



SATHYABAMA

INSTITUTE OF SCIENCE AND TECHNOLOGY

(DEEMED TO BE UNIVERSITY)

Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE

www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

PYTHON PROGRAMMING – SBSA1202

UNIT I

Overview of Programming: Structure of a Python Program, Elements of Python.

1. OVERVIEW OF PROGRAMMING

Before getting into computer programming, let us first understand computer programs and what they do. A computer program is a sequence of instructions written using a Computer Programming Language to perform a specified task by the computer.

The two important terms that we have used in the above definition are -

- ✓ Sequence of instructions
- ✓ Computer Programming Language

To understand these terms, consider a situation when someone asks you about how to go to a nearby KFC. What exactly do you do to tell him the way to go to KFC? You will use Human Language to tell the way to go to KFC, something as, First go straight, after half a kilometer, take left from the red light and then drive around one kilometer and you will find KFC at the right. Here, you have used the English Language to give several steps to be taken to reach KFC. If they are followed in the following sequence, then you will reach KFC –

1. Go straight
2. Drive half kilometer
3. Take left
4. Drive around one kilometer
5. Search for KFC on your right side

Now, try to map the situation with a computer program. The above sequence of instructions is a Human Program written in English Language, which instructs on how to reach KFC from a given starting point. This same sequence could have been given in Spanish, Hindi, Arabic, or any other human language provided the person seeking direction knows any of these languages.

Now, let's go back and try to understand a computer program, which is a sequence of instructions written in a Computer Language to perform a specified task by the computer.

Python Programming Language –

```
print "Hello, World!"
```

The above computer program instructs the computer to print "Hello, World!" on the computer screen.

- ✓ A computer program is also called computer software, which can range from two lines to millions of lines of instructions.
- ✓ Computer program instructions are also called program source code and computer programming is also called program coding.
- ✓ A computer without a computer program is just a dump box; it is programs that make computers active.

As we have developed so many languages to communicate among ourselves, computer scientists have developed several computer-programming languages to provide instructions to the computer (i.e., to write computer programs). We will see several computer programming languages in the subsequent chapters.

Introduction to Computer Programming

If you understood what a computer program is, then we will say: the act of writing computer programs is called computer programming. As we mentioned earlier, there are hundreds of programming languages, which can be used to write computer programs, and the following are a few of them –

- Java
- C
- C++
- Python
- PHP
- Perl
- Ruby

Uses of Computer Programs

Today computer programs are being used in almost every field, household, agriculture, medical, entertainment, defense, communication, etc. Listed below are a few applications of computer programs –

- ✓ MS Word, MS Excel, Adobe Photoshop, Internet Explorer, Chrome, etc., are examples of computer programs.
- ✓ Computer programs are being used to develop graphics and special effects in movie making.
- ✓ Computer programs are being used to perform Ultrasounds, X-Rays, and other medical examinations.
- ✓ Computer programs are being used in our mobile phones for SMS, Chat, and voice communication.

Computer Programmer

Someone who can write computer programs or in other words, someone who can do computer programming is called a Computer Programmer. Based on computer programming language expertise, we can name a computer programmer as follows –

- ✓ C Programmer
- ✓ C++ Programmer
- ✓ Java Programmer
- ✓ Python Programmer

- ✓ PHP Programmer
- ✓ Perl Programmer
- ✓ Ruby Programmer

Algorithm

From a programming point of view, an algorithm is a step-by-step procedure to resolve any problem. An algorithm is an effective method expressed as a finite set of well-defined instructions. Thus, a computer programmer lists down all the steps required to resolve a problem before writing the actual code. Following is a simple example of an algorithm to find out the largest number from a given list of numbers –

1. Get a list of numbers $L_1, L_2, L_3, \dots, L_N$
2. Assume L_1 is the largest, $\text{Largest} = L_1$
3. Take the next number L_i from the list and do the following
4. If Largest is less than L_i
5. $\text{Largest} = L_i$
6. If L_i is the last number from the list then
7. Print value stored in Largest and come out
8. Else repeat same process starting from step 3

The above algorithm has been written crudely to help beginners understand the concept. You will come across more standardized ways of writing computer algorithms as you move on to advanced levels of computer programming.

Computer Programming - Basics

We assume you are well aware of English Language, which is a well-known Human Interface Language. English has a predefined grammar, which needs to be followed to write English statements correctly. Likewise, most of the Human Interface Languages (Hindi, English, Spanish, French, etc.) are made of several elements like verbs, nouns, adjectives, adverbs, propositions, and conjunctions, etc.

Similar to Human Interface Languages, Computer Programming Languages are also made of several elements. We will take you through the basics of those elements and make you comfortable using them in various programming languages. These basic elements include –

- ✓ Programming Environment
- ✓ Basic Syntax
- ✓ Data Types
- ✓ Variables
- ✓ Keywords
- ✓ Basic Operators
- ✓ Decision Making
- ✓ Loops

- ✓ Numbers
- ✓ Characters
- ✓ Arrays
- ✓ Strings
- ✓ Functions
- ✓ File I/O

Computer Programming - Environment

Though Environment Setup is not an element of any Programming Language, it is the first step to be followed before setting on to write a program. When we say Environment Setup, it simply implies a base on top of which we can do our programming. Thus, we need to have the required software setup, i.e., installation on our PC which will be used to write computer programs, compile, and execute them. For example, if you need to browse the Internet, then you need the following setup on your machine –

- ✓ A working Internet connection to connect to the Internet
- ✓ A Web browser such as Internet Explorer, Chrome, Safari, etc.

If you are a PC user, then you will recognize the following screenshot, which we have taken from the Internet Explorer while browsing tutorialspoint.com. Similarly, you will need the following setup to start with programming using any programming language.

- ✓ A text editor to create computer programs.
- ✓ A compiler to compile the programs into binary format.
- ✓ An interpreter to execute the programs directly.

In case you don't have sufficient exposure to computers, you will not be able to set up either of this software. So, we suggest you take the help from any technical person around you to set up the programming environment on your machine from where you can start. But for you, it is important to understand what these items are.

Text Editor

A text editor is software that is used to write computer programs. Your Windows machine must have a Notepad, which can be used to type programs. You can launch it by following these steps – Start Icon → All Programs → Accessories → Notepad → Mouse Click on Notepad

It will launch Notepad with the following window –

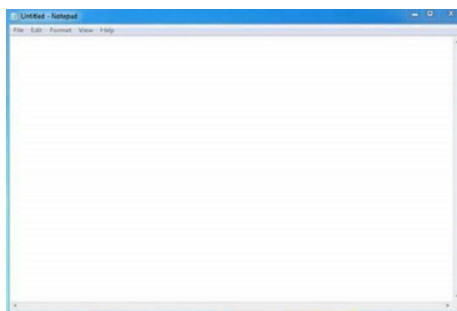


Fig 1.1 Note Pad

You can use this software to type your computer program and save it in a file at any location. You can download and install other good editors like Notepad++, which is freely available. If you are a Mac user, then you will have TextEdit or you can install some other commercial editor like BBEdit to start with.

Compiler?

You write your computer program using your favorite programming language and save it in a text file called the program file. Now let us try to get a little more detail on how the computer understands a program written by you using a programming language. The computer cannot understand your program directly given in the text format, so we need to convert this program into a binary format, which can be understood by the computer. The conversion from text program to binary file is done by another software called Compiler and this process of conversion from text formatted program to binary format file is called program compilation. Finally, you can execute a binary file to perform the programmed task. We are not going into the details of a compiler and the different phases of compilation.

The following flow diagram gives an illustration of the process –

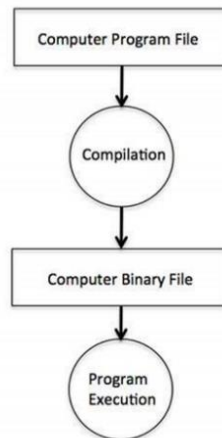


Fig 1.2 Compiler

So, if you are going to write your program in any such language, which needs compilation like C, C++, Java, and Pascal, etc., then you will need to install their compilers before you start programming.

Interpreter

We just discussed compilers and the compilation process. Compilers are required in case you are going to write your program in a programming language that needs to be compiled into binary format before its execution. There are other programming languages such as Python, PHP, and Perl, which do not need any compilation into binary format, rather an interpreter can be used to read such programs line by line and execute them directly without any further conversion.

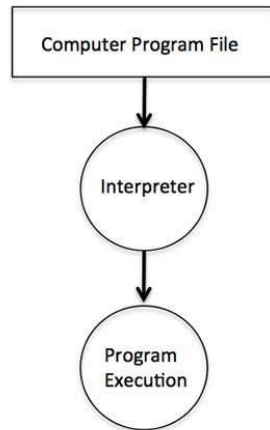


Fig 1.3 Interpreter

So, if you are going to write your programs in PHP, Python, Perl, Ruby, etc., then you will need to install their interpreters before you start programming.

Difference between C, C++, PYTHON

S.N.	Parameters	C	C++	PYTHON
1.	Language Type	procedure oriented programming	object oriented programming	both types of languages(pop & oop)
2.	Developer	Dennis Ritchie	Bjarne Stroustrup	Guido Van Rossum
3.	Language Level	Middle level language	High level language	High level language
4.	Building Block	Function driven	Object driven	Function, Object and Class driven
5.	Extensions	.c	.cpp	.PY
6.	Platform	Dependent	Independent	Independent
7.	Comment Style	/* */	//single line, /* */ for multi line	# single line, "" multi line comment ""
8.	Keywords	32	50	33
9.	Dynamic Variable	not support(int x =5)	not support(int x =5)	support(x =5) , no need to declare variables
10.	Data Security	not secure	secure(less than java)	secure (less than java)
11.	Coding Difference (Print Hello)	<pre>#include<stdio.h> int main() { printf("Hello"); return 0; }</pre> <p>Output:- Hello</p> <p>Note:- C is mostly used to develop system software.</p>	<pre>#include <iostream> using namespace std; int main() { cout << "Hello"; return 0; }</pre> <p>Output:- Hello</p> <p>Note:- C++ is also secured and used to solve many real time of problem.</p>	<pre>print("Hello")</pre> <p>Output:- Hello</p> <p>Note:- Python is future because it's fewer lines of code basis languages and It's have many library to solve different types of tasks.</p>

Table 1.1. Difference between C, C++, PYTHON

Real World Application of Python

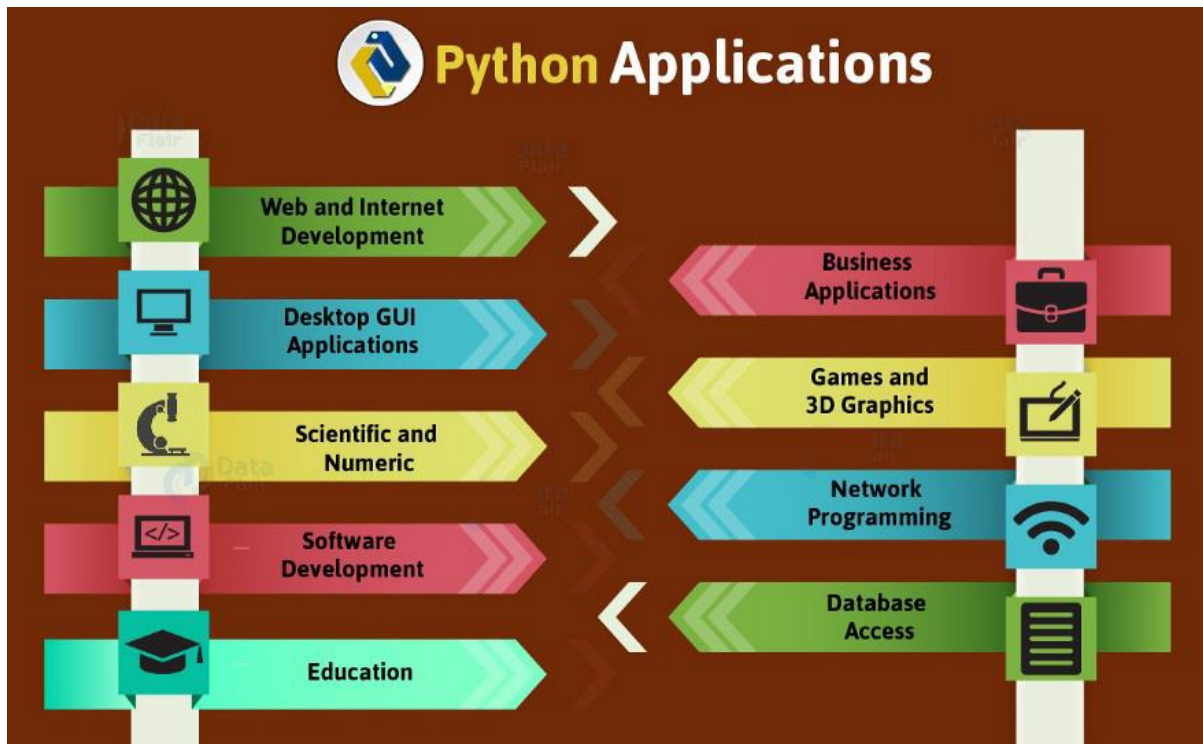


Fig 1.4 Python Application

1. Web and Internet Development

Python lets you develop a web application without too much trouble. It has libraries for internet protocols like HTML and XML, JSON, e-mail processing, FTP, IMAP, and easy-to-use socket interface. Yet, the package index has more libraries:

- Requests — An HTTP client library
- BeautifulSoup — An HTML parser
- Feedparser — For parsing RSS/Atom feeds
- Paramiko — For implementing the SSH2 protocol
- Twisted Python — For asynchronous network programming

We also have a gamut of frameworks available. Some of these are- Django, Pyramid. We also get microframeworks like flask and bottle. We've discussed these in our write-up on an Introduction to Python Programming. We can also write CGI scripts, and we get advanced content management systems like Plone and Django CMS.

2. Applications of Python Programming in Desktop GUI

Most binary distributions of Python ship with Tk, a standard GUI library. It lets you draft a user interface for an application. Apart from that, some toolkits are available:

wxWidgets

Kivy — for writing multitouch applications

Qt via pyqt or pyside

And then we have some platform-specific toolkits:

GTK+

Microsoft Foundation Classes through the win32 extensions

Delphi

3. Scientific and Numeric Applications

This is one of the very common applications of python programming. With its power, it comes as no surprise that python finds its place in the scientific community. For this, we have:

SciPy — A collection of packages for mathematics, science, and engineering.

Pandas- A data-analysis and -modeling library

IPython — A powerful shell for easy editing and recording of work sessions. It also supports visualizations and parallel computing.

Software Carpentry Course — It teaches basic skills for scientific computing and running bootcamps. It also provides open-access teaching materials.

Also, NumPy lets us deal with complex numerical calculations.

4. Software Development Application

Software developers make use of python as a support language. They use it for build-control and management, testing, and for a lot of other things:

SCons — for build-control

Buildbot, Apache Gump — for automated and continuous compilation and testing

Roundup, Trac — for project management and bug-tracking.

Roster of Integrated Development Environments

5. Python Applications in Education

Thanks to its simplicity, brevity, and large community, Python makes for a great introductory programming language. Applications of python programming in education has huge scope as it is a great language to teach in schools or even learn on your own. If you still haven't begun, we suggest you read up on what we have to say about the white and dark sides of Python. Also, check out Python Features.

6. Python Applications in Business

Python is also a great choice to develop ERP and e-commerce systems:

Tryton — A three-tier, high-level general-purpose application platform.

Odo — A management software with a range of business applications. With that, it's an all-rounder and forms a complete suite of enterprise-management applications in-effect.

7. Database Access

This is one of the hottest Python Applications.

With Python, you have:

- Custom and ODBC interfaces to MySQL, Oracle, PostgreSQL, MS SQL Server, and others. These are freely available for download.

- Object databases like Durus and ZODB

- Standard Database API

8. Network Programming

With all those possibilities, how would Python slack in network programming? It does provide support for lower-level network programming:

- Twisted Python — A framework for asynchronous network programming. We mentioned it in section 2.

- An easy-to-use socket interface

9. Games and 3D Graphics

Safe to say, this one is the most interesting. When people hear someone say they're learning Python, the first thing they get asked is — 'So, did you make a game yet?' PyGame, PyKyr are two frameworks for game-development with Python. Apart from these, we also get a variety of 3D-rendering libraries. If you're one of those game-developers, you can check out PyWeek, a semi-annual game programming contest.

10. Other Python Applications

These are some of the major Python Applications. Apart from what we just discussed, it still finds use in more places:

- Console-based Applications

- Audio — or Video- based Applications

- Applications for Images

- Enterprise Applications

- 3D CAD Applications

- Computer Vision (Facilities like face-detection and color-detection)

- Machine Learning

- Robotics

- Web Scraping (Harvesting data from websites)

- Scripting

- Artificial Intelligence

- Data Analysis (The Hottest of Python Applications)

2. PROGRAMMING STRUCTURE OF PYTHON

This explains the Programming structure of Python Programming. And the way you Divide a program into source files and mix parts into Total. By going with the process, we also discuss the topics, of Python Modules, objects, Imports. In practical Method, programs usually more than one file. For each one but simple scripts. so, every program will take multi-form systems. And can get a file by yourself with the coding process. In the first place, this explains the Programming structure of Python

Programming Structure of python

In General Python Program Consists of so many text files, which contain python statements. The program is designed as a single main, high file with one or more supplement files. Get hands-on experience of those files from live industry experts at python online training. In python high-level file has an Important path of control of your Program the file, you can start your application. The library tools are also known as Module files. These tools are implemented for making a collection of top-level files. High-level files use tools that are defined in Module files. And module files will Implement files that are Defined in other Modules. Coming to our point in python a file takes a module to get access to the tools it defines. And the tools made by a module type. The final thing is we take Modules and access attributes to their tools. In like manner, this shows the Programming structure of Python

Attributes and Imports:

The structure Python Program consists of three files such as a.py, b.py and c.py. The file model a.py is chosen for a high-level file. it is known as a simple text file of statements. And it can be executed from bottom to top when it is launched. Files b.py and c.py are modules. They are calculated as better text files of statements as well. But they are generally not started Directly. Identically these attributes define the Programming structure of Python.

