Unit 1 - Introduction to Python Programming

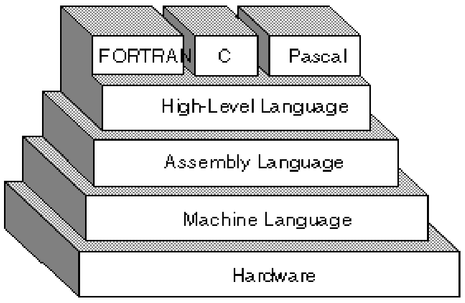
Module 1 - What is Python Programming Language and its Classifications

1.1 What is a Language?

- Language is the method of human communication, either spoken or written, it consisting of the use of words in a structured and conventional way.
- The Language is nothing but set of instructions we are used to communicate.
- To communicate a particular person, we are passing instruction using a particular language like English, Telugu, and Hindi…. etc.
- But while using a language we need to follow some of instructions, some rules are already they have given.
- What are the rules? If I want to speak in English, to form a sentence, first we should be good at grammatically, are else we cannot for m a sentence, to speak in English language.
- Similarly, computer language is also for communication sake only

1.2 What is Programming Language?

- Programming Language is also like English, Telugu…etc.
- It should contain vocabulary and set of grammatical rules for instructing computer to perform specific tasks.
- Each language has a unique set of keywords and a special syntax fc organizing program instructions
- Programming languages are classified as:
- Machine language, Assembly language and High-level language
- The term programming language usually refers to high-level languages, such as BASIC, C, C++, COBOL, FORTRAN, Ada, and Pascal



1.3 Types of Programming Languages

1.3.1 Machine Language: The language of 0s and 1s is called as machine language. This is the only language which can be understood computers directly.

Merits:

- It is directly understood by the processor so has faster execution time since the programs written in this language need not to be translated.
- It doesn't need larger memory.

Demerits:

- It is very difficult to program using Machine Language since all the instructions are to be represented by 0 and 1.
- Use of this language makes programming time consuming.
- It is difficult to find error and to debug.
- It can be used by experts only.

1.3.2 Assembly Languages: It is low level programming language in which the sequence of 0's and 1's are replaced by mnemonic (ni-monic) codes. Typical instructions for addition and subtraction.

Example: ADD for addition, SUB for subtraction etc.

Since our system only understands the language of 0s and 1s .therefore, a system program is known as assembler. Which is designed to translate an assembly language program into the machine language program?

Merits:

- It is makes programming easier than Machine Language since it uses mnemonics code for programming. Eg: ADD for addition, SUB for subtraction, DIV for division, etc.
- It makes programming process faster.
- Error can be identified much easily compared to Machine Language
- It is easier to debug than machine language.

Demerits:

- Programs written in this language is not directly understandable by computer so translators should be used.
- It is hardware dependent language so programmers are forced to think in terms of computer's architecture rather than to the problem being solved.
- Being machine dependent language, programs written in this language are very less or not portable.
- Programmers must know its mnemonics codes to perform any task.

1.3.3 High Level Language: High level languages are English like statements and programs Written in these languages are needed to be translated into machine language before execution using a system software such as compiler. Program written in high level languages are much easier to maintain and modified.

- High level language program is also called source code.
- Machine language program is also called object code

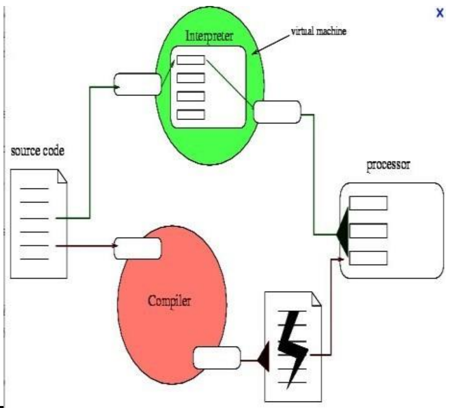
Merits:

- Since it is most flexible, variety of problems could be solved easily
- Programmer does not need to think in term of computer architecture which makes them focused on the problem.
- Programs written in this language are portable.

Demerits:

- It is easier but needs higher processor and larger memory.
- It needs to be translated therefore its execution time is more.

1.4 Interpreter and Compiler



- We generally write a computer program using a high-level language. A high-level language is one which is understandable by humans.
- But a computer does not understand high-level language. It only understands program written in 0's and 1's in binary, called the machine code or object code.

A program written in high-level language is called a source code. We need to convert the source code into machine code and this is accomplished (actioned) by compilers and interpreters.

Interpreter	Compiler
Translates program one statement at a time.	Scans the entire program and translates it into machine code.
It takes less amount of time to analyze the source code but the overall execution time is slower.	It takes large amount of time to analyze the source code but the overall execution time is comparatively faster.
No intermediate object code is generated, hence are memory efficient.	Generates intermediate object code which further requires linking, hence requires more memory.
Continues translating the program until the first error is met, in which case it stops. Hence debugging is easy.	It generates the error message only after scanning the whole program. Hence debugging is comparatively hard.
Programming language like Python, Ruby use interpreters.	Programming language like C, C++ use compilers.

Module 2 – Introduction to Python Programming

1.5 Introduction

- Python is an interpreter, object-oriented, high-level programming language with dynamic semantics (substance).
- Python's simple, easy to learn syntax emphasizes readability and therefore reduces the cost of program maintenance.
- Python supports modules and packages, which encourages program modularity and code reuse
- It consists of high-level built in data structures, combined with dynamic typing and dynamic binding; make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together.
- Python is an open-source programming language made to both look good and be easy to read.

1.6 History of Python

- Python is an old language created by Guido Van Rossum. The design began in the late 1980s and was first released in February 1991.
- Python is influenced by following programming languages:
    - ABC language.
    - Modula-3

1.7 Why Python was created?

- In late 1980s, Guido Van Rossum was working on the Amoeba distributed operating system group.
- He wanted to use an interpreted language like ABC (ABC has simple easy-to-understand syntax) that could access the Amoeba system calls.
- So, he decided to create a language that was extensible. This led to a design of new language which was later named Python.
- Python drew inspiration from other programming languages like C, C++, Java, Perl, and Lisp.

1.8 Why the name was Python?

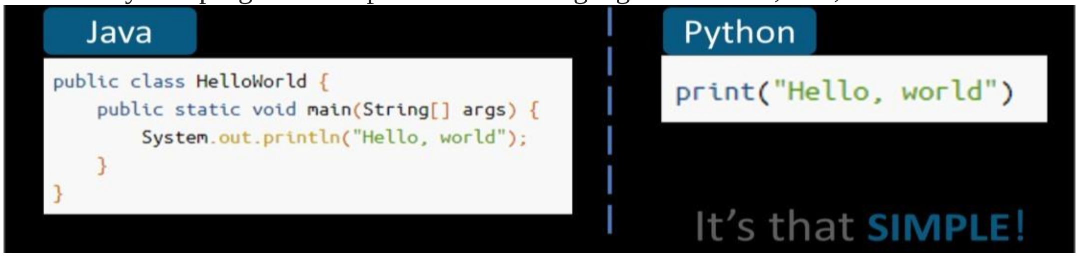
- No. It wasn't named after a dangerous snake. Rossum was fan of a comedy series Monty Python's Flying Circus in late seventies.
- The name "Python" was adopted from the same series "Monty Python's Flying Circus".

Release Dates of Different Versions

Version	Release Data
Python1.0(firststandardrelease) Python1.6 (Lastminorversion)	January1994 September5,2000
Python2.0(Introducedlistcomprehensions) Python2.7 (Last minorversion)	October 16,2000 July3, 2010
Python3.0(Emphasis onremovingduplicativeconstructs and module) Python3.5	December 3, 2008 September13,2015
Python 3.7	June 27, 2018
Python 3.8.0 (Last updated version)	Oct. 14, 2019

1.9 Features of Python Programming

A simple language which is easier to learn Python has a very simple and elegant graceful) syntax. It's much easier to read and write Python programs compared to other languages like: C++,Java, C#.



```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, world");
    }
}
```

```
print("Hello, world")
```

Free and open-source
You can even make changes to the Python's source code and update.

Portability
You can move Python programs from one platform to another, and run it without any changes. It runs smoothly on almost all platforms including Windows, Mac OS X and Linux

Extensible and Embeddable
Suppose an application requires high performance. You can easily combine pieces of C/C++ or other languages with Python code and other languages may not provide out of the box

A high-level, interpreted language
Unlike C/C++, you don't have to worry about daunting (Cause to lose courage) tasks like memory management, garbage collection and so on, likewise, when you run Python code, it automatically converts your code to the language your computer understands. You don't need to worry about any lower-level operations

Large standard libraries to solve common tasks
- Python has a number of standard libraries which makes life of a programmer much easier since you don't have to write all the code yourself
- For example: Need to connect **MySQL** database on a Web server? You can use **MySQLdb** library using **import MySQLdb**

Object-oriented
- Everything in Python is an object. Object oriented programming (OOP) helps you solve a complex problem intuitively.
- With OOP, you are able to divide these complex problems into smaller sets by creating objects.

Scientific and Numeric Computing
- There are numerous libraries available in Python for scientific and numeric computing. There are libraries like: **SciPy** and **NumPy** that are used in general purpose computing. And,
- There are specific libraries like: **EarthPy** for earth science, **AstroPy** for Astronomy and so on.
- Also, the language is heavily used in machine learning, data mining and deep learning.

1.10 Why Python is ver y easy to learn?
- One big change with Python is the use of white space to define code: spaces or tabs are used to organize code by the amount of spaces or tabs.
- This means at the end of each line; a **semicolon** is not needed and curly braces ({}) are not used to group the code.
- Which are both common in C. The combined effect makes Python a very easy to read language.

4 Reasons to Choose Python as First Language

1.10.1 Simple Elegant (Graceful) Syntax

- It's easier to understand and write Python code.
- Why? Syntax feels Naturals with Example Code

```
A=12
B=23
sum=A+B
print(sum)
```

- Even if you have never programmed before, you can easily guess that this program adds two numbers and prints it.

1.10.2 Not overly strict

- You don't need to define the type of a variable in Python. Also, it's not necessary to add semicolon at the end of the statement.
- Python enforces you to follow good practices (like proper indentation). These small things can make learning much easier for beginners.

1.10.3 Expressiveness of the language

- Python allows you to write programs having greater functionality with fewer (less) lines of code.
- We can build game **(Tic-tac-toe)**with Graphical interface in less than 500 lines of code
- This is just an example. You will be amazed how much you can do with Python once you learn the basics.

1.10.4 Great Community and Support

- Python has a large supporting community.
- There are numerous active forums online which can be handy if you are stuck

1.11 Install and Run Python

Ubuntu

1) Install the following dependencies:

```
$sudo apt-get install build-essential check install
$sudo apt-get installlibsqlite3-dev
$sudo apt-get installlibbz2-dev
(libreadline-gplv2-devlibncursesw5-devlibssl-devtk-devlibgdbm-devlibc6-dev)
```

2) Go to Download Python page on the official site and click **DownloadPython3.4.3**
3) In the terminal, go to the directory where the file is downloaded and run the command:
   a. $tar-xvfPython-3.4.3.tgz
   b. This will extract your zipped file.

   Note: The filename will be different if you've downloaded a different version. Use the appropriate filename
4) Go to the extracted directory.

```
$ cd Python-3.4.3
```

5) Issue the following commands to compile Python source code on your Operating system

```
$./configure
$ make
$ make install
```

6) Go to Terminal and type the following to run sample ‘helloworld‘ Program

```
ubuntu@~$python3.4.3
>>>print(‘HellowWorld’)
Hello World
```

Windows

1) Go to Download Python page on the official site and click Download Python 3.4.3
2) When the download is completed, double-click the file and follow the instructions to install it
3) When Python is installed, a program called IDLE is also installed along with it. It provides graphical user interface to work with Python
4) Open IDLE, copy the following code below and press enter

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

5) To create a file in IDLE, go to File > New Window (Shortcut: Ctrl+N).
6) Write Python code (you can copy the code below for now) and save (Shortcut: Ctrl+S) with .py file extension like: hello.py or your-first-program.py

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

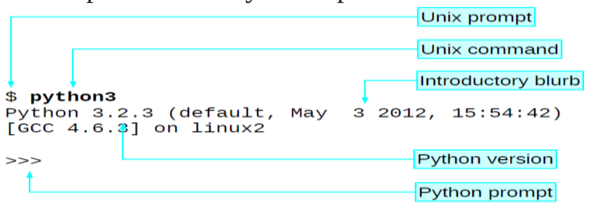
7) Go to Run > Run module (Shortcut: F5) and you can see the output. Congratulations, you've successfully run your first Python program

1.12 Modes of running

There are various ways to start Python

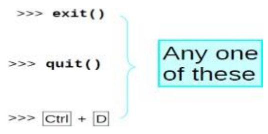
Immediate Mode or Interactive Mode

- Typing python in the command line will invoke the interpreter in immediate mode. We can directly type in Python expressions and press enter to get the output **(>>>)**
- Is the python prompt and it tells us interpreter is ready for input



```
$ python3
Python 3.2.3 (default, May  3 2012, 15:54:42)
[GCC 4.6.3] on linux2

>>>
```

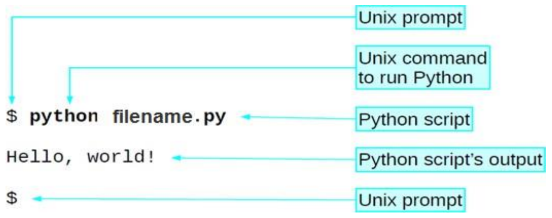
Quitting Python



```
>>> exit()

>>> quit()

>>> Ctrl + D
```

Script Mode

- This mode is used to execute Python program written in a file. Such a file is called a script. Scripts can be saved to disk for future use. Python scripts should have the extension .py, it means that the filename ends with **.py**.
- For example: **helloWorld.py**
- To execute this file in script mode we simply write **python3 helloWorld.py** at the command prompt.



```
$ python  filename.py

Hello, world!

$
```

Integrated Development Environment (IDE)

We can use any text editing software to write a Python script file. Like Notepad, Editplus, sublime…etc

1.13 Python Program to Print Helloworld!

- Type the following code in any text editor or an IDE and Save it as **helloworld.py**
- Now at the command window, go to the location of this file, use **cd** command to **change directory**
- To run the script, type **python3helloworld.py** in the command window then we get output like this: **HelloWorld**
- In this program we have used the built-in function **print(),** to print out a string to the screen
- String is the value inside the quotation marks **i.e. HelloWorld**

Let us execute programs in different modes of programming.

Interactive Mode Programming:

Invoking the interpreter without passing a script file as a parameter brings up the following prompt:

$ python3
Python 3.7 (r27:82525, Jul 4 2010, 09:01:59) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.

Type the following text at the Python prompt and press the Enter:

```
>>> print "Hello, IIIT RK Valley, RGUKT-AP!";
```

If you are running new version of Python, then you need to use print statement with parenthesis as in print ("Hello, IIIT RK Valley");.However in Python version 2.7, this produces the following result:

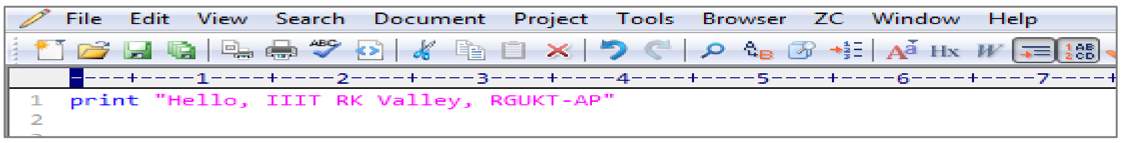
```
Hello, IIIT RK Valley, RGUKT-AP!
```

Script Mode Programming:

Python programs must be written with a structure. The syntax must be correct, or the interpreter will generate error messages and not execute the program. This section introduces Python by providing a simple example program.

To write Python programming in script mode we have to use editors. Let us write a simple Python program in a script mode using editors. Python files have extension .py. Type the following source code in a simple.py file.

Program (simple.py) is one of the simplest Python programs that does something



```
print "Hello, IIIT RK Valley, RGUKT-AP"
```

We assume that you have Python interpreter set in PATH variable. Now, try to run this program as follows:

```
$ python3 simple.py
```

This produces the following result:

```
Hello, IIIT RK Valley, RGUKT-AP!
```

Module 3 - Reserved key words, Identifiers, Variables and Constant

1.14 Keywords

- Keywords are the reserved words in Python and we cannot use a keyword as 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 35 keywords in Python 3.7.3 This number keep on growing with the new features coming in python
- All the keywords except True, False and None are in lowercase and they must be written as it is.
- The list of all the keywords is given below

We can get the complete list of keywords using python interpreter help utility.

```
$ python3
>>>help()
help> keywords
```

1.15 Identifiers

Identifier is the name given to entities like class, functions, variables etc. in Python.

Rules for writing identifiers

- Identifiers can be a combination of letters in lowercase (a to z) or uppercase (A to Z) or digits (0 to9) or an underscore (_).
- Names like myClass, var_1and print_this_to_screen, all are valid example.
- An identifier cannot start with a digit.1variable is invalid, but variable1is perfectly fine.
- Keywords cannot be used as identifiers.

```
>>> global=1File"<interactiveinput>",line1
global=1
^SyntaxError: invalid syntax
```

- We cannot use special symbols like!, @, #, $,% etc. in our identifier. >>>a@ =0

```
File"<interactive input>", line1a@= 0^
Syntax Error: invalid syntax
```

- Identifier can be of any length.

1.16 Things to care about

- Python is a case-sensitive language. This means, **Variable** and **variable** are not the same. Always give a valid name to identifiers so that it makes sense.
- While, c = 10is valid. Writing count =10would make more sense and it would be easier to figure out what it does even when you look at your code after a long gap.
- Multiple words can be separated using an underscore, this_is_a_long_variable.
- We can also use camel-case style of writing,
- i.e., capitalize every first letter of the word except the initial word without any spaces.
- For example: camelCase Example.

1.17 Variable

- A variable is a location in memory used to store some data.
- Variables are nothing but reserved memory locations to store values, this means that when we create a variable, we reserved some space in memory.
- They are given unique names to differentiate between different memory locations.
- The rules for writing a variable name are same as the rules for writing identifiers in Python.
- We don't need to declare a variable before using it.
- In Python, we simply assign a value to a variable and it will exist.
- We don't even have to declare the type of the variable. This is handled internally according to the type of value we assign to the variable.

Variable assignment: We use the assignment operator (=) to assign values to a variable. Any type of value can be assigned to any valid variable.

```
a= 5
b = 3.2
c = "Hello"
```

Here, we have three assignment statements. 5 is an integer assigned to the variable a. Similarly, 3.2 is a floating-point number and "Hello" is a string (sequence of characters) assigned to the variables b and c respectively.

Multiple assignments:

- In Python, multiple assignments can be made in a single statement as follows: *a, b, c = 5,3.2, "Hello"*
- If we want to assign the same value to multiple variables at once, we can do this as *x = y= z ="same"*
- This assigns the "same" string to all the three variables.

Constants: A constant is a type of variable whose value cannot be changed. It is helpful to think of constants as containers that hold information which cannot be changed later. You can think of constants as a bag to store some books which cannot be replaced once placed inside the bag.

Assigning value to constant in Python: In Python, constants are usually declared and assigned in a module. Here, the module is a new file containing variables, functions, etc which is imported to the main file. Inside the module, constants are written in all capital letters and underscores separating the words.

Example 1: Declaring and assigning value to a constant

Create a ***constant.py****:*

```
PI = 3.14
GRAVITY = 9.8
```

Create a ***main.py****:*

```
import constant
print(constant.PI)
print(constant.GRAVITY)
```

Output

```
3.14
9.8
```

In the above program, we create a **constant.py** module file. Then, we assign the constant value to *PI* and *GRAVITY*. After that, we create a **main.py** file and import the constant module. Finally, we print the constant value.

Note: In reality, we don't use constants in Python. Naming them in all capital letters is a convention to separate them from variables; however, it does not actually prevent reassignment.

1.18 Statements & Comments

Instructions that a Python interpreter can execute are called statements.

For example, *a = 1* is an assignment statement. If statement, for statement, while statement etc.

1.18.1 Multi-line statement

- In Python, end of a statement is marked by a new line character. But we can make a statement extend over multiple lines with the line continuation character (\).

For example:

```
a = 1 + 2 + 3 + \
4 + 5 + 6 + \
7 + 8 + 9
```

This is explicit line continuation.

- This is explicit line continuation. In Python, line continuation is implied inside parentheses( ), brackets[ ] and braces{}.
- For instance, we can implement the above multi-line statement as

  a = (1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9)
- Here, the surrounding parentheses ( ) do the line continuation implicitly. Same is the case with []and{ }.

