif** blocks*

This block will be executed when none of above conditions are true.

Consider following program that uses an if-elif statement.

Program 4.7 *Program that reads two numbers and an arithmetic operator and displays the computed result.*

```
num1 = float ( input( "Enter first number :" ) )
num2 = float ( input( "Enter second number :" ) )
op = input( "Enter operator [ + - * / % ] :" )
result = 0
if op == '+' :
    result = num1 + num2
elif op == '-' :
    result = num1 - num2
elif op == '*' :
    result = num1 * num2
elif op == '/' :
    result = num1 / num2
elif op == '%' :
    result = num1 % num2
else :
    print ("Invalid operator!!")
print (num1, op, num2 , '=', result)
```

Some sample runs of above program are being given below :

```
Enter first number : 5
Enter second number : 2
Enter operator [ + - * / % ] : *
5.0 * 2.0 = 10.0
========== RESTART ==========
Enter first number : 5
Enter second number : 2
Enter operator [ + - * / % ] : /
5.0 / 2.0 = 2.5
========== RESTART ==========
Enter first number : 5
Enter second number : 2
Enter operator [ + - * / % ] : %
5.0 % 2.0 = 1.0
```

Now try writing code of program 4.3 using **if-elif-else** statements.

4.5.4 The nested if Statement

Sometimes above discussed forms of if are not enough. You may need to test additional conditions. For such situations, Python also supports *nested-if form of if.*

A *nested if* is an *if* that has another if in its *if*'s body or in *elif*'s body or in its *else*'s body.

The nested if can have one of the following forms :

Form 1 (*if inside if's body*)

```
if <conditional expression> :
   if <conditional expression> :
        statement(s)
   else :
        statement(s)
elif <conditional expression> :
   statement
   [statements]
else :
   statement
   [statements]
```

Form 2 (*if inside elif's body*)

```
if <conditional expression> :
   statement
   [statements]
elif <conditional expression> :
   if <conditional expression> :
        statement(s)
   else :
        statement(s)
else :
   statement
   [statements]
```

Form 3 (*if inside else's body*)

```
if <conditional expression> :
   statement
   [statements]
elif <conditional expression> :
   statement
   [statements]
else :
   if <conditional expression> :
        statement(s)
   else :
        statement(s)
```

Form 4 (*if inside if's as well as else's or elif's body, i.e., multiple ifs inside*)

```
if <conditional expression> :
   if <conditional expression> :
        statement(s)
   else :
        statement(s)
elif <conditional expression> :
   if <conditional expression> :
        statement(s)
   else :
        statement(s)
else :
   if <conditional expression> :
        statement(s)
   else :
        statement(s)
```

In a nested if statement, either there can be *if* statement(s) in its *body-of-if* or in *its body-of-elif* or in its *body-of-else* or in any two of these or in all of these. Recall that you used **nested-if** unknowingly in program 4.3. Isn't that superb ? ;)

Following example programs illustrate the use of nested ifs.

Program 4.8 *Program that reads three numbers (integers) and prints them in ascending order.*

```
x = int(input( "Enter first number :" ))
y = int(input( "Enter second number :" ))
z = int(input( "Enter third number :" ))
min = mid = max = None

if x < y and x < z :
    if y < z :
        min, mid, max = x, y, z
    else :
        min, mid, max = x, z, y
elif y < x and y < z :
    if x < z :
        min, mid, max = y, x, z
    else :
        min, mid, max = y, z, x
else :
    if x < y :
        min, mid, max = z, x, y
    else:
        min, mid, max = z, y, x

print ("Numbers in ascending order :", min, mid, max)
```

Two sample runs of the program are as given below :

```
Enter first number : 5
Enter second number : 9
Enter third number : 2
Numbers in ascending order : 2 5 9
========== RESTART ==========
Enter first number : 9
Enter second number : 90
Enter third number : 19
Numbers in ascending order : 1 19 9
```

Let us now have a look at some programs that use different forms of *if* statement.

Program 4.9 *Program to print whether a given character is an uppercase or a lowercase character or a digit or any other character.*

```
ch = input( "Enter a single character :" )
if ch >= 'A' and ch <= 'Z' :
  print ("You entered an Upper case character.")
elif ch >= 'a' and ch <= 'z' :
  print ("You entered a lower case character.")
elif ch >= '0' and ch <= '9' :
  print ("You entered a digit.")
else :
  print ("You entered a special character.")
```

Sample run of the program is as shown below :

```
Enter a character : 5
You entered a digit.
========== RESTART ==========
Enter a character : a
You entered a lower case character.
========== RESTART ==========
Enter a character : H
You entered an Upper case character.
========== RESTART ==========
Enter a character : $
You entered a special character.
```

Program 4.10 *Program to calculate and print roots of a quadratic equation : $ax^2 + bx + c = 0$ $(a \neq 0)$.*

```
import math

print ("For quadratic equation, ax**2 + bx + c = 0, enter coefficients below")
a = int( input ( "Enter a :" ) )
b = int( input ( "Enter b :" ) )
c = int( input ( "Enter c :" ) )

if a == 0 :
    print ("Value of", a, ' should not be zero')
    print ("\n Aborting !!!!!!")
else :
    delta = b * b - 4 * a * c
    if  delta > 0 :
        root1 = ( - b + math.sqrt(delta)) / (2 * a)
        root2 = (- b - math.sqrt(delta)) / (2 * a)
        print ("Roots are REAL and UNEQUAL")
        print ("Root1 =", root1, ", Root2 =", root2)
    elif delta == 0 :
        root1 = - b / (2 * a) ;
        print ("Roots are REAL and EQUAL")
        print ("Root1 =", root1, ", Root2 =", root1)
    else :
        print ("Roots are COMPLEX and IMAGINARY")
```

NOTE

You can use sqrt( ) function of math package to calculate squareroot of a number. Use it as :

`math.sqrt(<number or expression>)`

To use this function, add first line as import math to your program

NOTE

An *if* statement is also known as a conditional.

```
For quadratic equation, ax**2 + bx + c = 0, enter coefficients below
      Enter a : 3
      Enter b : 5
      Enter c : 2
Roots are REAL and UNEQUAL
Root1 =  -0.666666666667 ,  Root2 =  -1.0
========================== RESTART ============================
For quadratic equation, ax**2 + bx + c = 0, enter coefficients below
      Enter a : 2
      Enter b : 3
      Enter c : 4
Roots are COMPLEX and IMAGINARY
========================== RESTART ============================
For quadratic equation, ax**2 + bx + c = 0, enter coefficients below
      Enter a : 2
      Enter b : 4
      Enter c : 2
Roots are REAL and EQUAL
Root1 =  -1 , Root2 =  -1
```

Storing Conditions

Sometimes the conditions being used in code are complex and repetitive. In such cases, to make your program more readable, you can use **named conditions** *i.e.*, you can store conditions in a name and then use that named conditional in the if statements.

For example, to create conditions similar to the ones given below :

- If a deposit is less than ₹2000 and for 2 or more years, the interest rate is 5 percent.
- If a deposit is ₹2000 or more but less than ₹6000 and for 2 or more years, the interest rate is 7 percent.
- If a deposit is more than ₹6000 and is for 1 year or more, the interest is 8 percent.
- On all deposits for 5 years or more, interest is 10 percent compounded annually.

The traditional way of writing code will be :

```
if deposit < 2000 and time >= 2 :
    rate = 0.05
    :
```

But if you name the conditions separately and use them later in your code, it adds to readability and understandability of your code, *e.g.*,

```
eligible_for_5_percent = deposit < 2000 and time >= 2
eligible_for_7_percent = 2000 <= deposit < 6000 and time >=2
eligible_for_8_percent = deposit > 6000 and time >= 1
eligible_for_10_percent = time >= 5
```

Now you can use these *named conditionals* in the code as follows :

```
if eligible_for_5_percent :
    rate = 0.05
elif eligible_for_7_percent :
    rate = 0.075
    :
```

TIP

You can use named conditions in if statements to enhance readability and reusability of conditions.

Check Point 4.2

1. What is a selection statement? Which selection statements does Python provide ?
2. How does a conditional expression affect the result of if statement?
3. Correct the following code fragment :

```
if (x = 1)
  k = 100
else
  k = 10
```

4. What will be the output of following code fragment if the input given is (*i*) 7 (*ii*) 5 ?

```
a = input('Enter a number')
if (a == 5) :
  print ("Five" )
else :
  print ("Not Five")
```

5. What will be the output of the following code fragment if the input given is (*i*) 2000 (*ii*) 1900 (*iii*) 1971 ?

```
year = int(input( "Enter 4-digit year"))
if (year % 100 == 0) :
  if (year % 400 == 0):
    print ("LEAP century year")
else :
  print ("Not century year")
```

6. What will be the output of the following code fragment if the input given is (*i*) 2000 (*ii*) 1900 (*iii*) 1991 ?

```
year = int(input( "Enter 4-digit year"))
if (year % 100 == 0) :
  if (year % 400 == 0):
    print ("LEAP century year")
  else :
    print ("Not century year")
```

DECISION CONSTRUCT *if* — Progress In Python 4.1

This 'Progress in Python' session is aimed at laying string foundation of decision constructs.

:

>>>❖<<<

4.6 REPETITION OF TASKS – A NECESSITY

We mentioned earlier that there are many situations where you need to repeat same set of tasks again and again *e.g.*, the tasks of making chapatis/dosas/appam at home, as mentioned earlier. Now if you have to write pseudocode for this task how would you do that.

Let us first write the task of making dosas/appam from a prepared batter.

1. Heat flat pan/tawa
2. Spread evenly the dosa/appam batter on the heated flat pan.
3. Flip it carefully after some seconds.
4. Flip it again after some seconds.
5. Take it off from pan.
6. To make next dosa/appam go to step 2 again.

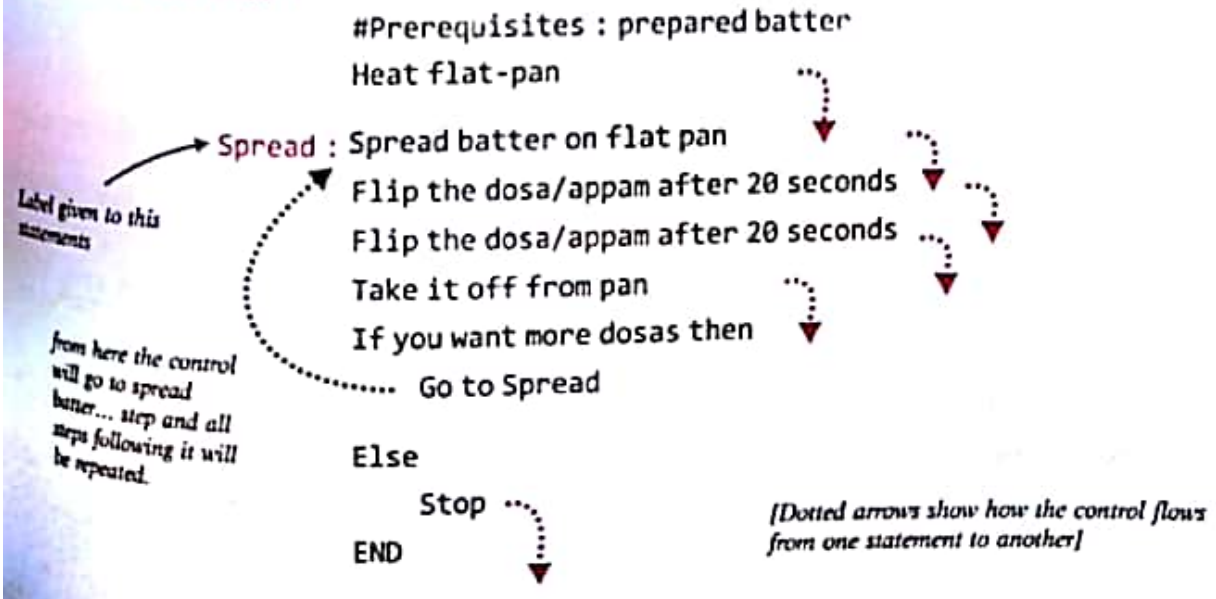
Let us try to write pseudocode for above task of making dosas/appam as many as you want.

```
#Prerequisites : Prepared batter
Heat flat-pan/tawa
Spread batter on flat pan
Flip the dosa/appam after 20 seconds
Flip the dosa/appam after 20 seconds
Take it off from pan
if you want more dosas then
    ????
else
    stop
```

Now what should you write at the place of ????, so that it starts repeating the process again from second step ? Since there is no number associated with steps, how can one tell from where to repeat ?

There is a way out : If there is a **label** that marks the statement from where you want to repeat the task, then we may send the control to that label and then that statement onwards, statements get repeated. That is,

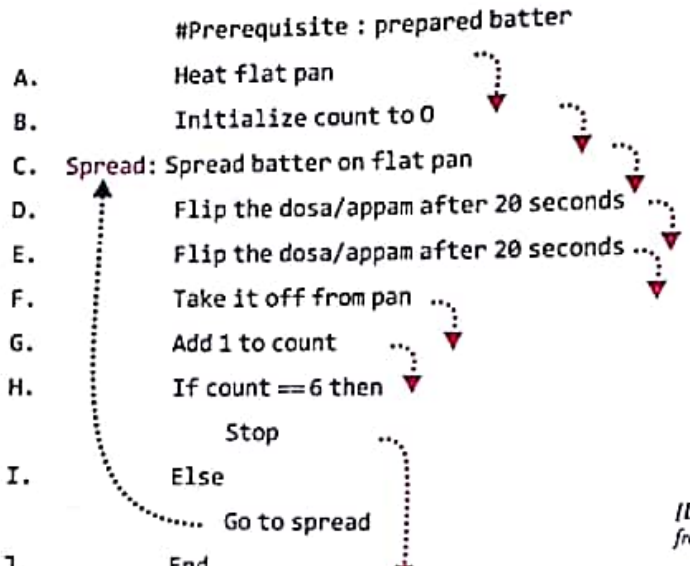
Pseudocode Reference 1



Now the above pseudocode is fit for making as many dosas/appam as you want. But what if you already know that you need only 6 dosas/appam ? In that case above pseudocode will not serve the purpose.

Let us try to write pseudocode for this situation. For this, we need to maintain a count of dosa/appam made.

Pseudocode Reference 2



[Dotted arrows show how the control flows from one statement to another]

If we number the pseudocode statements as A.B.C.... as shown above for our reference, if you notice carefully, the statements are executed as :

A B CDEFGHI CDEFGHI CDEFGHI

Statements repeated

CDEFGHI CDEFGHI CDEFGHI J.

You can see that statements C to I are repeated 6 times or we can say that loop repeated 6 times.

This way we can develop logic for carrying out repetitive tasks, either based on a condition (*pseudocode reference 1*) or a given number of times (*pseudocode reference 2*).

To carry out repetitive tasks, Python does not have any *go to statement* rather it provides following iterative/looping statements :

- *Conditional loop* **while** (condition based loop)
- *Counting loop* **for** (loop for a given number of times).

Let us learn to write code for repetitive tasks, in Python.

4.7 THE range( ) FUNCTION

Before we start with loops, especially *for loop* of Python, let us discuss the *range( )* function of Python, which is used with the Python *for loop*. The *range ( )* function of Python generates a list which is a special sequence type. A sequence in Python is a succession of values bound together by a single name. Some Python sequence types are : *strings, lists, tuples* etc. (see Fig. 4.6). There are some advance sequence types also but those are beyond the scope of our discussion.

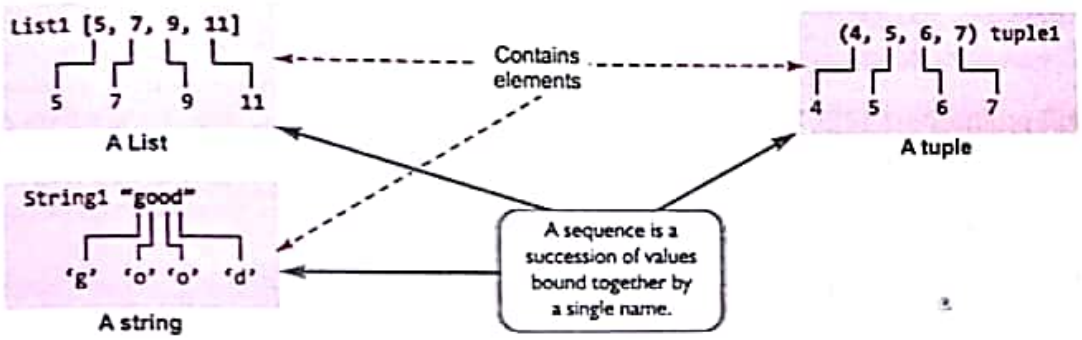


Figure 4.6 Some common sequence types

Let us see how *range( )* function works. The common use of *range( )* is in the form given below :

```
range( <lower limit>, <upper limit>)    # both limits should be integers
```

The function in the form range(l, u) will produce a list having values starting from $l, l+1, l+2$ ... $(u-1)$ (l and u being integers). Please note that the lower limit is included in the list but upper limit is not included in the list, *e.g.*,

```
range(0,5)
```
<---------- *the default step-value in values will be +1*

will produce list as [0, 1, 2, 3, 4].

As these are the numbers in arithmetic progression (a.p.) that begins with lower limit 0 and goes up till *upper limit minus 1 i.e.*, 5 – 1 = 4.

```
range(5,0)
```
<---------- *default step-value = +1*

will return empty list [ ] as no number falls in the a.p. beginning with 5 and ending at 0 (difference d or step-value = +1).

```
range(12, 18)
```
<---------- *default step-value = +1*

will give a list as [12, 13, 14, 15, 16, 17].

All the above examples of *range( )* produce numbers increasing by value 1. What if you want to produce a list with numbers having gap other than 1, *e.g.*, you want to produce a list like [2, 4, 6, 8] using range( ) function ?

For such lists, you can use following form of *range( )* function :

```
range( <lower limit>, <upper limit>, <step value>)    #all values should be integers
```

That is, function

```
range(l, u, s)          # l, u and s are integers
```

will produce a list having values as $l, l+s, l+2s, \ldots <= u-1$.

```
range(0, 10, 2)   <-------- step-value = +2
```

will produce list as [0, 2, 4, 6, 8]

```
range (5, 0, -1)  <-------- step-value = -1
```

will produce list as [5, 4, 3, 2, 1].

NOTE

A sequence in Python is a succession of values bound together by a single name e.g., list, tuple, string. The range( ) returns a sequence of list type.

Another form of range( ) is :

```
range(<number>)
```

that creates a list from 0 (zero) to <number> – 1, *e.g.*, consider following *range( )* function :

```
range(5)
```

The above function will produce list as [0, 1, 2, 3, 4]. Consider some more examples of *range( )* function :

Statement	Values generated
range(10)	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
range(5, 10)	5, 6, 7, 8, 9
range(3, 7)	3, 4, 5, 6
range(5, 15, 3)	5, 8, 11, 14
range(9, 3, –1)	9, 8, 7, 6, 5, 4
range(10, 1, –2)	10, 8, 6, 4, 2

Operators in and not in

Let us also take about **in** operator, which is used with *range( )* in for loops.

To check whether a value is contained inside a list you can use **in** operator, *e.g.*,

```
3 in [1,2,3,4]   <-------- This expression will test if value 3 is contained in the given sequence
```

will return *True* as value 3 is contained in sequence [1, 2, 3, 4].

NOTE

The In operator tests if a given value is contained in a sequence or not and returns True or False accordingly.

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

will return *False* as value 5 is not contained in sequence [1, 2, 3, 4]. But

```
5 not in [1,2,3,4]
```

will return *True* as this fact is true that as value 5 is not contained in sequence [1, 2, 3, 4]. The operator **not in** does opposite of **in** operator.

You can see that **in** and **not in** are the operators that check for membership of a value inside a sequence. Thus, **in** and **not in** are also called **membership operators**. These operators work with all sequence types *i.e., strings, tuples* and *lists* etc.

For example, consider following code :

```
'a' in "trade"
```

will return *True* as '*a*' is contained in sequence (string type) "trade".

```
"ash" in "trash"
```

will also return *True* for the same reason.

NOTE

Operators in and not in are also called membership operators.

Now consider the following code that uses the in operator :

```
line = input( "Enter a line :" )
string = input( "Enter a string :" )
if string in line :                    # if the given string is a part of line
    print (string, "is part of", line.)
else :
    print (string, "is not contained in", line.)
```

You can easily make out what the above code is doing – it is checking whether a string is contained in a line or not.

Now that you are familiar with range( ) function and in operator, we can start with Looping (Iteration) statements.

4.8 ITERATION/LOOPING STATEMENTS

The iteration statements or repetition statements allow a set of instructions to be performed repeatedly until a certain condition is fulfilled. The iteration statements are also called ***loops*** or ***looping statements***. Python provides *two* kinds of loops : *for loop* and *while loop* to represent *two* categories of loops, which are :

- *counting loops* — the loops that repeat a certain number of times ; Python's *for loop* is a counting loop.
- *conditional loops* — the loops that repeat until a certain thing happens *i.e.*, they keep repeating as long as some condition is *true* ; Python's *while loop* is conditional loop.

4.8.1 The for Loop

The ***for loop*** of Python is designed to process the items of any sequence, such as a list or a string, one by one. The general form of ***for loop*** is as given below :

```
for <variable> in <sequence> :
    statements_to_repeat
```

This determines how many time the loop will get repeated.

Colon is must here

here the statements to be repeated will be placed (body-of-the-loop)

For example, consider the following loop :

```
for a in [1, 4, 7] :
    print (a)
    print(a * a)
```

The loop variable a. Variable a will be assigned each value of list one by one, i.e., for the first time a will be 1, then 4 and then 7.

This is the body of the for loop. All statements in the body of the loop will be executed for each value of loop variable a, i.e., firstly for a = 1; then for a = 4 and then for a=7

A *for loop* in Python is processed as :

- The loop-variable is assigned the first value in the sequence.
- all the statements in the body of *for loop* are executed with assigned value of loop variable (step 2)
- once step 2 is over, the loop-variable is assigned the next value in the sequence and the loop-body is executed (*i.e.*, step 2 repeated) with the new value of loop-variable.
- This continues until all values in the sequences are processed.

NOTE

Each time, when the loop-body is executed, is called an *iteration*.

That is, the given *for loop* will be processed as follows :

(*i*) firstly, the loop-variable *a* will be assigned first value of list *i.e.*, *1* and the statements in the body of the loop will be executed with this value of *a*. Hence value 1 will be printed with statement **print(a)** and value 1 will again be printed with **print(a*a)**. (*see execution shown on right*)

(*ii*) Next, *a* will be assigned next value in the list *i.e.*, *4* and loop-body executed. Thus 4 (as result of **print(a)**) and 16 (as result of **print(a*a)**) are printed.

(*iii*) Next, *a* will be assigned next value in the list *i.e.*, *7* and loop-body executed. Thus 7 (as result of **print(a)**) and 49 (as result of **print(a*a)**) are printed.

(*iv*) All the values in the list are executed, hence loop ends.

```
for a in [1, 4, 7]:
    print (a)        <-- body of the loop
    print (a * a)
```

(*i*)
```
a = 1
a assigned first value and with
this value, all statements in
loop body are executed :
                              prints
print (1)  ------------------------> 1
print (1*1) -----------------------> 1
                              prints
```

(*ii*)
```
a = 4
a assigned next value and
loop body executed with a as 4:
                              prints
print (4)  ------------------------> 4
print (4*4) -----------------------> 16
                              prints
```

(*iii*)
```
a = 7
a assigned next value and
loop body executed with a as 7
                              prints
print (7)  ------------------------> 7
print (7*7) -----------------------> 49
                              prints
```

```
No more values
```

Therefore, the output produced by above *for loop* will be :

```
1
1
4
16
7
49
```

Consider another *for loop* :

```
for ch in 'calm' :
    print (ch)
```

The above loop will produce output as :

```
c
a
l
m
```

(variable ch given values as 'c', 'a', 'l', 'm' – one at a time from string 'calm'.)

Here is another *for loop* that prints the square of each value in the list :

```
for v in [1, 2, 3 ] :
    print (v * v * v)
```

The above loop will produce output as :

```
1
8
81
```

The *for loop* works with other sequence types such as *tuples* also, but we'll stick here mostly to *lists* to process a sequence of numbers through *for loops*.

As long as the list contains small number of elements, you can write the *list* in the *for loop* directly as we have done in all above examples. But what if we have to iterate through a very large list such as containing natural numbers between 1 – 100 ? Writing such a big list is neither easy nor good on readability. Then ? Then what ? ☺ Arey, our very own *range( )* function is there to rescue. Why worry? ;) For big number based lists, you can specify *range( )* function to represent a list as in :

```
for val in range(3, 18) :
    print (val)
```

In the above loop, *range(3, 18)* will first generate a list [3, 4, 5, ... , 16, 17] with which *for loop* will work. Isn't that easy and simple ? ☺

You need not define loop variable (val above) before-hand in a *for loop*. Now consider following code example that uses *range( )* function in a *for loop* to print the table of a number.

NOTE

(*I*) *for* loop ends when the loop is repeated for the last value of the sequence.

(*II*) The *for loop* repeats *n* number of times, where *n* is the length of sequence given in for-loop's header.

Program 4.11 *Program to print table of a number, say 5.*

```
num = 5
for a in range(1, 11) :
    print (num, 'x', a, '=', num * a)
```

The above code will print the output as shown here :

```
5 x 1 = 5
5 x 2 = 10
5 x 3 = 15
5 x 4 = 20
5 x 5 = 25
5 x 6 = 30
5 x 7 = 35
5 x 8 = 40
5 x 9 = 45
5 x 10 = 50
```

NOTE

You need not pre-define loop variable of a *for loop*.

Program 4.12 *Program to print sum of natural numbers between 1 to 7. Print the sum progressively, i.e., after adding each natural number, print sum so far.*

```
sum = 0
for n in range(1, 8) :
    sum += n
    print ("Sum of natural numbers <=", n, "is", sum)
```

The output produced by above program will be :

```
Sum of natural numbers <= 1 is 1
Sum of natural numbers <= 2 is 3
Sum of natural numbers <= 3 is 6
Sum of natural numbers <= 4 is 10
Sum of natural numbers <= 5 is 15
Sum of natural numbers <= 6 is 21
Sum of natural numbers <= 7 is 28
```

Program 4.13 *Program to print sum of natural numbers between 1 to 7.*

```
sum = 0
for n in range(1, 8) :
    sum += n
print ("Sum of natural numbers <=", n, 'is', sum)
```

The output produced by above program is :

```
Sum of natural numbers <= 7 is 28
```

Carefully look at above program. It again emphasizes that body of the loop is defined through indentation and one more fact and important one too – *the value of loop variable after the for loop is over, is the highest value of the list*. Notice, the above loop printed value of *n* after *for loop* as 7, which is the maximum value of the list generated by *range(1, 8)*.

NOTE

The loop variable contains the highest value of the list after the *for loop* is over. Program 4.13 highlights this fact.

4.8.2 The while Loop

A *while loop* is a conditional loop that will repeat the instructions within itself as long as a conditional remains *true* (Boolean *True* or truth value $true_{text}$).

The general form of Python while loop is :

```
while <logicalexpression> :
    loop-body
```

where the loop-body may contain a *single statement* or *multiple statements* or *an empty statement* (*i.e., pass* statement). The loop iterates while the *logical expression* evaluates to *true*. When the expression becomes *false*, the program control passes to the line after the loop-body.

To understand the working of ***while loop***, consider the following code :

```
a = 5
while a > 0 :
    print ("hello", a)
    a = a - 3
print ("Loop Over!!")
```

*This condition is tested, if it is **true**, the loop-body is executed. After the loop-body's execution, the condition is tested again, loop-body is executed as long as condition is **true**.*
*The loop ends when the condition evaluates to **false***

The above code will print :

```
hello 5
hello 2
Loop Over!!
```

*These two lines are the result of **while loops'** body execution (which executed twice).*

*This line is because of print statement after the **while loop**.*

Let us see how a *while loop* is processed :

```
while a > 0 :
    print ("hello", a)
    a = a - 3
```

body of the loop

[a has value 5 before entering into while loop]

Step 1 The *logical/conditional expression in the while loop* (*e.g.*, $a > 0$ above) is evaluated.

Step 2 *Case I* if the result of step 1 is *true* (*True* or $true_{bool}$) then all statements in the loop's body are executed, (see section (*i*) & (*ii*) on the right)

Case II if the result of step 1 is *false* (*False* or $false_{bool}$) then control moves to the next statement after the body-of the loop *i.e.*, loop ends. (see section (*iii*) on the right)

Step 3 Body-of-the loop gets executed. This step comes only if *Step 2, Case I* executed, *i.e.*, the result of *logical expression* was *true*. This step will execute two statements of loop body.

(*i*) while's condition tested with a's current value as 5
5 > 0 = True
hence loop's body will get executed
print ("hello", 5) ⟶ *prints* hello 5
a = 5 - 3 = 2. (a becomes 2 now)

(*ii*) while's condition tested with a's current values 2
2 > 0 = True
loop body will get executed
print ("hello", 2) ⟶ *prints* hello 2
a = 2 - 3 = -1 (a becomes -1 now)

(*iii*) while's condition tested with a's current value -1
-1 > 0 = False
loop body will not get executed
control passes to next statement after while loop

After executing the statements of loop-body again, *Step 1*, *Step 2* and *Step 3* are repeated as long as the condition remains *true*.

The loop will end only when the logical expression evaluates to *false*. (*i.e.*, *Step 2, Case II* is executed.)

Consider another example :

```
n = 1
while n < 5 :
    print ("Square of", n , 'is', n * n)
    n += 1
print ("Thank You.")
```

← *This statement is not part of the loop-body*

The above code will print output as:

```
Square of 1 is 1
Square of 2 is 4
Square of 3 is 9
Square of 4 is 16
Thank You.
```

← *These lines of output are because of the loop-body execution (loop executed 4 times)*

Thus you see that as long as the condition $n < 5$ is *true* above, the *while loop's* body is executed (for values $n = 1$, $n = 2$, $n = 3$, $n = 4$). After exiting from the loop the statement following the *while loop* is executed that prints *Thank You*.

NOTE

The variable used in the condition of *while* loop must have some value before entering into *while* loop.

Anatomy of a while Loop (Loop Control Elements)

Every while loop has its elements that control and govern its execution. A **while** loop has four elements that have different purposes.

These elements are as given below :

1. Initialization Expression(s) (Starting)

Before entering in a *while* loop, its loop variable must be initialized. The initialization of the loop variable (or control variable) takes place under initialization expression(s). The initialization expression(s) give(s) the loop variable(s) their first value(s). The initialization expression(s) for a *while loop* are outside the *while* loop before it starts.

2. Test Expression (Repeating or Stopping)

The test expression is an expression whose truth value decides whether the loop-body will be executed or not. If the test expression evaluates to *true*, the loop-body gets executed, otherwise the loop is terminated.

In a *while* loop, the test-expression is evaluated before entering into a loop.

3. The Body-of-the-Loop (Doing)

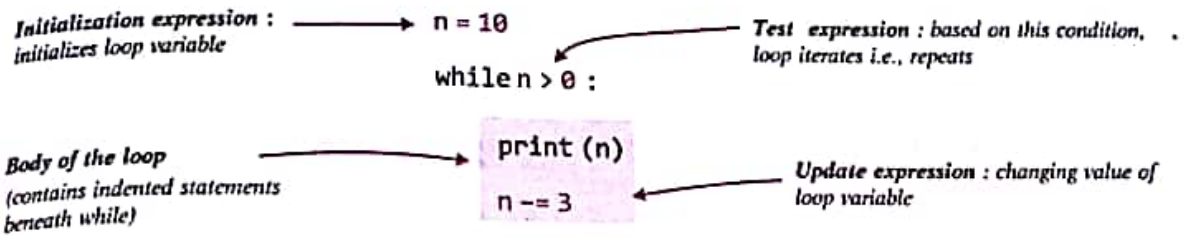
The statements that are executed repeatedly (as long as the test-expression is *true*) form the *body of the loop*.

In a *while* loop, before every iteration the test-expression is evaluated and if it is *true*, the *body-of-the-loop* is executed; if the test-expression evaluates to *false*, the loop is terminated.

4. Update Expression(s) (Changing)

The update expressions change the value of loop variable. The update expression is given as a statement inside the body of *while* loop.

Consider the following code that highlights these elements of a *while* loop :



Other important things that you need to know related to *while loop* are :

- In a *while loop*, a loop control variable should be initialized before the loop begins as an uninitialized variable cannot be used in an expression.

  Consider following code :

```
while p != 3 :
    :
```

This will result in error if variable p has not been created beforehand.

- The loop variable must be updated inside the *body-of the-while* in a way that after some time the test-condition becomes *false* otherwise the loop will become an **endless loop** or **infinite loop.**

  Consider following code :

```
a = 5
while a > 0 :
    print (a)
print ("Thank You")
```

ENDLESS LOOP! Because the loop-variable a remains 5 always as its value is not updated in the loop-body, hence the condition always remains true.

The above loop is an endless loop as the value of loop-variable *a* is not being updated inside loop body, hence it always remains 5 and the loop-condition *a* > 0 always remains *true*.

The corrected form of above loop will be :

```
a = 5
while a > 0 :
    print (a)
    a -= 1
print ("Thank You")
```

Loop variable **a** being updated *inside the loop-body.*

We will talk about infinite loops once again after the *break* statement is discussed.

Since in a *while loop*, the control in the form of condition-check is at the entry of the loop (loop is not entered, if the condition is *false*), it is an example of an **entry-controlled loop**. An *entry-controlled loop* is a loop that controls the entry in the loop by testing a condition. Python does not offer any *exit-controlled loop*.

Now that you are familiar with *while loop*, let us write some programs that make use of *while loop*.

NOTE

To cancel the running of an endless loop, you can press CTRL+C keys anytime during its endless repetition to stop it.

Program 4.14 *Program to calculate the factorial of a number.*

```
num = int( input ( "Enter a number :" ) )
fact = 1
a = 1
while a <= num :
    fact *= a                    # fact = fact*a
    a += 1                       # a = a + 1
print ("The factorial of" , num, "is" , fact)
```

Sample runs of above program are given below :

```
Enter a number : 3
The factorial of 3 is 6
========== RESTART ==========
Enter a number : 4
The factorial of 4 is 24
```

Program 4.15 *Program to calculate and print the sums of even and odd integers of the first n natural numbers.*

```
n = int(input( "Up to which natural number ?" ))
ctr = 1
sum_even = sum_odd = 0
while ctr <= n :
    if ctr % 2 == 0 :                # number is even
        sum_even += ctr
    else :
        sum_odd += ctr
    ctr += 1                         #increment the counter
print ("The sum of even integers is" , sum_even)
print ("The sum of odd integers is" , sum_odd)
```

The output produced by above code is :

```
Up to which natural number? 25
The sum of even integers is 156
The sum of odd integers is 169
```

4.8.3 Loop else Statement

Both loops of Python (*i.e.*, **for** loop and **while** loop) have an **else** clause, which is different from *else* of *if-else* statements. **The else of a loop executes only when the loop ends normally** (*i.e.*, only when the loop is ending because the *while* loop's test condition has resulted in *false* or the

for loop has executed for the last value in sequence.) You'll learn in the next section that if a **break** statement is reached in the loop, *while* loop is terminated pre-maturely even if the test-condition is still *true* or all values of sequence have not been used in a *for* loop.

NOTE

The else clause of a Python loop executes when the loop terminates normally, not when the loop is terminating because of a *break* statement.

In order to fully understand the working of *loop-else clause*, it will be useful to know the working of *break* statements. Thus let us first talk about **break** statement and then we'll get back to *loop-else* clause again with examples.

4.8.4 Jump Statements – break and continue

Python offers two jump statements to be used within loops to jump out of loop-iterations. These are *break* and *continue* statements. Let us see how these statements work.

NOTE

A *break* statement skips the rest of the loop and jumps over to the statement following the loop.

4.8.4A The break Statement

The ***break*** statement enables a program to skip over a part of the code. A ***break*** statement terminates the very loop it lies within. Execution resumes at the statement immediately following the body of the terminated statement.

The following figure (Fig. 4.7) explains the working of a break statement :

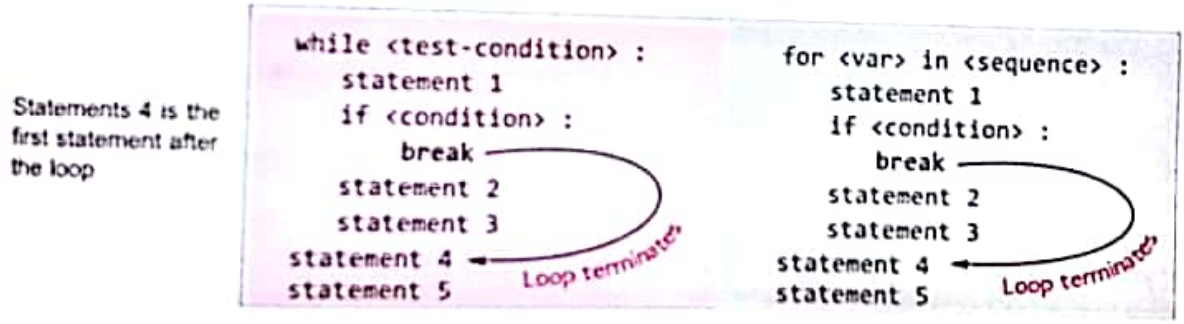


Figure 4.7 The working of a **break** statement.

The following code fragment gives you an example of a ***break*** statement :

```
a = b = c = 0
for i in range(1, 21) :
    a = int (input ( "Enter number 1 :" ))
    b = int (input ( "Enter number 2 :" ))
    if b == 0 :
        print ("Division by zero error! Aborting!")
        break
    else :
        c = a / b
        print ("Quotient = ", c)
print ("Program over !")
```

The above code fragment intends to divide ten pairs of numbers by inputting two numbers *a* and *b* in each iteration. If the number *b* is zero, the loop is immediately terminated displaying message **'Division by zero error! Aborting!'** otherwise the numbers are repeatedly input and their quotients are displayed.