Functions:

For example, b.py defines a function called spam. For external use. B.py has a python def statement to start the function. later operated by passing one or more values like the below.

```
Def spam(text): print text, 'spam'
```

If a.py wants to use spam, it has python statements like below

```
Import b b.spam ('gumby')
```

Statements:

Python import statement gives file a.py access to file b.py. it shows that “load fileb.py” and gives access to all its attributes by name b” import statements will execute and implement other files for at run-time. In python, the cross-file module is not updated until import statements are executed. The next part is statements will call the function spam. module b used by object attribute notation. B.spam means get value of name spam within object b. And we can implement a string in parenthesis if these files run by a.py. In regular if we see object. Attribute in total python scripts. Many objects have attributes traced by “python operators”. The process

of Importing is considered as general in total python. Any sort of file can get tools from any file. Getting of chains can go as deep as you can. By this Instance you will get it notified module a can import b and b can Import c, and c again Imports b. correspondingly these statements include Programming structure of Python

Modules:

If we take this as a part, python serves as the biggest company structure. Modules are having a top end of code. By coding components in module files used in any program files. If we take an example function b.spam is a regular purpose tool. We can again implement that in a different program. This is simply known as b.py from any other program files.

Standard library files:

Python has a large collection of modules known as the standard library.it contains 200 modules at the last count. It is platform-independent common programming works. Such as GUI Design, Internet, and network scripting. Text design matching, Operating system Interfaces. So, comparatively all the Above will explain the Programming structure of Python.

3. ELEMENTS OF PYTHON

The basic elements of Python are:

Keywords: and, assert, break, class, continue, def, del, elif, else, except, exec, finally, for, from, global, if, import, in, is, lambda, not, or, pass, print, raise, return, try, while, with, yield

Operators: + - * / % ** // > & | ^ ~ >= <> != ==

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

Data types: Numeric, Dictionary, Boolean, Set, Strings, List, Tuple

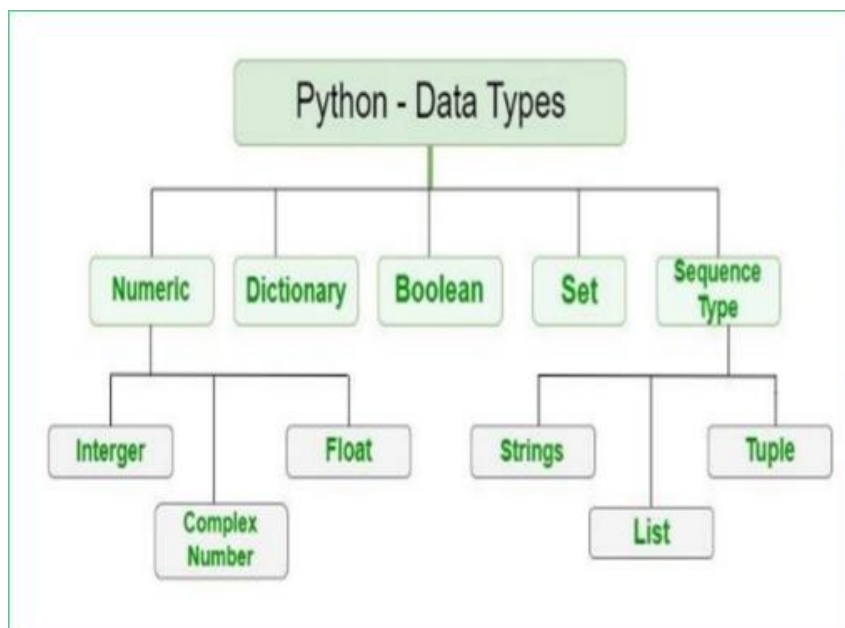


Fig 1.5 Data Type

Compound Data Type

- ✓ List
- ✓ Tuple
- ✓ Set
- ✓ Dictionary

Numeric

Integers

int – both +ve and –ve values can be represented

- ✓ Float – float class is used to represent floating-point number which is a real number with floating-point representation.
- ✓ float
- ✓ fractional point E.g.: 3.415, 5.15

Complex Numbers

- ✓ complex
- ✓ For example: $5 + 7j$
- ✓ where 5 is the real part and 7 is the imaginary part

Number Type Conversion

- ✓ `int(x)` - to convert x to a plain integer.
- ✓ `long(x)` -to convert x to a long integer.
- ✓ `float(x)` -to convert x to a floating-point number.
- ✓ `complex(x)` -to convert x to a complex number with real part x and imaginary part zero.
- ✓ `complex(x, y)` - to convert x and y to a complex number with real part x and imaginary part y. x and y are numeric expressions

Example

Program:

```
a = 10
print("Type of a: ", type(a))
b = 20.0
print("\nType of b: ", type(b))
c = 5 + 7j
print("\nType of c: ", type(c))
```

Output:

```
Type of a: <class 'int'>
Type of b: <class 'float'>
Type of c: <class 'complex'>
```

Mutable and Immutable Data Types

Example for mutable data types is: List, Set, and Dictionary

Examples for Immutable data types are Strings, tuple, int, float, bool, Unicode.

Immutable Objects: These are of in-built types like int, float, bool, string, Unicode, tuple. In simple words, an immutable object can't be changed after it is created.

```
# Python code to test that tuples are immutable
tuple1 = (0, 1, 2, 3)
tuple1[0] = 4
print(tuple1)
```

Output: Error

TypeError: 'tuple' object does not support item assignment

Mutable Objects : These are of type list, dict, set . Custom classes are generally mutable.

```
# Python code to test that lists are mutable
color = ["red", "blue", "green"]
print(color)
color[0] = "pink"
color[-1] = "orange"
print(color)
```

Output:

```
['red', 'blue', 'green']
['pink', 'blue', 'orange']
```

Boolean

- ✓ The Boolean data type has two built-in values True or False.
- ✓ It is denoted by the class bool.
- ✓ Note – True and False with capital ‘T’ and ‘F’ are valid boolean values.

```
# Python program to demonstrate Boolean type
print(type(True))
print(type(False))
print(type(true))
```

Output:

```
<class 'bool'>
<class 'bool'>
NameError: name 'false' is not defined
```

Sequence Type

- ✓ A sequence is an ordered collection of similar or different data.
- ✓ Using sequence, Multiple values can be efficiently stored in the data type.
- ✓ There are different types of sequence data type such as
- ✓ Strings ii) List iii) Tuple

Strings

- ✓ String is an array of bytes.
- ✓ Each byte represents a Unicode character.
- ✓ A string is a collection of one or more characters put in a single quote, double-quote, or triple quote.
- ✓ In python there is no character data type, a character is a string of length one.
- ✓ It is represented by str class.
- ✓ Individual characters of a string can be accessed by using the method of Indexing

Example

```
String1 = "VAL1003 STUDENTS"
print("Initial String: ")
print(String1)

# Printing First character
print("\nFirst character of String is: ")
print(String1[0])

# Printing Last character
print("\nLast character of String is: ")
print(String1[-1])

print("\n8th char: ")
print(String1[8])
```

```

# Updation or deletion of characters from a String is not
allowed

# because strings are immutable
string2 = 'I\'m doing "Python Progmg"'
print("\n Escaping Single Quote:" , string2)
string2 = " VAL1003 \"Python Progmg\" "
print("\n Escaping Double Quote:" , string2)

```

```

Initial String:
VAL1003 STUDENTS

First character of String is:
V

Last character of String is:
S

8th char:
S

Escaping Single Quote: I'm doing "Python Progmg"

Escaping Double Quote: VAL1003 "Python Progmg"

```

Strings: Sequence of characters inside single quotes or double-quotes.

E.g. myuniv = "Sathyabama !.."

PROGRAM:

```

# Python Program for String Manipulation
# Creating a String with single quotes, double quotes,
triple quotes
String1 = 'Welcome'
String2 = "Sathyabama"
String3 = '''CSE'''          # Triple Quotes allows multiple
lines
String4 = '''Welcome
                        To
                        Sathyabama'''

print("\nUsing Single quote")
print(String1)
print("\nUsing Double quote")
print(String2)

```



```
print("\nUsing Triple quote")
print(String3)
print("\nUsing Triple quote to print multiline")
print(String4)
```

```
Output:
Using Single quote
Welcome

Using Double quote
Sathyabama

Using Triple quote
CSE

Using Triple quote to print multiline
Welcome
    To
        Sathyabama
```

List

- ✓ The List is an ordered sequence of data items.
- ✓ It is one of the flexible and very frequently used data types in Python.
- ✓ All the items in a list are not necessary to be of the same data type.
- ✓ Declaring a list is straightforward method.
- ✓ Items in the list are just separated by commas and enclosed within brackets []

Example: `li=[10,2,3,45]`

`list1 =[3.141, 100, 'CSE', 'ECE', 'IT', 'EEE']`

List Methods

<code>list1.append(x)</code>	To add item x to the end of the list "list1"
<code>list1.reverse()</code>	Reverse the order of the element in the list "list1"
<code>list1.sort()</code>	To sort elements in the "list1"

- ✓ A single list may contain Data Types like Integers, Strings, as well as Objects.
- ✓ Lists are mutable.
- ✓ Lists are ordered and have a definite count.
- ✓ The list index starts with 0.
- ✓ Duplication of elements is possible in the list. The lists are implemented by the list class.

Example

#Python program to demonstrate Creation of List

```
# Creating a List
List = []
print("Intial blank List: ")
print(List)
```

Creating a List with the use of a String

```
List = ['VAL1003 STUDENTS']
print("\nList with the use of String: ")
print(List)
```

Creating a List with the use of multiple values

```
List = ["VAL1003", "PYTHON", "STUDENTS"]
print("\nList containing multiple values: ")
print(List[0])
print(List[2])
```

Creating a Multi-Dimensional List (By Nesting a list inside a List)

```
List1 = [['VAL1003', 'PYTHON'], ['STUDENTS']]
print("\nMulti-Dimensional List: ")
print(List1[0])
```

```
Initial blank List:
[]

List with the use of String:
['VAL1003 STUDENTS']

List containing multiple values:
VAL1003
STUDENTS

Multi-Dimensional List:
['VAL1003', 'PYTHON']
```

Example 2

#Methods used in a List

#Append an element (ADD an element)

```
List.append(4)
print("\nList after Adding a number: ")
print(List)
```

Addition of Element at specific Position

(using Insert Method)

```
List.insert(2, 12)
print(List)
List.insert(0, 'SATHYABAMA')
print("\nList after performing Insert Operation: ")
print(List)
```

Addition of multiple elements to the List at the end (using Extend Method)

```
List.extend([8, 'GOOD', 'ALWAYS'])
print("\nList after performing Extend Operation: ")
print(List)
```

accessing a element from the list using index number

```
print("Accessing element from the list")
print(List[0])
print(List[2])
```

```
List after Adding a number:
['VAL1003', 'PYTHON', 'STUDENTS', 4]
['VAL1003', 'PYTHON', 12, 'STUDENTS', 4]

List after performing Insert Operation:
['SATHYABAMA', 'VAL1003', 'PYTHON', 12, 'STUDENTS', 4]

List after performing Extend Operation:
['SATHYABAMA', 'VAL1003', 'PYTHON', 12, 'STUDENTS', 4, 8, 'GOOD', 'ALWAYS']
Accessing element from the list
SATHYABAMA
PYTHON
```

accessing an element using negative indexing

```
print("Accessing element using negative indexing")

# print the last element of list
print(List[-1])
# print the third last element of a list
print(list[-3])
List1=[1,2,3,4,5,6,7,8]
print("Original List")
print(List1)
```

Removing elements from List using Remove() method

```
List1.remove(2)
print("\nList after Removal of element: ")
print(List1)
List1.pop()
print("\nList after popping an element: ")
print(List1)
```

Removing element at a specific location from the set using the pop() method

```
List1.pop(2)
print("\nList after popping a specific element: ")
print(List1)
```

```
Accessing element using negative indexing
ALWAYS
8
Original List
[1, 2, 3, 4, 5, 6, 7, 8]

List after Removal of element:
[1, 3, 4, 5, 6, 7, 8]

List after popping an element:
[1, 3, 4, 5, 6, 7]

List after popping a specific element:
[1, 3, 5, 6, 7]
```

Tuple

- ✓ A tuple is also an ordered sequence of items of different data types like a list.
- ✓ But, in a list data can be modified even after the creation of the list whereas Tuples are immutable and cannot be modified after creation.
- ✓ **Example:** `t = (50,'python', 2+3j)`

List	Tuple
<pre>>>> list1[12,45,27] >>> list1[1] = 55 >>> print(list1) >>> [12,55,27]</pre>	<pre>>>> t1 = (12,45,27) >>> t1[1] = 55 >>> Generates Error Message # Because Tuples are immutable</pre>

- ✓ A tuple is also an ordered sequence of items of different data types like a list.
- ✓ But, in a list data can be modified even after the creation of the list whereas Tuples are immutable and cannot be modified after creation.
- ✓ **Example:** `t = (50,'python', 2+3j)`

Example

Python program to demonstrate creation of Tuple

Creating an empty tuple

```
tuple1 = ()  
print("Initial empty Tuple: ")  
print (tuple1)
```

Creating a Tuple with the use of Strings

```
tuple1 = ('VAL1003', 'PYTHON')  
print("\nTuple with the use of String: ")  
print(tuple1)
```

Creating a Tuple with the use of list

```
list1 = [1, 2, 4, 5, 6]  
print("\nTuple using List: ")  
print(tuple(list1))
```

Creating a Tuple with the use of built-in function

```
tuple1 = tuple('SATHYABAMA')  
print("\nTuple with the use of function: ")  
print(tuple1)
```

```
Initial empty Tuple:  
(  
  
Tuple with the use of String:  
( 'VAL1003', 'PYTHON' )  
  
Tuple using List:  
(1, 2, 4, 5, 6)  
  
Tuple with the use of function:  
( 'S', 'A', 'T', 'H', 'Y', 'A', 'B', 'A', 'M', 'A' )
```

Python program to demonstrate creation of Tuple

Creating a Tuple with nested tuples

```
Tuple1 = (0, 1, 2, 3)
Tuple2 = ('python', 'students')
Tuple3 = (Tuple1, Tuple2)
print("\nTuple with nested tuples: ")
print(Tuple3)
```

Accessing element using indexing

```
print("Frist element of tuple")
print(Tuple1[0])
```

Accessing element from last -- negative indexing

```
print("\nLast element of tuple")
print(Tuple1[-1])

print("\nThird last element of tuple")
print(Tuple1[-3])
```

#updation or deletion is not possible(as it is immutable)

```
Tuple1[0]=10
del Tuple1[2]
```

```
Tuple with nested tuples:
((0, 1, 2, 3), ('python', 'students'))
Frist element of tuple
0

Last element of tuple
3

Third last element of tuple
1
```

Set

- ✓ The Set is an unordered collection of unique data items.
- ✓ Items in a set are not ordered, separated by a comma and enclosed inside { } braces.
- ✓ Sets help perform operations like union and intersection.
- ✓ The major advantage of using a set, as opposed to a list, is that it has a highly optimized method for checking whether a specific element is contained in the set.

