Module 1

Chapter 1

INTRODUCTION TO PYTHON

Python was developed by **Guido van Rossum** at the National Research Institute for Mathematics and Computer Science in Netherlands during 1985-1990. Python is derived from many other languages, including ABC, Modula-3, C, C++, Algol-68, SmallTalk, Unix shell and other scripting languages. It is a general-purpose interpreted, interactive, object oriented, and high-level programming language. Python source code is available under the GNU General Public License (GPL) and it is now maintained by a core development team at the National Research Institute.

FEATURES OF PYTHON

a) Simple and easy-to-learn-

Python is a simple language with few keywords, simple structure and its syntax is also clearly defined. This makes Python a beginner's language.

b) Interpreted and Interactive –

Python is processed at runtime by the interpreter. We need not compile the program before executing it. The Python prompt interact with the interpreter to interpret the programs that we have written. Python has an option namely interactive mode which allows interactive testing and debugging of code.

c) Object-Oriented –

Python supports Object Oriented Programming (OOP) concepts that encapsulate code within objects. All concepts in OOPs like data hiding, operator overloading, inheritance etc, can be well written in Python. It supports functional as well as structured programming

d) Portable –

Python can run on a wide variety of hardware and software platforms and has the same interface on all platforms. All variants of Windows, Unix, Linux and Macintosh are to name a few.

e) Scalable –

Python provides a better structure and support for large programs than shell scripting. It can be used as a scripting language or can be compiled to bytecode (intermediate code that is platform independent) for building large applications.

f) Extendable –

You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient. It can be easily integrated with C, C++, COM, ActiveX, CORBA, and Java.

g) Dynamic –

Python provides very high-level dynamic data types and supports dynamic type checking. It also supports automatic garbage collection.

h) GUI Programming and Databases –

Python supports GUI applications that can be created and ported to many libraries and windows systems, such as Windows Microsoft Foundation Classes (MFC), Macintosh, and the X Window system of Unix. Python also provides interfaces to all major commercial databases.

i) **Broad Standard Library** –

Python's library is portable and cross platform compatible on UNIX, Linux, Windows and Macintosh. This helps in the support and development of a wide range of applications from simple text processing to browsers and complex games.

HOW TO RUN PYTHON

There are three different ways to start Python.

a) Using Interactive Interpreter

You can start Python from Unix, DOS, or any other system that provides you a command-line interpreter or shell window. Get into the command line of Python.

- For **Unix/Linux**, you can get into interactive mode by typing **$python or python%.**
- For **windows/Dos** it is **C:>python**.

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

```
$ python
Python 3.7 (default, Mar 27, 2019, 18:11:38) [GCC 5.1.1 20150422 (Red Hat 5.1.1-1)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
```

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

```
>>> print ("Programming in Python!")
```

The result will be as given below

```
Programming in Python!
```

b) Script from the Command Line

This method invokes the interpreter with a script parameter which begins the execution of the script and continues until the script is finished. When the script is finished, the interpreter is no longer active. A Python script can be executed at command line by invoking the interpreter on your application, as follows.

- For **Unix/Linux** it is **$python script.py or python; script.py.**
- For **Windows/ Dos** it is **C:>python script.py**

Let us write a simple Python program in a script. Python files have extension .py. Type the following source code in a first.py file.

```
print("Programming in Python!")
```

Now, try to run this program as follows.

```
$ python first.py
```

This produces the following result:

```
Programming in Python!
```

c) Integrated Development Environment

You can run Python from a Graphical User Interface (GUI) environment as well, if you have a GUI application on your system that supports Python. IDLE is the Integrated Development Environment (IDE) for UNIX and Python Win is the first Windows interface for Python.

IDENTIFIERS

A Python identifier is a name used to **identify a variable**, function, class, module or any other object. Python is **case sensitive** and hence uppercase and lowercase letters are considered distinct. The following are the rules for naming an identifier in Python.

- Identifiers can be a combination of letters in lowercase (a to z) or uppercase (A to Z) or digits (0 to 9) or an underscore (). For example Total and total is different.
- Reserved keywords (explained in Section 1.4) cannot be used as an identifier.
- Identifiers cannot begin with a digit. For example 2more, 3times etc. are invalid identifiers.
- Special symbols like @, !, #, $, % etc. cannot be used in an identifier. For example sum@, #total are invalid identifiers.
- Identifier can be of any length.

Some examples of valid identifiers are total, max_mark, count2, Student etc. Here are some naming conventions for Python identifiers.

- Class names start with an uppercase letter. All other identifiers start with a lowercase letter. Eg: Person
- Starting an identifier with a single leading underscore indicates that the identifier is private. Eg: _sum
- Starting an identifier with two leading underscores indicates a strongly private identifier. Eg: _sum
- If the identifier also ends with two trailing underscores, the identifier is a language defined special name. foo__

RESERVED KEYWORDS

These are keywords reserved by the programming language and prevent the user or the programmer from using it as an identifier in a program. There are 33 keywords in Python 3.3. This number may vary with different versions. To retrieve the keywords in Python the following code can be given at the prompt. All keywords except True, False and None are in lowercase. The following list in Table 1.1 shows the Python keywords.

```
>>> import keyword
>>> print (keyword.kwlist)
['false', 'None', 'True', 'and', 'as', 'assert', 'break', 'class',
continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda',
'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', yield']
```

Python Keywords

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

VARIABLES

Variables are reserved memory locations to store values. Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals or characters in these variables.

Python variables do not need explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable. The equal sign (=) is used to assign values to variables. The operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the value stored in the variable.

Example Program
```
a = 100          # An integer assignment
b = 1000.0       # A floating point
name = "John"    # A string
print (a)
print (b)
print (name)
```

Here, 100, 1000.0 and "John" are the values assigned to a, b, and name variables, respectively. This produces the following result

```
100
1000.0
John
```

Python allows you to assign a single value to several variables simultaneously. For example

```
a = b = c = 1
```

Here, an integer object is created with the value 1, and all three variables are assigned to the same memory location. You can also assign multiple objects to multiple variables. For example

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

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

COMMENTS IN PYTHON

Comments are very important while writing a program. It describes what the source code has done. Comments are for programmers for better understanding of a program. In Python, we use the **hash (#)** symbol to start writing a comment. A hash sign (#) that is not inside a string literal begins a comment. All characters after the # and up to the end of the physical line are part of the comment. It extends up to the newline character. Python interpreter ignores comment. Example Program

```
>>>#This is demo of comment
>>>#Display Hello
>>>print("Hello")
```

This produces the following result: Hello

For multiline comments one way is to use (#) symbol at the beginning of each line. The following example shows a multiline comment. Another way of doing this is to use **triple quotes, either ’’’or """.**

Example 1

```
>>>#This is a very long sentence
>>>#and it ends after
>>>#Three lines
```

Example 2

```
>>>""" This is a very long sentence
>>>and it ends after
>>>Three lines"""
```

Both Example 1 and Example 2 given above produces the same result.

INDENTATION IN PYTHON

Most of the programming languages like C, C++ and Java use braces {} to define a block of code. Python uses **indentation**. A code block (body of a function, loop etc.) starts with indentation and ends with the first unindented line. The amount of indentation can be decided by the programmer, but it must be consistent throughout the block. Generally **four whitespaces** are used for indentation and is preferred over tabspace. For example

```
if True :
    print("Correct")
else:
    print ("Wrong")
```

But the following block generates error as indentation is not properly followed.

```
if True:
    print ("Answer")
    print("Correct")
else:
    print("Answer")
  print("Wrong")
```

MULTI-LINE STATEMENTS

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. are other kinds of statements which will be discussed in coming Chapters. In Python, end of a statement is marked by a newline character. But **we can make a statement extend over multiple lines with the line continuation character(\)**. For example:

```
grand_total = first_item + \
          second_item + \
          third_item
```

Statements contained within the [ ], {}, or ( ) brackets do not need to use the line continuation character. For example

```
months = ['January', 'February', 'March', 'April',
    ''May', 'June', 'July', 'August', 'September',
    October', 'November', 'December']
```

We could also put multiple statements in a single line using semicolons, as follows

```
a = 1; b = 2; C = 3
```

MULTIPLE STATEMENT GROUP (SUITE)

A group of individual statements, which make **a single code block is called a suite** in Python. Compound or complex statements, such as **if, while, def, and class require a header line and a suite.** Header lines begin the statement (with the keyword) **and terminate with a colon (:)** and are followed by one or more lines which make up the suite. For example

```
if expression :
    suite
elif expression :
    suite
```

```
else :
        suite
```

QUOTES IN PYTHON

Python accepts single (‘), double (") and triple (''' or """) quotes to denote string literals. The same type of quote should be used to start and end the string. The triple quotes are used to span the string across multiple lines. For example, all the following are legal.

```
word = 'single word'
sentence = "This is a short sentence."
paragraph = """ This is a long paragraph. It consists of
                several lines and sentences. """
```

INPUT, OUTPUT AND IMPORT FUNCTIONS

Displaying the Output

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

Example 1

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

This will produce the following output on the screen.

```
Learning Python is fun and enjoy it.
```

Example 2

```
>>>a=2
>>>print("The value of a is", a)
```

This will produce the following output on the screen.

```
The value of a is 2
```

By default **a space is added after the text** and before the value of variable a. The syntax of print function is given below.

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

Here, **objects** are the **values to be printed**. **Sep** is the **separator used between the values. Space** character is the default separator. **end** is printed **after printing all the values**. The **default value for end is the new line**, **file is** the **object where the values are printed** and its default value is **sys.stdout (screen). Flush determines whether the output stream needs to be flushed for any waiting output**. A True value forcibly flushes the stream. The following shows an example for the above syntax.

Example

```
>>>print(1,2,3,4)
>>>print(1,2,3,4, sep='+')
>>> print (1,2,3,4, sep='+', end='')
```

The output will be as follows.

```
1 2 3 4
1+2+3+4
1+2+3+4 %
```

The output can also be formatted to make it attractive according to the wish of a user.

Reading the Input

Python provides **two built-in functions** to read a line of text from standard input, which by default comes from the keyboard. These functions are **raw_input and input**. The raw_input function is not supported by Python 3.

raw_input function

The raw_input([prompt]) function reads **one line** from standard input and returns it as a string **(removing the trailing newline**). This prompts you to enter any string and it would display same string on the screen.

Example

```
str = raw_input ("Enter your name:");
print("Your name is : ", str)
```

Output

```
Enter your name: Collin Mark
Your name is : Collin Mark
```

input function

The input([prompt]) function is equivalent to raw_input, except that **it assumes the input is a valid Python expression and returns the evaluated result to yo**u.

Example 1

```
n = input ("Enter a number:");
print ("The number is : ", n)
```

Output

```
Enter a number: 5
The number is : 5
```

Example 2

```
n = input ("Enter an expression:");
print("The result is : ", n)
```

Output

```
Enter an expression: 5*2
The result is : 10
```

Import function

When the program grows bigger or when there are segments of code that is frequently used, it can be stored in different modules. A module is a file containing Python definitions and statements. Python modules have a filename and end with the extension .py. **Definitions inside a module can be imported to another module or the interactive interpreter in Python**. We use the **import keyword** to do this. For example, we can import the math module by typing in import math.

```
>>> import math
>>> math.pi
3.141592653589793
```

OPERATORS

Operators are the constructs which can manipulate the value of operands. Consider the expression a = b + c. Here a, b and c are called the operands and +, = are called the operators. There are different types of operators. The following are the types of operators supported by Python.

• Arithmetic Operators
• Comparison (Relational) Operators
• Assignment Operators
• Logical Operators
• Bitwise Operators
• Membership Operators
• Identity Operators

1 Arithmetic Operators

Arithmetic operators are used for performing basic arithmetic operations. The following are the arithmetic operators supported by Python. Table 1.2 shows the arithmetic operators in Python

Operator

Table 1.2 Arithmetic Operators in Python

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

Example Program

```
a, b, c = 10, 5, 2
print("Sum=", (a+b))
print("Differences", (a-b))
print("Product=", (a*b))
print("Quotient=", (a/b))
print ("Remainder=", (b&c))
print("Exponent=", (b**2))
print("Floor Division=", (b//c))
```

Output

```
Sum= 15
Difference= 5
Product= 50
Quotient= 2
Remainders =1
Exponent= 25
Floor Division= 2
```

2. Comparison Operators

These operators compare the values on either sides of them and decide the relation among them. They are also called relational operators. Python supports the following relational operators. Table 1.3 shows the comparison or relational operators in Python.