Sample run of above code is given below :

```
Enter number 1 : 6
Enter number 2 : 2
Quotient =  3
Enter number 1 : 8
Enter number 2 : 3
Quotient =  2
Enter number 1 : 5
Enter number 2 : 0
Division by zero error! Aborting!
Program over !
```

This message is printed just prior to execution of break statement. Notice that the next message is because of the statement outside the loop. That is, after break, next statement to be executed is the statement outside the loop.

NOTE

A loop can end in two ways : (*i*) if the while's condition results in false or for loop executes with last value of the sequence (NORMAL TERMINATION). (*ii*) Or if during the loop execution, break statement is executed. In this case even if the condition is true or the for has not executed all the values of sequence, the loop will still terminate.

Consider another example of *break* statement in the form of following program.

Program 4.16 *Program to implement 'guess the number' game. Python generates a number randomly in the range[10, 50]. The user is given five chances to guess a number in the range 10 <= number <=50.*

- *If the user's guess matches the number generated, Python displays 'You win' and the loop terminates without completing the five iterations.*
- *If, however, cannot guess the number in five attempts, Python displays 'You lose'.*

To generate a random number in a range, use following function :

```
random.randint(lower limit, upper limit)
```

Make sure to add first line as import random

```
import random
number = random.randint(10,50)
ctr = 0
while ctr < 5 :
    guess = int(input( "Guess a number in range 10..50 :" ))
    if guess == number :
        print ("You win!! :)")
        break
    else :
        ctr += 1
if not ctr < 5 :
    print ("You lose :( \n The number was", number)
```

This will generate a number randomly, within the range 10-50.

The moment this statement is reached the loop will terminate without completing all iterations, even if the loop condition ctr < 5 is still true.

```
                    # i.e., whether the loop terminated after 5 iterations
```

Sample run of above program is as shown below :

```
Guess a number in range 10..50 : 31
Guess a number in range 10..50 : 23
Guess a number in range 10..50 : 34
Guess a number in range 10..50 : 40
Guess a number in range 10..50 : 50
You lose :(
The number was 28
========== RESTART ==========
Guess a number in range 10..50 : 39
Guess a number in range 10..50 : 46
You win!! :)
```

*At this point the **break** statement executed.*

From the above program, you can make out that if the *while loop* is terminated because of a **break** statement, the test-condition of the *while loop* will still be *true* even outside the loop. Thus, you can check the test-condition outside the loop to determine what caused the loop termination – the **break** statement or the test-condition's result. Recall the last part of above program, *i.e.*,

```
if not ctr < 5 :
    print ("You lose :( \n The number was" , number)
```

If the loop-condition is still true outside the loop that means break caused the loop termination.

Infinite Loops and break Statement

Sometimes, programmers create infinite loops purposely by specifying an expression which always remain *true*, but for such purposely created infinite loops, they incorporate some condition inside the loop-body on which they can break out of the loop using **break** statement.

For instance, consider the following loop example :

```
a = 2
while True :
    print (a)
    a *= 2
    if a > 100 :
        break
```

This will always remain True, hence infinite loop; must be terminated through a break statement

NOTE

For purposely created ***infinite*** (or ***endless***) loops, there must be a provision to reach break statement from within the body of the loop so that loop can be terminated.

4.8.4B The continue Statement

The **continue** statement is another jump statement like the **break** statement as both the statements skip over a part of the code. But the **continue** statement is somewhat different from **break**. Instead of forcing termination, the continue *statement forces the next iteration of the loop to take place*, skipping any code in between.

The following figure (Fig. 4.8) explains the working of **continue** statement :

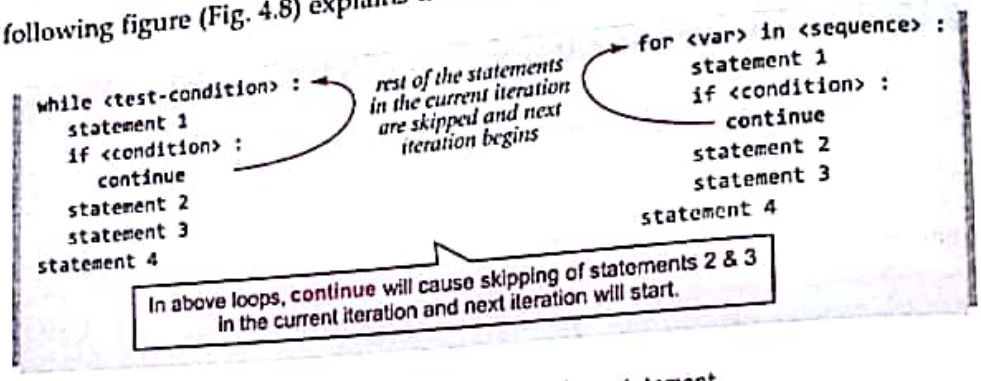


Figure 4.8 The working of a continue statement.

For the *for loop*, **continue** causes the next iteration by updating the loop variable with the next value in sequence ; and for the *while loop*, the program control passes to the conditional test given at the top of the loop.

NOTE

The **continue** statement skips the rest of the loop statements and causes the next iteration of the loop to take place.

```
a = b = c = 0
for i in range(0, 3) :
    print ("Enter 2 numbers" )
    a = int (input( "Number 1 :" ))
    b = int (input( "Number 2 :" ))
    if b == 0 :
        print ("\n The denominator cannot be zero. Enter again !")
        continue
    else :
        c = a/b
        print ("Quotient =", c)
```

Sample run of above code is shown below :

```
Enter 2 numbers
Number 1 : 5
Number 2 : 0
The denominator cannot be zero. Enter again !
Enter 2 numbers
Number 1 : 5
Number 2 : 2
Quotient = 2
Enter 2 numbers
Number 1 : 6
Number 2 : 2
Quotient = 3
```

At this point continue forced next iteration to take place.

Sometimes you need to abandon iteration of a loop prematurely. Both the statements **break** and **continue** can help in that but in different situations : statement **break** to terminate all pending iterations of the loop ; and **continue** to terminate just the current iteration, loop will continue with rest of the iterations.

Following program illustrates the difference in working of *break* and *continue* statements.

Program 4.17 *Program to illustrate the difference between break and continue statements.*

```
print ("The loop with 'break' produces output as :")
for i in range (1,11) :
    if i%3 == 0 :
        break
    else :
        print (i)

print ("The loop with 'continue' produces output as :")
for i in range (1,11) :
    if i%3 == 0 :
        continue
    else :
        print (i)
```

The output produced by above program is :

```
The loop with 'break' produces output as :
1
2
```

*because the loop terminated with break when condition i % 3 became **true** with i = 3 (Loop terminated all pending iterations)*

```
The loop with ' continue ' produces output as :
1
2
4
5
7
8
10
```

*only the values divisible by 3 are missing as the loop simply moved to next iteration with continue when condition i % 3 became **true**.*
(Only the iterations with i % 3 == 0 are terminated ; next iterations are performed as it is)

Though, **continue** is one prominent jump statement, however, experts discourage its use whenever possible. Rather, code can be made more comprehensible by the appropriate use of if/else.

4.8.5 Loop else Statement

Now that you know how **break** statement works, we can now talk about **loop-else** clause in details. As mentioned earlier, the *else clause of a Python loop* executes when the loop terminates normally, *i.e.*, when test-condition results into *false* for a ***while loop*** or ***for loop*** has executed for the last value in the sequence ; *not when* the **break** statement terminates the loop.

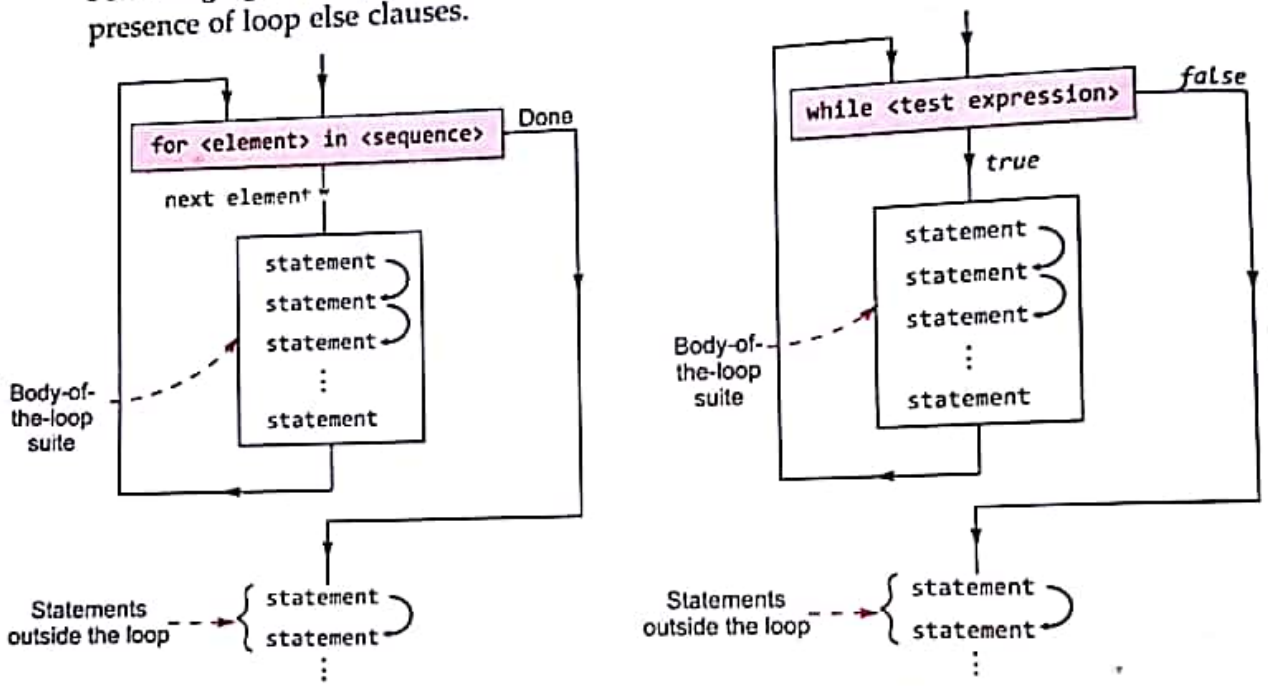
Before we proceed, have a look at the syntax of Python *loops along with else clauses*. Notice that the **else** clause of a loop appears at the same indentation as that of loop keyword **while** or **for**.

Complete syntax of Python loops along with else clause is as given below :

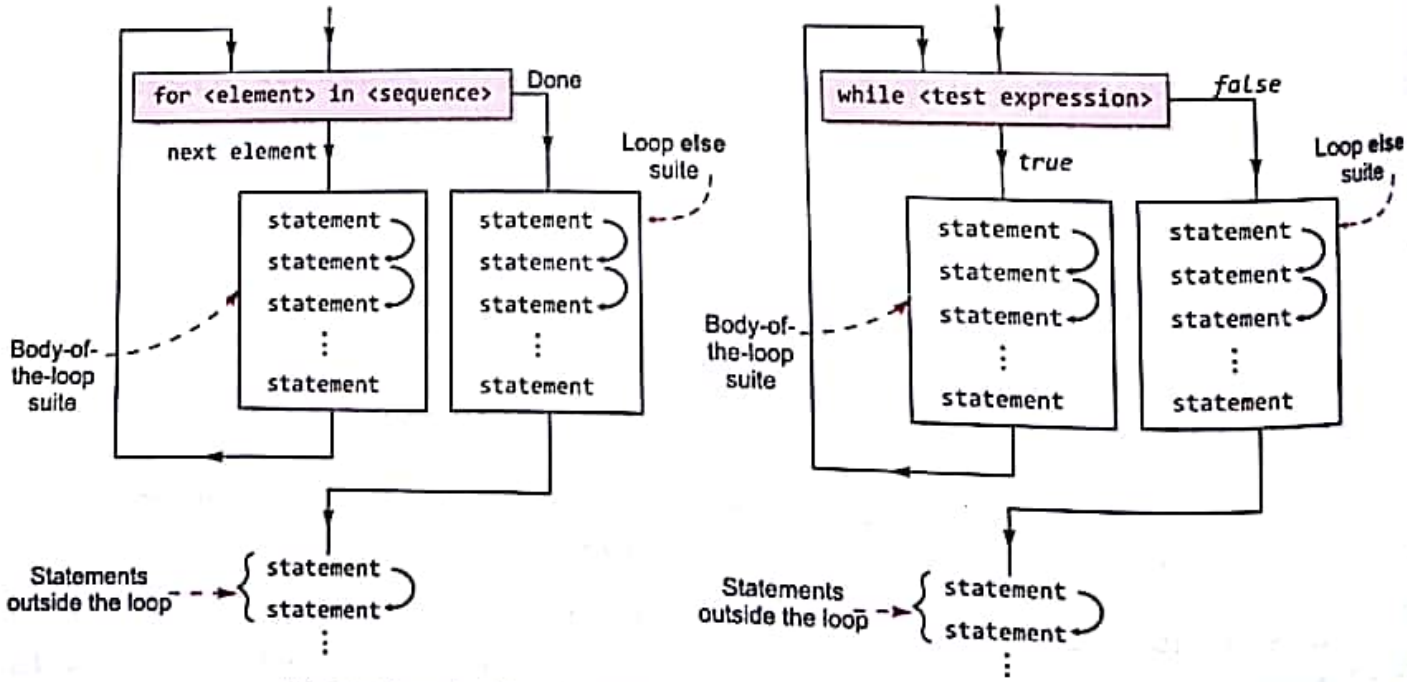
```
for <variable> in <sequence> :
    statement1
    statement2
    :
else :
    statement(s)
```

```
while <test condition> :
    statement1
    statement2
    :
else :
    statement(s)
```

Following figure (Fig. 4.9) illustrates the difference in working of these loops in the absence and presence of loop else clauses.



(a) Control flow in Python loops in the absence of loop-else change



(b) Control flow in Python loops in the presence of loop-else clause.

Figure 4.9 Working of Python Loops with or without else clauses.

Let us understand the working of the Python Loops with or without else clause with the help of code examples.

NOTE

The loop-else suite executes only in the case of normal termination of loop.

```
for a in range (1, 4) :
    print ("Element is", end = ' ')
    print (a)
else :
    print ("Ending loop after printing all elements of sequence")
```

The above will give the following output :

The output produced by for loop. Notice for loop executed for three values of a – 1, 2, 3

```
Element is 1
Element is 2
Element is 3
```

*This line is printed because the **else clause** of given for loop executed when the for loop was terminating , normally.*

```
Ending loop after printing all elements of sequence.
```

Now consider the same loop as above but with a **break** statement :

```
for a in range (1, 4) :
    if a % 2 == 0 :
        break
    print ("Element is", end = ' ')
    print (a)
else :
    print ("Ending loop after printing all elements of sequence")
```

The break statement will execute when a %2 is zero i e., when a has the value as 2

Now the output is

```
Element is 1
```

*Notice that for a = 2, the **break** got executed and loop terminated Hence just one element got printed (for a = 1). As break terminated the loop, the **else clause** also did not execute, so no line after the printing of elements.*

The **else** clause works identically in **while loop**, *i.e.*, executes if the test-condition goes *false* and not in case of **break** statement's execution.

Now try predicting output for following code, which is similar to above code but order of some statements have changed. Carefully notice the position of **else** too.

```
for a in range (1, 4) :
    print ("Element is", end = ' ')
    print (a)
    if a % 2 == 0 :
        break
    else:
        print ("Ending loop after printing all elements of sequence")
```

You are right. So smart you are. ☺ This code does not have loop-else suite. The **else** in above code is part of if-else, not loop-else. Now consider following program that uses **else clause** with a **while loop**.

Program 4.18 *Program to input some numbers repeatedly and print their sum. The program ends when the users say no more to enter (normal termination) or program aborts when the number entered is less than zero.*

```
count = sum = 0
ans = 'y'
while ans == 'y' :
    num = int( input ( "Enter number :" ) )
    if num < 0 :
        print ("Number entered is below zero. Aborting!")
        break
    sum = sum + num
    count = count + 1
    ans = input( "Want to enter more numbers? (y/n).." )
else :
    print ("You entered" , count, "numbers so far.")
print ("Sum of numbers entered is", sum)
```

Body-of-the loop suite

loop-else suite

This statement is outside the loop

```
Enter number: 3
Want to enter more numbers? (y/n)..y
Enter number: 4
Want to enter more numbers? (y/n)..y
Enter number : 5
Want to enter more numbers? (y/n)..n
You entered 3 numbers so far.
Sum of numbers entered is 12
========== RESTART ==========
Enter number : 2
Want to enter more numbers? (y/n)..y
Enter number : -3
Number entered is below zero. Aborting!
Sum of numbers entered is 2
```

Result of else clause. Loop terminated normally

No execution of else clause. Loop terminated because of break statement

Program 4.19 *Program to input a number and test if it is a prime number.*

```
num = int(input("Enter number :"))
lim = int(num/2) + 1
for i in range (2, lim) :
    rem = num % i
    if rem == 0 :
        print(num, "is not a prime number")
        break
else:
    print(num, "is a prime number")
```

```
Enter number :7
7 is a prime number
==========
Enter number :49
49 is not a prime number
```

Sometimes the loop is not entered at all, reason being empty sequence in a *for loop* or test-condition's result as *false* before entering in the while loop. In these cases, the body-of the loop is not executed but loop-else clause will still be executed.

Consider following code examples illustrating the same.

```
while ( 3 > 4) :
    print ("in the loop")
else :
    print ("exiting from while loop")
```

*Body of the loop will not be executed (condition is false) but **else clause** will*

The above code will print the result of execution of while loop's else clause, *i.e.*,

```
exiting from while loop
```

Consider another loop – this time a for loop with empty sequence.

```
for a in range(10,1):
    print ("in the loop")
else:
    print ("exiting from for loop")
```

*Body of the loop will not be executed (empty sequence) but **else clause** will*

The above code will print the result of execution of for loop's else clause, *i.e.*,

```
exiting from for loop
```

NOTE

The else clause of Python loops works in the same manner. That is, it gets executed when the loop is terminating normally – after the last element of sequence in for loop and when the test-condition becomes *false* in while loop.

4.8.6 Nested Loops

A loop may contain another loop in its body. This form of a loop is called **nested loop**. But in a nested loop, the inner loop must terminate before the outer loop. The following is an example of a nested loop, where a **for loop** is nested within another **for loop**.

```
for i in range(1, 6) :
    for j in range (1, i ) :
        print ("*" , end = ' ')
    print ()
```

end = ' ' at the end to cause printing at same line

```
# to cause printing from next line.
```

The output produced by above code is :

```
*
* *
* * *
* * * *
```

The inner for loop is executed for each value of *i*. The variable *i* takes values 1, 2, 3 and 4. The inner loop is executed once for $i = 1$ according to sequence in inner loop *range (1, i)* (because for *i* as 1, there is just one element 1 in *range (1, 1)* thus *j* iterates for one value, *i.e., 1*), twice for $i = 2$ (two elements in sequence *range(1, i)*, thrice for $i = 3$ and four times for $i = 4$.

While working with nested loops, you need to understand one thing and that is, the value of outer loop variable will change only after the inner loop is completely finished (or interrupted).

To understand this, consider the following code fragment :

```
for outer in range (5, 10, 4) :
    for inner in range(1, outer, 2) :
        print (outer, inner)
```

The above code will produce the output as :

```
5 1
5 3
```
The output produced when the outer =5

```
9 1
9 3
9 5
9 7
```
The output produced when the outer = 9

See the explanation below :

change in outer's value → 5, 5 ← 1, 3 *inner loop got over after value 3's iteration*

change in outer's value → 9, 9, 9, 9 ← 1, 3, 5, 7 *inner loop got over after value 7's iteration*

Check Point 4.3

1. What are iteration statements ? Name the iteration statements provided by Python.
2. What are the two categories of Python loops ?
3. What is the use of range() function ? What would range(3, 13) return ?
4. What is the output of the following code fragment ?
```
for a in "abcde" :
    print (a, ' +', end = ' ')
```
5. What is the output of the following code fragment ?
```
for i in range(0, 10) :
    pass
print (i)
```
6. Why does "Hello" not print even once ?
```
for i in range(10,1) :
    print ("Hello")
```
7. What is the output of following code ?
```
for a in range (2, 7) :
    for b in range(1, a) :
        print (' #', end = ' ')
    print ( )
```

Let us understand how the control moves in a nested loop with the help of one more example :

```
for a in range(3) :
    for b in range(5,7) :
        print ("for a =", a, "b now is", b)
```

The above nested loop will produce following output :

```
for a = 0   b now is 5
for a = 0   b now is 6
for a = 1   b now is 5
for a = 1   b now is 6
for a = 2   b now is 5
for a = 2   b now is 6
```

Inner loop iterations

Outer loop iterations

Notice, for each outer loop iteration, inner loop iterates twice

For the outer loop, firstly a takes the value as 0 and for $a = 0$, inner loop iterates as it is part of outer for's loop-body.

- for $a = 0$, the inner loop will iterate twice for values $b = 5$ and $b = 6$
- hence the first two lines of output given above are produced for the first iteration of outer loop ($a = 0$; which involves two iterations of inner loop)
- when a takes the next value in sequence, the inner will again iterate two times with values 5 and 6 for b.

Thus we see, that for each iteration of outer loop, inner loop iterates twice for values $b = 5$ and $b = 6$.

Check Point

4.3 (contd...)

8. Write a for loop that displays the even numbers from 51 to 60.
9. Suggest a situation where an empty loop is useful.
10. What is the similarity and difference between for and while loop ?
11. Why is while loop called an entry controlled loop ?
12. If the test-expression of a while loop evaluates to false in the beginning of the loop, even before entering in to the loop :
    (a) how many times is the loop executed?
    (b) how many times is the loop-else clause executed ?
13. What will be the output of following code:

```
while (6 + 2 > 8) :
    print ("Gotcha!")
else:
    print ("Going out!")
```

14. What is the output produced by following loop ?

```
for a in (2, 7) :
    for b in (1, a) :
        print (b, end = ' ')
        print ()
```

15. What is the significance of break and continue statements?
16. Can a single break statement in an inner loop terminate all the nested loops ?

Consider some more examples and try understanding their functioning based on above lines.

```
for a in range(3) :
    for b in range(5,7) :
        print ("* ", end = ' ')
    print ()
```

The output produced would be :

```
* *
* *
* *
```

Consider another nested loop :

```
for a in range(3) :
    for b in range(5,8) :
        print ("* ", end = ' ')
    print ()
```

notice end = ' ' at the end of print

The output produced would be :

```
* * *
* * *
* * *
```

The break Statement in a Nested Loop

If the break statement appears in a nested loop, then it will terminate the very loop it is in. That is, if the **break** statement is inside the inner loop then it will terminate the inner loop only and the outer loop will continue as it as. If the break statement is in outer loop, then outer loop gets terminated. Since inner loop is part of outer loop's body it will also not execute just like other statement of outer loop's body.

Following program illustrates this. Notice that break will terminate the inner loop only while the outer loop will progress as per its routine.

Program 4.20 *Program that searches for prime numbers from 15 through 25.*

```
for num in range(15,25):
    for i in range(2,num):
        if num % i == 0:                    #to determine factors
            j = num/i
            print ("Found a factor( ", i ," ) for", num)
            break                           # need not continue - factor found
    else:                                   # else part of the inner loop
        print (num, "is a prime number")
```

This break will terminate the inner for loop only

The output produced by above program will be :

```
Found a factor( 3 ) for 15
Found a factor( 2 ) for 16
17 is a prime number
Found a factor( 2 ) for 18
19 is a prime number
Found a factor( 2 ) for 20
Found a factor( 3 ) for 21
Found a factor( 2 ) for 22
23 is a prime number
Found a factor( 2 ) for 24
```

More examples of nested loops and other simple loops you'll find in solved problems given at the end of this chapter.

This brings us to the end of this chapter. Let us quickly revise what we have learnt so far.

PYTHON LOOPS — Progress In Python 4.2

This 'Progress in Python' session works on the objective of Python loops practice.

Carefully read the instruction and then carry out the problem. Once through with conceptual practice questions, you can move on to practicing programming problems.

:

:

Please check the practical component-book – Progress in Computer Science with Python and fill it there in PriP 4.2 under Chapter 4 after practically doing it on the computer.

>>>◆<<<

LET US REVISE

- *Statements are the instructions given to the computer to perform any kind of action.*
- *Python statements can be on one of these types : empty statement, single statement and compound statement.*
- *An empty statement is the statement that does nothing. Python offers pass statement as empty statement.*
- *Single executable statement forms a simple statement.*
- *A compound statement represents a group of statements executed as a unit.*
- *Every compound statement of Python has a header and an indented body below the header.*
- *Some examples of compound statements are : if statement, while statement etc.*
- *The flow of control in a program can be in three ways : sequentially (the sequence construct), selectively (the selection construct), and iteratively (the iteration construct).*
- *The sequence construct means statements get executed sequentially.*
- *The selection construct means the execution of statement(s) depending upon a condition-test.*
- *The iteration (looping) constructs mean repetition of a set-of-statements depending upon a condition-test.*
- *Python provides one selection statement if in many forms – if..else and if..elif..else.*
- *The if..else statement tests an expression and depending upon its truth value one of the two sets-of-action is executed.*
- *The if-else statement can be nested also i.e., an if statement can have another if statement.*
- *A sequence is a succession of values bound together by a single name. Some Python sequences are strings, lists and tuples.*
- *The range( ) function generates a sequence of list type.*
- *The statements that allow a set of instructions to be performed repeatedly are iteration statements.*
- *Python provides two looping constructs – for and while. The for is a counting loop and while is a conditional loop.*
- *The while loop is an entry-controlled loop as it has a control over entry in the loop in the form of test condition.*
- *Loops in Python can have else clause too. The else clause of a loop is executed in the end of the loop only when loop terminates normally.*
- *The break statement can terminate a loop immediately and the control passes over to the statement following the statement containing break.*
- *In nested loops, a break terminates the very loop it appears in.*
- *The continue statement abandons the current iteration of the loop by skipping over the rest of the statements in the loop-body. It immediately transfers control to the beginning of the next iteration of the loop.*

Solved Problems

1. *What is a statement ? What is the significance of an empty statement ?*

   Solution. A statement is an instruction given to the computer to perform any kind of action.

   An empty statement is useful in situations where the code requires a statement but logic does not. To fill these two requirements simultaneously, empty statement is used.

   Python offers pass statement as an empty statement.

2. *If you are asked to label the Python loops as determinable or non-determinable, which label would you give to which loop ? Justify your answer.*

Solution. The 'for loop' can be labelled as ***determinable loop*** as number of its iterations can be determined before-hand as the size of the sequence, it is operating upon.

The 'while loop' can be labelled as ***non-determinable loop***, as its number of iterations cannot be determined before-hand. Its iterations depend upon the result of a test-condition, which cannot be determined before-hand.

3. *There are two types of else clauses in Python. What are these two types of else clause ?*

Solution. The two types of Python else clauses are :

(*a*) else in an if statement (*b*) else in a loop statement

The *else* clause of an *if* statement is executed when the condition of the if statement results into *false*.

The *else* clause of a *loop* is executed when the loop is terminating normally *i.e.*, when its test-condition has gone *false* for a *while* loop or when the *for* loop has executed for the last value in sequence.

4. *Use the Python range( ) function to create the following list :* [7, 3, −1, −5]

Solution.

```
range (7, -6, -4)
```

5. *Identify and correct the problem with following code :*

```
countdown = 10              # Count from 10 down to 1
while countdown > 0 :
    print (countdown, end = ' ')
    countdown - 1
print ("Finally.")
```

Solution. *It is an infinite loop.* The reason being, in the above *while* loop, there is an expression **countdown – 1**, which subtracts 1 from value of **countdown** BUT this statement is not updating the loop variable **countdown**, hence loop variable **countdown** always remains unchanged and hence it is an **infinite loop**.

The solution to above problem will be replacement of **countdown – 1** with following statement in the body of while loop :

```
countdown = countdown - 1
```

Now the while loop's variable will be updated in every iteration and hence loop will be a finite loop.

6. *Following code is meant to be an interactive grade calculation script for an entrance test that converts from a percentage into a letter grade :*

```
90 and above is A+,
80-90 is A,
60-80 is A-,
and everything below is fail (U know high standards).
```

The program should prompt the user 100 times for grades. Unfortunately, the program was written by a terrible coder (me :/), so there are numerous bugs in the code. Find all the bugs and fix them.

```
For i : range(1 .. 100)
    grade == float (input( "What\'s the grade percentage ?" ))
    if grade > 90
        print ("That's an A + !")
```

```
if 80 > grade > 90:
    print (" " "An A is really good!" " ")
elsif 60 < grade :
    print (You got an A-!)
else grade < 60 :
    print ("Sorry, you need an A- to qualify!")
```

Solution. The corrected code is given below, with corrections marked in colour.

```
for i in range (0, 100) :
    grade = float (input( "What\'s the grade percentage ?" ))
    if grade >= 90 :
        print ("That's an A + ! ")
    if 80 <= grade < 90:
       print ("An A is really good!")        # although triple-quoted string is also OK
    elif 60 <= grade :
        print ("You got an A-!")
    else grade < 60:
        print ("Sorry, you need an A- to qualify!")
```

7. *What is the output of following code ?*

```
if (4 + 5 == 10) :
    print ("TRUE")
else :
    print ("FALSE")
print ("TRUE")
```

Solution.

```
FALSE
TRUE
```

8. *Following code contains an endless loop. Could you find out why ? Suggest a solution.*

```
n = 10
answer = 1
while ( n > 0 ) :
    answer = answer + n ** 2
    n = n + 1
print (answer)
```

Solution. For a while loop to terminate, it is necessary that its condition becomes false at one point. But in the above code, the test-condition of *while* loop will never become *false*, because the condition is $n > 0$ and n is always incrementing ; thus n will always remain > 0. Hence it will repeat endlessly.

There are *two* possible solutions :

(*i*) change the condition to some reachable limit (as per update expression) *e.g.*,

```
while (n < 100) :
```

(*ii*) change the updation equation of loop variable so that it makes the condition *false* at some point of time, *e.g.*,

```
while (n > 0 ) :
    :
    n = n - 1
```

9. *Consider below given two code fragments. What will be their outputs if the inputs are entered in the given order are (i) 'Jacob' (ii)'12345' (iii) 'END' (iv) 'end' ?*

(*a*)
```
name = " "
while True :
    name = input( "Enter name ('end' to exit):")
    if name == "end" :
        break
    print ("Hello", name)
else :
    print ("Wasn't it fun ?")
```

(*b*)
```
name = " "
while name != "end" :
    name = input( "Enter name ('end' to exit):" )
    if name == "end" :
        pass
    print ("Hello", name)
else :
    print ("Wasn't it fun ?")
```

Solution. **Code (*a*) will print**

for input (*i*)
```
Hello Jacob
```
for input (*ii*)
```
Hello 12345
```
for input (*iii*)
```
Hello END
```
for input (*iv*)
```
no output will be printed
```

Code (*b*) will print

for input (*i*)
```
Hello Jacob
```
for input (*ii*)
```
Hello 12345
```
for input (*iii*)
```
Hello END
```
for input (*iv*)
```
Hello end
Wasn't it fun ?
```

10. *Write Python code to add the odd numbers up to (and including) a given value N and print the result.*

Solution.
```
N = int (input ('Enter number'))
sum = 0
i = 1
while i <= N :
    sum = sum + i
    i = i + 2
print (sum)
```

11. *Consider the following Python program :*
```
N = int(input ( "Enter N :" ))
i = 1
sum = 0
```

```
while i < N :
    if i % 2 == 0 :
        sum = sum + i
    i = i + 1
print (sum)
```

(a) What is the output when the input value is 5 ?

(b) What is the output when the input value is 0 ?

Solution. (a) 6 (b) 0

12. *Consider the following Python program, intended to calculate factorials :*

```
number = int(input ("Enter number"))
n, result = number, 1
while True or n :
    result = result * n
    n = n – 1
factorial = result
print ("factorial of ", number, "is", factorial)
```

(a) What happens when the user enters a value of 5 ?

(b) How would you fix this program ?

Solution. The problem is that the program will repeat infinitely.

The problem lies with the condition of while loop this condition will never go false.

Correct condition will be any of these :

```
while n :
    :
```

or

```
while n > 0 :
:
```

13. *Write a program to find lowest and second lowest number from the 10 numbers input.*

Solution.

```
# to find lowest and second lowest integer from 10 integers
small = smaller = 0
for i in range(10):
  n = int(input("Enter number :"))
  if i == 0 :                          # first number read
      small = n
  elif i == 1 :                        # second number read
      if n <= small :
          smaller = n
      else:
          smaller = small
          small = n
  else:                                # for every integer read 3rd integer onwards
```

```
        if n < smaller :
            small = smaller
            smaller = n
        elif n < small :
            small = n

print("The lowest number is : ", smaller)
print("The second lowest number is : ", small)
```

Sample Run

```
The lowest number is : -6
The second lowest number is : -4
```

The sample run of above program (from the ten input numbers as : –6, 13, 20, –3, 15, 18, 99, –4, 11, 23) has been given above.

14. *Number Guessing Game – This program generates a random number between 1 and 100 and continuously asks the user to guess the number, giving hints along the way.*

Solution.

```
import random
secretNum = random.randint(1,100)          #generate a random integer between [1 100]
guessNumString = int(input ("Guess a number between 1 and 100 inclusive:" ))
while guessNum != secretNum :
  if guessNum < secretNum :
      print ("Your guess is too low.")
  else:
      print ("Your guess is too high.")
  guessNum = int(input ( "Guess a number between 1 and 100 inclusive:" ))
print ("Congratulations! You guessed the number correctly.")
```

15. *Write a Python script to print Fibonacci series' first 20 elements. Some initial elements of a Fibonacci series are:*

0 1 1 2 3 5 8...

Solution.

```
first = 0
second = 1
print (first)
print (second)
for a in range(1, 19) :
    third = first + second
    print (third)
    first, second = second, third
```

16. *Write a Python script to read an integer > 1000 and reverse the number.*

Solution.

```
num = int(input( "Enter an integer (>1000) :"))
tnum = num
reverse = 0
while tnum :
    digit = tnum % 10
    tnum = tnum / 10
    reverse = reverse * 10 + digit
print ("Reverse of", num , "is", reverse)
```

17. *Write a Python script to generate divisors of a number.*

Solution.

```
num = int(input("Enter an integer :" ))
mid = num /2
print ("The divisors of" num, "are :")
for a in range (2, mid + 1) :
    if num % a == 0 :
        print (a, end = ' ')
else :
    print ("-End-")
```

18. *Input three angles and determine if they form a triangle or not.*

Solution.

```
#check triangle from angles
angle1 = angle2 = angle3 = 0
angle1 = float(input("Enter angle 1 : "))
angle2 = float(input("Enter angle 2 : "))
angle3 = float(input("Enter angle 3 : "))
if (angle1 + angle2 + angle3) == 180 :
    print("These angles form a triangle.")
else:
    print("These angles do not form a triangle.")
```

19. *Numbers in the form $2^n - 1$ are called Mersenne Numbers, e.g., $2^1 - 1 = 1$, $2^2 - 1 = 3$, $2^3 - 1 = 7$. Write a Python script that displays first ten Mersenne numbers.*

Solution.

```
#program to print mersenne numbers
print("First 10 Mersenne numbers are:")
for a in range(1, 11):
    mersnum = 2 ** a - 1
    print(mersnum, end = " ")
print()
```

Sample Run

```
First 10 Mersenne numbers are :
1 3 7 15 31 63 127 255 511 1023
```

20. *Mersenne numbers that are prime numbers also, are called Mersenne Prime numbers. Make modifications in previous question's Python script so that it also displays 'Prime' next to the Mersenne Prime numbers. Also, this time print 20 Mersenne numbers.*

Solution.

```
# program to print mersenne prime numbers
for a in range(1, 21):
    mersnum = 2 ** a - 1
    mid = int(mersnum / 2) + 1
    for b in range(2, mid):
        if mersnum % b == 0 :
            print(mersnum)
            break
    else:
        print(mersnum, "\tPrime")
```

Sample Run

(With first 10 Mersenne numbers)

```
1       Prime
3       Prime
7       Prime
15
31      Prime
63
127     Prime
255
511
1023
```

21. *Write a program to calculate BMI of a person after inputting its weight in kgs and height in meters and then print the Nutritional Status as per following table :*

Nutritional Status	WHO criteria BMI cut-off
Underweight	< 18.5
Normal	18.5 – 24.9
Overweight	25 – 29.9
Obese	≥ 30

Solution.

```
weight_in_kg = float(input("Enter weight in kg :"))
height_in_meter = float(input("Enter height in meters :"))
bmi = weight_in_kg / ( height_in_meter * height_in_meter)
print("BMI is : ", bmi, end = " ")
if bmi < 18.5 :
    print("... Underweight")
elif bmi < 25 :
    print("... Normal")
elif bmi < 30 :
    print("... Overweight")
else:
    print("... Obese")
```

22. *Write Python script to print the following pattern :*

```
1
1 3
1 3 5
1 3 5 7
```

Solution.

```
for a in range(3, 10, 2) :
    for b in range(1, a, 2) :
        print (b, end = ' ')
    print ( )
```

23. *Write a program to find sum of the series :* $s = 1 + x + x^2 + x^3 ... + x^n$

Solution.

```
x = float ( input ( "Enter value of x :" ) )
n = int (input( "Enter value of n (for x ** n) :" ))
s = 0
for a in range(n + 1) :
    s += x ** a
print ("Sum of first" , n , "terms :", s)
```

24. *Write a Python script to input two numbers and print their lcm (Least common multiple) and gcd (greatest common divisor).*

Solution.

```
x = int(input("Enter first number: "))
y = int(input("Enter second number: "))
```

```
if x > y:
   smaller = y
else:
   smaller = x
for i in range(1, smaller + 1):
   if((x % i == 0) and (y % i == 0)):
     hcf = i
lcm = (x * y ) / hcf
print("The H.C.F. of", x, "and", y, "is", hcf)
print("The L.C.M. of", x, "and", y, "is", lcm)
```

25. *Write a Python script to calculate sum of following series :*

$$s = (1) + (1+2) + (1+2+3) + ... + (1+2+3+...+n)$$

Solution.

```
sum = 0
n = int ( input ( "How many terms ?" ) )
for a in range(2, n + 2) :              # added 2 to n because started with 2
    term = 0
    for b in range(1, a) :
        term += b
    print ("Term" , (a - 1), ":", term)
    sum += term
print ("Sum of", n, "terms is" , sum)
```

26 *Write a program to print the following using a single loop (no nested loops)*

```
1
11
111
1111
11111
```

Solution.

```
n = 1
for a in range(5) :
    print (n)
    n = n * 10 + 1
```

27. *Write a program to print a pattern like :*

```
4321
432
43
4
```

Solution.

```
for i in range(4):
    for j in range(4, i, -1):
        print(j, end =" ")
    else:
        print()
```

GLOSSARY

Block	A group of consecutive statements having same indentation level.
Body	The block of statements in a compound statement that follows the header.
Empty statement	A statement that appears in the code but does nothing.
Infinite Loop	A loop that never ends. *Endless loop.*
Iteration Statement	Statement that allows a set of instructions to be performed repeatedly.
Jump Statement	Statement that unconditionally transfers program control within a function.
Looping Statement	Iteration statement. Also called a loop.
Nesting	Same program-construct within another of same type of construct.
Nested Loop	A loop that contains another loop inside its body.
Suite	Block.