List	Set
<pre>>>> L1 = [1,20,25] >>> print(L1[1]) >>> 20</pre>	<pre>>>> S1= {1,20,25,25} >>> print(S1) >>> {1,20,25} >>> print(S1[1]) >>>Error , Set object does not support indexing.</pre>

Python program to demonstrate Set in Python

```
set1 = set()
print("Intial blank Set: ")
print(set1)
```

Creating a Set with the use of a String

```
set1 = set("Welcome to Python")
print("\nSet with the use of String: ")
print(set1)
```

Creating a Set with the use of a List

```
set1 = set(["Python", "is", "Simple"])
print("\nSet with the use of List: ")
print(set1)
```

Removing element from the Set using the pop() method

```
set1.pop()
print("\nSet after popping an element: ")
print(set1)
```

Removing all the elements from Set using clear() method

```
set1.clear()
print("\nSet after clearing all the elements: ")
print(set1)
```


Initial blank Set:

```
set()
```

Set with the use of String:

```
{'t', ' ', 'h', 'o', 'e', 'm', 'W', 'P', 'c', 'y', 'n', 'l'}
```

Set with the use of List:

```
['Python', 'is', 'Simple']
```

Set after popping an element:

```
{2, 3, 4, 7, 10, 11, 12}
```

Set after clearing all the elements:

```
set()
```

```
# Creating a Set with a mixed type of values
# (Having numbers and strings)
set1 = set([1, 2, 'Python', 4, 'is', 6, 'simple'])
print("\nSet with the use of Mixed Values")
print(set1)

# Addition of elements in a Set
set1 = set()
print("Initial blank Set: ")
print(set1)

# Adding element and tuple to the Set
set1.add(8)
set1.add(9)
set1.add((6, 7))
print("\nSet after Addition of Three elements: ")
print(set1)

# Addition of elements to the Set using Update function
set1.update([10, 11])
print("\nSet after Addition of elements using Update: ")
print(set1)
```

Set with the use of Mixed Values

```
{1, 2, 4, 6, 'is', 'Python', 'simple'}
```

Initial blank Set:

```
set()
```

Set after Addition of Three elements:

```
{8, 9, (6, 7)}
```

Set after Addition of elements using Update:

```
{8, 9, (6, 7), 10, 11}
```

```
# Accessing of elements in a set
# Creating a set
set1 = set(["Python", "is", "Excellent"])
print("\nInitial set")
print(set1)
```

```
print("\nElements of set: ")
for i in set1:
    print(i, end=" ")

# Checking the element using in keyword
print("Great" in set1)

# Deletion of elements in a Set

# Creating a Set
set1 = set([1, 2, 3, 4, 5, 6,
           7, 8, 9, 10, 11, 12])
print("Initial Set: ")
print(set1)

# Removing elements from Set using Remove() method
set1.remove(5)
set1.remove(6)
print("\nSet after Removal of two elements: ")
print(set1)

# Removing elements from Set using Discard() method
set1.discard(8)
set1.discard(9)
print("\nSet after Discarding two elements: ")
print(set1)
```

```
Initial set
{'Python', 'is', 'Excellent'}
```

```
Elements of set:
Python is Excellent False
```

```
Initial set
{'Python', 'is', 'Excellent'}
```

```
Elements of set:
Python is Excellent False
```

Dictionary

- ✓ Dictionary is an unordered collection of data values.
- ✓ It is used to store data values like a map.
- ✓ Dictionary holds the key: value pair.
- ✓ Key-value is provided in the dictionary to make it more optimized.
- ✓ Each key-value pair is separated by a colon :, whereas each key is separated by a 'comma'.
- ✓ Example: dict={1:"Jan", 2:"Feb", 3:"March"}

```
# Creating a Dictionary
# with Integer Keys
Dict = {1: 'Python', 2: 'Is', 3: 'Powerful'}
print("\nDictionary with the use of Integer Keys: ")
print(Dict)
# Creating a Dictionary
# with Mixed keys
Dict = {'Name': 'Python', 1: [1, 2, 3, 4]}
print("\nDictionary with the use of Mixed Keys: ")
print(Dict)

# Creating a Dictionary
# with dict() method
Dict = dict({1: 'Python', 2: 'Is', 3: 'Efficient'})
print("\nDictionary with the use of dict(): ")
print(Dict)
```

Dictionary with the use of Integer Keys:

```
{1: 'Python', 2: 'Is', 3: 'Powerful'}
```

Dictionary with the use of Mixed Keys:

```
{'Name': 'Python', 1: [1, 2, 3, 4]}
```

Dictionary with the use of dict():

```
{1: 'Python', 2: 'Is', 3: 'Efficient'}
```

```
# Creating a Dictionary
# with each item as a Pair
Dict = dict([(1, 'Python'), (2, 'Programming')])
print("\nDictionary with each item as a pair: ")
print(Dict)
# Creating an empty Dictionary
Dict = {}
print("Empty Dictionary: ")
print(Dict)
# Adding elements one at a time
Dict[0] = 'Python'
Dict[2] = 'Program'
Dict[3] = 1
print("\nDictionary after adding 3 elements: ")
print(Dict)
# Updating existing Key's Value
Dict[2] = 'Welcome'
print("\nUpdated key value: ")
print(Dict)
# Python program to demonstrate
# accessing a element from a Dictionary
```

Dictionary with each item as a pair:

```
{1: 'Python', 2: 'Programming'}
```

Empty Dictionary:

```
{}
```

Dictionary after adding 3 elements:

```
{0: 'Python', 2: 'Program', 3: 1}
```

Updated key value:

```
{0: 'Python', 2: 'Welcome', 3: 1}
```

```
# Creating a Dictionary
```

```
Dict = {1: 'Python', 'name': 'Is', 3: 'Case-Sensitive'}
```

```
    # accessing a element using key
```

```
print("Accessing a element using key:")
```

```
print(Dict['name'])
```

```
# accessing a element using get() method
```

```
print("Accessing a element using get:")
```

```
print(Dict.get(3))
```

```
# Initial Dictionary
```

```
Dict = { 5 : 'Welcome', 6 : 'To', 7 : 'Python',
```

```
    'A' : {1 : 'Python', 2 : 'Is', 3 : 'Simple'},
```

```
    'B' : {1 : 'Python', 2 : 'Prarming'}}
```

```
print("Initial Dictionary: ")
```

```
print(Dict)
```

```
# Deleting a Key value
```

```
del Dict[6]
```

```
print("\nDeleting a specific key: ")
```

```
print(Dict)
```

Accessing a element using key:

Is

Accessing a element using get:

Case-Sensitive

Initial Dictionary:

```
{5: 'Welcome', 6: 'To', 7: 'Python', 'A': {1: 'Python', 2: 'Is', 3: 'Simple'}, 'B': {1: 'Python', 2: 'Pramming'}}
```

Deleting a specific key:

```
{5: 'Welcome', 7: 'Python', 'A': {1: 'Python', 2: 'Is', 3: 'Simple'}, 'B': {1: 'Python', 2: 'Pramming'}}
```

```
# Deleting a Key
# using pop()
Dict.pop(5)
print("\nPopping specific element: ")
print(Dict)
# Deleting an arbitrary Key-value pair using popitem()
Dict.popitem()
print("\nPops an arbitrary key-value pair: ")
print(Dict)
# Deleting entire Dictionary
Dict.clear()
print("\nDeleting Entire Dictionary: ")
print(Dict)
```

Popping specific element:

```
{7: 'Python', 'A': {1: 'Python', 2: 'Is', 3: 'Simple'}, 'B': {1: 'Python', 2: 'Pramming'}}
```

Pops an arbitrary key-value pair:

```
{7: 'Python', 'A': {1: 'Python', 2: 'Is', 3: 'Simple'}}
```

Deleting Entire Dictionary:

```
{}
```

Difference Between List Tuple Set and Dictionary

Data Structure	Ordered	Mutable	Constructor	Example
List	Yes	Yes	[] or list()	[5.7, 4, 'yes', 5.7]
Tuple	Yes	No	() or tuple()	(5.7, 4, 'yes', 5.7)
Set	No	Yes	{ }* or set()	{5.7, 4, 'yes'}
Dictionary	No	Yes**	{ } or dict()	{'Jun': 75, 'Jul': 89}

QUESTION BANK

1. What is an algorithm? Explain characteristics of an algorithm.
2. What are the features and applications of Python?
3. What is data type? List out the types of data types with example
4. Explain variable assignment with suitable example.
5. Describe an algorithm for the following [7+5M]
 - i) Prime number or not
 - ii) Odd or even
6. Define Flowchart and explain symbols used in flowchart with example.
7. What is dictionary? Explain the methods available in dictionary.
8. Differentiate between the tuple and sets in python
9. What are the steps involved in algorithm development process?
10. Define Input, Output, Assignment statement
11. What is a program?
12. Define assignment statement
13. List the data types in python.
14. Point out any 5-programming language
15. Explain list

TEXT /REFERENCE BOOKS:

1. Allen Downey, Jeffrey Elkner, Chris Meyers. How to think like a computer scientist learning with Python / 1st Edition, 2012
2. Kenneth A. Lambert, The Fundamentals of Python: First Programs, 2011, Cengage Learning, ISBN: 978- 1111822705

USEFUL WEBSITES:

1. <http://docs.python.org/3/tutorial/index.html>
2. <http://www.ibiblio.org/g2swap/byteofpython/read/>



SATHYABAMA

**INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)**

Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE

www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

PYTHON PROGRAMMING – SBSA1202

UNIT II

Introduction to Python: Python Interpreter, Using Python as calculator, Python shell, Indentation. Atoms, Identifiers and keywords, Literals, Strings, Operators (Arithmetic operator, Relational operator, Logical or Boolean operator, Assignment, Operator, Ternary operator, Bit wise operator, Increment or Decrement operator).

1. INTRODUCTION TO PYTHON

A program performs a task in the computer. But, in order to be executed, a program must be written in the machine language of the processor of a computer. Unfortunately, it is extremely difficult for humans to read or write a machine language program. This is because a machine language is entirely made up of sequences of bits. However, high level languages are close to natural languages like English and only use familiar mathematical characters, operators and expressions. Hence, people prefer to write programs in high level languages like C, C++, Java, or Python. A high-level program is translated into machine language by translators like compiler or interpreter.

ABOUT PYTHON

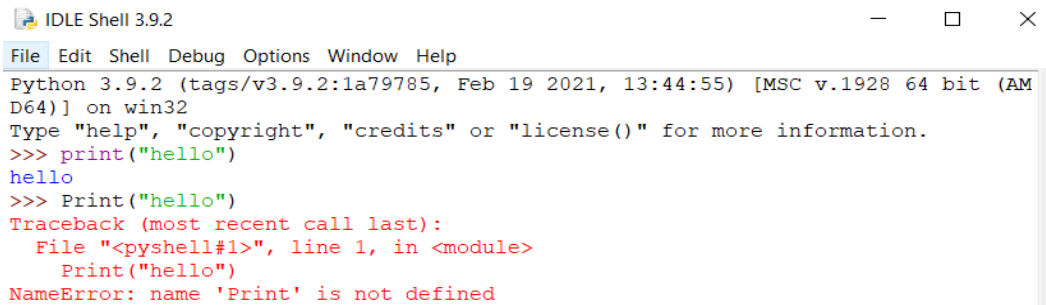
Python is a high-level programming language that is translated by the python interpreter. An interpreter works by translating line-by-line and executing. Python was developed by Guido-Van-Rossum in 1990, at the National Research Institute for Mathematics and Computer Science in Netherlands. Python doesn't refer to the snake but was named after the famous British comedy troupe, "Monty Python's Flying Circus".

The following are some of the features of Python:

- **Python is an Open Source:** It is freely downloadable
- **Python is portable:** It runs on different operating systems / platforms Python has automatic memory management
- **Python is flexible** with both procedural oriented and object- oriented programming
- **Python is easy to learn,** read and maintain
- It is very flexible with the console program, Graphical User Interface (GUI) applications, Web related programs etc.

POINTS TO REMEMBER WHILE WRITING A PYTHON PROGRAM

- **Case sensitive:** Example - In case of print statement use only lower case and not upper case



```
Python 3.9.2 (tags/v3.9.2:1a79785, Feb 19 2021, 13:44:55) [MSC v.1928 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> print("hello")
hello
>>> Print("hello")
Traceback (most recent call last):
  File "<pysshell#1>", line 1, in <module>
    Print("hello")
NameError: name 'Print' is not defined
```

Fig. 2.1 Single or double quotes

- Punctuation is not required at end of the statement
- In case of string use single or double quotes i.e. ‘ ‘ or “ ”
- Must use proper indentation: The screen shots given below show, how the value of “i” behaves with indentation and without indentation.

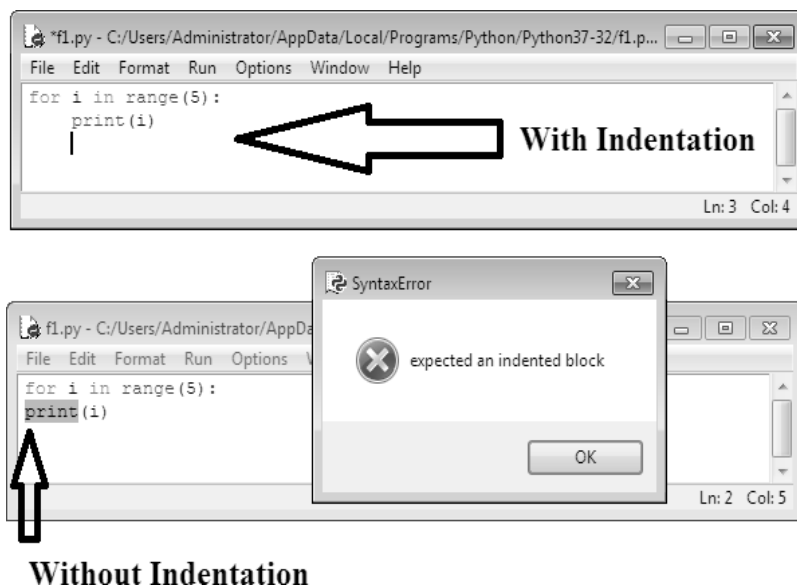


Fig. 2.2 With and without Indentation

Special characters like (,) , # etc. are used

() - Used in opening and closing parameters of functions

- The Pound sign is used to comment a line

2. PYTHON INTERPRETER

Python Program can be executed in two different modes:

1. Interactive mode programming
2. Script mode programming

1. Interactive Mode Programming

It is a command line shell which gives immediate output for each statement, while keeping previously fed statements in active memory. This mode is used when a user wishes to run one single line or small block of code. It runs very quickly and gives instant output. A sample code is executed using interactive mode as below.

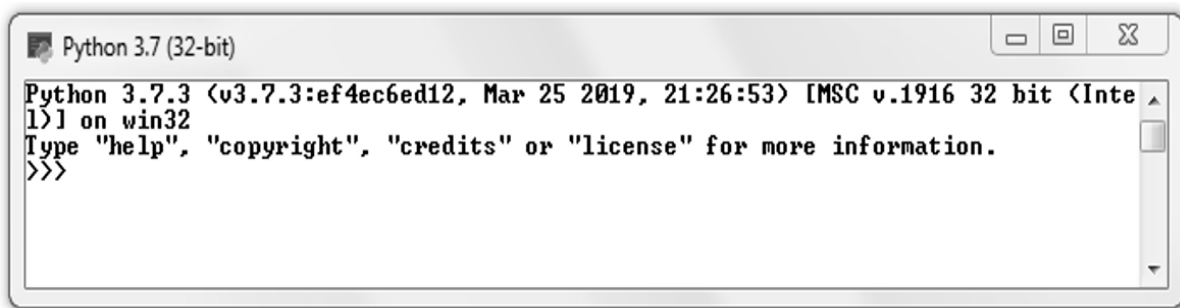


Fig. 2.3 Command Prompt

Interactive mode can also be opened using the following ways:

From command prompt `c :> users\\...>python`

The symbol “>>>” in the above screen indicates that the Python environment is in interactive mode.

- i. From the start menu select Python (As shown below)



Fig. 2.4 Python in Start Menu

2. Script Mode Programming

When the programmer wishes to use more than one line of code or a block of code, script mode is preferred. The Script mode works the following way:

- Open the Script mode
- Type the complete program. Comment, edit if required.
- Save the program with a valid name.
- Run

Correct errors, if any, Save and Run until proper output

The above steps are described in detail below:

- To open script mode, select the menu “IDLE (Python 3.7 32-bit)” from start menu

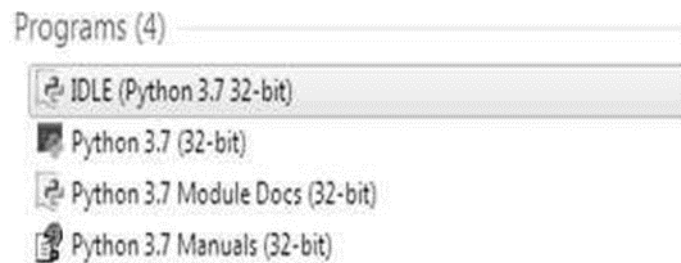


Fig. 2.5 IDLE in Start Menu

- After clicking on the menu “IDLE (Python 3.7 32-bit)”, a new window with the text Python 3.7.3 shell will be opened as shown below:
- ✓ **Select File : New**, to open editor. Type the complete program.

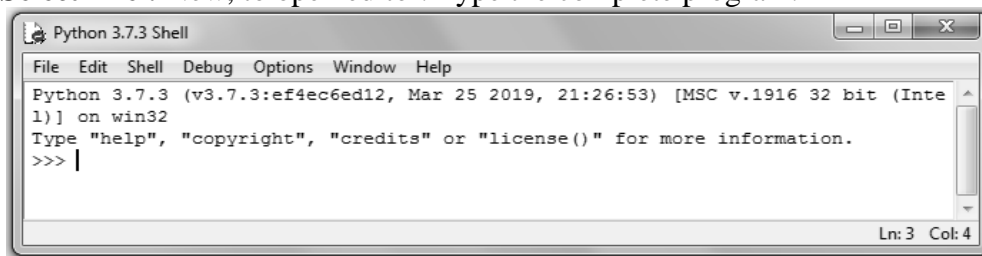


Fig. 2.6 Python 3.7.3 Shell

- ✓ **Select File again**; Choose Save.
- ✓ This will automatically save the file with an extension “.py”.
- ✓ **Select Run**: Run Module or Short Cut Key F5 (As shown in the screen below)

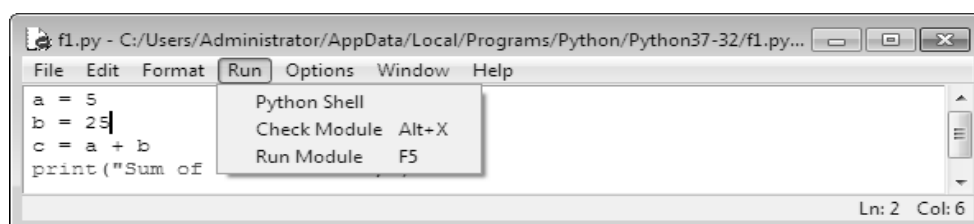


Fig. 2.7 Run Module

The output of the program will be displayed as below:

```
>> Sum of a and b is: 30
```

3. USING PYTHON AS A CALCULATOR

Let's try some simple Python commands. Start the interpreter and wait for the primary prompt, >>>.