  For example: *colors = ['red','blue','green']*
- We could also put multiple statements in a single line using semicolons, as follows a= 1; b=2;c =3

1.18.2 Comments

- Comments are very important while writing a program. It describes what's going on inside a program so that a person looking at the source code does not have a hard time figuring it out.
- You might forget the key details of the program you just wrote in a month's time. So taking time to explain these concepts in form of comments is always fruitful.
- In Python, we use the hash (#) symbol to start writing a comment. It extends up to the new line character.
- Comments are for programmers for better understanding of a program. Python Interpreter ignores comment.

```
#this is a comment
#print out Hello
print('Hello')
```

Multi-linecomments

- If we have comments that extend multiple lines, one way of doing it is to use hash(#) in the beginning of each line. For example:

```
#this is a long comment
#and it extends
#to multiple lines
```

- Another way of doing this is to use triple quotes, either ''' or """.
- These triple quotes are generally used for multi-line strings. But they can be used as multi-line comment as well. Unless they are not docstrings, they do not generate any extra code.

```
"""This is also a
perfect example of
multi-line comments"""
```

Python Operators

1.19 Get Started with Python Operators

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

  For example: >>> 2+3
- Here, + is the operator that performs addition. 2 and 3 are the operands and 5 is the output of the operation.
- Python has a number of operators which are classified below.
  - Arithmetic operators
  - Assignment operators
  - Comparison (Relational) operators
  - Logical (Boolean) operators
  - Bitwise operators
  - Assignment operators
  - Special operators

1.19.1 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 Program for Arithmetic operators

```
x = 25
y = 15
print('x + y = ',x+y)
print('x - y = ',x-y)
print('x * y = ',x*y)
print('x / y = ',x/y)
print('x // y = ',x//y)
print('x ** y = ',x**y)
```

1.19.2 Comparison (Relational) Operators

Comparison operators are used to compare values. It either returns 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)
```

1.19.3 Logical (Boolean) 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

Here is an 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)*

1.19.4 Bitwise Operators

Bitwise operators act on operands as if they were string of binary digits. It operates 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	~x = -11 (1111 0101)
^	Bitwise XOR	x ^ y = 14 (0000 1110)
>>	Bitwise right shift	x>> 2 = 2 (0000 0010)
<<	Bitwise left shift	x<< 2 = 40 (0010 1000)

1.19.5 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
Example: *a* += *5* that adds to the variable and later assigns the same. It is equivalent to a = a + 5.

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

1.19.6 Membership Operators

in and **notin** are the membership operators in Python. They are used to test whether a value or variable is found in a sequence (string, list, tuple, set and dictionary).

Operator	Meaning	Example
In	True if value/variable is found in the sequence	5 in x
not in	True if value/variable is not found in the sequence	5 not in x

Here is an example.
x = 'Hello world'
y = {1:'a',2:'b'}
print('H' in x)
print('hello' not in x)
print(1 in y)
print('a' in y)

Here, 'H' is in x but 'hello' is not present in x (remember, Python is case sensitive). Similarly, 1 is key and 'a' is the value in dictionary y. Hence, 'a' in y returns False.

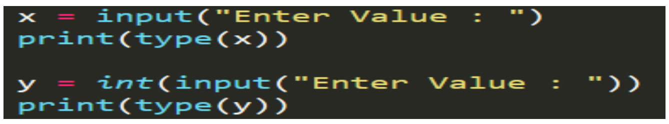
1.20 Rules for solving equations in Python

Order of Operations Worksheets - BEDMAS or PEMDAS
Step 1: First, **perform** the operations within the brackets or parenthesis.
Step 2: Second, evaluate the exponents.
Step 3: Third, **perform** multiplication and division from left to right.
Step 4: Fourth, **perform** addition and subtraction from left to right.
Example: >>>(40+20)*30/10
Output: 180

1.21 Reading Input value from the User

The print() function enables a Python program to display textual information to the user. Python provides built-in functions to get input from the user. The function is input(). Generally input() function is used to retrieve string values from the user.

Program shows the type of user enter values



```
x = input("Enter Value : ")
print(type(x))

y = int(input("Enter Value : "))
print(type(y))
```

Output:

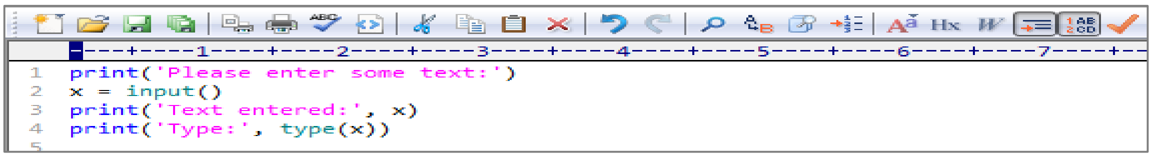
```
Enter Value : 4
<class 'str'>
Enter Value : 4
<class 'int'>
```

We can use the above mentioned functions in *python 2.x*, but not on *python 3.x*. input() in python 3.x always return a string. Moreover, raw_input() has been deleted from python 3

python2.x		python3.x
raw_input()	--------------	input()
input()	-------------------	eval(input())

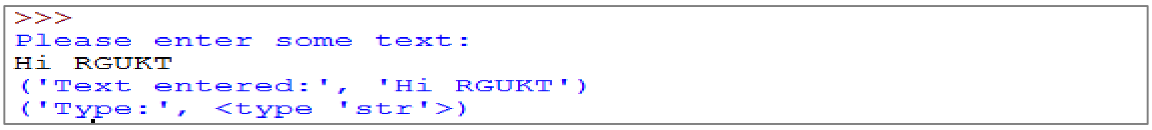
We can simply say in *python 3.x* only use of the input () function to read value from the user and it assigns a string to a variable. x = input()

The parentheses are empty because, the input function does not require any information to do its job.
Program demonstrates that the input function produces a string value.



```
print('Please enter some text:')
x = input()
print('Text entered:', x)
print('Type:', type(x))
```

Output:



```
>>>
Please enter some text:
Hi RGUKT
('Text entered:', 'Hi RGUKT')
('Type:', <type 'str'>)
```

The second line shown in the output is entered by the user, and the program prints the first, third, and fourth lines. After the program prints the message *Please enter some text:*, the program's execution stops and waits for the user to type some text using the keyboard. The user can type, backspace to make changes, and type some more. The text the user types is not committed until the user presses the Enter (or return) key. In Python 3.X, input() function produces only strings, by using conversion functions we can change the *type* of the input value. Example as int(), str() and float().

Unit 2 - Data types, I/O, Types of Errors and Conditional Constructs

Module 1 - Data types, I/O, Types of Errors

A data type is a data storage format that can contain a specific type or range of values. Variables can store data of different types, and different types can do different things. Python has the following data types built-in by default, in these categories.

Numeric Type:	int, float, complex
Sequence Types:	str,list, tuple, range
Mapping Type:	Dict
Set Types:	set, frozenset

2.1.1 Numeric Types

There are three numeric types in Python: *int, float and complex*
Variables of numeric types are created when you assign a value to them
Example: *x = 1 # int, y = 2.8 # float , z = 1j # complex*
Int: Int, or integer, is a whole number, positive or negative, without decimals, of unlimited length.
Example: *x = 1, y = 35656222554887711, z = -3255522*
Float: Float or "floating point number" is a number, positive or negative, containing one or more decimals.
Example: *x = 1.10, y = 1.0, z = -35.59*
Float can also be scientific numbers with an "e" to indicate the power of 10.
Example: *x = 35e3, y = 12E4, z = -87.7e100*
Complex: Complex numbers are written with a "j" as the imaginary part.
Example: *x = 3+5j, y = 5j, z = -5j*

2.1.2 Sequence Types

In Python, sequence is the generic term for an ordered set. There are several types of sequences in Python; the following are the most important.
String: Strings in python are surrounded by either single quotation marks, or double quotation marks. 'hello' is the same as "hello". You can display a string literal with the print() function:
Example: *print ("Hello"); print('Hello')*
Assign String to a Variable, Assigning a string to a variable is done with the variable name followed by an equal sign and the string. Example: *a = "Hello"; print(a)*
Multiline Strings: You can assign a multiline string to a variable by using three quotes, You can use three double quotes.
Example: *a = """Lorem ipsum dolor sit amet, consecteturadipiscingelit, sed do eiusmodemporincididuntutlabore et dolore magna aliqua."""*
print(a)
Or three single quotes:
a = '''Lorem ipsum dolor sit amet, consecteturadipiscingelit, sed do eiusmodtemporincididuntutlabore et dolore magna aliqua.'''
print(a)
However, Python does not have a character data type, a single character is simply a string with a length of 1.
Lists: List is an ordered sequence of items. All the items in a list do not need to be of the same type and lists are mutable - they can be changed. Elements can be reassigned or removed, and new elements can be inserted.
Declaring a list is pretty straight forward. Items separated by commas are enclosed within brackets [ ].
a = [1, 2.2, 'python']
Tuples: Tuples is an ordered sequences of items same as a list. The only difference is that Tuples are immutable. Tuples once created cannot be modified. Tuples are used to write-protect data and are usually faster than lists as they cannot change dynamically. It is defined within parentheses () where items are separated by commas.
t = (5,'program', 1+3j)
range(): The range() type returns an immutable sequence of numbers between the given start integer to the stop integer.
print(list(range(10))

2.1.3 Mapping type

Dictionary is an unordered collection of key-value pairs. In python there is mapping type called dictionary. It is mutable. In Python, dictionaries are defined within braces {} with each item being a pair in the form key-value. Key and value can be of any type. The values of the dictionary can be any type, but the key must be of any immutable data type such as strings, numbers, and Tuples.
>>>d = {1:'value','key':2} >>>type(d)

2.1.4 Set types
Set is an unordered collection of unique items. Set is defined by values separated by comma inside braces { }. Items in a set are not ordered.
a = {5,2,3,1,4}
print("a = ", a) # printing set variable
print(type(a)) # data type of variable a
We can use the set for some mathematical operations like set union, intersection, difference etc. We can also use set to remove duplicates from a collection.

2.2. Getting the Data type

We can use the type() function to know which class a variable or a value belongs to. Similarly, the isinstance() function is used to check if an object belongs to a particular type.
a = 5
print(a, "is of type", type(a))
a = 2.0
print(a, "is of type", type(a))
a = 1+2j
print(a, "is complex number?", isinstance(a,complex))
Output
5 is of type <class 'int'>
2.0 is oftype <class 'float'>
(1+2j) is complexnumber? True

2.3. Type Conversion

The process of converting the value of one data type (integer, string, float, etc.) to another data type is called type conversion. Python has two types of type conversion.
1. Implicit Type Conversion
2. Explicit Type Conversion

2.3.1 Implicit Type Conversion
In Implicit type conversion, Python automatically converts one data type to another data type. This process doesn't need any user involvement. Let's see an example where Python promotes the conversion of the lower data type (integer) to the higher data type (float) to avoid data loss.
Example 1: Converting integer to float.

```
num_int = 123
num_flo = 1.23
num_new = num_int + num_flo
print("datatype of num_int:",type(num_int))
print("datatype of num_flo:",type(num_flo))
print("Value of num_new:",num_new)
print("datatype of num_new:",type(num_new))
```

Output

```
datatype of num_int: <class 'int'>
datatype of num_flo: <class 'float'>
Value of num_new: 124.23
datatype of num_new: <class 'float'>
```

In the above program,
- We add two variables num_int and num_flo, storing the value in num_new.
- We will look at the data type of all three objects respectively.
- In the output, we can see the data type of num_int is an integer while the data type of num_flo is a float.
- Also, we can see the num_new has a float data type because Python always converts smaller data types to larger data types to avoid the loss of data.

Now, let's try adding a string and an integer, and see how Python deals with it.
Example 2: Addition of string(higher) data type and integer(lower) datatype
num_int = 123
num_str = "456"
print("Data type of num_int:",type(num_int))
print("Data type of num_str:",type(num_str))
print(num_int+num_str)
When we run the above program, the output will be:

```
Data type of num_int: <class 'int'>
Data type of num_str: <class 'str'>
Traceback (most recent call last):
  File "python", line 7, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

In the above program,

- We add two variables num_int and num_str.
- As we can see from the output, we got TypeError. Python is not able to use Implicit Conversion in such conditions.
- However, Python has a solution for these types of situations which is known as Explicit Conversion.

2.3.2 Explicit Type Conversion

In Explicit Type Conversion, users convert the data type of an object to required data type. We use the predefined functions like int(), float(), str(), etc to perform explicit type conversion. This type of conversion is also called typecasting because the user casts (changes) the data type of the objects.

Syntax: <required_datatype>(expression)

Typecasting can be done by assigning the required data type function to the expression.

Example 3: Addition of string and integer using explicit conversion

```
num_int = 123
num_str = "456"
print("Data type of num_int:",type(num_int))
print("Data type of num_str before Type Casting:",type(num_str))
num_str = int(num_str)
print("Data type of num_str after Type Casting:",type(num_str))
num_sum = num_int + num_str
print("Sum of num_int and num_str:",num_sum)
print("Data type of the sum:",type(num_sum))
```

When we run the above program, the output will be:

```
Data type of num_int: <class 'int'>
Data type of num_str before Type Casting: <class 'str'>
Data type of num_str after Type Casting: <class 'int'>
Sum of num_int and num_str: 579
Data type of the sum: <class 'int'>
```

In the above program,

- We add num_str and num_int variable.
- We converted num_str from string(higher) to integer(lower) type using int() function to perform the addition.
- After converting num_str to an integer value, Python is able to add these two variables.
- We got the num_sum value and data type to be an integer.

Module 2 - Mutable and Immutable types

A first fundamental distinction that Python makes on data is about whether the value changes are not. If the value can change, then is called mutable, while if the value cannot change, that is called immutable.

Mutable: list, dict and set

Immutable: int, float, complex, string, and tuple

2.5 Input and Output Operations and Formats

Python provides numerous built-in functions that are readily available at the Python prompt.

Some of the functions like input() and print() are widely used for standard input and output operations respectively.

2.5.1 Python Output Using print() function

We use the print() function to output data to the standard output device (screen).

Example 1: *print('This sentence is output to the screen')*

Output: This sentence is output to the screen

Example 2:

```
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.

The actual syntax of the print() function is:

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

- Here, objects are 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). Here is an example to illustrate this.

```
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&
```

2.5.2 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 is5and y is10
```

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

```
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 = 'Goodmorning', 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 is12.35
>>>print('The value of x is %3.4f' %x)
The value of x is12.3457
```

2.5.3 Python Input

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
```

2.6 Types of Errors in python

No matter how smart or how careful you are, errors are your constant companion. With practice, you will get slightly better at not making errors, better at finding and correcting them.There are three kinds of errors: syntax errors, runtime errors, and logic errors.

2.6.1 Syntax errors: Syntax errors are produced by Python when it is translating the source code into byte code. They usually indicate that there is something wrong with the syntax of the program.

Example: Omitting the colon at the end of if statement yields the somewhat redundant message.

Syntax Error: invalid syntax.

Here are some ways to avoid the most common syntax errors:

- Make sure you are not using a Python keyword for a variable name.
- Check that you have a colon at the end of the header of every compound statement, including for, while, if, and def statements.
- Check that indentation is consistent. You may indent with either spaces or tabs but it’s better not to mix them. Each level should be nested the same amount.
- Make sure that strings in the code have matching quotation marks.
- If you have multiline strings with triple quotes (single or double), make sure you have terminated the string properly. An un-terminated string may cause an invalid token error at the end of your program, or it may treat the following part of the program as a string until it comes to the next string. In the second case, it might not produce an error message at all!

- An unclosed bracket – (, {, or [– makes Python continue with the next line as part of the current statement. Generally, an error occurs almost immediately in the next line.
- Check for the classic = instead of == inside a conditional.

2.6.2 Run Time Error: Errors that occur after the code has been executed and the program is running. The error of this type will cause your program to behave unexpectedly or even crash. An example of a runtime error is the division by zero. Consider the following example:

```
x = float(input('Enter a number: '))
y = float(input('Enter a number: '))
z = x/y
print (x,'dividedby',y,'equals: ',z)
```

The program above runs fine until the user enters 0 as the second number:

```
>>> Enter a number: 9
>>>Enter a number: 2
9.0 divided by 2.0 equals: 4.5
>>> Enter a number: 11
>>> Enter a number: 3
11.0 divided by 3.0 equals: 3.6666666666666665
>>> Enter a number: 5
>>> Enter a number: 0
Traceback (most recent call last):
File "C:/Python34/Scripts/error1.py", line 3, in <module>
z = x/y
ZeroDivisionError: float division by zero
```

It is also called semantic errors, logical errors cause the program to behave incorrectly, but they do not usually crash the program. Unlike a program with syntax errors, a program with logic errors can be run, but it does not operate as intended. Consider the following example of logical error:

```
x = float(input('Enter a number: '))
y = float(input('Enter a number: '))
z = x+y/2
print ('The average of the two numbers you have entered is:',z)
```

The example above should calculate the average of the two numbers the user enters. But, because of the order of operations in arithmetic (the division is evaluated before addition) the program will not give the right answer:

```
>>> Enter a number: 3
>>> Enter a number: 4
The average of the two numbers you have entered is: 5.0
```

To rectify this problem, we will simply add the parentheses: z = (x+y)/2

Now we will get the right result:

```
>>> Enter a number: 3
>>> Enter a number: 4
The average of the two numbers you have entered is: 3.5
```

Module 3 - Conditional Constructs

There come situations in real life when we need to make some decisions and based on these decisions, we decide what we should do next. Similar situations arise in programming also where we need to make some decisions and based on these decisions we will execute the next block of code.

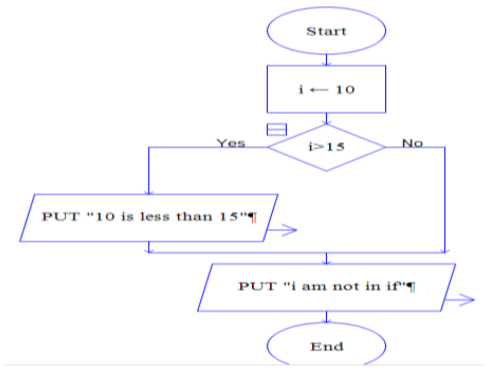
Decision making statements in programming languages decides the direction of flow of program execution. Decision making statements available in python are:

- if statement
- if..else statements
- nested if statements
- if-elif ladder
- Short Hand if statement
- Short Hand if-else statement

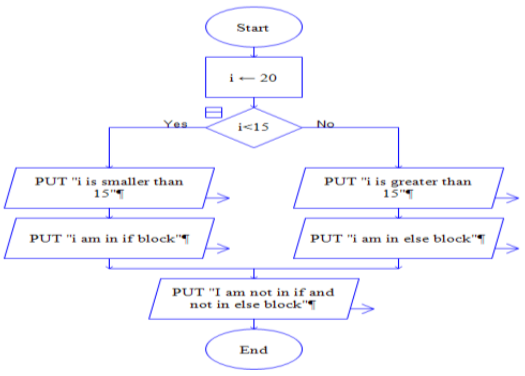
2.7.1 If statement: if statement is the simplest decision-making statement. It is used to decide whether a certain statement or block of statements will be executed or not i.e. if a certain condition is true then a block of statement is executed otherwise not.

Syntax:

if *condition*:

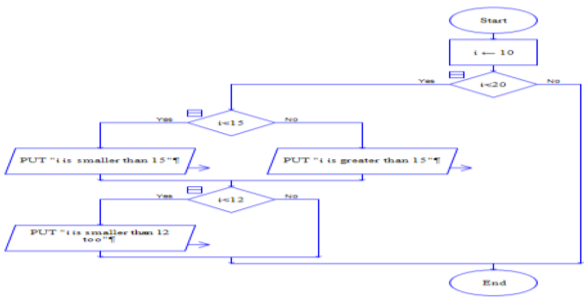
```
   # Statements to execute if
   # condition is true
```
Here, condition after evaluation will be either true or false. if statement accepts boolean values – if the value is true then it will execute the block of statements below it otherwise not. We can use *condition* with bracket '(' ')' also.
As we know, python uses indentation to identify a block. So the block under an if statement will be identified as shown in the below example:
```
if condition:
   statement1
statement2
# Here if the condition is true, if block
# will consider only statement1 to be inside
# its block.
```


Flowchart:- # python program to illustrate If statement
```
i = 10
if (i> 15):
   print ("10 is less than 15")
print ("I am Not in if")
```
Output:
```
I am Not in if
```
2.7.2 if- else: if statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won't. Here comes the *else* statement. We can use the *else* statement with *if* statement to execute a block of code when the condition is false.


Start
i ← 20
i<15
Yes
No
PUT "i is smaller than 15"¶
PUT "i is greater than 15"¶
PUT "i am in if block"¶
PUT "i am in else block"¶
PUT "I am not in if and not in else block"¶
End

Syntax:
```
if (condition):
    # Executes this block if
    # condition is true
else:
    # Executes this block if
    # condition is false
```
```
# Python program to illustrate If else statement
#!/usr/bin/python
i = 20;
if (i< 15):
    print ("i is smaller than 15")
    print ("i'm in if Block")
else:
    print ("i is greater than 15")
    print ("i'm in else Block")
print ("i'm not in if and not in else Block")
```
Output:
```
i is greater than 15
i'm in else Block
i'm not in if and not in else Block
```
The block of code following the else statement is executed as the condition present in the if statement is false after call the statement which is not in block (without spaces).

2.7.3 nested-if: A nested if is if statement that is the target of another if statement. Nested if statements means an if statement inside another if statement. Yes, Python allows us to nest if statements within if statements. i.e, we can place an if statement inside another if statement.


Start
i ← 10
i<20
Yes
No
i<15
Yes
No
PUT "i is smaller than 15"¶
PUT "i is greater than 15"¶
i<12
Yes
No
PUT "i is smaller than 12 too"¶
End

Syntax:
```
if (condition1):
   # Executes when condition1 is true
   if (condition2):
      # Executes when condition2 is true
   # if Block is end here
# if Block is end here
```
```
# python program to illustrate nested If statement
#!/usr/bin/python
i = 10
if (i<20):
    #  First if statement
    if (i< 15):
        print ("i is smaller than 15")
    # Nested - if statement
    # Will only be executed if statement above
    # it is true
else:
        print ("i is greater than 15")
    if (i< 12):
        print ("i is smaller than 12 too")
```
Output:

I. i is smaller than 15
II. i is smaller than 12 too

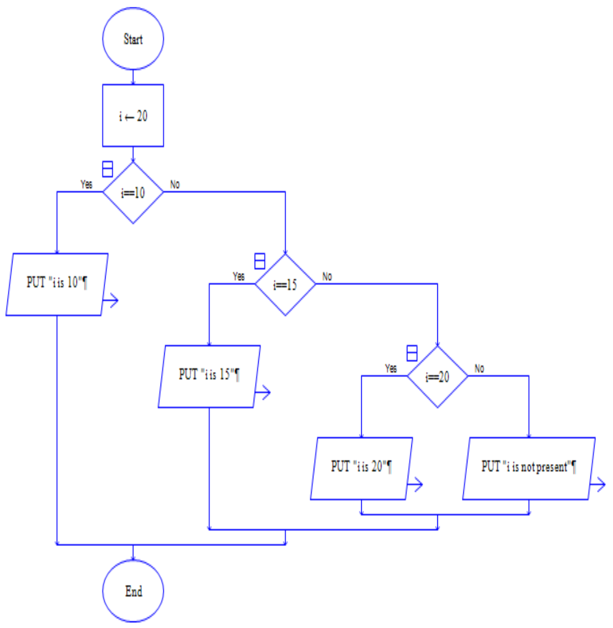
2.7.4 if-elif-else ladder: Here, a user can decide among multiple options. The if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed.

Syntax:-
```
if (condition):
    statement
elif (condition):
    statement

else:
    statement
```
Example:-
```
# Python program to illustrate if-elif-else ladder
#!/usr/bin/python
i = 20
if (i == 10):
    print ("i is 10")
elif (i == 15):
    print ("i is 15")
elif (i == 20):
    print ("i is 20")
else:
    print ("i is not present")
```
Output:
```
i is 20
```
2.8 Short Hand if statement

Whenever there is only a single statement to be executed inside if block then shorthand if can be used. The statement can be put on the same line as if statement.

Syntax:
```
if condition: statement
```
Example: # Python program to illustrate short hand if i = 10

if i< 15: print("i is less than 15")

Output:
```
i is less than 15
```


Short Hand if-else statement: This can be used to write the if-else statements in a single line where there is only one statement to be executed in both if and else block.

Syntax:
```
statement_when_True if condition else statement_when_False
```
Example: # Python program to illustrate short hand if-else i = 10

print(True) if i< 15 else print(False)

Output:
```
True
```

Unit 3 – Loops Control Statements and Functions

Module 1 – While Loop

Introduction

Looping is a powerful programming technique through which a group of statements is executed repeatedly, until certain specified condition is satisfied. Looping is also called repetitive or iterative control statements.

A loop in a program essentially consists of two parts, one is called the body of the loop and other is known as a control statement. The control statement performs a logical test whose result is either **True or False**. If the result of this logical test is true, then the statements contained in the body of the loop are executed. Otherwise, the loop is terminated.

There must be a proper logical test condition in the control statement, so that the statements are executed repeatedly and the loop terminates gracefully. If the logical test condition is carelessly designed, then there may be possibility of formation of an infinite loop which keeps executing the statements over and over again.

3.1 while loop

This is used to execute a set of statements repeatedly as long as the specified condition is true. It is an indefinite loop. In the while loop, the condition is tested before executing body of the statements. If the condition is True, only body of the block of statements will be executed otherwise if the condition is False, the control will jump to other statements which are outside of the while loop block.

Syntax:

```
        while(logexp):
                block of Statements
```

Where,

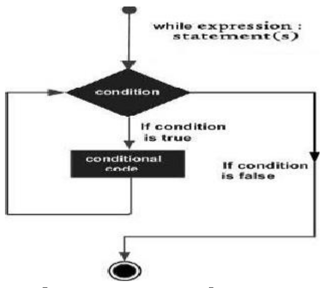
while → is a keyword

logexp →is a logical expression that results in either **True or Fasle**

Statement→ may be simple or a compound statement

Here, first of all the logical expressions is evaluated and if the result is true (non-zero) then the statement is repeatedly executed. If the result is false (zero), then control comes out of the loop and continues with the next executable statements.

Flowchart:



In while loop there are mainly it contains three parts, which are as follows

- Initialization: to set the initial value for the loop counter. The loop counter may be an increment counter or a decrement counter
- Decision: an appropriate test condition to determine whether the loop be executed or not
- Updation: incrementing or decrementing the counter value.

Example Programs:

Q1. Write a program to display your name up to 'n' times

```
#from the above line we take input from the user
i=1# assigning value to i variable
while(i<=n):# checking the condition
    print('RGUKT')#displaying output
    i=i+1#increasing i value here
print('good bye')
```

Output:

```
Enter the number
3
RGUKT
RGUKT
RGUKT
good bye
```

Q2. Write a program to display natural numbers up to n

```
n=int(input('Enter the number\n'))
i=1
print('Natural numbers are up to %d'%n)
while(i<=n):
    print(i)
    i=i+1
```

Output:

```
Enter the number
7
Natural numbers are up to 7
1
2
3
4
5
6
7
```

Q3.Write a program to display even numbers up to n

```
n=int(input('Enter the number: '))
i=1# assigning value 1 to i or intilization
print('Even numbers up to %d'%n)
while(i<=n):
    if(i%2==0):# here checking the condition whether the value is divisible or not
        print(i)# if it is divisible printing even numbers
    i=i+1# increment operator
```

Output:

```
Enter the number: 10
Even numbers up to 10
2
4
6
8
10
```

3.1.1 The infinite while loop: A loop becomes infinite loop if a condition never becomes FALSE. You must use caution when using while loops because of the possibility that this condition never resolves to a FALSE value. This results in a loop that never ends. Such a loop is called an infinite loop.
An infinite loop might be useful in client/server programming where the server needs to run continuously so that client programs can communicate with it as and when required.

Program to demonstrate infinite loop:

```
while True:
    print('Hi')
print('Good bye')
```

```
x=1
while(x==1):
    print('Hi')
print('Good bye')
```

Above two programs that never stop, that executes until your Keyboard Interrupts (ctrl+c). Otherwise, it would have gone on unendingly. Many 'Hi' output lines will display when you executes above two programs and that never display 'Good bye'

3.1.2 While loop with else clause/statement: Python allows an optional else clause at the end of a while loop. This is a unique feature of Python. If the else statement is used with a while loop, the else statement is executed when the condition becomes false. But, the while loop can be terminated with a break statement. In such cases, the else part is ignored. Hence, a while loop's else part runs if no break occurs and the condition is False

Syntax:

```
while <expr>:
    <statement(s)>
else:
    <additional_statement(s)>
```

Example Program:

Q1. Write a program to demonstrate while loop with else block.