Assignments

Type A : Short Answer Questions/Conceptual Questions

1. What is the common structure of Python compound statements ?
2. What is the importance of the three programming constructs ?
3. What is empty statement ? What is its need ?
4. Which Python statement can be termed as empty statement ?
5. What is an algorithm ?
6. What is a flowchart ? How is it useful ?
7. What is pseudocode ? How is it useful in developing logic for the solution of a problem ?
8. What is decision tree ? Where are these useful ?
9. Write pseudocode for checking if a number is a factor of 3215 ?
10. Draw flowchart for displaying first 10 odd numbers.
11. Draw decision tree for grade calculation as per :

Marks	> 90	80-90	70-80	60-70	50-60	40-50	< 40
Grade	A1	A2	B1	B2	C1	C2	F

12. Write pseudocode for comparing two products of two sets of numbers (a, b and c, d) and displaying if both products are same or not.
13. What is entry-controlled loop ?
14. What are the four elements of a while loop in Python ?
15. What is the difference between else clause of if-else and else clause of Python loops ?
16. In which cases, the else clause of a loop does not get executed ?
17. What are jump statements ? Name them.
18. How and when are named conditions useful ?
19. What are endless loops ? Why do such loops occur ?

Type B : Application Based Questions

1. Rewrite the following code fragment that saves on the number of comparisons :

```
if (a == 0) :
  print ("Zero")
if (a == 1) :
  print ("One")
if (a == 2) :
  print ("Two")
if (a == 3) :
  print ("Three")
```

2. Under what conditions will this code fragment print "water" ?

```
if temp < 32 :
  print ("ice")
elif temp < 212:
  print ("water")
else :
  print ("steam")
```

3. What is the output produced by the following code ?

```
x = 1
if x > 3 :
  if x > 4 :
      print ("A", end = ' ')
  else :
      print ("B", end = ' ')
elif x < 2:
  if (x != 0):
      print ("C", end = ' ')
print ("D")
```

4. What is the error in following code ? Correct the code :

```
weather = 'raining'
if weather = 'sunny' :
  print ("wear sunblock")
elif weather = "snow":
  print ("going skiing")
else :
  print (weather)
```

5. What is the output of the following lines of code ?

```
if int('zero') == 0 :
  print ("zero" )
elif str(0) == 'zero' :
  print (0)
elif str(0) == '0 ' :
  print (str(0))
else:
  print ("none of the above")
```

6. Find the errors in the code given below and correct the code :

```
if n == 0
  print ("zero")
```

```
elif : n == 1
  print ("one")
elif
  n == 2:
  print ("two")
else n == 3:
  print ("three")
```

7. What is following code doing ? What would it print for input as 3 ?

```
n = int (input( "Enter an integer:" ))
if n < 1 :
  print ("invalid value")
else :
  for i in range(1, n + 1):
      print (i * i)
```

8. How are following two code fragments different from one another ? Also, predict the output of the following code fragments :

(a)
```
n = int(input( "Enter an integer:" ))
if n > 0 :
    for a in range(1, n + n ) :
        print (a / (n/2))
    else :
        print ("Now quiting")
```

(b)
```
n = int (input( "Enter an integer:"))
if n > 0 :
    for a in range(1, n + n) :
        print (a / (n/2))
    else :
        print ("Now quiting")
```

9. Rewrite the following code fragments using for loop :

(a)
```
i = 100
while (i > 0) :
    print (i)
    i -= 3
```

(b)
```
while num > 0 :
        print (num % 10)
        num = num/10
```

(c)
```
while num > 0 :
    count += 1
    sum += num
        num -= 2
        if count == 10 :
            print (sum/float(count))
            break
```

10. Rewrite following code fragments using while loops :

(a)
```
min = 0
max = num
if num < 0 :
    min = num
    max = 0          # compute sum of integers from min to max
    for i in range(min, max + 1):
        sum += i
```

(b)
```
for i in range(1, 16) :
    if i % 3 == 0 :
        print (i)
```

(c)
```
for i in range(4) :
    for j in range(5):
        if i + 1 == j or j + i == 4 :
            print ("+", end = ' ')
    else :
        print ("o", end = ' ')
print ()
```

11. Predict the output of the following code fragments :

(a)
```
count = 0
while count < 10:
    print ("Hello")
    count += 1
```

(b)
```
x = 10
y = 0
    while x > y:
        print (x, y)
        x = x - 1
        y = y + 1
```

(c)
```
keepgoing = True
x = 100
while keepgoing :
    print (x)
    x = x - 10
    if x < 50 :
        keepgoing = False
```

(d)
```
x = 45
while x < 50 :
    print (x)
```

(e)
```
for x in [1,2,3,4,5]:
        print (x)
```

(f)
```
for x in range(5):
    print (x)
```

(g)
```
for p in range(1,10):
    print (p)
```

(h)
```
for q in range(100,50,-10):
    print (q)
```

(i)
```
for z in range(-500, 500, 100):
    print (z)
```

(j)
```
for y in range(500,100,100):
    print (" * ", y)
```

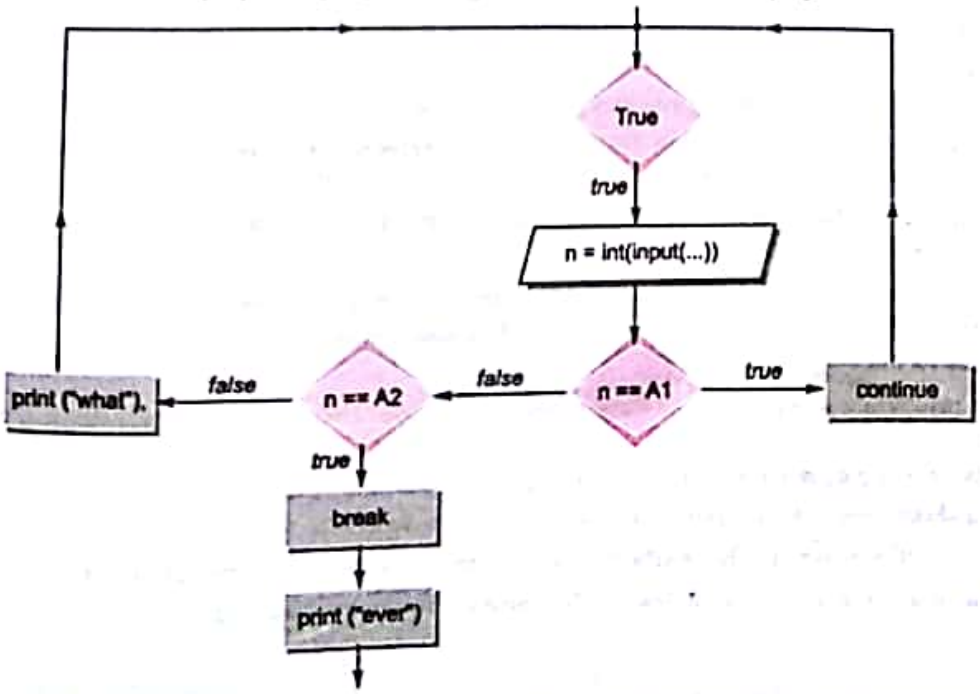
(k)
```
x = 10
y = 5
for i in range(x-y * 2):
    print (" % ", i)
```

(l)
```
for x in [1,2,3]:
    for y in [4, 5, 6]:
        print (x, y)
```

(m)
```
for x in range(3):
    for y in range(4):
        print (x, y, x + y)
```

(n)
```
c = 0
for x in range(10):
    for y in range(5):
        c += 1
print (c)
```

12. Which of the following Python programs implement the control flow graph shown ?



(a)
```
while True :
    n = int(input("Enter an int:"))
    if n == A1 :
        continue
    elif n == A2 :
        break
    else :
        print ("what")
else :
    print ("ever")
```

(b)
```
while True :
    n = int(input("Enter an int: "))
    if n == A1 :
        continue
    elif n == A2 :
        break
    else :
        print ("what")
print ("ever")
```

(c)
```
while True :
    n = int(input("Enter an int: "))
    if n == A1 :
        continue
    elif n == A2 :
        break
print ("what")
print ("ever")
```

13. What is the output of the following code ?

```
for i in range(4):
    for j in range(5):
        if i + 1 == j or j + i == 4:
            print ("+", end = ' ')
        else:
            print ("o", end = ' ')
    print()
```

14. In the nested for loop code above (Q. 13), how many times is the condition of the if clause evaluated ?

Type C : Programming Practice/Knowledge based Questions

1. Write a Python script that asks the user to enter a length in centimetres. If the user enters a negative length, the program should tell the user that the entry is invalid. Otherwise, the program should convert the length to inches and print out the result. There are 2.54 centimetres in an inch.
2. A store charges ₹ 120 per item if you buy less than 10 items. If you buy between 10 and 99 items, the cost is ₹ 100 per item. If you buy 100 or more items, the cost is ₹ 70 per item. Write a program that asks the user how many items they are buying and prints the total cost.
3. Write a program that asks the user for two numbers and prints **Close** if the numbers are within .001 of each other and **Not close** otherwise.
4. A year is a leap year if it is divisible by 4, except that years divisible by 100 are not leap years unless they are also divisible by 400. Write a program that asks the user for a year and prints out whether it is a leap year or not.
5. Write a program to input length of three sides of a triangle. Then check if these sides will form a triangle or not.
   (Rule is : a + b > c ; b + c > a ; c + a > b)
6. Write a short program to input a digit and print it in words.
7. Write a short program to check whether square root of a number is prime or not.
8. Write a short program to print first *n* odd numbers in descending order.

9. Write a short program to print the following series :
   (*i*) 1 4 7 10 . . . . . 40.
   (*ii*) 1 – 4 7 – 10 . . . . . – 40
10. Write a short program to find average of list of numbers entered through keyboard.
11. Write a program to input 3 sides of a triangle and print whether it is an equilateral, scalene or isosceles triangle.
12. Write a program to take an integer a as an input and check whether it ends with *4* or *8*. If it ends with 4, print "ends with 4", if it ends with 8, print "ends with 8", otherwise print "ends with neither".
13. Write a program to take N ($N > 20$) as an input from the user. Print numbers from 11 to N. When the number is a multiple of 3, print "Tipsy", when it is a multiple of 7, print "Topsy". When it is a multiple of both, print "TipsyTopsy".
14. Write a short program to find largest number of a list of numbers entered through keyboard.
15. Write a program to input N numbers and then print the second largest number.
16. Given a list of integers, write a program to find those which are palindromes. For example, the number 4321234 is a palindrome as it reads the same from left to right and from right to left.
17. Write a complete Python program to do the following :
   (*i*) read an integer X.
   (*ii*) determine the number of digits n in X.
   (*iii*) form an integer Y that has the number of digits n at ten's place and the most significant digit of X at one's place.
   (*iv*) Output Y.

   (For example, if X is equal to 2134, then Y should be 42 as there are 4 digits and the most significant number is 2).
18. Write a Python program to print every integer between 1 and n divisible by m. Also report whether the number that is divisible by m is even or odd.
19. Write Python programs to sum the given sequences :

   (*a*) $\frac{2}{9} - \frac{5}{13} + \frac{8}{17}$..... (print 7 terms) (*b*) $1^2 + 3^2 + 5^2 + ... + n^2$ (Input n)
20. Write a Python program to sum the sequence

$$1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + ... + \frac{1}{n!}$$ (Input n)

21. Write a program to accept the age of n employees and count the number of persons in the following age group :
   (*i*) 26 – 35 (*ii*) 36 – 45 (*iii*) 46 – 55.
22. Write programs to find the sum of the following series :

   (*a*) $x - \frac{x^2}{2!} + \frac{x^3}{3!} - \frac{x^4}{4!} + \frac{x^5}{5!} - \frac{x^6}{6!}$ (Input x) (*b*) $x + \frac{x^2}{2} + \frac{x^3}{3} + ... + \frac{x^n}{n}$ (Input x and n both)
23. Write programs to print the following shapes :

   (*a*) (*b*) (*c*) (*d*)

24. Write programs using nested loops to produce the following patterns :

(a)
```
A
A B
A B C
A B C D
A B C D E
A B C D E F
```

(b)
```
0
2 2
4 4 4
8 8 8 8
```

25. Write a program using nested loops to produce a rectangle of *'s with 6 rows and 20 *'s per row.
26. Given three numbers *A*, *B* and *C*, write a program to write their values in an ascending order. For example, if *A* = 12, *B* = 10, and *C* = 15, your program should print out :

```
Smallest number = 10
Next higher number = 12
Highest number = 15
```

27. Write a Python script to input temperature. Then ask them what units, Celsius or Fahrenheit, the temperature is in. Your program should convert the temperature to the other unit. The conversions are F = 9/5C + 32 and *C* = 5/9 (F 32).
28. Ask the user to enter a temperature in Celsius. The program should print a message based on the temperature :
    - If the temperature is less than −273.15, print that the temperature is invalid because it is below absolute zero.
    - If it is exactly − 273.15, print that the temperature is absolute 0.
    - If the temperature is between −273.15 and 0, print that the temperature is below freezing.
    - If it is 0, print that the temperature is at the freezing point.
    - If it is between 0 and 100, print that the temperature is in the normal range.
    - If it is 100, print that the temperature is at the boiling point.
    - If it is above 100, print that the temperature is above the boiling point.

5 String Manipulation

In This Chapter

5.1 INTRODUCTION

You all have basic knowledge about Python strings. You know that Python strings are characters enclosed in quotes of any type – *single quotation marks, double quotation marks* and *triple quotation marks*. You have also learnt things like – an empty string is a string that has 0 characters (*i.e.*, it is just a pair of quotation marks) and that Python strings are immutable. You have used strings in earlier chapters to store text type of data.

You know by now that strings are sequence of characters, where each character has a unique position-id/index. The indexes of a string begin from *0 to (length – 1)* in forward direction and –1, –2, –3, ..., – length in backward direction.

In this chapter, you are going to learn about many more string manipulation techniques offered by Python like operators, methods etc.

5.2 TRAVERSING A STRING

You know that individual characters of a string are accessible through the unique index of each character. Using the indexes, you can traverse a string character by character. Traversing refers to iterating through the elements of a string, one character at a time. You have already traversed through strings, though unknowingly, when we talked about sequences along with *for loops*. To traverse through a string, you can write a loop like :

```
name = "superb"
for ch in name :
    print (ch, '-', end = ' ')
```

This loop will traverse through string name character by character.

NOTE

Traversing refers to iterating through the elements of a string, one character at a time.

The above code will print :

```
s-u-p-e-r-b-
```

The information that you have learnt till now is sufficient to create wonderful programs to manipulate strings. Consider the following programs that use the Python string indexing to display strings in multiple ways.

Program 5.1 *Program to read a string and display it in reverse order - display one character per line. Do not create a reverse string, just display in reverse order.*

```
string1 = input( "Enter a string :" )
print ("The", string1, "in reverse order is:")
length = len(string1)
for a in range(-1, (-length-1), -1) :
    print (string1[a] )
```

Since the range() excludes the number mentioned as upper limit, we have taken care of that by increasing the limit accordingly.

Sample run of above program is :

```
Enter a string : python
The python in reverse order is:
n
o
h
t
y
p
```

Program 5.2 *Program to read a string and display it in the form :*

first character *last character*
second character *second last character*
⋮ ⋮

For example, string "try" should print as :

```
t   y
r   r
y   t
```

```
string1 = input( "Enter a string :" )
length = len(string1)
i = 0
for a in range(-1, (-length-1), -1) :
    print (string1[i], "\t", string1[a])
    i += 1
```

NOTE

When you give a negative index, Python adds length of string to get its forward index *e.g.*, for a 5-lettered string S, S[-1] will give *S* [-1 + 5] *i.e.*, *S* [4] letter ; for S[-5], it will give you *S* [-5 + 5] *i.e.*, *S* [0]

Sample run of above program is :

```
Enter a string : python
p       n
y       o
t       h
h       t
o       y
n       p
```

5.3 STRING OPERATORS

In this section, you'll be learning to work with various operators that can be used to manipulate strings in multiple ways. We'll be talking about basic operators + and *, membership operators **in** and **not in** and comparison operators (all relational operators) for strings.

5.3.1 Basic Operators

The two basic operators of strings are : + and *. You have used these operators as *arithmetic operators* before for *addition* and *multiplication* respectively. But when used with strings, + operator performs ***concatenation*** rather than addition and * operator performs ***replication*** rather than multiplication. Let us see, how.

Also, before we proceed, recall that strings are immutable *i.e.*, un-modifiable. Thus every time you perform something on a string that changes it, Python will internally create a new string rather than modifying the old string in place.

String Concatenation Operator +

The + operator creates a new string by joining the two operand strings, *e.g.*,

```
"tea" + "pot"
```

will result into

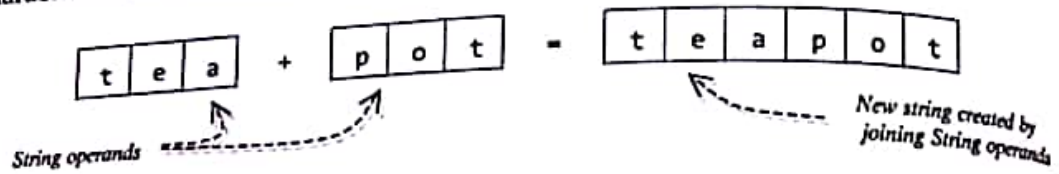
```
'teapot'
```

Two input strings joined (concatenated) to form a new string

Consider some more examples :

Expression	will result into
'1' + '1'	'11'
"a" + "0"	'a0'
'123' + 'abc'	'123abc'

Let us see how concatenation takes place internally. Python creates a new string in the memory by storing the individual characters of first string operand followed by the individual characters of second string operand. (see below)



Original strings are not modified as strings are immutable ; new strings can be created but existing strings cannot be modified.

Caution!

Another important thing that you need to know about + operator is that this operator can work with *numbers* and *strings* separately for *addition* and *concatenation* respectively, but in the same expression, you cannot combine *numbers* and *strings* as operands with a + operator.

For example,

```
2 + 3 = 5              # addition – VALID
'2' + '3' = '23'       # concatenation – VALID
```

But the expression

```
'2' + 3
```

is invalid. It will produce an error like :

```
>>> '2' + 3
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    '2' + 3
TypeError: cannot concatenate 'str' and 'int' objects
```

NOTE

The + operator has to have both operands of the same type either of number types (for addition) or of string types (for multiplication). It cannot work with one operand as string and one as a number.

Thus we can summarize + operator as follows :

Table 5.1 *Working of Python + operator*

Operands' data type	Operation performed by +	Example
numbers	addition	9 + 9 = 18
string	concatenation	"9" + "9" = "99"

String Replication Operator *

The * operator when used with numbers (*i.e.*, when both operands are numbers), it performs *multiplication* and returns the *product* of the two number operands.

To use a * operator with strings, you need two types of operands – *a string* and *a number, i.e.,* as number * string or string * number.

Where *string operand* tells the string to be replicated and *number operand* tells the number of times, it is to be repeated; Python will create a new string that is a number of repetitions of the string operand.

For example,

```
3 * "go!"
```

will return

```
'go!go!go!'
```

Input strings repeated specified number of times to form a new string.

NOTE

For replication operator *, Python creates a new string that is a number of repetitions of the input string.

Consider some more examples :

Expression	will result into
"abc" * 2	"abcabc"
5 * "@"	"@@@@@"
":-" * 4	":-:-:-:-"
"1" * 2	"11"

Caution!

Another important thing that you need to know about * operator is that this operator can work with numbers as both operands for *multiplication* and with a string and a number for *replication* respectively, but in the same expression, you cannot have *strings* as both the operands with a * operator.

For example,

```
2 * 3 = 6              # multiplication - VALID
"2" * 3 = "222"        # replication - VALID
```

But the expression

```
"2" * "3"
```

is invalid. It will produce an error like :

```
>>> "2" * "3"
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    "2" * "3"
TypeError: can't multiply sequence by non-int of type 'str'
```

NOTE

The * operator has to either have both *operands of the number types* (for multiplication) or *one string type and one number type* (for replication). It cannot work with both operands of string types.

Thus we can summarize + operator as follows :

Table 5.2 *Working of Python * operator*

Operands' data type	Operation performed by *	Example
numbers	multiplication	9 * 9 = 18
string, number	replication	"#" * 3 = "###"
number , string	replication	3 * "#" = "###"

5.3.2 Membership Operators

There are two membership operators for strings (in fact for all sequence types). These are **in** and **not in**. We have talked about them in previous chapter, briefly. Let us learn about these operators in context of strings.

Recall that :

in	Returns *True* if a character or a substring exists in the given string ; *False* otherwise
not in	Returns *True* if a character or a substring does not exist in the given string; *False* otherwise

Both membership operators (when used with strings), require that both operands used with them are of string type, *i.e.*,

```
<string> in <string>
<string> not in <string>
```

e.g.,

```
"12" in "xyz"
"12" not in "xyz"
```

Now, let's have a look at some examples :

`"a" in "heya"`	will give	True
`"jap" in "heya"`	will give	False
`"jap" in "japan"`	will give	True
`"jap" in "Japan"`	will give	False because j letter's cases are different; hence "jap" is not contained in "Japan"
`"jap" not in "Japan"`	will give	True because string "jap" is not contained in string "Japan"
`"123" not in "hello"`	will give	True because string "123" is not contained in string "hello"
`"123" not in "12345"`	will give	False because "123" is contained in string "12345"

The **in** and **not in** operators can also work with string variables. Consider this :

```
>>> sub = "help"
>>> string = 'helping hand'
>>> sub2 = 'HELP'
>>> sub in string
True
>>> sub2 in string
False
>>> sub not in string
False
>>> sub2 not in string
True
```

5.3.3 Comparison Operators

Python's standard comparison operators *i.e.*, all relational operators (<, <=, >, >=, ==, !=) apply to strings also. The comparisons using these operators are based on the standard character-by-character comparison rules for Unicode (*i.e.*, dictionary order).

Thus, you can make out that

`"a" == "a"`	will give	*True*
`"abc" == "abc"`	will give	*True*
`"a" != "abc"`	will give	*True*
`"A" != "a"`	will give	*True*
`"ABC" == "abc"`	will give	*False* (letters' case is different)
`"abc" != "Abc"`	will give	*True* (letters' case is different)

Equality and non-equality in strings are easier to determine because it goes for exact character matching for individual letters including the case (upper-case or lower-case) of the letter. But for other comparisons like *less than* (<) or *greater than* (>), you should know the following piece of useful information.

As internally Python compares using Unicode values (called ordinal value), let us know about some most common characters and their ordinal values. For most common characters, the ASCII values and Unicode values are the same.

The most common characters and their ordinal values are :

Table 5.3 *Common Characters and their Ordinal Values*

Characters	*Ordinal Values*
'0' to '9'	48 to 57
'A' to 'Z'	65 to 90
'a' to 'z'	97 to 122

Thus upper-case letters are considered smaller than the lower-case letters. For instance,

`'a' < 'A'`	will give	*False* because the Unicode value of lower-case letters is higher than upper case letters ; hence 'a' is greater than 'A', not lesser.
`'ABC' > 'AB'`	will give	*True* for obvious reasons.
`'abc' <= 'ABCD'`	will give	*False* because letters of 'abc' have higher ASCII values compared to 'ABCD'.
`'abcd' > 'abcD'`	will give	*True* because strings 'abcd' and 'abcD' are same till first three letters but the last letter of 'abcD' has lower ASCII value than last letter of string 'abcd'.

Thus, you can say that Python compares two strings through relational operators using character-by-character comparison of their Unicode values.

Determining Ordinal/Unicode Value of a Single Character

Python offers a built-in function *ord( )* that takes a single character and returns the corresponding ordinal Unicode value. It is used as per following format :

```
ord(<single-character>)
```

Let us see how, with the help of some examples :

- To know the ordinal value of letter 'A', you'll write **ord('A')** and Python will return the corresponding ordinal value (see below) :

```
>>> ord('A')
65
```

But you need to keep in mind that *ord( )* function requires single character string only. You may even write an escape sequence enclosed in quotes for *ord( )* function.

The opposite of **ord( )** function is **chr( )**, *i.e.*, while **ord( )** returns the ordinal value of a character, the **chr( )** takes the ordinal value in integer form and returns the character corresponding to that ordinal value. The general syntax of **chr( )** function is :

```
chr(<int>)          # the ordinal value is given in integer
```

Have a look at some examples (compare with the Table 5.3 given above) :

```
>>> chr(65)
'A'
>>> chr(97)
'a'
```

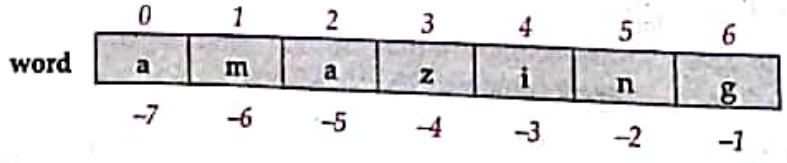
5.4 STRING SLICES

As an English term, you know the meaning of word 'slice', which means – 'a part of'. In the same way, in Python, the term *'string slice'* refers to a part of the string, where strings are sliced using a range of indices.

STRING SLICE

Part of a string containing some contiguous characters from the string.

That is, for a string say name, if we give *name[n : m]* where *n* and *m* are integers and legal indices, Python will return a slice of the string by returning the characters falling between indices *n* and *m* – starting at *n*, *n*+1, *n*+2 ... till *m*–1 . Let us understand this with the help of examples. Say we have a string namely *word* storing a string 'amazing' *i.e.*,



Then,

`word[ 0 : 7 ]` will give `'amazing'` (the letters starting from index 0 going up till 7 – 1 *i.e.*, 6 : from indices 0 to 6, both inclusive)

`word[ 0 : 3 ]` will give `'ama'` (letters from index 0 to 3 – 1 *i.e.*, 0 to 2)

`word[ 2 : 5 ]`	will give	`'azi'`	(letters from index 2 to 4 (*i.e.*, 5 – 1))
`word[-7 : -3]`	will give	`'amaz'`	(letters from indices –7, –6, –5, –4 excluding index –3)
`word[-5 : -1]`	will give	`'azin'`	(letters from indices –5, –4, –3, –2 excluding –1)

From above examples, one thing must be clear to you :

- In a string slice, the character at last index (the one following colon (:)) is not included in the result.

In a string slice, you give the slicing range in the form *[<begin-index> : <last >]*. If, however, you skip either of the ***begin-index*** or ***last***, Python will consider the limits of the string *i.e.*, for missing ***begin-index***, it will consider 0 (the first index) and for missing ***last*** value, it will consider length of the string.

Consider following examples to understand this :

`word[:7]`	will give	`'amazing'`	(missing index before colon is taken as 0 (zero))
`word[:5]`	will give	`'amazi'`	(-do-)
`word[3:]`	will give	`'zing'`	(missing index after colon is taken as 7 (the length of the string))
`word[5:]`	will give	`'ng'`	(-do-)

The string slice refers to a part of the string **s[start:end]** that is the elements beginning at **start** and extending up to but not including **end.**

Following figure (Fig. 5.1) shows some string slices :

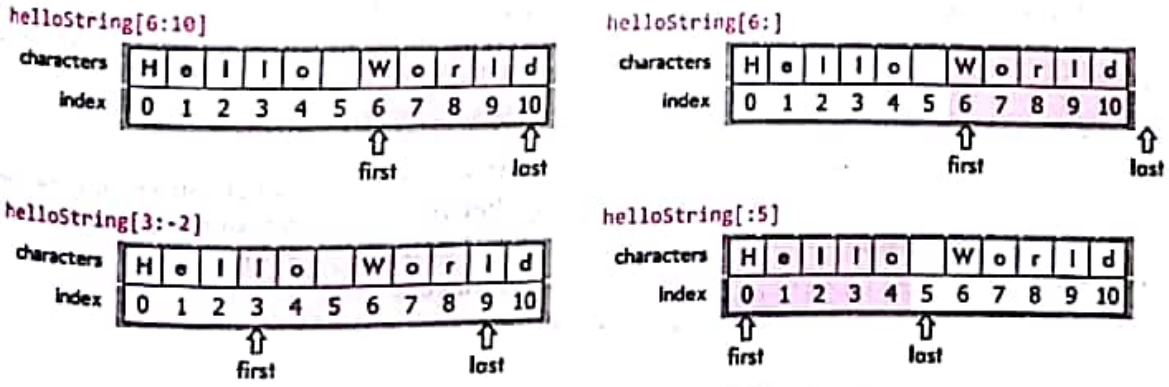


Figure 5.1 String Slicing in Python.

Interesting Inference

Using the same string slicing technique, you will find that

- for any index *n*, `s[:n] + s[n:]` will give you original string s.

This works even for *n* negative or out of bounds.

Let us prove this with an example. Consider the same string namely word storing *'amazing'*.

```
>>> word[3:], word [:3]
'zing' 'ama'
>>> word[:3] + word[3:]
'amazing'
>>> word[:-7], word [-7:]
'' 'amazing'
>>> word[:-7] + word[-7:]
'amazing'
```

TIP

String [: : –1] is an easy way to reverse a string.

You can give a third (optional) index (say *n*) in string slice too. With that every *n*th element will be taken as part of slice e.g., for word = 'amazing', look at following examples.

```
>>> word [1:6:2]
'mzn'
```
It will take every 2nd character starting from index = 1 till index < 6.

```
>>> word [-7:-3:3]
'az'
```
It will take every 3rd character starting from index = –7 to index < -3

```
>>> word [: : -2]
'giaa'
```
Every 2nd character taken backwards

```
>>> word [: : -1]
'gnizama'
```
Every character taken backwards.

Another interesting inference is :

- Index out of bounds causes error with strings but slicing a string outside the bounds does not cause error.

```
s = "Hello"
print (s[5])
```
Will cause error because 5 is invalid index-out of bounds, for string 'Hello'

But if you give

```
s = "Hello"
print (s[4 : 8])
print (s[5 : 10])
```
One limit is outside the bounds (length of Hello is 5 and thus valid indexes are 0-4)

Both limits are outside the bounds

the above will not give any error and print output as :

```
o

```
empty string

i.e., letter o followed by empty string in next line. The reason behind this is that when you use an index, you are accessing a constituent character of the string, thus the index must be valid and out of bounds index causes error as there is no character to return from the given index. But slicing always returns a subsequence and empty sequence is a valid sequence. Thus when you slice a string outside the bounds, it still can return empty subsequence and hence Python gives no error and returns empty subsequence.

Truly amazing :). Isn't it ?.

Check Point 5.1

1. How are strings internally stored ?
2. For a string s storing 'Goldy', what would s[0] and s[-1] return ?
3. The last character of a string s is at index len(s) – 1. True / False ?
4. For strings, + means (1) ; * means (2) . Suggest words for positions (1) and (2).
5. Given that

   ```
   s1 = "spam"
   s2 = "nil"
   ```

   What is the output produced by following expressions ?
   (a) " The Knights who say, " + s2
   (b) 3 * s1 + 2 * s2
   (c) s1[1]
6. What are membership operators? What do they basically do ?
7. On what principles, the strings are compared in Python ?
8. What will be the result of following expressions ?
   (a) "Wow Python" [1]
   (b) "Strings are fun." [5]
   (c) len("awesome")
   (d) "Mystery" [:4]
   (e) "apple" > "pineapple"
   (f) "pineapple" < "Peach"
   (g) "cad" in "abracadabra"
   (h) "apple" in "Pineapple"
   (i) "pine" in "Pineapple"
9. What do you understand by string slices ?
10. Considering the same strings s1 and s2 of question 5 above, evaluate the following expressions :
    (a) s1[1:3]
    (b) s1[2] + s2[:2]
    (c) s1 + s2[-1]
    (d) s1[:3] + s1[3:]
    (e) s2[:-1] + s2[-1:]

5.9 *Program that prints the following pattern without using any nested loop*

```
#
# #
# # #
# # # #
# # # # #
```

```
string = '#'
pattern = ""                    # empty string
for a in range(5) :
    pattern += string
    print (pattern)
```

Sample run of the program is as shown below :

```
#
##
###
####
#####
```

5.5 STRING FUNCTIONS AND METHODS

Python also offers many built-in functions and methods for string manipulation. You have already worked with one such method **len( )** in earlier chapters. In this chapter, you will learn about many other built-in powerful string methods of Python used for string manipulation.

Every string object that you create in Python is actually an instance of String class (you need not do anything specific for this ; Python does it for you – you know built-in). The string manipulation methods that are being discussed below can be applied to string as per following syntax :

```
<stringObject>.<method name> ()
```

In the following table we are referring to <stringObject> as *string* only (no angle brackets) but the meaning is intact *i.e.*, you have to replace *string* with a legal string (*i.e.*, either a string literal or a string variable that holds a string value).

Let us now have a look at some useful built-in string manipulation methods.

Table 5.4 *Python's Built-in String Manipulation Methods*

`string.capitalize()`	Returns a copy of the *string* with its first character capitalized.
	Example
	`>>>'tru .capitalize()`
	`'True'`
	`>>>' i love my India'.capitalize()`
	`I love my India`

`string.find(sub[, start[, end]])`	Returns the lowest index in the *string* where the substring *sub* is found within the slice range of *start* and *end*.
	Returns –1 if *sub* is not found.
	Example
	`>>> string = 'it goes as - ringa ringa roses'`
	`>>> sub = 'ringa'`
	`>>> string.find(sub)`
	`13`
	`>>> string.find(sub, 15, 22)`
	`-1`
	`>>> string.find(sub, 15, 25)`
	`19`

`string.isalnum()`	Returns **True** if the characters in the ***string*** are alphanumeric (alphabets or numbers) and there is at least one character, **False** otherwise.
	Example
	`>>> string = "abc123"`
	`>>> string2 = 'hello'`
	`>>> string3 = '12345'`
	`>>> string4 = ' '`
	`>>> string.isalnum()`
	`True`
	`>>> string2.isalnum()`
	`True`
	`>>> string3.isalnum()`
	`True`
	`>>> string4.isalnum()`
	`False`
	`>>>`

`string.isalpha()`	Returns **True** if all characters in the ***string*** are alphabetic and there is at least one character, **False** otherwise.
	Example
	(considering the same string values as used in example of previous function - isalnum)
	`>>> string.isalpha()`
	`False`
	`>>> string2.isalpha()`
	`True`
	`>>> string3.isalpha()`
	`False`
	`>>> string4.isalpha(')`
	`False`

`string.isdigit()`	Returns **True** if all the characters in the ***string*** are digits. There must be at least one character, otherwise it returns **False**.
	Example
	(considering the same string values as used in example of previous function - isalnum)
	`>>> string.isdigit()`
	`False`
	`>>> string2.isdigit()`
	`False`
	`>>> string3.isdigit()`
	`True`
	`>>> string4.isdigit()`
	`False`

`string.islower()`	Returns **True** if all cased characters in the ***string*** are lowercase. There must be at least one cased character. It returns **False** otherwise.
	
	Example
	
	`>>> string = 'hello'`
	`>>> string2 = 'THERE'`
	`>>> string3 = 'Goldy'`
	`>>> string.islower()`
	`True`
	`>>> string2.islower()`
	`False`
	`>>> string3.islower()`
	`False`

`string.isspace()`	Returns **True** if there are only whitespace characters in the ***string***. There must be at least one character. It returns **False** otherwise.
	
	Example
	
	`>>> string = "   "  # stores three spaces`
	`>>> string2 = ""  # an empty string`
	`>>> string.isspace()`
	`True`
	`>>> string2.isspace()`
	`False`

`string.isupper()`	Tests whether all cased characters in the ***string*** are uppercase and requires that there be at least one cased character. Returns **True** if so and **False** otherwise.
	
	Example
	
	`>>> string = "HELLO"`
	`>>> string2 ="There"`
	`>>> string3 = "goldy"`
	`>>> string4 = "U123"  # character in uppercase`
	`>>> string5 = "123f"  # character in lowercase`
	`>>> string.isupper()`
	`True`
	`>>> string2.isupper()`
	`False`
	`>>> string3.isupper()`
	`False`
	`>>> string4.isupper()`
	`True`
	`>>> string5.isupper()`
	`False`

`string.lower()`	Returns a copy of the *string* converted to lowercase.
	Example (considering the same string values as used in example of previous function - **isupper**)
	`>>> string.lower()`
	`'hello'`
	`>>> string2.lower()`
	`'there'`
	`>>> string3.lower()`
	`'goldy'`
	`>>> string4.lower()`
	`'u123'`
	`>>> string5.lower()`
	`'123f'`

`string.upper()`	Returns a copy of the *string* converted to uppercase.
	Example (considering the same string values as used in example of previous function - **isupper**)
	`>>> string.upper()`
	`'HELLO'`
	`>>> string2.upper()`
	`'THERE'`
	`>>> string3.upper()`
	`'GOLDY'`
	`>>> string4.upper()`
	`'U123'`
	`>>> string5.upper()`
	`'123F'`

`string.lstrip([chars])`	Returns a copy of the *string* with leading characters removed. If used without any argument, it removes the leading whitespaces. One can use the optional **chars** argument to specify a set of characters to be removed. The **chars** argument is not a prefix ; rather, all combinations of its values (all possible substrings from the given string argument **chars**) are stripped when they lead the *string* : **Example** ▲ The example for lstrip( ) requires a detailed discussion, hence we are giving example of lstrip( ) at the end of this table.
`string.rstrip([chars])`	Returns a copy of the *string* with trailing characters removed. If used without any argument, it removes the leading whitespaces. The **chars** *argument* is a *string* specifying the set of characters to be removed. The **chars** argument is not a suffix; rather, all combinations of its values are stripped : **Example** ▲ The example for rstrip( ) requires a detailed discussion, hence we are giving example of rstrip( ) at the end of this table.