Operator	Description
==	If the values of two operands are equal, then the condition becomes true.
!=	If values of two operands are not equal, then condition becomes true.
>	If the value of left operand is greater than the value of right operand, then condition becomes true.
<	If the value of left operand is less than the value of right operand, then condition becomes true.
>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.
<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.

Example Program

```
a, b=10,5
print ("a==b is", (a==b))
```

print ("a!=b is", (a!=b))

print ("a>b is", (a>b))

print ("asb is", (a<b))

print ("a>=b is", (a>=b))

print ("a<=b is", (a<=b))

Output

a==b is False

a !=b is True

a>b is True

a<b is False

a>=b is True

a<=b is False

3. Assignment Operators

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

Table: Assignment Operators

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

The assignment operator = is used to assign values or values of expressions to a variable. Example: a = b+ c. For example c+=a is equivalent to c=c+a. Similarly c-=a is equivalent to C=C-a, c*ra is equivalent to c=c*a, c/=a is equivalent to c=c/a, c%=a is equivalent to c=c%a, c**=a is equivalent to c=c**a and c//=a is equivalent to c=c//a.

Example Program

a, b=10,5

a+=b

```
print (a)
a, b=10,5
a-=b
print (a)
a, b=10,5
a*=b
print (a)
a,b=10,5
a/=b print (a)
b, c=5,2
b%=c
print (b) b,
c=5,2
b**=c
print (b)
b, c-5,2
b//=C
print (b)
```

Output

```
15
5
50
2
1
25
2
```

4 Bitwise Operators

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

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

		specified by the right operand.
>>	Binary Right Shift	The left operand's value is moved right by the number of bits specified by the right operand.

Let a - 60 (0011 1100 in binary) and b- 2 (0000 0010 in binary)

Example Program

```
a,b=60,2
print (a&b)
print (a|b)
print (a^b)
print (~a)
print(a>>b)
print (a<<b)
```

Output

```
0
62
62
-61
15
240
```

Consider the first print statement a&b. Here a=0011 1100 and b=0000 0010. When a bitwise and is performed, the result is 0000 0000. This is 0 in decimal. Similar is the case for all the print statements given above.

5. Logical Operators

Table shows the various logical operators supported by Python language.

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

Example Program

```
a, b, c, d=10,5,2,1
print((a>b) and (c>d))
print((a>b) or (d>c))
print (not (a>b))
```

Output

```
True
True
False
```

6. Membership Operators

Python's membership operators test for membership in a sequence, such as strings, lists, or tuples. There are two membership operators as explained below in Table.

Table Membership Operators

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

Example Program

```
s='abcde'
print('a' in s)
print('f' in s)
print('f'not in s)
```

Output

```
True
False
True
```

7. Identity Operators

Identity operators **compare the memory locations** of two objects. There are two identity operators as shown in below Table.

Table Identity Operators

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

Example Program

```
a, b, c=10,10,5
print (a is b)
print (a ia c)
print (a is not b)
```

Output

```
True
False
False
```

Operator Precedence

The following Table lists all operators from highest precedence to lowest. Operator associativity determines the order of evaluation, when they are of the same precedence, and are not grouped by parenthesis. An operator may be left-associative or right-associative. In left-associative, the operator falling on the left side will be evaluated first, while in right associative, operator falling on the right will be evaluated first. In Python,'=' and '**' are right-associative while all other operators are left-associative.

Table Operator Precedence

Operator	**Description**
**	Exponentiation (raise to the power)
~,+,-	Complement, unary plus and minus
*, /, %, //	Multiply, divide, modulo and floor division
+, -	Addition and subtraction
>>, <<	Right and left bitwise shift
&	Bitwise AND
^, |	Bitwise exclusive 'OR' and regular 'OR'
<=, < >, >=	Comparison operators
<>,= =, !=	Equality operators
=, %=, /=, //=, -=, +=, *=, **=	Assignment operators
is, is not	Identity operators
in, not in	Membership operators
not, or, and	Logical operators

Chapter 2

Data Types and Operations

The data stored in memory can be of many types. For example, a person's name is stored as alphabets, age is stored as a numeric value and his or her address is stored as alphanumeric characters. Python has the following standard data types.

- Numbers
- String
- List
- Tuple
- Set
- Dictionary

1. NUMBERS

Number data types store **numeric values**. Number objects are created when you assign a value to them. For example a = 1, b = 20. You can also delete the reference to a number object by using the **del** statement.

The syntax of the del statement is as follows.

del variable1[, variable2 [, variable3[...., variableN]]]]

You can delete a single object or multiple objects by using the del statement. For example

```
del a
del a, b
```

Python supports four different numerical types.

- int (signed integers)
- long (long integers, they can also be represented in octal and hexadecimal)
- float (floating point real values)
- complex (complex numbers)

Integers can be of any length, it is only limited by the memory available. A floating point number is accurate up to 15 decimal places. Integer and floating points are separated by decimal points. 1 is integer, 1.0 is floating point number. Complex numbers are written in the form, x + yj, where x is the real part and y is the imaginary part.

Example Program

```
a, b, c, d=1,1.5,231456987,2+9j
print("a",a)
print("b ", b)
print("c=",c)
print("d",d)
```

Output

```
a=1
b= 1.5
c=231456987
d=(2+9j)
```

1.1 Mathematical Functions

Python provides various built-in mathematical functions to perform mathematical calculations. The following Table provides various mathematical functions and its purpose. For using the below functions, all functions except abs(x), max(x1,x2,... xn), min(x1,x2,... xn), round(xl.nl) and pow(x,y) need to import the **math** module because these functions reside in the math module. The math module also defines two mathematical constants pi and e.

Table 2.1 Mathematical Functions

Function	Description
abs(x)	Returns the absolute value of x.
sqrt(x)	Finds the square root of x.
ceil(x)	Finds the smallest integer not less than x.
floor(x)	Finds the largest integer not greater than X.
pow(x,y)	Finds x raised to y.
exp(x)	Returns e^x ie exponential of x.
fabs(x)	Returns the absolute value of x.
log(x)	Finds the natural logarithm of x for >0.
log10(x)	Finds the logarithm to the base 10 for x>0.
max(x1,x2,...xn)	Returns the largest of its arguments.
min(x1,x2,...,xn)	Returns the smallest of its arguments.

round(x,[n])	In case of decimal numbers, x will be rounded to n digits.
modf(x)	Returns the integer and decimal part as a tuple. The integer part is returned as a decimal

Example Program

```
import math
print (“Absolute value of -120:", abs(-120))
print("Square root of 25:", math.sqrt (25))
print("Ceiling of 12.2:", math.ceil(12.2))
print("Floor of 12.2:", math.floor (12.2))
print("2 raised to 3:", pow(2,3) )
print("Exponential of 3", math.exp (3) )
print("Absolute value of -123:", math.fabs (-123))
print("Natural Logarithm of 2:", math.log(2))
print("Logarithm to the Base 10 of 2:", math.log10 (2))
print("Largest among 10, 4, 2:", max (10,4,2))
print("Smallest among 10,4, 2:", min(10.4.2) )
print("12.6789 rounded to 2 decimal places:", round (12.6789,2))
print("Decimal part and integer part of 12.090876:", math,modf (12.090876))
```

Output

```
Absolute value of -120: 120
Square root of 25: 5.0
Ceiling of 12.2: 13
Floor of 12.2: 12
2 raised to 3: 8
Exponential of 3: 20.085536923187668
Absolute value of -123: 123.0
Natural Logarithm of 2: 0.6931471805599453
Logarithm to the Base 10 of 2: 0.3010299956639812
Largest among 10, 4, 2: 10
Smallest among 10,4, 2: 2
12.6789 rounded to 2 decimal places: 12.68
Decimal part and integer part of 12.090876: 10.09087599999999973, 12.0)
```

1.2 Trigonometric Functions

There are several built-in trigonometric functions in Python. The function names and its purpose are listed in Table.

Table Trigonometric Functions

Function	Description
sin(x)	Returns the sine of x radians.
cos(x)	Returns the cosine of x radians.

tan(x)	Returns the tangent of x radians.
asin(x)	Returns the arc sine of x, in radians.
acos(x)	Returns the arc cosine of x, in radians.
atan(x)	Returns the arc tangent of x, in radians.
atan2(y,x)	Returns atan (y / x), in radians.
hypot(x,y)	Returns the Euclidean form, sqrt(x*x + y*y).
degrees(x)	Converts angle x from radians to degrees.
radians(x)	Converts angle x from degrees to radians.

Example

```
import math
print("Sin (90):", math.sin(90))
print ("Cos (90):", math.cos (90))
print ("Tan (90):", math. tan (90))
print("asin(1):", math.asin(1))
print (acos (1):", math.acos (1))
print("atan (1):", math.atan (1))
print("atan2 (3,2):", math.atan2 (3,2))
print("Hypotenuse of 3 and 4:", math.hypot (3, 4))
print("Degrees of 90:", math. degrees (90)
print("Radians of 90:", math.radians (90))
```

Output

```
Sin(90): 0.893996663601
Cos (90): -0.448073616129
Tan (90): -1.99520041221
asin(1): 1.57079632679
acos (1): 0.0
atan (1): 0.785398163397
atan2 (3,2): 0.982793723247
Hypotenuse of 3 and 4: 5.0
Degrees of 90: 5156.62015618
Radians of 90: 1.57079632679
```

1.3 Random Number Functions

There are several random number functions supported by Python. Random numbers have applications in research, games, cryptography, simulation and other applications. Table shows the commonly used random number functions in Python. For using the random number functions, we need to import the module **random** because all these functions reside in the random module.

Table Random Number Functions

Function	Description

choice(sequence)	Returns a random value from a sequence like list, tuple, string etc.
shuffle(list)	Shuffles the items randomly in a list. Returns none.
random()	Returns a random floating-point number which lies between **0 and 1.**
randrange ([start ,] stop [, step])	Returns a randomly selected number from a range where start shows the starting of the range, stop shows the end of the range and step decides the number to be added to decide a random number.
seed([x])	Gives the starting value for generating a random number. Returns **none**. This function is called before calling any other random module function.
uniform(x, y)	Generates a random floating-point number n such that n>x and n<y,

Example

```
import random
s= 'abcde'
print("choice (abcde):", random.choice(s) )
list = [10,2,3,1,8,19]
random. shuffle (list)
print("shuffle (list):", list)
print("random.seed (20):", random.seed (20))
print ("random(): ", random.random() )
print("uniform (2,3):", random.uniform(2,3) )
print("randrange (2,10,1):", random.randrange (2,10,1))
```

Output

```
choice (abcde): d
shuffle (list): [3, 1, 2, 10, 19, 8]
random.seed (20): None
random():  0.9056396761745207
uniform (2,3): 2.6862541570267027
randrange (2,10,1): 4
```

2. STRINGS

Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows for either pairs of single or double quotes. Subsets of strings can be taken using the **slice** operator (and :) with indexes starting at 0 in the beginning of the string and ending at -1.The plus (+) sign is the string **concatenation** operator and the asterisk (*) is the **repetition** operator. The % operator is used for string **formatting**.

Example Program

```
str = 'Welcome to Python Programming'
print (str)                         # Prints complete string
print (str[0])                      # Prints first character of the string
print (str[11 : 17])                # Prints characters starting from 11th to 17th
print (str [11:])             # Prints string starting from 11th character
print (str * 2 )              # Prints string two times
print (str + " Session")      # Prints concatenated string
```

Output

```
Welcome to Python Programming
W
Python
Python Programming
Welcome to Python Programming Welcome to Python Programming
Welcome to Python Programming Session
```

2.1 Escape Characters

An escape character is a character that gets interpreted when placed in **single or double quotes**. They are represented using **backslash notation**. These characters are non printable. The following Table shows the most commonly used escape characters and their description.

Table Escape Characters

Escape Characters	Description
\a	Bell or alert
\b	Backspace
\n	Newline
\s	Space
\t	Tab
\v	Vertical Tab

2.2 String Formatting Operator

This operator is unique to strings and is similar to the formatting operations of the printf() function of the programming language. The following Table shows various format symbols and their conversion.

Table Format Symbols and their Conversion

Format Symbols	Conversion
%c	Character
%s	String
%i	Signed Decimal Integer
%d	Signed Decimal Integer
%u	Unsigned Decimal Integer
%o	Octal Integer
%x	Hexadecimal Integer (lowercase letters)
%X	Hexadecimal Integer (uppercase letters)
%e	Exponential notation with lowercase letter ‘e’
%E	Exponential notation with uppercase letter 'E'

%f	Floating point real number

The following example shows the usage of various string formatting functions.