```
i=1
while(i<=3):
    print('RGUKT')
    i=i+1
else:
    print('Good bye')
```

Output:

```
RGUKT
RGUKT
RGUKT
Good bye
```

Q2. Write a program to demonstrate else block with break statement

```
i=1
while(i<=6):
    if(i==4):
        break
    print(i)
    i=i+1
else:
    print('Never executes thie else block')
```

Output:

```
1
2
3
```

<u>Module 2 – For Loop</u>

3.2 for loop

Like the while loop, for loop works, to repeat statements until certain condition is True. The for loop in python is used to iterate over a sequence (list, tuple, string and range () function) or other iterable types. Iterating over a s sequence is called traversal. Here, by sequence we mean just an ordered collection of items. for loop is usually known as definite loop because the programmer knows exactly how many times the loop will repeat.

Syntax:

for counter_variable in sequence:
    block of statements

Here counter_variable is the variable name that takes the value of the item inside the sequence on each iteration. Loop continues until we reach the last item in the sequence. The body of for the loop is separated from the rest of the code using indentation.

Example programs:

Q1. Write a program to make sum of all numbers stored in a list.

```
#listof numbers
numbers=[5, 2, 7, 10, 14]
sum=0#assigning value to variable
for i in numbers:#iterate over the list
    sum=sum+i #making sum here
print('The sum is',sum)#displayiing output
```

Output :

```
The sum is 38
```

3.2.1 The range () function:

The range () function is a built-in function in python that is used to iterate over a sequence of numbers. The range() function contains three arguments/parameters like

range (start, end, step)

The range () generates a sequence of numbers starting with start(inclusive) and ending with one less than the number 'end'. The step argument is optional

By default, every number in the range is incremented by 1. Step can be either a positive or negative value but it cannot be equal to zero

Q2. Write a program to print first n natural numbers using a for loop

```
n=int(input('Enter the number: '))
for i in range(1,n,1):
    print(i,end=' ')
n=int(input('Enter the number: '))
for i in range(1,n):
    print(i,end=' ')
```

Output:

```
Enter the number: 10
1 2 3 4 5 6 7 8 9
```

Q3. If you want to display including the given input number you can write a program like

```
n=int(input('Enter the number: '))
for i in range(1,n+1):
    print(i,end=' ')
```

Output:

```
Enter the number: 10
1 2 3 4 5 6 7 8 9 10
```

3.2.2 for loop with else:

A for loop can have an optional else block. The else part is executed when the loop has exhausted iterating the list. But, when the break statement use in for loop, the else part is ignored.

Hence, a for loop's else part runs if no break occurs.

Q1. Program to demonstrate else block in for loop

```
for i in range(5):
    print(i)
else:
    print("No items left.")
```

Output:

```
0
1
2
3
4
No items left.
```

Here, the for loop prints numbers from the starting number to less than ending number. When the for loop exhausts, it executes the block of code in the else and prints No items left.

Q2. Program to demonstrate else block in for loop with break statement:

```
for i in range(5):
    if(i==3):
        break
    print(i)
else:
    print('This else block not executes')
```

Output:

```
0
1
2
```

3.3 Nested loops

Python programming language allows using one loop inside another loop which is called a nested loop. Although this feature will work for both while loop as well as for loop. The syntax for a nested while loop statement in Python programming language is as follows

Syntax:

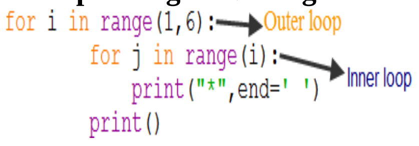
```
for iterating_var in sequence:
   for iterating_var in sequence:
      statements(s)
   statements(s)
```

```
while expression:
   while expression:
      statement(s)
   statement(s)
```

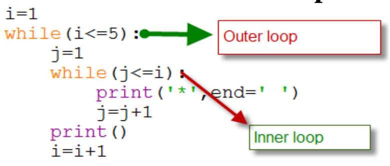
Note:

You can put any type of loop inside of any other type of loop. For example, a for loop can be inside a while loop or vice versa.

Example Program: Program to demonstrate nested for loop **Nested while loop:**



```
for i in range(1,6):
    for j in range(i):
        print("*",end=' ')
    print()
```



```
i=1
while(i<=5):
    j=1
    while(j<=i):
        print('*',end=' ')
        j=j+1
    print()
    i=i+1
```

Output:

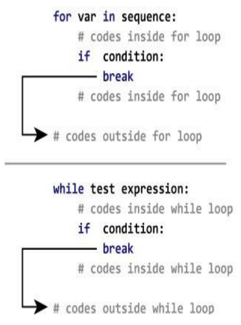
```
*
* *
* * *
* * * *
* * * * *
```

3.4 Control statements

Loops iterate over a block of code until the test expression is false, but sometimes we wish to terminate the current iteration or even the whole loop without checking test expression. These can be done by loop control statements. Loop control statements change execution from its normal sequence.

There are three different kinds of control statements which are as follows

1. break
2. continue
3. pass



```
for var in sequence:
    # codes inside for loop
    if condition:
        break
    # codes inside for loop

# codes outside for loop
```

```
while test expression:
    # codes inside while loop
    if condition:
        break
    # codes inside while loop

# codes outside while loop
```

3.4.1 Break Statement: The break statement in Python terminates the current loop and resumes execution at the next statement (out of the loop). This break statement is used in both for and while loops.

Syntax:

break

Flowchart
Example Programs:
Write a program to demonstrate break statement by using for loop and while loop
for loop

```python
for i in range(1, 11):
    if(i==5):
        break
    print(i)
print("Broke out of loop at i=",i)
```

when i equals 5 the loops break

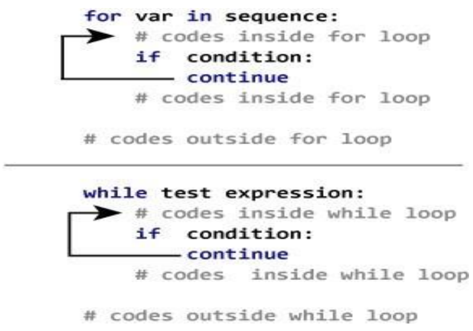
while loop

```python
i=1
while(i<11):
    if(i==5):
        break
    print(i)
    i=i+1
print("Broke out of loop at i=",i)
```

Output:

```
1
2
3
4
Broke out of loop at i= 5
```

3.4.2 Continue Statement: The continue statement is used to skip the rest of the code inside a loop for the current iteration only. Loop does not terminate but continues on with the next iteration. Continue statement is opposite to break statement, instead of terminating the loop; it forces to execute the next iteration of the loop.
Syntax: continue
Flowchart:



```
for var in sequence:
    # codes inside for loop
    if   condition:
         continue
    # codes inside for loop

# codes outside for loop
```

```
while test expression:
    # codes inside while loop
    if   condition:
         continue
    # codes   inside while loop

# codes outside while loop
```

Example Programs:
Write a program to demonstrate continue statement by using for loop

```python
for i in range(1, 10):
    if(i==5):
        continue
    print(i)
```

Output:
```
1  2  3  4  6  7  8  9
```

In output, 5 integer value is skipped based on if condition and continues flow of the loop until the condition satisfied.
3.4.3 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. It is a null statement. However, nothing happens when the pass statement is executed. It results in no operation (NOP). Pass statement can also be used for writing empty loops, functions and classes.
Syntax: pass
Example Program:
Program to demonstrate pass statement using for loop

```python
for i in range(1, 10):
    if(i==5):
        pass
    print(i, end=' ')
```

Output:
```
1  2  3  4  5  6  7  8  9
```

Solved Questions

1) Write a program to display your name up to n times.

Using while loop

```python
n=int(input("Enter number of Times:"))
i=1
while(i<=n):
    print("RGUKT-RK Valley")
    i=i+1
```

Using for loop

```python
n=int(input("Enter number of Times:"))
for i in range(n):
    print("RGUKT-RK Valley")
```

Output:

```
Enter number of Times:4
RGUKT-RK Valley
RGUKT-RK Valley
RGUKT-RK Valley
RGUKT-RK Valley
```

2) Write a program to display sum of odd numbers up to n

#Using while loop

```
n=int(input("Enter n Value:"))
i=1
s=0
while(i<=n):
    if(i%2!=0):
        s=s+1
    i=i+1
print("Sum of Odd Values", s)
```

#Using for loop

```
n=int(input("Enter n Value:"))
s=0
for i in range(1,n+1):
    if(i%2!=0):
        s=s+1
print("Sum of Odd Values", s)
```

3) Write a program to display numbers of factors to the given number

# Using while loop

```
n=int(input("Enter n Value:"))
i=1
c=0
while(i<=n):
    if(n%i==0):
        c=c+1
    i=i+1
print("count of factors is ", c)
```

# Using for loop

```
n=int(input("Enter n Value:"))
c=0
for i in range(1,n+1):
    if(n%i==0):
        c=c+1
print("count of factors is ", c)
```

4) Write a program to find given number is prime number or not

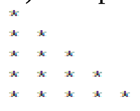
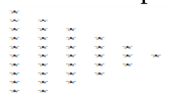
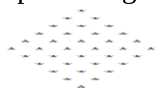
```
n=int(input("Enter n Value:"))
c=0
for i in range(1,n+1):
    if(n%i==0):
        c=c+1
if(c==2):
    print("given number is prime number")
else:
    print("given number is not prime number")
```

Descriptive Questions:

1) Explain about while loop and for loop?
2) Explain about loop control statements?
3) Explain about range function with different arguments?

Unsolved Questions:

1) Write a program to print all-natural numbers in reverse (from n to 1).
2) Write a program to print all even numbers up to n
3) Write a program to print sum of all odd numbers between two intravel given.
4) Write a program to print table of any number.
5) Write a program to enter any number and calculate sum of its digits.
6) Write a program to print all Prime numbers between 1 to n
7) Write a program to enter any number and display perfect numbers between 1 to n
8) Write a program to enter any number and find its first and last digit.
9) Write a program to display given number is a number palindrome or not
10) Write a program to print all alphabets from a to z.
11) Write a program to enter any number and check whether it is Armstrong number or not.
12) Write a program to print all Strong numbers between 1 to n.
13) Write a program to print Fibonacci series up to n terms.
14) Star pattern programs - Write a program to print the given star patterns.

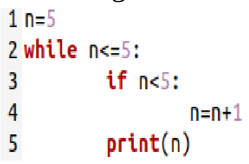




15) Write a Python program to construct the following patterns, using a nested loop number

```
1                                    P
2 3                                  PY
4 5 6                                PYT
7 8 9 10                             PYTH
11 12 13 14 15                       PYTHO
                                     PYTHON
```

Multiple Choice Questions:

1) A while loop in Python is used for what type of iteration?
   a) Indefinite b) Discriminate c) Definite d) Indeterminate
2) When does the else statement written after loop executes?
   a) When break statement is executed in the loop b) When loop condition becomes false
   c) Else statement is always executed d) None of the above
3) What do we put at the last of for/while loop?
   a) Semicolon b) Colon c) Comma d) None of the above
4) Which of the following loop is work on the particular range in python?
   a) For loop b) While loop c) Do-while loop d) Recursion
5) How many times it will print the statement?, for i in range(100): print(i)
   a) 101 b) 99 c) 100 d) 0
6) What is the result of executing the following code?



```
n=5
while n<=5:
        if n<5:
                n=n+1
        print(n)
```

   a) The program will loop indefinitely b) The value of number will be printed exactly 1 time
   b) The while loop will never get executed d) The value of number will be printed exactly 5 times
7) Which of the following sequences would be generated by given line of the code?
   a) 5 4 3 2 1 0 -1 b) 5 4 3 2 1 0 c)5 3 1 d) Error
8) Which of the following is a valid for loop in Python?
   a) for (i=0;i<=n; i++) b) for i in range(5) c) for i in range (1, 5): d) for i in range(0,5,1)
9) Which statement is used to terminate the execution of the nearest enclosing loop in which it appears?
   a) pass b) break c) continue d) jump
10) Which statement indicates a NOP?
   a) Pass b) break c) continue d) jump

Module 3- Built-In Functions and User defined functions

A function is a named sequence of statement(s) that performs a computation. It contains line of code(s) that are executed sequentially from top to bottom by Python interpreter. They are the most important building blocks for any software in Python. Functions can be categorized as belonging to

1) Modules
2) Built in Functions
3) User Defined Functions

Module:

A module is a file containing Python definitions (i.e. functions) and statements. Standard library of Python is extended as module(s) to a programmer. Definitions from the module can be used within the code of a program. To use these modules in the program, a programmer needs to import the module. Once you import a module, you can reference (use), any of its functions or variables in your code. There are many ways to import a module in your program, the one‟s which you should know are:

1) import
2) from

Import

It is simplest and most common way to use modules in our code.

Syntax: **import modulename1 [,modulename2,---------]**

Ex: >>> import math

On execution of this statement, Python will

(i) search for the file „**math.py**‟.

(ii) Create space where modules definition & variable will be created,

(iii) then execute the statements in the module.

Now the definitions of the module will become part of the code in which the module was imported.

To use/ access/invoke a function, you will specify the module name and name of the function- separated by dot (.). This format is also known as *dot notation*

>>> *value= math.sqrt (25) # dot notation*

The example uses sqrt( ) function of module **math** to calculate square root of the value provided in parenthesis, and returns the result which is inserted in the ***value***. The expression (variable) written in parenthesis is known as argument (actual argument). It is common to say that the function takes arguments and return the result. This statement invokes the sqrt ( ) function.

From Statement

It is used to get a specific function in the code instead of the complete module file. If we know beforehand which function(s), we will be needing, then we may use **from**. For modules having large no. of functions, it is recommended to use **from** instead of import.

Syntax: **>>> from modulename import functionname [, functionname…..]**

Ex >>> *from math import sqrt*
value = sqrt (25)

Here, we are importing sqrt function only, instead of the complete math module. Now sqrt( ) function will be directly referenced to. These two statements are equivalent to previous example.

from modulename import *
will import everything from the file.

Note: You normally put all import statement(s) at the beginning of the Python file but technically they can be anywhere in program.

Lets explore some more functions available in **math module:**

ceil(x):

It returns the smallest integer not less than x, where *x is a numericexpression.*

>> *math.ceil(-45.17)*
-45.0
>> *math.ceil(100.12)*
101.0
>> *math.ceil(100.72)*
101.0

floor(x) :

It returns the largest integer not greater than x, where *x is a numeric expression.*

>> *math.floor(-45.17)*
-46.0
>> *math.floor(100.12)*
100.0
>> *math.floor(100.72)*
100.0

fabs(x):

It returns the absolute value of x, *where x is a numeric value.*

>> *math.fabs(-45.17)*
45.17
>> *math.fabs(100.12)*
100.12
>> *math.fabs(100.72)*
100.72

exp(x) :

It returns exponential of x:ex, where x is a numeric expression.

>> *math.exp(-45.17)*
2.41500621326e-20
>> *math.exp(100.12)*
3.03084361407e+43
>> *math.exp(100.72)*
5.52255713025e+43

log(x):

It returns natural logarithm of x, for x > 0, where *x is a numeric expression.*

>> *math.log(100.12)*
4.60636946656
>> *math.log(100.72)*
4.61234438974

log10(x):
It returns base-10 logarithm of x for x > 0, where *x is a numeric expression.*
>> *math.log10(100.12)*
2.00052084094
>> *math.log10(100.72)*
2.0031157171
pow(x, y):
It returns the value of xy, *where x and* **y** *are numeric expressions.*
>> *math.pow(100, 2)*
10000.0
>> *math.pow(100, -2)*
0.0001
>> *math.pow(2, 4)*
16.0
>> *math.pow(3, 0)*
1.0
sqrt (x):
It returns the square root of x for x > 0, where x is a numeric expression.
>> *math.sqrt(100)*
10.0
>> *math.sqrt(7)*
2.64575131106
cos (x):
It returns the cosine of x in radians, *where x is a numeric expression*
>> *math.cos(3)*
-0.9899924966
>> *math.cos(-3)*
-0.9899924966
>> *math.cos(math.pi)*
-1.0
sin (x):
It returns the sine of x, in radians, *where x must be a numeric value.*
>> *math.sin(3)*
0.14112000806
>> *math.sin(-3)*
-0.14112000806
>> *math.sin(0)*
0.0
tan (x):
It returns the tangent of x in radians, *where x must be a numeric value.*
>> *math.tan(3)*
-0.142546543074
>> *math.tan(-3)*
0.142546543074
>> *math.tan(0)*
0.0
degrees (x):
It converts angle x from radians to degrees, *where x must be a numeric value.*
>> *math.degrees(3)*
171.887338539
>> *math.degrees(-3)*
-171.887338539
>> *math.degrees(0)*
0.0
radians(x):
It converts angle x from degrees to radians, *where x must be a numeric value.*
>> *math.radians(3)*
0.0523598775598
>> *math.radians(-3)*
-0.0523598775598

Some functions from **random module** are:

random ( ):

It returns a random float x, such that 0 ≤ x<1

```
>>>random.random ( )
0.281954791393
>>>random.random ( )
0.309090465205
```

randint (a, b):

It returns a int x between a & b such that a ≤ x ≤ b

```
>>>  random.randint (1,10)
5
>>> random.randint (-2,20)
-1
```

uniform (a,b):

It returns a floating point number x, such that a <= x < b

```
>>> random.uniform (5,10)
5.52615217015
```

randrange ([start,] stop [,step]):

It returns a random item from the given range

```
>>> random.randrange(100,1000,3)
150
```

Built in Function:

Built in functions are the function(s) that are built into Python and can be accessed by a Programmer. These are always available and for using them, we don"t have to import any module (file). Python has a small set of built-in functions as most of the functions have been partitioned to modules. This was done to keep core language precise.

abs (x): It returns distance between x and zero, where *x is a numeric expression.*

```
>>> abs(-45)
45
>>> abs(119L)
119
```

max(x, y, z,.....): It returns the largest of its arguments: where x, y and z are numeric variable/expression.

```
>>> max(80, 100, 1000)
1000
>>> max(-80, -20, -10)
-10
```

min(x, y, z, ...): It returns the smallest of its arguments; where x, y, and z are numeric variable/expression.

```
>>> min(80, 100, 1000)
80
>>> min(-80, -20, -10)
-80
```

cmp(x, y): It returns the sign of the difference of two numbers: -1 if x < y

0 if x == y

1 if x > y

where x and y are numeric variable/expression.

```
>>> cmp(80, 100)  -1
>>> cmp(180, 100)  1
```

divmod (x,y): Returns both quotient and remainder by division through a tuple, when x is divided by y; where x & y are variable/expression.

```
>>> divmod (14,5) (2,4)
>>> divmod (2.7, 1.5)
(1.0, 1.20000)
```

len (s): Return the length (the number of items) of an object. The argument may be a sequence (string, tuple or list) or a mapping (dictionary).

```
>>> a= [1,2,3]
>>> len (a)
3
>>> b= "Hello"
>>> len (b)
5
```

range (*start*, *stop*[, *step*]): This is a versatile function to create lists containing arithmetic progressions. It is most often used in for loops. The arguments must be plain integers.

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(1, 11)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> range(0, 30, 5)
[0, 5, 10, 15, 20, 25]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(0, -10, -1)
[0, -1, -2, -3, -4, -5, -6, -7, -8,-9]
>>> range(0)
[]
>>> range(1, 0)
[]
```

round(x [, n]): It returns float x rounded to n digits from the decimal point, *where x and n are numeric expressions.* If n is not provided then x is rounded to 0 decimal digits.

```
>>> round(80.23456, 2)
80.23
>>> round(-100.000056, 3)
-100.0
>>> round (80.23456)
80.0
```

Composition

Composition is an art of combining simple function(s) to build more complicated ones, i.e., result of one function is used as the input to another.

Example

Suppose we have two functions fn1 & fn2, such that

a= fn2 (x)

b= fn1 (a)

then call to the two functions can be combined as

b= fn1 (fn2 (x))

Similarly, we can have statement composed of more than two functions. In that result of one function is passed as argument to next and result of the last one is the final result.

