CHAITANYA BHARATHI INSTITUTE OF TECHNOLOGY (A)
Kokapet (Village), Gandipet, Hyderabad, Telangana-500075. www.cbit.ac.in
ISO Certified 9001:2015

COMMITTED TO RESEARCH, INNOVATION AND EDUCATION 42 years

Course Code: **20CS C03**

Course Name: Object Oriented Programming

Instructor's Name:

Course Objective: The objectives of this course are

1. Describe the principles of Object-Oriented Programming.
2. Enable the students to solve problems using OOPs features.
3. Debugging in programs and files.
4. Use of library modules to develop GUI applications.

Course Outcome: On Successful completion of the course students will be able to:

1. Demonstrate the concepts of Object-Oriented Programming languages to solve problems.
2. Apply the constructs like selection, repetition, functions and packages to modularize the programs.
3. Design and build applications with classes/modules.
4. Find and rectify coding errors in a program to assess and improve performance.
5. Develop packages for solving simple real-world problems.
6. Analyze and use appropriate library software to create graphical interface, mathematical software.

Unit-I

Lecture-No	Topic	Learning Objective
01	Computer Programming and Programming Languages	Able to get the knowledge of basics of computers and programs
02	Programming Paradigms	Able to understand the division of languages
03	Features of Object Oriented Programming, Merits and Demerits of OOPs	Able to understand the reason to learn OOP in curriculum.
04	Features of Python, Variables, Identifiers	Able to understand the Python basics.

05	Datatypes, Input/ Output operations	Able to understand the flow of data from and to system.
06	Operators	Able to understand the operators.
07	Expressions, Operations on Strings, Type Conversion	Able to write and evaluate proper expressions.

Computer programming and programming languages

Computer program is a sequence of instructions written in some computer language to perform a particular task. The act of writing computer programs is called computer programming.

Programming Languages: language is designed to implement an algorithm that can be performed by a computer to solve a problem.

Programming languages are also called high level languages such as BASIC(Beginners All purpose Symbolic Instruction Code), C, C++, COBOL(Common Business Oriented Language), FORTRAN(Formula Translator), Python, Ada, and Pascal etc.

Machine language: language which can understand only by computer consists of
only numbers 1's and 0's.

Assembly language: it is a low level language for the micro processor and other programmable devices. Symbolic names are used to represent the machine language.

High level language: every human being can read this language like english statements. Ex: all programming languages.

Generations of programming languages

First genenration (machine level language):

- Machine language was used to program the first stored-program computer systems.
- In 1950s, each computer had its own native language, and programmers had primitive systems for combining numbers to represent instructions such as add or subtract.
- In machine language, all instructions , memory locations, numbers and characters are represented in strings of 0's and 1s.

Second generation (assembly language):

- It consists of series of statements and the statement consists of label, operation code and one or more operands.
- It needs assembler to convert code to machine language.
- Ex mov AX,4 ; MOV BX, 6 ; ADD AX,BX;

Advantages	Disadvantages

It is easy to understand	Code is machine dependent and thus non portable
Easy to write in AL than ML	Programmers should have a good knowledge of the hardware internal architechture of the CPU
It is easy to detect and correct errors	The code cannot be directly executed by the computer, as it needs assembler to convert AL to ML
It is easy to modify	
It is less prone to errors	

Third generation (high level languages)

Advantages	Disadvantages
The code is machine dependent	Code may not be optimized
It is easy to learn and use the language	The code is less efficient
There are few errors	It is difficult to write a code that controls the CPU, memory and registers.
It is easy to document and understand the code	
It is easy to maintain the code	
It is easy to detect and correct errors	
Note: python ruby and perl are 3GL languages that combine some 4GL abilities within a general purpose 3GL environment.	

Fourth generation

Characteristics of 4GL

- the instructions of the code are written in english like sentences.
- they are nonprocedural, so users concentrate on ‘what’ instead of ‘how’ aspect of the task.
- the code written in a 4GL is easy to maintain.
- the code written in 4GL become 10 times more productive that 3GL as fewer lines of code is written.