Example Program

```
#Demo of String Formatting Operators
print("The first letter of %s is %c" %('apple', 'a'))
print("The sum=%d" %(-12))
print("The sum=%i" %(-12))
print("The sum=%u" %(12))
print("%o is the octal octal equivalent of %d" %(8,8))
print("%x is the hexadecimal equivalent of %d" %(10,10))
print("%X is the hexadecimal equivalent of %d" %(10,10))
print("%e is the exponential equivalent of %f" %(10.98765432, 10.98765432))
print("%E is the exponential equivalent of %f" %(10.98765432,10.98765432))
print("%.3f" %(32.1274))
```

Output

```
The first letter of apple is a
The sum= -12
The sum= -12
The sum=12
10 is the octal octal equivalent of 8
a is the hexadecimal equivalent of 10
A is the hexadecimal equivalent of 10
1.098765e+01 is the exponential equivalent of 10.987654
1.098765E+01 is the exponential equivalent of 10.987654
32.127
```

2.3 String Formatting Functions

Python includes a large number of built-in functions to manipulate strings.

1. len(string)- Returns the length of the string

Example Program

```
 #Demo of len(string)
s='Learning Python is fun!'
print("Length of ", s, " is", len(s) )
```

Output

```
Length of Learning Python is fun! is 23
```

2. lower() - Returns a copy of the string in which all uppercase alphabets in a string are converted to lowercase alphabets.

Example Program

```
#Demo of lower()
s='Learning Python is fun!'
print (s.lower())
```

Output

```
learning python is fun!
```

3. upper() - Returns a copy of the string in which all lowercase alphabets in a string are converted to uppercase alphabets.

Example Program

```
#Demo of upper
s='Learning Python is fun!'
print (s. upper())
```

Output

```
LEARNING PYTHON IS FUN!
```

4. swapcase() - Returns a copy of the string in which the case of all the alphabets are swapped. ie. all the lowercase alphabets are converted to uppercase and vice versa.

Example Program

```
#Demo of swapcase ()
s='LEARNing PYTHON is fun!'
print (s. swapcase( ))
```

Output

```
learnING Python IS FUN!
```

5. capitalize() - Returns a copy of the string with only its first character capitalized.

Example Program

```
#Demo of capitalize()
s='learning Python is fun!'
print (s.capitalize())
```

Output

```
Learning python is fun!
```

6. title() - Returns a copy of the string in which first character of all the words are capitalized.

Example Program

```
#Demo of title()
s='learning Python is fun!'
print (s.title())
```

Output

```
Learning Python Is Fun!
```

7. lstrip() - Returns a copy of the string in which all the characters have been stripped(removed) from the beginning. The default character is whitespaces.

Example Program

```
#Demo of lstrip ()
s='        learning Python is fun!'
print (s.lstrip())
s='*******learning Python is fun!'
print (s.lstrip('*'))
```

Output

```
learning Python is fun!
learning Python is fun!
```

8. rstrip() - Returns a copy of the string in which all the characters have been stripped(removed) from the beginning. The default character is whitespaces.

Example Program

```
#Demo of rstrip
s='learning Python is fun!      '
print (s.rstrip())
s='learning Python is fun! ******* '
print (s.rstrip ('*'))
```

Output

```
learning Python is fun!
learning Python is fun!
```

9. strip() - Returns a copy of the string in which all the characters have been stripped (removed) from the beginning and end. It performs both lstrip) and rstrip (). The default character is whitespaces. Example Program

```
#Demo of strip()
s='      learning Python is fun!    '
print (s.strip())
s='*****learning Python is fun! ******'
print(s.strip('*'))
```

Output

```
learning Python is fun!
learning Python is fun!
```

10. max(str) - Returns the maximum alphabetical character from string str.

Example Program

```
#Demo of max (str)
s='learning Python is fun'
print("Maximum character is :", max (s) )
```

Output

```
Maximum character is : y
```

11. min(str) - Returns the minimum alphabetical character from string atr.

Example Program

```
#Demo of min(str)
s='learning-Python-is-fun'
print ('Minimum character is :', min(s) )
```

Output

```
Minimum character is : -
```

12. replace(old, new [,max]) - Returns a copy of the string with all the occurrences of substring old is replaced by new. The max is optional and if it is specified, the first occurrences specified in max are replaced.

Example Program

```
# Demo of replace(old, new[, max] )
s="This is very new. This is good"
print(s.replace('is', 'was'))
print(s.replace('is', 'was', 2))
```

Output

```
Thwas was very new. Thwas was good
Thwas was very new. This is good
```

13. center(width, fillchar) - Returns a string centered in a length specified in the width variable. Padding is done using the character specified in the fillchar. Default padding is space.

Example Program

```
# Demo of center (width, fillchar)
s="This is Python Programming"
print (s.center (30, '*'))
print (s.center (30))
```

Output

```
**This is Python Programming**
This is Python Programming
```

14. ljust(width[,fillchar]) - Returns a string left-justified in a length specified in the width variable. Padding is done using the character specified in the fillchar. Default padding is space.

Example Program

```
# Demo of ljust (width[, fillchar])
s="This is Python Programming"
print (s.ljust (30, '*'))
print (s.ljust (30) )
```

Output

```
This is Python Programming****
This is Python Programming
```

15. rjust(width[,fillchar]) - Returns a string right-justified in a length specified in the width variable. Padding is done using the character specified in the fillchar. Default padding is space.

Example Program

```
# Demo of rjust (width, fillchar])
s="This is Python Programming"
print (s.rjust (30, '*'))
print (s.rjust (30))
```

Output

```
****This is Python Programming
This is Python Programming
```

16. zfill(width) - The method zfill pads string on the left with zeros to fill width.

Example Program

```
# Demo of zfill(width)
s="This is Python Programming"
print (s.zfill (30) )
```

Output

```
0000This is Python Programming
```

17. count(str, beg=0,end=len(string)) - Returns the number of occurrences of str in the range beg and end.

Example Program

```
# Demo of count (str, beg=0, end-len (string))
s="This is Python Programming"
```

```
print (s.count('i', 0,10))
print (s.count('i', 0,25))
```

Output

```
2
3
```

18. find(str,beg=0,end=len(string)) - Returns the index of str if str occurs in the range beg and end and returns -1 if it is not found.

Example Program

```
# Demo of find (str, beg-0, end-len(string))
s="This is Python Programming"
print (s.find('thon', 0,25))
print (s.find('thy'))
```

Output

```
10
-1
```

19. rfind(str, beg=0,end=len(string)) - Same as find, but searches backward in a string.

Example Program

```
# Demo of rfind (str, beg=0, end=len (string)
s="This is Python Programming"
print (s.rfind('thon', 0,25))
print (s.rfind('thy'))
```

Output

```
10
-1
```

20. index(str, beg=0,end=len(string)) - Same as that of find but raises an exception if the item is not found.

Example Program

```
# Demo of index (str, beg=0, end-len(string))
s="This is Python Programming"
print (s.index ('thon', 0,25))
print (s. index ('thy'))
```

Output

```
10
Traceback (most recent call last):
File "main.py", line 4, in <module>
print s.index (thy')
ValueError: substring not found
```

21. rindex(str,beg=0,end=len(string)) - Same as index but searches backward in a string.

Example Program

```
# Demo of rindex (str, beg=0, end=len (string)
s="This is Python Programming"
print (s.rindex ('thon', 0,25))
print(s.rindex ('thy'))
```

Output

```
10
Traceback (most recent call last):
```

```
File "main.py", line 4, in <module>
print s.index ('thy')
ValueError: substring not found
```

22. startswith(suffix, beg=0,end=len(string)- It returns True if the string begins with the specified suffix, otherwise return false.

Example Program

```
# Demo of startswith(suffix, beg=0, end-len (string))
s="Python programming is fun"
print (s.startswith('is', 10,21))
s="Python programming is fun"
print (s.startswith('is',19,25))
```

Output

```
False
True
```

23. endswith(suffix, beg=0,end=len(string))- It returns True if the string ends with the specified suffix, otherwise return false.

Example Program

```
# Demo of endswith(suffix, beg=0, end-len (string))
s="Python programming is fun"
print (s.endswith('is', 10,21))
s="Python programming is fun"
print (s.endswith('is', 0,25))
```

Output

```
True
False
```

24. isdecimal( )- Returns True if a unicode string contains only decimal characters and False otherwise. To define a string as unicode string, prefix 'u' to the front of the quotation marks.

Example Program

```
# Demo of isdecimal()
s=u"This is Python 1234"
print (s.isdecimal() )
s=u"123456"
print (s.isdecimal( ))
```

Output

```
False
True
```

25. isalpha() - Returns True if string has at least 1 character and all characters are alphabetic and False otherwise.

Example Program

```
# Demo of isalpha()
s="This is Python1234"
print(s.isalpha())
s="Python"
print (s.isalpha())
```

Output

```
False
```

```
True
```

26. isalnum()- Returns True if string has at least 1 character and all characters are alphanumeric and False otherwise.

Example Program

```
# Demo of isalnum ()
s="*** Python1234"
print (s.isalnum())
s="Python1234"
print (s.isalnum())
```

Output

```
False
True
```

27. isdigit() - Returns True if string contains only digits and False otherwise.

Example Program

```
# Demo of isdigit()
s="*** Pythoni234"
print (s.isdigit())
s="123456"
print (s. isdigit())
```

Output

```
False
True
```

28. islower() - Returns True if string has at least 1 cased character and all cased characters are in lowercase and False otherwise.

Example Program

```
# Demo of islower()
s="Python Programming"
print (s.islower())
s="python programming"
print (s.islower())
```

Output

```
False
True
```

29. isupper() - Returns True if string has at least one cased character and all cased characters are in uppercase and False otherwise.

Example Program

```
# Demo of isupper()
s="Python Programming"
print (s.isupper())
s="PYTHON PROGRAMMING"
print (s.isupper( ))
```

Output

```
False
True
```

30. isnumeric( ) - Returns True if a unicode string contains only numeric characters and False otherwise.

Example Program

```
# Demo of isnumeric()
s=u" Python Programming1234"
print (s.isnumeric())
s="12345"
print (s.isnumeric())
```

Output

```
False
True
```

31. isspace() - Returns True if string contains only whitespace characters and False otherwise

Example Program

```
# Demo of isspace ()
s="Python Programming"
print (s.isspace())
s="   "
print (s. isspace())
```

Output

```
False
True
```

32. istitle() - Returns True if string is properly "titlecased" and False otherwise. Title case means each word in a sentence begins with uppercase letter.

Example Program

```
#Demo of istitle()
s="Python programming is fun"
print (s.istitle())
s="Python Programming Is Fun"
print (s.istitle())
```

Output

```
False
True
```

33. expandtabs(tabsize=8) - It returns a copy of the string in which tab characters ie.'\t are expanded using spaces using the given tabsize. The default tabsize is 8.

Example Program

```
#Demo of expandtabs (tabsize)
s="Python\tprogramming\tis\tfun"
print( s.expandtabs())
s="Python\tprogramming \tis\tfun"
print (s.expandtabs (10))
```

Output

```
Python          programming     is      fun
Python                  programming             is                  fun
```

34. join(seq) - Returns a string in which the string elements of sequence have been joined by a separator.

Example Program

```
# Demo of join(seq)
s="-"
```

```
        seq=("Python", "Programming")
        print (s.join(seq))
        s="*"
        seq= ("Python", "Programming")
        print (s.join(seq))
```
Output
```
        Python-Programming
        Python*Programming
```
35. split(str="", num=string.count(str)) - Returns a list of all the words in the string, using str as the separator (splits on all whitespace if left unspecified), optionally limiting the number of splits to num.

Example Program
```
        #Demo of split (str "", num=string.count (str))
        s="Python programming is fun"
        print (s.split(' ' ))
        s="Python*programming*is*fun"
        print (s.split('*'))
        s="Python*programming*is*fun"
        print (s.split('*',2))
```
Output
```
        ['Python', 'programming', 'is', 'fun']
        ['Python', programming', 'is', 'fun']
        ['Python', 'programming', 'is*fun']
```
36. splitlines(num=string.count('\n') - Splits string at all (or num) NEWLINEs and returns a list of each line with NEWLINEs removed. If num is specified, it is assumed that line breaks need to be included in the lines.