I. Numbers

The interpreter acts as a simple calculator: you can type an expression at it and it will write the value. Expression syntax is straightforward: the operators +, -, * and / work just like in most other languages (for example, Pascal or C); parentheses (()) can be used for grouping.

For example:

```
>>>
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5*6) / 4
5.0
>>> 8 / 5 # division always returns a floating point number
1.6
```

The integer numbers (e.g., 2, 4, 20) have type int, the ones with a fractional part (e.g., 5.0, 1.6) have type float.

Division (/) always returns a float. To do floor division and get an integer result (discarding any fractional result) you can use the // operator; to calculate the remainder you can use %:

```
>>>
>>> 17 / 3 # classic division returns a float
5.666666666666667
>>>
>>> 17 // 3 # floor division discards the fractional
part
5
>>> 17 % 3 # the % operator returns the remainder of
the division
2
```

```
>>> 5 * 3 + 2 # result * divisor + remainder
17
```

With Python, it is possible to use the `` operator to calculate powers 1:**

```
>>>
>>> 5 ** 2 # 5 squared
25
>>> 2 ** 7 # 2 to the power of 7
128
```

The equal sign (=) is used to assign a value to a variable. Afterwards, no result is displayed before the next interactive prompt:

```
>>>
>>> width = 20
>>> height = 5 * 9
>>> width * height
900
```

If a variable is not “defined” (assigned a value), trying to use it will give you an error:

```
>>>
>>> n # try to access an undefined variable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

There is full support for floating point; operators with mixed type operands convert the integer operand to floating point:

```
>>>
>>> 4 * 3.75 - 1
14.0
```

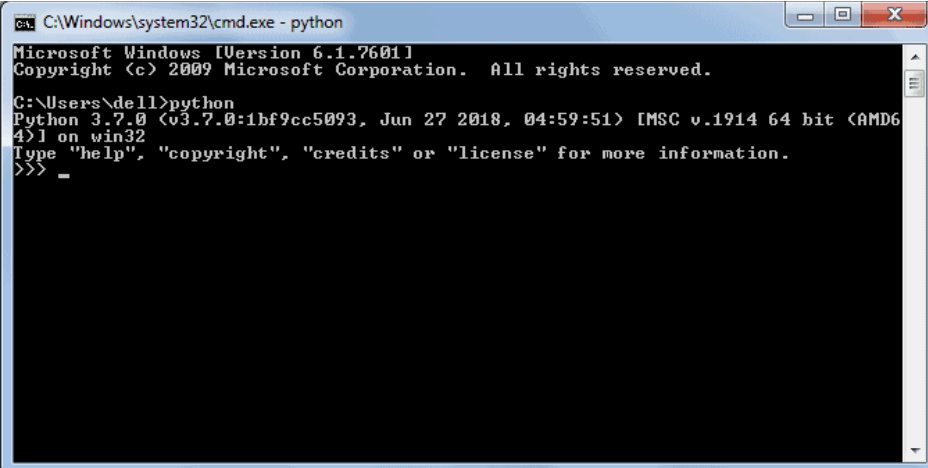
In interactive mode, the last printed expression is assigned to the variable `_`. This means that when you are using Python as a desk calculator, it is somewhat easier to continue calculations, for example:

```
>>>
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
>>> round(_, 2)
113.06
```

This variable should be treated as read-only by the user. Don't explicitly assign a value to it — you would create an independent local variable with the same name masking the built-in variable with its magic behavior. In addition to int and float, Python supports other types of numbers, such as Decimal and Fraction. Python also has built-in support for complex numbers, and uses the j or J suffix to indicate the imaginary part (e.g., 3+5j).

4. PYTHON SHELL/REPL

Python is an interpreter language. It means it executes the code line by line. Python provides a Python Shell, which is used to execute a single Python command and display the result. It is also known as REPL (Read, Evaluate, Print, Loop), where it reads the command, evaluates the command, prints the result, and loop it back to read the command again. To run the Python Shell, open the command prompt or power shell on Windows and terminal window on mac, write python and press enter. A Python Prompt comprising of three greater-than symbols >>> appears, as shown below.

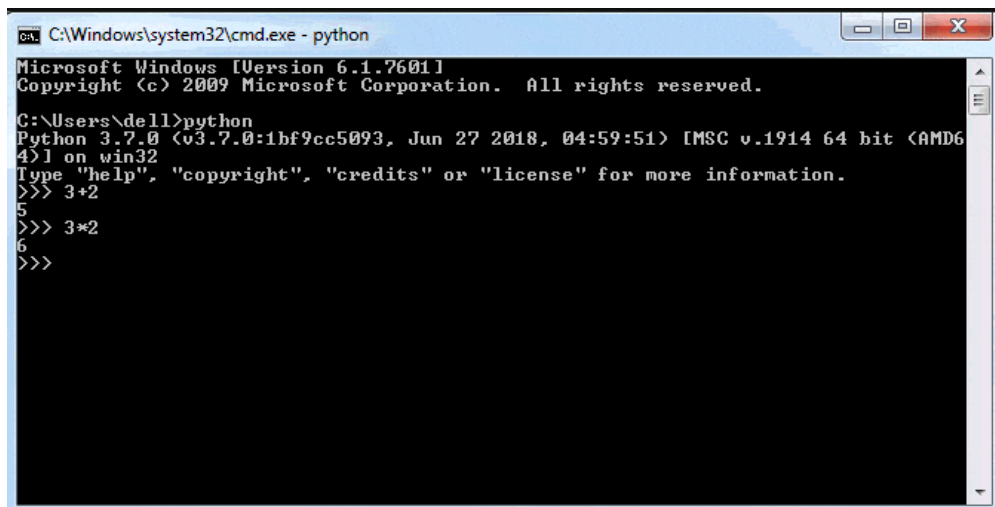


```
C:\Windows\system32\cmd.exe - python
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\dell>python
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:59:51) [MSC v.1914 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> _
```

Fig. 2.8 Python Shell/REPL

Now, you can enter a single statement and get the result. For example, enter a simple expression like $3 + 2$, press enter and it will display the result in the next line, as shown below.



```
C:\Windows\system32\cmd.exe - python
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\dell>python
Python 3.7.0 (tags/v3.7.0:1bf9cc5093, Jun 27 2018, 04:59:51) [MSC v.1914 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more
>>> 3+2
5
>>> 3*2
6
>>>
```

Fig. 2.9 Python Shell simple expression

Execute Python Script

As you have seen above, Python Shell executes a single statement. To execute multiple statements, create a Python file with extension `.py`, and write Python scripts

For example, enter the following statement in a text editor such as Notepad.

Example: `myPythonScript.py`

```
print ("This is Python Script.")
print ("Welcome to Python")
```

Save it as `myPythonScript.py`, navigate the command prompt to the folder where you have saved this file and execute the `python myPythonScript.py` command, as shown below. It will display the result. Thus, you can execute Python expressions and commands using Python REPL to quickly execute Python code.

5. INDENTATION IN PYTHON

Indentation is a very important concept of Python because without proper indenting the Python code, you will end up seeing `IndentationError` and the code will not get compiled.

Indentation

In simple terms indentation refers to adding white space before a statement. But the question arises is it even necessary? To understand this, consider a situation where you are reading a

book and all of a sudden, all the page numbers from the book went missing. So, you don't know, where to continue reading and you will get confused. This situation is similar with Python. Without indentation, Python does not know which statement to execute next or which statement belongs to which block. This will lead to IndentationError.

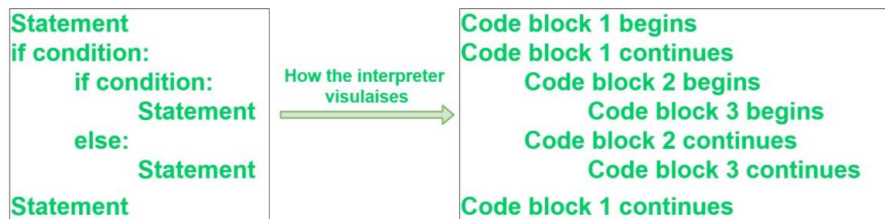


Fig. 2.10 Indentation

- Statement (line 1), if condition (line 2), and statement (last line) belongs to the same block which means that after statement 1, if condition will be executed. and suppose the if condition becomes False then the Python will jump to the last statement for execution.
- The nested if-else belongs to block 2 which means that if nested if becomes False, then Python will execute the statements inside the else condition.
- Statements inside nested if-else belongs to block 3 and only one statement will be executed depending on the if-else condition.

Python indentation is a way of telling a Python interpreter that the group of statements belongs to a particular block of code. A block is a combination of all these statements. Block can be regarded as the grouping of statements for a specific purpose.

Most of the programming languages like C, C++, Java use braces { } to define a block of code. Python uses indentation to highlight the blocks of code. Whitespace is used for indentation in Python. All statements with the same distance to the right belong to the same block of code. If a block has to be more deeply nested, it is simply indented further to the right. You can understand it better by looking at the following lines of code.

Example 1:

```
# Python program showing indentation
site = 'gfg'
if site == 'gfg':
    print('Logging on to yahoo...')
else:
    print('retype the URL.')
print('All set !')
```

Output:

```
Logging on to yahoo...  
All set !
```

The lines `print('Logging on to yahoo...')` and `print('retype the URL.')` are two separate code blocks. The two blocks of code in our example `if`-statement are both indented four spaces. The final `print('All set!')` is not indented, and so it does not belong to the `else`-block.

Example 2:

```
j = 1  
while(j<= 5):  
    print(j)  
    j = j + 1
```

Output:

```
1  
2  
3  
4  
5
```

To indicate a block of code in Python, you must indent each line of the block by the same whitespace. The two lines of code in the `while` loop are both indented four spaces. It is required for indicating what block of code a statement belongs to.

For example, `j=1` and `while(j<=5):` is not indented, and so it is not within `while` block. So, Python code structures by indentation. Python uses 4 spaces as indentation by default. However, the number of spaces is up to you, but a minimum of 1 space has to be used.

6. ATOM PYTHON TEXT EDITOR

Atom is free and open source. It is a desktop application which is designed to serve Python developers. The most basic way to create and run a Python program is to create an empty file with a `.py` extension and then point to that file from the command line with `python filename.py`. Alternatively, you can use IDLE which comes as a default application along with Python to execute your code. Atom does not have features in the traditional sense, it creates packages that add to its hackable core. These packages provide features like auto-complete, code lines, and code highlighters.

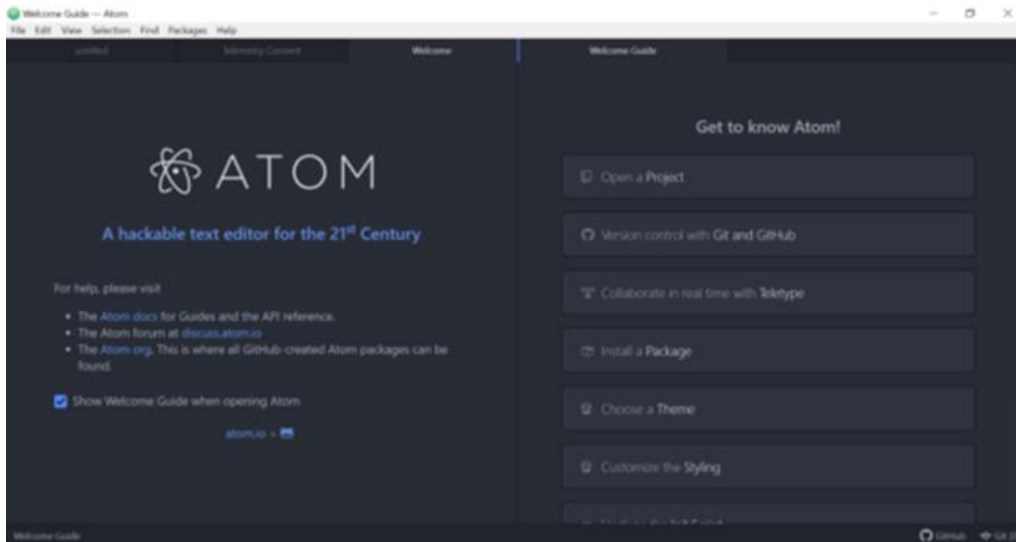


Fig. 2.11 Atom IDE

Atom is an open-source text editor for multiple platforms, which supports packages developed in Node.js and has support for Git version control. Most of the packages are freely available and built by open-source communities. It is developed and maintained by GitHub, built using web technologies as a desktop application.

7. PYTHON IDENTIFIERS

An identifier is a name given to entities like class, functions, variables, etc. It helps to differentiate one entity from another.

Rules for writing identifiers

1. Identifiers can be a combination of letters in lowercase (a to z) or uppercase (A to Z) or digits (0 to 9) or an underscore `_`. Names like `myClass`, `var_1` and `print_this_to_screen`, all are valid example.
2. An identifier cannot start with a digit. `1variable` is invalid, but `variable1` is a valid name.
3. Keywords cannot be used as identifiers.

```
global = 1
```

Output

```
File "<interactive input>", line 1
  global = 1
    ^
SyntaxError: invalid syntax
```

4. We cannot use special symbols like `!`, `@`, `#`, `$`, `%` etc. in our identifier.

```
a@ = 0
```

Output

```
File "<interactive input>", line 1
    a@ = 0
      ^
SyntaxError: invalid syntax
```

5. An identifier can be of any length.

Things to Remember

- ✓ Python is a case-sensitive language. This means, Variable and variable are not the same.
- ✓ Always give the identifiers a name that makes sense. While `c = 10` is a valid name, writing `count = 10` would make more sense, and it would be easier to figure out what it represents when you look at your code after a long gap.
- ✓ Multiple words can be separated using an underscore, like `this_is_a_long_variable`.

8. PYTHON KEYWORDS

- ✓ Keywords are the reserved words in Python.
- ✓ We cannot use a keyword as a variable name, function name or any other identifier. They are used to define the syntax and structure of the Python language.
- ✓ In Python, keywords are case sensitive.
- ✓ There are 33 keywords in Python 3.7. This number can vary slightly over the course of time.
- ✓ All the keywords except True, False and None are in lowercase and they must be written as they are. The list of all the keywords is given below.

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

Fig. 2.12 Python Keywords

9. LITERALS IN PYTHON

Literals are a notation for representing a fixed value in source code. They can also be defined as raw value or data given in variables or constants.

Example:

```
# Numeric literals
x = 24
y = 24.3
z = 2+3j
print(x, y, z)
```

Output

```
24 24.3 (2+3j)
```

Here 24, 24.3, 2+3j are considered as literals.

Python has different types of literals.

1. String literals

A string literal can be created by writing a text (a group of Characters) surrounded by the single (’), double (“”), or triple quotes. By using triple quotes, we can write multi-line strings or display in the desired way.

Example:

```
s = 'welcome'           # in single quote
t = "welcome"          # in double quotes
m = '''how             # multi-line String
    are
    you'''

print(s)
print(t)
print(m)
```

Output

```
welcome
```

```
welcome
```

```
how
```

```
are
```

```
you
```

2. Character literal

It is also a type of string literals where a single character surrounded by single or double-quotes.

Example:

```
v = 'n'           # character literal in single quote
w = "a"          # character literal in double quotes
print(v)
print(w)
```

Output

```
n
a
```

3. Numeric literals

They are immutable and there are three types of numeric literal:

- A. Integer
- B. Float
- C. Complex

A. Integer:

Both positive and negative numbers including 0. There should not be any fractional part.

Example:

```
a = 0b10100      # Binary Literals
b = 50           # Decimal Literal
c = 0o320        # Octal Literal
d = 0x12b        # Hexadecimal Literal
print(a, b, c, d)
```

Output

```
20 50 208 299
```

In the program above we assigned integer literals (0b10100, 50, 0o320, 0x12b) into different variables. Here, 'a' is binary literal, 'b' is a decimal literal, 'c' is an octal literal and 'd' is a hexadecimal literal. But on using print function to display value or to get output they were converted into decimal.

B. Float

These are real numbers having both integer and fractional parts.

Example:

```
e = 24.8
f = 45.0
print(e, f)
```

Output

```
24.8 45.0
24.8 and 45.0 are floating-point literals because both 24.8
and 45.0 are floating-point numbers.
```

C. Complex Literal

The numerals will be in the form of $a+bj$, where 'a' is the real part and 'b' is the complex part.

Example:

```
z = 7 + 5j
k = 7j          # real part is 0 here.
print(z, k)
```

Output

```
(7+5j) 7j
```

4. Boolean literals

There are only two boolean literals in Python. They are true and false.

Example:

```
a = (1 == True)
b = (1 == False)
c = True + 3
d = False + 7
print("a is", a)
print("b is", b)
print("c:", c)
print("d:", d)
```

Output

```
a is True
b is False
c: 4
d: 7
```


In python, True represents the value as 1 and False represents the value as 0. In the above example 'a' is True and 'b' is False because 1 equal to True.

10. PYTHON STRING

- ✓ String is defined as sequence of characters represented in quotation marks (either single quotes (') or double quotes (").
- ✓ An individual character in a string is accessed using an index.
- ✓ The index should always be an integer (positive or negative).
- ✓ An index starts from 0 to n-1.
- ✓ Strings are immutable i.e.; the contents of the string cannot be changed after it is created.
- ✓ Python will get the input at run time by default as a string.
- ✓ Python does not support character data type.
- ✓ A string of size 1 can be treated as characters.

1. single quotes (')
2. double quotes (")
3. triple quotes(" " " " " " " ")

Operations on string:

1. Indexing
2. Slicing
3. Concatenation
4. Repetitions
5. Member ship

String A	H	E	L	L	O
Positive Index	0	1	2	3	4
Negative Index	-5	-4	-3	-2	-1

Fig. 2.13. String Indexing

1. Indexing

```
>>>a="HELLO"  
>>>print(a[0]) # Positive indexing helps in accessing the string from  
the beginning  
>>>H  
  
>>>print(a[-1]) # Negative subscript helps in accessing the string from  
the end.  
>>>O  
>>>print[0:4] - HELL #The Slice[start : stop] operator extracts
```

2. Slicing

```
print[ :3] - HEL # sub string from the strings.  
print[0: ]- HELLO # A segment of a string is  
called a slice
```

3. Concatenation

```
a="save" # The + operator joins the text on  
both  
b="earth" # sides of the operator.  
>>>print(a+b)  
Save earth
```

4. Repetitions

```
>>>a="panimalar" # The * operator repeats the string  
on the  
>>>print(3*a) #left hand side times the value on right  
hand side.  
panimalarpanimalar  
panimalar
```

5. Member ship

```
>>> s="good morning" # Using membership operators to  
check a
```

```

>>>"m" in s           # particular character is in
string or not.
True                  # Returns true if present
>>> "a" not in s
True

```

String built in functions and methods:

A method is a function that "belongs to" an object.

Syntax to access the method

```
Stringname.method()
```

```
a="happy birthday"
```

here, a is the string name.

syntax

1. a.capitalize()

```

>>> a.capitalize()    #capitalize only the first
letter

```

```
' Happy birthday'    #in a string
```

2. a.upper()

```

>>> a.upper()         #change string to upper case

```

```
'HAPPY BIRTHDAY'
```

3. a.lower()

```

>>> a.lower()         #change string to lower case

```

```
' happy birthday'
```

4. a.title()

```

>>> a.title()         #change string to title case

```

i.e.