Example for lstrip( ) function

```
>>> string = "hello"
>>> string.lstrip()
'hello'
```

No argument supplied to lstrip(), hence is removed leading whitespaces of the string.

```
>>> string2 = 'There'
'There'
>>> string2.lstrip( 'the' )
'There'
```

All possible substrings of given argument 'the' are matched with left of the string2 and if found, removed. That is, 'the', 'th', 'he', 'te', 't', 'h', 'e' and their reversed strings are matched, if any of these is found, it is removed from the left of the string.

```
>>> string2.lstrip( 'The' )
're'
```

'The', 'Th', 'he', 'Te', 'T', 'h', 'e" and their reversed strings are matched, if any of these found, is removed from the left of the string 'The' found , hence removed

```
>>> string2.lstrip('he')
'There'
>>> string2.lstrip( 'Te')
'here'
```

No match in the left of string found for 'he', 'eh', 'h', 'e' ; thus same string returned.

```
>>> string2.lstrip('Teh')
're'
>>> string2.lstrip('heT')
're'
```

'heT', 'Th', 'he', 'Te', 'T', 'h', 'e" and their reversed strings are matched, if any of these found, is removed from the left of the string 'The' found , hence removed

On the same lines, you'll be able to justify that

```
>>> "saregamapadhanisa".lstrip( "tears" )
'gamapadhanisa'
>>> "saregamapadhanisa".lstrip( "races" )
'gamapadhanisa'
```

Check Point 5.2

1. What is the role of these functions ?
   (*i*) isalpha( ) (*ii*) isalnum( )
   (*iii*) isdigit( ) (*iv*) isspace( )
2. Name the case related string manipulation functions.
3. How is islower( ) function different from lower( ) function ?
4. What is the utility of find( ) function ?
5. How is capitalize( ) function different from upper( ) function ?
6. How do lstrip( ) and rstrip( ) functions work ?

Example for rstrip( ) function

The *rstrip( )* works on identical principle as that of *lstrip( )*, the only difference is that it matches from right direction. Consider following examples. We know, you'll be able to understand if you have carefully read the reasons for *lstrip( )* examples.

```
>>> string
'hello'
>>> string.rstrip()
'hello'
>>> string2
'There'
>>> string2.rstrip('ere')
```

```
'Th'
>>> string2.rstrip('care')
'Th'
>>> string2.rstrip('car')
'There'
>>>
>>> "saregamapadhanisa".rstrip( "tears" )
'saregamapadhani'
>>> "saregamapadhani".rstrip( "races" )
'saregamapadhani'
```

One more function that you have used with strings is **len( )** function which gives you the length of the string as the count of characters contained in it. Recall that you use it as :

```
len(<string>)
```

For example,

```
>>> string = 'hello'
>>> len(string)
5
```

Consider following program that applies some of the string manipulation functions that you have learnt so far.

Program 5.4 *Program that reads a line and prints its statistics like :*

Number of uppercase letters :
Number of lowercase letters:
Number of alphabets :
Number of digits:

```
line = input( "Enter a line :" )
lowercount = uppercount = 0
digitcount = alphacount = 0
for a in line :
    if a.islower() :
        lowercount += 1
    elif a.isupper() :
        uppercount += 1
    elif a.isdigit() :
        digitcount += 1
    if a.isalpha() :
        alphacount += 1
print ("Number of uppercase letters :" , uppercount)
print ("Number of lowercase letters :", lowercount)
print ("Number of alphabets :", alphacount)
print ("Number of digits :", digitcount)
```

Sample run of the program is :

```
Enter a line : Hello 123, ZIPPY zippy Zap
Number of uppercase letters :  7
Number of lowercase letters :  11
Number of alphabets :  18
Number of digits :  3
```

Program 5.5 *Program that reads a line and a substring. It should then display the number of occurrences of the given substring in the line.*

```
line = input( "Enter line :" )
sub = input( "Enter substring :" )
length = len(line)
lensub = len(sub)
start = count = 0
end = length
while True :
    pos = line.find(sub, start, end)
    if pos != -1 :
        count += 1
        start = pos + lensub
    else :
        break
    if start >= length :
        break
print ("No. of occurrences of", sub, ':', count)
```

Sample runs of above program is :

```
Enter line : jingle bells jingle bells jingle all the way
Enter substring : jingle
No. of occurrences of jingle : 3
========== RESTART ==========
Enter line : jingle bells jingle bells jingle all the way
Enter substring : bells
No. of occurrences of bells : 2
```

PriP STRING MANIPULATION — Progress In Python 5.1

This '*Progress in Python*' session works on the objective of practicing String manipulation operators and functions.

Please check the practical component-book – Progress in Computer Science with Python and fill it there in PriP 5.1 under Chapter 5 after practically doing it on the computer.

>>>❖<<<

LET US REVISE

- *Python strings are stored in memory by storing individual characters in contiguous memory locations.*
- *The memory locations of string characters are given indexes or indices.*
- *The index (also called subscript sometimes) is the numbered position of a letter in the string.*
- *In Python, indices begin 0 onwards in the forward direction up to length-1 and -1, -2, ... up to –length in the backward direction. This is called two-way indexing.*
- *For strings, + means concatenation, * means replication.*
- *The in and not in are membership operators that test for a substring's presence in the string.*
- *Comparison in Python strings takes place in dictionary order by applying character-by-character comparison rules for ASCII or Unicode.*
- *The ord( ) function returns the ASCII value of given character.*
- *The string slice refers to a part of the string s[start:end] is the element beginning at start and extending up to but not including end.*
- *The string slice with syntex s[start : end : n] is the element beginning at start and extending up to but not including end, taking every nth character.*
- *Python also provides some built-in string manipulation methods like : capitalize( ), find( ), isalnum( ), isalpha( ), isdigit( ), islower( ), isupper( ), lower( ), upper( ), lstrip( ), rstrip( ) etc.*

Solved Problems

1. *What is a string slice ? How is it useful ?*

   Solution. A sub-part or a slice of a string, say s, can be obtained using s [n : m] where n and m are integers. Python returns all the characters at indices n, $n+1$, $n+2 \ldots m-1$ *e.g.,*

   ```
   'Well done' [1 : 4]  will give 'ell'
   ```

2. *Figure out the problem with following code fragment. Correct the code and then print the output.*

   ```
   1. s1 = 'must'
   2. s2 = 'try'
   3. n1 = 10
   4. n2 = 3
   5. print (s1 + s2)
   6. print (s2 * n2)
   7. print (s1 + n1)
   8. print (s2 * s1)
   ```

   Solution. The problem is with lines 7 and 8.

   - **Line 7 – print s1 + n1** will cause error because **s1** being a string cannot be concatenated with a number **n1**.

     This problem can be solved either by changing the operator or operand *e.g.,* all the following statements will work :

     (*a*) `print (s1 * n1)`

     (*b*) `print (s1 + str(n1))`

     (*c*) `print (s1 + s2)`

◆ **Line 8** – print (s2 * s1) will cause error because two strings cannot be used for replication. The corrected statement will be :

```
print (s2 + s1)
```

If we replace the **Line 7** with its suggested solution (*b*), the output will be :

```
must try
try try try
must 10
try must
```

3. *Consider the following code :*

```
string = input("Enter a string :")
count = 3
while True :
    if string[0] == 'a' :
        string = string[2 :]
    elif string[-1] == 'b' :
        string = string [: 2]
    else :
        count += 1
        break
print (string)
print (count)
```

What will be the output produced, if the input is : (i) aabbcc (ii) aaccbb (iii) abcc ?

Solution.

(*a*) *bbcc* (*b*) *cc* (*c*) *cc*
4 4 4

4. *Consider the following code :*

```
Inp = input( "Please enter a string :" )
while len(Inp) <= 4 :
  if Inp[-1] == 'z' :                    #condition 1
      Inp = Inp [0 : 3] + 'c'
  elif ' a ' in Inp :                    #condition 2
      Inp = Inp[0] + 'bb'
  elif not int(Inp[0]) :                 #condition 3
      Inp = '1' + Inp[1 :] + 'z'
  else :
      Inp = Inp + '*'
print (Inp)
```

What will be the output produced if the input is (i) 1bzz, (ii) ' 1a ' (iii) ' abc ' (iv) ' 0xy ', (v) ' xyz ' ?

Solution.

(*i*) 1*bzc**

(*ii*) 1*bb***

(iii) endless loop because 'a' will always remain at index 0 and condition 3 will be repeated endlessly.

(iv) 1× yc•

(v) Raises an error as *lup[0]* cannot be converted to int.

5. *Write a program that takes a string with multiple words and then capitalizes the first letter of each word and forms a new string out of it.*

Solution.

```
string = input( "Enter a string :" )
length = len(string)
a = 0
end = length
string2 = ''                #empty string
while a < length :
    if a == 0 :
        string2 += string[0].upper()
        a += 1
    elif (string[a] == '' and string[a+1] != '') :
        string2 += string[a]
        string2 += string[a+1].upper()
        a += 2
    else :
        string2 += string[a]
        a += 1
print ("Original String :", string)
print ("Capitalized words String", string2)
```

6. *Write a program that reads a string and checks whether it is a palindrome string or not.*

Solution.

```
string = input( "Enter a string :" )
length = len(string)
mid = length/2
rev = -1
for a in range(mid)  :
    if string[a] == string[rev] :
        a += 1
        rev -= 1
    else :
        print (string, "is not a palindrome")
        break
else :
        print (string, "is a palindrome")
```

7. *Write a program that reads a string and displays the longest substring of the given string having just the consonants.*

Solution.

```
string = input( "Enter a string :" )
length = len(string)
```

```
maxlength = 0
maxsub = ''
sub = ''                    # empty string
lensub = 0                  # empty string
for a in range(length):
    if string[a] in 'aeiou or string[a] in 'AEIOU :
        if lensub > maxlength :
            maxsub = sub
            maxlength = lensub
            sub = ''
            lensub = 0
    else :
        sub += string[a]
        lensub = len(sub)
    a += 1
print ("Maximum length consonant substring is :" , maxsub, end = ' ')
print ("with" , maxlength,"characters")
```

8. *Write a program that reads a string and then prints a string that capitalizes every other letter in the string e.g. passion becomes pAsSiOn.*

Solution.

```
string = input( "Enter a string :" )
length = len(string)
print ("Original string :", string)
string2 = " "                          # empty string
for a in range(0, length, 2) :
    string2 += string[a]
    if a < (length-1) :
        string2 += string[a + 1].upper()
print ("Alternatively capitalized string :" , string2)
```

9. *Write a program that reads email-id of a person in the form of a string and ensures that it belongs to domain @edupillar.com. (Assumption : No invalid characters are there in email-id).*

Solution.

```
email = input( "Enter your email id :" )
domain = '@edupillar.com'
ledo = len(domain)              # ledo – length of domain
lema = len(email)               # lema – length of email
sub = email[lema-ledo :]
if sub == domain :
    if ledo != lema :
        print ("It is valid email id")
    else :
        print ("This is invalid email id - contains just the domain name.")
else:
    print ("This email-id is either not valid or belongs to some other domain.")
```

6 Debugging Programs

In This Chapter

6.1 INTRODUCTION

Computerised information systems are developed to provide computer solutions to user problems. If all the requirements of user are served by the computerized solution, the user is satisfied enough. Thus, the software should be designed effectively so as to meet its objective of user satisfaction. And one criterion of user satisfaction is an error-free code working as per desired outcome.

When you write program for something, errors are bound to creep in during development phase. Programmers find the error(s), rectify them and recompile the code until the code is error free. This process is called **debugging**. You can effectively debug your program if you know about types of errors and exceptions. This chapter talks about the same. It gives you idea of debugging with errors and exceptions in Python. This chapter also talks about Python debugger **pdb**.

6.2 WHAT IS DEBUGGING ?

Debugging in simple English means to remove 'bugs' from a program. An error causing disruption in program's running or in producing right output, is a **'program bug'**. Debugging involves rectifying the code so that the reason behind the bug gets resolved and thus bug is also removed.

In this section, we shall talk about general debugging techniques and how you can debug code in Python.

Before we talk about debugging techniques, it is important for you to know the errors, error types, what causes them etc. So, let us first talk about errors and exceptions.

DEBUGGING

Debugging refers to the process of locating the place of error, cause of error, and correcting the code accordingly.

6.3 ERRORS AND EXCEPTIONS

Errors and Exceptions, both disrupt a program and stop its execution. However, errors and exceptions are not the same. Let us first talk about errors and later we shall talk about **exceptions.**

6.3.1 Errors in a Program

An error, sometimes called 'a bug', is anything in the code that prevents a program from compiling and running correctly. Some program bugs are catastrophic in their effects, while others are comparatively less harmful and still others are so unclear that you will ever discover them. There are broadly *three* types of errors : *Compile-time errors*, *run-time errors* and *logical errors.*

6.3.1A Compile-Time Errors

Compile-Time Errors

- Syntax errors
- Semantics errors

Errors that occur during compile-time, are compile-time errors. When a program compiles, its source code is checked for whether it follows the programming language's rules or not. *Two* types of errors fall into category of compile-time errors.

1. Syntax Errors. Syntax errors occur when rules of a programming language are misused *i.e.*, when a grammatical rule of Python is violated.

For example, observe the following two statements :

```
X <- Y * Z
if X = (X * Y)
```

SYNTAX

Syntax refers to formal rules governing the construction of valid statements in a language.

These two statements will result in syntax errors as '<-' is not an assignment operator in Python and '=' is the assignment operator, not a relational operator. Also, if is a block statement, it requires a colon (:) at the end of it, which is missing above.

Therefore, the correct statements will be as follows :

```
X = Y * Z
if X == (X * Y) :
```

One should always try to get the syntax right the first time, as a syntax error wastes computing time and money, as well as programmer's time and it is preventable.

2. Semantics Errors. Semantics Errors occur when statements are not meaningful. For instance, the statement 'Sita plays Guitar' is syntactically and semantically correct as it has some meaning but the statement 'Guitar plays Sita' is syntactically correct (as the grammar is correct) but semantically incorrect. Similarly, there are semantics rules of a programming language, violation of which results in semantical errors.

SEMANTICS

Semantics refers to the set of rules which give the meaning of a statement.

For instance, the statement

```
X * Y = Z
```

will result in a semantical error as an expression cannot come on the left side of an assignment statement.

6.3.1B Run-Time Errors

Errors that occur during the execution of a program are run-time errors. These are harder to detect errors. Some run-time errors stop the execution of the program which is then called program "crashed" or "abnormally terminated".

Most run-time errors are easy to identify because program halts when it encounters them e.g., an infinite loop or wrong value (of different data type other than required) is input.

Normally, programming languages incorporate checks for run-time errors, so does Python. However, Python usually takes care of such errors by terminating the program, but a program that crashes whenever it detects an error condition is not desirable. Therefore, a program should be robust so as to recover and continue following an error.

6.3.1C Logical Errors

Sometimes, even if you don't encounter any error during compile-time and run-time, your program does not provide the correct result. This is because of the programmer's mistaken analysis of the problem he or she is trying to solve. Such errors are logical errors. For instance, an incorrectly implemented algorithm, or use of a variable before its initialization, or unmarked end for a loop, or wrong parameters passed are often the hardest to prevent and to locate. These must be handled carefully. Sometimes logical errors are treated as a subcategory of run-time errors.

6.3.2 Exceptions

Errors and exceptions are similar but different terms. While *error* represents, any bug in the code that disrupt running of the program or cause improver output, an Exception refers to any irregular situation occurring during execution/run-time, which you have no control on. Errors in a program can be fixed by making corrections in the code, fixing exceptions is not that simple.

Let us try to understand the difference between an error and exception with the help of real life example.

For instance, if you operate an ATM then

- entering wrong account number or wrong pin number is an ERROR.
- 'not that much amount in account' is an EXCEPTION.
- 'ATM machine struck' is also an EXCEPTION.

NOTE

While Error is a bug in the code that causes irregular output or stops a program from executing, an Exception is an irregular unexpected situation occurring during execution on which programmer has no control.

So you can think of **Exceptions** in a program as a situation that occurs during runtime and you have no control on it *e.g.*, you write a code that opens a data file and displays its contents on screen. This program's code is syntactically correct and logically correct too. But when you run this program, the file that you are opening, does not exist on disk — this will cause an EXCEPTION. So the program has no errors but an exception occurred. A good program is the one that has a code that handles an exception well *i.e.*, in case an exception occurs, then the exception handling code written for it, will take over and do the necessary action.

If a program does not have exception handling routine, then in case of an exception's occurrence, the program will halt.

In the following lines, we are giving a brief idea about how exception handling routines are written in Python.

NOTE

Unhandled exceptions will cause Python to halt execution.

Exception Handling in Python – Basic Idea

Exception Handling in Python involves the use of try and except clauses in the following format wherein the code that may generate an exception is written in the *try block* and the code for handling exception when the exception is raised, is written in *except block*.

See below :

```
try :
    # write here the code that may generate an exception
except :
    # write code here about what to do the exception has occurred
```

For instance, consider the following code :

These keywords are used to handle exceptions in Python

```
try :
    print ("result of 10 / 5 = ", (10/5))
    print ("result of 10 / 0 = ", (10/0))
except :
    print ("Divide by Zero Error! Denominator must not be zero!")
```

The code that may raise an exception (e.g.,a division may have divisor as zero) is written in try block

This is except block; this will execute when the exception is raised

The output produced by above code is as shown below :

```
result of 10/5 = 2
result of 10/0 = Divide by Zero Error! Denominator must not be zero!
```

See, the expression (10 /0) raised exception which is then handled by except block

See, now the output produced does not show the scary red-coloured standard error message; it is now what you defined under the exception block.

Python comes with many predefined exceptions, also known as *built-in exceptions*. Some common built-in exceptions in Python are listed in Table 6.1.

Table 6.1 *Some Built-in Exceptions*

Exception Name	Description
EOFError	Raised when one of the built-in functions (input( )) hits an end-of-file condition (EOF) without reading any data.
IOError	Raised when an I/O operation (such as a **print( )**, the built-in open( ) function or a method of a file object) fails for an I/O-related reason, *e.g., "file not found"* or *"disk full"*.
NameError	Raised when an identifier name is not found.
IndexError	Raised when a sequence subscript or index is *out of range, e.g.,* from a string of length 4 if you try to read a value of index like 4 or more *i.e.,* string[4], string[5], string[–5] etc. will raise exception as legal indexes for a string of length 4 are 0, 1, 2, 3 and –1, –2, –3, –4 only.
ImportError	Raised when an **import** statement fails to find the module definition or when a **from ... import** fails to find a name that is to be imported.
TypeError	Raised when an operation or function is applied to an object of inappropriate type, *e.g.,* if you try to compute a square-root of a string value.
ValueError	Raised when a built-in operation or function receives an argument with an inappropriate value *e.g., int("z10")* will raise *ValueError*.
ZeroDivisionError	Raised when the second argument of a division or modulo operation is zero.
OverflowError	Raised when the result of an arithmetic operation is too large to be represented.
KeyError	Raised when a mapping (dictionary) key is not found in the set of existing keys.

We are not going in further details of exceptions here.

6.4 HOW TO DEBUG A PROGRAM ?

Debugging involves correction of code so that the cause of errors is removed. But how do you know what are the errors ? Well, the **compile time errors** (*syntax errors* and *semantic errors*) are caught during compilation and for **logical errors**, you perform **testing**. Testing is done to verify correct behaviour *i.e.,* with some sample values (test cases) whose output is already known, the code is tested – if it produces the anticipated output, code is correct and if it does not, there is some error. Once you know the errors, you can debug your program.

Let us now learn some useful debugging techniques.

6.4.1 Debugging Techniques

Debugging a program is a skill. There are many traditional debugging techniques that you can follow to debug your code. These are :

1. Carefully spot the origin of error.
2. Print variables' intermediate values.
3. Code tracing and stepping.

1. Carefully Spot the Origin of Error

When you run your code, Python interpreter will run a line of code if it is free from any syntax and semantic error, and will list the error if the running line of code has some error, e.g.,

Consider the following code that **inputs a number and then prints its powers from 0 to 3**:

```
1.  a = int(input("Enter a number"))
2.  j = "a"
3.  for i in range(4):
4.      print(j ** i)
```

Upon running the above code, Python reports

See, the interpreter reported error in line4
But the real error does not lie in line4

```
File "E:/temp.py", line 4, in <module>
    print(j ** i)

TypeError : unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

But the statement

```
print(j **i)
```

is syntactically and semantically correct. Then how to figure out the real origin of the error.

When Python gives you a line number for the error, then :

(*i*) It means the error has manifested in this line.

(*ii*) Its origin may be in this line or in the lines above it.

So, start looking backwards from the line of error, carefully reading the statements and you will find the real cause of the error, *e.g.*, for the above error, the real cause of error is in **line 2**, where rather than assigning the variable *a*, we assigned string "*a*" to variable j .

```
1.  a = int(input("Enter a number"))
2.  j = "a"
3.  for i in range(4):
4.      print(j ** i)
```

Error occurred because j was mistakingly assigned string "a" in place of variable a

So, we must correct line 2 as :

```
j = a
```

And now, the code runs perfectly fine and gives output as :

```
1
5
25
125
```

NOTE

The origin of error may be in or before the line of error. So, fix the cause, not the symptom.

2. Print Variables' Intermediate Values

Depending upon your algorithms, the variables' values change during the execution. Sometimes you get incorrect output but you cannot figure out what is causing it. In that case, it is a good idea to add many print( ) statements (temporarily) to inspect values of variables after each step *e.g.*,

Consider the following code that is trying to print some Fibonacci terms :

```
a = 0
b = 1
print(a)
print(b)
for i in range(5):
    c = a + b
    print(c, end = " ")
    b = c
    a = b
```

But it starts producing incorrect terms after printing some correct terms :

```
0
1
1   2   4   8   16
```

Now to debug this code, it is a good idea to temporarily add print( ).statements with some hint string identifying variable, to check changing values of variables **a, b** and **c**, as this :

```
:
for i in range(5):
    c = a + b
    print("c = ", c)
    a = b
    b = c
    print("a = ", a, end = " ")
    print("b = ", b)
```

The print( ) statements added to check on intermediate values of variables a and b

Now upon running the code, you can carefully look at the output produced to figure out the error :

```
0
1
c = 1
a = 1    b = 1
c = 2
a = 2    b = 2
c = 4
a = 4    b = 4
c = 8
a = 8    b = 8
c = 16
a = 16   b = 16
```

As per Fibonacci logic, a should have been 1 but it is 2, which means some problem with the assignment : either the assignment statement is incorrect or the order of the assignment statements is incorrect

Now carefully look at the code. You will find out that the **order of the assignment statements is incorrect** : variable b first loses its value before assigning it to variable a, *i.e.*, :

```
a = b
b = c
```

So correct this and the code now becomes :

```
a = 0
b = 1
print(a)
print(b)
for i in range(5) :
    c = a + b
    print(c)
    a = b
    b = c
```

This change of order of statements will correct the problem and print the correct Fibonacci terms

NOTE

So you can say that debugging broadly means :

(a) Figure out the origin of error in code

(b) Fix the erroneous code

(c) Review and rerun your code to ensure that error is fixed

3. Code Tracing

Another useful technique is code tracing. Code tracing means executing code one line at a time and watching its impact on variables. Code tracing is often done with built-in debugging tools or debuggers.

6.5 USING DEBUGGER TOOL

A Debugging tool or debugger is a specialized computer program/software that can be used to test and debug programs written in a specific programming language. **A Debugger** can be a separate program or integrated with IDEs (Integrated Development Environment). Python also provides a separate debugger program called pdb. In this section, we shall learn to work with both these types of debuggers. First, we shall learn to work with integrated debugger of **Spyder IDE** of Python and then we shall learn to use **pdb** — separate Python debugger program.

6.5.1 Working with Integrated Debugger Tool of Spyder IDE

Spyder IDE provides a graphical front end to Python's debugger **pdb**, thus makes it easier to use. This is called **winpdb**. Let us first learn to use this GUI debugger of Spyder IDE and later we shall learn to work with Python's command based debugger pdb.

To debug your code through Spyder IDE's interactive debugger, follow these steps :

1. Before you start with debugging process, make sure that **Variable Explorer** pane is visible. (see on next page)

   If, however, this pane is not visible, you can open it using command :

   View menu → Panes → Variable explorer

   Or by pressing shortcut key **Ctrl+Shift+V**.

PriP DEBUGGING — Progress In Python 6.1

Load a program that is producing some error or giving you incorrect output.

1. Figure out the problem in your program using Spyder IDE's interactive GUI debugger.
2. Using **pdb**, pinpoint the error causing statements in your program and fix them.

Please check the practical component-book – Progress in Computer Science with Python and fill it there in PriP 6.1 under Chapter 6 after practically doing it on the computer.

>>>❖<<<

LET US REVISE

- *Debugging refers to the process of locating the place of error, cause of error, and correcting the code accordingly.*
- *Compile-time errors (syntax error and semantics errors) refer to the errors that violate the grammatical rules and regulations of a programming language.*
- *Runtime errors occur during the execution of a program.*
- *Logical errors occur due to mistaken analysis of the problem.*
- *Irregular unexpected situations occurring during runtime are called Exceptions.*
- *Exceptions may occur even if the program is free from all types of errors.*
- *Debugger is a tool that lets you trace and execute code line by line.*
- *Python provides debugger named as pdb.*

Solved Problems

1. *What do you mean by Syntax errors and Semantics errors ?*

   Solution. Syntax errors are the errors that occur when rules of a programming language are violated. Semantics errors occur when statements are not meaningful.

2. *Why are logical errors harder to locate ?*

   Solution. In spite of logical errors' presence, the program executes without any problems but the output produced is not correct. Therefore, each and every statement of the program needs to be scanned and interpreted. Thus the logical errors are harder to locate.

3. *What is an Exception ?*

   Solution. Exception in general refers to some contradictory or unusual situation which can be encountered unexpectedly while executing a program.

4. *Why is Exception Handling required ?*

Solution. The exception handling is ideal for processing exceptional situations in a controlled way so that program ends gracefully rather than abrupt crashing of the program.

5. *What is the need for debugger tool?*

Solution. Debugger tools are very useful especially if the code is big or the error is not very clear, it becomes very difficult to manually figure out the origin and cause of problem. Debugger tools here prove very handy and useful. They show us the line by line execution and its result on variables interactively and help a programmer get to the root of the problem.

GLOSSARY

Error	Violation of syntax, semantics or problem-specifications.
Exception	Occurrence of irregular, unreasonable, exceptional situations during runtime.
Code Tracing	Executing code one line at a line while showing the code line being executed.

Assignment

1. What are main error types ? Which types are most dangerous and why ?
2. Correct any false statements :
   (*a*) Compile-time errors are usually easier to detect and to correct than run-time errors.
   (*b*) Logically errors can usually be detected by the compiler.
3. Differentiate between a syntax error and a semantics error.
4. Differentiate between a syntax error and a logical error in a program. When is each type of error likely to be found ?
5. What is the difference between an error and exception ?
6. How can you handle an exception in Python ? Write sample code to illustrate it.
7. Name some common built-in exceptions in Python.
8. When does these exceptions occur ?
   (*a*) Type Error (*b*) Index Error (*c*) Name Error
9. What is debugging and code tracing ?
10. What are these pdb commands used for ?
    (*a*) b (*b*) cl (*c*) l (*d*) h (*e*) n (*f*) p

7 List Manipulation

In This Chapter

7.1 INTRODUCTION

The Python lists are containers that are used to store a list of values of any type. Unlike other variables **Python lists are mutable** *i.e.,* you can change the elements of a list in place ; Python will not create a fresh list when you make changes to an element of a list. List is a type of sequence like strings and tuples but it differs from them in the way that lists are *mutable* but strings and tuples are *immutable.*

This chapter is dedicated to basic list manipulation in Python. We shall be talking about creating and accessing lists, various list operations and list manipulations through some built-in functions.

7.2 CREATING AND ACCESSING LISTS

A list is a standard data type of Python that can store a sequence of values belonging to any type. The *Lists* are depicted through square brackets, *e.g.*, following are some lists in Python:

```
[]                          # list with no member, empty list
[1, 2, 3]                   # list of integers
[1, 2.5, 3.7, 9]            # list of numbers (integers and floating point)
['a', 'b', 'c']             # list of characters
['a', 1, 'b', 3.5, 'zero']  # list of mixed value types
['One', 'Two', 'Three']     # list of strings
```

Before we proceed and discuss how to create lists, one thing that must be clear is that **Lists are mutable** (*i.e.*, modifiable) *i.e.*, you can change elements of a list in place. In other words, the memory address of a list will not change even after you change its values. List is one of the two mutable types of *Python – Lists* and *Dictionaries* are mutable types ; all other data types of Python are immutable.

NOTE

Lists are mutable sequences of Python *i.e.*, you can change elements of a list in place.

7.2.1 Creating Lists

To create a list, put a number of expressions in square brackets. That is, use square brackets to indicate the *start* and *end* of the list, and separate the items by commas. *For example* :

```
[2, 4, 6]
['abc', 'def']
[1, 2.0, 3, 4.0]
[]
```

NOTE

Lists can contain values of mixed data types.

Thus to create a list you can write in the form given below :

```
L = []
L = [value, ...]
```

This construct is known as a **list display construct.**

Consider some more examples :

- **The empty list**

  The empty list is [ ]. It is the list equivalent of 0 or '' and like them it also has truth value as *false*. You can also create an empty list as :

  ```
  L = list()
  ```

  It will generate an empty list and name that list as L.

NOTE

Lists are formed by placing a comma-separated-list of expressions in square brackets.

- **Long lists**

  If a list contains many elements, then to enter such long lists, you can split it across several lines, like below :

  ```
  sqrs = [ 0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169,
  196, 225, 256, 289, 324, 361, 400, 441, 484, 529, 576, 625 ]
  ```

  Notice the opening square bracket and closing square brackets appear just in the beginning and end of the list.

Nested lists

A list can have an element in it, which itself is a list. Such a list is called nested list, *e.g.*,

```
L1 = [3, 4, [5, 6], 7]
```

L1 is a nested list with four elements : 3, 4, [5, 6] and 7. L1[2] element is a list [5, 6]. Length of L1 is 4 as it counts [5, 6] as one element.

Creating Lists from Existing Sequences

You can also use the built-in list type object to create lists from sequences as per the syntax given below :

```
L = list(<sequence>)
```

where *<sequence>* can be any kind of sequence object including strings, tuples, and lists. Python creates the individual elements of the list from the individual elements of passed sequence. If you pass in another list, the list function makes a copy.

Consider following examples :

```
>>> l1 = list('hello')
>>> l1
['h', 'e', 'l', 'l', 'o']
```

List l1 is created from another sequence – a string "hello"
It generated individual elements from the individual letters of the string

```
>>> t = ('w', 'e' , 'r' , 't', 'y')
>>> l2 = list(t)
>>> l2
['w', 'e', 'r', 't', 'y']
```

List l2 is created from another sequence – a tuple t
It generated individual elements from the individual elements of the passed tuple t.

You can use this method of creating lists of single characters or single digits via keyboard input. Consider the code below :

```
l1 = list(input( 'Enter list elements: '))
enter list elements: 234567
>>> l1
['2', '3', '4', '5', '6', '7']
```

See, it created the elements of list l using each of the character input

Notice, this way the data type of all characters entered is *string* even though we entered digits. To enter a list of integers through keyboard, you can use the method given below.

Most commonly used method to input lists is **eval(input( ))** as shown below :

```
list = eval(input( "Enter list to be added:" ))
print ("list you entered :", list)
```

eval( ) tries to identify type by looking at the given expression (Read on next page)

when you execute it, it will work somewhat like :

```
Enter list to be added:[67, 78, 46, 23]
list you entered : [67, 78, 46, 23]
```

Please note, sometimes (not always) **eval( )** does not work in Python Shell. At that time, you can run it through a script or program too.

The eval( ) Function

The eval( ) function of Python can be used to evaluate and return the result of an expression given as string. For example :

```
eval('5+8')
```

will give you result as 13

Similarly, following code fragment

```
y = eval("3*10")
print(y)
```

will print value as *30*

Since eval( ) can interpret an expression given as string, you can use it with input( ) too :

```
var1 = eval(input("Enter Value:"))
print(var1, type(var1))
```

Executing this code will result as :

```
Enter Value: 15 + 3
18 <class 'int'>
```

See, the eval( ) has not only interpreted the string "15+3" as 18 but also stored the result as *int* value. Thus with eval( ), if you enter an integer or float value , it will interpret the values as the intended types :

```
var1 = eval(input("Enter Value:"))
print(var1, type(var1))
```

```
Enter Value: 75
75 <class 'int'>
```

```
var1 = eval(input("Enter Value: "))
print(var1, type(var1))
```

```
Enter Value: 89.9
89.9 <class 'float'>
```

You can use eval( ) to enter a list or tuple also. Use [ ] to enter lists' and ( ) to enter tuple values

```
var1 = eval(input("Enter Value: "))
print(var1, type(var1))
```

```
Enter Value: [1, 2, 3]
[1, 2, 3] <class 'list'>
```

```
var1 = eval(input("Enter Value: "))
print(var1, type(var1))
```

```
Enter Value: (2, 4, 6, 8)
(2, 4, 6, 8) <class 'tuple'>
```

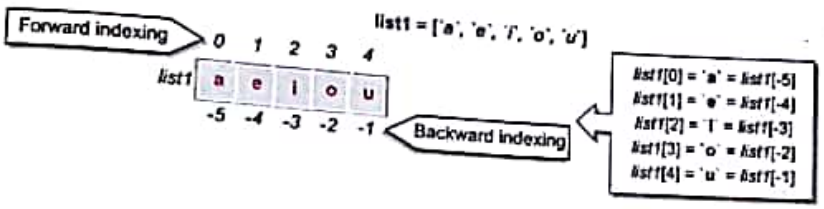
However, use of eval( ) may lead to unforeseen problems, and thus, use of eval( ) is always discouraged.

7.2.2 Accessing Lists

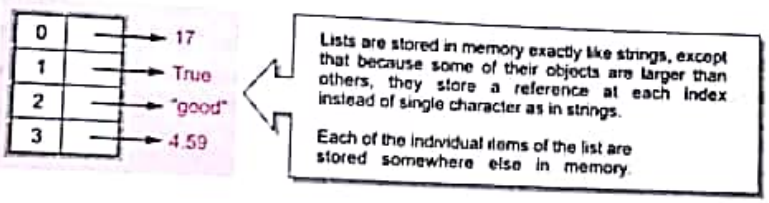
Lists are mutable (editable) sequences having a progression of elements. There must be a way to access its individual elements and certainly there is. But before we start accessing individual elements, let us discuss the similarity of Lists with strings that will make it very clear to you how individual elements are accessed in lists – the way you access string elements. Following subsection will make the things very clear to you, I bet ! ☺

Similarity with Strings

Lists are sequences just like strings that you have read in previous chapter. They also index their individual elements, just like strings do. Recall figure 3.1 of chapter 3 that talks about indexing in strings. In the same manner, list-elements are also indexed, *i.e.*, forward indexing as *0, 1, 2, 3,...* and backward indexing as *–1, –2, –3, ...* [See Fig 7.1(*a*)].



(*a*) List Elements' two way indexing



(*b*) How lists are internally organized

Figure 7.1

Thus, you can access the list elements just like you access a string's elements *e.g.*, **List[i]** will give you the element at *i*th index of the list ; **List[a:b]** will give you elements between indexes *a* to *b* – 1 and so on.

Put in another words, Lists are similar to strings in following ways :

- **Length**

  Function **len(L)** returns the number of items (count) in the list *L*.

- **Indexing and slicing**

  L[i] returns the item at index *i* (the first item has index 0), and

  L[i : j] returns a new list, containing the objects at indexes between *i* and *j* (excluding index *j*).

- **Membership operators**

  Both **'in'** and **'not in'** operators work on *Lists* just like they work for other sequences. That is, in tells if an element is present in the list or not, and not in does the opposite.

- **Concatenation and replication operators + and ***

  The + operator adds one list to the end of another. The * operator repeats a list. We shall be talking about these two operations in a later section 7.3 – *List Operations*.

Accessing Individual Elements

As mentioned, the individual elements of a list are accessed through their indexes. Consider following examples :

```
>>> vowels = ['a', 'e', 'i', 'o', 'u']
>>> vowels[0]
'a'
>>> vowels[4]
'u'
>>> vowels[-1]
'u'
>>> vowels[-5]
'a'
```

NOTE

While accessing list elements, if you pass in a negative index, Python adds the length of the list to the index to get element's forward index. That is, for a 6-element list L, *L[–5]* will be internally computed as : *L [–5 + 6] = L [1]*, and so on.

Like strings, if you give index outside the legal indices (0 to *length* – 1 or – *length*, – *length* +1, ..., uptill –1) while accessing individual elements, Python will raise **Index Error** (see below)

```
>>> vowels = ['a', 'e', 'i', 'o', 'u']
>>> vowels[5]
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    vowels[5]
IndexError: list index out of range
```

5 is not legal index for vowels list, thus cannot be used to access individual element

Difference from Strings

Although lists are similar to strings in many ways, yet there is an important **difference in mutability of the two. Strings are not mutable, while lists are.** You cannot change individual elements of a string in place, but Lists allow you to do so. That is, following statement is fully valid for Lists (though not for strings) :

```
L[i] =<element>
```

For example, consider the same *vowels* list created above, that stores all vowels in lower case. Now, if you want to change some of these vowels, you may write something as shown below:

```
>>> vowels[0] = 'A'
>>> vowels
['A', 'e', 'i', 'o', 'u']
>>> vowels[-4] = 'E'
>>> vowels
['A', 'E', 'i', 'o', 'u']
```

Notice, it changed the element in place; ('a' changed to 'A') no new list created – because Lists are MUTABLE.

NOTE

Lists are similar to strings in many ways like indexing, slicing and accessing individual elements but they are different in the sense that ***Lists are mutable while strings are not.***

Traversing a List

Recall that traversal of a sequence means accessing and processing each element of it. Thus traversing a list also means the same and same is the tool for it, *i.e.*, the Python loops. That is why sometimes we call a traversal as looping over a sequence.

The for loop makes it easy to traverse or loop over the items in a list, as per following syntax :

```
for <item> in <List>:
    process each item here
```

For example, following loop shows each item of a list L in separate lines :

```
L = ['P', 'y', 't', 'h', 'o', 'n']
for a in L :
    print (a)
```

The above loop will produce result as :

```
P
y
t
h
o
n
```

How it works

The loop variable a in above loop will be assigned the List elements, one at a time. So, loop-variable a will be assigned 'P' in *first* iteration and hence 'P' will be printed ; in *second* iteration, a will get element 'y' and 'y' will be printed; and so on.

If you only need to use the indexes of elements to access them, you can use functions ***range( )*** and ***len( )*** as per following syntax :

```
for index in range(len(L)):
    process List[index] here
```

Consider program 7.1 that traverses through a list using above format and prints each item of a list L in separate lines along with its index.

Program 7.1 *Program to print elements of a list ['q', 'w', 'e', 'r', 't', 'y'] in separate lines along with element's both indexes (positive and negative).*

```
L = [ 'q', 'w', 'e', 'r', 't', 'y']
length = len(L)
for a in range(length) :
    print("At indexes", a, "and", (a - length), "element :", L[a])
```

Sample run of above program is :

```
At indexes 0 and -6 element : q
At indexes 1 and -5 element : w
At indexes 2 and -4 element : e
At indexes 3 and -3 element : r
At indexes 4 and -2 element : t
At indexes 5 and -1 element : y
```

Comparing Lists

You can compare two lists using standard comparison operators of Python, *i.e.*, <, >, ==, !=, etc. Python internally compares individual elements of lists (and tuples) in lexicographical order. This means that to compare equal, each corresponding element must compare equal and the two sequences must be of the same type *i.e.*, having comparable types of values.

Consider following examples :

```
>>> L1 , L2 = [1, 2, 3], [1, 2, 3]
>>> L3 = [1, [2, 3]]
>>> L1 == L2
True
>>> L1 == L3
False
```

For comparison operators >, <, >= , <=, the corresponding elements of two lists must be of comparable types , otherwise Python will give error.

Consider the following considering the above two lists :

```
>>> L1 < L2
False
>>> L1 < L3
Traceback (most recent call last):
    File "<ipython-input-180-84fdf598c3f1>", line 1, in <module>
        L1 < L3
    TypeError : '<' not supported between instances of 'int' and 'list'
```

For first comparison, Python did not give any error as both lists have values of same type.

For second comparison, as the values are not of comparable types, Python raised error

Python raised error above because the corresponding second elements of lists *i.e.*, L1[1] and L3[1] are not of comparable types. L1[1] is *integer* (2) and L3[1] is a *list* [2, 3] and list and numbers are not comparable types in Python.

Python gives the final result of non-equality comparisons as soon as it gets a result in terms of True/False from corresponding elements' comparison. If corresponding elements are equal, it goes on to the next element, and so on, until it finds elements that differ. Subsequent elements are not considered (even if they are really big). See below.

Table7.1 *Non-equality Comparison in List Sequences*

Comparison	Result	Reason
`[1, 2, 8, 9] < [9, 1]`	True	Gets the result with the comparison of corresponding first elements of two lists `1 < 9 is True`
`[1, 2, 8, 9] < [1, 2, 9, 1]`	True	Gets the result with the comparison of corresponding third elements of two lists `8 < 9 is True`
`[1, 2, 8, 9] < [1, 2, 9, 10]`	True	Gets the result with the comparison of corresponding third elements of two lists `8 < 9 is True`
`[1, 2, 8, 9] < [1, 2, 8, 4]`	False	Gets the result with the comparison of corresponding fourth elements of two lists `9 < 4 is False`

So, for comparison purposes, Python internally compares individual elements of two lists, applying all the comparison rules that you have read earlier. Consider following code :

```
>>> a = [2, 3]
>>> b = [2, 3]
>>> c = ['2', '3']
>>> d = [2.0, 3.0]
>>> e = [2, 3, 4]
>>> a == b
True
>>> a == c
False
>>> a > b
False
>>> d > a
False
>>> d == a
True
>>> a < e
True
```

Notice for comparison purposes, Python ignored the types of elements and compared values only

For two lists to be equal, they must have same number of elements and matching values (recall int and float with matching values are considered equal)

There is also a *cmp( )* function that can be used for sequences' comparisons but we are not covering it here.

7.3 LIST OPERATIONS

The most common operations that you perform with lists include *joining* lists, *replicating* lists and *slicing* lists. In this section, we are going to talk about the same.

7.3.1 Joining Lists

Joining two lists is very easy just like you perform addition, literally ;-) . The concatenation operator +, when used with two lists, joins two lists. Consider the example given below :

```
>>> lst1 = [ 1, 3, 5]
>>> lst2 = [6, 7, 8 ]
>>> lst1 + lst2
[1, 3, 5, 6, 7, 8]
```

The + operator concatenates two lists and creates a new list

As you can see that the resultant list has firstly elements of first list *lst1* and followed by elements of second list *lst2*. You can also join two or more lists to form a new list, *e.g.*,

```
>>> lst1 = [10, 12, 14]
>>> lst2 = [20, 22, 24]
>>> lst3 = [30, 32, 34]
>>> lst = lst1 + lst2 + lst3
>>> lst
[10, 12, 14, 20, 22, 24, 30, 32, 34]
```

The + operator is used to concatenate three individual lists to get a new combined list lst.

The + operator when used with lists requires that both the operands must be of list types. You cannot add a number or any other value to a list. *For example,* following expression will result into error :

```
list + number
list + complex-number
list + string
```

Consider the following examples

See errors generated when anything other than a list is added to a list

```
>>> lst1 = [10, 12, 14]
>>> lst1 + 2
Traceback (most recent call last):
 File "<pyshell#42>", line 1, in <module>
    lst1+2
TypeError: can only concatenate list (not "int") to list
>>> lst1 + "abc"
Traceback (most recent call last):
 File "<pyshell#44>", line 1, in <module>
    lst1 + "abc"
TypeError: can only concatenate list (not "str") to list
```

7.3.2 Repeating or Replicating Lists

Like strings, you can use * operator to replicate a list specified number of times, *e.g.*, (considering the same list *lst1 = [1, 3, 5]*)

*The * operator repeats a list specifies number of times and creates a new list*

```
>>> lst1 * 3
[1, 3, 5, 1, 3, 5, 1, 3, 5]
```

Like strings, you can only use an integer with a * operator when trying to replicate a list.

NOTE

When used with lists, the + operator requires both the operands as list-types; and the * operator requires a list and an integer.

7.3.3 Slicing the Lists

List slices, like string slices are the sub part of a list extracted out. You can use indexes of list elements to create *list slices* as per following format :

```
seq = L[start:stop]
```

The above statement will create a *list slice* namely **seq** having elements of list L on indexes *start*, *start+1*, *start+2*,...., *stop–1*. Recall that index on last limit is not included in the *list slice*. The *list slice* is a list in itself, that is, you can perform all operations on it just like you perform on lists.

Consider the following example :

```
>>> lst =[10, 12, 14, 20, 22, 24, 30, 32, 34]
>>> seq = lst [ 3: -3]
>>> seq
[20, 22, 24]
>>> seq[1] = 28
>>> seq
[20, 28, 24]
```

Trying to modify an element of seq in place ; and it successfully does because the list slice seq is a list in itself.

NOTE

L[start:stop] creates a list slice out of list L with elements falling between indexes ***start*** and ***stop***, not including ***stop***.

For normal indexing, if the resulting index is outside the list, Python raises an **IndexError exception**. Slices are treated as boundaries instead, and the result will simply contain all items between the boundaries. For the ***start*** and ***stop*** given beyond list limits in a list slice (*i.e.*, out of bounds), Python simply returns the elements that fall between specified boundaries, if any, without raising any error.

For example, consider the following :

```
>>> lst =[10, 12, 14, 20, 22, 24, 30, 32, 34]
>>> lst [3:30]
[20, 22, 24, 30, 32, 34]
>>> lst [ -15 :7]
[10, 12, 14, 20, 22, 24, 30]

>>> L1 [2, 3, 4, 5, 6, 7, 8]
>>> L1 [2 : 5]
[4, 5, 6]
>>> L1 [ 6 : 10]
[8]
>>> L1 [10 : 20]
[ ]
```

Giving upper limit way beyond the size of the list, but Python return elements from list falling in range 3 onwards <30

Giving lower limit much lower, but Python returns elements from list falling in range -15 onwards <7

Legal index range is 0..6 and -7 .. -1

One limit is out of bounds

No error ! Python returns a sub-list as per given range

Both limits are out of bounds

Python gives no error and returns an empty sequence as no element of L1 has index falling in range of 10 to 20.

Lists also support ***slice steps***. That is, if you want to extract, not consecutive but every other element of the list, there is a way out – the *slice steps*. The *slice steps* are used as per following format :

```
seq = L[start:stop:step]
```

Consider some examples to understand this.

```
>>> lst
[10, 12, 14, 20, 22, 24, 30, 32, 34]

>>> lst[0 : 10 : 2 ]
[10, 14, 22, 30, 34]

>>> lst[2 : 10 : 3 ]
[14, 24, 34]

>>> lst[:: 3 ]
[10, 20, 30]
```

*Include every 2nd element, i.e., **skip 1 element** in between. Check resulting list slice*

*Include every 3rd element, i.e., **skip 2 elements** in between*

No start and stop given. Only step is given as 3. That is, from the entire list, pick every 3rd element for the list slice.

NOTE

L[start : stop: step] creates a list slice out of list L with elements falling between indexes ***start*** and ***stop***, *not including stop*, skipping step-1 elements in between.

Consider some more examples :

```
seq = L[::2]      # get every other item, starting with the first
Seq = L[5::2]     # get every other item, starting with the sixth element, i.e., index 5
```

Like strings, if you give <Listname> [::–1], it will reverse the list e.g., for a list say List = [5, 6, 8, 11, 3], following expression will reverse it :

```
>>> List [ :: -1]
[3, 11, 8, 6, 5]
```

See, List reversed

Program 7.2 *Extract two list-slices out of a given list of numbers. Display and print the sum of elements of first list-slice which contains every other element of the list between indexes 5 to 15. Program should also display the average of elements in second list slice that contains every fourth element of the list. The given list contains numbers from 1 to 20.*

```
lst = [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
slc1 = lst[5 : 15 :2]
slc2 = lst[::4]
sum = avg = 0
print ("Slice 1")
for a in slc1 :
    sum += a
    print (a, end = ' ')
print ()
print ("Sum of elements of slice 1:", sum)
print ("Slice 2")
sum = 0
for a in slc2 :
    sum += a
    print (a, end = ' ')
print ()
avg = sum /len(slc2)
print ("Average of elements of slice 2:", avg)
```

Sample run of above program is :

```
Slice 1
6 8 10 12 14
Sum of elements of slice 1: 50
Slice 2
1 5 9 13 17
Average of elements of slice 2: 9
```

Using Slices for List Modification

You can use slices to overwrite one or more list elements with one or more other elements. Following examples will make it clear to you :

```
>>> L = ["one", "two", "THREE"]
>>> L[0:2] = [ 0, 1 ]
>>> L
[0, 1, "THREE"]
>>> L =["one", "two", "THREE"]
>>> L[0:2] = "a"
>>> L
["a", "THREE"]
```

assigning new values to list slice

Notice, what the list changes to.

Notice, what the list changes to.

In all the above examples, we have assigned new values in the form of a sequence. **The values being assigned must be a sequence, *i.e.*,** a list or string or tuple etc.

For example, following assignment is also valid.

```
>>> L1 = [1, 2, 3]
>>> L1 [2 :] = "604"
>>> L1
[1, 2, 3, '6', '0', '4']
```

String is also a sequence

But if you try to assign a non-sequence value to a list slice such as a number, Python will give an error, *i.e.*,

```
>>> L1 [2 :] = 345
File "<.....>", line 1, in <module>
    L1 [2:] = 345
TypeError: can only assign an iterable
```

345 is a number, not a sequence

But here, you should also know something. If you give a list slice with range much outside the length of the list, it will simply add the values at the end *e.g.*,

```
>>> L1 = [1, 2, 3]
>>> L1 [ 10 : 20] = "abcd"
>>> L1
[1, 2, 3, 'a', 'b', 'c', 'd']
```

no error even though list slice limits are outside the length. Now Python will append the letters to the end of list L1.

7.4 WORKING WITH LISTS

Now that you have learnt to access the individual elements of a list, let us talk about how you can perform various operations on lists like : *appending, updating, deleting* etc.

Appending Elements to a List

You can also add items to an existing sequence. **The append( ) method adds a single item to the end of the list**. It can be done as per following format :

```
L.append(item)
```

Consider some examples :

```
>>> lst1 = [10, 12, 14]
>>> lst1.append(16)
>>> lst1
[10, 12, 14, 16]
```

The element specified as argument to append() is added at the end of existing list

Updating Elements to a List

To update or change an element of the list in place, you just have to assign new value to the element's index in list as per syntax :

```
L[index] = <new value>
```

Consider following example :

```
>>> lst1 = [10, 12, 14, 16]
>>> lst1[2] = 24
>>> lst1
[10, 12, 24, 16]
```

Statement updating an element (3rd element – having index 2) in the list.
Display the list to see the updated list.

Deleting Elements from a List

You can also remove items from lists. The del statement can be used to remove an individual item, or to remove all items identified by a slice. It is to be used as per syntax given below :

```
del List [ <index>]              # to remove element at index
del List [<start> : <stop> ]     # to remove elements in list slice
```

Consider following examples :

```
>>> lst =[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
>>> del lst[10]
>>> lst
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13, 14, 15, 16, 17, 18, 19, 20]
>>> del lst[10:15]
>>> lst
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 17, 18, 19, 20]
```

Delete element at index 10 in list namely lst.

See 11 got deleted. Compare with list above

Delete all elements between indexes 10 to 15 in list namely lst. Compare the result displayed below

If you use del <lstname> only *e.g.*, del lst, it will delete all the elements and the list object too. After this, no object by the name lst would be existing.

You can also use pop( ) method to remove single element, not list slices. The *pop( ) method* removes an individual item and returns it. The **del statement** and the **pop method** do pretty much the same thing, except that pop method also returns the removed item along with deleting it from list. The *pop( ) method* is used as per following format :

```
List.pop( <index>)   # index optional; if skipped, last element is deleted
```

Consider examples below :

```
>>> lst = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
16, 17, 18, 19, 20]
>>> lst.pop()
20
>>> lst.pop(10)
11
```

NOTE

While **del statement** can remove a single element or a list-slice from a list, the **pop( )** can remove only single element, not list slices. Also pop( ) retruns the deleted element too.

The *pop( )* method is useful only when you want to store the element being deleting for later use, *e.g.*,

```
item1 = L.pop()      # last item
item2 = L.pop(0)     # first item
item3 = L.pop(5)     # sixth item
```

Now you can use item1, item2 and item3 in your program as per your requirement.

LISTS IN PYTHON – I

Progress In Python 7.1

This PriP session works on the objective of practicing List Manipulation operators.

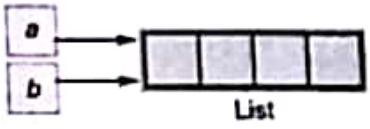
Making True Copy of a List

Sometimes you need to make a copy of a list and you generally tend to do it using assignment, *e.g.*,

```
a = [ 1, 2, 3]
b = a
```

Trying to make copy of list a in list b

But it will not make b as a duplicate list of a ; rather just like Python does, it will make label b to point to where label a is pointing to, *i.e.*, as shown in adjacent figure.

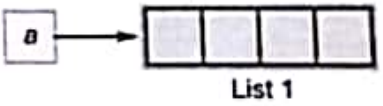


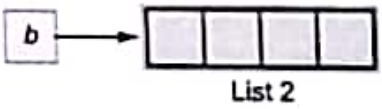
It will work fine as long as we do not modify lists. Now, if you make changes in any of the lists, it will be reflected in other because *a* and *b* are like aliases for the same list. See below :

```
>>> a = [1, 2, 3]
>>> b = a
>>> a[1] = 5
>>> a
[1, 5, 3]
>>> b
[1, 5, 3]
```

See, list b is also reflecting the changed value of list a.

See, along with list *a*, list *b* also got modified. What if you want that the copy list *b* should remain unchanged while we make changes in *list a* ? That is, you want both lists to be independent of each other as shown in adjacent figure. For this, you should create copy of list as follows :






```
b = list(a)
```

Now *a* and *b* are separate lists. See below :

```
>>> a = [1, 2, 3]
>>> b = list(a)
>>> a[1] = 5
>>> a
[1, 5, 3]
>>> b
[1, 2, 3]
>>>
```

Now, list b is not reflecting the changed value. List b is independent of list a.

Thus, true independent copy of a list is made via **list( )** method; assigning a list to another identifier just creates an alias for same list.

7.5 LIST FUNCTIONS AND METHODS

Python also offers many built-in functions and methods for list manipulation. You have already worked with one such method **len( )** in earlier chapters. In this chapter, you will learn about many other built-in powerful list methods of Python used for list manipulation.

Every list object that you create in Python is actually an instance of List class (you need not do anything specific for this; Python does it for you – you know *built-in*). The list manipulation methods that are being discussed below can be applied to list as per following syntax :

```
<listObject>.<method name>()
```

In the following examples, we are referring to <listObject> as *List* only (no angle brackets but the meaning is intact *i.e.*, you have to replace *List* with a legal list (*i.e.*, either a *list literal* or a *list object* that holds a list).

Let us now have a look at some useful built-in list manipulation methods.

1. The index method

This function returns the index of first matched item from the list. It is used as per following format :

```
List.index (<item>)
```

For example, for a list L1 = [13, 18, 11, 16, 18, 14],

```
>>> L1.index (18)
1
```

returns the index of first value 18, even if there is another value 18 at index 4.

However, if the given item is not in the list, it raises exception value Error (see below)

```
List.index(33)
Traceback (most recent call last):
  File "<ipython-input-60-038fc9bf9c5c>", line 1, in <module>
    list.index(33)
ValueError: 33 is not in list
```

2. The append method

As you have read earlier in section 7.4 that the *append( )* method **adds an item to the end of the list**. It works as per following syntax :

```
List.append(<item>)
```

– *Takes exactly one element and returns no value*

For example, to add a new item "yellow" to a list containing colours , you may write :

```
>>> colours = ['red', 'green', 'blue']
>>> colours.append('yellow')
>>> colours
['red', 'green', 'blue', 'yellow']
```

See the item got added at the end of the list

The *append( )* **does not return the new list**, just modifies the original. To understand this, consider the following example :

```
>>> lst = [1, 2, 3]
>>> lst2 = lst.append(12)
>>> lst2
>>> lst
[1, 2, 3, 12]
```

Trying to assign the result of **lst.append()** *to another list*

See, **lst2** *is empty as* **append()** *did not return any value*

But list **lst**, *on which* **append()** *was applied, has the newly added element*

3. The extend method

The *extend( )* method is also used for adding multiple elements (given in the form of a list) to a list. But it is different from *append( )*. First, let us understand the working of *extend( )* function then we'll talk about the difference between these *two* functions.

The *extend( )* function works as per following format :

```
List.extend(<list>)
```

– Takes exactly one element (a list type) and returns no value

That is *extend( )* takes a list as an argument and appends all of the elements of the argument list to the list object on which *extend( )* is applied.

Consider following example :

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> t1
['a', 'b', 'c', 'd', 'e']
>>> t2
['d', 'e']
```

Extend the list t1, by adding all elements of t2

See the elements of list t2 are added at the end of list t1

But list t2 remains unchanged.

The above example left list t2 unmodified. Like *append(), extend()* also does not return any value.

Consider following example :

```
>>> t3 = t1.extend(t2)
>>> t3
```

Trying to assign the result of lst.extend() to another list

See, t3 is empty as extend() did not return any value. But list t1 is extended with elements of list t2.

Difference between append( ) and extend( )

While append( ) function adds one element to a list, *extend( )* can add multiple elements from a list supplied to it as argument. Consider the following example that will make it clear to you :

```
>>> t1 = [ 1, 3, 5 ]
>>> t2 = [ 7, 8 ]
>>> t1.append(10)
>>> t1
[1, 3, 5, 10]
>>> t1.append(12, 14)
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    t1.append(12, 14)
TypeError: append() takes exactly one argument (2 given)
```

append( ) can take single object – whether single item or single sequence

Notice when we try to give multiple arguments to append(), what happens?

Notice the error reported.

```
>>> t1.append([12,14])
>>> t1
[1, 3, 5, 10, [12, 14]]

>>> len(t1)
5

>>> t2.extend(10)

Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    t2.extend(10)
TypeError: 'int' object is not iterable

>>> t2.extend([12, 14])
>>> t2
[7, 8, 12, 14]
>>> t3 = [20, 40]
>>> t2.extend(t3)
>>> t2
[7, 8, 12, 14, 20, 40]
>>> len (t2)
6
```

If you try to enter multiple elements in the form of a list (notice square brackets [ ])

Python will add the complete list as one element. Check the length of the list, it will show that now it has 5 elements, because newly added list [12, 14] is treated as single object.

The extend() cannot add a single element; it requires a list as argument.

Either provide list value or a list object

Notice, extend( ) added the individual elements of the list argument t3. t2, which earlier had 4 elements has now 6 elements – including 2 elements of list t3.

NOTE

The append( ) can add a single element to the end of a list while extend( ) can add argument-list's all elements to the end of the list.

After append( ), the length of the list will increase by 1 element only ; after extend( ), the length of the list will increase by the length of inserted list.

4. The insert method

The *insert( )* method is also an insertion method for lists, like append and extend methods. However, both *append( )* and *extend( )* insert the element(s) at the end of the list. If you want to insert an element somewhere in between or any position of your choice, both *append( )* and *extend( )* are of no use. For such a requirement *insert( )* is used.

The *insert( )* function inserts an item at a given position. It is used as per following syntax:

`List.insert( <pos>, <item>)`

– *Takes two arguments and returns no value.*

The first argument **<pos> is the index of the element before which the second argument <item> is to be added.**

Consider the following example:

```
>>> t1 = [ 'a', 'e', 'u']
>>> t1.insert(2, 'i')          # inset element 'i' at index 2.
>>> t1
['a', 'e', 'i', 'u']
```

See element 'i' inserted at index 2

For function insert( ), we can say that :

`list.insert(0, x)` will insert element *x* at the front of the list *i.e.*, at index 0 (zero)

`list.insert( len(a), x )` will insert element *x* at the end of the list – index equal to length of the list ; thus it is equivalent to list.append(x).
If index is greater than *len(list)*, the object is simply appended.

If, however, index is less than zero and not equal to any of the valid negative indexes of the list (depends on the size of the list), the object is *prepended*, *i.e.*, added in the beginning of list *e.g.*, for a list t1 = ['a', 'e', 'i', 'u'], if we do :

```
>>> t1.insert(-9, 'k')          # valid negative indexes are -1, -2, -3, -4
>>> t1
['k', 'a', 'e', 'i', 'u']       # list prepended with element 'k'
```

5. The pop method

You have read about this method earlier. The *pop( )* is used to remove the item from the list. It is used as per following syntax :

```
List.pop(<index>)               # <index is optional argument
```

– *Takes one optional argument and returns a value – the item being deleted*

Thus, pop( ) removes an element from the given position in the list, and return it. If no index is specified, pop( ) removes and returns the last item in the list. Consider some examples :

```
>>>t1
['k', 'a', 'e', 'i', 'p', 'q', 'u']
>>>ele1 = t1.pop(0)
>>>ele1
'k'
```

*Remove element at index 0 i.e., first element and store it in **ele1***

The removed element

```
>>>t1
['a', 'e', 'i', 'p', 'q', 'u']
>>>ele2 = t1.pop()
>>>ele2
'u'

>>>t1
['a', 'e', 'i', 'p', 'q']
```

List after removing first element

No index specified, it will remove the last element

The pop( ) method raises an exception(runtime error) if the list is already empty. Consider this :

```
>>> t2 = []                     # empty list
>>> t2.pop()
Traceback (most recent call last):
  File "<pyshell#31>", line 1, in <module>
    t2.pop()
IndexError: pop from empty list
```

Trying to delete from empty list raises exception

6. The remove method

While pop( ) removes an element whose position is given, what if you know the value of the element to be removed, but you do not know its index or position in the list ? Well, Python thought it in advance and made available the remove( ) method.

The remove( ) method removes the first occurrence of given item from the list. It is used as per following format :

```
List.remove ( <value> )
```
– *Takes one essential argument and does not return anything*

The remove( ) will report an error if there is no such item in the list. Consider some examples :

```
>>> t1 = ['a', 'e', 'i', 'p', 'q', 'a', 'q', 'p']
>>> t1.remove('a')
>>> t1
['e', 'i', 'p', 'q', 'a', 'q', 'p']
>>> t1.remove('p')
>>> t1
['e', 'i', 'q', 'a', 'q', 'p']
>>> t1.remove('k')
Traceback (most recent call last):
  File "<pyshell#45>", line 1, in <module>
    t1.remove('k')
ValueError: list.remove(x): x not in list
```

First occurrence of 'a' is removed from the list

First occurrence of 'p' is removed from the list

Trying to remove an element which is not present in the list, raises error.

NOTE

The pop method removes an individual item and returns it, while remove searches for an item, and removes the first matching item from the list.

7. The clear method

This method removes all the items from the list and the list becomes empty list after this function. This function returns nothing. It is used as per following format.

```
List.clear( )
```

For instance, if you have a list L1 as

```
>>> L1 = [2, 3, 4, 5]
>>> L1.clear( )
>>> L1
[ ]
```

it will remove all the items from list L1.

Now the L1 is an empty list.

Unlike **del <lstname>** statement, **clear( )** removes only the elements and not the list element. After clear( ), the list object still exists as an empty list.

8. The count method

This function returns the count of the item that you passed as argument. If the given item is not in the list, it returns zero. It is used as per following format :

```
List.count(<item> )
```

For instance, for a list L1 = [13, 18, 20, 10, 18, 23]

```
>>> L1.index [18]
2
>>> L1.index(28)
0
```

returns 2 as there are two items with value 18 in the list.

No item with value 28 in the list, hence it returned 0 (zero)

Check Point 7.1

1. Why are lists called mutable types ?
2. What are immutable counterparts of lists ?
3. What are different ways of creating lists ?
4. What values can we have in a list? Do they all have to be the same type ?
5. How are individual elements of lists are accessed and changed ?
6. How do you create the following lists ?
   (a) [4, 5, 6]
   (b) [-2, 1, 3]
   (c) [-9, -8, -7, -6, -5]
   (d) [-9, -10, -11, -12]
   (e) [0, 1, 2]
7. If a = [5, 4, 3, 2, 1, 0] evaluate the following expressions:
   (a) a[0]
   (b) a[-1]
   (c) a[a[0]]
   (d) a[a[-1]]
   (e) a[a [a [a [2] +1]]]
8. Can you change an element of a sequence? What if the sequence is a string? What if the sequence is a list?
9. What does a + b amounts to if a and b are lists ?
10. What does a * b amounts to if a and b are lists ?
11. What does a + b amounts to if a is a list and b = 5 ?
12. Is a string the same as a list of characters ?
13. Which functions can you use to add elements to a list ?
14. What is the difference between append( ) and insert( ) methods of list?
15. What is the difference between pop( ) and remove( ) methods of list?
16. What is the difference between append( ) and extend( ) methods of list?
17. How does the sort( ) work? Give example.
18. What does list.clear( )function do ?

9. The reverse method

The *reverse( )* reverses the items of the list. This is done "in place", *i.e.*, it does not create a new list. The syntax to use reverse method is :

```
List.reverse()
```

– *Takes no argument, returns no list ; reverses the list 'in place' and does not return anything.*

For example,

```
>>> t1
['e', 'i', 'q', 'a', 'q', 'p']
>>> t1.reverse()
>>> t1                          <-- The reversed list
['p', 'q', 'a', 'q', 'i', 'e']
>>> t2 = [3, 4, 5]
>>> t3 = t2.reverse( )
>>> t3                          <-- t3 stores nothing as reverse( ) does not return anything.
>>> t2
[5, 4, 3]
```

10. The sort method

The *sort( )* function sorts the items of the list, by default in increasing order. This is done "in place", *i.e.*, it does not create a new list. It is used as per following syntax :

```
List.sort()
```

For example,

```
>>> t1 = ['e', 'i', 'q', 'a', 'q', 'p']
>>> t1.sort()
>>> t1                          <-- Sorted list in default ascending order
['a', 'e', 'i', 'p', 'q', 'q']
```

Like reverse( ), sort( ) also performs its function and does not return anything.

To sort a list in decreasing order using sort( ), you can write :

```
>>> List.sort(reverse = True)
```

NOTE

Please note, *sort( )* won't be able to sort the values of a list if it contains a complex number as an element, because Python has *no ordering relation defined for complex numbers.*

Program 7.3 Program to find minimum element from a list of element along with its index in the list.

```
lst = eval(input("Enter list : "))
length = len(lst)
min_ele = lst[0]
min_index = 0
for i in range(1, length-1) :
    if lst[i] < min_ele :
        min_ele = lst[i]
        min_index = i
print("Given list is : ", lst)
print("The minimum element of the given list is :")
print(min_ele, "at index", min_index)
```

```
Enter list : [2, 3, 4, -2, 6, -7, 8, 11, -9, 11]
Given list is : [2, 3, 4, -2, 6, -7, 8, 11, -9, 11]
The minimum element of the given list is :
-9 at index 8
```

Program 7.4 Program to calculate mean of a given list of numbers.

```
lst = eval(input("Enter list : "))
length = len(lst)
mean = sum = 0
for i in range(0, length-1) :
    sum += lst[i]
mean = sum /length
print("Given list is : ", lst)
print("The mean of the given list is :", mean)
```

```
Enter list : [7, 23, -11, 55, 13.5, 20.05, -5.5]
Given list is : [7, 23, -11, 55, 13.5, 20.05, -5.5]
The mean of the given list is : 15.364285714285714
```

Program 7.5 Program to search for an element in a given list of numbers.

```
lst = eval(input("Enter list :"))
length = len(lst)
element = int(input("Enter element to be searched for :"))
for i in range(0, length-1) :
    if element == lst[i] :
        print(element, "found at index", i)
        break
else :                          # else of for loop
    print(element, "not found in given list")
```

Two sample runs of above program are being given below :

```
Enter list : [2, 8, 9, 11, -55, -11, 22, 78, 67]
Enter element to be searched for : -11
-11 found at index 5
```

```
Enter list : [2, 8, 9, 11, -55, -11, 22, 78, 67]
Enter element to be searched for : -22
-22 not found in given list
```

Program 7.6 Program to count frequency of a given element in a list of numbers.

```
lst = eval(input("Enter list :"))
length = len(lst)
element = int(input("Enter element :"))
count = 0
for i in range(0, length-1):
    if element == lst[i]:
        count += 1
if count == 0 :
    print(element, "not found in given list")
else :
    print(element,"has frequency as", count, "in given list")
```

Sample run of above program is given below :

```
Enter list : [1, 1, 1, 2, 2, 3, 4, 2, 2, 5, 5, 2, 2, 5]
Enter element : 2
2 has frequency as 6 in given list
```

Program 7.7 Program to find frequencies of all elements of a list. Also, print the list of unique elements in the list and duplicate elements in the given list.

```
lst = eval(input("Enter list :"))
length = len(lst)
uniq = []                    # list to hold unique elements
dupl = []                    # list to hold duplicate elements
count = i = 0
while i < length :
    element = lst[i]
    count = 1                # count as 1 for the element at lst[i]
```

```
            if element not in uniq and element not in dupl:
                i += 1
                for j in range(i, length):
                    if element == lst[j]:
                        count += 1
                #when inner loop - for loop ends
                else:
                    print("Element", element, "frequency:", count)
                    if count == 1:
                        uniq.append(element)
                    else:
                        dupl.append(element)
                            #when element is found in uniq or dupl lists
            else:
                i += 1
        print("Original list", lst)
        print("Unique elements list", uniq)
        print("Duplicates elements list", dupl)
```

These print( ) statements are not part of while loop.

```
Enter list : [2, 3, 4, 5, 3, 6, 7, 3, 5, 2, 7, 1, 9, 2]
Element 2 frequency: 3
Element 3 frequency: 3
Element 4 frequency: 1
Element 5 frequency: 2
Element 6 frequency: 1
Element 7 frequency: 2
Element 1 frequency: 1
Element 9 frequency: 1
Original list [2, 3, 4, 5, 3, 6, 7, 3, 5, 2, 7, 1, 9, 2]
Unique elements list     [4, 6, 1, 9]
Duplicates elements list [2, 3, 5, 7]
```

PriP LISTS IN PYTHON – II — Progress In Python 7.2

This PriP session works on the objective of practicing List Manipulation functions.

Please check the practical component-book – Progress in Computer Science with Python and fill it there in PriP 7.2 under Chapter 7 after practically doing it on the computer.

>>>❖<<<

With this we have come to the end of our chapter. Let us quickly revise what we have learnt so far.

LET US REVISE

- *Lists are mutable sequences of Python i.e., you can change elements of a list in place.*
- *To create a list, put a number of comma-separated expressions in square brackets.*
- *The empty square brackets i.e., [ ] indicate an empty list.*
- *Lists index their elements just like strings, i.e., two way indexing.*
- *Lists are stored in memory exactly like strings, except that because some of their objects are larger than others, they store a reference at each index instead of single character as in strings.*
- *Lists are similar to strings in many ways like **indexing, slicing** and **accessing individual elements** but they are different in the sense that Lists are **mutable** while strings are not.*
- *Function **len(L)** returns the number of items (count) in the list L.*
- *Membership operator **in** tells if an element is present in the list or **not** and **not in** does the opposite.*
- *The + operator adds one list to the end of another. The * operator repeats a list.*
- *List slice is an extracted part of a list; list slice is a list in itself.*
- ***L[start:stop]** creates a **list slice** out of list L with elements falling between indexes **start** and **stop**, not including **stop**.*
- *Common list manipulation functions are : append( ), insert( ), extend( ), sort( ), remove( ), reverse( ) and pop( ).*

Solved Problems

1. *How are lists different from strings when both are sequences ?*

Solution. The lists and strings are different in following ways :

(*i*) The lists are mutable sequences while strings are immutable.

(*ii*) In consecutive locations, strings store the individual characters while list stores the references of its elements.

(*iii*) Strings store single type of elements – all characters while lists can store elements belonging to different types.

2. *What are nested lists ?*

Solution. When a list is contained in another list as a member-element, it is called nested list, *e.g.*,

$a = [2, 3, [4, 5]]$

The above list a has three elements – an integer 2, an integer 3 and a list [4, 5], hence it is nested list.

3. *What does each of the following expression evaluate to ?*

Solution. Suppose that *L* is the list

["These", "are", "a", ["few", "words"], "that", "we", "will", "use"].

(*a*)
```
L[3:4]
L[3:4][0]
L[3:4][0][1]
L[3:4][0][1][2]
```

(b) `"few" in L`
(c) `"few" in L[3]`
(d) `[L[1]] + L[3]`
(e) `L[4:]`
(f) `L[0::2]`

Solution.

(a)
```
L[3:4] = [['few', 'words']]
```
it is a nested list
```
L[3:4][0] = ['few', 'words']
L[3:4][0][1] = 'words'
L[3:4][0][1][2] = 'r'
```

(b) **False.** The string "few" is part of a list which is an element of the list *L*, the literal string is not an element of *L*.

(c) **True.** String "few" is part of list, which is at the index 3 of List *L i.e.*, *L*[3].

(d)
```
[L[1]] = [ 'are' ]
L[3] = [ 'few', 'words']
[L[1]] + L[3] = ['are', 'few', 'words']
```

(e) `[ 'that', 'we', 'will', 'use']`

(f) `[ 'These', 'a', 'that', 'will']`

4. *What does each of the following expressions evaluate to?*

Suppose that L is the list

["These", ["are", "a"], ["few", "words"], "that", "we", "will", "use"].

(a) `len (L)`
(b) `L[3:4] + L[1:2]`
(c) `"few" in L[2:3]`
(d) `"few" in L[2:3] [0]`
(e) `"few" in L[2]`
(f) `L[2][1:]`
(g) `L[1] + L[2]`

Solution.

(a) `7`

(b)
```
L[3:4] = ['that']
L[1:2] = [['are', 'a']]
L[3:4] + L[1:2] = ['that', ['are', 'a']]
```

(c) **False.** The string "few" is not an element of this range. *L*[2 : 3] returns a list of elements from *L*–> [['few', 'words']], this is a list with one element, a list.

(d) **True.** As 0th element of L[2 : 3] list is a list ["few", "words"] which has a member "few" in it.

(e) **True.** *L*[2] returns the list ['few', 'words'], "few" is an element of this list.