Example Program
```
        #Demo of splitlines (num=string.count('\n'))
        s="Python\nprogramming\nis\nfun"
        print (s.splitlines())
        print (s.splitlines (0))
        print (s.splitlines (1))
```
Output
```
        ['Python', 'programming', 'is', 'fun']
        ['Python', 'programming', 'is', 'fun']
        ['Python\n', 'programming\n', 'is\n', 'fun']
```

3. LIST

List is an ordered sequence of items. It is one of the most used data type in Python and is very flexible. All the items in a list do not need to be of the same type. Items separated by commas are enclosed within brackets [ ]. To some extent, lists are similar to arrays in C. One difference between them is that all the items belonging to a list can be of different data type. The values stored in a list can be accessed using the slice operator [ ] and [:]) with indices starting at 0 in the beginning of the list and ending with -1. The plus (+) sign is the list concatenation operator and the asterisk (*) is the repetition operator.

Example Program

```
first_list = ['abcd', 147, 2.43, 'Tom', 74.9]
small_list = [111, 'Tom']
print (first_list)            # Prints complete list
print (first_list [0] )       # Prints first element of the list
print (first_list[1:3])       # Prints elements starting from 2nd till 3rd
print (first_list [2:])       # Prints elements starting from 3rd element
print (small_list * 2)        # Prints list two times
print (first_list + small_list)  # Prints concatenated lists
```

Output

```
['abcd', 147, 2.43, 'Tom', 74.9]
abcd
[147, 2.43]
[2.43, 'Tom', 74.91]
[111, 'Tom', 111, 'Tom']
['abcd', 147, 2.43, 'Tom', 74.9, 111, 'Tom']
```

We can update lists by using the slice on the left hand side of the assignment operator. Updates can be done on single or multiple elements in a list.

Example Program

```
 #Demo of List Update
list = ['abcd', 147, 2.43, 'Tom', 74.9]
print("Item at position 2=", list [2])
list [2] =500
print("Item at position 2=", list [2])
print("Item at Position 0 and 1 is", list [0], list[1])
list [0] =20
list [1] ='apple'
print("Item at Position 0 and 1 is", list[0], list [1])
```

Output

```
Item at position 2= 2.43
Item at position 2= 500
Item at Position 0 and 1 is abcd 147
Item at Position 0 and 1 is 20 apple
```

To remove an item from a list, there are two methods. We can use del statement or remove() method.

Example Program

```
#Demo of List Deletion
list = ['abcd', 147, 2.43, 'Tom', 74.9]
print (list)
del list [2]
print("List after deletion: ", list)
```

Output

```
['abcd', 147, 2.43, 'Tom', 74.91
List after deletion: ["abcd', 147, Tom', 74.9]
```

3.1 Built-in List Functions

1. len(list) - Gives the total length of the list.
Example Program

```
#Demo of len(list)
list1 = ['abcd', 147, 2.43, 'Tom']
print (len (list1))
```

Output

```
4
```

2. max(list) - Returns item from the list with maximum value.
Example Program

```
#Demo of max (list)
list1 =[1200, 147, 2.43, 1.12]
list2 = [213, 100, 289]
print ("Maximum value in: ", list1, " is ",max(list1))
print ("Maximum value in: ", list2, " is ", max(list2))
```

Output

```
Maximum value in: [1200, 147, 2.43, 1.12] is 1200
Maximum value in: [213, 100, 289] is 289
```

3. min(list) - Returns item from the list with minimum value.
Example Program

```
#Demo of min (list)
list1 = [1200, 147, 2.43, 1.12]
list2 = [213, 100, 289]
print("Minimum value in: ", list1, " is ", min(list1))
print ("Minimum value in: ", list2, " is ", min(list2))
```

Output

```
Minimum value in: [1200, 147, 2.43, 1.12] is 1.12
Minimum value in: [213, 100, 289] is 100
```

4. list(seq) - Returns a tuple into a list.
Example Program

```
#Demo of list (seg)
tuple = ('abcd', 147, 2.43, 'Tom')
print ("List:", list (tuple) )
```

Output

```
List: ['abcd', 147, 2.43, 'Tom']
```

5. map(aFunction,aSequence) - One of the common things we do with list and other sequences is applying an operation to each item and collect the result. The map(a Function, aSequence) function applies a passed-in function to each item in an iterable object and returns a list containing all the function call results.
Example Program

```
# Python program to demonstrate working of map.
def addition(n):
   return n + n

numbers = (1, 2, 3, 4)
result = map(addition, numbers)
print(list(result))
```

Output

```
[2, 4, 6, 8]
```

Example Program 2

```
def calc_sum(x1, x2):
    return x1+x2

map_obj = map(calc_sum, [1, 2, 3], [10, 20, 30])
print(list(map_obj))
```

Output

```
[11, 22, 33]
```

3.2 Built-in List Methods

1. list.append(obj) - This method appends an object obj passed to the existing list.

Example Program

```
#Demo of list.append(obj)
list = ['abcd', 147, 2.43, 'Tom']
print ("Old List before Append:", list)
list.append(100)
print("New List after Append:", list)
```

Output

```
old List before Append: ['abcd', 147, 2.43, Tom']
New List after Append: [''abcd', 147, 2.43, Tom', 100)
```

2. list.count(obj) - Returns how many times the object obj appears in a list.

Example Program

```
#Demo of list.count (obj)
list = ['abcd', 147, 2.43, 'Tom', 147,200, 147]
print ("The number of times", 147, appears in", list, "=", list.count (147))
```

Output

```
The number of times 147 appears in 'abcd', 147, 2.43, 'Tom', 147, 200, 147] = 3
```

3. list.remove(obj) - Removes object obj from the list.

Example Program

```
#Demo of list.remove(obj)
list1 = ['abcd', 147, 2.43, 'Tom']
list1.remove('Tom')
print (list1)
```

Output

```
['abcd', 147, 2.43]
```

4. list.index(obj) - Returns index of the object obj if found, otherwise raise an exception indicating that value does not exist.

Example Program

```
#Demo of list.index (obj)
listi = ['abcd', 147, 2.43, 'Tom']
print (listi.index (2.43))
print(listi.index('hai'))
```

Output

```
2
```

```
Traceback (most recent call last):
File "main.py", line 4, in <module>
print(listi.index('hai'))
ValueError: 'hai' is not in list
```

5. list.extend(seq) - Appends the contents in a sequence seq passed to a list.

Example Program

```
#Demo of list.extend (seq)
list1 = ['abcd', 147, 2.43, 'Tom']
list2 = ['def', 100]
list1.extend (list2)
print (list1)
```

Output

```
['abcd', 147, 2.43, 'Tom', 'def', 100]
```

6. list.reverse() - Reverses objects in a list.

Example Program

```
#Demo of list.reverse()
list1 = ['abcd', 147, 2.43, 'Tom']
list1.reverse()
print (list1)
```

Output

```
['Tom', 2.43, 147, ‘abcd']
```

7. list.insert(index,obj) - Returns a list with object obj inserted at the given index.

Example Program

```
#Demo of list.insert(index, obj)
list1 = ['abcd', 147, 2.43, 'Tom']
print ("List before insertion:", list1)
list1.insert (2,222)
print("List after insertion:", list1)
```

Output

```
List before insertion: l'abcd', 147, 2.43, 'Tom']
List after insertion: ['abcd', 147, 222, 2.43, 'Tom']
```

8. list.sort([Key=None, Reverse=False]) - Sorts the items in a list and returns the list. If a function is provided, it will compare using the function provided.

Example Program

```
#Demo of list.sort ( [Key=None, Reverse=False])
list1 = [890, 147, 2.43, 1001]
print("List before sorting:", list1)
list1.sort()
print("List after sorting in ascending order:", list1)
```

Output

```
List before sorting: [840, 147, 2.43, 100]
List after sorting in ascending order: [2.43, 100, 147,840]
```

The following example illustrates how to sort the list in descending order.

Example Program

```
#Demo of list.sort ( [Key=None, Reverse=True])
list1 = [890, 147, 2.43,100]
```

```
print("List before sorting:", list1)
list1.sort (reverse=True)
print("List after sorting in descending order:", list1)
```

Output

```
List before sorting: (840, 147, 2.43, 100]
List after sorting in descending order: [840, 147, 100, 2.43]
```

9. list.pop([index]) - Removes or returns the last object obj from a list. We can even pop out any item in a list with the index specified.

Example Program

```
#Demo of list.pop ([index] )
list1 = ['abcd', 147, 2.43, 'Tom']
print("List before poping", list1)
list1.pop(-1)
print("List after poping: ",list 1)
item=list1.pop(-3)
print("Popped item:", item)
print("List after poping: ", list1)
```

Output

```
List before poping ['abcd', 147, 2.43, 'Tom']
List after poping: ['abcd', 147, 2.43]
```

10. list.clear() - Removes all items from a list.

Example Program

```
#Demo of list.clear()
list1 = ['abcd', 147, 2.43, 'Tom']
print("List before clearing:", list1)
list1.clear()
print("List after clearing:", list1)
```

Output

```
List before clearing: ['abcd', 147, 2.43, 'Tom']
List after clearing: [ ]
```

11. list.copy( ) - Returns a copy of the list

Example Program

```
#Demo of list.copy ( )
list1 = ['abcd', 147, 2.43, 'Tom']
print("List before clearing:", list1)
list2=list1.copy( )
list1.clear()
print("List after clearing:", list1)
print ("Copy of the list:",list2)
```

Output

```
List before clearing: ['abcd', 147, 2.43, 'Tom']
List after clearing: [ ]
Copy of the list: ['abcd', 147, 2.43, Tom']
```

3.3 Using List as Stacks

The list can be used as a stack (Last In First Out). Stack is a data structure where the last element added is the first element retrieved. The list methods make it very easy to use a list as a stack. To add

an item to the top of the stack, use **append().** To retrieve an item from the top of the stack, use **pop( )** without an explicit index.
Example Program

```
#Demo of List as Stack
stack=[10,20,30,40,50]
stack.append(60)
print ("Stack after appending:" ,stack)
stack.pop()
print("Stack after poping.", stack)
```

Output

```
Stack after appending: [10, 20, 30, 40, 50, 60]
Stack after poping: [10, 20, 30, 40, 50]
```

2.3.4 Using List as Queues

It is also possible to use a list as a queue, where the first element added is the first element retrieved. Queues are First In First Out (FIFO) data structure. But lists are not efficient for this purpose. While appends and pops from the end of list are fast, doing inserts or pops from the beginning of a list is slow since all of the other elements have to be shifted by one.
To implement a queue, Python provides a module called **collections** in which a method called **deque** is designed to have fast appends and pops from both ends.
Example Program

```
#Demo of List as Queue from collections import deque
from collections import deque
queue = deque (["apple", "orange", "pear"])
queue. append("cherry")                  # cherry arrives
print(queue)
queue. append("grapes")                  # grapes arrives
print(queue)
queue.popleft()                          # The first to arrive now leaves
print(queue)
queue.popleft()                          # The second to arrive now leaves
print (queue)                            # Remaining queue in order of arrival
```

Output

```
deque(['apple', 'orange', 'pear', 'cherry'])
deque(['apple', 'orange', 'pear', 'cherry', 'grapes'])
deque(['orange', 'pear', 'cherry', 'grapes'])
deque(['pear', 'cherry', 'grapes'])
```

4 TUPLE

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

```
first_tuple = ('abcd', 147, 2.43, 'Tom', 74.9)
small tuple = (111, 'Tom')
```

```
print (first_tuple)                  # Prints complete tuple
print (first_tuple [0])              # Prints first element of the tuple
print (first_tuple(1:3])             # Prints elements starting from 2nd till 3rd
print (first_tuple [2:])             # Prints elements starting from 3rd element
print (small_tuple * 2)              # Prints tuple two times
print (first_tuple + small_tuple)    # Prints concatenated tuples
```

Output

```
('abcd', 147, 2.43, 'Tom', 74.9)
abcd
(147, 2.43)
(2.43, 'Tom', 74.9)
(111, Tom', 111, 'Tom')
 ('abcd', 147, 2.43, 'Tom', 74.9, 111, 'Tom')
```

The following code is invalid with tuple whereas this is possible with lists

```
first_list = ['abcd', 147, 2.43, 'Tom', 74.9 ]
first_tuple = ('abcd', 147, 2.43, 'Tom', 74.9)
tuple [2] = 100                # Invalid syntax with tuple
list [2] = 100                 # Valid syntax with list
```

To delete an entire tuple we can use the **del** statement. Example del tuple. It is **not possible to remove individual items from a tuple**. However it is possible to create tuples which contain mutable objects, such as lists.