```
' Happy Birthday ' #first characters of all the words
are capitalized.
```

5. a.swapcase()

```
>>> a.swapcase()          #change lowercase characters
'HAPPY BIRTHDAY'         #to uppercase and vice versa
```

6. a.split()

```
>>> a.split()             #returns a list of words
['happy', 'birthday']    #separated by space
```

7. a.center(width, "fillchar")

```
>>>a.center(19,"*")      #pads the string with the
'***happy birthday***'  #specified "fillchar"
till the
#length is equal to
"width"
```

8. a.count(substring)

```
>>> a.count('happy')    #returns the number of occurrences
of substring
1
```

9. a.replace(old, new)

```
>>>a.replace('happy', 'wishyou happy') #replace all old
substrings
#with new
substrings
'wishyou happy birthday'
```

10. a.join(b)

```
>>> b="happy"           #returns a string concatenated
>>> a="-"               #with the elements of an
```

```

>>> a.join(b)           #iterable. (Here "a" is the
'h-a-p-p-y'           #iterable)

11. a.isupper()
>>> a.isupper()        #checks whether all the case-
False                 #based characters (letters) of
                    #the string are uppercase.

12. a.islower()
>>> a.islower()        #checks whether all the case-
True                 #based characters (letters) of
                    #the string are lowercase.

13. a.isalpha()
>>> a.isalpha()        #checks whether the string
False                 #consists of alphabetic
                    #characters only.

```

Escape sequences in string

Escape Sequence	Description	example
\n	new line	>>> print ("hai hello") hai hello
\\	prints Backslash (\)	>>> print("hai\\hello") hai\hello
\'	prints Single quote(')	>>> print("'") '
\"	prints Double quote	>>>print ("\"") (") "

```
\t          prints tab space
>>>print("hai\thello")
           hai hello
\a          ASCII Bell (BEL)
>>>print("\a")
```

11. OPERATORS IN PYTHON

Types of Operators

1. Arithmetic operator
2. Relational operator
3. Logical or Boolean operator
4. Assignment Operator
5. Ternary operator
6. Bit wise operator
7. Increment or Decrement operator

Operators are special symbols in Python that carry out arithmetic or logical computation. The value that the operator operates on is called the operand.

Example:

```
>>> 2+3
5
```

Here, + is the operator that performs addition. 2 and 3 are the operands and 5 is the output of the operation.

1. Arithmetic operators

Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication, etc.

Operator	Meaning	Example
+	Add two operands or unary plus	$x + y + 2$
-	Subtract right operand from the left or unary minus	$x - y - 2$
*	Multiply two operands	$x * y$
/	Divide left operand by the right one (always results into float)	x / y
%	Modulus - remainder of the division of left operand by the right	$x \% y$ (remainder of x/y)
//	Floor division - division that results into whole number adjusted to the left in the number line	$x // y$
**	Exponent - left operand raised to the power of right	$x ** y$ (x to the power y)

Example :

```

x = 15
y = 4
print('x + y =', x+y)           # Output: x + y = 19
print('x - y =', x-y)           # Output: x - y = 11
print('x * y =', x*y)           # Output: x * y = 60
print('x / y =', x/y)           # Output: x / y = 3.75
print('x // y =', x//y)         # Output: x // y = 3
print('x ** y =', x**y)         # Output: x ** y = 50625

```

Output

```

x + y = 19
x - y = 11
x * y = 60
x / y = 3.75
x // y = 3
x ** y = 50625

```

2. Relational operators

Relational operators are used to compare values. It returns either True or False according to the condition.

Operator	Meaning	Example
>	Greater than - True if left operand is greater than the right	$x > y$
<	Less than - True if left operand is less than the right	$x < y$
==	Equal to - True if both operands are equal	$x == y$
!=	Not equal to - True if operands are not equal	$x != y$
>=	Greater than or equal to - True if left operand is greater than or equal to the right	$x >= y$
<=	Less than or equal to - True if left operand is less than or equal to the right	$x <= y$

Example:

```
x = 10
y = 12
print('x > y is',x>y) # Output: x > y is False
print('x < y is',x<y) # Output: x < y is True
print('x == y is',x==y) # Output: x == y is False
print('x != y is',x!=y) # Output: x != y is True
print('x >= y is',x>=y) # Output: x >= y is False
print('x <= y is',x<=y) # Output: x <= y is True
```


Output

```
x > y is False
x < y is True
x == y is False
x != y is True
x >= y is False
x <= y is True
```

3. Logical operators

Logical operators are the and, or, not operators.

Operator	Meaning	Example
and	True if both the operands are true	x and y
or	True if either of the operands is true	x or y
not	True if operand is false (complements the operand)	not x

Example:

```
x = True
y = False
print('x and y is',x and y)
print('x or y is',x or y)
print('not x is',not x)
```

Output

```
x and y is False
x or y is True
not x is False
```

4. Assignment operators

Assignment operators are used in Python to assign values to variables.

`a = 5` is a simple assignment operator that assigns the value 5 on the right to the variable `a` on the left.

There are various compound operators in Python like `a += 5` that adds to the variable and later assigns the same. It is equivalent to `a = a + 5`.

Operator	Example	Equivalent to
<code>=</code>	<code>x = 5</code>	<code>x = 5</code>
<code>+=</code>	<code>x += 5</code>	<code>x = x + 5</code>
<code>-=</code>	<code>x -= 5</code>	<code>x = x - 5</code>
<code>*=</code>	<code>x *= 5</code>	<code>x = x * 5</code>
<code>/=</code>	<code>x /= 5</code>	<code>x = x / 5</code>
<code>%=</code>	<code>x %= 5</code>	<code>x = x % 5</code>
<code>//=</code>	<code>x //= 5</code>	<code>x = x // 5</code>
<code>**=</code>	<code>x **= 5</code>	<code>x = x ** 5</code>
<code>&=</code>	<code>x &= 5</code>	<code>x = x & 5</code>
<code> =</code>	<code>x = 5</code>	<code>x = x 5</code>
<code>^=</code>	<code>x ^= 5</code>	<code>x = x ^ 5</code>
<code>>>=</code>	<code>x >>= 5</code>	<code>x = x >> 5</code>
<code><<=</code>	<code>x <<= 5</code>	<code>x = x << 5</code>

5. Bitwise operators

Bitwise operators act on operands as if they were strings of binary digits. They operate bit by bit, hence the name.

For example, 2 is 10 in binary and 7 is 111.

In the table below: Let `x = 10` (0000 1010 in binary) and `y = 4` (0000 0100 in binary)

Operator	Meaning	Example
&	Bitwise AND	$x \& y = 0$ (0000 0000)
	Bitwise OR	$x y = 14$ (0000 1110)
~	Bitwise NOT	$\sim x = -11$ (1111 0101)
^	Bitwise XOR	$x \wedge y = 14$ (0000 1110)
>>	Bitwise right shift	$x \gg 2 = 2$ (0000 0010)
<<	Bitwise left shift	$x \ll 2 = 40$ (0010 1000)

6. Ternary Operator

Many programming languages support ternary operator, which basically define a conditional expression.

Similarly the ternary operator in python is used to return a value based on the result of a binary condition. It takes binary value(condition) as an input, so it looks similar to an “if-else” condition block. However, it also returns a value so behaving similar to a function.

Syntax

[on_true] if [expression] else [on_false]

Let’s write one simple program, which compare two integers -

a. Using python if-else statement -

```
>>> x, y = 5, 6
>>> if x>y:
    print("x")
else:
    print("y")
y
```

b. Using ternary operator

```
>>> x, y = 5, 6
>>> print("x" if x> y else "y")
y
```

With ternary operator, we are able to write code in one line. So python basically first evaluates the condition, if true – evaluate the first expression else evaluates the second condition.

7. Increment and Decrement Operators

If you're familiar with Python, you would have known Increment and Decrement operators (both pre and post) are not allowed in it. Python is designed to be consistent and readable. One common error by a novice programmer in languages with ++ and -- operators is mixing up the differences (both in precedence and in return value) between pre and post increment/decrement operators. Simple increment and decrement operators aren't needed as much as in other languages.

You don't write things like :

```
for (int i = 0; i < 5; ++i)
```

In Python, instead we write it like below and syntax is as follow:

```
for variable_name in range(start, stop, step)
```

start: Optional. An integer number specifying at which position to start. Default is 0

stop: An integer number specifying at which position to end.

step: Optional. An integer number specifying the incrementation. Default is 1

```
# A Sample Python program to show loop (unlike many
# other languages, it doesn't use ++)
# this is for increment operator here start = 1,
# stop = 5 and step = 1 (by default)
print("INCREMENTED FOR LOOP")
for i in range(0, 5):
    print(i)

# this is for increment operator here start = 5,
# stop = -1 and step = -1
print("\n DECREMENTED FOR LOOP")
for i in range(4, -1, -1):
    print(i)
```

Output-1: INCREMENTED FOR LOOP

0
1
2
3
4

Output-2: DECREMENTED FOR LOOP

4
3
2
1
0

QUESTION BANK

1. List various types of operators in Python and write any 4 types of operators.
2. Outline with an example the assignment and bitwise operators supported in Python
3. Write a Python program to print prime number series up to N.
4. Write a Python program to Swapping of two numbers with and without using temporary variable.
5. If the age of Ram, Sam, and Khan are input through the keyboard, write a python program to determine the eldest and youngest of the three.
6. Arithmetic operations (Addition, Subtraction, Multiplication, and Division) on integers. Input the two integer values and operator for performing arithmetic operation through keyboard.
7. Write a python program to find the given number is odd or even.
8. Define Identifiers in Python
9. What is the comment statement in python?
10. Define variables in python.
11. Explain Interpreter & Interactive mode in Python
12. Compose the importance of indentation in python.
13. Give the reserved words in Python.
14. Define the scope and lifetime of a variable in python.
15. How the area of circle is calculated explain with an example.

TEXT /REFERENCE BOOKS:

1. Allen Downey, Jeffrey Elkner, Chris Meyers.How to think like a computer scientist learning with Python / 1st Edition,2012
2. Kenneth A. Lambert, The Fundamentals of Python: First Programs, 2011, Cengage Learning, ISBN: 978- 1111822705

USEFUL WEBSITES:

1. <http://docs.python.org/3/tutorial/index.html>
2. <http://www.ibiblio.org/g2swap/byteofpython/read/>



SATHYABAMA

**INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)**

Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE

www.sathyabama.ac.in

SCHOOL OF COMPUTING

**DEPARTMENT OF COMPUTER SCIENCE AND
ENGINEERING**

PYTHON PROGRAMMING – SBSA1202

Unit III

Creating Python Programs: Input and Output Statements, Control statements(Branching, Looping, Conditional Statement, Exit function, Difference between break, continue and pass.), Defining Functions, default arguments, Errors and Exceptions.

1. Input and Output Statements

Here we focus on two built-in functions `print()` and `input()` to perform I/O task in Python. Python provides numerous built-in functions that are readily available to us at the Python prompt. Some of the functions like `input()` and `print()` are widely used for standard input and output operations respectively.

1.1. Python Output Using `print()` function

We use the `print()` function to output data to the standard output device (screen). We can also output data to a file.

Example:

```
print('This sentence is output to the screen')
```

Output

```
This sentence is output to the screen
```

Example:

```
a = 5
print('The value of a is', a)
```

Output

```
The value of a is 5
```

In the second `print()` statement, we can notice that space was added between the string and the value of variable `a`. This is by default, but we can change it.

Syntax of the `print()` function is:

```
print(*objects, sep=' ', end='\n', file=sys.stdout,
flush=False)
```

Here, `objects` is the value(s) to be printed. The `sep` separator is used between the values. It defaults into a space character. After all values are printed, `end` is printed. It defaults into a new line. The `file` is the object where the values are printed and its default value is `sys.stdout` (screen).

Example :

```
print(1, 2, 3, 4)
print(1, 2, 3, 4, sep='*')
```



```
print(1, 2, 3, 4, sep='#', end='&')
```

Output

```
1 2 3 4
1*2*3*4
1#2#3#4&
```

Output formatting

Sometimes we would like to format our output to make it look attractive. This can be done by using the `str.format()` method. This method is visible to any string object.

```
>>> x = 5; y = 10
>>> print('The value of x is {} and y is {}'.format(x,y))
The value of x is 5 and y is 10
```

Here, the curly braces `{}` are used as placeholders. We can specify the order in which they are printed by using numbers (tuple index).

```
print('I love {0} and {1}'.format('bread', 'butter'))
print('I love {1} and {0}'.format('bread', 'butter'))
```

Output

```
I love bread and butter
I love butter and bread
```

We can even use keyword arguments to format the string.

```
>>>print('Hello{name},{greeting}'.format(greeting='Goodmo
rning',      name = 'John'))
Hello John, Goodmorning
```

We can also format strings like the old `sprintf()` style used in C programming language. We use the `%` operator to accomplish this.

```
>>> x = 12.3456789
>>> print('The value of x is %3.2f' %x)
The value of x is 12.35
>>> print('The value of x is %3.4f' %x)
The value of x is 12.3457
```

1.2.Python Input

Up until now, our programs were static. The value of variables was defined or hard coded into the source code. To allow flexibility, we might want to take the input from the user. In Python, we have the `input()` function to allow this.

The syntax for `input()` is:

```
input([prompt])
```

where prompt is the string we wish to display on the screen. It is optional.

```
>>> num = input('Enter a number: ')
Enter a number: 10
>>> num
'10'
```

Here, we can see that the entered value 10 is a string, not a number. To convert this into a number we can use int() or float() functions.

```
>>> int('10')
10
>>> float('10')
10.0
```

This same operation can be performed using the eval() function. But eval takes it further. It can evaluate even expressions, provided the input is a string

```
>>> int('2+3')
Traceback (most recent call last):
  File "<string>", line 301, in runcode
  File "<interactive input>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '2+3'
>>> eval('2+3')
5
```

Example 1: Taking input from the user with a message.

```
name = input("Enter your name: ")
```

Output

```
print("Hello, " + name)
```

Output:

```
Enter your name: Arun
Hello, Arun
```

Example 2: By default input() function takes the user's input in a string. So, to take the input in the form of int, you need to use int() along with input function.

```
num = int(input("Enter a number: "))
add = num + 1
```

Output

```
print(add)
```

Output

```
Enter a number: 25
```

```
26
```

Example: Python program to demonstrate print() method

```
print("HELLO")  
  
# code for disabling the softspace feature  
print('H', 'E', 'L', 'L', 'O', sep = ' ')  
  
# using end argument  
print("Python", end = '@')  
print("HELLO")
```

Output:

```
HELLO
```

```
HELLO
```

```
Python@HELLO
```

2. Control statements

When there is no condition placed before any set of statements, the program will be executed in sequential manure. But when some condition is placed before a block of statements the flow of execution might change depends on the result evaluated by the condition. This type of statement is also called decision making statements or control statements. This type of statement may skip some set of statements based on the condition.

Logical Conditions Supported by Python

- ✓ Equal to (==) Eg : a == b
- ✓ Not Equal (!=)Eg : a != b
- ✓ Greater than (>) Eg : a > b
- ✓ Greater than or equal to (>=) Eg : a >= b
- ✓ Less than (<) Eg : a < b
- ✓ Less than or equal to (<=) Eg : a <= b

Indentation

To represent a block of statements other programming languages like C, C++ uses “{ ...}” curly – brackets , instead of this curly braces python uses indentation using white space which defines scope in the code. The example given below shows the difference between usage of Curly bracket and white space to represent a block of statement.

Table 1.6 : C- Program Vs Python

C Program	Python
<pre>x = 500 y = 200 if (x > y) { printf("x is greater than y") } else if(x == y) { printf("x and y are equal") } else { printf("x is less than y") }</pre>	<pre>x = 500 y = 200 if x > y: print("x is greater than y") elif x == y: print("x and y are equal") else: print("x is less than y")</pre> <p>Indentation (At least one WhiteSpace instead of curly bracket)</p>

Without proper Indentation:

```
x = 500
y = 200
if x > y:
    print("x is greater than y")
```

In the above example there is no proper indentation after if statement which will lead to Indentation error.

If statement:

The “if” statement is written using “if” keyword, followed by a condition. If the condition is true the block will be executed. Otherwise, the control will be transferred to the first statement after the block.

Syntax:

```
if <Boolean>:
    <block>
```

In this statement, the order of execution is purely based on the evaluation of boolean expression.

Example:

```
x = 200
y = 100
if x > y:
    print("X is greater than Y")
print("End")
```

Output :

```
X is greater than Y
End
```

In the above the value of x is greater than y, hence it executed the print statement whereas in the below example x is not greater than y hence it is not executed the first print statement.