(f)
```
L[2] = ['few', 'words']
L[2][1:] = ['words']
```

(g)
```
L[1] = ['are', 'a']
L[2] = ['few', 'words']
L[1] + L[2] = ['are', 'a', 'few', 'words'
```

5. *L is a non-empty list of ints. Print the smallest and largest integer in L. Write the code without using a loop.*

Solution.

```
length = len(L)
L.sort()
print ("smallest :", L[0])
print ("largest:", L[length-1])
```

6. *Write the most appropriate list method to perform the following tasks.*

(a) *Delete a given element from the list.* (b) *Delete 3rd element from the list.*

(c) *Add an element in the end of the list.* (d) *Add an element in the beginning of the list.*

(e) *Add elements of a list in the end of a list.*

Solution. (a) remove( ) (b) pop( ) (c) append( ) (d) insert( ) (e) extend( )

7. *Given a list of integers, L, write code to add the integers and display the sum.*

Solution.

```
pos = 0                          # start of list
sum = 0                          # initial sum
while pos < len (L) :            # loop through entire list
    sum = sum + L[pos]           # add current item to sum
    pos = pos + 1                # move to next item in ilst
print (sum )                     # the answer
```

8. *Given a list of integers, L, write code to calculate and display the sum of all the odd numbers in the list.*

Solution.

```
pos = 0                          # start of list
sum = 0                          # initially no sum
while pos < len (L) :            # loop through list
    if L[pos] % 2 == 1 :         # this is an odd number
        sum = sum + L[pos]       # so add it
    pos = pos + 1                # next item in list
print (sum )                     # the answer
```

9. *Examine the following code :*

```
numlist = eval(input("Enter list:"))
pos = 0
odds = evens = 0
length = len (num list)
while pos < length :
    if numlist[pos] % 2 == 0 :
        evens = evens + 1
    else :
        odds = odds + 1
    pos = pos + 1
if odds > evens :
    print ("Balanced oddity")
```

(a) *What is this program calculating ?* (b) *What does the program for the list* [1, 5, 2, 3, 6, 6, 9] ?

(c) *What does the program print for the list* [2, 5, 2, 3, 6, 6, 9] ? *How can we fix this ?*

Solution. (a) The given code is checking whether a list of numbers is 'mostly odd' – *i.e.*, whether there are more odd numbers than even numbers in the list.

(b) Balanced oddity

(c) The code prints nothing. This is because there is only one print statement, the fix is :

```
if odds > evens :
    print ("Balanced oddity")
else :
    print ("Unbalanced oddity")
```

10. *Given two lists, write a program that prints "Overlapped" if they have at least one member in common, otherwise prints "Separated". You may use the in operator, but for the sake of the exercise, write it using loops.*

Solution.

```
listA = eval(input("Enter list1 :"))
listB = eval(input("Enter list2 :"))
len1 = len(listA)
len2 = len(listB)
for a in range(len1) :
  ele = listA[a]
  if ele in listB :
      print("Overlapped")
      break
else :                    #loop ends normally, not because of break
  print("Separated")
```

11. *Write a program to find the second largest number of a list of numbers.*

Solution.

```
lst = eval(input("Enter list :"))
length = len(lst)
biggest = secondbiggest = lst[0]        # though logically not fair
for i in range(1, length):
  if lst[i] > biggest:
      secondbiggest = biggest
      biggest = lst[i]
  elif lst[i] > secondbiggest:
      secondbiggest = lst[i]
print("Largest number of the list :", biggest)
print("Second Largest number of the list :", secondbiggest )
```

12. *Write a program that inputs a list of numbers and shifts all the zeros to right and all non-zero numbers to left of the list.*

Solution.

```
lst = eval(input("Enter list :"))
length = len(lst)
end = length-1
print("Original list : ", lst)
i = 0
while (i <= end):
  ele = lst[i]
  if ele == 0 :
      for j in range(i, end):
          lst[j] = lst[j+1]
      else:
          lst[end] = 0
          end -= 1
  if lst[i]!= 0:
      i += 1                    # if after shifting, a consecutive 0 shifted to current place
print("Zero shifted : ", lst)
```

8 Tuples

In This Chapter

8.1 INTRODUCTION

The Python tuples are sequences that are used to store a tuple of values of any type. You have learnt in earlier chapters that Python **tuples are immutable** *i.e.*, you cannot change the elements of a tuple *in place* ; Python will create a fresh tuple when you make changes to an element of a tuple. Tuple is a type of sequence like *strings* and *lists* but it differs from them in the way that lists are *mutable* but strings and tuples are *immutable*.

This chapter is dedicated to basic tuple manipulation in Python. We shall be talking about creating and accessing tuples, various tuple operations and tuple manipulations through some built-in functions.

8.2 CREATING AND ACCESSING TUPLES

A tuple is a standard data type of Python that can store a sequence of values belonging to any type. The *Tuples* are depicted through parentheses *i.e.*, round brackets, *e.g.*, following are some tuples in Python :

```
()                          # tuple with no member, empty tuple
(1, 2, 3)                   # tuple of integers
(1, 2.5, 3.7, 9)            # tuple of numbers (integers and floating point)
('a', 'b', 'c')             # tuple of characters
('a', 1, 'b', 3.5, 'zero')  # tuple of mixed value types
('One', 'Two', 'Three')     # tuple of strings
```

NOTE

Tuples are ***Immutable sequences*** of Python *i.e.*, you cannot change elements of a tuple in place.

Before we proceed and discuss how to create tuples, one thing that must be clear is that **Tuples are immutable** (*i.e.*, non-modifiable) *i.e.*, you cannot change elements of a tuple in place.

8.2.1 Creating Tuples

Creating a tuple is similar to list creation, but here you need to put a number of expressions in parentheses. That is, use round brackets to indicate the *start* and *end* of the tuple, and separate the items by commas. For example :

```
(2, 4, 6)
('abc', 'def')
(1, 2.0, 3, 4.0)
()
```

Thus to create a tuple you can write in the form given below :

```
T = ()
T = (value, ...)
```

This construct is known as a **tuple display construct.**

Consider some more examples :

1. The Empty Tuple

The empty tuple is ( ). It is the tuple equivalent of 0 or ''. You can also create an empty tuple as:

```
T = tuple()
```

It will generate an empty tuple and name that tuple as *T*.

2. Single Element Tuple

Making a tuple with a single element is tricky because if you just give a single element in round brackets, Python considers it a value only, *e.g.*,

```
>>> t = (1)
>>> t
1
```

(1) was treated as an integer expression, hence t stores an integer 1, not a tuple

To construct a tuple with one element just add a comma after the single element as shown below :

```
>>> t = 3,
>>> t
(3,)
>>> t2 = (4,)
>>> t2
(4,)
```

Both these ways will create tuples

NOTE

Tuples are formed by placing a comma-separated tuple of expressions in parentheses.

3. Long tuples

If a tuple contains many elements, then to enter such long tuples, you can split it across several lines, as given below :

```
sqrs = ( 0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225,
        256, 289, 324, 361, 400, 441, 484, 529, 576, 625 )
```

Notice the opening parenthesis and closing parenthesis appear just in the beginning and end of the tuple.

4. Nested tuples

If a tuple contains an element which is a tuple itself then it is called nested tuple e.g., following is a nested tuple :

```
t1 = (1, 2, (3, 4))
```

The tuple t1 has three elements in it : 1, 2 and (3, 4). The third element of tuple t1 is a tuple itself, hence, t1 is a nested tuple.

Creating Tuples from Existing Sequences

You can also use the built-in tuple type object (*tuple( )*) to create tuples from sequences as per the syntax given below :

```
T = tuple(<sequence>)
```

where <*sequence*> can be any kind of sequence object including strings, lists and tuples.

Python creates the individual elements of the tuple from the individual elements of passed sequence. If you pass in another tuple, the tuple function makes a copy.

Consider following examples :

```
>>> t1 = tuple('hello')
>>> t1
('h', 'e', 'l', 'l', 'o')
```

Tuple t1 is created from another sequence - a string "hello"
It generated individual elements from the individual

```
>>> L = ['W', 'e', 'r' , 't', 'y']
>>> t2 = tuple(L)
>>> t2
('W', 'e', 'r', 't', 'y')
```

Tuple t2 is created from another sequence - a list L
It generated individual elements from the individual elements of the passed list L

You can use this method of creating tuples of single characters or single digits via keyboard input. Consider the code below :

```
t1 = tuple( input('Enter tuple elements:'))
Enter tuple elements : 234567
>>> t1
('2', '3', '4', '5', '6', '7')
```

See, it created the elements of tuple t1 using each of the character inputs

See, with tuple( ) around input( ), even if you not put parenthesis, it will create a tuple using individual characters as elements. But most commonly used method to input tuples is **eval(input())** as shown below :

```
tuple = eval(input("Enter tuple to be added:"))
print ("Tuple you entered :", tuple)
```

when you execute it, it will work somewhat like :

```
Enter tuple to be added: (2, 4, "a", "hjkjl", [3,4])
Tuple you entered : (2, 4, "a", "hjkjl", [3,4])
```

If you are inputting a tuple with eval( ), then make sure to enclose the tuple elements in parenthesis.

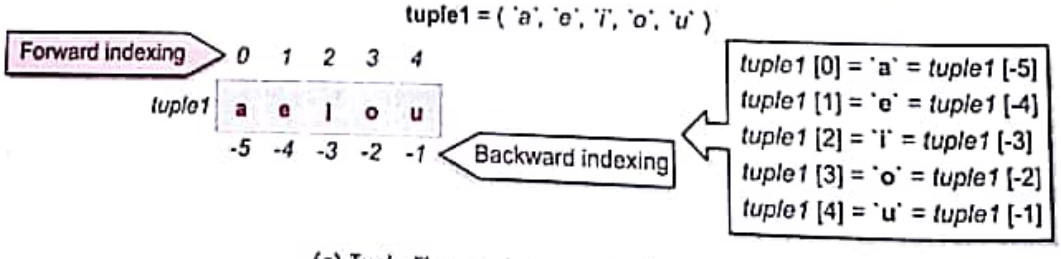
Please note sometimes (not always) eval( ) does not work in Python shell. At that time, you can run it through a script too.

8.2.2 Accessing Tuples

Tuples are immutable (non-editable) sequences having a progression of elements. Thus, other than editing items, you can do all that you can do with lists. Thus like lists, you can access its individual elements. Before we talk about that, let us learn about how elements are indexed in tuples.

Similarity with Lists

Tuples are very much similar to lists except for the mutability. In other words, *Tuples* are immutable counter-parts of lists. Thus, like lists, tuple-elements are also indexed, *i.e.*, forward indexing as 0, 1, 2, 3,... and backward indexing as –1, –2, –3,... [See Fig 8.1(*a*)]



(*a*) Tuple Elements' two way indexing

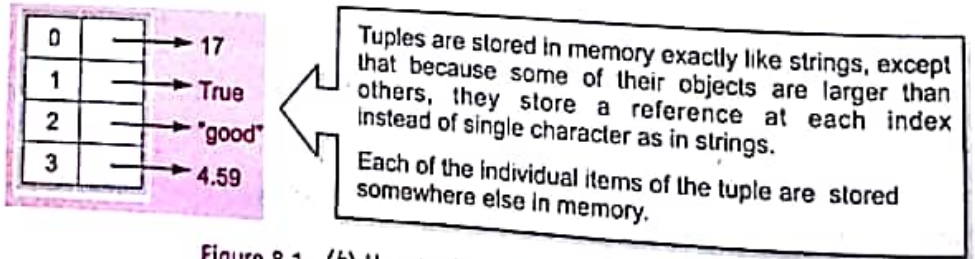


Figure 8.1 (*b*) How tuples are internally organized

Thus, you can access the tuple elements just like you access a list's or a string's elements e.g., **Tuple[i]** will give you element at i^{th} index ; **Tuple[a:b]** will give you elements between indexes *a* to *b-1* and so on. Put in other words, tuples are similar to lists in following ways :

- **Length.** Function **len(T)** returns the number of items (count) in the tuple T.
- **Indexing and Slicing**

  T[*i*] returns the item at index *i* (the first item has index 0),

  T[*i* : *j*] returns a new tuple, containing the objects between *i* and *j* excluding index *j*,

  T[*i* : *j* : *n*] returns a new tuple containing every *n*th item from index *i* to *j*, excluding index *j* . Just like lists.
- **Membership operators.** Both 'in' and 'not in' operators work on *Tuples* just like they work for other sequences. That is, in tells if an element is present in the tuple or not and not in does the opposite.
- **Concatenation and Replication operators + and *.** The + operator adds one tuple to the end of another. The * operator repeats a tuple. We shall be talking about these two operations in a later section 8.3 – *Tuple Operations*.

Accessing Individual Elements

As mentioned, the individual elements of a tuple are accessed through their indexes given in square brackets. Consider the following examples :

```
>>> vowels = ('a', 'e', 'i', 'o', 'u')
>>> vowels[0]
'a'
>>> vowels[4]
'u'
>>> vowels[-1]
'u'
>>> vowels[-5]
'a'
```

NOTE

While accessing tuple elements, if you pass in a negative index, Python adds the length of the tuple to the index to get element's forward index.

Recall that like strings, if you pass in a negative index, Python adds the length of the tuple to the index to get its forward index. That is, for a 6-element tuple *T*, *T*[−5] will be internally computed as :

T[−5 + 6] = *T*[1], and so on.

Difference from Lists

Although tuples are similar to lists in many ways, yet there is an important difference in mutability of the two. Tuples are not mutable, while lists are. You cannot change individual elements of a tuple in place, but lists allow you to do so.

That is, following statement is fully valid for lists (BUT not for tuples). That is, if we have a list L and a tuple T, then

```
L[i] = element
```

is VALID for Lists. BUT

```
T[i] = element
```

is INVALID as you cannot perform item-assignment in immutable types.

NOTE

Tuples are similar to lists in many ways like indexing, slicing and accessing individual elements but they are different in the sense that tuples are immutable while lists are not.

Traversing a Tuple

Recall that traversal of a sequence means accessing and processing each element of it. Thus traversing a tuple also means the same and same is the tool for it, *i.e.*, the Python loops. The for loop makes it easy to traverse or loop over the items in a tuple, as per following syntax :

```
for <item> in <Tuple> :
    process each item here
```

For example, following loop shows each item of a tuple T in separate lines :

```
T = ('P', 'y', 't', 'h', 'o', 'n')
for a in T :
    print (T[a])
```

The above loop will produce result as :

```
P
y
t
h
o
n
```

How it works

The loop variable a in above loop will be assigned the Tuple elements, one at a time. So, loop-variable a will be assigned 'P' in first iteration and hence 'P' will be printed; in second iteration, a will get element 'y' and 'Y' will be printed ; and so on.

If you only need to use the indexes of elements to access them, you can use functions *range()* and *len()* as per following syntax :

```
for index in range(len(T)):
    process Tuple[index] here
```

Consider program 8.1 that traverses through a tuple using above format and prints each item of a tuple L in separate lines along with its index.

Program 8.1 *Program to print elements of a tuple ('Hello', "Isn't", 'Python', 'fun', '?') in separate lines along with element's both indexes (positive and negative).*

```
T = ('Hello', 'Isn't', 'Python', 'fun', '?' )
length = len(T)
for a in range(length) :
    print ('At indexes', a, 'and ', (a - length), 'element :', T[a])
```

```
At indexes 0 and -5 element : Hello
At indexes 1 and -4 element : Isn't
At indexes 2 and -3 element : Python
At indexes 3 and -2 element : fun
At indexes 4 and -1 element : ?
```

Now that you know about tuple traversal, let us talk about tuple operations.

8.3 TUPLE OPERATIONS

The most common operations that you perform with tuple include joining tuples and slicing tuples. In this section, we are going to talk about the same.

8.3.1 Joining Tuples

Joining two tuples is very easy just like you perform addition, literally ;) . The + operator, the concatenation operator, when used with two tuples, joins two tuples.

Consider the example given below :

```
>>> tpl1 = ( 1, 3, 5)
>>> tpl2 = (6, 7, 8 )
>>> tpl1 + tpl2
(1, 3, 5, 6, 7, 8)
```

The + operator concatenates two tuples and creates a new tuple

As you can see that the resultant tuple has firstly elements of first tuple *lst1* and followed by elements of second tuple *lst2*. You can also join two or more tuples to form a new tuple, *e.g.,*

```
>>> tpl1 = (10, 12, 14)
>>> tpl2 = (20, 22, 24)
>>> tpl3 = (30, 32, 34)
>>> tpl = tpl1 + tpl2 + tpl3
>>> tpl
(10, 12, 14, 20, 22, 24, 30, 32, 34)
```

The + operator is used to concatenate three individual tuples to get a new combined tuple tpl

The + operator when used with tuples requires that **both the operands must be of tuple types.** You cannot add a number or any other value to a tuple. For example, following expressions will result into error :

```
tuple + number
tuple + complex-number
tuple + string
tuple + list
```

Consider the following examples :

See errors generated when anything other than a tuple is added to a tuple

```
>>> tpl1 = (10, 12, 14)
>>> tpl1 + 2
Traceback (most recent call last):
  File "<pyshell#42>", line 1, in <module>
    tpl1+2
TypeError: can only concatenate tuple (not "int") to tuple
>>> tpl1 + "abc"
Traceback (most recent call last):
  File "<pyshell#44>", line 1, in <module>
    tpl1 + "abc"
TypeError: can only concatenate tuple (not "str") to tuple
```

IMPORTANT

Sometimes you need to concatenate a tuple (say tpl) with another tuple containing only one element. In that case, if you write statement like :

```
>>> tpl + (3)
```

Python will return an error like :

```
Traceback (most recent call last):
  File "<pyshell#14>", line 1, in <module>
    tuple + (3)
TypeError: can only concatenate tuple (not "int") to tuple
```

NOTE

A single value in ( ) is treated as single value not as tuple. That is, expressions (3) and ('a') are integer and string respectively but (3,) and ('a',) are examples of tuples with single element.

The reason for above error is that a number enclosed in ( ) is considered number only. To make it a tuple with just one element, just add a comma after the only element, *i.e.*, make it (3,). Now Python won't return any error and successfully concatenate the two tuples.

```
>>> tpl =(10, 12, 14, 20, 22, 24, 30, 32, 34)
>>> tpl + (3, )
(10, 12, 14, 20, 22, 24, 30, 32, 34, 3)
```

Single element tuple

Repeating or Replicating Tuples

Like strings and lists, you can use * operator to replicate a tuple specified number of times, *e.g.*, if tpl1 is (1, 3, 5), then

```
>>> tpl1 * 3
(1, 3, 5, 1, 3, 5, 1, 3, 5)
```

*The * operator repeats a tuple specifies number of times and creates a new tuple*

NOTE

When used with tuples, the + operator requires both the operands as tuple-types; and the * operator requires a tuple and an integer as operands.

8.3.2 Slicing the Tuples

Tuple slices, like list-slices or string slices are the sub parts of the tuple extracted out. You can use indexes of tuple elements to create *tuple slices* as per following format :

```
seq = T[start:stop]
```

The above statement will create a *tuple slice* namely seq having elements of tuple T on indexes *start, start+1, start+2, ..., stop-1*. Recall that index on last limit is not included in the tuple slice. The *tuple slice* is a tuple in itself that is you can perform all operations on it just like you perform on tuples.

Consider the following example :

```
>>> tpl = (10, 12, 14, 20, 22, 24, 30, 32, 34)
>>> seq = tpl [ 3:-3]
>>> seq
(20, 22, 24)
```

For normal indexing, if the resulting index is outside the tuple, Python raises an **IndexError exception**. Slices are treated as boundaries instead, and the result will simply contain all items between the boundaries. For the *start* and *stop* given beyond tuple limits in a tuple slice, Python simply returns the elements that fall between specified boundaries, if any.

NOTE

T[start:stop] creates a tuple slice out of tuple **T** with elements falling between indexes ***start*** and ***stop***, not including ***stop***.

For example, consider the following :

```
>>> tpl =(10, 12, 14, 20, 22, 24, 30, 32, 34)
>>> tpl [3:30]
(20, 22, 24, 30, 32, 34)
>>> tpl [-15 :7]
(10, 12, 14, 20, 22, 24, 30)
```

Giving upper limit way beyond the size of the tuple, but Python return elements from tuple falling in range 3

Giving lower limit much lower, but python returns elements from tuple falling in range – 15 onwards < 7

Tuples also support slice steps too. That is, if you want to extract, not consecutive but every other element of the tuple, there is a way out – the *slice steps*. The *slice steps* are used as per following format :

```
seq = T[start:stop:step]
```

Consider some examples to understand this.

```
>>> tpl
(10, 12, 14, 20, 22, 24, 30, 32, 34)
>>> tpl[0 : 10 : 2 ]
(10, 14, 22, 30, 34)
>>> tpl[2 : 10 : 3 ]
(14, 24, 34)
>>> tpl[:: 3 ]
(10, 20, 30)
```

Include every 2nd element, i.e., skip 1 element in between. Check resulting tuple slice

Include every 3rd element, i.e., skip 2 element in between

No start and stop given. Only step is given as 3. That is, from the entire tuple, pick every 3rd element for the tuple

Consider some more examples :

```
seq = T[::2]      # get every other item, starting with the first
seq = T[5::2]     # get every other item, starting with the
                  # sixth element, i.e., index 5
```

You can use the + and * operators with tuple slices too. For example, if a tuple namely Tp has values as (2, 4, 5, 7, 8, 9, 11, 12, 34), then :

```
>>> Tp[2:5] * 3
(5, 7, 8, 5, 7, 8, 5, 7, 8)
>>> Tp[2:5] + (3, 4)
(5, 7, 8, 3, 4)
```

*See, the * operation has multiplies the tuple slice and not the full tuple*

Here, the + operator has added the given tuple to the tuple slice

NOTE

T[start : stop: step] creates a tuple slice out of tuple **T** with elements falling between indexes ***start*** and ***stop***, not including ***stop***, skipping ***step-1*** elements in between.

Comparing Tuples

You can compare two tuples without having to write code with loops for it. For comparing two tuples, you can using comparison operators, *i.e.*, <, >, ==, != etc. For comparison purposes, Python internally compares individual elements of two tuples, applying all the comparison rules that you have read earlier. Consider following code :

```
>>> a = (2, 3)
>>> b = (2, 3)
>>> a == b
True
>>> c = ('2', '3')
>>> a == c
False
>>> a > b
False
>>> d = (2.0, 3.0)
>>> d > a
False
>>> d == a
True
>>> e = (2, 3, 4)
>>> a < e
True
```

For two tuples to be equal, they must have same number of elements and matching values (recall **int** *and* **float** *with matching values are considered equal)*

Notice for comparison purposes, Python ignored the types of elements and compared values only

You can refer to table 7.1 that discusses non-equality comparisons of two sequences. Elements in tuples are also compared on similar lines.

Unpacking Tuples

Creating a tuple from a set of values is called **packing** and its reverse, *i.e.*, creating individual values from a tuple's elements is called **unpacking**.

Unpacking is done as per syntax :

```
<variable1>, <variable2>, <variable3>, ... = t
```

where the number of variables in the left side of assignment must match the number of elements in the tuple.

For example, if we have a tuple as :

```
t = (1, 2, 'A', 'B')
```

The length of above tuple t is 4 as there are four elements in it. Now to unpack it, we can write :

```
w, x, y, z = t
```

You may even enclosed the variables on LHS in parenthesis. It will give you same result.

NOTE

Forming a tuple from individual values is called packing and creating individual values from a tuple's elements is called unpacking

Python will now assign each of the elements of tuple *t* to the variables on the left side of assignment operator. That is, you can now individually print the values of these variables, somewhat like :

```
print (w)
print (x)
print (y)
print (z)
```

The above code will yield the result as :

```
1
2
'A'
'B'
```

As per *Guido van Rossom*, the creator of Python language —

> *"Tuple unpacking requires that the list of variables on the left has the same number of elements as the length of the tuple."*

Deleting Tuples

The del statement of Python is used to delete elements and objects but as you know that tuples are immutable, which also means that individual elements of a tuple cannot be deleted, *i.e.*, if you give a code like :

```
>>> del t1[2]
```

Then Python will give you a message like :

```
Traceback (most recent call last):
  File "<ipython-input-83-cad7b2ca8ce3>", line 1, in <module>
    del t1[2]
TypeError: 'tuple' object doesn't support item deletion
```

But you can delete a complete tuple with del statement as :

```
del <tuple_name>
```

For example,

```
>>> t1 = (5, 7, 3, 9, 12)
>>> t1
(5, 7, 3, 9, 12)
>>> del t1
>>> print(t1)
Traceback (most recent call last):
  File "<ipython-input-89-04f730070f35>," line 1, in <module>
    print(t1)
NameError: name 't1' is not defined
```

*See, after using **del** statement on a tuple, if you try to access/ print it, Python gives error as the tuple has been deleted and this objects exists no more*

8.4 TUPLE FUNCTIONS AND METHODS

Python also offers many built-in functions and methods for tuple manipulation. You have already worked with one such method len( ) in earlier chapters. In this section, you will learn about many other built-in powerful tuple methods of Python used for tuple manipulation. Let us now have a look at some useful built-in tuple manipulation methods.

1. The len( ) method

This method returns length of the tuple, *i.e.*, the count of elements in the *tuple*.

Syntax :

```
len(<tuple>)
```

– *Takes tuple name as argument and returns an integer.*

Example :

```
>>> employee = ('John', 10000, 24, 'Sales')
>>> len(employee)
4
```

← *The len( ) returns the count of elements in the tuple*

2. The max( ) method

This method returns the element from the tuple having **maximum value.**

Syntax :

```
max(<tuple>)
```

– *Takes tuple name as argument and returns an object (the element with maximum value).*

Example :

```
>>> tpl = (10, 12, 14, 20, 22, 24, 30, 32, 34)
>>> max(tpl)
34
```

← *Maximum value from tuple* ***tpl*** *is returned*

```
>>> tpl2 = ("Karan", "Zubin", "Zara", "Ana")
>>> max(tpl2)
'Zubin'
```

← *Maximum value from tuple* ***tpl2*** *is returned*

You can use max( ) on lists too to find maximum element from a list. Please note that max( ) applied on sequences like tuples/lists etc. will return a maximum value ONLY IF the sequence contains values of same type. If your tuple (or list) contains values of different datatypes then, it will give you an error stating that mixed type comparison is not possible :

(*a*)
```
>>> ab = (1, 2.5, "1", [3,4], (3,4) )
>>> max(ab)
Traceback (most recent call last):
  File "<ipython-input-54-fd79f7408dd4>", line 1, in <module>
    max(ab)
TypeError: '>' not supported between instances of 'str' and 'float'
```

(b)
```
>>> ab = ([3,4], (3,4) )
>>> max(ab)
Traceback (most recent call last) :
  File "<ipython-input-63-fd79f7408dd4>", line 1, in <module>
    max(ab)
TypeError: '>' not supported between instances of 'tuple' and 'list'
```

3. The min( ) method

This method returns the element from the tuple having minimum value.

Syntax :

```
min(<tuple>)
```

– *Takes tuple name as argument and returns an object (the element with minimum value)*

Example :

```
>>> tpl = (10, 12, 14, 20, 22, 24, 30, 32, 34)
>>> min(tpl)
10
```

← *Maximum value from tuple tpl is returned*

```
>>> tpl2 = ("Karan", "Zubin","Zara", "Ana")
>>> min(tpl2)
'Ana'
```

← *Maximum value from tuple tpl2 is returned*

Like max( ), for min( ) to work, the elements of tuple should be of same type.

4.The index( ) method

The index( ) works with tuples in the same way it works with lists. That is, it returns the index of an existing element of a tuple.

It is used as :

```
<tuplename> . index (<item>)
```

Example :

```
>>> t1 = [3, 4, 5, 6.0]
>>> t1.index(5)
2
```

But if the given item does not exist in tuple, it raises ValueError exception.

5. The count( ) function

The count( ) method returns the count of a member element/object in a given sequence (list/tuple). You can use the count( ) function as per following syntax :

```
<sequence name>.count(<object>)
```

Example :

```
>>> t1 = (2, 4, 2, 5, 7, 4, 8, 9, 9, 11, 7, 2)
>>> t1.count(2)
3
>>> t1.count(7)
2
>>> t1.count(11)
1
```

There are 3 occurrence of element 2 in given tuple , hence count() return 3 here

For an element not in tuple, it returns 0 (zero).

6. The tuple( ) method

This method is actually constructor method that can be used to create tuples from different types of values.

Syntax :

```
tuple(<sequence>)
```

- *Takes an optional argument of sequence type ; Returns a tuple.*
- *With no argument, it returns empty tuple*

Example :

- *Creating empty tuple*

```
>>> tuple()
()
```

- *Creating tuple from a string*

```
>>> t = tuple("abc")
>>> t
('a', 'b', 'c')
```

- *Creating a tuple from a list*

```
>>> t = tuple([1,2,3])
>>> t
(1, 2, 3)
```

- *Creating a tuple from keys of a dictionary*

```
>>> t1 = tuple ( {1:"1", 2:"2"})
>>> t1
(1, 2)
```

As you can notice that, Python has considered only the keys of the passed dictionary to create tuple. But one thing that you must ensure is that the tuple( ) can receive argument of sequence types only, *i.e.*, either a string or a list or even a dictionary. Any other type of value will lead to an error. See below :

```
>>> t = tuple(1)
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    t = tuple(1)
TypeError: 'int' object is not iterable
```

NOTE

With tuple( ), the argument must be a sequence type *i.e.*, a string or a list or a dictionary.

The *tuple( )* and *list( )* are constructors that let you create tuples and lists respectively from passed sequence. This feature can be exploited as a trick to modify tuple, which otherwise is not possible. Following information box talks about the same.

Check Point 8.1

1. Why are tuples called immutable types ?
2. What are mutable counterparts of tuples ?
3. What are different ways of creating tuples ?
4. What values can we have in a tuple ? Do they all have to be the same type* ?
5. How are individual elements of tuples accessed ?
6. How do you create the following tuples ?
   (a) `(4, 5, 6)`
   (b) `(-2, 1, 3)`
   (c) `(-9, -8, -7, -6, -5)`
   (d) `(-9, -10, -11, -12)`
   (e) `(0, 1, 2)`
7. If `a = (5, 4, 3, 2, 1, 0)` evaluate the following expressions :
   (a) `a[0]`
   (b) `a[1]`
   (c) `a[a[0]]`
   (d) `a[a[-1]]`
   (e) `a[a[a[2]+1]]`
8. Can you change an element of a sequence ? What if the sequence is a dictionary ? What if the sequence is a tuple ?
9. What does `a + b` amount to if `a` and `b` are tuples ?
10. What does `a * b` amount to if `a` and `b` are tuples ?
11. What does `a + b` amount to if `a` is a tuple and `b = 5` ?
12. Is a string the same as a tuple of characters ?
13. Can you have an integer, a string, a tuple of integers and a tuple of strings in a tuple ?

Indirectly Modifying Tuples

Tuples are immutable counterparts of lists. If you need to modify the contents often of a sequence of mixed types, then you should go for lists only. However, if you want to ensure that a sequence of mixed types is not accidently changed or modified, you go for tuples. But sometimes you need to retain the sequence as tuple and still need to modify the contents at one point of time, then you can use one of the two methods given here.

(a) Using Tuple Unpacking

Tuples are immutable. To change a tuple, we would need to first unpack it, change the values, and then again repack it :

```
tpl = (11, 33, 66, 99)
```

1. First unpack the tuple :

```
a,b,c,d = tpl
```

2. Redefine or change desired variable say, `c`

```
c = 77
```

3. Now repack the tuple with changed value

```
tpl = (a, b, c, d)
```

(b) Using the constructor functions of lists and tuples i.e., list( ) and tuple( )

There is another way of doing the same as explained below :

```
>>> tpl = ("Anand", 35000, 35, "Admin")
>>> tpl
('Anand', 35000, 35, 'Admin')
```

1. Convert the tuple to list using list() :

```
>>> lst = list(tpl)
>>> lst
['Anand', 35000, 35, 'Admin']
```

2. Make changes in the desired element in the list

```
>>> lst[1] = 45000
>>> lst
['Anand', 45000, 35, 'Admin']
```

3. Create a tuple from the modified list with tuple()

```
>>> tpl = tuple(lst)
>>> tpl
['Anand', 45000, 35, 'Admin']
```

Isn't the trick simple? ☺

PriP TUPLES HANDLING AND MANIPULATION — Progress In Python 8.1

This PriP session aims at strengthening skills related to Tuples handling and manipulation.

Please check the practical component-book – Progress in Computer Science with Python and fill it there in PriP 8.1 under Chapter 8 after practically doing it on the computer.

>>>❖<<<

With this we have come to the end of our chapter. Let us quickly revise what we have learnt so far.

LET US REVISE

- *Tuples are immutable sequences of Python i.e., you cannot change elements of a tuple in place.*
- *To create a tuple, put a number of comma-separated expressions in round brackets.*
- *The empty round brackets i.e., ( ) indicate an empty tuple.*
- *Tuples index their elements just like strings or lists, i.e., two way indexing.*
- *Tuples are stored in memory exactly like strings, except that because some of their objects are larger than others they store a reference at each index instead of single character as in strings.*
- *Tuples are similar to strings in many ways like indexing, slicing and accessing individual elements and they are immutable just like strings are.*
- *Function len(L) returns the number of items (count) in the tuple L.*
- *Membership operator in tells if an element is present in the tuple or not and not in does the opposite.*
- *The + operator adds one tuple to the end of another. The * operator repeats a tuple.*
- *Tuple slice is an extracted part of tuple; tuple slice is a tuple in itself.*
- *T[start:stop] creates a tuple slice out of tuple T with elements falling between indexes start and stop, not including stop.*
- *Common tuple manipulation functions are : len( ), max( ), min( ), and tuple( ).*

Solved Problems

1. *How are tuples different from lists when both are sequences ?*

Solution. The tuples and lists are different in following ways :

- The tuples are immutable sequences while lists are mutable.
- Lists can grow or shrink while tuples cannot.

2. *How can you say that a tuple is an ordered list of objects ?*

Solution. A tuple is an ordered list of objects. This is evidenced by the fact that the objects can be accessed through the use of an ordinal index and for a given index, same element is returned everytime.

3. *Following code is trying to create a tuple with a single item. But when we try to obtain the length of the tuple is, Python gives error. Why? What is the solution ?*

```
>>> t = (6)
>>> len(t)
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    len(t)
TypeError: object of type 'int' has no len()
```

Solution. The syntax for a tuple with a single item requires the item to be followed by a comma as shown below :

```
t = ("a",)
```

Thus, above code is not creating a tuple in t but an integer, on which *len( )* cannot be applied. To create a tuple in t with single element, the code should be modified as :

```
>>> t = (6,)
>>> len(t)
```

4. *What is the length of the tuple shown below ?*

```
t = ((('a', 1), 'b', 'c'), 'd', 2), 'e', 3)
```

Solution. The length of this tuple is 3 because there are just three elements in the given tuple. Because a careful look at the given tuple yields that tuple *t* is made up of :

```
t1 = "a", 1
t2 = t1, "b", "c"
t3 = t2, "d", 2
t = ( t3,"e", 3)
```

5. *Can tuples be nested.?*

Solution. Tuples can contain other compound objects, including lists, dictionaries, and other tuples. Hence, tuples can be nested.

6. *Given a tuple namely cars storing car names as elements :*

```
('Toyota', 'Honda', 'GM', 'Ford', 'BMW', 'Volkswagon', 'Mercedes', 'Ferrai', 'Porshe')
```

Write a program to print names of the cars in the index range 2 to 6, both inclusive.

The output should also include the index in words as shown below :

```
One     Honda
Two     GM
:
```

Solution.

```
cars = ('Toyota','Honda','GM','Ford','BMW,Volkswagon','Mercedes','Ferrari','Porshe')
Number = ("Zero", "One", "Two", "Three", "Four", "Five", "Six", "Seven", "Eight", "Nine")
for x in range(2,7):
  print (Number[x], cars[x])
```

7. *Given a tuple T (1, 2, "a", "b"). There is a list L with some elements. Replace first four elements of list with all four elements of the tuple in single statement. Print the list before and after the list is modified.*

Solution.

```
T = (1, 2, "a", "b")
L = ["hello", "there", "are", "some", "numbers", "coming", "from", "tuple"]
print ("Before Unpacking the list is")
print (L)
L[0], L[1], L[2], L[3] = T
print ("After Unpacking the list is")
print (L)
```

8. *In the answer code of previous question, if we change the square brackets of second line to round brackets, i.e., as*

```
L = (-------)
```

What would be the impact of this change ?

Solution. It will result into error because now L is not a list but a tuple (because brackets changed). A tuple is immutable, hence item assignments are not possible.

So, Python will display like :

```
Before Unpacking the list is
('hello', 'there', 'are', 'some', 'numbers', 'coming', 'from', 'tuple')

Traceback (most recent call last):
  File "C:/Python36/PythonWork/prog.py", line 5, in <module>
    L[0], L[1], L[2], L[3] = T
TypeError: 'tuple' object does not support item assignment
```

9. *Write a Python function secondLargest(T) which takes as input a tuple T and returns the second largest element in the tuple. You can use any of the standard Python functions to obtain your result.*

Solution.

```
T = (23, 45, 34, 66, 77, 67, 70)
maxvalue = max(T)
length  = len(T)
secmax = 0
for a in range(length) :
    if secmax < T[a] < maxvalue :
        secmax = T[a]
print ("Second Largest value is :", secmax)
```

10. *Write a function getPowers(x) that returns a tuple containing x, x^2, x^3 and x^4. Read five integers from the user, and for each integer read, print that value raised to the powers 2, 3 and 4.*

Solution.

```
for n in range(5) :
    x = int(input("Enter Number :"))
    T = ( x, x**2, x**3, x**4)
    print (number, "raised to powers 1, 2, 3, 4 := ", T)
```

9 Dictionaries

In This Chapter

9.1 INTRODUCTION

You must have realized that Python offers many different ways to organize ***collections*** (*i.e.*, a bunch of values in a single "variable") of data items, such *as strings, lists, dictionaries, tuples* etc. Of these you have already worked with *strings, lists* and *tuples* in previous chapters. Now is the time to work with other collection types. So, in this chapter, we shall be talking about dictionaries.

In this chapter, you shall be learning about how data-items are organized in a Python dictionary, how you can access items from a dictionary, various operations that you can perform on a dictionary and various related functions and methods.

9.2 DICTIONARY – KEY:VALUE PAIRS

Among the built-in Python data types is a very versatile type called a *dictionary*. Dictionaries are simply another type of *collection* in Python, but with a twist. Rather than having an *index* associated with each data-item (just like in *lists* or *strings*), Dictionaries in Python have a "**key**" and a "**value of that key**". That is, Python dictionaries are a collection of some **key-value pairs**.

Do not get confused, just read on. *Just like in English dictionaries you can search for a word's meaning, because for each word, there is a meaning associated with it. In the same manner, Python dictionaries have some keys (just like English dictionaries have words) and* ***associated values*** *(just like English dictionaries have associated meanings for every word).*

DICTIONARIES

Dictionaries are mutable, unordered collections with elements in the form of a key:value pairs that associate keys to values.