(ex: query language(SQL), which can be used to access the content of database using english like sentences). easier to learn than COBOL and C
to generate the report which displays the total number of students enrolled in each class and in each semester.
TABLE FILE ENROLMENT
SUM STUDENTS BY SEMESTER BY CLASS

- still evolving so difficult to define or standardize them.
- it doesnot make efficient use of a machine's resources.

Fifth generation

- centered on solving problems using constraints given to a program rather than using an algorithm.
- most constraint based and logic programming languages and some declarative languages form a part of 5GL.
- widely used in AI research and contains visual tools to help develop a program. ex: PROLOG, OPS5, Mercury , Visual Basic.

5GL s were built upon LISP ex:ICAD. there are many frame languages such as KL-ONE

Programming Paradigms

- Paradigm means way of doing something(like programming).
- When you program you need to follow some strategy that strategy is called programming paradigm.
- A programming paradigm is a style or way of programming that defines how the structure and elements of a program will be built.
- There are different styles of programming paradigms.

1. Monolithic programming
2. Procedural programming/imperative paradigm
3. Structured programming/modular paradigm
4. Object oriented programming
5. Logic oriented programming
6. Rule oriented programming
7. Constraint oriented programming.

Every paradigm has its own style of writing, capabilities and limitations

1. **Monolithic programming paradigm:**

- monolithic means a single block of statements.
- monolithic paradigm consists of sequence of statements uses global data.
- Useful for developing high level language (Basic) and assembly language (languages related to micro processors).
- Global data defined on top of the all the statements.

- Global data can be accessed by any sections of a program.
- How big the program all the statements should be in a same program
- There is no subroutines.
- We can jump from one section to another section within program using goto and jump instructions.

Drawbacks

- no security for the data
- since all statements embedded in single program, debugging is difficult.
- no reusability.

2. **Procedural programming/imperative paradigm:**

- Main program task is sub divided into sub task, each sub task implement as a separate procedure.
- Each procedure perform a specific task.
- Each procedure is also called as modules.
- The sequence of execution of instructions can be altered.
- There is no repetition of the code.
- Portan and cobol languages developed with this paradigm.

Advantages:

- Program can be controlled better than monolithic paradigm.
- Debugging is easy.
- global data can be accessed by the all the sub tasks of a program so no security for the data.

Disadvantages:

- Global data can be accessed by the all the sub tasks of a program so no security for the data.
- No reusability.

3. **Structured programming/modular programming:**

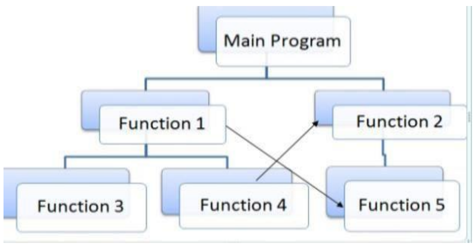
- Entire program is divided into modules.
- Each module has set of functions each function performs single identifiable sub task.
- Each function has a local data and can also access global data if required.
- Local data of one function cannot access from another function.
- Follows the top down approach.
- Removes the goto statement is replaced with else if etc.

Ex: C,C++,java, C#

Advantages:

1. Larger programs are implemented easilly.
2. Debugging is easy.
3. Security for the data as local data cannot access outside of the function.
4. Reusability.
5. Control of the project is easy
6. Maintenance is easy

Disadvantages:

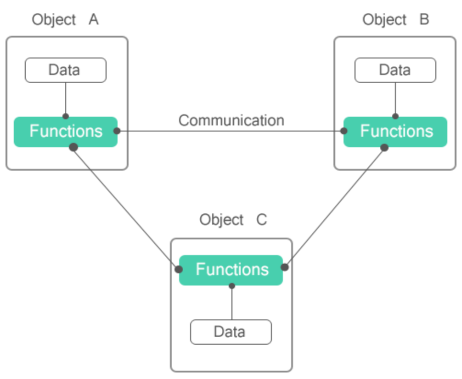


1. Global data can be shared among the modules.
2. Main focus on functions than the data

4. Object oriented programming paradigm:

- Models real world objects
- Uses bottom up approach
- Problem is decomposed into objects and build the data and functions around these objects.
- Program organized around the objects , grouped into classes.
- Data of the objects can be accessed only with the function associated with that objects.
- Objects can communicate with each other
- Empasize more on data.