```
x = 100
y = 200
if x > y:
    print("X is greater than Y")
print("End")
```

Output:

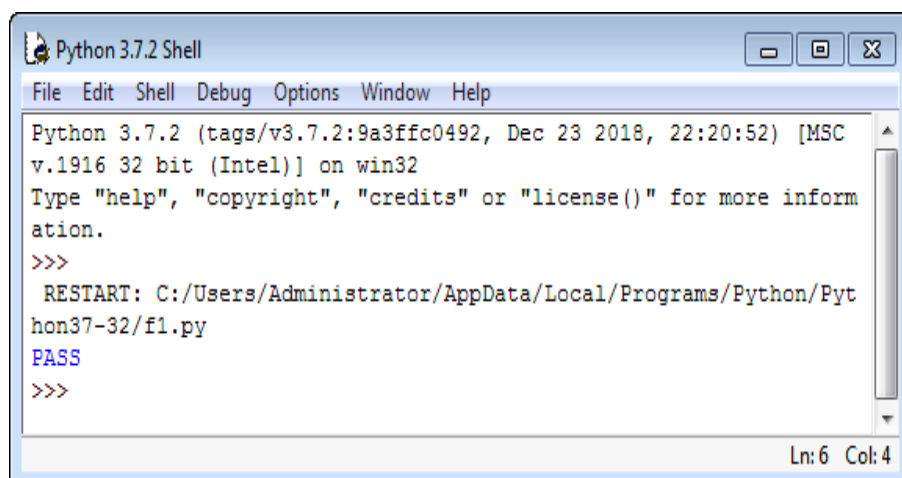
```
End
```

Elif

The elif keyword is useful for checking another condition when one condition is false.

Example:

```
mark = 55
if (mark >=75):
    print("FIRST CLASS")
elif mark >= 50:
    print("PASS")
```

Output:

```
Python 3.7.2 Shell
File Edit Shell Debug Options Window Help
Python 3.7.2 (tags/v3.7.2:9a3ffc0492, Dec 23 2018, 22:20:52) [MSC
v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more inform
ation.
>>>
RESTART: C:/Users/Administrator/AppData/Local/Programs/Python/Pyt
hon37-32/f1.py
PASS
>>>
```

In the above the example, the first condition (mark >=75) is false then the control is transferred

to the next condition (mark >=50), Thus, the keyword elif will be helpful for having more than one condition.

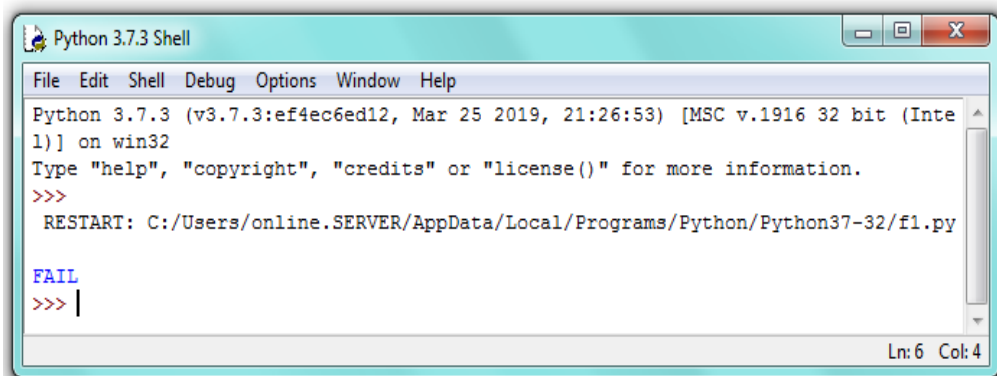
Else

The else keyword will be used as a default condition. i.e. When there are many conditions, when the if-condition is not true and all elif-conditions are also not true, then else part will be executed.

Example:

```
mark = 10
if mark >= 75:
    print("FIRST CLASS")
elif mark >= 50:
    print("PASS")
else:
    print("FAIL")
```

Output:



```
Python 3.7.3 Shell
File Edit Shell Debug Options Window Help
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
RESTART: C:/Users/online.SERVER/AppData/Local/Programs/Python/Python37-32/f1.py
FAIL
>>> |
```

In the example above, condition 1 and condition 2 fail. None of the preceding condition is true. Hence, the **else** part is executed.

ITERATIVE STATEMENTS

Sometimes certain section of the code (block) may need to be repeated again and again as long as certain condition remains true. In order to achieve this, the iterative statements are used. The number of times the block needs to be repeated is controlled by the test condition used in that statement. This type of statement is also called as the “Looping Statement”. Looping statements add a surprising amount of new power to the program.

Need for Looping / Iterative Statement

Suppose the programmer wishes to display the string “Sathyabama !...” 150 times. For this,

one can use the print command 150 times.

```
print("Sathyabama!...")  
print("Sathyabama!...")  
... .  
... .
```

The above method is somewhat difficult and laborious. The same result can be achieved by a loop using just two lines of code. (As below)

```
for count in range(1,150) :  
    print ("Sathyabama")
```

Types of looping statements

- for loop
- while loop

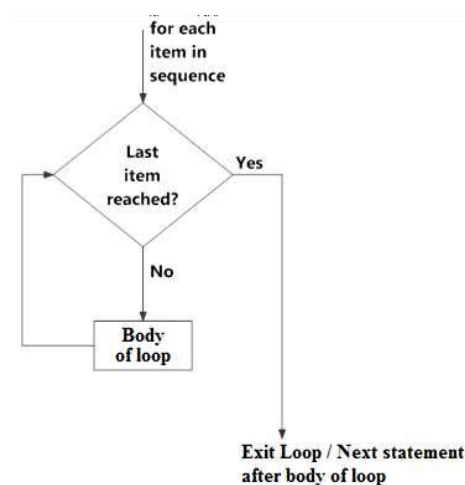
The 'for' Loop

The for loop is one of the powerful and efficient statements in python which is used very often. It specifies how many times the body of the loops needs to be executed. For this reason, it uses control variables which keep tracks, the count of execution.

The general syntax of a "for" loop looks as below:

```
for <variable> in range (A,B) :  
  
<body of the loop >
```

Flow Chart:



Example 1: To compute the sum of first n numbers (i.e. 1 + 2 + 3 + + n)

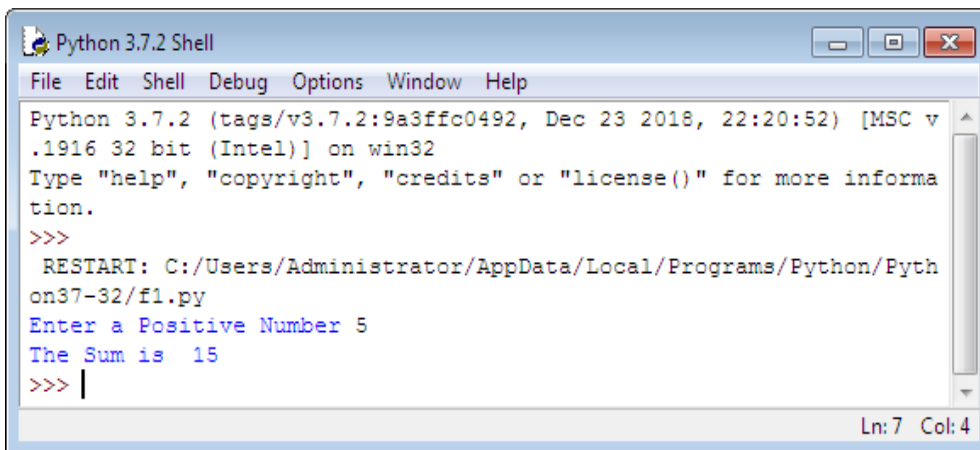
```

# Sum.py
total = 0
n = int (input ("Enter a Positive Number"))
for i in range(1,n+1):
    total = total + i
    print ("The Sum is ", total)

```

Note: Why (n+1)? Check in table given below.

Output:



In the above program, the statement `total = total + i` is repeated again and again „n“ times. The number of execution count is controlled by the variable „i“. The range value is specified earlier before it starts executing the body of loop. The initial value for the variable `i` is 1 and final value depends on „n“. You may also specify any constant value.

The range() Function:

The `range()` function can be called in three different ways based on the number of parameters. All parameter values must be integers.

Table 1.7 : Categories of range function

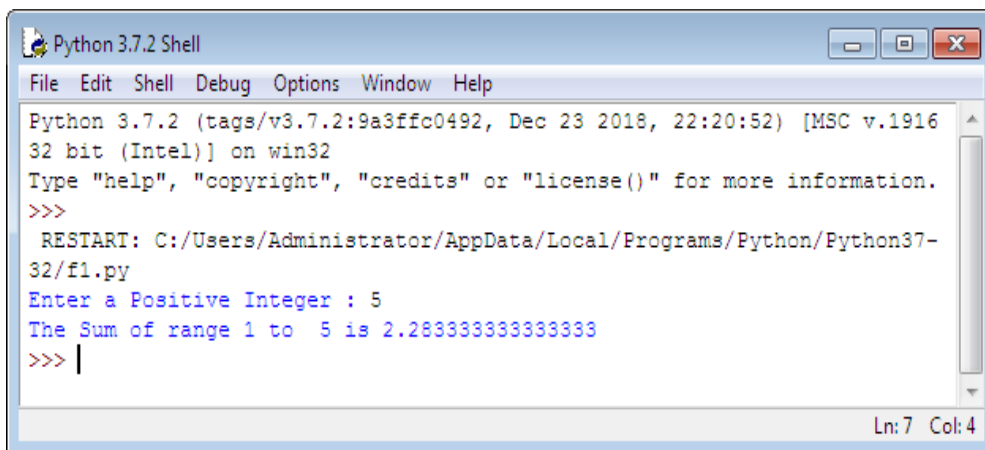
Type	Example	Explanation
<code>range(end)</code>	for i in range(5): print(i) Output : 0,1,2,3,4	This is begins at 0. Increments by 1. End just before the value of end parameter.
<code>range(begin, end)</code>	for i in range(2,5): print(i) Output : 2,3,4	Starts at begin, End before end value, Increment by 1

<code>range(begin,end,step)</code>	<pre>for i in range(2,7,2) print(i) Output : 2,4,6</pre>	Starts at begin, End before end value, increment by step value
------------------------------------	---	--

Example : To compute Harmonic Sum (ie: $1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{n}$)

```
# harmonic.py
total = 0
n= int(input("Enter a Positive Integer:"))
for i in range(1,n+1):
    total+= 1/i
    print("The Sum of range 1 to ",n, "is", total)
```

Output:



```
Python 3.7.2 Shell
File Edit Shell Debug Options Window Help
Python 3.7.2 (tags/v3.7.2:9a3ffc0492, Dec 23 2018, 22:20:52) [MSC v.1916
32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
RESTART: C:/Users/Administrator/AppData/Local/Programs/Python/Python37-
32/f1.py
Enter a Positive Integer : 5
The Sum of range 1 to 5 is 2.2833333333333333
>>> |
Ln:7 Col:4
```

Example: Factorial of a number “n”

```
n= int(input("Enter a Number :"))
factorial = 1 # Initialize factorial value by 1
# To verify whether the given number is negative /
positive / zero if n < 0:
    print("Negative Number , Enter valid Number !...")
elif n == 0:
    print("The factorial of 0 is 1") else:
for i in range(1, n + 1):
    factorial = factorial*i
    print("The factorial of" ,n, "is", factorial)
```

Output:

```
Python 3.7.2 Shell
File Edit Shell Debug Options Window Help
Python 3.7.2 (tags/v3.7.2:9a3ffc0492, Dec 23 2018, 22:20:52) [MSC v.1916 32 bit
(Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
RESTART: C:/Users/Administrator/AppData/Local/Programs/Python/Python37-32/f1.py

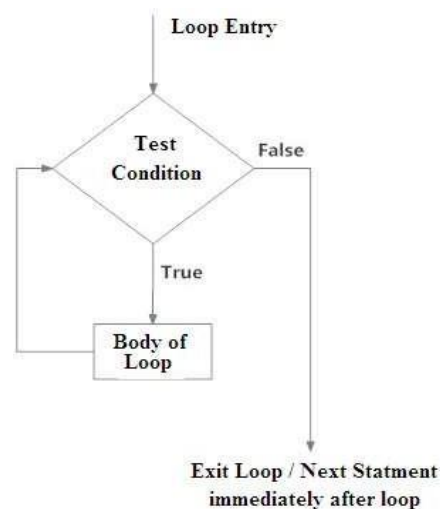
Enter a Number : -1
Negetive Number , Enter valid Number !...
>>>
RESTART: C:/Users/Administrator/AppData/Local/Programs/Python/Python37-32/f1.py

Enter a Number :10
The factorial of 10 is 3628800
>>> |
```

The while Loop

The while loop allows the program to repeat the body of a loop, any number of times, when some condition is true. The drawback of while loop is that, if the condition is not proper it may lead to infinite looping. So, the user has to carefully choose the condition in such a way that it will terminate at a particular stage.

Flow Chart:



Syntax:

```
while (condition):
    <body of the loop>
```

In this type of loop, the execution of the loop body is purely based on the output of the given condition. As long as the condition is TRUE or in other words until the condition becomes FALSE the program will repeat the body of loop.

<p>Valid Example :</p> <pre>i = 10 while i<15 : print(i) i = i + 1</pre> <p>Output : 10,11,12,13,14</p>	<p>Invalid Example:</p> <pre>i = 10 while i<15 : print(i)</pre> <p>Output : 10,10,10,10..... Indeterminate number of times</p>
--	---

Example: Program to display Fibonacci Sequence

```
n = int(input("Enter number of terms in the sequence you
want to display"))
n1 = 0
n2 = 1
count = 0
if n <= 0:
    print ("Enter a positive integer")
elif n == 1:
    print("Fibonacci sequence up to ", n,":")
    print(n2)
else:
    print("Fibonacci sequence of ",n, " terms :")
while count < n:
    print(n1,end=' ,
    ')
    nth = n1 + n2
    n1 = n2
    n2 = nth
    count = count + 1
```

Python break, continue and pass Statements

You might face a situation in which you need to exit a loop completely when an external condition is triggered or there may also be a situation when you want to skip a part of the loop and start next execution. Python provides break and continue statements to handle such situations and to have good control on your loop.

The break Statement:

The break statement in Python terminates the current loop and resumes execution at the next statement, just like the traditional break found in C. The most common use for break is when some external condition is triggered requiring a hasty exit from a loop. The break statement can be used in both while and for loops.

Example:

```
#!/usr/bin/python
for letter in 'Python':      # First Example
    if letter == 'h':
        break
    print 'Current Letter :', letter

var = 10                      # Second Example
while var > 0:
    print 'Current variable value :', var
    var = var -1
    if var == 5:
        break
print "Good bye!"
```

This will produce the following result:

```
Current Letter : P
Current Letter : y
Current Letter : t
Current variable value : 10
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Good bye!
```

The continue Statement:

The continue statement in Python returns the control to the beginning of the while loop. The continue statement rejects all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop. The continue statement can be used in both while and for loops.

Example:

```
#!/usr/bin/python
for letter in 'Python':      # First Example
    if letter == 'h':
```

```

        continue
    print 'Current Letter :', letter

var = 10                                # Second Example
while var > 0:
    var = var -1
    if var == 5:
        continue
    print 'Current variable value :', var
print "Good bye!"

```

This will produce following result:

```

Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : o
Current Letter : n
Current variable value : 10
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Current variable value : 4
Current variable value : 3
Current variable value : 2
Current variable value : 1
Good bye!

```

The pass Statement:

The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute. The pass statement is a null operation; nothing happens when it executes. The pass is also useful in places where your code will eventually go, but has not been written yet (e.g., in stubs for example):

Example:

```

#!/usr/bin/python
for letter in 'Python':

```

```
    if letter == 'h':  
        pass  
        print 'This is pass block'  
    print 'Current Letter :', letter  
print "Good bye!"
```

This will produce following result:

```
Current Letter : P  
Current Letter : y  
Current Letter : t  
This is pass block  
Current Letter : h  
Current Letter : o  
Current Letter : n  
Good bye!
```

The preceding code does not execute any statement or code if the value of letter is 'h'. The pass statement is helpful when you have created a code block but it is no longer required. You can then remove the statements inside the block but let the block remain with a pass statement so that it doesn't interfere with other parts of the code.

exit() function

Apart from the above mentioned techniques, we can use the in-built exit() function to quit and come out of the execution loop of the program in Python.

Syntax:

```
exit()
```

Example:

```
for x in range(1,10):  
    print(x*10)  
    exit()
```

The exit() function can be considered as an alternative to the quit() function, which enables us to terminate the execution of the program.

3. Python Functions

A function is a block of code which only runs when it is called. You can pass data, known as parameters, into a function. A function can return data as a result.

Creating a Function

In Python a function is defined using the def keyword:

Example:

```
def my_function():  
    print("Hello from a function")
```

Calling a Function

To call a function, use the function name followed by parenthesis:

Example

```
def my_function():  
    print("Hello from a function")  
  
my_function()
```

Arguments

Information can be passed into functions as arguments. Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma. The following example has a function with one argument (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

Example

```
def my_function(fname):  
    print(fname + " Refsnes")  
  
my_function("Emil")  
my_function("Tobias")  
my_function("Linus")
```

Number of Arguments

By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less.

Example:

This function expects 2 arguments, and gets 2 arguments:

```
def my_function(fname, lname):  
    print(fname + " " + lname)  
  
my_function("Emil", "Refsnes")
```

4. Errors and Exceptions in Python

Errors are problems in the program that the program should not recover from. If at any point in the program an error occurs, then the program should exit gracefully. On the other hand, Exceptions are raised when an external event occurs which in some way changes the normal flow of the program.

Handling Exceptions with Try/Except/Finally

Errors and Exceptions in Python are handled with the Try: Except: Finally construct. You put the unsafe code in the try: block. You put the fall-back code in the Except: block. The final code is kept in the Finally: block.

For example, look at the code below.