Ex

```
>>> math.exp (math.log (a+1))
```

Ex

```
>>> degrees=270
>>> math.sin (degrees/360.0 *2*math.pi)
```

Composition is used to package the code into modules, which may be used in many different unrelated places and situations. Also it is easy to maintain the code.

User Defined Functions

So far we have only seen the functions which come with Python either in some file (module) or in interpreter itself (built in), but it is also possible for programmer to write their own function(s). These functions can then be combined to form a module which can then be used in other programs by importing them. To define a function keyword **def** is used. After the keyword comes an identifier i.e. name of the function, followed by parenthesized list of parameters and the colon which ends up the line. Next follows the block of statement(s) that are the part of function.

Before learning about Function header & its body, let's explore block of statements, which become part of function body

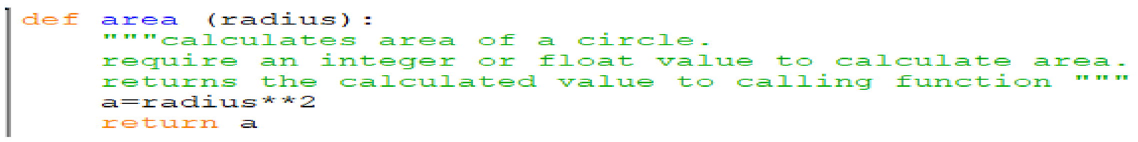
Block of statements

A block is one or more lines of code, grouped together so that they are treated as one big sequence of statements while executing. In Python, statements in a block are written with indentation. Usually, a block begins when a line is indented (by four spaces) and all the statements of the block should be at same indent level. A block within block begins when its first statement is indented by four space, i.e., in total eight spaces. To end a block, write the next statement with the same indentation before the block started

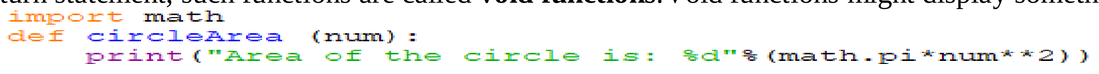
Syntax:

```
def fun_name (PARAMETER1, PARAMETER2, …..): #Square brackets include
statement(s) #optional part of statement
```

Let's write a function to greet the world:
def sayHello (): # Line No. 1
print ("Hello World!") # Line No.2
The first line of function definition, i.e., Line No. 1 is called **header** and the rest, i.e.
Line No. 2 in our example is known as **body**. Name of the function is ***sayHello***, and empty parenthesis indicates no parameters. Body of the function contains one Python statement, which displays a string constant on screen.
Function Header
It begins with the keyword **def** and ends with colon and contains the function identification details. As it ends with colon, we can say that what follows next is, block of statements.
Function Body
Consisting of sequence of indented (4 space) Python statement(s), to perform a task. Defining a function will create a variable with same name, but does not generate any result. The body of the function gets executed only when the function is called/invoked. Function **call** contains the name of the function (being executed) followed by the list of values (i.e. arguments) in parenthesis
Ex:
def sayHello (): # Function Definition
print ("Hello World!") # Block of Statements
sayHello () # Call/invoke statement of this function
O/P:
Hello World!
Apart from this, you have already seen many examples of invoking of functions in Modules & Built-in Functions.
Let‟s know more about **def.** It is an executable statement. At the time of execution a function is created and a name (name of the function) is assigned to it. Because it is a statement, **def** can appear anywhere in the program. It can even be nested
if condition:
def fun ( ): # function definition one way
else:
def fun ( ): # function definition other way
fun ( ) # calls the function selected.
This way we can provide an alternative definition to the function. This is possible because **def** is evaluated when it is reached and executed.
Let's explore Function body
The first statement of the function body can optionally be a string constant, **docstring, enclosed** in triple quotes. It contains the essential information that someone might need about the function, such as
- What function does (**without How it does**) i.e. summary of its purpose
- Type of parameters it takes
- Effect of parameter on behavior of functions, etc.

DocString is an important tool to document the program better, and makes it easier to understand. We can actually access docstring of a function using ___doc___(function name). Also, when you used help (), then Python will provide you with docstring of that function on screen. So it is strongly recommended to use docstring … when you write functions


```
def area (radius):
    """calculates area of a circle.
    require an integer or float value to calculate area.
    returns the calculated value to calling function """
    a=radius**2
    return a
```

Function is pretty simple and its objective is pretty much clear from the docString added to the body.
The last statement of the function, i.e. return statement returns a value from the function. Return statement may contain a constant/literal, variable, expression or function, if return is used without anything, it will return **None.** In our example value of a variable **area** is returned.
Instead of writing two statements in the function, i.e.
a = radius **2
return a
We could have written **return radius **2**
Here the function will first calculate and then return the value of the expression. It is possible that a function might not return a value, as sayHello( ) was not returning a value. sayHello( ) prints a message on screen and does not contain a return statement, such functions are called **void functions**.Void functions might display something on the screen or


```
import math
def circleArea (num):
    print("Area of the circle is: %d"%(math.pi*num**2))
```

have some other effect, but they don‟t have a return value. If you try to assign the result of such function to a variable, you get a special value called **None**.

Parameters and Arguments

Parameters are the value(s) provided in the parenthesis when we write function header. These are the values required by function to work. Let's understand this with the help of function written for calculating area of circle.

radius is a parameter to function area. If there is more than one value required by the function to work on, then, all of them will be listed in parameter list separated by comma.

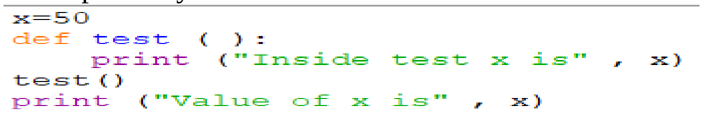
Arguments are the value(s) provided in function call/invoke statement. List of arguments should be supplied in same way as parameters are listed. Bounding of parameters to arguments is done 1:1, and so there should be same number and type of arguments as mentioned in parameter list.

Scope of Variables

Scope of variable refers to the part of the program, where it is visible, i.e., area where you can refer (use) it. We can say that scope holds the current set of variables and their values. We will study two types of scope of variables- global scope or local scope.

Global Scope

A variable, with global scope can be used anywhere in the program. It can be created by defining a variable outside the scope of any function/block.

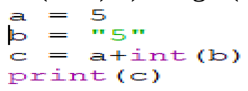


```
x=50
def test ( ):
    print ("Inside test x is" , x)
test()
print ("Value of x is" , x)
```

Multiple Choice Questions

1) What is the functionality of input ()?
   a) Take integer values from keyboard   b) Take character from the keyboard
   c) Take both integer and character from Keyboard   d) All of the above

2) Which one of the following is correct function to convert string to integer? If num = input("Enter a number: ")
   a) int(num) b) integer(num) c) raw_input(num) d) None of the mentioned

3) 

```
a = 5
b = "5"
c = a+int(b)
print(c)
```

What is output for above program?
   a) 55 b) Error c) 5, 5 d) 10

4) Select the correct function definition in the following statements?
   a) def funct() b) def funct c) funct() d) definition funct()

5) Select the correct calling function statement from the following
   def fun():
   print ("Hello RGUKT")
   a) fun() b) fun c) fun(): d) None of the mentioned

6. In the following example, purpose of Line3
   def fun(): #Line1
   print("Hello RGUKT") #Line2
   fun() #Line3
   a) function definition b) function calling c) initializing d) all of the above

7) What is the output for following example?
   def add(a,b):
   print(a+b)
   add(5,10)
   a) 15 b) 2 c) 10 d) 5

8) What would be the output for the following example
   def function1():   print("Nuzvid")
   def function2():   print("Srikakulam")
   function2()
   a) Nuzvid b) Srikakulam c) Error d) None of the mentioned

9) Functions which are created by the user are called as
   a) Builtin Functions b) Predefined Functions c) Userdefined Functions d) None

10) What would be the output for following example?
   def student(): print("Student")
   def teacher(): print("Teacher")
   teacher()
   student()
   a) Student b) Teacher c) Error d) First defined function should be called first Teacher Student

11) We can pass arguments to function.
   a) True b) False

Unit 4 – List and Tuples

Module 1 - List

4.1 Definition

Python offers a range of compound data types often referred to as sequences. List is one of the most frequently used and very versatile data types used in Python. Like a string, a list is a sequence of values. In a string, the values are characters; in a list, they can be any type. The values in list are called elements or sometimes items.
There are several ways to create a new list; the simplest is to enclose the elements in square brackets and are separated by commas. It can have any number of items and they may be of different types (integer, float, string etc.).
[10, 20, 30, 40]
['crunchy frog', 'ram bladder', 'lark vomit']
The first example is a list of four integers. The second is a list of three strings. 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:
['spam', 2.0, 5, [10, 20]]
A list can also have another list as an item. This is called a nested list. It can have any number of items and they may be of different types (integer, float, string etc.).

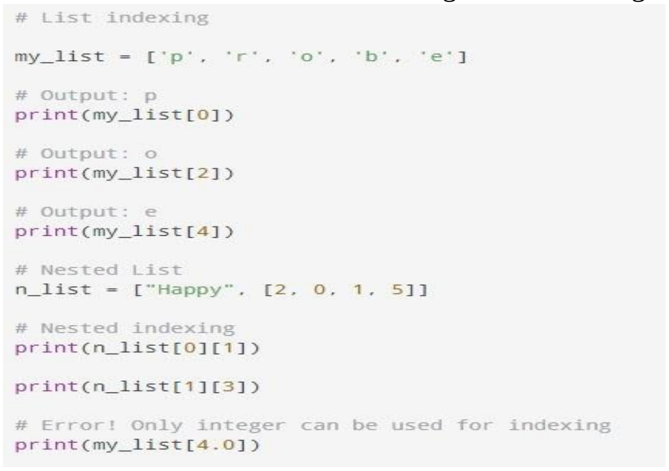
4.2 How to create a list?

In Python programming, a list is created by placing all the items (elements) inside square brackets [], separated by commas.
# empty list
my_list = []
# list of integers
my_list = [1, 2, 3]
# list with mixed data types
my_list = [1, "Hello", 3.4]

4.3 How to access elements from a list?

There are various ways in which we can access the elements of a list.
5.3.1 List Index: We can use the index operator [] to access an item in a list. In Python, indices start at 0. So, a list having 5 elements will have an index from 0 to 4. Trying to access indexes other than these will raise an IndexError. The index must be an integer. We can't use float or other types, this will result in TypeError.

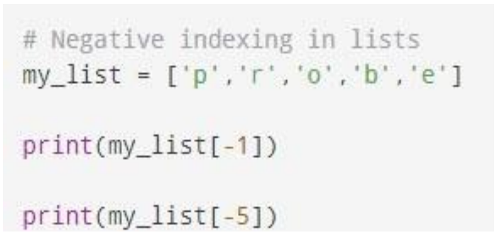
- Nested lists are accessed using nested indexing.



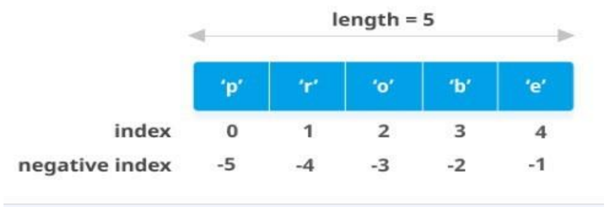
```
# List indexing

my_list = ['p', 'r', 'o', 'b', 'e']

# Output: p
print(my_list[0])

# Output: o
print(my_list[2])

# Output: e
print(my_list[4])

# Nested List
n_list = ["Happy", [2, 0, 1, 5]]

# Nested indexing
print(n_list[0][1])

print(n_list[1][3])

# Error! Only integer can be used for indexing
print(my_list[4.0])
```

4.3.2 Negative indexing

Python allows negative indexing for its sequences. The index of -1 refers to the last item, -2 to the second last item and so on.



```
# Negative indexing in lists
my_list = ['p','r','o','b','e']

print(my_list[-1])

print(my_list[-5])
```



List indexing in Python

Output:

```
e
p
```

4.4 Change or add elements to a list

The syntax for accessing the elements of a list is the same as for accessing the characters of a string the bracket operator. The expression inside the brackets specifies the index. Remember that the indices start at 0:
Unlike strings, lists are mutable because you can change the order of items in a list or reassign an item in a list. When the bracket operator appears on the left side of an assignment, it identifies the element of the list that will be assigned.

```
>>> numbers = [17, 123]
>>> numbers [1] = 5
>>> print numbers
[17, 5]
```

The one-eth element of numbers, which used to be 123, is now 5.
You can think of a list as a relationship between indices and elements. This relationship is called a mapping; each index "maps to" one of the elements.
List indices work the same way as string indices:

4.5 Traversing a list

The most common way to traverse the elements of a list is with a *for* loop. The syntax is the same as for strings:

```
cheeses = ['Cheddar', 'Edam', 'Gouda']
for cheese in cheeses:
        print (cheese)
```

This works well if you only need to read the elements of the list. But if you want to write or update the elements, you need the indices. A common way to do that is to combine the functions *range* and *len*:

```
numbers = [17, 123]
for i in range(len(numbers)):
numbers[i] = numbers[i] * 2
```

This loop traverses the list and updates each element.*len* returns the number of elements in the list. *range* returns a list of indices from 0 to *n*−1, where *n* is the length of the list. Each time through the loop, *i* gets the index of the next element. The assignment statement in the body uses *i* to read the old value of the element and to assign the new value.
A *for* loop over an empty list never executes the body:

```
for x in empty:
print ('This never happens.')
```

Although a list can contain another list, the nested list still counts as a single element. The length of this list is four:
['spam', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]

4.6 List operations

The + operator concatenates lists:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print (c)
[1, 2, 3, 4, 5, 6]
```

Similarly, the * operator repeats a list a given number of times:

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

The first example repeats [0] four times. The second example repeats the list [1, 2, 3] three times.
List slices: The slice operator also works on lists

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
['b', 'c']
>>> t[:4]
['a', 'b', 'c', 'd']
>>> t[3:]
['d', 'e', 'f']
```

If you omit the first index, the slice starts at the beginning. If you omit the second, the slice goes to the end. So if you omit both, the slice is a copy of the whole list.
Since lists are mutable, it is often useful to make a copy before performing operations that fold, spindle, or mutilate lists. A slice operator on the left side of an assignment can update multiple elements:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>>t[1:3] = ['x', 'y']
>>> print (t)
['a', 'x', 'y', 'd', 'e', 'f']
```

Syntax: List[start:end:step]
start –starting point of the index value end
stop –ending point of the index value
step- increment of the index values

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3:1]
['b', 'c']
```

Module 2 - List methods

4.7 - Adding Elements into a list

4.7.1 list.append()

Python provides methods that operate on lists. For example, *append* adds a new element to the end of a list

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> print (t)
['a', 'b', 'c', 'd']
```

4.7.2 list.extend()

extend takes a list as an argument and appends all of the elements

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> print (t1)
['a', 'b', 'c', 'd', 'e']
```

This example leaves t2 unmodified.
sort arranges the elements of the list from low to high:

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>>t.sort()
>>> print(t)
['a', 'b', 'c', 'd', 'e']
```

4.7.3 list.insert(*i*, *x*)

Insert an item at a given position. The first argument is the index of the element before which to insert,
so a.insert(0, x) inserts at the front of the list, and a.insert(len(a), x) is equivalent to a.append(x).

Example 1: Inserting an Element to the List

```
# vowel list
vowel = ['a', 'e', 'i', 'u']
# 'o' is inserted at index 3
# the position of 'o' will be 4th
vowel.insert(3, 'o')
print('Updated List:', vowel)
```

Deleting elements

4.7.4 list.remove(*x*)

Remove the first item from the list whose value is equal to *x*. It raises a ValueError if there is no such item.

4.7.5 list.clear()

Remove all items from the list. Equivalent to del a[:].

Example 1: Working of clear() method

```
# Defining a list
list = [{1, 2}, ('a'), ['1.1', '2.2']]
# clearing the list
list.clear()
print('List:', list)
```

4.7.6 list.pop([*i*])

Remove the item at the given position in the list, and return it. If no index is specified, a.pop() removes and returns the last item in the list. (The square brackets around the *I* in the method signature denote that the parameter is optional, not that you should type square brackets at that position. You will see this notation frequently in the Python Library Reference.)

```
>>> t = [‘a’, ‘b’, ‘c’]
>>> x = t.pop(1)
>>> print (t)
[‘a’, ‘c’]
>>> print (x)
b
```

If you don’t need the removed value, you can use the del operator:

```
>>> t = [‘a’, ‘b’, ‘c’]
>>> del t[1]
>>> print (t)
[‘a’, ‘c’]
```

If you know the element you want to remove (but not the index), you can use remove:

```
>>> t = [‘a’, ‘b’, ‘c’]
>>>t.remove(‘b’)
>>> print (t)
[‘a’, ‘c’]
```

The return value from remove is None.
To remove more than one element, you can use del with a slice index:

```
>>> t = [‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’]
>>> del t[1:5]
>>> print (t)
[‘a’, ‘f’]
```

As usual, the slice selects all the elements up to, but not including, the second index.

4.7.6 list.index(*x*[, *start*[, *end*]])

Return zero-based index in the list of the first item whose value is equal to *x*. Raises a ValueError if there is no such item. The optional arguments *start* and *end* are interpreted as in the slice notation and are used to limit the search to a particular subsequence of the list. The returned index is computed relative to the beginning of the full sequence rather than the *start* argument.

Example:

```
# vowels list
vowels = ['a', 'e', 'i', 'o', 'i', 'u']
# index of 'e' in vowels
index = vowels.index('e')
print('The index of e:', index)
# element 'i' is searched
# index of the first 'i' is returned
index = vowels.index('i')
print('The index of i:', index)
```

Example: Working of index() With Start and End Parameters

```
# alphabets list
alphabets = ['a', 'e', 'i', 'o', 'g', 'l', 'i', 'u']
# index of 'i' in alphabets
index = alphabets.index('e')  # 2
print('The index of e:', index)
# 'i' after the 4th index is searched
index = alphabets.index('i', 4)  # 6
print('The index of i:', index)
# 'i' between 3rd and 5th index is searched
index = alphabets.index('i', 3, 5)  # Error!
print('The index of i:', index)
```

4.7.7 list.count(*x*)

Return the number of times *x* appears in the list.

Example:

```
# vowels list
vowels = ['a', 'e', 'i', 'o', 'i', 'u']
# count element 'i'
count = vowels.count('i')
# print count
print('The count of i is:', count)
# count element 'p'
count = vowels.count('p')
# print count
print('The count of p is:', count)
```

4.7.8 list.sort(*key=None*, *reverse=False*)
Sort the items of the list in place (the arguments can be used for sort customization, see sorted() for their explanation). Most list methods are void; they modify the list and return None. If you accidentally write list= list.sort(), you will be disappointed with the result.
Example: Sort a given list

```
# vowels list
vowels = ['e', 'a', 'u', 'o', 'i']
# sort the vowels
vowels.sort()
# print vowels
print('Sorted list:', vowels)
```

Example : Sort the list in Descending order

```
# vowels list
vowels = ['e', 'a', 'u', 'o', 'i']
# sort the vowels
vowels.sort(reverse=True)
# print vowels
print('Sorted list (in Descending):', vowels)
```

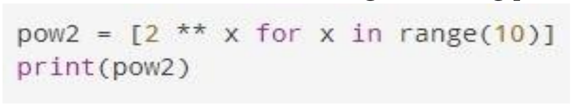
4.7.9 list.reverse()
Reverse the elements of the list in place.

```
# Operating System List
systems = ['Windows', 'macOS', 'Linux']
print('Original List:', systems)
# List Reverse
systems.reverse()
# updated list
print('Updated List:', systems)
```

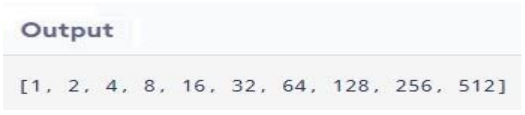
4.7.10 list.copy()
Return a shallow copy of the list. Equivalent to a[:]

```
# mixed list
my_list = ['cat', 0, 6.7]
# copying a list
new_list = my_list.copy()
print('Copied List:', new_list)
```

4.8 List Comprehension
List comprehension is an elegant and concise way to create a new list from an existing list in Python. A list comprehension consists of an expression followed by for statement inside square brackets. Here is an example to make a list with each item being increasing power of 2.



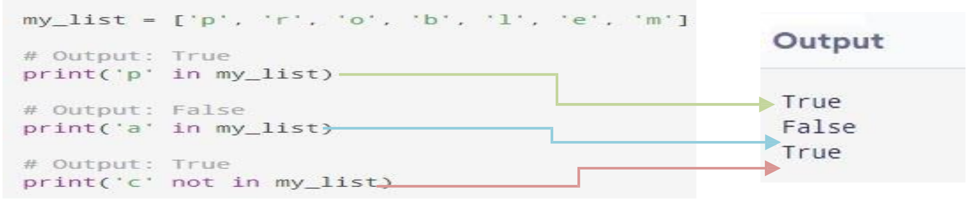
```
pow2 = [2 ** x for x in range(10)]
print(pow2)
```



Output

```
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

List Membership Test:
We can test if an item exists in a list or not, using the keyword *in*.



```
my_list = ['p', 'r', 'o', 'b', 'l', 'e', 'm']

# Output: True
print('p' in my_list)

# Output: False
print('a' in my_list)

# Output: True
print('c' not in my_list)
```

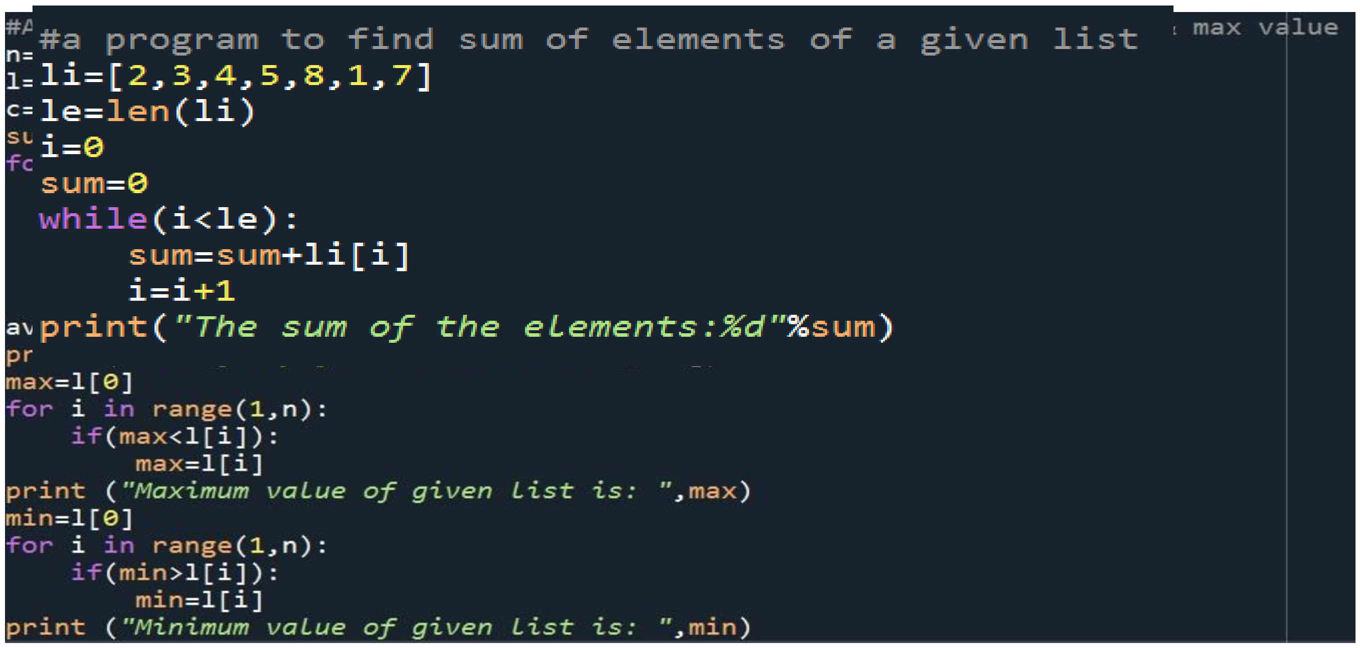
Output

```
True
False
True
```

4.9 Built-in functions
There are a number of built-in functions that can be used on lists that allow you toquickly look through a list without writing your own loops:

```
>>>nums = [3, 41, 12, 9, 74, 15]
>>> print len(nums)
6
>>> print max(nums)
74
>>> print min(nums) 3
>>> print sum(nums) 154
>>> print sum(nums)/len(nums) 25
```

The sum() function only works when the list elements are numbers. The other functions (max(), len(), etc.) work with lists of strings and other types that can be comparable.