Example Program

```
#Demo of Tuple containing Lists
t=([1,2,3], ['apple', 'pear', 'orange'])
print (t)
```

Output

```
([1, 2, 3], ['apple', 'pear', 'orange'])
```

It is possible to pack values to a tuple and unpack values from a tuple. We can create tuples even without parenthesis. The reverse operation is called sequence unpacking and works for any sequence on the right-hand side. Sequence unpacking requires that there are as many variables on the left side of the equals sign as there are elements in the sequence,

Example Program

```
#Demo of Tuple packing and unpacking
t="apple", 1,100
print(t)
x, y, z=t
print(x)
print(y)
print(z)
```

Output

```
('apple', 1, 100)
apple
100
```

4.1 Built-in Tuple Functions

1. len(tuple) - Gives the total length of the tuple.

Example Program

```
#Demo of len (tuple)
tuplel = ('abcd', 147, 2.43, 'Tom')
print(len (tuplel))
```

Output

```
4
```

2. max(tuple) - Returns item from the tuple with maximum value.

Example Program

```
#Demo of max (tuple)
tuple1 = (1200, 147, 2.43, 1.12)
tuple2 = (213, 100, 289)
print ("Maximum value in: ", tuple1, "is", max (tuple1))
print ("Maximum value in: ", tuple2, "is", max (tuple2))
```

Output

```
Maximum value in: (1200, 147, 2.43, 1.12) is 1200
Maximum value in: (213, 100, 289) is 289
```

3. min(tuple) - Returns item from the tuple with minimum value.

Example Program

```
#Demo of min (tuple)
tuple1 = (1200, 147, 2.43, 1.12)
tuple2 = (213, 100, 289)
print("Minimum value in: ", tuple1, "is", min (tuple1))
print("Minimum value in: ", tuple2, "is", min (tuple2))
```

Output

```
Minimum value in: (1200, 147, 2.43, 1.12) is 1.12
Minimum value in: (213, 100, 289) is 100
```

4. tuple(seq) - Returns a list into a tuple.

Example Program

```
#Demo of tuple(seg)
list = ['abcd', 147, 2.43, ' Tom']
print("Tuple:", tuple (list))
```

Output

```
Tuple: ('abcd', 147, 2.43, 'Tom')
```

5 SET

Set is an unordered collection of **unique items**. Set is defined by values separated by comma inside braces **{ }**. It can have any number of items and they may be of different types (integer, float, tuple, string etc.). **Items in a set are not ordered.** Since they are unordered we cannot access or change an element of set using indexing or slicing. We can perform set operations like union, intersection, difference on two sets. Set have unique values. They eliminate duplicates. The slicing operator [ ] does not work with sets. An empty set is created by the function **set().**

Example Program

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

```
s2={1,2,3,2,1,2 }               #output contains only unique values
print (s2)
s3={1, 2.4, 'apple', 'Tom', 3 } #set of mixed data types
print (s3)
#s4={1,2, [3, 4] }              #sets cannot have mutable items
#print s4                       # Hence not permitted
s5=set([1,2,3,4] )              #using set function to create set from a list
print (s5)
```

Output

```
{1, 2, 3)
{1, 2, 3}
(1, 3, 2.4, 'apple, Tom' }
{1, 2, 3, 4}
```

5.1 Built-in Set Functions

1. len(set) - Returns the length or total number of items in a set.

Example Program

```
#Demo of len (set)
set1 = {'abcd', 147, 2.43, 'Tom'}
print (len (set1))
```

Output

```
4
```

2. max(set) - Returns item from the set with maximum value.

Example Program

```
#Demo of max (set)
set1= {1200, 147, 2.43, 1.12}
set2 = {213, 100, 289}
print ("Maximum value in: ", set1, "is", max (set1))
print ("Maximum value in: ",set2, "is", max (set2))
```

Output

```
Maximum value in: {1200, 1.12, 2.43, 147} is 1200
Maximum value in: {289, 100, 213} is 289
```

3. min(set) - Returns item from the set with minimum value.

Example Program

```
#Demo of min (set)
set1 = {1200, 147, 2.43,1.12}
set2 = {213, 100, 289}
print("Minimum value in:", set1, "is", min(set1))
print("Minimum value in:", set2, "is", min(set2))
```

Output

```
Minimum value in: {1200, 1.12, 2.43, 147} is 1.12
Minimum value in: {289, 100, 213} is 100
```

4. sum(set) - Returns the sum of all items in the set.

Example Program

```
#Demo of sum (set)
set1 = {147, 2.43}
```

```
set2 = {213, 100, 289}
print("Sum of elements in", set1, "is", sum (set1))
print ("Sum of elements in", set2, "is" ,sum (set2) )
```

Output

```
Sum of elements in {147, 2.43} is 149.43
Sum of elements in {289, 100, 213} is 602
```

5. sorted(set)- Returns a new sorted list. The set does not sort itself.

Example Program

```
#Demo of sorted (set)
set1 = {213, 100, 289, 40, 23, 1, 1000}
set2 = sorted (set1)
print ("Sum of elements before sorting:", set1)
print ("Sum of elements after sorting:", set2)
```

Output

```
Sum of elements before sorting:{1, 100, 289, 1000, 40, 213, 23)
Sum of elements after sorting: [1, 23, 40, 100, 213, 289, 1000]
```

6. enumerate(set) - Returns an enumerate object. It contains the index and value of all the items of set as a pair.

Example Program

```
#Demo of enumerate (set)
set1 = {213, 100, 289, 40, 23, 1, 1000}
print("enumerate (set):", enumerate (set1))
```

Output

```
enumerate (set): <enumerate object at Ox7f0a73573690>
```

7. any(set) - Returns True, if the set contains at least one item, False otherwise.

Example Program

```
#Demo of any (set)
set1 = set()
set2={1,2,3,4}
print ("any (set):", any (set1))
print("any (set):", any (set2))
```

Output

```
any (set): False
any (set) : True
```

8. a11(set) - Returns True, if all the elements are true or the set is empty.

Example Program

```
#Demo of all (set)
set1 = set ()
set2={1,2,3,4}
s={0,0}
print("all (set):", all (set1))
print("all (set):", all (set1))
print(all(s))
```

Output

```
all(set) : True
all (set) : True
```

```
False
```

5.2 Built-in Set Methods

1. set.add(obj) - Adds an element obj to a set.

Example Program

```
#Demo of set.add(obj)
set1={3, 8, 2,6}
print("Set before addition:", set1)
set1.add (9)
print ("Set after addition:", set1)
```

Output

```
Set before addition:{8, 2, 3, 6}
Set after addition:{8, 9, 2, 3, 6}
```

2. set.remove(obj) - Removes an element obj from the set. Raises KeyError if the set is empty.

Example Program

```
#Demo of set.remove(obj)
set1={3,8,2,6}
print ("Set before deletion:", set1)
set1.remove (8)
print("Set after deletion:", set1)
set1.remove(10)
print("Set after deletion:", set1)
```

Output

```
Set before deletion:{8, 2, 3, 6}
Set after deletion:{2, 3, 6}
Traceback (most recent call last):
File "main.py", line 6, in <module>
set1.remove(10)
KeyError: 10
```

3. set.discard(obj) - Removes an element obj from the set. Nothing happens if the element to be deleted is not in the set.

Example Program

```
#Demo of set discard (obj)
set1={3, 8, 2,6}
print("Set before discard:", set1)
set1.discard (8)
print("Set after discard:", set1)          # Element is present
set1.discard (9)
print ("Set after discard:", set1)         #Element is not present
```

Output

```
Set before discard:{8, 2, 3, 6}
Set after discard: {2, 3, 6}
Set after discard: {2, 3, 6}
```

4. set.pop() - Removes and returns an arbitrary set element. Raises KeyError if the set is empty.

Example Program

```
#Demo of set.pop()
set1={3,8,2,6}
print ("Set1 before pop:", set1)
set1.pop()
print("Set1 after poping:", set1)
set1.pop()
print("Set1 after poping:", set1)
set1.pop()
print("Set1 after poping:", set1)
set1.pop()
print("Set1 after poping:", set1)
set1.pop()
print("Set1 after poping:", set1)
```

Output

```
Set1 before pop: {8, 2, 3, 6}
Set1 after poping: {2, 3, 6}
Set1 after poping: {3, 6}
Set1 after poping: {6}
Set1 after poping: set()
Traceback (most recent call last):
File "main.py", line 12, in <module>
set1.pop()
KeyError: 'pop from an empty set'
```

5. set1.union(set2) - Returns the union of two sets as a new set.

Example Program

```
#Demo of set1.union (set2)
set1={3, 8, 2,6}
print ("Set 1:",set1)
set2={4,2,1,9}
print ("Set 2:", set2)
set3=set1.union (set2)          #Unique values will be taken
print("Union:", set3)
```

Output

```
Set 1:{8, 2, 3, 6}
Set 2:{9, 2, 4, 1}
Union:{1, 2, 3, 4, 6, 8, 9}
```

6. set1.update(set2) - Update a set with the union of itself and others. The result will be stored in set1.

Example Program

```
#Demo of set1. update (set2)
set1={3, 8, 2, 6}
print("Set1:", set1)
set2 = {4,2,1,9}
print("Set 2:", set2)
set1.update (set2)              #Unique values will be taken
```

```
print("Update Method:", set1)
```

Output

```
Set 1:{8, 2, 3, 6}
Set 2:{9, 2, 4, 1}
Update Method: {1, 2, 3, 4, 6, 8, 9}
```

7. set1.intersection(set2) - Returns the intersection of two sets as a new set.

Example Program

```
#Demo of set1.intersection (set2)
setl={3, 8, 2,6}
print ("Set 1:", set1)
set2= {4,2,1,9}
print ("Set 2:", set2)
set3 = set1.intersection (set2)
print("Intersection:", set3)
```

Output

```
Set 1:{8, 2, 3, 6}
Set 2:{9, 2, 4, 1}
Intersection:{2}
```

8. set1.intersection update() - Update the set with the intersection of itself and another. The result will be stored in set1.

Example Program

```
#Demo of set1. intersection_update (set2)
set1={3, 8, 2,6}
print("Set 1:", set1)
set2={4,2,1,9}
print ("Set 2:", set2)
set1.intersection_update (set2)
print ("Intersection:", set1)
```

Output

```
Set 1:{8, 2, 3, 6}
Set 2: {9, 2, 4, 1}
Intersection_update: {2}
```

9. set1.difference(set2) - Returns the difference of two or more sets into a new set.

Example Program

```
#Demo of set1.difference (set 2)
set1={3,8,2,6}
print ("Set 1:", set1)
set2={4,2,1,9}
print ("Set 2:", set2)
set3= set1.difference (set2)
print("Difference:", set3)
```

Output

```
Set 1:{8, 2, 3, 6}
Set 2:{9, 2, 4, 1}
Difference: {8, 3, 6}
```

10. set1.difference_update(set2) - Remove all elements of another set set2 from set1 and the result is stored in set1.

Example Program

```
#Demo of set1.difference_update (set 2)
set1={3, 8, 2, 6}
print("Set 1:", set1)
set2= {4,2,1,9}
print ("Set 2:", set2)
set1.difference_update (set2)
print("Difference Update:", set1)
```

Output

```
Set 1:{8, 2, 3, 6}
Set 2:{9, 2, 4, 1}
Difference Update: {8, 3, 6}
```

11. set1.symmetric_difference(set2)- Return the symmetric difference of two sets as a new set.

Example Program

```
#Demo of set1.Symmetric_difference (set2)
set1={3, 8, 2,6}
print("Set 1:", set1)
set2= {4,2,1,9}
print ("Set 2:", set2)
set3= set1.symmetric_difference (set2)
print("Symmetric Difference :", set3)
```

Output

```
Set 1:{8, 2, 3, 6}
Set 2:{9, 2, 4, 1}
Symmetric Difference:{ 1, 3, 4, 6, 8, 9}
```

12. set1.symmetric_difference_update(set2) - Update a set with the symmetric difference of itself and another.