```
>>> try:  
...     print("in the try block")  
...     print(1/0)  
... except:  
...     print("In the except block")  
... finally:  
...     print("In the finally block")  
...  
in the try block  
In the except block  
In the finally block
```

Raising exceptions for a predefined condition

Exceptions can also be raised if you want the code to behave within specific parameters. For example, if you want to limit the user-input to only positive integers, raise an exception.

```
# exc.py  
while True:  
    try:
```



```
        user = int(input())
        if user < 0:
            raise ValueError("please give positive number")
        else:
            print("user input: %s" % user)
except ValueError as e:
    print(e)
```

So the output of the above program is:

```
→ python exc.py
4
user input: 4
3
user input: 3
2
user input: 2
1
user input: 1
-1
please give positive number
5
user input: 5
2
user input: 2
-5
please give positive number
^C
Traceback (most recent call last):
  File "exc.py", line 3, in <module>
    user = int(input())
KeyboardInterrupt
```

QUESTION BANK

1. What is meant by conditional If?
2. What is chained conditional statement?
3. Write the syntax and usage of for loop
4. Write the syntax and usage of while loop
5. What is python break statement?
6. What is python continue statement?
7. What is python pass statement?
8. Explain the function arguments in python
9. Briefly explain about function prototypes
10. Briefly explain about Errors and Exceptions.
11. Briefly explain about Input Statement
12. Briefly explain about Output Statements
13. What is python Branching Statement
14. What is python Looping Statement
15. What is python Conditional Statement

TEXT /REFERENCE BOOKS:

1. Allen Downey, Jeffrey Elkner, Chris Meyers.How to think like a computer scientist learning with Python / 1st Edition,2012
2. Kenneth A. Lambert, The Fundamentals of Python: First Programs, 2011, Cengage Learning, ISBN: 978- 1111822705

USEFUL WEBSITES:

1. <http://docs.python.org/3/tutorial/index.html>
2. <http://www.ibiblio.org/g2swap/byteofpython/read/>



SATHYABAMA

INSTITUTE OF SCIENCE AND TECHNOLOGY

(DEEMED TO BE UNIVERSITY)

Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE

www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

PYTHON PROGRAMMING – SBSA1202

Unit IV

Iteration and Recursion: Conditional execution, Alternative execution, Nested conditionals, The return statement, Recursion, Stack diagrams for recursive functions, Multiple assignment, The while statement, Tables, Two-dimensional tables.

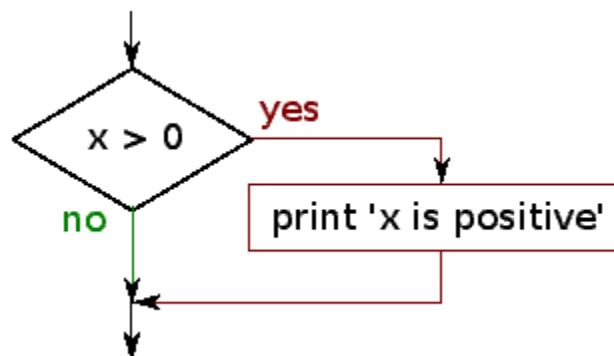
1. Iteration and Recursion

1.1. Conditional execution

In order to write useful programs, we almost always need the ability to check conditions and change the behavior of the program accordingly. **Conditional statements** give us this ability. The simplest form is the if statement:

```
if x > 0 :  
    print 'x is positive'
```

The boolean expression after the if statement is called the **condition**. We end the if statement with a colon character (:) and the line(s) after the if statement are indented.



If the logical condition is true, then the indented statement gets executed. If the logical condition is false, the indented statement is skipped. If statements have the same structure as function definitions or for loops. The statement consists of a header line that ends with the colon character (:) followed by an indented block. Statements like this are called **compound statements** because they stretch across more than one line. There is no limit on the number of statements that can appear in the body, but there has to be at least one. Occasionally, it is useful to have a body with no statements (usually as a place keeper for code you haven't written yet). In that case, you can use the pass statement, which does nothing.

```
if x < 0 :  
    pass          # need to handle negative values!
```

If you enter an if statement in the Python interpreter, the prompt will change from three chevrons to three dots to indicate you are in the middle of a block of statements as shown below:

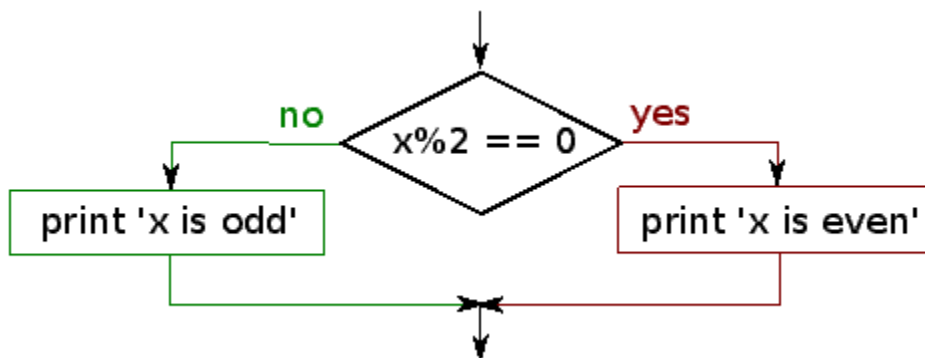
```
>>> x = 3
>>> if x < 10:
...     print 'Small'
...
Small
>>>
```

1.2. Alternative execution

A second form of the if statement is **alternative execution**, in which there are two possibilities and the condition determines which one gets executed. The syntax looks like this:

```
if x%2 == 0 :
    print 'x is even'
else :
    print 'x is odd'
```

If the remainder when x is divided by 2 is 0, then we know that x is even, and the program displays a message to that effect. If the condition is false, the second set of statements is executed.



Since the condition must be true or false, exactly one of the alternatives will be executed. The alternatives are called **branches**, because they are branches in the flow of execution.

1.3. Nested conditionals

One conditional can also be nested within another. We could have written the trichotomy example like this:

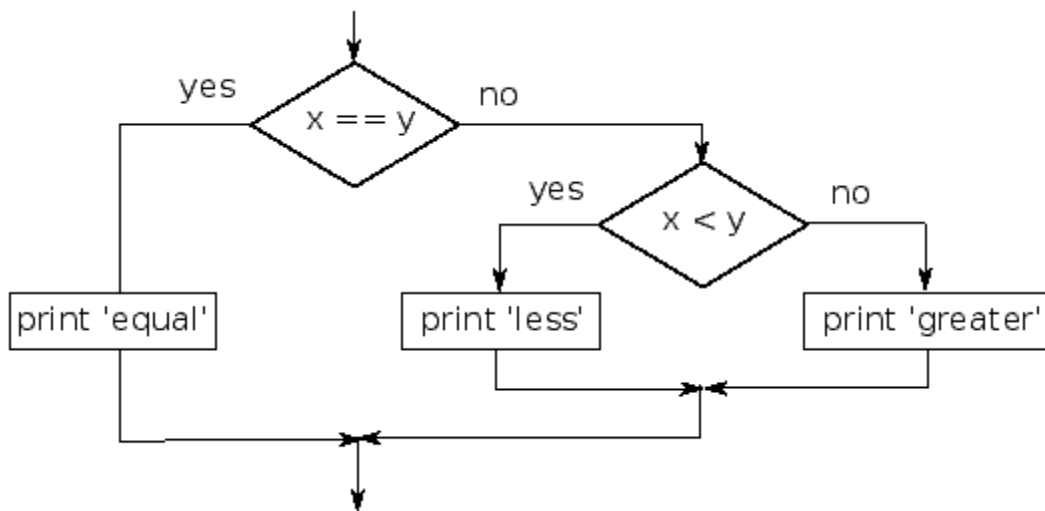
```
if x == y:
    print 'x and y are equal'
```

```

else:
    if x < y:
        print 'x is less than y'
    else:
        print 'x is greater than y'

```

The outer conditional contains two branches. The first branch contains a simple statement. The second branch contains another if statement, which has two branches of its own. Those two branches are both simple statements, although they could have been conditional statements as well.



Although the indentation of the statements makes the structure apparent, **nested conditionals** become difficult to read very quickly. In general, it is a good idea to avoid them when you can. Logical operators often provide a way to simplify nested conditional statements. For example, we can rewrite the following code using a single conditional:

```

if 0 < x:
    if x < 10:
        print 'x is a positive single-digit number.'

```

The print statement is executed only if we make it past both conditionals, so we can get the same effect with the and operator:

```

if 0 < x and x < 10:
    print 'x is a positive single-digit number.'

```

2. The return Statement in Python

The statement `return [expression]` exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as `return None`.

Example

All the above examples are not returning any value. You can return a value from a function as follows –

```
#!/usr/bin/python
Function definition is here
def sum( arg1, arg2 ):
    # Add both the parameters and return them."
    total = arg1 + arg2
    print "Inside the function : ", total
    return total;

# Now you can call sum function
total = sum( 10, 20 );
print "Outside the function : ", total
```

Output

When the above code is executed, it produces the following result –

```
Inside the function : 30
Outside the function : 30
```

3. Recursion

It is legal for one function to call another; it is also legal for a function to call itself. It may not be obvious why that is a good thing, but it turns out to be one of the most magical things a program can do. For example, look at the following function:

```
def countdown(n):
    if n <= 0:
        print 'Blastoff!'
    else:
        print n
        countdown(n-1)
```

If `n` is 0 or negative, it outputs the word, “Blastoff!” Otherwise, it outputs `n` and then calls a function named `countdown`—itself—passing `n-1` as an argument.

What happens if we call this function like this?

```
>>> countdown(3)
```

- ✓ The execution of `countdown` begins with `n=3`, and since `n` is greater than 0, it outputs the value 3, and then calls itself...
- ✓ The execution of `countdown` begins with `n=2`, and since `n` is greater than 0, it outputs the value 2, and then calls itself...
- ✓ The execution of `countdown` begins with `n=1`, and since `n` is greater than 0, it outputs the value 1, and then calls itself...
- ✓ The execution of `countdown` begins with `n=0`, and since `n` is not greater than 0, it outputs the word, "Blastoff!" and then returns.
- ✓ The countdown that got `n=1` returns.
- ✓ The countdown that got `n=2` returns.
- ✓ The countdown that got `n=3` returns.

And then you're back in `__main__`. So, the total output looks like this:

```
3
2
1
Blastoff!
```

A function that calls itself is **recursive**; the process is called **recursion**.

As another example, we can write a function that prints a string `n` times.

```
def print_n(s, n):
    if n <= 0:
        return
    print s
    print_n(s, n-1)
```

If `n <= 0` the `return` statement exits the function. The flow of execution immediately returns to the caller, and the remaining lines of the function are not executed. The rest of the function is similar to `countdown`: if `n` is greater than 0, it displays `s` and then calls itself to display `s` $n - 1$ additional times. So the number of lines of output is $1 + (n-1)$, which adds up to `n`.

4. Stack diagrams for recursive functions

In Stack diagrams, we used a stack diagram to represent the state of a program during a function call. The same kind of diagram can help interpret a recursive function. Every time a function gets called, Python creates a new function frame, which contains the function's local variables and parameters. For a recursive function, there might be more than one frame on the stack at the same time.

Figure 4.1 shows a stack diagram for `countdown` called with `n=3`. As usual, the top of the stack is the frame for `__main__`. It is empty because we did not create any variables in `__main__` or pass any arguments to it. The four `countdown` frames have different values for the parameter `n`. The bottom of the stack, where `n=0`, is called the **base case**. It does not make a recursive call, so there are no more frames. *Draw a stack diagram for `print_n` called with `s = 'Hello'` and `n=2`. Write a function called `do_n` that takes a function object and a number, `n`, as arguments, and that calls the given function `n` times.*

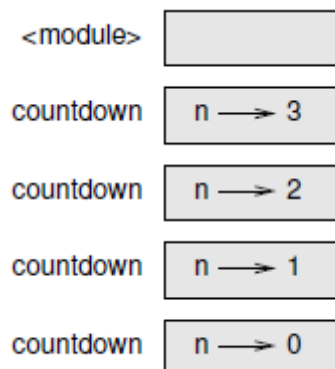


Figure 4.1 Stack diagram

5. Multiple Assignment

Python allows you to assign a single value to several variables simultaneously. For example: `a = b = c = 1`. Here, an integer object is created with the value 1, and all three variables are assigned to the same memory location. You can also assign multiple objects to multiple variables.

For example:

```
a, b, c = 1, 2, "john"
```

Here, two integer objects with values 1 and 2 are assigned to variables `a` and `b` respectively, and one string object with the value "john" is assigned to the variable `c`.

6. The while statement

The while statement allows you to repeatedly execute a block of statements as long as a condition is true. A while statement is an example of what is called a *looping* statement. A while statement can have an optional else clause.

Example

```
#!/usr/bin/python
# Filename: while.py
number = 23
running = True
while running:
    guess = int(raw_input('Enter an integer : '))
    if guess == number:
        print 'Congratulations, you guessed it.'
        running = False # this causes the while loop
to stop
    elif guess < number:
        print 'No, it is a little higher than that.'
    else:
        print 'No, it is a little lower than that.'
else:
    print 'The while loop is over.'
    # Do anything else you want to do here
print 'Done'
```

Output

```
$ python while.py
Enter an integer : 50
No, it is a little lower than that.
Enter an integer : 22
No, it is a little higher than that.
Enter an integer : 23
Congratulations, you guessed it.
The while loop is over.
Done
```

How It Works

In this program, we are still playing the guessing game, but the advantage is that the user is allowed to keep guessing until he guesses correctly - there is no need to repeatedly execute the program for each guess as we have done previously. This aptly demonstrates the use of the while statement. We move the `raw_input` and `if` statements to inside the while loop and set the variable `running` to `True` before the while loop. First, we check if the variable `running` is `True` and then proceed to execute the corresponding *while-block*. After this block is executed, the condition is again checked which in this case is the `running` variable. If it is true, we execute the while-block again, else we continue to execute the optional *else-block* and then continue to the next statement.

The *else* block is executed when the while loop condition becomes `False` - this may even be the first time that the condition is checked. If there is an *else* clause for a while loop, it is always executed unless you have a while loop which loops forever without ever breaking out!

The `True` and `False` are called Boolean types and you can consider them to be equivalent to the value 1 and 0 respectively. It's important to use these where the condition or checking is important and not the actual value such as 1.

The *else-block* is actually redundant since you can put those statements in the same block (as the while statement) after the while statement to get the same effect.

7. Create Tables in Python

Being able to quickly organize our data into a more readable format, such as when data wrangling, can be extremely helpful in order to analyze the data and plan the next steps. Python offers the ability to easily turn certain tabular data types into nicely formatted plain-text tables, and that's with the *tabulate* function.

install tabulate

We first install the **tabulate** library using `pip install` in the command line:

```
pip install tabulate
```

import tabulate function

We then import the *tabulate* function from the **tabulate** library in our code:

```
from tabulate import tabulate
```

And now we are ready to use the *tabulate* function!

tabular data types supported by tabulate

*The **tabulate** function can transform any of the following into an easy to read plain-text table: (from the [tabulate documentation](#))*

- *list of lists or another iterable of iterables*
- *list or another iterable of dicts (keys as columns)*
- *dict of iterables (keys as columns)*
- *two-dimensional NumPy array*
- *NumPy record arrays (names as columns)*
- *pandas.DataFrame*

list of lists

For example, if we have the following list of lists:

```
table = [['First Name', 'Last Name', 'Age'], ['John', 'Smith', 39], ['Mary', 'Jane', 25], ['Jennifer', 'Doe', 28]]
```

We can turn it into a much more readable plain-text table using the *tabulate* function:

```
print(tabulate(table))
```

```
print(tabulate(table))
```

First Name	Last Name	Age
John	Smith	39
Mary	Jane	25
Jennifer	Doe	28

Since the first list in the list of lists contains the names of columns as its elements, we can set it as the column or header names by passing *'firstrow'* as the argument for the *headers* parameter:

```
print(tabulate(table, headers='firstrow'))
```

```
print(tabulate(table, headers='firstrow'))
```

First Name	Last Name	Age
John	Smith	39
Mary	Jane	25
Jennifer	Doe	28

The *tabulate* function also contains a *tablefmt* parameter, which allows us to improve the appearance of our table using pseudo-graphics:

```
print(tabulate(table, headers='firstrow', tablefmt='grid'))
```

```
print(tabulate(table, headers='firstrow', tablefmt='grid'))
```

```
+-----+-----+-----+
| First Name | Last Name | Age |
+=====+=====+=====+
| John      | Smith    | 39  |
+-----+-----+-----+
| Mary     | Jane     | 25  |
+-----+-----+-----+
| Jennifer | Doe     | 28  |
+-----+-----+-----+
```

I prefer to use the *'fancy_grid'* argument for *tablefmt*:

```
print(tabulate(table, headers='firstrow', tablefmt='fancy_grid'))
```

```
print(tabulate(table, headers='firstrow', tablefmt='fancy_grid'))
```

First Name	Last Name	Age
John	Smith	39
Mary	Jane	25
Jennifer	Doe	28

QUESTION BANK

1. Write brief notes on Iteration
2. Discuss Conditional execution with example
3. Explain Alternative execution with example
4. Describe Nested conditionals
5. What is return statement discuss with example
6. What is Recursion
7. Explain recursion with example
8. Explain Stack diagrams for recursive functions
9. Write brief notes on Multiple assignment
10. Discuss while statement
11. How to create tables using python
12. Discuss Two-dimensional tables
13. Discuss the steps for creating tables
14. Write brief notes on Recursion with example
15. What are recursive functions

TEXT /REFERENCE BOOKS:

1. Allen Downey, Jeffrey Elkner, Chris Meyers. How to think like a computer scientist learning with Python / 1st Edition,2012
2. Kenneth A. Lambert, The Fundamentals of Python: First Programs, 2011, Cengage Learning, ISBN: 978- 1111822705

USEFUL WEBSITES:

1. <http://docs.python.org/3/tutorial/index.html>
2. <http://www.ibiblio.org/g2swap/byteofpython/read/>



SATHYABAMA

**INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)**

Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE

www.sathyabama.ac.in

SCHOOL OF COMPUTING

**DEPARTMENT OF COMPUTER SCIENCE AND
ENGINEERING**

PYTHON PROGRAMMING – SBSA1202

Unit V

Strings and Lists: String as a compound data type, Length, Traversal and the for loop, String slices, String comparison, A find function, Looping and counting, List values, accessing elements, List length, List membership, Lists and for loops, List operations, List deletion. Cloning lists, Nested lists.

1. String as a compound data type

A compound data type

Strings are qualitatively different from the other four because they are made up of smaller pieces - **characters**. Types that comprise smaller pieces are called **compound data types**. Depending on what we are doing, we may want to treat a compound data type as a single thing, or we may want to access its parts.

The **bracket operator** selects a single character from a string:

```
>>> fruit = "banana"
>>> letter = fruit[1]
>>> print letter
```

The expression `fruit[1]` selects character number 1 from `fruit`. The variable `letter` refers to the result. When we display `letter`, we get a surprise:

```
a
```

The first letter of "banana" is not a, unless you are a computer scientist. For perverse reasons, computer scientists always start counting from zero. The 0th letter (zero-eth) of "banana" is b. The 1th letter (one-eth) is a, and the 2th (two-eth) letter is n.

If you want the zero-eth letter of a string, you just put 0, or any expression with the value 0, in the brackets:

```
>>> letter = fruit[0]
>>> print letter
b
```

The expression in brackets is called an **index**. An index specifies a member of an ordered set, in this case the set of characters in the string. The index *indicates* which one you want, hence the name. It can be any integer expression.

Length

The `len` function returns the number of characters in a string:

```
>>> fruit = "banana"
>>> len(fruit)
6
```

To get the last letter of a string, you might be tempted to try something like this:

```
length = len(fruit)
last = fruit[length]    # ERROR!
```

That won't work. It causes the runtime error `IndexError: string index out of range`. The reason is that there is no 6th letter in "banana". Since we started counting at zero, the six letters are numbered 0 to 5. To get the last character, we have to subtract 1 from `length`:

```
length = len(fruit)
last = fruit[length-1]
```

Alternatively, we can use **negative indices**, which count backward from the end of the string. The expression `fruit[-1]` yields the last letter, `fruit[-2]` yields the second to last, and so on.

Traversal and the for loop

A lot of computations involve processing a string one character at a time. Often they start at the beginning, select each character in turn, do something to it, and continue until the end. This pattern of processing is called a **traversal**. One way to encode a traversal is with a `while` statement:

```
index = 0
while index < len(fruit):
    letter = fruit[index]
    print letter
    index += 1
```

This loop traverses the string and displays each letter on a line by itself. The loop condition is `index < len(fruit)`, so when `index` is equal to the length of the string, the condition is false, and the body of the loop is not executed. The last character accessed is the one with the index `len(fruit)-1`, which is the last character in the string.

Using an index to traverse a set of values is so common that Python provides an alternative, simpler syntax — the `for` loop:

```
for char in fruit:  
    print char
```

Each time through the loop, the next character in the string is assigned to the variable `char`. The loop continues until no characters are left.

The following example shows how to use concatenation and a `for` loop to generate an abecedarian series. Abecedarian refers to a series or list in which the elements appear in alphabetical order. For example, in Robert McCloskey's book *Make Way for Ducklings*, the names of the ducklings are Jack, Kack, Lack, Mack, Nack, Ouack, Pack, and Quack. This loop outputs these names in order:

```
prefixes = "JKLMNOPQ"  
suffix = "ack"  
for letter in prefixes:  
    print letter + suffix
```

The output of this program is:

```
Jack  
Kack  
Lack  
Mack  
Nack  
Oack  
Pack  
Qack
```

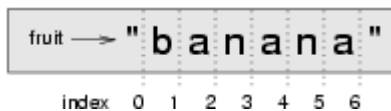
Of course, that's not quite right because `Ouack` and `Quack` are misspelled. You'll fix this as an exercise below.

String slices

A substring of a string is called a **slice**. Selecting a slice is similar to selecting a character:

```
>>> s = "Peter, Paul, and Mary"
>>> print s[0:5]
Peter
>>> print s[7:11]
Paul
>>> print s[17:21]
Mary
```

The operator `[n:m]` returns the part of the string from the *n*-eth character to the *m*-eth character, including the first but excluding the last. This behavior is counterintuitive; it makes more sense if you imagine the indices pointing *between* the characters, as in the following diagram:



If you omit the first index (before the colon), the slice starts at the beginning of the string. If you omit the second index, the slice goes to the end of the string. Thus:

```
>>> fruit = "banana"
>>> fruit[:3]
'ban'
>>> fruit[3:]
'ana'
```

What do you think `s[:]` means?

String comparison

The comparison operators work on strings. To see if two strings are equal:

```
if word == "banana":
    print "Yes, we have no bananas!"
```

Other comparison operations are useful for putting words in [lexicographical order](#):

```
if word < "banana":
    print "Your word, " + word + ", comes before banana."
elif word > "banana":
```

```
print "Your word, " + word + ", comes after banana."
```

else:

```
print "Yes, we have no bananas!"
```

This is similar to the alphabetical order you would use with a dictionary, except that all the uppercase letters come before all the lowercase letters. As a result:

```
Your word, Zebra, comes before banana.
```

A common way to address this problem is to convert strings to a standard format, such as all lowercase, before performing the comparison. A more difficult problem is making the program realize that zebras are not fruit.

Strings are immutable

It is tempting to use the `[]` operator on the left side of an assignment, with the intention of changing a character in a string. For example:

```
greeting = "Hello, world!"
greeting[0] = 'J'          # ERROR!
print greeting
```

Instead of producing the output `Jello, world!`, this code produces the runtime error `TypeError: 'str' object doesn't support item assignment`.

Strings are **immutable**, which means you can't change an existing string. The best you can do is create a new string that is a variation on the original:

```
greeting = "Hello, world!"
new_greeting = 'J' + greeting[1:]
print new_greeting
```

The solution here is to concatenate a new first letter onto a slice of `greeting`. This operation has no effect on the original string.

The `in` operator

The `in` operator tests if one string is a substring of another:

```
>>> 'p' in 'apple'
True
>>> 'i' in 'apple'
```

```
False
>>> 'ap' in 'apple'
True
>>> 'pa' in 'apple'
False
```

Note that a string is a substring of itself:

```
>>> 'a' in 'a'
True
>>> 'apple' in 'apple'
True
```

Combining the `in` operator with string concatenation using `+`, we can write a function that removes all the vowels from a string:

```
def remove_vowels(s):
    vowels = "aeiouAEIOU"
    s_without_vowels = ""
    for letter in s:
        if letter not in vowels:
            s_without_vowels += letter
    return s_without_vowels
```

Test this function to confirm that it does what we wanted it to do.

A find function

What does the following function do?

```
def find(strng, ch):
    index = 0
    while index < len(strng):
        if strng[index] == ch:
            return index
        index += 1
    return -1
```

In a sense, `find` is the opposite of the `[]` operator. Instead of taking an index and extracting the corresponding character, it takes a character and finds the index where that character appears. If the character is not found, the function returns `-1`. This is the first example we have seen of a `return` statement inside a loop. If `strng[index] == ch`, the function returns immediately, breaking out of the loop prematurely. If the character doesn't appear in the string, then the program exits the loop normally and returns `-1`. This pattern of computation is sometimes called a eureka traversal because as soon as we find what we are looking for, we can cry Eureka! and stop looking.

Looping and counting

The following program counts the number of times the letter `a` appears in a string, and is another example of the counter pattern introduced in [Counting digits](#):

```
fruit = "banana"
count = 0
for char in fruit:
    if char == 'a':
        count += 1
print count
```

Lists

A **list** is a sequential collection of Python data values, where each value is identified by an index. The values that make up a list are called its **elements**. Lists are similar to strings, which are ordered collections of characters, except that the elements of a list can have any type and for any one list, the items can be of different types.

List Values

There are several ways to create a new list. The simplest is to enclose the elements in square brackets (`[` and `]`).

```
[10, 20, 30, 40]
```

```
["spam", "bungee", "swallow"]
```

The first example is a list of four integers. The second is a list of three strings. As we said above, the elements of a list don't have to be the same type. The following list contains a string, a float, an integer, and another list.

```
["hello", 2.0, 5, [10, 20]]
```

A list within another list is said to be nested and the inner list is often called a sublist. Finally, there is a special list that contains no elements. It is called the empty list and is denoted []. As you would expect, we can also assign list values to variables and pass lists as parameters to functions.

```
vocabulary = ["iteration", "selection", "control"]
numbers = [17, 123]
empty = []
mixedlist = ["hello", 2.0, 5*2, [10, 20]]
print(numbers)
print(mixedlist)
newlist = [ numbers, vocabulary ]
print(newlist)
```

List Length

As with strings, the function len returns the length of a list (the number of items in the list). However, since lists can have items which are themselves lists, it important to note that len only returns the top-most length. In other words, sublists are considered to be a single item when counting the length of the list.

```
alist = ["hello", 2.0, 5, [10, 20]]
print(len(alist))
print(len(['spam!', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]))
```

Accessing Elements

The syntax for accessing the elements of a list is the same as the syntax for accessing the characters of a string. We use the index operator ([] – not to be confused with an empty list). The expression inside the brackets specifies the index. Remember that the indices start at 0. Any integer expression can be used as an index and as with strings, negative index values will locate items from the right instead of from the left.

```
numbers = [17, 123, 87, 34, 66, 8398, 44]
print(numbers[2])
print(numbers[9 - 8])
print(numbers[-2])
print(numbers[len(numbers) - 1])
```

List Membership

in and not in are boolean operators that test membership in a sequence. We used them previously with strings and they also work here.

```
fruit = ["apple", "orange", "banana", "cherry"]
print("apple" in fruit)
print("pear" in fruit)
```

Lists and for loops

It is also possible to perform list traversal using iteration by item as well as iteration by index.

```
fruits = ["apple", "orange", "banana", "cherry"]
for afruit in fruits: # by item
    print(afruit)
```

It almost reads like natural language: For (every) fruit in (the list of) fruits, print (the name of the) fruit.

We can also use the indices to access the items in an iterative fashion.

```
fruits = ["apple", "orange", "banana", "cherry"]
for position in range(len(fruits)): # by index
    print(fruits[position])
```

In this example, each time through the loop, the variable position is used as an index into the list, printing the position-th element. Note that we used len as the upper bound on the range so that we can iterate correctly no matter how many items are in the list.

Any sequence expression can be used in a for loop. For example, the range function returns a sequence of integers.

```
for number in range(20):
    if number % 3 == 0:
        print(number)
```

This example prints all the multiples of 3 between 0 and 19.

Since lists are mutable, it is often desirable to traverse a list, modifying each of its elements as you go. The following code squares all the numbers from 1 to 5 using iteration by position.

```
numbers = [1, 2, 3, 4, 5]
print(numbers)
for i in range(len(numbers)):
    numbers[i] = numbers[i] ** 2
print(numbers)
```


List Methods

The dot operator can also be used to access built-in methods of list objects. `append` is a list method which adds the argument passed to it to the end of the list. Continuing with this example, we show several other list methods. Many of them are easy to understand.

```
mylist = []
mylist.append(5)
mylist.append(27)
mylist.append(3)
mylist.append(12)
print(mylist)
mylist.insert(1, 12)
print(mylist)
print(mylist.count(12))
print(mylist.index(3))
print(mylist.count(5))
mylist.reverse()
print(mylist)
mylist.sort()
print(mylist)
mylist.remove(5)
print(mylist)
lastitem = mylist.pop()
```

There are two ways to use the `pop` method. The first, with no parameter, will remove and return the last item of the list. If you provide a parameter for the position, `pop` will remove and return the item at that position. Either way the list is changed.

The following table provides a summary of the list methods shown above. The column labeled `result` gives an explanation as to what the return value is as it relates to the new value of the list. The word `mutator` means that the list is changed by the method but nothing is returned (actually `None` is returned). A hybrid method is one that not only changes the list but also returns a value as its result. Finally, if the result is simply a return, then the list is unchanged by the method.

Be sure to experiment with these methods to gain a better understanding of what they do.

Method	Parameters	Result	Description
append	item	mutator	Adds a new item to the end of a list
insert	position, item	mutator	Inserts a new item at the position given
pop	none	hybrid	Removes and returns the last item
pop	position	hybrid	Removes and returns the item at position
sort	none	mutator	Modifies a list to be sorted
reverse	none	mutator	Modifies a list to be in reverse order
index	item	return idx	Returns the position of first occurrence of item
count	item	return ct	Returns the number of occurrences of item
remove	item	mutator	Removes the first occurrence of item

It is important to remember that methods like `append`, `sort`, and `reverse` all return `None`. This means that re-assigning `mylist` to the result of sorting `mylist` will result in losing the entire list.

Calls like these will likely never appear as part of an assignment statement

```
mylist = []
mylist.append(5)
mylist.append(27)
mylist.append(3)
mylist.append(12)
print(mylist)
mylist = mylist.sort() #probably an error
print(mylist)
```

List Deletion

Using slices to delete list elements can be awkward and therefore error-prone. Python provides an alternative that is more readable. The `del` statement removes an element from a list by using its position.

```
a = ['one', 'two', 'three']
del a[1]
print(a)
alist = ['a', 'b', 'c', 'd', 'e', 'f']
del alist[1:5]
```

```
print(alist)
```

As you might expect, `del` handles negative indices and causes a runtime error if the index is out of range. In addition, you can use a slice as an index for `del`. As usual, slices select all the elements up to, but not including, the second index, but do not cause runtime errors if the index limits go too far.

Cloning Lists

If we want to modify a list and also keep a copy of the original, we need to be able to make a copy of the list itself, not just the reference. This process is sometimes called cloning, to avoid the ambiguity of the word copy. The easiest way to clone a list is to use the slice operator. Taking any slice of a creates a new list. In this case the slice happens to consist of the whole list.

```
a = [81, 82, 83]
b = a[:]    # make a clone using slice
print(a == b)
print(a is b)
b[0] = 5
print(a)
print(b)
```

Now we are free to make changes to `b` without worrying about `a`. Again, we can clearly see in code lense that `a` and `b` are entirely different list objects.

Nested Lists

A nested list is a list that appears as an element in another list. In this list, the element with index 3 is a nested list. If we `print(nested[3])`, we get `[10, 20]`. To extract an element from the nested list, we can proceed in two steps. First, extract the nested list, then extract the item of interest. It is also possible to combine those steps using bracket operators that evaluate from left to right.

```
nested = ["hello", 2.0, 5, [10, 20]]
innerlist = nested[3]
print(innerlist)
item = innerlist[1]
print(item)
print(nested[3][1])
```

QUESTION BANK

1. Define Strings and Lists
2. Discuss String as a compound data type,
3. Explain string Length with example
4. Write brief notes on Traversal and the for loop
5. Describe String slices with example
6. Explain String comparison with example
7. Explain find function
8. Write brief notes on Looping
9. Write brief notes on counting
10. Describe List and List values
11. How to Accessing elements in the list
12. What is List membership
13. What are the List operations?
14. Write brief notes on Cloning lists, Nested lists
15. Discuss about List deletion.

TEXT /REFERENCE BOOKS:

1. Allen Downey, Jeffrey Elkner, Chris Meyers.How to think like a computer scientist learning with Python / 1st Edition,2012
2. Kenneth A. Lambert, The Fundamentals of Python: First Programs, 2011, Cengage Learning, ISBN: 978- 1111822705

USEFUL WEBSITES:

1. <http://docs.python.org/3/tutorial/index.html>
2. <http://www.ibiblio.org/g2swap/byteofpython/read/>