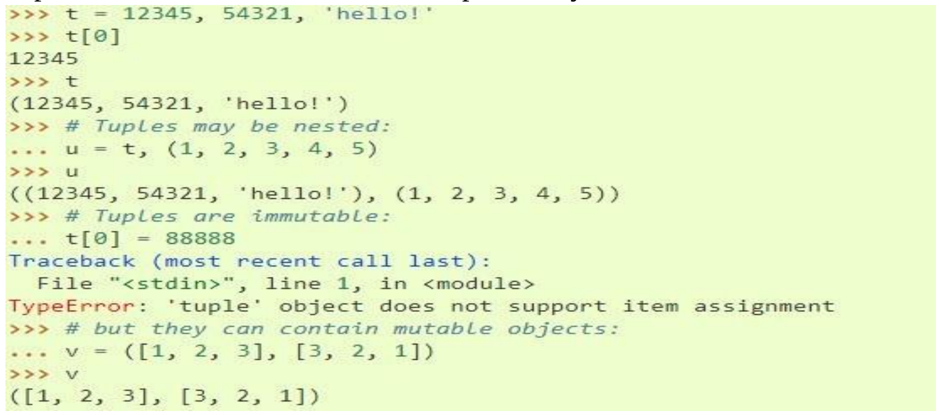
```
#a program to find sum of elements of a given list
li=[2,3,4,5,8,1,7]
le=len(li)
i=0
sum=0
while(i<le):
      sum=sum+li[i]
      i=i+1
print("The sum of the elements:%d"%sum)
```

```
max=l[0]
for i in range(1,n):
    if(max<l[i]):
        max=l[i]
print ("Maximum value of given list is: ",max)
min=l[0]
for i in range(1,n):
    if(min>l[i]):
        min=l[i]
print ("Minimum value of given list is: ",min)
```

Module 3 – Tuples

We saw that lists and strings have many common properties, such as indexing and slicing operations. They are two examples of *sequence* data types (Sequence Types — list, tuple, range). Since Python is an evolving language, other sequence data types may be added. There is also another standard sequence data type: the *tuple*.

A tuple consists of a number of values separated by commas, for instance:

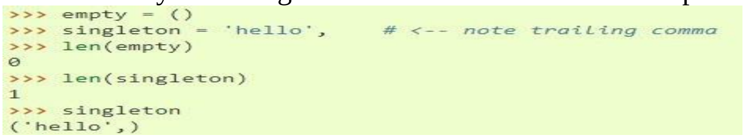


```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> # Tuples are immutable:
... t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # but they can contain mutable objects:
... v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

As you see, on output tuples are always enclosed in parentheses, so that nested tuples are interpreted correctly; they may be input with or without surrounding parentheses, although often parentheses are necessary anyway (if the tuple is part of a larger expression). It is not possible to assign to the individual items of a tuple, however it is possible to create tuples which contain mutable objects, such as lists.

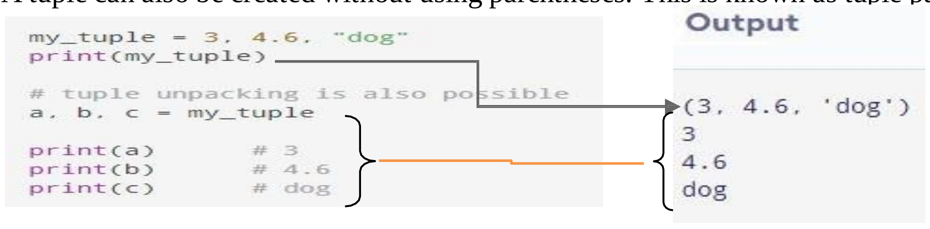
Though tuples may seem similar to lists, they are often used in different situations and for different purposes. Tuples are immutable, and usually contain a heterogeneous sequence of elements that are accessed via unpacking or indexing (or even by attribute in the case of namedtuples). Lists are mutable, and their elements are usually homogeneous and are accessed by iterating over the list.

A special problem is the construction of tuples containing 0 or 1 items: the syntax has some extra quirks to accommodate these. Empty tuples are constructed by an empty pair of parentheses; a tuple with one item is constructed by following a value with a comma. For example:



```
>>> empty = ()
>>> singleton = 'hello',    # <-- note trailing comma
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

A tuple can also be created without using parentheses. This is known as tuple packing.



```
my_tuple = 3, 4.6, "dog"
print(my_tuple)

# tuple unpacking is also possible
a, b, c = my_tuple

print(a)      # 3
print(b)      # 4.6
print(c)      # dog
```

Output

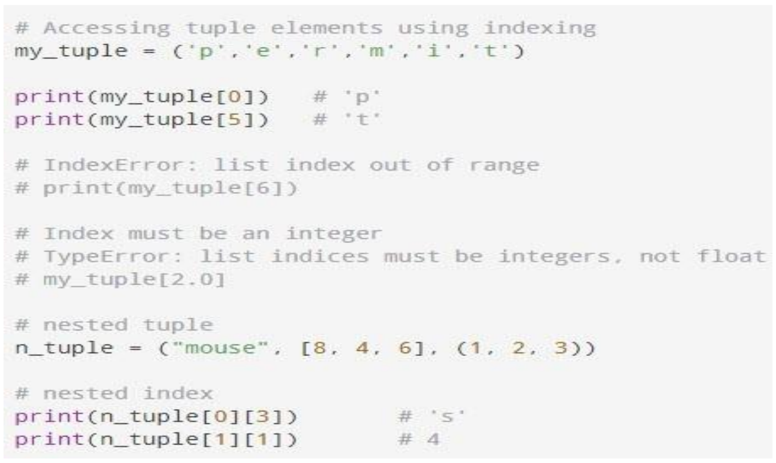
```
(3, 4.6, 'dog')
3
4.6
dog
```

4.10 Access Tuple Elements

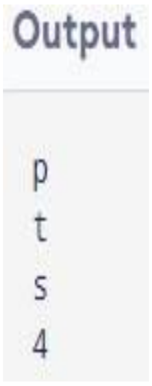
There are various ways in which we can access the elements of a tuple.

Indexing:

We can use the index operator [] to access an item in a tuple, where the index starts from 0. So, a tuple having 6 elements will have indices from 0 to 5. Trying to access an index outside of the tuple index range (6,7,... in this example) will raise an IndexError.



```python
# Accessing tuple elements using indexing
my_tuple = ('p','e','r','m','i','t')

print(my_tuple[0])   # 'p'
print(my_tuple[5])   # 't'

# IndexError: list index out of range
# print(my_tuple[6])

# Index must be an integer
# TypeError: list indices must be integers, not float
# my_tuple[2.0]

# nested tuple
n_tuple = ("mouse", [8, 4, 6], (1, 2, 3))

# nested index
print(n_tuple[0][3])       # 's'
print(n_tuple[1][1])       # 4
```

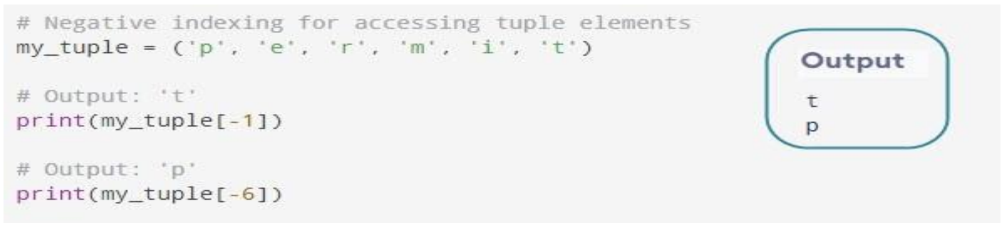


```
Output

p
t
s
4
```

The index must be an integer, so we cannot use float or other types. This will result in TypeError.
Likewise, nested tuples are accessed using nested indexing, as shown in the example below.

Negative Indexing:

Python allows negative indexing for its sequences.
The index of -1 refers to the last item, -2 to the second last item and so on.



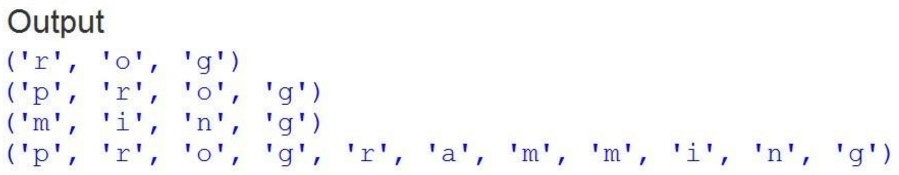
```python
# Negative indexing for accessing tuple elements
my_tuple = ('p', 'e', 'r', 'm', 'i', 't')

# Output: 't'
print(my_tuple[-1])

# Output: 'p'
print(my_tuple[-6])
```

```
Output
t
p
```

Slicing:

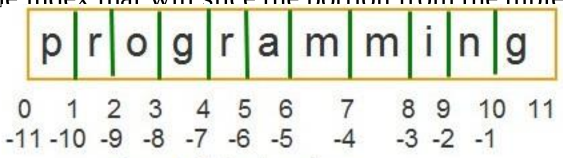
We can access a range of items in a tuple by using the slicing operator ***colon:***.

```python
# Accessing tuple elements using slicing
my_tuple = ('p','r','o','g','r','a','m','m','i','n','g')

# elements 2nd to 4th
# Output: ('r', 'o', 'g')
print(my_tuple[1:4])

# elements beginning to 2nd
# Output: ('p', 'r','o', 'g')
print(my_tuple[:-7])

# elements 8th to end
# Output: ('m','i', 'n','g')
print(my_tuple[7:])

# elements beginning to end
# Output: ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'n','g')
print(my_tuple[:])
```



```
Output
('r', 'o', 'g')
('p', 'r', 'o', 'g')
('m', 'i', 'n', 'g')
('p', 'r', 'o', 'g', 'r', 'a', 'm', 'm', 'i', 'n', 'g')
```

Slicing can be best visualized by considering the index to be between the elements as shown below. So if we want to access a range, we need the index that will slice the portion from the tuple.



element slicing in python

4.11 Changing a Tuple

Unlike lists, tuples are immutable. This means that elements of a tuple cannot be changed once they have been assigned. But, if the element is itself a mutable data type like list, its nested items can be changed.
We can also assign a tuple to different values (reassignment).

```
>>> t=(5,46,34)
>>> t[0]=43
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    t[0]=43
TypeError: 'tuple' object does not support item assignment
```

Deleting a Tuple

As discussed above, we cannot change the elements in a tuple. It means that we cannot delete or remove items from a tuple. Deleting a tuple entirely, however, is possible using the keyword *del*.

```
# Accessing tuple elements using slicing
my_tuple = ('p','r','o','g','r','a','m','m','i','n','g')

#deleting entire tuple using del keyword
del my_tuple

#trying to print the deleted tuple,
#it will cause an error because tuple has already deleted
print(my_tuple)
```

Output:

```
Traceback (most recent call last):
  File "C:/Users/VJ/Desktop/jaffa.py", line 9, in <module>
    print(my_tuple)
NameError: name 'my_tuple' is not defined
```

4.12 Tuple operations

We can use + operator to combine two tuples. This is called **concatenation**.
We can also **repeat** the elements in a tuple for a given number of times using the * operator.
Both + and * operations result in a new tuple.

```
# Concatenation
# Output: (1, 2, 3, 4, 5, 6)
print((1, 2, 3) + (4, 5, 6))

# Repeat
# Output: ('Repeat', 'Repeat', 'Repeat')
print(("Repeat",) * 3)
```

Output

```
(1, 2, 3, 4, 5, 6)
('Repeat', 'Repeat', 'Repeat')
```

Tuple Membership Test

We can test if an item exists in a tuple or not, using the keyword *in*.

```
# Membership test in tuple
my_tuple = ('a', 'p', 'p', 'l', 'e',)

# In operation
print('a' in my_tuple)
print('b' in my_tuple)

# Not in operation
print('g' not in my_tuple)
```

Output

```
True
False
True
```

4.13 Iterating Through a Tuple

We can use a *for* loop to iterate through each item in a tuple.

```
# Using a for loop to iterate through a tuple
for name in ('John', 'Kate'):
    print("Hello", name)
```

Output

```
Hello John
Hello Kate
```

4.14 Built-in functions

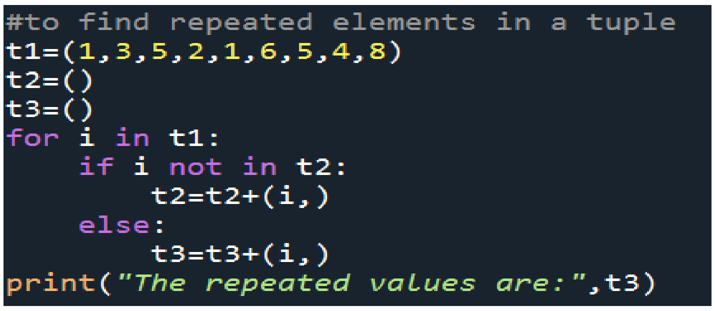
len()- to determine how many elements in the tuple
tuple()-a function to make a tuple
count()-Returns the number of times a specified value occurs in a tuple
index()-Searches the tuple for a specified value and returns the position where it was found
min()-Returns the minimum value in a tuple
max()-Returns the maximum value in a tuple

4.15 Advantages of Tuple over List

Since tuples are quite similar to lists, both of them are used in similar situations. However, there are certain advantages of implementing a tuple over a list. Below listed are some of the main advantages:
We generally use tuples for heterogeneous (different) data types and lists for homogeneous (similar) data types.
Since tuples are immutable, iterating through a tuple is faster than with list. So there is a slight performance boost.
Tuples that contain immutable elements can be used as a key for a dictionary. With lists, this is not possible.

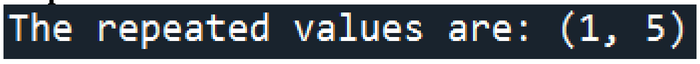
If you have data that doesn't change, implementing it as tuple will guarantee that it remains write-protected.

Example 1



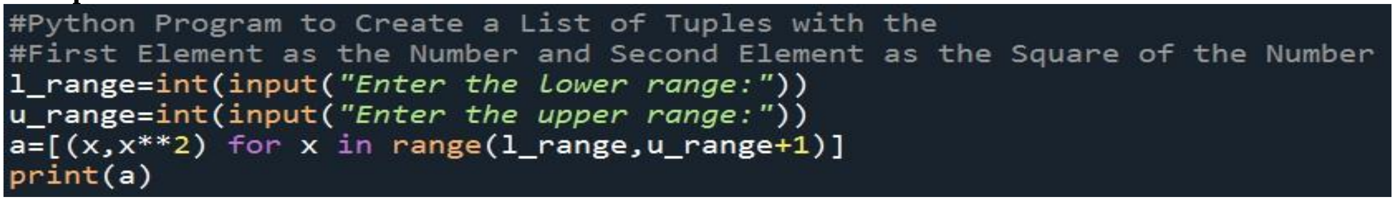
```python
#to find repeated elements in a tuple
t1=(1,3,5,2,1,6,5,4,8)
t2=()
t3=()
for i in t1:
    if i not in t2:
        t2=t2+(i,)
    else:
        t3=t3+(i,)
print("The repeated values are:",t3)
```

Output:



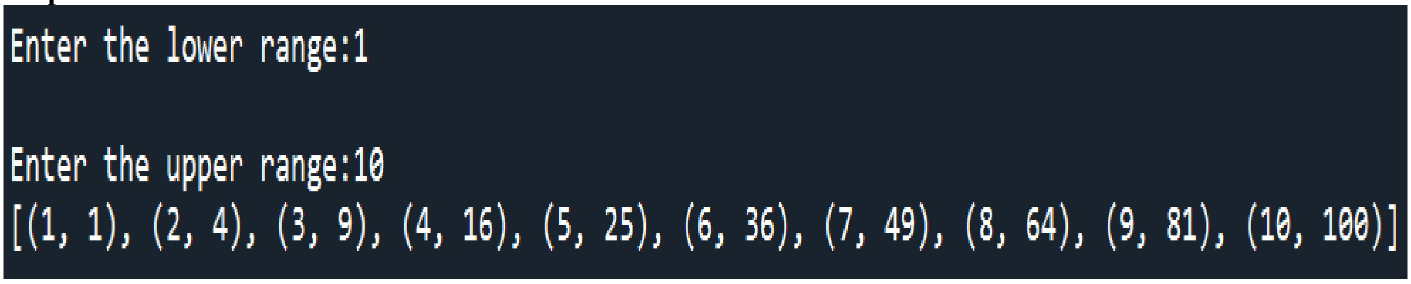
```
The repeated values are: (1, 5)
```

Example: 2



```python
#Python Program to Create a List of Tuples with the
#First Element as the Number and Second Element as the Square of the Number
l_range=int(input("Enter the lower range:"))
u_range=int(input("Enter the upper range:"))
a=[(x,x**2) for x in range(l_range,u_range+1)]
print(a)
```

Output:



```
Enter the lower range:1

Enter the upper range:10
[(1, 1), (2, 4), (3, 9), (4, 16), (5, 25), (6, 36), (7, 49), (8, 64), (9, 81), (10, 100)]
```

Unit 5 - Strings and Dictionary

Module 1 -Strings

1.1 Definition

In python, consecutive sequence of characters is known as a string. Strings are immutable and amongst the most popular types in Python. In python, String literals are surrounded by single or double or triple quotation marks. ‘RGUKT’ is same as “RGUKT”

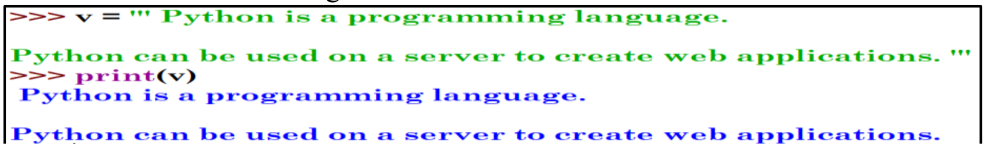
1.1.1 Declaring & Initializing StringVariables

Assigning a string to a variable is done with the variable name followed by an equal sign and the string

>>> s1 = “This is a stringone”
>>> s2 = “This is a string two”
>>>s3=” “   #emptystring
>>>type(s1)<class'str'>

- Multi-line strings can be denoted using triple single or double quotes, ''' or""".
- Even triple quotes can be used in Python but generally used to represent **multiline strings** or**docstrings**.

>>> s4 = ‘’’amultiline String’’’



```
>>> v = ''' Python is a programming language.

Python can be used on a server to create web applications. '''
>>> print(v)
 Python is a programming language.

Python can be used on a server to create web applications.
```

1.1.2 How to access characters in String

- We can access individual characters using indexing and a range of characters using slicingoperator **[ ]**or **TheSubscriptOperator**
- Index starts from 0. Trying to access a character out of index range will raise an Index Error.
- The index must be an integer. We can't use float or other types;this will result into Type Error.
- Python allows negative indexing for itssequences.
- The index of -1 refers to the last item, -2 to the second last item and soon.
- We can access a range of items in a string by using the slicing operator withcolon**[:]**.

S=”hello”

0	1	2	3	4
H	E	L	L	O
-5	-4	-3	-2	-1

Important points about accessing elements in the strings using index

- Positive index helps in accessing the string from the beginning
- Negative index helps in accessing the string from the end.
- Index 0 or –ve n(where n is length of the string) displays the first element.
  Example: A[0] or A[-5] will display “H”
- Index -1 or (n-1) displays the last element.
  Example: A[-1] or A[4] will display “O”

Note: Python does not support character data type. A string of size 1 can be treated as characters.
If we try to access index out of the range or use decimal number, we will get errors.

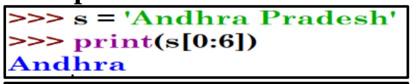
Accessing sub strings/Slicing Operator:

To access substrings, use the square brackets for slicing, using slice operator, along with the index or indices to obtain your substring.