Dictionaries are containers that associate *keys* to *values*. This is, in a way, similar to lists. In *lists*, you must remember the *index value* of an element from the list, but with the case of dictionaries, you'll have to know the *key* to find the element in the dictionaries.

All this will become clear to you while you go through following sections.

9.2.1 Creating a Dictionary

To create a dictionary, you need to include the *key : value* pairs in curly braces as per following syntax :

```
<dictionary-name> = { <key>:<value>, <key>:<value>... }
```

Following is an example dictionary by the name *teachers* that stores the *names of teachers* as keys and the *subjects being taught* by them as values of respective keys.

```
teachers = { "Dimple" : "Computer Science", "Karen" : "Sociology",
             "Harpreet" : "Mathematics", "Sabah" : "Legal Studies" }
```

Notice that

- the curly brackets mark the beginning and end of the dictionary,
- each entry (*Key* : Value) consists of a pair separated by a colon – the *key* and corresponding *value* is given by writing colon (:) between them,
- the *key-value pairs* are separated by commas (,).

NOTE

Internally, dictionaries are indexed (*i.e.*, arranged) on the basis of *keys*.

As you can see that there are *four* key : value pairs in above dictionary. Following table illustrates *the key-value relationships* in above dictionary *teachers*.

Key-Value pair	Key	Value
"Dimple" : "Computer Science"	"Dimple"	"Computer Science"
"Karen" : "Sociology"	"Karen"	"Sociology"
"Harpreet" : "Mathematics"	"Harpreet"	"Mathematics"
"Sabah" : "Legal Studies"	"Sabah"	"Legal Studies"

Consider some more dictionary declarations :

```
dict1 = {}                  # it is an empty dictionary with no elements
DaysInMonths = { "January" : 31, "February" : 28, "March" : 31, "April" : 30, "May" : 31,
                 "June" : 30, "July" : 31, "August" : 31, "September" : 30,
                "October" : 31, "November" : 30, "December" : 31 }

BirdCount = {"Finch" : 10, "Myna" : 13, "Parakeet" : 16,
             "Hornbill" : 15, "Peacock" : 15}
```

NOTE

Dictionaries are also called associative arrays or mappings or hashes.

Now you can easily identify the *keys* and corresponding *values* from above given dictionaries.

One thing that you must know is that *keys* of a dictionary must be of immutable types, such as :

- a Python string,
- a number,
- a tuple (containing only immutable entries).

The *keys* of a dictionary must be of immutable types.

If you try to give a mutable type as key, Python will give you an error as: **"unhashable type"** . see below :

```
>>> dict3 = {[2,3]:"abc"}
Traceback (most recent call last):
  File "<ipython-input-63-477d407bcac3>", "line 1, in <module>
    dict3 = {[2,3]:"abc"}
TypeError: unhashable type: 'list'
```

IMPORTANT The above error **TypeError : unhashable type** always means that you have tried to assign a key with mutable type and Python dictionaries do not allow this.

9.2.2 Accessing Elements of a Dictionary

While accessing elements from a dictionary, you need the key. While in lists, the elements are accessed through their *index* ; in dictionaries, the elements are accessed through the *keys* defined in the *key:value* pairs, as per the *syntax* shown below :

```
<dictionary-name> [ <key>]
```

Thus to access the value for *key* defined as "Karen" in above declared *teachers* dictionary, you will write :

```
>>> teachers["Karen"]
```

and Python will return

```
Sociology
```

Similarly, following statement

```
>>> print ("Karen teaches", teachers['Karen'])
```

will give output as :

```
Karen teaches Sociology
```

While giving *key* inside square brackets gives you access only to the value corresponding the mentioned key, mentioning the dictionary name without any square brackets prints/ displays the entire contents of the dictionary.

NOTE

A dictionary operation that takes a key and finds the corresponding value, is called lookup.

Consider following example :

```
>>> d = {"Vowel1" : "a", "Vowel2" : "e", "Vowel3" : "i", "Vowel4" : "o", "Vowel5" : "u"}
```

Dictionary d contains five key : value pairs

Displaying the contents of dictionary d : Notice the output has shown elements in different order

```
>>> d
{'Vowel5' : 'u', 'Vowel4' : 'o', 'Vowel3' : 'i', 'Vowel2' : 'e', 'Vowel1' : 'a'}
```

Printing the contents of dictionary d : Notice the order of elements is different from input order

```
>>> print (d)
{ 'Vowel5' : 'u', 'Vowel4' : 'o', 'Vowel3' : 'i', 'Vowel2': 'e', 'Vowel1' : 'a'}

>>> d["Vowel1"]
'a'

>>> d["Vowel4"]
'o'
```

Accessing elements using their keys : "Vowel1" and "Vowel4" are the keys used to access corresponding values.

Notice that keys given here are in quotation marks *i.e.*, are of string type.

As per above examples, we can say that *key order* is not guaranteed in Python. This is because in Python dictionaries, the elements (*key : value* pairs) are *unordered* ; one cannot access element as per specific order. The only way to access a value is through *key*. Thus we can say that *keys* act like indexes to access *values* from a dictionary.

Also, attempting to access a key that doesn't exist causes an error. Consider the following statement that is trying to access a non-existent key (13) from dictionary d.

```
>>> d[13]
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    d KeyError : 13
[13]
```

NOTE

In Python dictionaries, the elements (*key : value* pairs) are unordered; one cannot access elements as per specific order.

The above error means that before we can access the value of a particular key using expression such as d["13"], we must first ensure the *key* (13 here) exists in the dictionary.

Like list elements, the *keys* and *values* of a dictionary are stored through their references. Following Fig. 9.1 illustrates the same for a dictionary {"goose" : 3, "tern" : 3, "hawk" : 1}

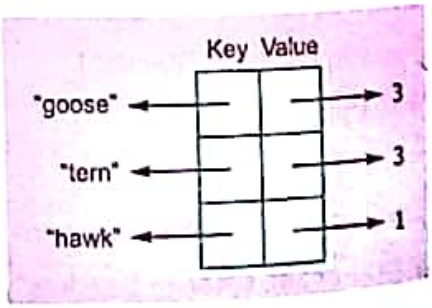


Figure 9.1 References of keys and values are stored in a dictionary

9.2.2A Traversing a Dictionary

Traversal of a collection means accessing and processing each element of it. Thus traversing a *dictionary* also means the same and same is the tool for it , *i.e.*, the Python loops.

The **for loop** makes it easy to **traverse or loop over** the items in a *dictionary*, as per following syntax :

```
for <item> in <Dictionary> :
    process each item here
```

The loop variable <item> will be assigned the keys of <Dictionary> one by one, (just like, they are assigned indexes of *strings* or *lists* while traversing them), which you can use inside the body of the *for loop*.

Consider following example that will illustrate this process. A dictionary namely **d1** is defined with *three* keys – a number, a string, a tuple of integers.

```
d1 = { 5 : "number",\
       "a" : "string",\
       (1,2) : "tuple" }
```

Recall that to break a statement/expression in multiple lines, you can use '\' at the end of physical line.

To traverse the above dictionary, you can write **for loop** as :

```
for key in d1 :
    print (key, ":", d1[key])
```

The above loop will produce the output as shown here.

```
a : string
(1, 2) : tuple
5 : number
```

How it works

The loop variable key in above loop will be assigned the keys of the Dictionary **d1, one at a time.** As dictionary elements are unordered, the order of assignment of keys may be different from what you stored.

Using the loop variable, which has been assigned one *key* at a time, the corresponding *value* is printed along with the *key* inside the loop-body using through statement :

```
print (key, ":", d1[key])
```

*it will access **value** from dictionary **d1** for given **key**.*

As per above output, loop-variable key will be assigned 'a' in first iteration and hence the key "a" and its value "string" will be printed ; in second iteration, key will get element (1, 2) and its value 'tuple' will be printed along with it ; and so on.

So, you can say that traversing all collections is similar. You can use *a for loop* to get hold of *indexes* or *keys* and then print or access corresponding values inside the loop-body.

Program 9.1 *Write a program to create a phone dictionary for all your friends and then print it.*

```
PhoneDict = { "Madhav" : 1234567, "Steven" : 7654321, "Dilpreet" : 6734521,
              "Rabiya" : 4563217, "Murughan" : 3241567, "Sampree" : 4673215 }

for name in PhoneDict :
    print (name, ":", PhoneDict[name])
```

The output produced by above program will be :

```
Rabiya :      4563217
Murughan :    3241567
Madhav :      1234567
Dilpreet :    6734521
Steven :      7654321
Sampree :     4673215
```

NOTE

Dictionaries are unordered set of elements, the printed order of elements is not same as the order you stored the elements in.

As you can make out from the output, that the loop variable **name** got assigned the keys of dictionary **PhoneDict**, one at a time. For the first iteration, it got the key "Rabiya", for the second iteration, it was assigned "Murughan" and so on. Since the dictionaries are unordered set of elements, the printed order of elements is not same as the order you stored the elements in.

9.2.2B Accessing Keys or Values Simultaneously

To see all the *keys* in a dictionary in one go, you may write <dictionary>.keys( ) and to see all *values* in one go, you may write <dictionary>.values( ), as shown below (for the same dictionary **d** created above) :

```
>>> d.keys()
dict_keys([ 'Vowel5', 'Vowel4', 'Vowel3', 'Vowel2', 'Vowel1'])
>>> d.values()
dict_values(['u', 'o', 'i', 'e', 'a'])
```

As you can see that the keys( ) function returns all the keys defined in a dictionary in the form of a sequence and also the values( ) function returns all the values defined in that dictionary in the form of a *sequence*.

You can convert the sequence returned by keys( ) and values( ) functions by using list( ) as shown below :

```
>>> list(d.keys())
['Vowel5', 'Vowel4', 'Vowel3', 'Vowel2', 'Vowel1' ]
>>> list(d.values( ))
['u', 'o', 'i', 'e', 'a']
```

9.2.3 Characteristics of a Dictionary

Dictionaries like lists are **mutable** and that is the only similarity they have with lists. Otherwise, dictionaries are different type of data structures with following characteristics :

1. Unordered Set

A dictionary is a unordered set of ***key : value*** pairs. Its values can contain references to any type of object.

2. Not a Sequence

Unlike a *string*, *list* and *tuple*, **a dictionary is not a sequence** because it is unordered set of elements. The sequences are indexed by a range of ordinal numbers. Hence, they are ordered, but a dictionary is an unordered collection.

3. Indexed by Keys, Not Numbers

Dictionaries are indexed by keys. According to Python, a key can be "any non-mutable type." Since strings and numbers are not mutable, you can use them as a key in a dictionary. And if a tuple contains immutable objects such as integers or strings etc., then only it can also be used as a key. But the values in a dictionary can be of any type, and types can be mixed within one dictionary.

Following dictionary **dict1** has keys of different immutable types :

```
>>> dict1 = {
        0 : "Value for key 0", 1 : "Value for key 1",
        "3" : "Value for a string-as-a-key",
        (4, 5): "Value for a tuple-as-a-key",
        "and for fun" : 7
    }
>>> dict1[0]
'Value for key 0'
```

Key of Integer type

```
>>> dict1[(4, 5)]
'Value for a tuple-as-a-key'

>>> dict1[4, 5]
'Value for a tuple-as-a-key'
```

Two ways of writing key of tuple type : tuple can be given as key – with or without parentheses.

```
>>> dict1["3"]
'Value for a string-as-a-key'
```

NOTE

The keys of a dictionary must be of immutable types such as numbers or strings. A tuple can also be a key provided it contains immutable elements.

A dictionary value can be any object, though.

4. Keys must be Unique

Each of the *keys* within a dictionary must be unique. Since *keys* are used to identify values in a dictionary, there cannot be duplicate *keys* in a dictionary.

However, *two* unique keys can have same *values*, e.g., consider the same *BirdCount* dictionary declared above :

Two unique keys having same value

```
BirdCount = { "Finch" : 10, "Myna" : 13, "Parakeet" : 16, "Hornbill" : 15, "Peacock" : 15}
```

Notice that two different keys "*Hornbill*" and "*Peacock*" have same value 15.

5. Mutable

Like *lists*, *dictionaries* are also ***mutable***. We can change the value of a certain key "in place" using the assignment statement as per syntax :

```
<dictionary>[<key>] = <value>
```

For example, consider the dictionary dict1 defined above :

```
>>> dict1["3"]
'Value for a string-as-a-key'
>>> dict1["3"] = "Changed to new string"
>>> dict1["3"]
'Changed to new string'
```

Value for key "3" changed using assignment statement

You can even add a new *key:value* pair to a dictionary using a simple assignment statement. But the key being added should be unique. If the key already exists, then value is simply changed as in the assignment statement above.

```
>>> dict1["new"] = "a new pair is added"
>>> dict1
{ 0 : 'Value for key 0', 1 : 'Value for key 1', (4, 5): 'Value for a
  tuple-as-a-key', '3' : 'Value for a string-as-a-key', 'and for fun' : 7,
  'new' : 'a new pair is added'}
```

6. Internally Stored as Mappings

Internally, the *key:value* pairs of a dictionary are associated with one another with some internal function (called hash-function[1]). This way of linking is called *mapping*. (Fig. 9.2)

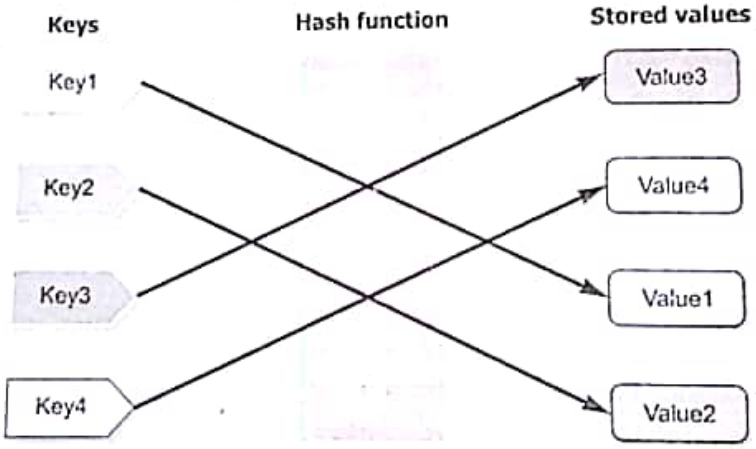


Figure 9.2 Internally *Keys* being mapped to *Values*

9.3 WORKING WITH DICTIONARIES

After basics of dictionaries, let us discuss about various operations that you can perform on dictionaries, such as adding elements to dictionaries, updating and deleting elements of dictionaries. This section is dedicated to the same.

9.3.1 Multiple Ways of Creating Dictionaries

You have learnt to create dictionaries by enclosing *key:value pairs* in curly braces. There are other ways of creating dictionaries too that we going to discuss here.

1. Hash-function is an internal algorithm to map and link a key with a stored value.

1. Initializing a Dictionary

In this method all the key:value pairs of a dictionary are written collectively, separated by commas and enclosed in curly braces.

You have already worked with this method. All the dictionaries created so far have been created through this method, *e.g.*, following dictionary **Employee** is also being created the same way :

```
Employee = { 'name' : 'John', 'salary' : 10000, 'age' : 24 }
```

2. Adding key : value Pairs to an Empty Dictionary

In this method, firstly an empty dictionary is created and then keys and values are added to it one pair at a time.

- **To create an empty dictionary,** there are *two* ways :
  - (*i*) by giving dictionary contents in empty curly braces as shown below :

    ```
    Employees = {}
    ```

  - (*ii*) by using dictionary constructor **dict( )** as shown below :

    ```
    Employee = dict()
    ```

- Next step is to add **key : value pairs, one at a time** as per syntax given below :

  ```
  <dictionary> [<key>] = <value>
  ```

For example, in the adjoining statements we are first adding a key : value pair of **'name': 'John'** , next statements adds *value* **10000** for *key* **'salary'** and the third and last statement adds *value* **24** for *key* **'age'**.

```
Employee['name'] = 'John'
Employee['salary'] = 10000
Employee['age'] = 24
```

This method of creating dictionaries is used to create dictionaries dynamically, at runtime.

3. Creating a Dictionary from *name* and *value* Pairs

Using the **dict( )** constructor of dictionary, you can also create a new dictionary initialized from specified set of keys and values. There are multiple ways to provide *keys* and *values* to *dict( )* constructor.

(*i*) **Specify Key : Value pairs as keyword arguments to dict( ) function,** *Keys* as arguments and *Values* as their values. Here the argument names are taken as keys of *String* type.

The keys are given as arguments, NOTICE the ***keys are not enclosed in quotes*** *here ; the values are given after = sign.*

```
Employee = dict(name = 'John', salary = 10000, age = 24)
```

See, Python automatically converted argument names to string type keys

```
>>> Employee
{ 'salary' : 10000, 'age' : 24, 'name' : 'John' }
```

(*ii*) **Specify comma-separated key : value pairs.** To give *key : value* pairs in this format, you need to enclose them in curly braces as shown below :

```
>>> Employee = dict({'name' : 'John', 'salary' : 10000, 'age' : 24})
>>> Employee
{'salary' : 10000, 'age' : 24, 'name' : 'John'}
```

(iii) Specify keys separately and corresponding values separately. In this method, the keys and values are enclosed separately in parentheses[2] and are given as arguments to the zip( ) function, which is then given as argument of dict( ).

All the keys are given separately enclosed in ( ) and all the values are given separately enclosed in ( ) as arguments to zip function. The zip( ) function is given as argument of dict( ) constructor

First group contains all the keys and second group contains all values in same order as that of corresponding keys.

```
>>> Employee = dict( zip( ('name', 'salary', 'age'), ('John', 10000, 24) ) )
>>> Employee
{'salary' : 10000, 'age' : 24, 'name' : 'John'}
```

The *zip function* clubs *first value* from first set with the *first value* of second set, *second value* from first set with the *second value* of second set, and so on, *e.g.*, in above example, **'name'** is clubbed with **'John'**, **'salary'** with **10000** and **'age'** with **24**. Using these clubbed values, the *dict()* constructor method creates *key : value* pairs for the dictionary.

(iv) Specify *Key : Value* pairs separately in form of sequences. In this method, one *list* or *tuple* argument is passed to dict( ). This argument (passed list or tuple) contains lists/tuples of individual *key : value* pairs.

That is, a *key : value* pair is specified in the form of a list and there are as many lists as items of the outer list as there are *key : value* pairs in the dictionary.

Consider the following example :

See, there is one list type argument passed to dict( ) constructor. This list argument, in turn contains all key:value pairs as lists. (3 list type entries in one list here.)

```
>>> Employee = dict([['name','John'], ['salary ', 10000], ['age', 24]])
>>> Employee
{'salary' : 10000, 'age' : 24, 'name' : 'John'}
```

You can also pass a tuple containing key : value pairs as list-element or tuples as elements. See following examples :

*The dict( ) method is provided **tuple argument**. The passed tuple contains **list-elements** of key, values (3 list entries in one tuple*

```
>>> Empl2 = dict(( ['name', 'John'], ['salary', 10000], ['age', 24] ))
>>> Empl2
{'salary' : 10000, 'age' : 24, 'name' : 'John' }
```

The dict( ) method is provided tuple argument. The passed tuple contains tuple-elements of key, values (3 tuple entries in one tuple)

```
>>> Empl3 = dict((('name','John'), ('salary', 10000), ('age', 24)))
>>> Empl3
{'salary' : 10000, 'age' : 24, 'name' : 'John'}
>>>
```

NOTE

Important. Using dict( ) method to create a dictionary takes longer time compared to traditional method of enclosing values in { }. Thus, it should be avoided until it becomes a necessity.

2. *i.e.*, as tuples.

9.3.2 Adding Elements to Dictionary

You can add new elements (*key* : *value* pair) to a dictionary using assignment as per following syntax. **BUT the key being added must not exist in dictionary and must be unique.** If the *key* already exists, then this statement will change the value of existing *key* and no new entry will be added to dictionary.

```
<dictionary>[<key>] = <value>
```

Consider the following example :

```
>>> Employee = { 'name' : 'John', 'salary' : 10000, 'age' : 24}
>>> Employee['dept'] = 'Sales'
>>> Employee
{'salary' : 10000, 'dept' : 'Sales', 'age' : 24, 'name' : 'John'}
```

New entry added

Nesting Dictionaries

You can even add dictionaries as *Values* inside a dictionary. Program 9.2 uses such a dictionary. Storing a dictionary inside another dictionary is called *nesting of dictionaries*. But one thing you must know about nesting of dictionaries is that, you can store a dictionary as a *value* only, inside a dictionary. You cannot have a *key* of dictionary type ; you know the reason – only immutable types can form the keys of a dictionary.

Program 9.2 *A Dictionary contains details of two workers with their names as keys and other details in the form of dictionary as value. Write a program to print the workers' information in records format.*

```
Employees = {'John' : {'age' : 25, 'salary' : 20000}, 'Diya' : {'age' : 35, 'salary' : 50000}}
for key in Employees :
    print ("Employee", key, ':' )
    print (' Age : ', str ( Employees[key]['age'] ))
    print ('Salary : ', str ( Employees[key]['salary'] ))
```

Carefully notice, how the elements are being accessed from inner dictionaries, stores as values

```
Employee John :
   Age : 25
   Salary:   20000
Employee Diya :
   Age : 35
   Salary:   50000
```

9.3.3 Updating Existing Elements in a Dictionary

Updating an element is similar to what we did just now. That is, you can change value of an existing *key* using assignment as per following syntax :

```
dictionary>[<key>] = <value>
```

Consider following example :

```
>>> Employee = {'name' : 'John' , 'salary' : 10000, 'age' : 24}
>>> Employee['salary'] = 20000
>>> Employee
{'salary' : 20000, 'age' : 24, 'name' : 'John'}
```

But make sure that the key must exist in the dictionary otherwise new entry will be added to the dictionary.

Using this technique of adding *key:value* pairs, you can create dictionaries interactively at runtime by accepting input from user.

Following program illustrates this.

NOTE

In Dictionaries, the updation and addition of elements are similar in syntax. But for addition, the key must not exist in the dictionary and for updation, the key must exist in the dictionary.

Program 9.3 *Write a program to create a dictionary containing names of competition winner students as keys and number of their wins as values.*

```
n = int(input("How many students ? "))
CompWinners = { }
for a in range(n) :
    key = input("Name of the student :")
    value = int (input("Number of competitions won :"))
    CompWinners[key] = value

print ("The dictionary now is :")
print (CompWinners)
```

The sample run of above program is :

```
How many students ? 5
Name of the student : Rohan
Number of competitions won : 5
Name of the student : Zeba
Number of competitions won : 3
Name of the student : Nihar
Number of competitions won : 3
Name of the student : Roshan
Number of competitions won : 1
Name of the student : James
Number of competitions won : 5
The dictionary now is :
{'Nihar' : 3, 'Rohan' : 5, 'Zeba' : 3, 'Roshan' : 1, 'James' : 5}
```

9.3.4 Deleting Elements from a Dictionary

There are *two* methods for deleting elements from a dictionary.

(*i*) To delete a dictionary element or a dictionary entry, *i.e.*, a *key:value* pair, you can use del command.

The syntax for doing so is as given below :

```
del <dictionary>[ <key>]
```

Consider the following example :

```
>>> empl3
{'salary' : 10000, 'age' : 24, 'name' : 'John'}
>>> del empl3['age']
>>> empl3
{'salary' : 10000, 'name' : 'John'}
```

See, the dictionary now has that entry removed

But with **del** statement, the key that you are giving to delete must exist in the dictionary, otherwise Python raises exception (**KeyError**). See below :

```
>>> del empl3['new']
```

Trying to delete a non-existent key

```
Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    del empl3['new']
KeyError : 'new'
```

(*ii*) Another method to delete elements from a dictionary is by using **pop( )** method as per following syntax :

```
<dictionary>.pop(<key>)
```

The *pop( )* method will not only delete the key:value pair for mentioned *key* but also return the corresponding value. Consider following code example

```
>>> employee
{'salary' : 10000, 'age' : 24, 'name' : 'John'}
>>> employee.pop('age')
24
```

value shown here for the element deleted

```
>>> employee
{'salary' : 10000, 'name' : 'John'}
```

Element deleted from dictionary

If you try to delete a *key* which does not exist, the Python returns error. See below :

```
>>> employee.pop('new')
Traceback (most recent call last):
  File "<pyshell#38>", line 1, in <module>
    employee.pop('new')
KeyError : 'new'
```

However, **pop( )** method allows you to specify what to display when the given key does not exist, rather than the default error message. This can be done as per following syntax :

```
<dictionary>.pop(<key>, <in-case-of-error-show-me>)
```

For example :

```
>>> employee.pop('new', "Not Found")
'Not Found'
```

See, now Python returned the specified message in place of error-message.

9.3.5 Checking for Existence of a Key

Usual membership operators in and not in work with dictionaries as well. But they can check for the existence of keys only. To check for whether a value is present in a dictionary (called *reverse lookup*), you need to write proper code for that. (Solved Problem 8 is based on *reverse lookup*.)

To use a membership operator for a key's presence in a dictionary, you may write statement as per syntax given below :

```
<key> in <dictionary>
<key> not in <dictionary>
```

- The in operator will return *True* if the given key is present in the dictionary, otherwise *False*.
- The not in operator will return *True* if the given key is not present in the dictionary, otherwise *False*.

Consider the following examples :

```
>>> empl = {'salary' : 10000, 'age' : 24, 'name' : 'John'}
>>>'age' in empl
True
>>>'John' in empl
False
>>>'John' not in empl
True
>>>'age' not in empl
False
```

'John' is not a key, hence Python returned False, as in only checks in keys of the dictionary.

Please remember that the operators in and not in do not apply on *values* of a dictionary, if you use them with dictionary name directly. That is, if you search for something with in or not in operator with a dictionary, it will only look for in the keys of the dictionary and not values, *e.g.*,

```
>>> dict1
{'age': 25, 'name': 'xyz', 'salary': 23450.5}
>>>'name' in dict1
True
>>>'xyz' in dict1
False
```

See, it returned False if you tried to search for a value in a dictionary using

NOTE

The in and not in operator check for membership only in *keys* of the dictionary and not in *values*.

However, if you need to search for a value in a dictionary, then you can use the in operator with <dictionary_name>.values( ), *i.e.*,

```
>>>'xyz' in dict1.values()
True
```

Using <dictionary>.values(), you can look for a value using in operator.

9.3.6 Pretty Printing a Dictionary

You generally use print function to print a dictionary in Python, e.g., if you have a dictionary as Winners = {'Nihar': 3, 'Rohan' : 5, 'Zeba': 3, 'Roshan': 1, 'James': 5} and if you give a print( ) to print it, Python will give result as :

```
>>> print(Winners)
{'Nihar': 3, 'Rohan': 5, 'Zeba': 3, 'Roshan': 1, 'James': 5 }
```

This is default way of printing dictionary in Python

From above style of printing dictionary, you can still make out the keys and values, but if the dictionary is large, then shouldn't there be any way of printing dictionary in a way that makes it more readable and presentable ?

Well there certainly is a way. For this, you need to import *json module* by giving statement **import json** (recall that in a program the **import** statement should be given at the top.)

And then you can use the function **dumps( )** of **json** *module* as per following syntax inside **print( ) :**

```
json.dumps(<dictionary name>, indent = <n>)
```

This n is a number

For example, consider following result

```
>>> print(json.dumps(Winners, indent = 2))
{
  "Nihar" : 3,
  "Rohan": 5,
  "Zeba": 3,
  "Roshan": 1,
  "James": 5
}
```

Dictionary name

Key:value pairs in separate lines and 2 spaces in front of every line because you gave value 2 for indent (indent = 2)

9.3.7 Counting Frequency of Elements in a list using Dictionary

Since a dictionary uses *key:value pairs* to store its elements, you can use this feature effectively if you trying to calculate frequency of elements in sequence such as lists.

You can do this as :

- Create an empty dictionary
- Take up an element from the list List (first element 1st time, second element 2nd time and so on)
- Check if this element exists as a key in the dictionary :

  If not then add {key:value} to dictionary in the form { List-element : count of List-element in List}

Following program illustrates this. But before we move on to the program code, it will be good if we talk about an interesting and useful function **split( )** that you can use with text to split it in words. (it will come handy especially if try to find frequency of words in a sentence or in a text file). The **split( )** works on string type data and breaks up a string into words and creates a list out of it, *e.g.*,

```
>>> text = "This is sample string"
>>> words = text.split()

>>> words
['This', 'is', 'sample', 'string']
```

```
>>> text = "one, two, three"
>>> words = text.split(" , " )

>>> words
['one', 'two', 'three']
```

Code on the left separates individual words on tthe basis of default separator white space from the string text , and code on the right separates individual words on the basis of given separator (,). The result is stored in a list namely words.

By default **split( )** breaks up a text based on white spaces, but you can give a separator character also *e.g.*, see sample code on the right above. Now consider the following program.

Program 9.4 Program to count the frequency of a list-elements using a dictionary.

```
import json
sentence = "This is a super idea This \
idea will change the idea of learning"
words = sentence.split()
d = {}
for one in words:
    key = one
    if key not in d:
        count = words.count(key)
        d[key] = count
print("Counting frequencies in list \n", words)
print(json.dumps(d, indent = 1))
```

Dictionary to hold words and frequencies

Add (key : count-of-key) to dictionary d

The output produced by above code is shown on the right:

```
Counting frequencies in list
['This', 'is', 'a', 'super',
'idea', 'This', 'idea', 'will',
'change', 'the', 'idea',
'of', 'learning']
{
 "This": 2,
 "is": 1,
 "a": 1,
 "super": 1,
 "idea": 3,
 "will": 1,
 "change": 1,
 "the": 1,
 "of": 1,
 "learning": 1
}
```

Words and their frequencies shown in dictionary.

9.4 DICTIONARY FUNCTIONS AND METHODS

Let us now talk about various built-in functions and methods provided by Python to manipulate Python dictionaries.

1. The len( ) method

This method returns length of the *dictionary* , *i.e.*, the count of elements (*key:value* pairs) in the *dictionary*. The syntax to use this method is given below :

```
len(<dictionary>)
```

– *Takes dictionary name as argument and returns an integer.*

Example :

```
>>>employee = {'name' : 'John', 'salary' : 10000, 'age' : 24}
>>>len(employee)
3
>>>employee['dept'] = 'Sales'
>>>len(employee)
4
```

The len( ) returns the count of key:value pairs from the dictionary

See, after adding a new entry, the length of dictionary changed

2. The clear( ) method

This method removes all items from the *dictionary* and the dictionary becomes empty dictionary post this method. The syntax to use this method is given below :

```
<dictionary>.clear()
```

– *Takes no argument , returns no value*

Example :

```
>>> Employee = {'name' : 'John', 'salary' : 10000, 'age' : 24}
>>> Employee.clear()
>>> Employee
{}
```

See, now the dictionary is empty

The clear method does not delete the *dictionary*, it just deletes all the elements inside the dictionary. If, however, you want to delete the dictionary itself so that dictionary is also removed and no object remains, then you can use del statement along with dictionary name.

For example :

```
>>> Employee = {'name' : 'John', 'salary' : 10000, 'age' : 24}
>>> del Employee
>>> Employee
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
    Employee
NameError : name 'Employee' is not defined
```

it will delete the complete dictionary

See, no 'Employee' object now exists.

NOTE

The clear( ) removes all the elements of a dictionary and makes it empty dictionary while del statement removes the complete dictionary as an object. After del statement with a dictionary name, that dictionary object no more exists, not even empty dictionary.

3. The get( ) method

With this method, you can get the item with the given key, similar to dictionary [key]. If the key is not present, Python by default gives error, but you can specify your own message through default argument as per following syntax :

```
<dictionary>.get( key , [ default ])
```

Takes key as essential argument and returns corresponding value if the key exists or when key does not exist, it returns Python's Error if *default* argument is missing otherwise returns default argument .

Example :

```
>>> empl1
{'salary' : 10000, 'dept' : 'Sales', 'age': 24, 'name' : 'John'}
>>> empl1.get('dept')
'Sales'
```

See it returned value for given *key if key* exists in the dictionary. If you only specify the key as argument, which does not exist in the dictionary, then Python will return error :

```
>>> empl1.get('desig')
Traceback (most recent call last):
  File "<pyshell#50>", line 1, in <module>
    empl1.get('desig')
NameError: name 'desig' is not defined
```

In place of error, you can also specify your own custom error message as second argument :

```
>>> empl1.get('desig', "Not yet")
'Not yet'
>>> empl1.get('desig', "Error!! Key not found")
'Error!! Key not found'
```

4. The items( ) method

This method returns all of the items in the *dictionary* as a sequence of (key, value) tuples. Note that these are returned in no particular order.

```
<dictionary>.items()
```

– *Takes no argument ; Returns a sequence of (key, value) pairs*

Example :

```
employee = {'name' : 'John', 'salary' : 10000, 'age' : 24}
myList = employee.items()
for x in myList :
    print (x)
```

The adjacent code gives output as :

```
( 'salary', 10000)
('age', 24)
( 'name', 'John' )
```

As the items( ) function returns a sequence of (key value) pairs, you can write a loop having two variables to access key value pairs, *e.g.*,

```
empl = {'age': 25, 'name': 'Bhuvan', 'salary': 20000}
seq = empl.items()
for ky, vl in seq:
    print(val, key)
```

The adjacent code gives output as :

```
25 age
Bhuvan name
20000 salary
```

This loop has two iteration variables ky, vl because the function items( ) returns a list of tuples and ky, vl is a tuple assignment that successively iterates through each of the *key-value pairs* in the dictionary *i.e.*, each key-value pair is assigned to loop variables ky and vl at a time.

5. The keys( ) method

You have already worked with this method. This method returns all of the *keys* in the *dictionary* as a sequence of *keys* (in form of a list). Note that these are returned in no particular order.

```
<dictionary>.keys()
```

– *Takes no argument ; Returns a list sequence of keys*

Example :

```
>>> employee
{'salary' : 10000, 'dept' : 'Sales', 'age' : 24, 'name' : 'John'}
>>> employee.keys()
['salary', 'dept', 'age', 'name']
```

6. The values( ) method

This method returns all the values from the *dictionary* as a sequence (a list). Note that these are returned in no particular order. The syntax to use this method is given below :

```
<dictionary>.values()
```

– *Takes no argument ; Returns a list sequence of values*

Example :

```
>>> employee
{'salary' : 10000, 'dept' : 'Sales', 'age' : 24, 'name' : 'John'}
>>> employee.values()
[10000, 'Sales', 24, 'John']
```

7. The update( ) method

This method merges *key : value* pairs from the new *dictionary* into the original *dictionary*, adding or replacing as needed. The items in the new dictionary are added to the old one and override any items already there with the same keys.

The syntax to use this method is given below :

Dictionary to be updated → `<dictionary>.update ( <other-dictionary>)` ← *This dictionary's items will be taken for updating other dictionary.*

Example :

Set. the elements of dictionary employee2 have overridden the elements of dictionary employee1 having the same keys i.e. of keys 'name' and "salary"

```
>>> employee1 = {'name' : 'John', 'salary' : 10000, 'age' : 24}
>>> employee2 = {'name' : 'Diya' , 'salary' : 54000, 'dept' : 'Sales'}
>>> employee1.update(employee2)
>>> employee1
{'salary' : 54000, 'dept' : 'Sales', 'name' : 'Diya', 'age' : 24}
>>> employee2
{'salary' : 54000, 'dept' : 'Sales', 'name' : 'Diya'}
```

This method is equivalent to the following Python statement :

```
for key in newDict.keys() :
                        #newDict is employee2 here
    dictionary[k]= newDict[k]
```

Check Point 9.1

1. Why are dictionaries called mutable types ?
2. What are different ways of creating dictionaries ?
3. What values can we have in a dictionary ?
4. How are individual elements of dictionaries accessed ?
5. How is indexing of a dictionary different from that of a list or a string ?
6. Which of the following types qualify to be the keys of a dictionary ?
   (a) String (b) tuple (c) Integer
   (d) float (e) list
   (f) dictionary
7. Can you change an element of a sequence or collection ? What if the collection is a dictionary ? What if the sequence is a string ?
8. What do you understand by ordered collection and unordered collection ? Give examples.
9. How do you add key:value pairs to an existing dictionary ?
10. Can you remove key:value pairs from a dictionary and if so, how ?

DICTIONARY HANDLING AND MANIPULATION — PriP — Progress In Python 9.1

This PriP session aims at strengthening skills related to Dictionary handling and manipulation.

⋮

>>>❖<<<

With this we have come to the end of our chapter. Let us quickly revise what we have learnt so far.

LET US REVISE

- *Dictionaries are mutable with elements in the form of a key:value pair that associate keys to values.*
- *The keys of a dictionary must be of immutable types.*
- *Dictionaries are also called associative arrays or mappings or hashes.*
- *In Python dictionaries, the elements (key:value pairs) are unordered ; one cannot access element as per specific order.*
- *Keys of a dictionary must be unique.*
- *Dictionaries can be created in multiple ways – through { } and through dict() constructor.*
- *In Dictionaries, the updation and addition of elements are similar in syntax. But for addition, the key must not exist in the dictionary and for updation, the key must exist in the dictionary.*
- *To delete an element, you can either use del statement or use pop( ) method.*
- *Some useful dictionary methods and functions are : len(), clear(), get(), items(), keys(), values() and update().*
- *The membership operators in and not in only work with dictionary keys.*
- *The clear( ) function removes all the items from dictionary but the dictionary object exists as an empty dictionary.*
- *The del <dict> statement removes a dictionary object along with its items. After this, there exists no dictionary object.*

Solved Problems

1. *How are dictionaries different from lists ?*

   Solution. The dictionary is similar to lists in the sense that it is also a collection of data-items just like lists BUT it is different from lists in the sense that lists are *sequential collections* (ordered) and dictionaries are *non-sequential collections* (unordered).

   In lists, where there is an order associated with the data-items because they act as storage units for other objects or variables you've created. Dictionaries are different from lists and tuples because the group of objects they hold aren't in any particular order, but rather each object has its own unique name, commonly known as a key.

2. *How are objects stored in lists and dictionaries different ?*

   Solution. The objects or values stored in a dictionary can basically be anything (even the nothing type defined as *None*), but keys can only be immutable type-objects. *e.g.*, strings, tuples, integers, etc.