Example Program

```
#Demo of set1.symmetric difference_update (set 2)
set1={3,8,2,6}
print ("Set1:", set1)
set2= {4,2,1,9}
print("Set 2:", set2)
set1.symmetric_difference_update (set2)
print ("Symmetric Difference Update:", set1)
```

Output

```
Set 1: {8, 2, 3, 6}
Set 2: {9, 2, 4, 1}
Symmetric Difference Update: {1, 3, 4, 6, 8, 9}
```

13. set1.isdisjoint(set2) - Returns True if two sets have a null intersection.

Example Program

```
#Demo of set1.isdisjoint (set 2)
set1={3,8,2,6}
print ("Set 1:", set1)
```

```
set2={4,7,1,9}
print ("Set 2:", set2)
print("Result:", set1.isdisjoint (set2))
```

Output

```
Set 1:{8, 2, 3, 6}
Set 2:{9, 7, 4, 1}
Result: True
```

14. set1.issubset(set2) - Returns True if set1 is a subset of set2.

Example Program

```
#Demo of set1.issubset (set2)
set1={3,8}
print ("Set 1:", set1)
set2= {3, 8,4,7,1,9 }
print ("Set 2:", set2)
print("Result :", set1.issubset (set2))
```

Output

```
Set 1:{8, 3}
Set 2:{1, 3, 4, 7, 8, 9)
Result : True
```

15.set1.issuperset(set2) - Returns True, if set1 is a super set of set2.

Example Program

```
#Demo of set1.issuperset (set 2)
Set1={3,8,4,6 }
print ("Set 1:", set1)
set2={3,8,}
print ("Set 2:", set2)
print("Result :", set1.issuperset (set 2))
```

Output

```
Set 1:{8, 3, 4, 6}
Set 2:{8, 3}
Result : True
```

5.3 Frozenset

Frozenset is a new class that **has the characteristics of a set, but its elements cannot be changed once assigned.** While tuples are immutable lists, frozensets are immutable sets. Sets being mutable are unhashable, so they can't be used as dictionary. On the other hand, frozensets are hashable and can be used as keys to a dictionary.

Frozensets can be created using the function **frozenset()**. This datatype supports methods like difference, intersection(), isdisjoint(), issubset), issuperset(), symmetric difference() and union(). Being immutable it does not have methods like add(), remove(), update(), difference_update(), intersection_update(),symmetric difference_update() etc.

Example Program

```
#Demo of frozenset()
set1= frozenset({3, 8,4,6})
print ("Set 1:", set1)
set2=frozenset({3,8})
print("Set 2:", set2)
print("Result of set1.intersection (set 2):", set1.intersection (set2))
```

Output

```
Set 1: frozenset({8, 3, 4, 6})
Set 2: frozenset (8, 3})
Result of set1.intersection (set2): frozenset({8, 3})
```

6 DICTIONARY

Dictionary is an unordered collection of **key-value pairs**. It is generally used when we have a huge amount of data. We must know the key to retrieve the value. In Python, dictionaries are defined within braces { } with each item being a pair in the form **key:value**. Key and value can be of any type. Keys are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object. Dictionaries are sometimes found in other languages as **"associative memories" or "associative arrays".**

Dictionaries are enclosed by curly braces ({}) and values can be assigned and accessed using square braces ([ ]).

Example Program

```
dict={}
dict['one'] = "This is one"
dict [2] ="This is two"
tinydict={'name': 'john', 'code': 6734, 'dept': 'sales'}
studentdict={'name': 'john', 'marks': [35,80,90]}
print (dict['one'])              # Prints value for 'one' key
print (dict [2] )                # Prints value for 2 key
print (tinydict)                 # Prints complete dictionary
print (tinydict.keys ())         # Prints all the keys
print (tinydict.values())        # Prints all the values
print (studentdict)              #Prints studentdict
```

Output

```
This is one
This is two
{'dept': 'sales', 'name': 'john', 'code': 6734}
dict_keys ( ['dept', 'name', 'code'])
dict_values(['sales', 'john', 6734])
{"marks': [35, 80, 90], 'name': 'john'}
```

We can update a dictionary by adding a new key-value pair or modifying an existing entry.

Example Program

```
#Demo of updating and adding new values to dictionary
dict1= { 'Name':'Tom', 'Age':20, 'Height' :160}
print (dict1)
dict1['Age'] =25             #updating existing value in Key-Value pair
print("Dictionary after update:", dict1)
dict1['weight'] =60          #Adding new Key-value pair
print("After adding new Key-value pair:", dict1)
```

Output

```
{'Name': 'Tom', 'Age': 20, 'Height': 160}
Dictionary after update: {'Name': 'Tom', 'Age': 25, 'Height': 160}
```

```
After adding new Key-value pair: {'Name': 'Tom', 'Age': 25, 'Height': 160, 'weight': 60}
```

We can delete the entire dictionary elements or individual elements in a dictionary. We can use **del** statement to delete the dictionary completely. To remove entire elements of a dictionary, we can use the **clear()** method.

Example Program

```
#Demo of Deleting Dictionary
dict1= {'Name':'Tom', 'Age':20, 'Height':160}
print (dict1)
del dict1['Age']          #deleting Key-value pair 'Age : 20
print("Dictionary after deletion:", dict1)
dict1.clear()             #Clearing entire dictionary
print (dict1)
```

Output

```
{'Age': 20, 'Name': Tom', 'Height: 160)
Dictionary after deletion: { 'Name': 'Tom', 'Height': 160}
```

Properties of Dictionary Keys

1. More than one entry per key is not allowed.ie, no duplicate key is allowed. When duplicate keys are encountered during assignment, the last assignment is taken.

2. Keys are immutable. This means keys can be numbers, strings or tuple. But it does not permit mutable objects like lists.

6.1 Built-in Dictionary Functions

1. len(dict) - Gives the length of the dictionary.

Example Program

```
#Demo of len (dict)
dict1= {'Name': 'Tom', 'Age':20, 'Height' :160}
print (dict1)
print("Length of Dictionary=",len (dict1))
```

Output

```
{'Age': 20, 'Name': Tom', 'Height': 160}
Length of Dictionary: 3
```

2. str(dict) - Produces a printable string representation of the dictionary.

Example Program

```
#Demo of str (dict)
dict1={'Name': 'Tom', 'Age':20, 'Height' :160}
print (dict1)
print("Representation of Dictionary=", str (dict1))
```

Output

```
{'Age': 20, 'Name': Tom', 'Height': 160}
Representation of Dictionary= {'Age': 20, 'Name': 'Tom', 'Height': 160}
```

3. type(variable) - The method type() returns the type of the passed variable. If passed variable is dictionary then it would return a dictionary type. This function can be applied to any variable type like number, string, list, tuple etc.

Example Program

```
#Demo of type (variable)
```

```
dict1= {'Name': 'Tom', 'Age':20, 'Height' :160}
print (dict1)
print("Type (variable) =" , type (dict1))
s="abcde"
print("Type (variable)", type (s) )
list1= [1, 'a', 23, 'Tom']
print ("Type (variable) =" , type (list1))
```

Output

```
{'Age' : 20, 'Name': 'Tom', 'Height': 160}
Type (variable) = <class 'dict'>
Type (variable) = <class 'str'>
Type (variable) = <class 'list'>
```

6.2 Built-in Dictionary Methods

1. dict.clear() - Removes all elements of dictionary dict**.**

Example Program

```
#Demo of dict.clear()
dict1= {'Name': 'Tom', 'Age':20, 'Height' :160}
print (dict1)
dict1.clear()
print (dict1)
```

Output

```
{'Age': 20, 'Name': 'Tom', 'Height': 160}
```

2. dict.copy - Returns a copy of the dictionary dict.

Example Program

```
#Demo of dict.copy)
dict1= { 'Name':'Tom', 'Age' :20, 'Height' :160}
print (dict1)
dict2=dict1.copy( )
print (dict2)
```

Output

```
{'Age' : 20, 'Name': Tom', 'Height': 160)
{'Age': 20, 'Name': 'Tom', 'Height': 160}
```

3. dict.keys() - Returns a list of keys in dictionary dict.

Example Program

```
#Demo of dict.keys ()
dict1= {'Name': 'Tom', 'Age':20, 'Height' :160}
print (dict1)
print("Keys in Dictionary:", dict1.keys())
```

Output

```
{'Age': 20, 'Name': 'Tom', 'Height': 160)
 Keys in Dictionary: ('Age', 'Name', 'Height'l
```

The values of the keys will be displayed in a random order. In order to retrieve the keys in sorted order, we can use the **sorted()** function. But for using the sorted() function, **all the keys should of the same type**.

Example Program

```
#Demo of dict.keys () in sorted order
dict1={'Name':'Tom', 'Age':20, 'Height' :160}
print (dict1)
print("Keys in sorted order:", sorted(dict1.keys()))
```

Output

```
{'Name': 'Tom', 'Age': 20, 'Height': 160}
Keys in sorted order: ['Age', 'Height', 'Name']
```

4.dict.values()- This method returns list of all values available in a dictionary.

Example Program

```
#Demo of dict.valuess ()
dict1={'Name':'Tom', 'Age':20, 'Height' :160}
print (dict1)
print("Values in dictionary :", dict1.values())
```

Output

```
{'Name': 'Tom', 'Age': 20, 'Height': 160}
Values in dictionary : dict_values(['Tom', 20, 160])
```

The values will be displayed in a random order. In order to retrieve the values in sorted order, we can use the **sorted()** function. But for using the sorted() function, **all the values should of the same type.**

Example Program

```
#Demo of dict.values ()
dict1={'Weight':55, 'Age':20, 'Height' :160}
print (dict1)
print("Values in dictionary in sorted order :", sorted(dict1.values()))
```

Output

```
{'Weight': 55, 'Age': 20, 'Height': 160}
Values in dictionary in sorted order : [20, 55, 160]
```

5.dict.items()- Returns a list of dictionary dict's(key, value) tuple pairs.

```
#Demo of dict.items()
dict1={'Name':'Tom', 'Age':20, 'Height' :160}
print (dict1)
print("Items in dictionary :", dict1.items())
```

Output

```
{'Name': 'Tom', 'Age': 20, 'Height': 160}
Items in dictionary : dict_items([('Name', 'Tom'), ('Age', 20), ('Height', 160)])
```

6.dict1.update(dict2)- The dictionary dict2's key-value pair will be updated in dictionary dict1.

Example Program

```
#Demo of dict1.update(dict2)
dict1={'Name':'Tom', 'Age':20, 'Height' :160}
print (dict1)
dict2={'Weight' :50}
print (dict2)
dict1.update(dict2)
print("Dict1 upated dict2 :", dict1)
```

Output

```
{'Name': 'Tom', 'Age': 20, 'Height': 160}
{'Weight': 50}
```

```
Dict1 upated dict2 : {'Name': 'Tom', 'Age': 20, 'Height': 160, 'Weight': 50}
```

7. dict.get(key, default=None) - Returns the value corresponding to the key specified and if the key specified is not in the dictionary, it returns the default value.

Example Program

```
#Demo of dicti.get (key, default='Name')
dict1= { 'Name': 'Tom', 'Age' :20, 'Height' :160}
print (dict1)
print("Dict1.get('Age') :", dict1.get('Age'))
print ("Dict1.get('Phone') :", dict1.get('Phone', 0))  # Phone not a key, hence 0 is given as default
print(dict1)
```

Output

```
{'Age' : 20, 'Name': 'Tom', 'Height': 160}
Dict1.get('Age') : 20
Dict1.get('Phone') : 0
{'Name': 'Tom', 'Age': 20, 'Height': 160}
```

8. dict.setdefault(key, default=None) - Similar to dict.get(key,default=None) but will set the key with the value passed and if key is not in the dictionary, it will set with the default value.

Example Program

```
#Demo of dict1.set(key, default='Name')
dict1= {'Name': 'Tom', 'Age' :20, 'Height' :160}
print (dict1)
print("Dict1.setdefault('Age') :", dict1.setdefault ('Age'))
print("Dict1. set de fault(Phone) :", dict1.setdefault('Phone', 0)) # Phone not a #key, hence O is given as default value.
print(dict1)
```

Output

```
{'Name': 'Tom', 'Age': 20, 'Height': 160}
Dict1.setdefault('Age') : 20
Dict1. set de fault(Phone) : 0
{'Name': 'Tom', 'Age': 20, 'Height': 160, 'Phone': 0}
```

9. dict.fromkeys(seq,[val]) - Creates a new dictionary from sequence seq and values from val.