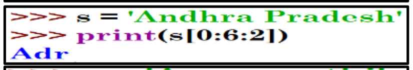
Syntax:

stringname[start:end:step]
start – starting point of the index value (default value: 0)
end or stop – ending point of the index value (default value: len(stringname))
step- increment of the index values (default value: 1)

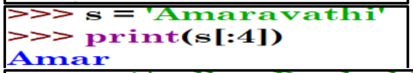
Examples:



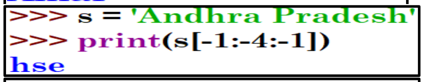
```
>>> s = 'Andhra Pradesh'
>>> print(s[0:6])
Andhra
```



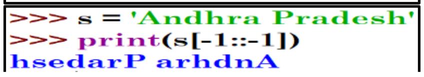
```
>>> s = 'Andhra Pradesh'
>>> print(s[0:6:2])
Adr
```



```
>>> s = 'Amaravathi'
>>> print(s[:4])
Amar
```



```
>>> s = 'Andhra Pradesh'
>>> print(s[-1:-4:-1])
hse
```



```
>>> s = 'Andhra Pradesh'
>>> print(s[-1::-1])
hsedarP arhdnA
```

1.1.3 How to change or delete a string?

Strings are immutable

Strings are immutable means that the contents of the string cannot be changed after it is created.
Let us understand the concept of immutability with help of an example.We can simply reassign different strings to the samename.

```
my_string = 'Hello world'
my_string[5]='j'

Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    my_string[5]='j'
TypeError: 'str' object does not support item assignment
```

We cannot delete or remove characters from string,deleting the string is possible use the keyword **del**.

```
my_string = 'Hello world'
del my_string[5]

TypeError: 'str' object doesn't support item deletion
>>> del my_string
>>> my_string
...
NameError: name 'my_string' is not defined
```

5.2. Python String Operations

There are many operations that can be performed with string which makes it one of the most used data types in python

Operator	Description	Example
+ (Concatenation)	The + operator joins the text on both sides of the operator	>>> 'Save'+'Earth' 'Save Earth' To give a white space between the two words, insert a space before the closing single quote of the first literal.
* (Repetition)	The * operator repeats the string on the left hand side times the value on right hand side.	>>>3*'Save Earth ' 'Save Earth Save Earth Save Earth '
in (Membership)	The operator displays 1 if the string contains the given character or the sequence of characters.	>>>A='Save Earth' >>> 'S' in A True >>>'Save' in A True >>'SE' in A False
not in	The operator displays 1 if the string does not contain the given character or the sequence of characters. (working of this operator is the reverse of **in** operator discussed above)	>>>'SE' not in 'Save Earth' True >>>'Save ' not in 'Save Earth' False

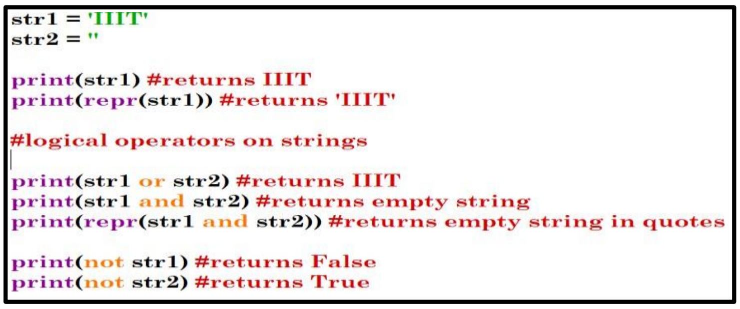
Comparing Strings: Python allows you to compare strings using relational (or comparison) operator such as

- = =,! =, >, <, < =, >=, etc.
- == if two strings are equal, it returns True
- != or <> if two strings are not equal, it returns True
    - ➢ if first string is greater than the second, it returns True
- < if second string is greater than the first ,it returns True
- >= if first string is greater than or equal to the second, it returns True
- <= if second string is greater than or equal to the first, it returns True

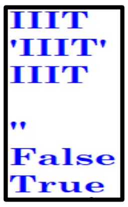
Note:

- *These operators compare the strings by using the lexicographical order i.e using ASCII values of the characters.*
- *The ASCII values of A-Z is 65 -90 and ASCII code for a-z is 97-122 .*

Logical Operators on String in Python: Python considers empty strings as having boolean value of 'false' and non-empty string as having boolean value of 'true'.
Let us consider the two strings namely str1 and str2 and try boolean operators on them:


```
str1 = 'IIIT'
str2 = ''

print(str1) #returns IIIT
print(repr(str1)) #returns 'IIIT'

#logical operators on strings

print(str1 or str2) #returns IIIT
print(str1 and str2) #returns empty string
print(repr(str1 and str2)) #returns empty string in quotes

print(not str1) #returns False
print(not str2) #returns True
```

Output:


```
IIIT
'IIIT'
IIIT

''
False
True
```

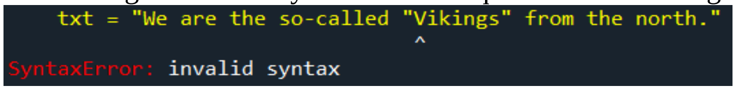
5.3 Escape sequence characters

- To insert characters that are illegal in a string, use an escape character.
- Escape sequence characters or non-printable characters that can be represented with backslash notation.
- An escape character gets interpreted; in a single quoted as well as double quoted strings

An example of an illegal character is a double quote inside a string that is surrounded by double quotes

Example

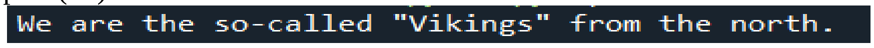
You will get an error if you use double quotes inside a string that is surrounded by double quotes:
txt = "We are the so-called "Vikings" from the north."
#You will get an error if you use double quotes inside a string that are surrounded by double quotes:


```
    txt = "We are the so-called "Vikings" from the north."
                                       ^
SyntaxError: invalid syntax
```

To fix this problem, use the escape character \":

Example

The escape character allows you to use double quotes when you normally would not be allowed:
txt = "We are the so-called \"Vikings\" from the north."
print(txt)


```
We are the so-called "Vikings" from the north.
```

Other escape characters used in Python

Escape Character	Description	Example	Result

\'	Single Quote	txt = 'It\'s alright.'
		print(txt)
			It's alright.

\\	Backslash	txt = "This will insert one \\ (backslash)." print(txt)	This will insert one \ (backslash).

\n	New Line	txt = "Hello\nWorld!"
		print(txt)
			Hello
			World!

\r	Carriage Return	txt = "Hello\rWorld!"
		print(txt)
			Hello
			World!

\t	Tab	txt = "Hello\tWorld!"
		print(txt)
			Hello World!

\b	Backspace	#This example erases one character (backspace):
		txt = "Hello \bWorld!"
		print(txt)
			HelloWorld!

\ooo	Octal value	#A backslash followed by three integers will result in a octal value:
		txt = "\110\145\154\154\157"
		print(txt)
			Hello

\xhh	Hex value	#A backslash followed by an 'x' and a hex number represents a hex value:
		txt = "\x48\x65\x6c\x6c\x6f"
		print(txt)
			Hello

5.4 Iterating through String/Traversing a string

Traversing a string means accessing all the elements of the string one after the other by using the subscript. A string can be traversed using: for loop or while loop.

Example:

```
"""
Python Program:
 Using for loop to iterate over a string in Python
"""
string_to_iterate = "RGUKT"
for char in string_to_iterate:
   print(char)
```

Example:

```
"""
Python Program:
 Using for loop to iterate using index of a string in Python
"""
string_to_iterate = "RGUKT"
for i in range(len(string_to_iterate)):
   print(string_to_iterate[i])
```

Example:

```
"""
Python Program:
 Using while loop to iterate using index of a string in Python
"""
string_to_iterate = "RGUKT"
i=0
while i< len(string_to_iterate):
   print(string_to_iterate[i])
   i=i+1
```

Example:

Python program to print reverse string using for loop

```
s1='Hello' # Assign String here
s2='' #empty string
l=0 # length
for i in s1:
    l=l+1
#finaly legth of the string l=5
for i in range(1,l+1):
    s2+=s1[l-i]
print("The Reverse String of ",s1,"is",s2)
```

5.5 The format() Method for Formatting Strings

The format() method that is available with the string object is very versatile and powerful in formatting strings. Format strings contains curly braces {} as placeholders or replacement fields which gets replaced. We can use positional arguments or keyword arguments to specify the order.

```
default_order = "{} and {}".format('John','Bill','sean')
print("\n --- default Order ---")
print(default_order)
positional_order = '{1},{0} and {2}'.format('John','Bill','sean')
print("\n --- Positional Order ---")
print(positional_order)
Keyword_order = '{s},{b} and {j}'.format(j='John',b='Bill',s='sean')
print("\n --- Keyword Order ---")
print(Keyword_order)
```

5.6 Built-in functions to Work with Python Strings

- Various built-in functions that work with sequence, works with string as well.
- The **enumerate()** function returns an enumerate object. It contains the index and value of all the items in the string as pairs. This can be useful for iteration.
- The **len()** function returns the length (number of characters) of the string.

```
s='RGUKT'
#enumerate()
list_enumerate=list(enumerate(s))
print('list(enumerate(s))=',list_enumerate)
#list(enumerate(s))= [(0,'R'),(1,'G'),(2,'U'),(3,'K'),(4,'T')]

#character count
print('length of the string =',len(s))
#length of the string = 5
```

The method **center()** makes str centered by taking width parameter into account. Padding is specified by parameter ***fillchar***. Default filler is a space.

Syntax: str.center(width[, fillchar])

```
str = "RGUKT IIIT"
str1= str.center(30,'a')
str2= str.center(30)
print(str1)
print (str2 )
```

The function **max()** returns the max character from string str according to ASCII value.in first print statement y is max character, because ASCII code of "y" is 121. In second print statement "s" is max character, ASCII code of "s" is 115.

Syntax: max(str)

The function **min()** returns the min character from string str according to ASCII value.

Syntax: min(str)

```
s="RGUKT-RKV RGUKT-RKV"
print(s.split())
print(s.split('-',1))

s="RGUKT-RKVRGUKT-RKV"
print(max(s))

print(min(s))
```

The **ord()** function returns ASCII code of the character.

```
ch='A'
print(ord(ch)) # 65
```

The **chr()** function returns character represented by a ASCII number.

```
num=90
print(chr(num)) # Z
```

<u>Module 2 - String Methods</u>

count(): The method **count()** returns the number of occurrence of Python string *substr* in string *str*. By using parameter *start* and *end* you can give slice of *str*.This function takes 3 arguments, substring, beginning position (by default 0) and end position (by default string length).Return Int Value

Syntax: str.count(substr [, start [, end]])

```
str = "This is a count example"
sub = "i"
print(str.count(sub, 4, len(str)))
sub = "a"
print (str.count(sub) )
```

endswith("string", beg, end): This function returns true if the string ends with mentioned string(suffix) else return false.Return Bool Value

The use of *start* and *end* to generate slice of Python string str.

Syntax: str.endswith(suffix[, start[, end]])

```
str = "RGUKT IIIT RKVALLEY"
sub="IIIT"
l=len(str)
print(l)
print(str.endswith(sub,2,6))
print(str.endswith(sub,10))
sub='sdfs'
print(str.endswith(sub))
```

startswith("string", beg, end): This function returns true if the string begins with mentioned string(prefix) else return false.

```
s = 'Amaravathi, Tirupathi'
i = s.startswith("Amar") #checks from the beginning of the string and returns result
print(i) #displays True

i = s.startswith("Amar", 10, 20) #checks from 10th index and returns result
print(i) #displays False
```

find("string", beg, end): This function is used to find the position of the substring within a string. It takes 3 arguments, substring , starting index(by default 0) and ending index(by default string length).

- It returns "-1" if string is not found in given range.
- It returns first occurrence of string if found.

Return the lowest index in S where substring sub is found

If given Python string is found, then the **find()** method returns its index. If Python string is not found then -1 would be returned.

Syntax : str.find(str, beg=0 end=len(string))

```
str = "RGUKT IIIT RKVALLEY"
sub1="IIIT"
sub2='RKV'
print(str.find(sub1))
print(str.find(sub1,20))
print(str.find(sub2))
```

rfind("string", beg, end): This function is similar to find(), but it returns the position of the last occurrence of sub string.

```
s = "Amaravathi, Tirupathi"
i = s.rfind("thi") #checks entire string and returns the last occurrence of substring i.e., 18
print(i)
```

replace(): This function is used to replace the substring with a new substring in the string. This function has 3 arguments. The string to replace, new string which would replace and max value denoting the limit to replace action (by default unlimited).

```
#string.replace(oldstring, newstring)
st="RGUKT RKV"
n=st.replace('RKV','ONG')
print(n)
```

The method **isalnum()** is used to determine whether the Python string consists of alphanumeric characters, false otherwise. **Syntax:str.isalnum()**

The method **isalpha()** return true if the Python string contains only alphabetic character(s), false otherwise.

Syntax: str.isalpha()

The method **isdigit()** return true if the Python string contains only digit(s),false otherwise.

Syntax: str.isdigit()

The method **islower()** return true if the Python string contains only lower cased character(s), false otherwise

Syntax: str.isdigit()

```
s='rgukt1234'
print(s.isalnum())
s1='#$@#%'
print(s1.isalnum())
s2='PinCode'
print(s2.isalpha())
s3='rgukt1234'
print(s3.isalnum())
s4='1234'
print(s3.isdigit())
s5='rgukt1234'
print(s5.isdigit())
```

```
s='rguktrkv'
print(s.islower())
s1='pin343223'
print(s1.islower())
s2='PinCode'
print(s2.islower())
```

The method **isspace()** return true if the Python string contains only white space(s).

Syntax: str.isspace()

The method **istitle()** return true if the string is a titlecased

Syntax: str.istitle()

The method **isupper()** return true if the string ontains only upper cased character(s), false otherwise.

Sytax: str.isupper()

The method **ljust(),** it returns the string left justified. Total length of string is defined in first parameter of method *width*. Padding is done as defined in second parameter *fillchar* .(default is space)

Syntax:-str.ljust(width[, fillchar])

```
s=" "
print(s.isspace())

s1="pin 2342"
print(s1.isspace())

s2="rgukt rkv"
print(s2.istitle())

s3="Rgukt rkv"
print(s3.istitle())

s4="Rgukt Rkv"
print(s4.istitle())
```

```
s5="RGUKT"
print(s5.isupper())

s6="RGUKT123"
print(s6.isupper())

s7="rGUKT"
print(s7.isupper())

s="rgukt"
print(s.ljust(10,"k"))
print(s.ljust(10))
print(s.ljust(3,"k"))
```

In above example you can see that if you don't define the *fillchar* then the method **ljust()** automatically take space as *fillchar*.

The method **rjust(),** it returns the string right justified. Total length of string is defined in first parameter of method*width*. Padding is done as defined in second parameter *fillchar* .(default is space)

Syntax: str.rjust(width[, fillchar])

This function **capitalize()** first letter of string.

Syntax: str.capitalize()

The method **lower()** returns a copy of the string in which all case-based characters have been converted to lower case. **Syntax: str.lower()**

The method **upper()** returns a copy of the string in which all case-based characters have been converted to upper case.

Syntax: str.upper()

The method **title()** returns a copy of the string in which first character of all words of string are capitalised.

Syntax: str.title()

The method **swapcase()** returns a copy of the string in which all cased based character swap their case

Syntax: str.swapcase()

```
s = "APJ Kalam is GREAT Scientist"

l = s.lower() #returns a new string with all lower case letters in it
print(l)

u = s.upper() #returns a new string with all upper case letters in it
print(u)

sc = s.swapcase() #returns a new string 'apj kALAM IS great sCIENTIST'
print(sc)

t = s.title() #returns a new string 'Apj Kalam Is Great Scientist'
print(t)

c = s.capitalize() #returns a new string 'Apj kalam is great scientist'
print(c)
```

This method **join()** returns a string which is the concatenation of given sequence and stringas shown in example.
seq = it contains the sequence of separated strings.
str = it is the string which is used to replace the separator of sequence
Syntax: str.join(seq)
The method **lstrip()** returns a copy of the string in which specified char(s) have been stripped from left side of string. If char is not specified then space is taken as default.
Syntax: str.lstrip([chars])
The method **rstrip()** returns a copy of the string in which specified char(s) have been stripped from right side of string. If char is not specified then space is taken as default.
Syntax: str.rstrip([chars])
The method **strip()** returns a copy of the string in which specified char(s) have been stripped from both side of string. If char is not specified then space is taken as default.
Syntax: str.strip([chars])
The method **split()** returns a list of all words in the string, delimiter separates the words. If delimiter is not specified then whitespace is taken as delimiter, parameter num
Syntax: str.split("delimiter", num)
Multiple Choice Questions

```
str1="6/4"

print("str1")
```

1) What will be the output of above Python code?
   a) 1 b) 6/4 **c) 1.5** d) str1

```
str1="Information"

print(str1[2:8])
```

2) What will be the output of below Python code?
   a) **format** B. formation C. orma D. ormat

```
str1="Aplication"

str2=str1.replace('a','A')

print(str2)
```

3) What will be the output of below Python code?
   a) Application b) Application c) **ApplicAtion d)** Application

```
str1="poWer"

str1.upper()

print(str1)
```

4) What will be the output of below Python code?
   a) **POWER b)** Power c) power d) power

```
list1=[0,2,5,1]

str1="7"

for i in list1:

        str1=str1+i

print(str1)
```

5) What will the below Python code will return?
   a) 70251 b) 7 c) 15 d) **Error**
6) Which of the following will give "Simon" as output?

```
If str1="John,Simon,Aryan"
```

   a) print(str1[-7:-12]) b) print(str1[-11:-7]) c) **print(str1[-11:-6])** d) print(str1[-7:-11])

```
str1="Stack of books"
print(len(str1))
```

7) What will return following Python code?
   a) 13 B. 14 **C.15** D. 16

Solved Problems

1) Write a program to print Alphabet using ASCII Numbers?

```
ch='A'
print(ord(ch)) # A=65,a=97

ch='Z'
print(ord(ch)) # Z=90,z=122

for i in range(65,91):
    print(chr(i),end=" ")
#A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
print()
for i in range(97,123):
    print(chr(i),end=" ")
#a b c d e f g h i j k l m n o p q r s t u v w x y z
```

2) Write a Python program to calculate the length of a string.

```python
def string_length(str1):
    count = 0
    for char in str1:
        count += 1
    return count
print(string_length('w3resource.com'))
```

3) Write a Python program to get a string made of the first 2 and the last 2 chars from a given a string. If the string length is less than 2, return instead of the empty string.

```python
def string_both_ends(str):
  if len(str) < 2:
    return ''

  return str[0:2] + str[-2:]

print(string_both_ends('w3resource'))
print(string_both_ends('w3'))
print(string_both_ends('w'))
```

Programs to DO

1) Write a program to find number of digits, alphabets and symbols?
2) Write a program to convert lower case to upper case from given string?
3) Write a program to print the following output? image like

```
R
R G
R G U
R G U K
R G U K T
R G U K
R G U
R G
R
```

4) Write a program to check whether given string is palindrome or not?
5) Write a program to find no_ words, no_letters, no_digits and no_blanks in a line?
6) Write a program to sort list names in alphabetical order?
7) To find the first character from given string,count the number of times repeated and replaced with * except first character then print final string?
8) To find the strings in a list which are matched with first character equals to last character in a string?
9) Write a program that accepts a string from user and redisplays the same string after removing vowels from it?
10) This is a Python Program to take in two strings and display the larger string without using built-in functions?
11) Python Program to Read a List of Words and Return the Length of the Longest One?
12) Python Program to Calculate the Number of Upper-Case Letters and Lower-Case Letters in a String?

Module 3 - Dictionaries

A **dictionary** is like a list, but more general. In a list, the index positions have to be integers; in a dictionary, the indices can be (almost) any type.

You can think of a dictionary as a mapping between a set of indices (which are called **keys**) and a set of values. Each key map to a value. The association of a key and a value is called a **key-value pair** or sometimes an **item**.

As an example, we'll build a dictionary that map from English to Spanish words, so the keys and the values are all strings.

The function **dict** creates a new dictionary with no items. Because **dict** is the name of a built-in function, you should avoid using it as a variable name.

```
>>> eng2sp = dict()
>>> print (eng2sp)
{}
```

The curly brackets, {}, represent an empty dictionary. To add items to the dictionary,you can use square brackets:

```
>>> eng2sp['one'] = 'uno'
```

This line creates an item that maps from the key 'one' to the value 'uno'. If weprint the dictionary again, we see a key-value pair with a colon between the keyand value:

```
>>> print (eng2sp)
{'one': 'uno'}
```

This output format is also an input format. For example, you can create a newdictionary with three items:

```
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

But if you print eng2sp, you might be surprised:

```
>>> print (eng2sp)
{'one': 'uno', 'three': 'tres', 'two': 'dos'}
```

The order of the key-value pairs is not the same. In fact, if you type the same example on your computer, you might get a different result. In general, the order of items in a dictionary is unpredictable.

But that's not a problem because the elements of a dictionary are never indexed with integer indices. Instead, you use the keys to look up the corresponding values:

```
>>> print (eng2sp['two'])
'dos'
```

The key **'two'** always maps to the value 'dos' so the order of the items doesn't matter.

If the key isn't in the dictionary, you get an exception:
>>> print (eng2sp['four'])
KeyError: 'four'

5.7 Accessing Elements from Dictionary

While indexing is used with other data types to access values, a dictionary uses keys. Keys can be used either inside square brackets **[]** or with the **get()** method.
If we use the square brackets **[], KeyError** is raised in case a key is not found in the dictionary. On the other hand, the **get()** method returns **None** if the key is not found.

```
# get vs [] for retrieving elements
my_dict = {'name': 'Jack', 'age': 26}

# Output: Jack
print(my_dict['name'])

# Output: 26
print(my_dict.get('age'))

# Trying to access keys which doesn't exist throws error
# Output None
print(my_dict.get('address'))

# KeyError
print(my_dict['address'])
```

Output:

```
Jack
26
None
Traceback (most recent call last):
  File "<string>", line 15, in <module>
    print(my_dict['address'])
KeyError: 'address'
```

5.8 Changing and Adding Dictionary elements

Dictionaries are mutable. We can add new items or change the value of existing items using an assignment operator. If the key is already present, then the existing value gets updated. In case the key is not present, a new **(key: value)** pair is added to the dictionary.

```
# Changing and adding Dictionary Elements
my_dict = {'name': 'Jack', 'age': 26}

# update value
my_dict['age'] = 27

#Output: {'age': 27, 'name': 'Jack'}
print(my_dict)

# add item
my_dict['address'] = 'Downtown'

# Output: {'address': 'Downtown', 'age': 27, 'name': 'Jack'}
print(my_dict)
```

Output:

```
{'name': 'Jack', 'age': 27}
{'name': 'Jack', 'age': 27, 'address': 'Downtown'}
```

5.9 Looping Techniques

When looping through dictionaries, the key and corresponding value can be retrieved at the same time using the **items()** method.

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.items():
...     print(k, v)
...
gallahad the pure
robin the brave
```

When looping through a sequence, the position index and corresponding value can be retrieved at the same time using the **enumerate()** function.

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print(i, v)
...
0 tic
1 tac
2 toe
```

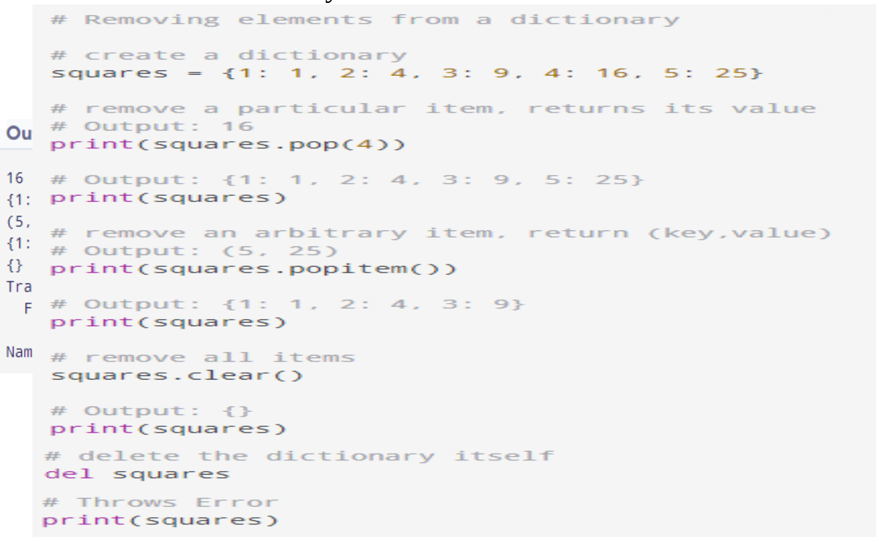
To loop over two or more sequences at the same time, the entries can be paired with the **zip()** function.

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print('What is your {0}?  It is {1}.'.format(q, a))
...
What is your name?  It is lancelot.
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
```

5.10 Removing elements from Dictionary

We can remove a particular item in a dictionary by using the **pop()** method. This method removes an item with the provided **key** and returns the **value**.