3. *When are dictionaries more useful than lists ?*

   Solution. Dictionaries can be much more useful than lists. *For example*, suppose we wanted to store all our friends' cell-phone numbers. We could create a list of pairs, (name of friend, phone number), but once this list becomes long enough searching this list for a specific phone number will get time-consuming. Better would be if we could index the list by our friend's name. This is precisely what a dictionary does.

4. *Can sequence operations such as slicing and concatenation be applied to dictionaries ? Why ?*

   Solution. No, sequence operations like slicing and concatenation cannot be applied on dictionaries. The reason being, a dictionary is not a sequence. Because it is not maintained in any specific order, operations that depend on a specific order cannot be used.

5. *Why can't Lists can be used as keys ?*

Solution. Lists cannot be used as keys in a dictionary because they are mutable. And a Python dictionary can have only keys of immutable types.

6. *If the addition of a new key:value pair causes the size of the dictionary to grow beyond its original size, an error occurs. True or false ?*

Solution. **False.** There cannot occur an error because Dictionaries being the mutable types, they can grow or shrink on and as-needed basis.

7. *Consider the following code fragments. What outputs will they produce ?*

(*i*)
```
aDict = {'Bhavna':1, "Richard":2, "Firoza":10, "Arshnoor":20}
temp = 0
for value in aDict.values() :
    temp = temp+value
print(temp)
```

(*ii*)
```
aDict = { 'Bhavna':1, "Richard":2, "Firoza":10, "Arshnoor":20}
temp = " "
for key in aDict:
    if temp < key:
        temp = key
print(temp)
```

(*iii*)
```
aDict = { 'Bhavna':1, "Richard":2, "Firoza":10, "Arshnoor":20}
k = "Bhavna
v = -1
if k in aDict:
    aDict[k] = v
print(aDict)
```

Solution.

(*i*) 33 (*ii*) Richard

(*iii*) { 'Bhavna': - 1, 'Richard' :2, 'Firoza' = 10, 'Arshnoor' : 20 }

8. *The membership operators only work with keys of a dictionary but to check for the presence of a value in a dictionary, you need to write a code.*

Write a Python program that takes a value and checks whether the given value is part of given dictionary or not. If it is, it should print the corresponding key otherwise print an error message.

Solution.
```
dict1 = {0 :"Zero", 1 :"One", 2 :"Two", 3 :"Three", 4 :"Four", 5 :"Five"}
ans = "y"
while ans == 'y' or ans == 'Y':
   val = input("Enter value:")
   print("Value", val, end = " ")
   for k in dict1 :
      if dict1[k]== val:
         print("exists at", k)
   else :
      print("not found")
   ans = input("Want to check more values? (y/n) :")
```

9. *Marks of three students "Suniti", "Ryna" and "Zeba" in three subjects are available in following three dictionaries respectively :*

```
d1 = {1 : 40, 2 : 70, 3 : 70}
d2 = {1 : 40, 2 : 50, 3 : 60}
d3 = {1 : 70, 2 : 80, 3 : 90}
```

Create a nested dictionary that stores the marks details along with student names and then prints the output as shown below :

```
Name
Ryna
Subject (key)    Marks (value)
    1                40
    2                50
    3                60

Name
Zeba
Subject (key)    Marks(value)
    1                70
    2                80
    3                90

Name
Suniti
Subject(key)     Marks(value)
    1                40
    2                70
    3                70
```

Solution.

```
d1 = {1 : 40, 2 : 70, 3 : 70}
d2 = {1 : 40, 2 : 50, 3 : 60}
d3 = {1 : 70, 2 : 80, 3 : 90}
d4 = {"Suniti" : d1, "Ryna" : d2, "Zeba" : d3}
for x in d4.keys() :
    print ("Name")
    print (x)
    print (" Subject(key)", '\t', "Marks(value)")
    for y in d4[x].keys() :
        print ("  ", y, '\t\t', d4[x][y])
    print()
```

10. *Consider a dictionary my_points with single-letter keys, each followed by a 2-element tuple representing the coordinates of a point in an x-y coordinate plane.*

```
my_points = { 'a' : (4, 3), 'b' : (1, 2), 'c' : (5, 1) }
```

Write a program to print the maximum value from within all of the values tuples at same index .

For example, maximum for 0th index will be computed from values 4, 1 and 5, i.e., all the entries at 0th index in the value-tuple.

Print the result in following format :

```
Maximum Value at index(my_points, 0) = 5
Maximum Value at index(my_points, 1) = 3
```

Solution.

```
my_points = { 'a' : (4, 3), 'b' : (1, 2), 'c' : (5, 1) }
highest = [0, 0]
init = 0
for a in range(2) :
  init = 0
  for b in my_points.keys():
      val = my_points[b][a]

      if init == 0 :
          highest[a] = val
      init += 1

      if val > highest[a] :
          highest[a] = val
  print ("Maximum Value at index(my_points, ", a, ") = ", highest[a])
```

GLOSSARY

Dictionary	A mutable, unordered collection with elements in the form of a key:value pairs that associate keys to value.
Mapping	Linking of a key with a value through some internal function (hash function).
Nesting	Having an element of similar type inside another element.
Lookup	A dictionary operation that takes a key and finds the corresponding value.

Assignments

Type A : Short Answer Questions/Conceptual Questions

1. Why is a dictionary termed as an unordered collection of objects ?
2. What type of objects can be used as keys in dictionaries ?
3. Though tuples are immutable type, yet they cannot always be used as keys in a dictionary. What is the condition to use tuples as a key in a dictionary ?
4. What all types of values can you store in : (*a*) dictionary-values ? (*b*) dictionary-keys ?
5. Can you change the order of dictionary's contents, *i.e.*, can you sort the contents of a dictionary ?
6. In no more than one sentence, explain the following Python error and how it could arise:

   `TypeError: unhashable type: 'list'`
7. Can you check for a value inside a dictionary using **in** operator? How will you check for a value inside a dictionary using **in** operator ?
8. Dictionary is a mutable type, which means you can modify its contents ? What all is modifiable in a dictionary ? Can you modify the keys of a dictionary ?
9. How is del D and del D[<key>] different from one another if *D* is a dictionary ?
10. How is clear( ) function different from **del <dict>** statement ?

10 Understanding Sorting

In This Chapter

10.1 INTRODUCTION

In general terms, as you see in many junior level activities, sorting means — from a pile of mixed objects, picking and placing identical objects into separate groups. But in computer terms, sorting has different meaning. Computers often deal with sequences having multiple elements. Sorting in computer terms means arranging these elements in a specific order *e.g.*, in increasing order or decreasing order. In this chapter, we shall discuss what sorting means in computer terms and two techniques of doing it — *bubble sort* and *insertion sort*.

10.2 WHAT IS SORTING ?

Sorting in computer terms means arranging elements in a specific order — ascending or increasing order or descending or decreasing order. For instance, if you have sequence as :

S1 = [4, 6, 2, 8, 1, 6]

then sorted sequence in ascending order will be :

[1, 2, 4, 5, 6, 8]

and in decreasing order, it would be :

[8, 6, 5, 4, 2, 1]

SORTING

Sorting, in computer terms, refers to arranging elements in a specific order — ascending or descending.

You have used 'sorting techniques' unknowingly, indirectly in your day to day life, *e.g.*, if you want to keep a pile of containers with different sizes, what do you do ? You put the biggest container first, then smaller than it into it, further smaller into it and so on and you put the smallest container in the last. This is sorting only. (see below)

Similarly, when you arrange your books in the order of their thickness, it is also sorting.

There are multiple ways or techniques or algorithms that you can apply to sort a group of elements such as *selection sort, insertion sort, bubble sort, heap sort, quick sort* etc.

In this chapter, we shall talk about two sorting algorithms – *bubble sort* and *insertion sort.*

10.3 BUBBLE SORT

The basic idea of bubble sort is to compare two adjoining values and exchange them if they are not in proper order. To understand this, have a look at figure 10.1 (on next page) that visually explains the process of *bubble sort*. For instance, in following figure (Fig. 10.1) unsorted array is being sorted in ascending order using bubble sort. In every pass, the heaviest element settles as its appropriate position in the bottom.

Algorithm BUBBLE SORT IN LINEAR LIST

```
1. For I = L TO U
2.     For J = L TO [ (U - 1) - I ]
                    # need not consider already settled heavy elements
                    # that is why (u - 1) - I.
3.         if AR[J] > AR[J + 1] then
                                # swap the values
4.             temp = AR[J]
5.             AR[J] = AR[J + 1]
6.             AR[J + 1] = temp
           # end of if
       # end of inner loop
7. END.
                                # end of outer loop
```

Two adjacent elements being compared are shown in shaded boxes.

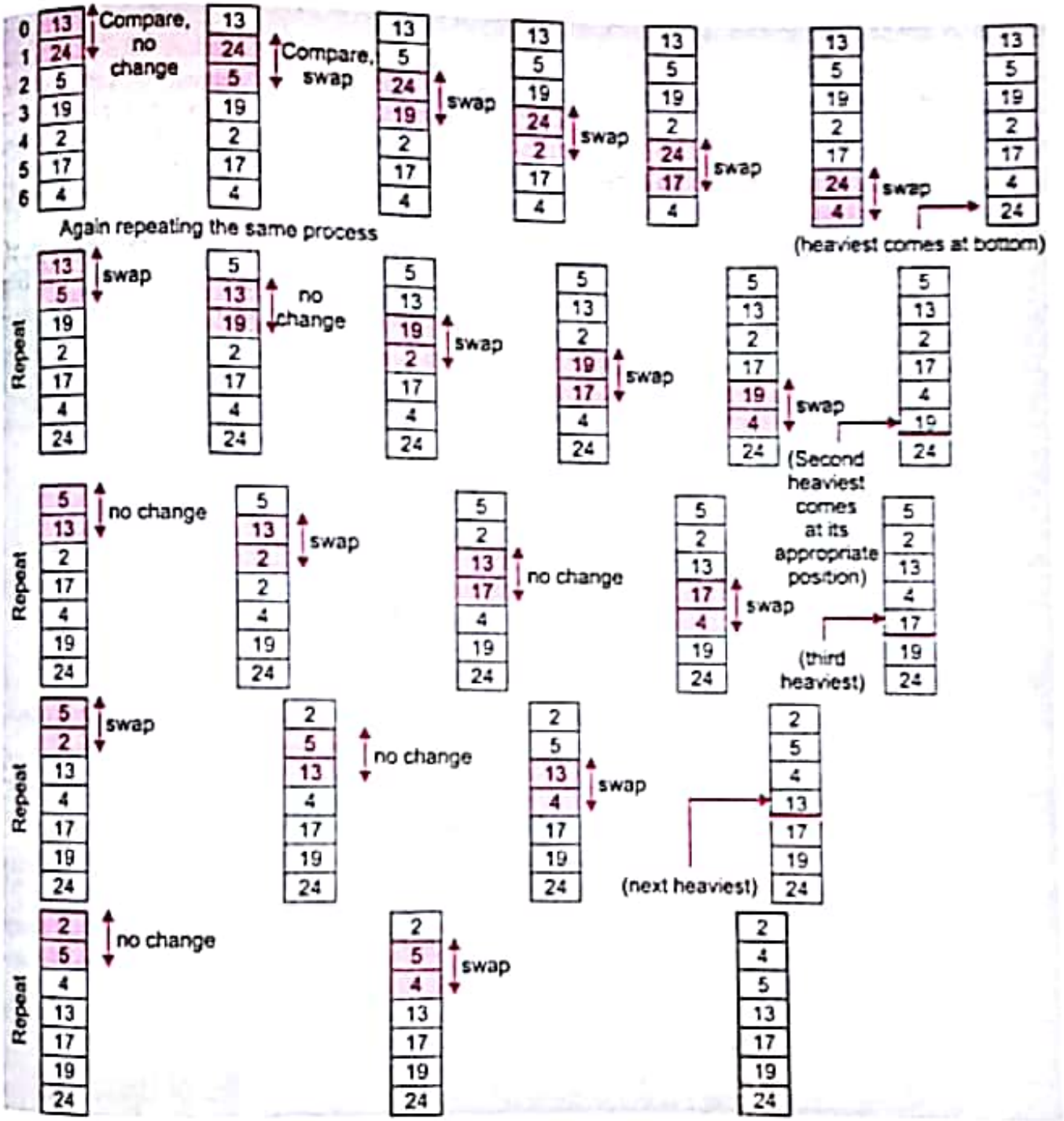


Figure 10.1 Bubble sort.

We can now write code for bubble sort as shown in algorithm on previous page. Notice we have kept in mind that since one heavy element settles at its right position in one complete iteration, next time, we need not compare that element. That is, after first complete iteration, we can have 1 less number of comparisons in next iteration (as 1 heaviest element has settled in the end) ; after second complete iteration, we can have 2 less number of comparisons in next iteration (as 2 heavy elements have settled in their correct position by now) and so on.

Following program shows Python code for the same, the bubble sort technique.

Program 10.1 Program to sort a list using Bubble sort.

```
aList = [ 15, 6, 13, 22, 3, 52, 2]
print ("Original list is :", aList)
n = len(aList)
# Traverse through all list elements
for i in range(n):
    # Last i elements are already in place
    for j in range(0, n-i-1):
        # traverse the list from 0 to n-i-1
        # Swap if the element found is greater
        # than the next element
        if aList[j] > aList[j + 1] :
            aList[j], aList[j + 1] = aList[j + 1], aList[j]
print ("List after sorting :", aList)
```

We have taken a pre-initialized list. You can even input a list prior to sorting it.

This expression will ensure that we do not compare the heavier elements that have already settled at correct position.

The output produced by above code is like.

```
Original list is : [15, 6, 13, 22, 3, 52, 2]
List after sorting : [2, 3, 6, 13, 15, 22, 52]
```

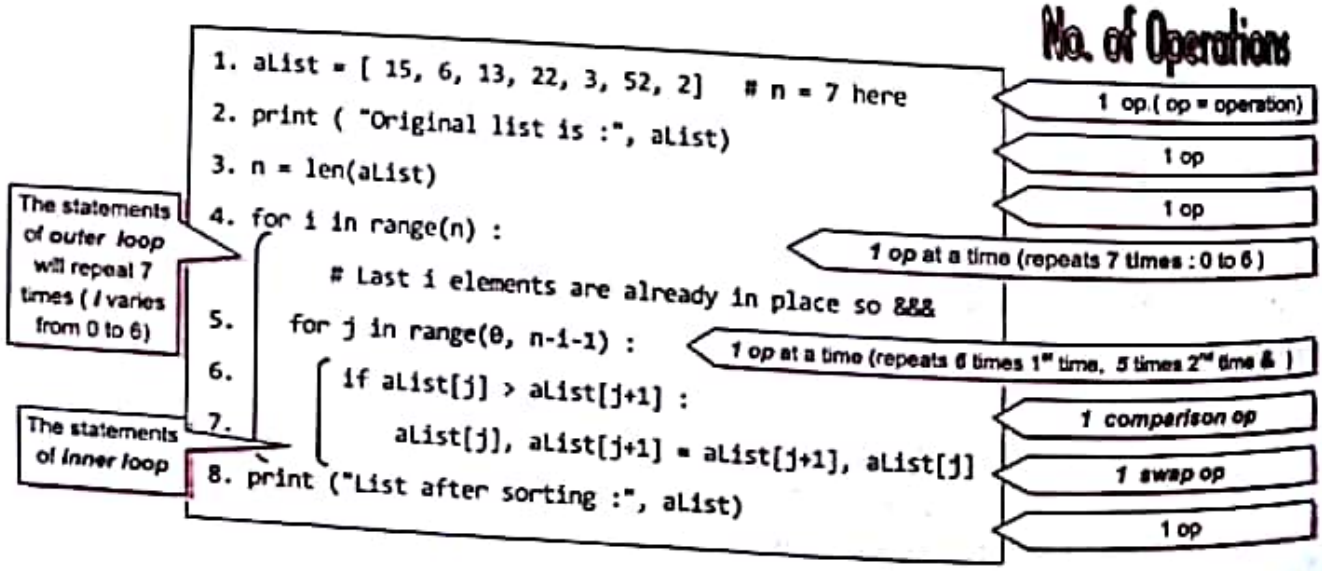
Calculating Number of Operations

Let us now consider number of operations taking place in above program. Number of operations is an important aspect to know as more number of operations means more usage of CPU time. Two programs with different logic can deliver the same output but the efficient one will accomplish the task in lesser number of operations.

To calculate number of operations taking place in above program, let us keep the things simple:

- We'll consider **1 operation** per statement that will include *assignment, comparison swap etc.* [Please note comparison and *swap* may take more than 1 operation to perform but we are considering it as 1 operation for simplicity sake and understanding sake]

Now consider this first :



So total number of operations taking place in above code can be calculated as :

Operations in Line 1 + Line 2 + Line 3	= 3 operations	
+ Operations in Lines 4..7 (for loop)	= ? (to be calculated)	...Eqn. (1)
+ Operation in Line 8	= 1 operation	

Let us calculate operations in **Outer for loop :**

Outer for loop repeats n times [for n as 7, it is repeating 7 times]

So total number of operations in outer loop, **for loop** are :

Operations in Line 4..7 × 7 times

= Line 4 = 1 op

= Line 5..7 = **Inner for Loop**

Inner for loop is repeated $(n - i - 1)$ times

(so 6 times *first time*, 5 times *second time*, 4 times *third time* and so on)

= Line 5 = 1 op

+ Line 6 = 1 op

+ Line 7 = 1 op = **3 ops**

Inner for loop performs 3 operations in single iteration

So outer for loop operations are :

1st iteration $(i = 0)$

= 1 op (Line 4) + 3 ops (Line 5-7) × 6 times = **19 ops**

inner for loop

2nd iteration $(i = 1)$ = 1 op (Line 4) + 3 ops (Lines 5-7) × 5 times = **16 ops**

3rd iteration $(i = 2)$ = 1 op + 3 ops × 4 times = **13 ops**

4th iteration $(i = 3)$ = 1 op + 3 ops × 3 times = **10 ops**

5th iteration $(i = 4)$ = 1 op + 3 ops × 2 times = **7 ops**

6th iteration $(i = 5)$ = 1 op + 3 ops × 1 times = **4 ops**

7th iteration = 1 op + 3 × 0 times = **1 op**

Total ops in outer for loop = **70 ops**

So putting this value in reference eqn. (1) given above

Total operations = 3 ops + 70 ops + 1 op = 74 ops

One above given Bubble sort program takes approximately **74 operations** when n is 7

for $n = 8$, it will be about = 96 ops

for $n = 9$ it will be about = 121 ops

for $n = 6$ it will be about = 55 ops

for $n = 5$ it will be about = 39 ops

So with larger values of n it increases.

Calculating number of operations this way will be cumbersome. Let us concentrate on crucial operations taking place — **Comparisons** and *Swappings* as these are major operations taking place.

Total number of comparisons in bubble sort is :

$$(N-1)+(N-2)+(N-3)+(N-4)+(N-5) \ \ldots \ (2)+(1) = N(N-1)/2 \text{ i.e., } \cong N^2$$

Let us see how we have reached at this number :

The **outer loop** iterates once for each element in the given sequence *i.e.*, N times, where N is the number of elements in the given sequence.

The inner loop iterates $N-1$ times for first iteration of outer loop, $N-2$ times for second iteration of outer loop, $N-3$ times for third iteration of outer loop and this process continues.

- For the first iteration of the outer loop, when we are trying to place the largest element in its correct position , $N-1$ ***comparisons*** takes place [the first comparison is made between the *first* and *second* elements, the second comparison is made between the *second* and *third* elements, and so on until the $(N-1)^{th}$ comparison is made between the $(N-1)^{th}$ and the N^{th} element.]
- For the second iteration of the outer loop, there is no need to compare against the last element of the list, because it was put in the correct position on the previous pass. Therefore, the second iteration requires only $N-2$ *comparisons.*
- Similarly the third iteration requires $(N-3)$ *comparisons,* and so on.

Thus,

Pass	Comparisons
1	$N-1$
2	$N-2$
3	$N-3$
...	...
$N-1$	1

This gives us total number of comparisons as :

$$(N-1)+(N-2)+(N-3)+(N-4)+(N-5).....(2)+(1) = N(N-1)/2 \cong N^2$$

The number of swapping will vary depending upon the elements of the sequence :

- In the **best case**, *i.e.*, when all the elements of the sequence are already sorted (*i.e.*, in desired sort order), no swapping will take place.

$$\cong n^2 \text{ comparisons} + 0 \text{ swappings} = N^2 \text{ ops}$$

- However, in the **worst case**, *i.e., when all elements are in opposite order,* every comparison will result into a swapping.

$$\cong N^2 \text{ comparisons} + N^2 \text{ swappings} = 2N^2 \text{ ops}$$

In general, we determine the efficiency of a program using the highest degree term in the expression for number of operations. [The lesser degree terms do not really matter]

So, programs with lesser degree in number of operations are considered more efficient program than programs with higher degree.

Now that we have talked about number of operations and efficiency of a bubble sort algorithms, let us quickly recap how bubble sort works.

Look at the figure below that shows the working of Bubble sort, *i.e.*, it shows the array after every pass and changes after every swap takes place. The elements being compared in every pass are shown with colour.

```
original list is : [15, 6, 13, 22, 3, 52, 2]

----Iteration 1 of outer loop ----
List after pass 1 : [6, 15, 13, 22, 3, 52, 2]
List after pass 2 : [6, 13, 15, 22, 3, 52, 2]
List after pass 3 : [6, 13, 15, 22, 3, 52, 2]
List after pass 4 : [6, 13, 15, 3, 22, 52, 2]
List after pass 5 : [6, 13, 15, 3, 22, 52, 2]
List after pass 6 : [6, 13, 15, 3, 22, 2, 52]

—Iteration 2 of outer loop ——
List after pass 1 : [6, 13, 15, 3, 22, 2, 52]
List after pass 2 : [6, 13, 15, 3, 22, 2, 52]
List after pass 3 : [6, 13, 3, 15, 22, 2, 52]
List after pass 4 : [6, 13, 3, 15, 22, 2, 52]
List after pass 5 : [6, 13, 3, 15, 2, 22, 52]

—Iteration 3 of outer loop ——
List after pass 1 : [6, 13, 3, 15, 2, 22, 52]
List after pass 2 : [6, 3, 13, 15, 2, 22, 52]
List after pass 3 : [6, 3, 13, 15, 2, 22, 52]
List after pass 4 : [6, 3, 13, 2, 15, 22, 52]

——Iteration 4 of outer loop ——
List after pass 1 : [3, 6, 13, 2, 15, 22, 52]
List after pass 2 : [3, 6, 13, 2, 15, 22, 52]
List after pass 3 : [3, 6, 2, 13, 15, 22, 52]

——Iteration 5 of outer loop ——
List after pass 1 : [3, 6, 2, 13, 15, 22, 52]
List after pass 2 : [3, 2, 6, 13, 15, 22, 52]

——Iteration 6 of outer loop ——
List after pass 1 : [2, 3, 6, 13, 15, 22, 52]
```

PriP SORTING WITH BUBBLE SORT — Progress In Python 10.1

This PriP session aims at practice of sorting with Bubble Sort.

Please check the practical component-book – Progress in Computer Science with Python and fill it there in PriP 10.1 under Chapter 10 after practically doing it on the computer.

>>>❖<<<

10.4 INSERTION SORT

Have you every observed your class teacher after she has collected your answer sheets ? To arrange all the answer sheets in the order of your roll numbers, what does she do often ? Doesn't she do something like this :

Take first two sheets and put them in correct order – swapping them it needed. Then take the third sheet and put it in the correct position of previously arranged sheets ; then take fourth sheet and do the same, and so on.

She repeats it until all the answer sheets are arranged in the desired order.

Do you know what this process of arranging things is called ? Well, this is **insertion sort**. Even playing cards are often sorted this way, mostly.

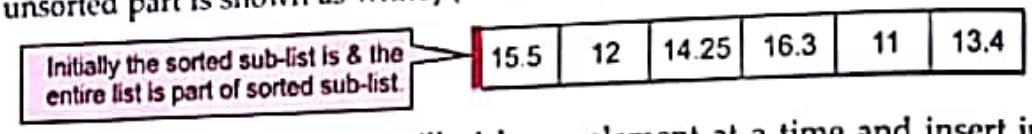
Insertion sort is a sorting algorithm that builds a sorted list one **element at a time** from the unsorted list. It virtually divides the list of elements in two sub-lists : ***sorted sub-list*** and ***unsorted sub-list***. Initially the entire list is unsorted sub-list and the sorted sub-list is empty. Then it takes one element from the unsorted sub-list and inserts it into the correct position in the sorted sub-list. Although it is not very efficient on large datasets, it is an efficient sorting technique for small datasets.

INSERTION SORT

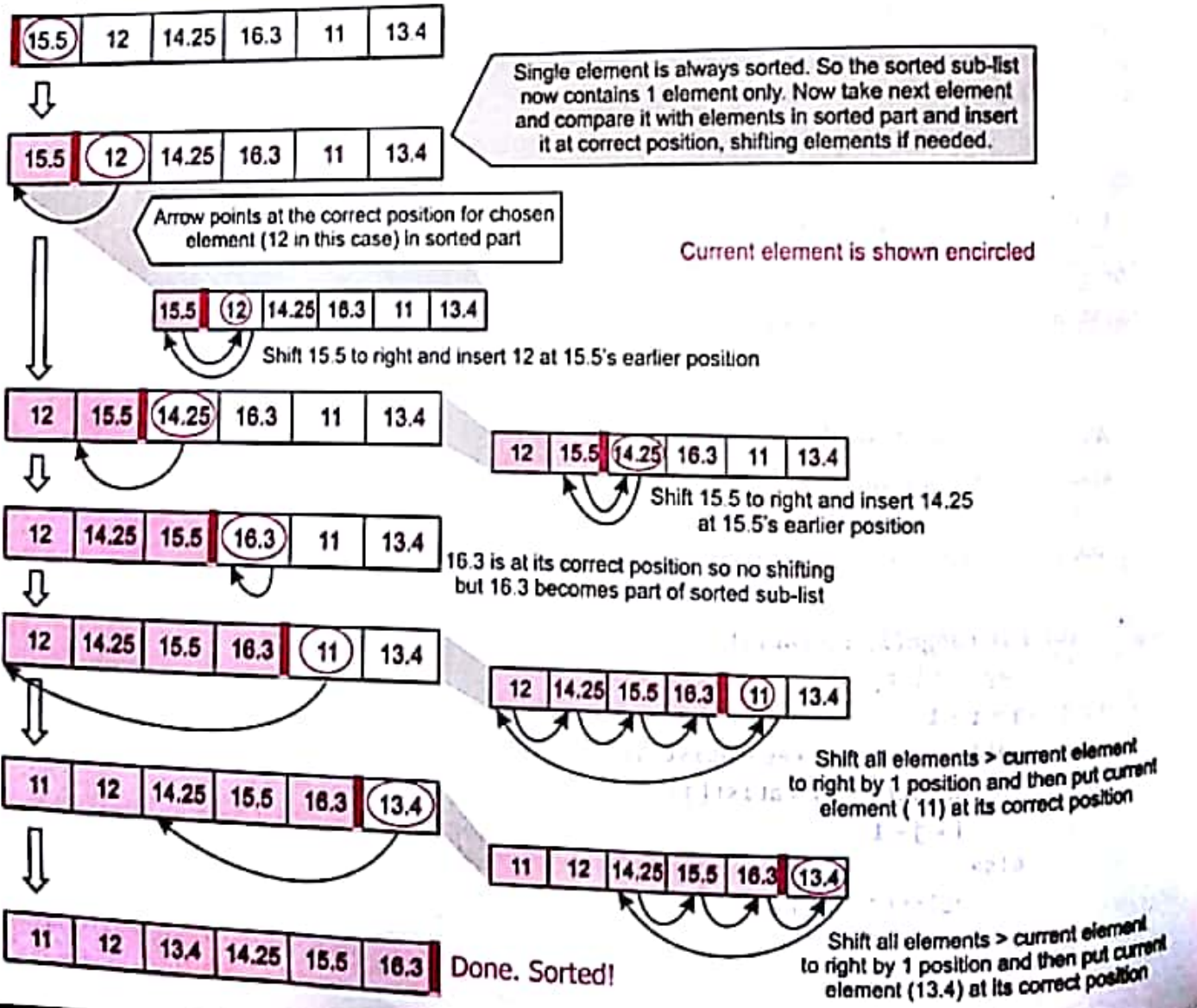
Insertion sort is a sorting algorithm that builds a sorted list one element at a time from the unsorted list by inserting the element at its correct position in sorted list.

Let us see how it works :

The colored line divides the sorted and unsorted sub-lists : on the left of colored line is the sorted sub-list and on the right is the unsorted sub-list. Initially the sorted sub-list is empty (no element in it) and the entire list is part of sorted sub-list. [Notice sorted part is shown shaded while unsorted part is shown as white] (see below)



Let us now see how insertion sort will pick one element at a time and insert in into correct position in sorted part. The element being picked is shown as encircled :



The pseudocode for sorting a sequence A using insertion sort is as given below. Notice that loop is started from 2nd element onwards (*i.e.*, index 1 onwards) because left to it is single element sub-list and a single element list is always sorted.

NOTE

Unlike the other sorts, the insertion sort passes through the array only once.

Pseudocode

```
for i = 1 to n
    key = A[i],   j = i - 1
    while(j > 0) and (A[i] > key)do
        A[j + 1] = A[j]
        j = j - 1
        A[j + 1] = key
```

Python code for the same will be as listed in following program :

Program 10.2 Program to sort a sequence using insertion sort.

```
aList = [15, 6, 13, 22, 3, 52, 2]
print ("Original list is :", aList)
for i in range(1, len(aList)) :
    key = aList[i]
    j = i - 1
    while j >= 0 and key < aList[j] :
        aList[j + 1] = aList[j]     #shift elements to right to make room for key
        j = j - 1
    else:
        aList[j + 1] = key
print ("List after sorting :", aList)
```

Output produced by above program is as shown below :

```
Original list is : [15, 6, 13, 22, 3, 52, 2]
List after sorting : [2, 3, 6, 13, 15, 22, 52]
```

Number of Operations

As we mentioned during bubble sort discussion that two costly operations are **comparisons** and **swapping/exchanges**. Let us concentrate on these two costly operations while considering number of operations in insertion sort. Look at insertion sort code given below with highlighted *comparison and exchange operations*.

```
:
for i in range(1, len(aList)) :
    key = aList[i]
    j = i - 1
    while j >= 0 and key < aList[j] :        # Comparison operation
        aList[j + 1] = aList[j]              # Exchange operation
        j = j - 1
    else:
        aList[j+1] = key
    :
```

The operations that affect the efficiency of insertion sort in terms of number of operations are:

(*i*) the comparison operation in the inner loop, and

(*ii*) the exchange operations in the inner loop.

Let us calculate number of operations of insertion sort with respect to each of these two operations for a sequence having N number of elements.

(*i*) **The number of comparisons :**

If you carefully go through the code, you will find that :

- during the first iteration of the outer loop, in the inner loop, there is **1 comparison**;
- during the second iteration of the outer loop, there are **at most 2 comparisons**;
- .....
- during the $(N-1)^{th}$ iteration of the outer loop, there are **at most $(N-1)$ comparisons.**

Thus, maximum number of comparisons :

$$1+2+\ldots+(N-2)+(N-1)=(N\times(N-1))/2=(N^2-N)/2<N^2$$

That means there can be maximum $\cong N^2$ comparisons in insertion sort.

(*ii*) **The number of exchanges :**

If you carefully go through the code, you will find that :

- during the first iteration of the outer loop, there is **at most 1 exchange**;
- during the second iteration of the outer loop, there are **at most 2 exchanges**;
- .....
- during the $(N-1)^{th}$ iteration of the outer loop, there are **at most (–1) exchanges.**

Thus , maximum number of exchanges :

$$1+2+\ldots+(N-2)+(N-1)=(N\times(N-1))/2=(N^2-N)/2<N^2$$

That means there can be maximum $< N^2$ exchanges in insertion sort.

Note that the number of comparisons and exchanges depends on the sequence to be sorted, that is insertion sort is sensitive to the input sequence.

There are 3 possibilities :

- In the best case, the sequence has all the elements in desired sort order.
- In the worst case, no element at the correct position *i.e.*, sequence is completely in reverse order.
- In the average case, the sequence will have a mix of these two : some elements are correct position and some are not.

Check Point 10.1

1. Name some commonly used sorting techniques.
2. Show the contents of following array after the second iteration of Bubble Sort : 35, 6, 8, 11, 15, 9, 7, 22, 41, 65
3. Considering the following key set : 42, 29, 74, 11, 65, 58, use insertion sort to sort the data in ascending order and indicate the sequences of steps required.

Now,

- In the **best case** when the sequence is already in desired sorted order, there will be maximum $N - 1$ number of comparisons and exchanges (*i.e.*, $N - 1$ ops).
- In the **worst case**, there will be maximum possible number ($< N^2$) of comparisons and exchanges (*i.e.*, $< N^2$ ops).

NOTE

The insertion sort algorithm is efficient for small sized sequences but with increase in size of the sequence, it becomes costlier in terms of number of operations.

Calculating number of operations is always beneficial for determining the efficiency of an algorithm, especially in cases where we have an option to choose from two or more algorithms that can perform same task. Studying efficiency of algorithms is a specialized study field of computer science, which you will learn about if you take up 'Computer Science' major in your higher education. Here, just having an idea about number of operations an algorithm performs, is sufficient for you. :).

Applications of Bubble Sort and Insertion Sort

Bubble is one of the simplest and easiest to implement sorting algorithm although it is not considered an efficient sorting algorithm. Even for smaller datasets, insertion sort has proven to be more efficient than bubble sort.

So, what does that mean ? Is bubble sort only a learning tool or has it got any practical application too in today's world when machines have excellent configurations and are capable of handling huge datasets.

In a rare situation where one has to *sort a dataset which is contained in a file on tape*, bubble sort can be the best option for sorting. Tape, as you must be knowing, lets you read data in sequential manner only (you need to read all previous records in order to reach a particular record). With bubble sort it is easier to sort as two successive records are to be read and compared at a time and even if one has to swap, then rewinding is not much. Though this situation is a rare situation but for such situation bubble sort is the ideal choice

Insertion sort, on the other hand, has been a preferred choice for sorting small datasets. It is easy to implement and at the same time, it does not require additional memory; it can be fast if the data is almost nearly sorted, which is great. You have read about practical usage of insertion sorting in sorting of answer sheets or for arranging clothes in the order of their size in showrooms, etc.

PriP SORTING WITH INSERTION SORT — Progress In Python 10.2

This PriP session aims at practice of sorting with Insertion Sort.

Please check the practical component-book – Progress in Computer Science with Python and fill it there in PriP 10.2 under Chapter 10 after practically doing it on the computer.

>>>❖<<<

LET US REVISE

- *Sorting of an array means arranging the array elements in a specified order.*
- *In bubble sort, the adjoining values are compared and exchanged if they are not in proper order. This process is repeated until the entire array is sorted.*
- *In insertion sort, each successive element is picked & inserted at an appropriate position in the previously sorted array.*

Solved Problems

1. *Given an array : 89, 20, 31, 56, 20. Sort this array in ascending order using (i) Bubble Sort (ii) Insertion Sort.*

Solution. Given array is 89, 20, 31, 56, 20

(i) Bubble Sort

```
I      89, 20, 31, 56, 20
II     20, 89, 31, 56, 20
III    20, 31, 89, 56, 20
IV     20, 31, 56, 89, 20
V      20, 31, 56, 20, 89
VI     20, 31, 56, 20, 89
VII    20, 31, 56, 20, 89
VIII   20, 31, 20, 56, 89
IX     20, 31, 20, 56, 89
X      20, 31, 20, 56, 89
XI     20, 20, 31, 56, 89
```

(Bold elements depict that they are to be compared in the next pass.)

(ii) Insertion Sort

```
I    89, 20, 31, 56, 20
II   20, 89, 31, 56, 20
III  20, 31, 89, 56, 20
IV   20, 31, 56, 89, 20
V    20, 20, 31, 56, 89
```

(As you can see that Bubble Sort is taking up more number of operations (as more iterations) than insertion sort to sort this same array (a very small array, though)

2. *Given the following steps to sort an array, which sort is this ?*

82	4	16	32	8
4	82	16	32	8
4	16	82	32	8
4	16	32	82	8
4	8	16	32	82

Solution. Bubble sort.

3. *Which of the following sorting methods will be the best if number of swappings done, is the only measure of efficienty, considering the fact that the sequence size is not very large ?*

(A) Bubble sort (B) Insertion sort

Solution. For smaller sized sequences, there are lesser number of swappings in insertion sort as compared to bubble sort.

4. *You are asked to sort 15 randomly generated numbers. You should prefer*

(A) Bubble sort (B) Insertion sort

Solution. Insertion sort is a better choice for small sized sequences.

5. *As part of the maintenance work, you are entrusted with the work of rearranging the library books in a shelf in proper order, at the end of each day. The ideal choice will be*

(A) Bubble sort (B) Insertion sort

Solution. Insertion sort

6. *Consider the follow list : [32, 33, 5, 2, 14, –4, 22, 39, 34, –9]*

Each of the following is a view of a sort in progress (after a few passes) of the above list. Identify the sorting algorithm used :

(a) [–4, 2, –9, 5, 14, 22, 32, 33, 34, 39] (b) [–9, –4, 2, 5, 14, 33, 22, 39, 34, 32]

Solution. (a) Bubble sort (b) Insertion sort

7. *What will the following list look like after 3 passes of insertion sort algorithm ? Show the list after every pass.*

16	19	11	15	10

Solution. Elements being compared are shown in black shaded cells.

First pass

16	19	11	15	10

No swapping –

16	19	11	15	10

Second Pass

16	19	11	15	10

16	19	11	15	10

i.e.,

11	16	19	15	10

Third Pass

11	16	19	15	10

11	16	19	15	10

11	15	16	19	10

8. *In which situation, would you prefer bubble sort over insertion sort ?*

Solution. Bubble sort would be ideal sorting technique for situations where data can be read only sequentially (*e.g.*, data stored on magnetic tapes). As bubble sort also compares two adjacent records, it will be suitable for such a situation. But, nowadays, such a situation is rare.