Example Program

```
#Demo of dict.fromkeys (seg, (val])
list= ['Name', 'Age', 'Height']
dict= dict. fromkeys (list)
print ("New Dictionary:", dict)
```

Output

```
New Dictionary: {'Age': None, 'Name': None, "Height': None}
```

2.7 MUTABLE AND IMMUTABLE OBJECTS

Objects whose value can change are said to be mutable and objects whose value is unchangeable once they are created are called immutable. An object's mutability is determined by its type. For instance, **numbers, strings and tuples** are **immutable,** while **lists, sets and dictionaries** are **mutable**. The following example illustrates the mutability of objects.

Example Program

```
#Mutability illustration
#Numbers
```

```
a=10
print (a) #Value of a before subtraction
print (a-2) #value of a-2
print(a) #Value of a after subtraction
#Strings
str="abcde"
print (str) #Before applying upper() function
print (str.upper()) #result of upper()
print (str) #After applying upper() function
#Lists
list=[1,2,3,4,5]
print (list) #Before applying remove() method
list.remove(3)
print (list) #After applying remove () method
```

Output

```
10
8
10
abcde
ABCDE
abcde
[1, 2, 3, 4, 5]
[1, 2, 4, 5]
```

From the above example, it is clear that the values of numbers and strings are not changed even after applying a function or operation. But in the case of list, the value is changed or the changes are reflected in the original variable and hence called mutable.

2.8 DATA TYPE CONVERSION

We may need to perform conversions between the built-in types. To convert between different data types, use the type name as a function. There are several built-in functions to perform conversion from one data type to another. These functions return a new object representing the converted value. Table 2.6 shows various functions and their descriptions used for data type conversion.

Table 2.6 Functions for Data Type Conversions

Function	**Description**
int(x)	Converts x to an integer
long(x)	Converts x to a long integer.
float(x)	Converts x to a floating-point number.
complex(real [,imag])	Creates a complex number.
str(x)	Converts object x to a string representation.
tuple(s)	Converts s to a tuple.
list(s)	Converts s to a list.

set(s)	Converts s to a set
dict(d)	Creates a dictionary, d must be a sequence of (key,value) tuples.
frozenset(s)	Converts s to a frozen set.
chr(x)	Converts an integer to a character.
unichr(x)	Converts an integer to a Unicode character.
ord(x)	Converts a single character to its integer value.
hex(x)	Converts an integer to a hexadecimal string.
oct(x)	Converts an integer to an octal string

Chapter 3- Flow Control

3.1 DECISION MAKING

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

3.1.1 if statement

Syntax

```
if test expression:
        statement(s)
```

The following Fig. 3.1 shows the flow chart of the if statement.

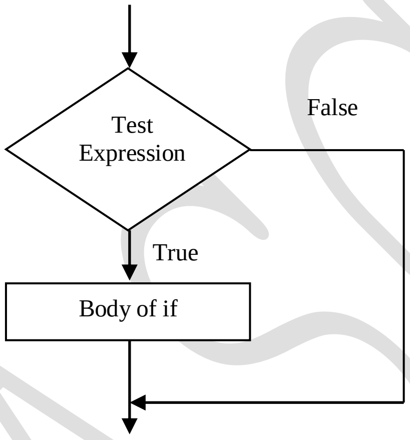

Fig. 3.1 Flow chart of the if statement

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

Example Program

```
num =int (input("Enter a number: "))
if num == 0:
   print("Zero")
print("This is always printed")
```

Output 1

```
Enter a number:
0
Zero
This is always printed
```

Output 2

```
Enter a number:
2
This is always printed
```

In the above example, num == 0 is the test expression. The body of if is executed only if this evaluates to True. When user enters 0, test expression is True and body inside if is executed. When user enters 2, test expression is False and body inside if is skipped. The statement “This is always printed” because the print statement falls outside the if block. The statement outside the if block is shown by unindentation. Hence, it is executed regardless of the test expression or it is always executed.

3.1.2 if....else statement

Syntax

```
if test expression:
        Body of if
else:
        Body of else
```

The following Fig. 3.2 shows the flow chart of if.....else. The if..else statement evaluates test expression and will execute body of if only when the test condition is True. If the condition is False, body of else is executed. Indentation is used to separate the blocks.

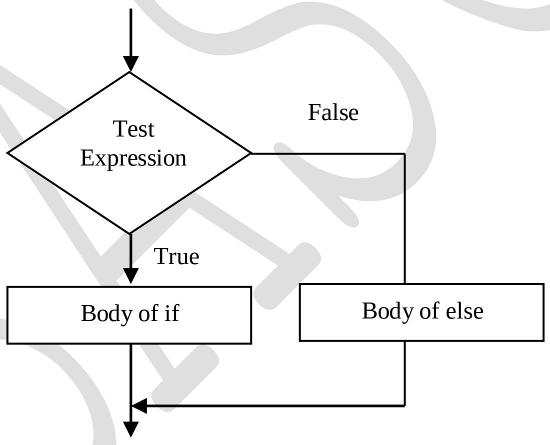

Fig. 3.2 Flow chart of if.....else

Example Program

```
num=int (input ("Enter a number: "))
if num >= 0:
   print("Positive Number or Zero")
else:
   print ("Negative Number")
```

Output 1

```
Enter a number:
5
Positive Number or Zero
```

Output 2

```
Enter a number:
-2
Negative Number
```

In the above example, when user enters 5, the test expression is True. Hence the body if is executed and body of else is skipped. When user enters -2, the test expression is False and body of else is executed and body of if is skipped.

3.1.3 if...elif...else statement

Syntax

```
if test expression:
        Body of if
elif test expression:
        Body of elif
else:
        Body of else
```

The elif is short for else if. It allows us to check for multiple expressions. If the condition for if is False, it checks the condition of the next elif block and so on. If all the conditions are False, body of else is executed. Only one block among the several if...elif...else block is executed according to the condition. An if block can have only one else block. Bu it can have multiple elif blocks. The following Fig. 3.3 shows the flow chart for if...elif..else statement.

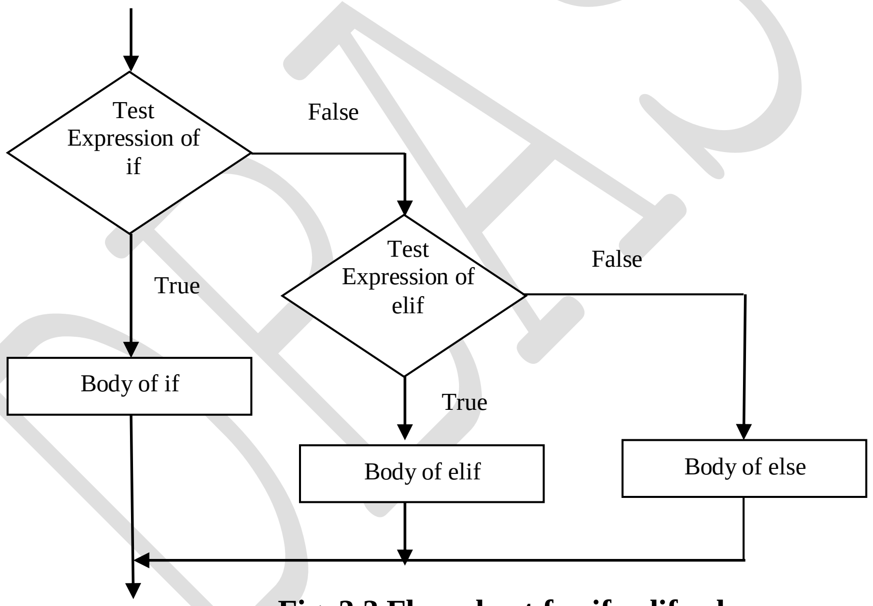

Fig. 3.3 Flow chart for if...elif...else

Example Program

```
num =int (input ("Enter a number: "))
if num > 0:
        print ("Positive number" )
elif num == 0:
        print("Zero")
else:
        print ("Negative number")
```

Output 1

```
Enter a number: 5
```

```
        Positive Number
```
Output 2
```
        Enter a number: 0
        Zero
```
Output 3
```
        Enter a number: -2
        Negative Number
```
3.1.4 Nested if statement

We can have a if...elif...else statement inside another if..elif…else statement. This is called nesting in computer programming. Indentation is the only way to identify the level of nesting.

Example Program
```
num = int (input ("Enter a number: "))
if num >= 0:
        if num == 0:
                print ("Zero")
        else:
                print ("Positive number")
else:
        print ("Negative number")
```
Output 1
```
        Enter a number: 5
        Positive Number
```
Output 2
```
        Enter a number: 0
        Zero
```
Output 3
```
        Enter a number: -2
        Negative Number
```

3.2 LOOPS

Generally, statements are executed sequentially. The first statement in a function is executed first, followed by the second, and so on. There will be situations when we need to execute a block of code several number of times. Python provides various control structures that allow for repeated execution. A loop statement allows us to execute a statement or group of statements multiple times.

3.2.1 for loop

The for loop in Python is used to iterate over a sequence (list, tuple, string) or other objects that can be iterated. Iterating over a sequence is called traversal. Fig. 3.4 shows flow chart of for loop.

Syntax
```
        for item in sequence:
```

Body of for

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

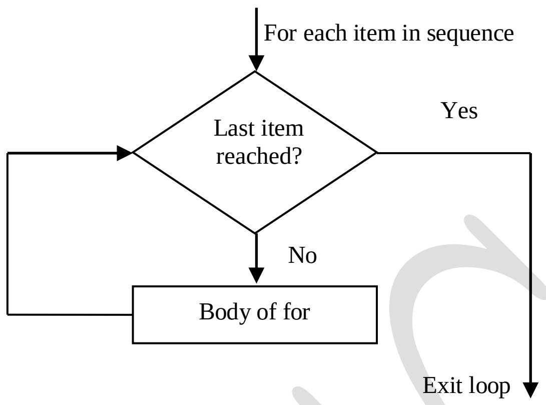

Fig. 3.4 Flow chart of For loop

Example Program 1

```
#Program to find the sum of all numbers stored in a list
# List of numbers
numbers = [2,4,6,8,10]
# variable to store the sum
sum = 0
# iterate over the list
for item in numbers:
        sum = sum-item
# print the sum
print("The sum is", sum)
```

Output 1

```
The sum is -30
```

Example Program 2

```
flowers=['rose', 'lotus','jasmine']
for flower in flowers:
   print('Current flower :', flower)
```

Output 2

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

Example Program 3

```
for letter in 'program':
   print('Current letter :', letter)
```

Output 3

```
Current letter : p
Current letter : r
Current letter : o
Current letter : g
Current letter : r
Current letter : a
Current letter : m
```

3.2.2 range() function

We can use the range () function in for loops to iterate through a sequence of numbers. It can be combined with the len() function to iterate though a sequence using indexing len() function is used to find the length of a string or number of elements in a list, tuple, so etc.

Example Program

```
flowers = ['rose', 'lotus', 'jasmine']
for i in range (len (flowers)):
   print ('Current flower :', flowers [i])
```

Output

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

We can generate a sequence of numbers using range () function. range (10) will generate numbers from 0 to 9 (10 numbers). We can also define the start, stop and step_sizes as **range (start, stop, step_size).** The default value of step_size is 1, if not provided. This function does not store all the values in memory. It keeps track of the start, stop, step_size and generates the next number.

Example Program

```
for num in range (2, 10, 2):
   print ("Number = ", num)
```

Output

```
Number = 2
Number = 4
Number = 6
Number = 8
```

3.2.3 enumerate (iterable, start=0)function

This is one of the built-in functions. Returns an enumerate object. The parameter iterable must be a sequence, an iterator, or some other object like list which supports iteration.

Example Program

```
#Demo of enumerate function using list
flowers= ['rose', 'lotus','jasmine' ,'sunflower']
print (list (enumerate (flowers)))
for index, item in enumerate (flowers):
   print (index, item)
```

Output

```
[(0, 'rose'), (1, 'lotus'), (2, 'jasmine'), (3, 'sunflower')]
0 rose
1 lotus
2 jasmine
3 sunflower
```

3.2.4 for loop with else statement

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

Example Program

```
#Program to find whether an item is present in the list
list=[10, 12, 13,34,27,98]
num = int(input ("Enter the number to be searched in the list: "))
for item in range (len (list)):
    if list [item] == num:
        print("Item found at: ", (item+1))
        break
else:
    print ("Item not found in the list")
```

Output 1

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

Output 2

```
Enter the number to be searched in the list:99
Item not found in the list
```

3.2.5 while loop

The while loop in Python is used to it is used to iterate over a block of code as long as the test expression (condition) is True. We generally use this loop when we don't know the number of times to iterate in advance. Fig. 3.5 shows the flow chart of a while loop.