The **popitem()** method can be used to remove and return an arbitrary **(key, value)** item pair from the dictionary. All the items can be removed at once, using the **clear()** method.
We can also use the **del** keyword to remove individual items or the entire dictionary itself.

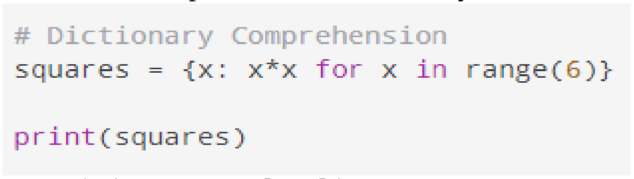


```
# Removing elements from a dictionary

# create a dictionary
squares = {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

# remove a particular item, returns its value
# Output: 16
print(squares.pop(4))

# Output: {1: 1, 2: 4, 3: 9, 5: 25}
print(squares)

# remove an arbitrary item, return (key,value)
# Output: (5, 25)
print(squares.popitem())

# Output: {1: 1, 2: 4, 3: 9}
print(squares)

# remove all items
squares.clear()

# Output: {}
print(squares)
# delete the dictionary itself
del squares

# Throws Error
print(squares)
```

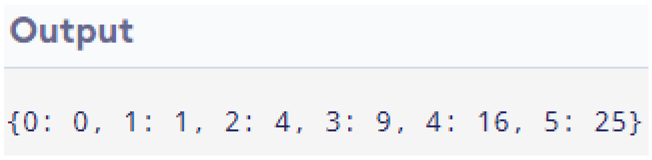
```
Ou

16
{1:
(5,
{1:
{}
Tra
  F

Nam
```

5.11 Dictionary Comprehension

Dictionary comprehension is an elegant and concise way to create a new dictionary from an iterable in Python.
Dictionary comprehension consists of an expression pair **(key: value)** followed by a **for** statement inside curly braces **{}**.
Here is an example to make a dictionary with each item being a pair of a number and its square.



```
# Dictionary Comprehension
squares = {x: x*x for x in range(6)}

print(squares)
```

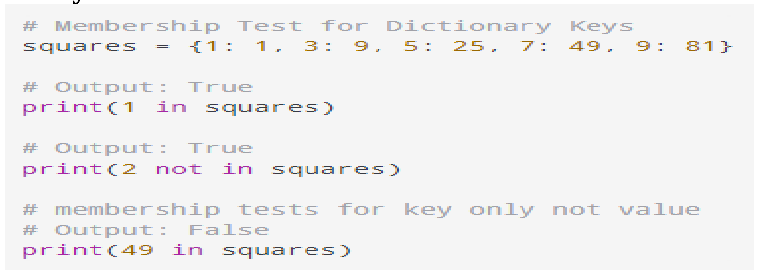


Output

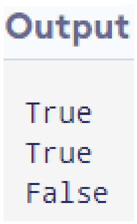
```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

5.12 Dictionary Membership Test

We can test if a **key** is in a dictionary or not using the keyword **in**. Notice that the membership test is only for the **keys** and not for the **values**.



```
# Membership Test for Dictionary Keys
squares = {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}

# Output: True
print(1 in squares)

# Output: True
print(2 not in squares)

# membership tests for key only not value
# Output: False
print(49 in squares)
```

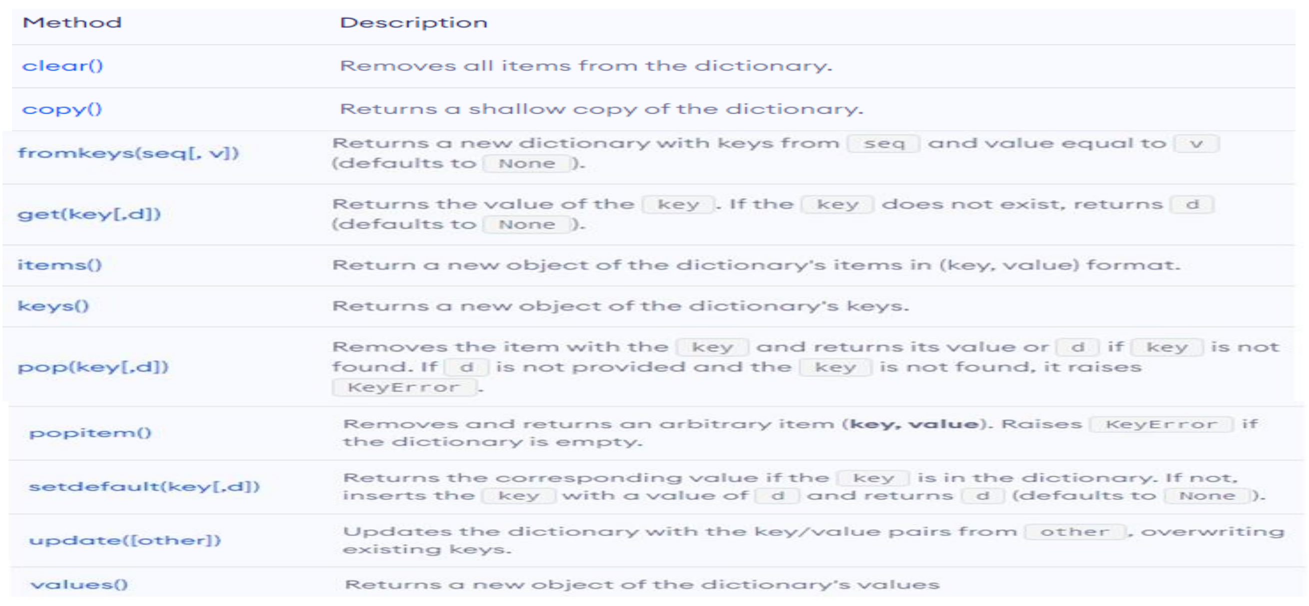


Output

```
True
True
False
```

5.13 Dictionary Methods

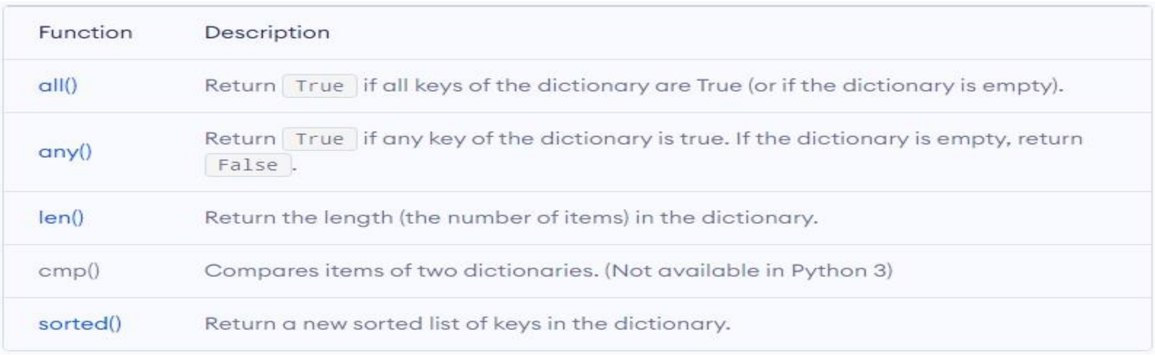
Methods that are available with a dictionary are tabulated below. Some of them have already been used in the above examples.



Method	Description
clear()	Removes all items from the dictionary.
copy()	Returns a shallow copy of the dictionary.
fromkeys(seq[, v])	Returns a new dictionary with keys from `seq` and value equal to `v` (defaults to `None`).
get(key[,d])	Returns the value of the `key`. If the `key` does not exist, returns `d` (defaults to `None`).
items()	Return a new object of the dictionary's items in (key, value) format.
keys()	Returns a new object of the dictionary's keys.
pop(key[,d])	Removes the item with the `key` and returns its value or `d` if `key` is not found. If `d` is not provided and the `key` is not found, it raises `KeyError`.
popitem()	Removes and returns an arbitrary item (**key, value**). Raises `KeyError` if the dictionary is empty.
setdefault(key[,d])	Returns the corresponding value if the `key` is in the dictionary. If not, inserts the `key` with a value of `d` and returns `d` (defaults to `None`).
update([other])	Updates the dictionary with the key/value pairs from `other`, overwriting existing keys.
values()	Returns a new object of the dictionary's values

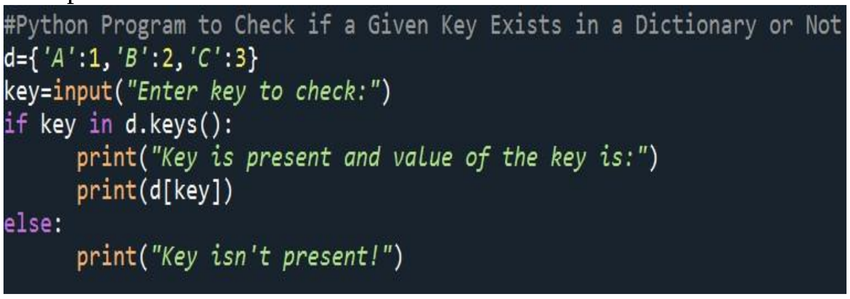
5.14 Dictionary Built-in Functions

Built-in functions like **all(), any(), len(), cmp(), sorted(),** etc. are commonly used with dictionaries to perform different tasks.

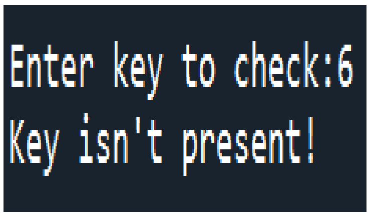


Function	Description
all()	Return `True` if all keys of the dictionary are True (or if the dictionary is empty).
any()	Return `True` if any key of the dictionary is true. If the dictionary is empty, return `False`.
len()	Return the length (the number of items) in the dictionary.
cmp()	Compares items of two dictionaries. (Not available in Python 3)
sorted()	Return a new sorted list of keys in the dictionary.

Example:1

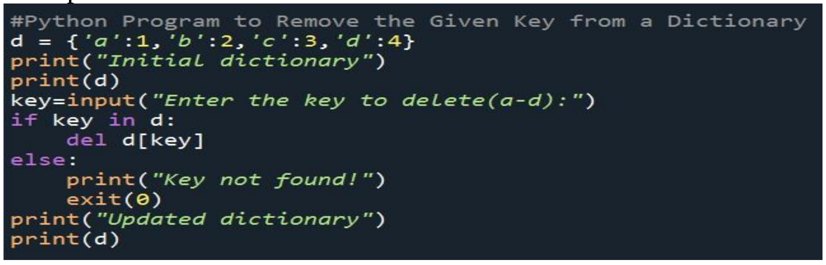


```
#Python Program to Check if a Given Key Exists in a Dictionary or Not
d={'A':1,'B':2,'C':3}
key=input("Enter key to check:")
if key in d.keys():
      print("Key is present and value of the key is:")
      print(d[key])
else:
      print("Key isn't present!")
```

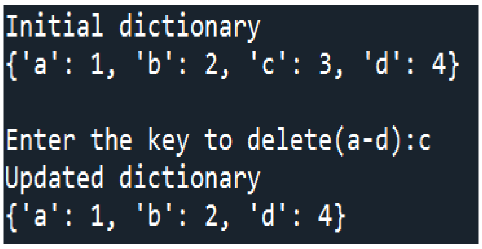


```
Enter key to check:6
Key isn't present!
```

Example:2

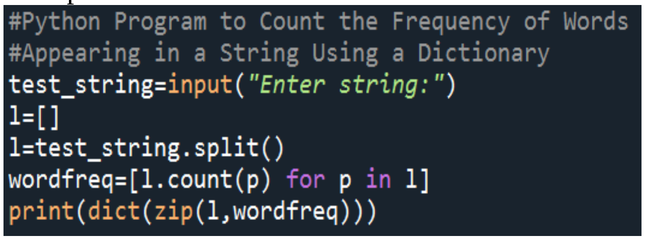


```
#Python Program to Remove the Given Key from a Dictionary
d = {'a':1,'b':2,'c':3,'d':4}
print("Initial dictionary")
print(d)
key=input("Enter the key to delete(a-d):")
if key in d:
    del d[key]
else:
    print("Key not found!")
    exit(0)
print("Updated dictionary")
print(d)
```

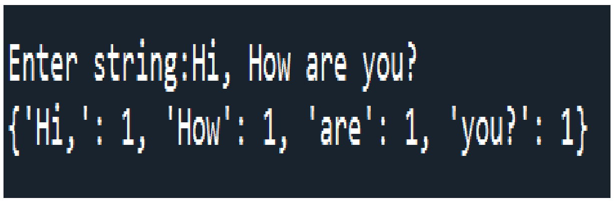


```
Initial dictionary
{'a': 1, 'b': 2, 'c': 3, 'd': 4}

Enter the key to delete(a-d):c
Updated dictionary
{'a': 1, 'b': 2, 'd': 4}
```

Example:3



```
#Python Program to Count the Frequency of Words
#Appearing in a String Using a Dictionary
test_string=input("Enter string:")
l=[]
l=test_string.split()
wordfreq=[l.count(p) for p in l]
print(dict(zip(l,wordfreq)))
```



```
Enter string:Hi, How are you?
{'Hi,': 1, 'How': 1, 'are': 1, 'you?': 1}
```

Unit 6 - Exceptions and File Handling

Module 1 – Exception Handling

Exception: An exception is an error that happens during execution of a program. When that error occurs, Python generate an exception that can be handled, which avoids your program to crash.

Why we use Exceptions: Exceptions are convenient in many ways for handling errors and special conditions in a program. When you think that you have a code which can produce an error then you can use exception handling.

Some example exceptions(errors):

IOError

If the file cannot be opened.

ImportError

If python cannot find the module

ValueError

Raised when a built-in operation or function receives an argument that has the right type but an inappropriate value

KeyboardInterrupt

Raised when the user hits the interrupt key (normally Control-C or Delete)

EOFError

Raised when one of the built-in functions (input() or raw_input()) hits an end-of-file condition (EOF) without reading any data

ZeroDivisionError

Raised when division or modulo by zero takes place for all numeric types.

NameError

Raised when an identifier is not found in the local or global namespace.

TypeError

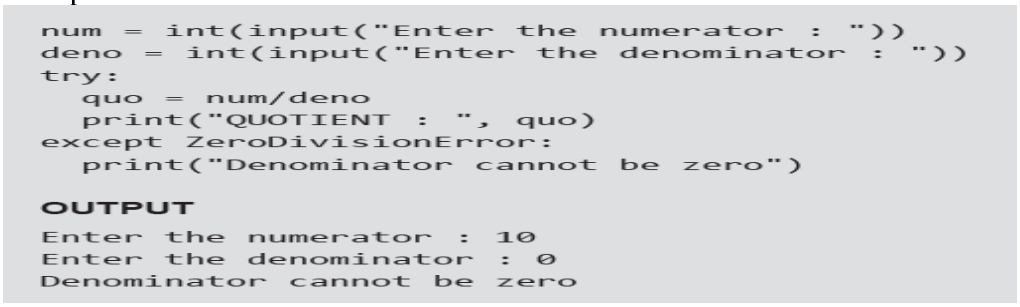
Raised when an operation or function is attempted that is invalid for the specified data type.

Handling Exceptions: We can handle exceptions in our program by using try block and except block. A critical operation which can raise exception is placed inside the try block and the code that handles exception is written in except block. The syntax for try–except block can be given as,

Syntax:

```
try:
        You do your operations here;
         ......................
except Exception:
        If there is Exception1, then execute this block.
```

Example



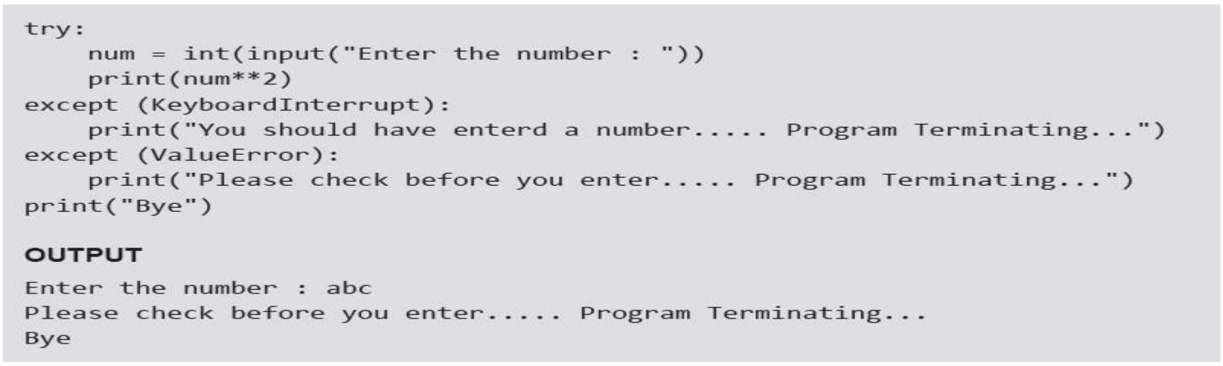
```
num = int(input("Enter the numerator : "))
deno = int(input("Enter the denominator : "))
try:
  quo = num/deno
  print("QUOTIENT : ", quo)
except ZeroDivisionError:
  print("Denominator cannot be zero")

OUTPUT
Enter the numerator : 10
Enter the denominator : 0
Denominator cannot be zero
```

Note:

1. A single try statement can have multiple except statements. This is useful when the try block contains statements that may throw different types of exceptions.
2. After the except clause(s), you can include an else-clause. The code in the else-block executes if the code in the try: block does not raise an exception.

try block with multiple except statements:

```
try:
        You do your operations here;
         ......................
except Exception1:
        If there is Exception1, then execute this block.
except Exception2:
        If there is Exception2, then execute this block.
        ......................
```



```
try:
    num = int(input("Enter the number : "))
    print(num**2)
except (KeyboardInterrupt):
    print("You should have enterd a number..... Program Terminating...")
except (ValueError):
    print("Please check before you enter..... Program Terminating...")
print("Bye")
```

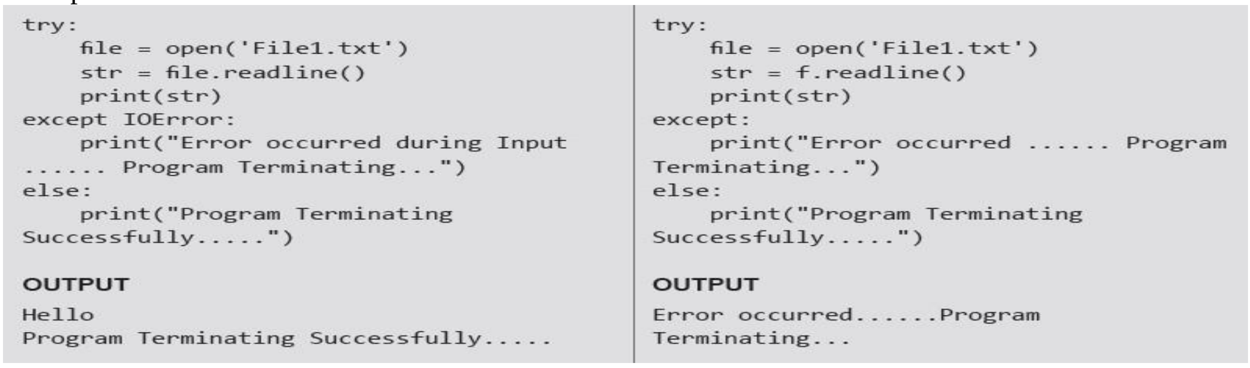
OUTPUT

```
Enter the number : abc
Please check before you enter..... Program Terminating...
Bye
```

The else clause:
The try ... except block can optionally have an else clause, which, when present, must follow all except blocks. The statement(s) in the else block is executed only if the try clause does not raise an exception.
Syntax:

```
try:
   You do your operations here;
   ......................
except:
   If there is any exception, then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

Example:



```
try:
    file = open('File1.txt')
    str = file.readline()
    print(str)
except IOError:
    print("Error occurred during Input
...... Program Terminating...")
else:
    print("Program Terminating
Successfully.....")
```

OUTPUT

```
Hello
Program Terminating Successfully.....
```

```
try:
    file = open('File1.txt')
    str = f.readline()
    print(str)
except:
    print("Error occurred ...... Program
Terminating...")
else:
    print("Program Terminating
Successfully.....")
```

OUTPUT

```
Error occurred......Program
Terminating...
```

Except block without exception name:
You can even specify an except block without mentioning any exception (i.e., except:). This type of except block if present should be at the end of the try-except block.

```
try:
   You do your operations here;
   ......................
except exception1:
   If there is any exception, then execute this block.
   ......................
except exception2:
   If there is any exception, then execute this block.
   ......................
except:
   If there is any exception, then execute this block.
   ......................
```

This kind of a try-except statement catches all the exceptions that occur. Using this kind of try-except statement is not considered a good programming practice though, because it catches all exceptions but does not make the programmer identify the root cause of the problem that may occur.

```
try:
    file = pen('File1.txt')
    str = f.readline()
    print(str)
except IOError:
    print("Error occured during Input ...... Program Terminating...")
except ValueError:
    print("Could not convert data to an integer.")
except:
    print("Unexpected error.... Program Terminating...")
```

Programming Tip: When an exception occurs, it may have an associated value, also known as the exception's *argument*.