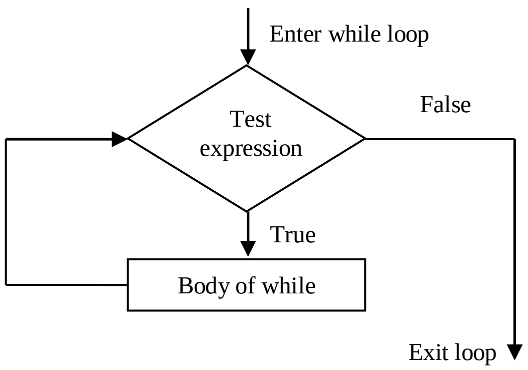

Fig. 3.5 Flow chart for while loop

Syntax

```
while test expression:
        Body of while
```

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

Example Program 1

```
# Program to find the sum of first N natural numbers
n=int(input("Enter the limit: "))
sum=0
i=1
while i<=n:
   sum=sum+i
   i=i+1
print("Sum of first ",n," natural numbers is ", sum)
```

Output

```
Enter the limit:
10
Sum of first 10 natural numbers is 55
```

Illustration

In the above program, when the text Enter the limit appears, 10 is given as the input. ie, n=10. Initially, a counter i is set to 1 and sum is set to 0. When the condition is checked the first time i<=n, it is True. Hence the execution enters the body of while loop. Sum is added with i and i is incremented. Since the next statement unindented, it assumes the end of while block. This process repeated when the condition given in the while loop is False and the control goes to the next print statement to print the sum.

3.2.6 while loop with else statement

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

Example Program 1

```
#Demo of while with else
count=1
while count<=3:
   print("Python Programming")
```

```
    count=count+1
else:
    print("Exit")
print("End of Program")
```

Output 1

```
Python Programming
Python Programming
Python Programming
Exit
End of Program
```

Illustration

In this example, initially the count is 1. The condition in the while loop is checked and since it is True, the body of the while loop is executed. The loop is terminated when the condition is False and it goes to the else statement. After executing the else block it will move on to the rest of the program statements. In this example the print statement after the else block is executed.

Example Program 2

```
# Demo of while with else
count=1
while count<=3:
    print("Python Programming")
    count=count+1
    if count==2:
        break
else:
    print("Exit")
print("End of Program")
```

Output 2

```
Python Programming
End of Program
```

Illustration

In this example, initially the count is 1. The condition in the while loop is checked since it is True, the body of the while loop is executed. When the if condition inside while loop is True, ie. when count becomes 2, the control is exited from the while loop. It will not execute the else part of the loop. The control will moves on to the next statement after the else. Hence the print statement End of Program is executed.

3.3 NESTED LOOPS

Sometimes we need to place a loop inside another loop. This is called nested loop. We can have nested loops for both while and for.

Syntax for nested for loop

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

Syntax for nested while loop

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

Example Program

```
adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]
for x in adj:
  for y in fruits:
    print(x, y)
```

Output

```
red apple
red banana
red cherry
big apple
big banana
big cherry
tasty apple
tasty banana
tasty cherry
```

The above program is rewritten using nested while and is given below.

```
#Program using nested while
adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]
i=0
while i<3:
    j=0
    while j<3:
        print(adj[i], fruits[j])
        j=j+1
    i=i+1
```

Output

```
red apple
red banana
red cherry
big apple
big banana
big cherry
tasty apple
tasty banana
tasty cherry
```

3.4 CONTROL STATEMENTS

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

1. **break**
2. **continue**
3. **pass**

3.4.1 break statement

The break statement terminates the loop containing it. Control of the program flows to the statement immediately after the body of the loop. If it is inside a nested loop, break will terminate the innermost loop. It can be used with both for and while loops. Fig. 3.6 shows the flow chart of break statement.

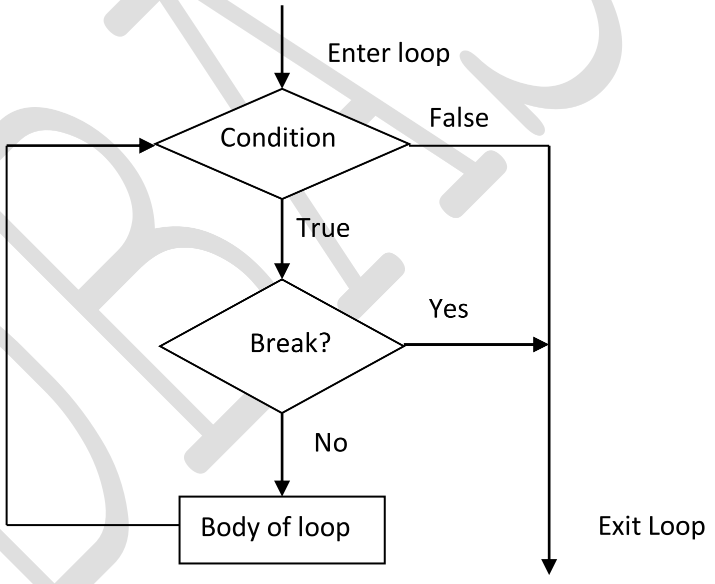

Fig 3.6 Flow chart of break statement

Example Program 1

```
#Demo of break statement in Python
for i in range (2,10,2):
   if i==6:
      break
   print(i)
print("End of Program")
```

Output 1

```
2
```

```
4
End of Program
```

Illustration

In this the for loop is intended to print the even numbers from 2 to 10. The if condition checks whether i=5. If it is 6, the control goes to the next statement after the for loop. Hence it goes to the print statement End of Program.

Example Program 2

```
#Demo of break statement in Python
i=5
while i>0:
   if i==3:
      break
   print(i)
   i=i-1
print("End of Program")
```

Output 2

```
5
4
End of Program
```

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 returns the control to the beginning of the loop. The continue statement rejects all the remaining statements in the current iteration of the loop and moves the control back to the loop. The continue statement can be used in both while and for loops. Fig. 3.7 shows the flow chart of continue statement.

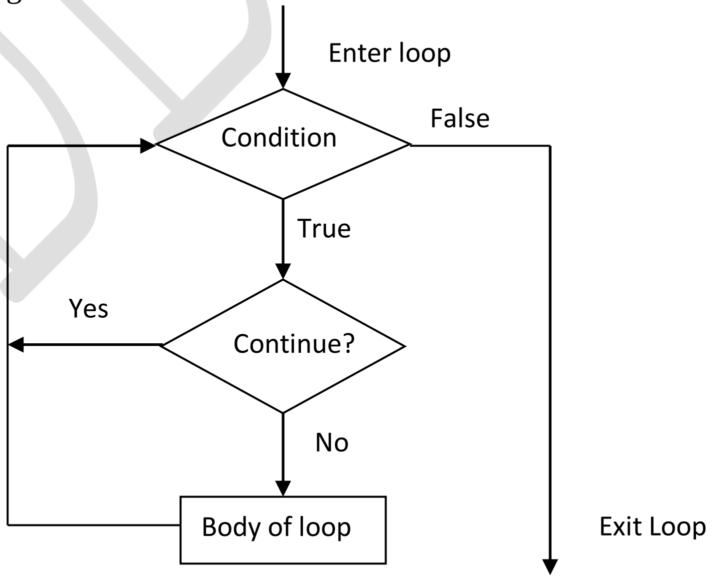

Fig. 3.7 Flow chart of continue

Example Program

```
# Demo of continue in Python
for letter in 'abcd':
   if letter =='c':
      continue
   print (letter)
```

Output

```
a
b
d
```

Illustration

When the letter=c, the continue statement is executed and the control goes to the beginning of the loop, bypasses the rest of the statements in the body of the loop. In the output the letters from "a" to "d" except "c" gets printed.

3.4.3 pass statement

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

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

Example

```
for letter in 'Python':
         if letter == 'h':
                  pass
                  print('This is pass block')
         print('Current Letter :', letter)
print ("Good bye!")
```

Output

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

3.5 TYPES OF LOOPS

There are different types of loops depending on the position at which condition is checked.

3.5.1 Infinite Loop

A loop becomes infinite loop if a **condition never becomes False**. This results in a loop that never ends. Such a loop is called an infinite loop. We can create an infinite loop using while statement. If the condition of while loop is always True, we get an infinite loop. We must use while loops with caution because of the possibility that the condition may never resolves to a False value.

Example Program

```
count=1
while count==1:
   n=input("Enter a Number:")
   print("Number",n)
```

Output

```
Enter a Number:
1
Number 1 Enter a Number:
2
Number 2 Enter a Number:
1
Number 1
…….
```

This will continue running unless you give CTRL-C to exit from the loop.

3.5.2 Loops with condition at the top

This is a normal while loop without break statements. The condition of the while loop is at the top and the loop terminates when this condition is False. This loop is already explained in Section 3.2.5.

3.5.3 Loop with condition in the middle

This kind of loop can be implemented using an infinite loop along with a conditional break in between the body of the loop. Fig. 3.8 shows the flow chart of a loop with condition in the middle.

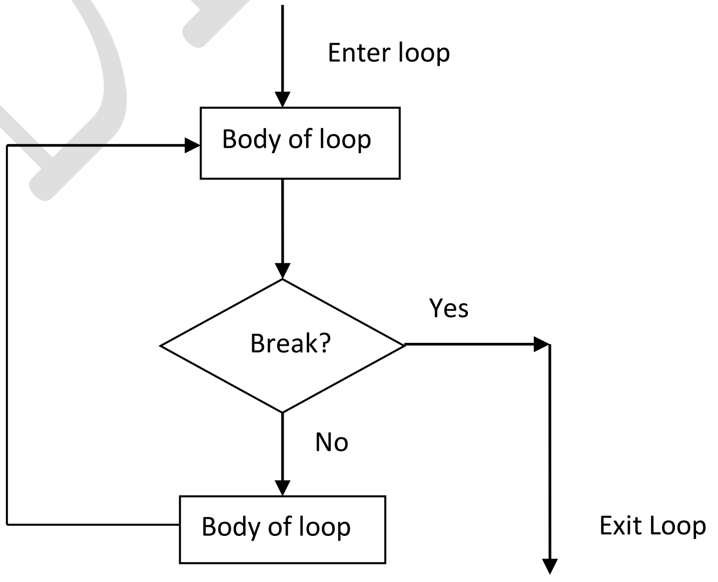

Fig. 3.8 Loop with condition

Example Program

```
vowels='aeiou'
while True:
   v=input ("Enter a letter:")
   if v in vowels:
      print (v, "is a vowel")
      break
   print("This is not vowel, ENter another letter")
print("End of Program")
```

Output

```
Enter a letter:
q
This is not vowel, ENter another letter
Enter a letter:
a
a is a vowel
End of Program
```

3.5.4 Loop with condition at the bottom

This kind of loop ensures that the body of the loop is executed at least once. It can be implemented using an infinite loop along with a conditional break at the end. This is similar to the do...while loop in C. Fig. 3.9 shows the flow chart of loop with condition at the bottom.

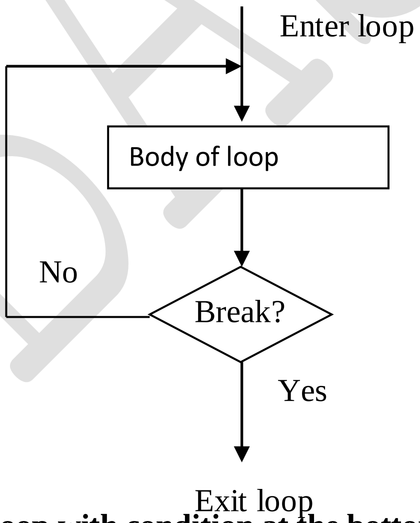

Fig 3.9 Loop with condition at the bottom

Example Program

```
#Demo of loop with condition at the bottom
choice=0
a,b=6,3
while choice !=3:
   print("Menu")
   print("1. Addition")
   print("2. Subtraction")
   print("3. exit")
```

```
    choice =int (input("enter your choice:"))
    if choice==1:
        print ("sum=", (a+b))
    if choice==2:
        print("Differences=", (a-b))
    if choice==3:
        break
print("End of Program")
```

Output

```
Menu
1. Addition
2. Subtraction
3. exit
enter your choice:
1
sum= 9
Menu
1. Addition
2. Subtraction
3. exit
enter your choice:
3
End of Program
```