OUTPUT

```
Unexpected error.... Program Terminating...
```

The except Clause with Multiple Exceptions

We can also use the same except statement to handle multiple exceptions as follows

```
try:
   You do your operations here;
except(Exception1[, Exception2[,...ExceptionN]]]):
   If there is any exception from the given exception list, then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

Raising an exception:

We can deliberately raise an exception using the raise keyword. The general syntax for the raise statement is,
raise Exception-Name

Example:

```
try:
     num = 10
     print(num)
     raise ValueError
except:
     print("Exception occurred .... Program Terminating...")
```

OUTPUT

```
10
Exception occurred .... Program Terminating...
```

The try-finally clause:

We can use a finally along with a try block. The finally block is a place to put any code that must execute, whether the try-block raised an exception or not. The syntax of the try-finally statement is this,

```
try:
     print("Raising Exception.....")
     raise ValueError
finally:
     print("Performing clean up in Finally......")
```

OUTPUT

```
Raising Exception.....
Performing clean up in Finally......
Traceback (most recent call last):
  File "C:\Python34\Try.py", line 4, in <module>
     raise ValueError
ValueError
```

Syntax:

```
try:
   You do your operations here;
   Due to any exception, this may be skipped.
finally:
   This would always be executed.
```

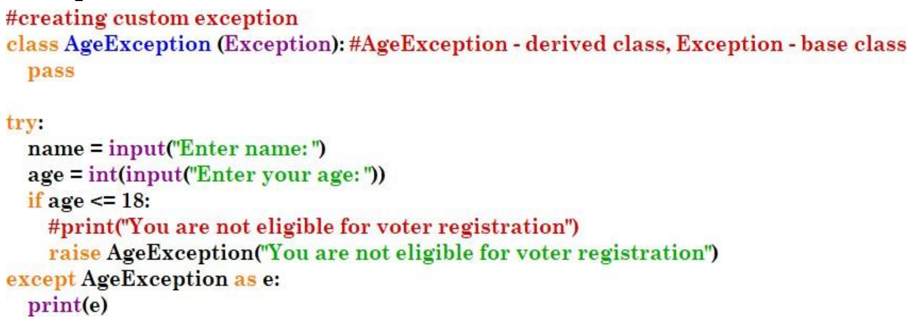
Example:

User Defined Exceptions:

Python has many built-in exceptions. If we want to create user-defined exception, we need to use Exception class. Programs may name their own exceptions by creating a new exception class.

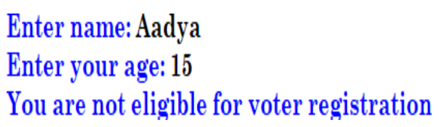
```
class UnderAge(Exception):
   pass
def verify_age(age):
   if int(age) < 18:
       raise UnderAge
   else:
       print('Age: '+str(age))
verify_age(23)  # won't raise exception
verify_age(17)  # will raise exception
```

Example



```
#creating custom exception
class AgeException (Exception): #AgeException - derived class, Exception - base class
   pass

try:
   name = input("Enter name: ")
   age = int(input("Enter your age: "))
   if age <= 18:
      #print("You are not eligible for voter registration")
      raise AgeException("You are not eligible for voter registration")
except AgeException as e:
   print(e)
```

Output:



```
Enter name: Aadya
Enter your age: 15
You are not eligible for voter registration
```

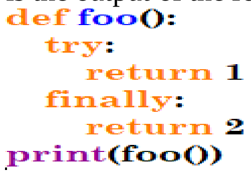
Multiple Choice Questions

1) How many except statements can a try-except block have?
   a) zero b) one c) more than one d) at least one

2. When will the else part of try-except-else be executed?
   a) always b) when an exception occurs
   c) when no exception occurs d) when an exception occurs in to except block

3) When is the finally block executed?
   a) when there is no exception b) when there is an exception
   c) only if some condition that has been specified is satisfied d) always

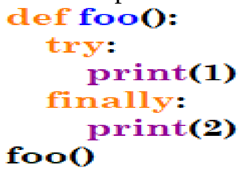
4) What is the output of the following code?



```
def foo():
   try:
      return 1
   finally:
      return 2
print(foo())
```

a) 1 b) 2 c) 3 d) error, there is more than one return statement in a single try-finally block

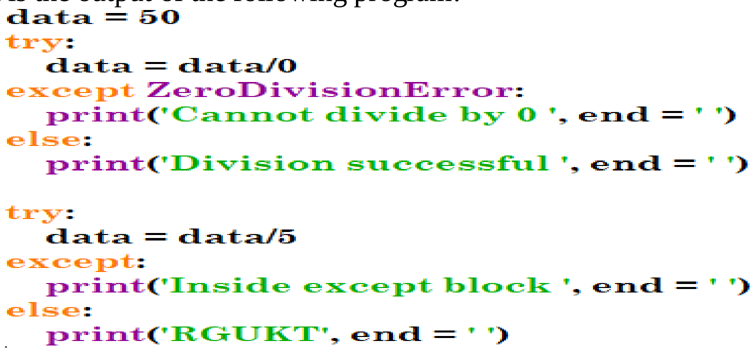
5) What is the output of the following code?



```
def foo():
   try:
      print(1)
   finally:
      print(2)
foo()
```

a) 1 2 b) 1 c) 2 d) none of the mentioned

6) What is the output of the following program?



```
data = 50
try:
   data = data/0
except ZeroDivisionError:
   print('Cannot divide by 0 ', end = ' ')
else:
   print('Division successful ', end = ' ')

try:
   data = data/5
except:
   print('Inside except block ', end = ' ')
else:
   print('RGUKT', end = ' ')
```

a) Cannot divide by 0 RGUKT b) Cannot divide by 0
c) Cannot divide by 0 Inside except block RGUKT d) Cannot divide by 0 Inside except block

7) What is the output of the following program?

```
data = 50
try:
    data = data/10
except ZeroDivisionError:
    print('Cannot divide by 0 ', end = ' ')
finally:
    print('PYTHON ', end = ' ')
else:
    print('Division successful ', end = ' ')
```

a) Runtime error b) Cannot divide by 0 PYTHON c) PYTHON Division successful d) PYTHON

8) What is the output of the following code?

```
valid = False
while not valid:
   try:
      n=int(input("Enter a number"))
      while n%2==0:
         print("Bye")
         valid = True
   except ValueError:
      print("Invalid")
```

a) PYTHON b) RGUKT c) RGUKT PYTHON d) Compilation error

9) What is the output of the following code?

```
value = [1, 2, 3, 4, 5]
try:
    value = value[5]/0
except (IndexError, ZeroDivisionError):
    print('PYTHON', end = ' ')
else:
    print('RGUKT ', end = ' ')
finally:
    print('IIIT ', end = ' ')
```

a) Compilation error b) Runtime error c) PYTHON RGUKT IIIT d) PYTHON IIIT

Problem Sets

1) Write a program to handle simple runtime error.
2) Write a program to handle multiple errors with one except statement.
3) Write a program to take a number from keyboard, raise an exception if the input value is not a number.
4) Write a program to read a list from the keyboard. Ask the user to enter an index and print the index value as output. Raise an exception if the index is invalid index?

Descriptive Questions

1) List the situation(s) in which the code may result in IOError?
2) When do we get TypeError?
3) Explain about finally with a suitable example.
4) How do you create a custom exception? Explain with a suitable example.
5) Why do we need exception handling? What happens if do not handle exceptions properly?
6) How do you throw an exception with a custom message?

Module 2 – Introduction to Files

What is a file?

What if the data with which, we are working or producing as output is required for later use? Result processing done in Term Exam is again required for Annual Progress Report. Here if data is stored permanently, its processing would be faster. This can be done if we are able to store data in secondary storage media i.e. Hard Disk, which we know is permanent storage media. Data is stored using file(s) permanently on secondary storage media. You have already used the files to store your data permanently - when you were storing data in Word processing applications, Spreadsheets, Presentation applications, etc. All of them created data files and stored your data, so that you may use the same later on. Apart from this, you were permanently storing your python scripts (as .py extension) also. Here is a basic definition of file handling in Python, ―File is a named location on the system storage which records data for later access. It enables persistent storage in a non-volatile memory i.e. Hard disk.― In python files are simply stream of data, so the structure of data is not stored in the file, along with data. Basic operations performed on a data file are:

- Naming a file
- Opening a file
- Reading data from the file
- Writing data in the file
- Closing a file

Using these basic operations, we can process a file in many ways, such as creating a file traversing a file for displaying the data on screen Appending data in file Inserting data in file Deleting data from file create a copy of file Updating data in the file, etc

File types

Python allows us to create and manage two types of file

Text:

A text file is usually considered as a sequence of lines. A line is a sequence of characters (ASCII), stored on permanent storage media. Although default character coding in python is ASCII but using constant u with string, supports Unicode as well. As we talk of lines in a text file, each line is terminated by a special character, known as End of Line (EOL). From strings, we know that \n is a newline character. So at the lowest level, a text file will be a collection of bytes. Text files are stored in human readable form and they can also be created using any text editor.

Binary:

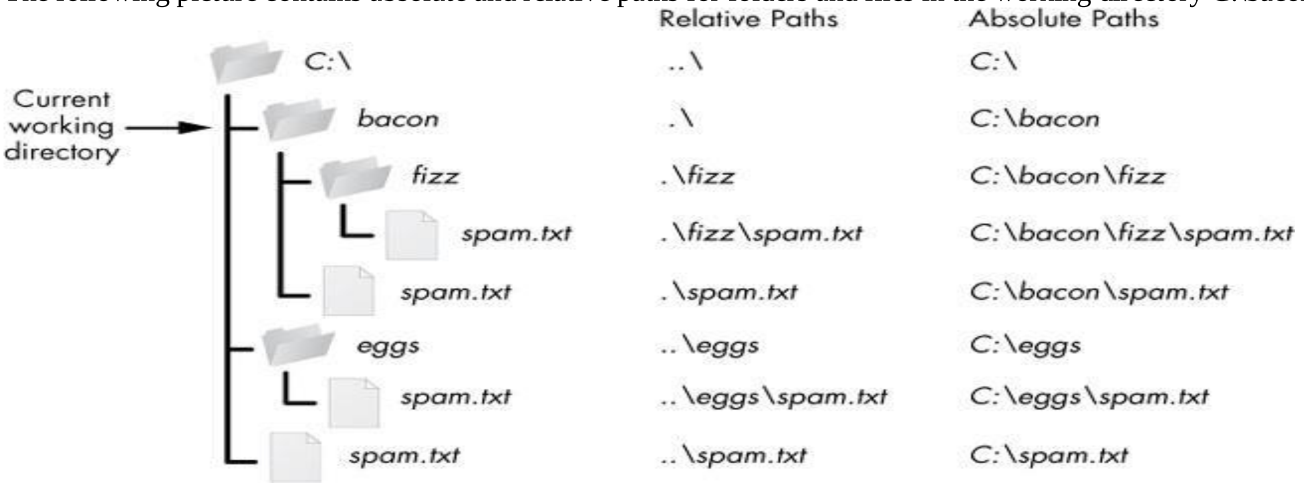
A binary file contains arbitrary binary data i.e. numbers stored in the file, can be used for numerical operation(s). So when we work on a binary file, we have to interpret the raw bit pattern(s) read from the file into the correct type of data in our program. It is perfectly possible to interpret a stream of bytes originally written as a string, as a numeric value. But we know that will be an incorrect interpretation of data and we are not going to get desired output after the file processing activity. So in the case of a binary file, it is extremely important that we interpret the correct data type while reading the file. Python provides a special module(s) for encoding and decoding of data for the binary file.

Text File	Binary File
Its Bits represent character.	Its Bits represent a custom data.
Less prone to get corrupt as change reflects as soon as made and can be undone.	Can easily get corrupted, corrupt on even a single bit change
Store only plain text in a file.	Can store different types of data (audio, text, and image) in a single file.
Widely used file format and can be opened in any text editor.	Developed for an application and can be opened in that application only.
Mostly .txt and .rtf are used as extensions to text files.	Can have any application defined extension.

File Operations

Opening a file: How do you write data to a file and read the data back from a file? You need to first create a file object that is associated with a physical file. This is called opening a file. The syntax for opening a file is: fileVariable = open(filename, mode) The open function returns a file object for filename. The mode parameter is a string that specifies how the file will be used (for reading or writing, Note: you can find the tables for more file modes) For example, the following statement opens a file named Scores.txt in **the current directory** for reading:
file= open("Scores.txt", "r") You can also use the absolute filename to open the file in Windows, as follows: file = open(r"c:\pybook\Scores.txt", "r") Example code for Opening a file: file = open("Scores.txt", "r") ## in a open(file_name,file_accessmode) here Scores.txt file name. ## r is the file access mode i.e read only print(file.read()) ## read() method will read the data from file ## print() method will prints the data from file object file.close() ## closing the file[will be explained in the closing file] **Absolute and relative paths** Absolute Path: An absolute path is a path that contains the entire path to the file or directory that you need to access. This path will begin at the home directory of your computer and will end with the file or directory that you wish to access. Example: /home/your-username/earth-analytics/data/field-sites/california/colorado/streams.csv (in Unix/Linux OS) G:\RGUKT\PUC\Sem2\index.html (in windows OS) Relative Path: A relative path is the path that (as the name sounds) is relative to the working directory location on your computer. Example: /data/field-sites/california/colorado/streams.csv (in Unix/Linux OS) PUC\Sem2\index.html (in windows OS)

The following picture contains absolute and relative paths for folders and files in the working directory C:\bacon



S.No	Mode	Description
1	**r (or) rt**	Opens a text file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
2	**r+ (or)rt+**	Opens a text file for both reading and writing. The file pointer placed at the beginning of the file.
3	**rb**	Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode.
4	**rb+**	Opens a file for both reading and writing in binary format. The file pointer placed at the beginning of the file.
5	**w (or) wt**	Opens a text file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
6	**w+ (or) wt+**	Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
7	**wb**	Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
8	**wb+**	Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
9	**a (or) at**	Opens a text file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
10	**a+ (or) at+**	Opens a text file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.
11	**ab**	Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
12	**ab+**	Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.
13	**x**	Opens a file for exclusive creation. If the file already exists, the operation fails.

Module 3 – File Handling Methods

File methods: A file object contains the methods for reading and writing data are as follows
read([number.int]): str ---> Returns the specified number of characters from the file. If the argument is omitted, the entire remaining contents in the file are read.
readline(): str---- > Returns the next line of the file as a string
readlines(): list ----> Returns a list of the remaining lines in the file.
write(s: str): None --- > Writes the string to the file.
close(): None --- > Closes the file
write(str): Write a string to the file. There is no return value. Due to buffering, the string may not actually show up in the file until the flush() or close() method is called.
writelines(sequence): Write a sequence of strings to the file. The sequence can be any iterable object producing strings, typically a list of strings. There is no return value. (The name is intended to match readlines(); writelines() does not add line separators.) After a file is opened for writing data, you can use the write method to write a string to the file

Here is writing demo into files

```
def main():
    # Open file for output
    outfile = open("student.txt", "w")
    # Write data to the file
    outfile.write("Ramesh\n")## \n is a newline character in the files
    outfile.write("Suresh\n")
    outfile.write("Rajesh\n")
    outfile.write("Sreenu\n")
    outfile.close()# Close the output file
main()# Calling main function
```

The program opens a file named student.txt using the w mode for writing data (line 3). If the file does not exist, the open function creates a new file. If the file already exists, the contents of the file will be overwritten with new data. You can now write data to the file. When a file is opened for writing or reading, a special marker called a file pointer is positioned internally in the file. A read or write operation takes place at the pointer‘s location. When a file is opened, the file pointer is set at the beginning of the file. When you read or write data to the file, the file pointer moves forward

Testing a File’s Existence:

To prevent the data in an existing file from being erased by accident, you should test to see if the file exists before opening it for writing. The isfile function in the os.path module can be used to determine whether a file exists.

Testing a File‘s Existence Code:

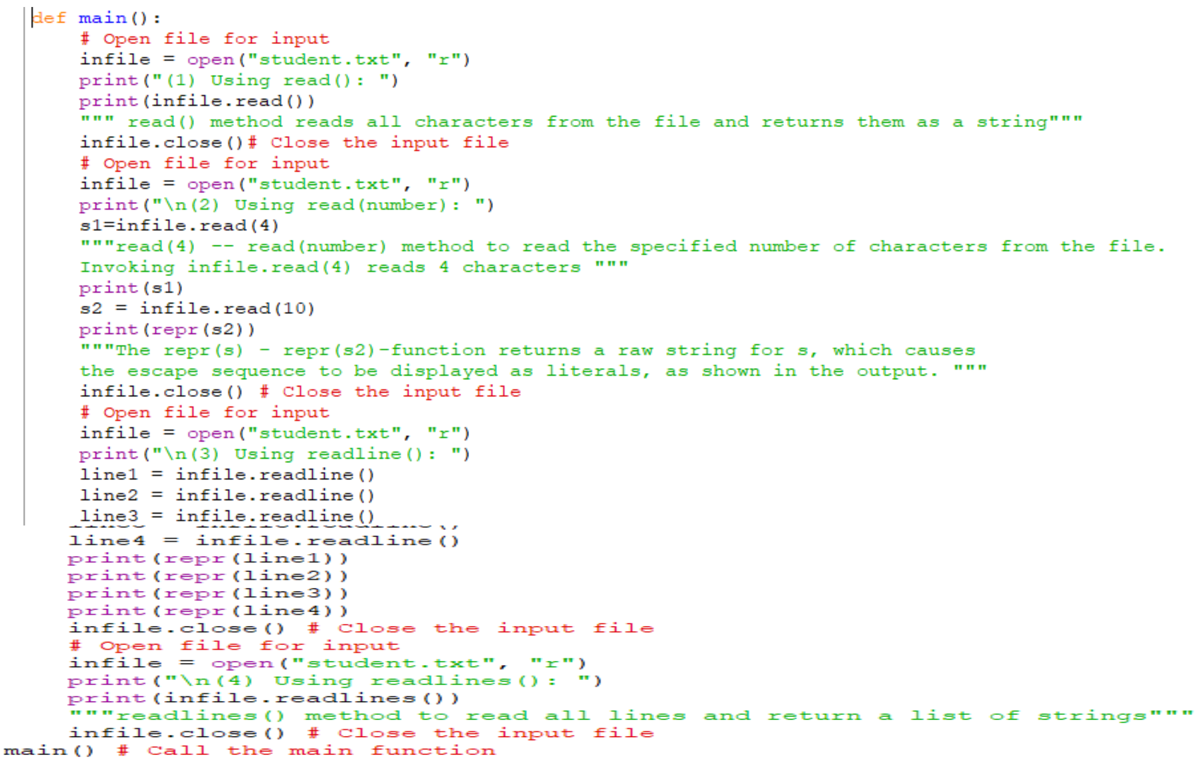
```
import os.path
if os.path.isfile("student.txt"):
    print("student.txt file existed")
else:
    print("student.txt DOES NOT exists")
```

Here isfile("students.txt.txt") returns True if the file students.txt exists in the current directory.

Reading Data:

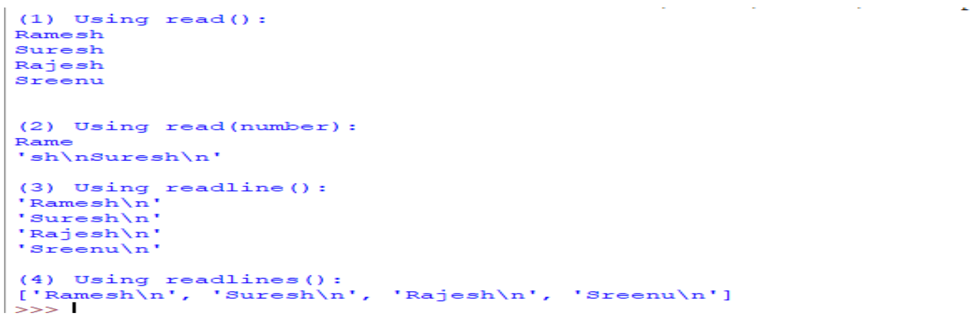
After a file is opened for reading data, you can use the read method to read a specified number of characters or all characters from the file and return them as a string, the readline() method to read the next line, and the readlines() method to read all the lines into a list of strings.

Reading Data demo code:



```
def main():
    # Open file for input
    infile = open("student.txt", "r")
    print("(1) Using read(): ")
    print(infile.read())
    """ read() method reads all characters from the file and returns them as a string"""
    infile.close()# Close the input file
    # Open file for input
    infile = open("student.txt", "r")
    print("\n(2) Using read(number): ")
    s1=infile.read(4)
    """read(4) -- read(number) method to read the specified number of characters from the file.
    Invoking infile.read(4) reads 4 characters """
    print(s1)
    s2 = infile.read(10)
    print(repr(s2))
    """The repr(s) - repr(s2)-function returns a raw string for s, which causes
    the escape sequence to be displayed as literals, as shown in the output. """
    infile.close() # Close the input file
    # Open file for input
    infile = open("student.txt", "r")
    print("\n(3) Using readline(): ")
    line1 = infile.readline()
    line2 = infile.readline()
    line3 = infile.readline()
    line4 = infile.readline()
    print(repr(line1))
    print(repr(line2))
    print(repr(line3))
    print(repr(line4))
    infile.close() # Close the input file
    # Open file for input
    infile = open("student.txt", "r")
    print("\n(4) Using readlines(): ")
    print(infile.readlines())
    """readlines() method to read all lines and return a list of strings"""
    infile.close() # Close the input file
main() # Call the main function
```

OUTPUT:



```
(1) Using read():
Ramesh
Suresh
Rajesh
Sreenu


(2) Using read(number):
Rame
'sh\nSuresh\n'

(3) Using readline():
'Ramesh\n'
'Suresh\n'
'Rajesh\n'
'Sreenu\n'

(4) Using readlines():
['Ramesh\n', 'Suresh\n', 'Rajesh\n', 'Sreenu\n']
>>>
```

Reading all data from file
Programs often need to read all data from a file. Here are two common approaches to accomplishing this task:
1. Use the read() method to read all data from the file and return it as one string.
2. Use the readlines() method to read all data and return it as a list of strings.
These two approaches are simple and appropriate for small files, but what happens if the file is so large that its contents cannot be stored in the memory? You can write the following loop to read one line at a time, process it, and continue reading the next line until it reaches the end of the file

Using while loop

```
infile = open("student.txt", "r")
line = infile.readline() # Read a line
while line != '':
    print(line)
    # Process the line here ...
    # Read next line
    line = infile.readline()
infile.close()
```

Using for loop

```
infile = open("student.txt", "r")
for line in infile:
    print(line)
infile.close()
```

Source code for copying a file
Illustrates a program that copies data from a source file to a target file and counts the number of lines and characters in the file.
Appending Data
You can use the mode to open a file for appending data to the end of an existing file.

```
def main():
    # Open file for appending data
    outfile = open("Info.txt", "a")
    outfile.write("\nPython is interpreted\n")
    outfile.close()# Close the file
main() # Call the main function
```

Writing and Reading Numeric Data

```
from random import randint
def main():
    # Open file for writing data
    outfile = open("Numbers.txt", "w")
    for i in range(10):
        outfile.write(str(randint(0, 9)) + " ")
    outfile.close()
    # Open file for reading data
    infile = open("Numbers.txt", "r")
    s = infile.read()
    numbers = [eval(x) for x in s.split()]
    for number in numbers: print(number, end = " ")
    infile.close() # Close the file
main() # Call the main function
```

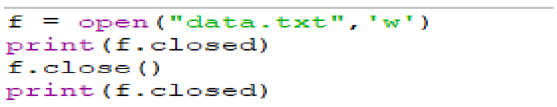
```
import os.path
import sys
def main():
    # Prompt the user to enter filenames
    f1 = input("Enter a source file: ").strip()
    f2 = input("Enter a target file: ").strip()
    # Check if target file exists
    if os.path.isfile(f2):
        print(f2 + " already exists")
        sys.exit()
    # Open files for input and output
    infile = open(f1, "r")
    outfile = open(f2, "w")
    # Copy from input file to output file
    countLines = countChars = 0
    ##initialize countLines and countChars
    for line in infile:
        ##increase countLines
        countLines += 1
        ##increase countChars
        countChars += len(line)
        outfile.write(line)
    print(countLines, "lines and", countChars, "chars copied")
    infile.close() # Close the input file
    outfile.close() # Close the output file
main() # Call the main function
```

To write numbers to a file, you must first convert them into strings and then use the write method to write them to the file. In order to read the numbers back correctly, separate them with whitespace characters, such as " " or \n.

Closing file:

When you ‘re done working, you can use the ***file.close()*** command to end things. What this does is close the file completely, terminating resources in use, in turn freeing them up for the system to deploy elsewhere. It‘s important to understand that when you use the ***file.close()*** method, any further attempts to use the file object will fail.

Syntax: fileobject.close()



```
f = open("data.txt",'w')
print(f.closed)
f.close()
print(f.closed)
```