Ex: c++,java, python, ruby and php.



Advantages:

- Provides security to data
- Reusability.
- Models the real world

Features of Oop

1. Class
2. Object
3. Data hiding
4. Data abstraction
5. Data encapsulation
6. Inheritance
7. Polymorphism
8. Containership or composition
9. Delegation
10. Method and message passing

Class: acts as blue print of an object which defines properties and methods example: planning of a building is a class and each building is an object

Object: Physical existence of a class in computer memory is called object.

example: consider a person is a class. Every person has name, Gender, legs, hands etc. People are like an objects. They have properties specified in a person Class.

Data hiding: hide data from others or outside world.

Data Abstraction: defines data and functions and hide implementation details is called data abstraction.

Data Encapsulation: the process of binding data and functions into a single unit is called data encapsulation

Encapsulation= data hiding+data abstraction

Inheritance:

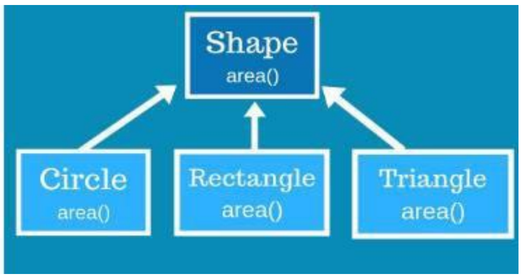
- Also known as **is a relationship**.
- one class acquires features of (properties and methods) another class is called

inheritance.

Advantages:

- Code reusability
- Save time and reduce project cost.
- Polymorphism
- Having several forms

- Polymorphism is related to methods.



Containership or composition

- A class contains object of another class is called containership or composition.

Increases reusability.

- Containership specifies has a relationship.

Example class car:

e=Engine() class engine:

– – –

– – – –

Delegation

- delegation is a relationship between objects where one object forward request to a method call to another method.

Advantage is change the behavior of an object during run time.

Method and message passing

- method is a group of statements performs a specific task and returns to caller.
- message passing is a communication between objects
- message passing involves name of object, function and information to be send.
- message passing is like calling methods with object and may send some parameters

Merits and Demerits of Oops

Merits

- Model real world objects.
- Redundancy is eliminated through inheritance
- Reduces development time due to reusability
- Secure data through data abstraction and data encapsulation
- Model real world objects
- Maintainability is easy.
- Easy to use.

Demerits

1. Requires more resources
2. Applicable for large and complex programs
3. Requires more skills to learn and implement the objects.

Application of Oops

- To create user interfaces such as design GUI for windows.
- Client- server systems

- To create data bases
- Artificial interlligence
- Automation systems
- Network programming, routers, and firewalls
- Computer aided design systems

Introduction to Python

1. Simple and easy to learn
   Python is a simple programming language as we can read python statements like a english language.
   We can write python programs with less code. Hence more readability and simplicity.
   Syntaxes of python are very easy to learn and
2. Freeware and open source
   we can use python software without any license and it is available free of cost.
   Source code of python is open to all that code can be customized as per our requirements.
   Ex: linux is open source it is customized to different versions like min, redhat, unix etc.
3. High level programming language
   Every human being can understand . Programmer need not aware of memory management and security etc.
4. Platform independent
   A python program can run on any operating system without rewriting agrain.
   Write once and run on any operating system.
5. Portability
   Python program can move form one platform to another platform without effecting performance and changes.
6. Dynamically typed language
   Python does not require any variable declaration. Based on type of value assigned to variable python will allocate the memory automatically.
7. Both procedure oriented and object oriented
   Python barrowed its syntaxes from c, and java so it supports both procedure and object oriented features.
8. Interpreted
   Line by line interprets the code need not compile it.
9. Extensible
   Python is extensible to any language
   We can use other language programs in python
   We can bring other languages functionalities and benefits into python where performance is required.
10. Embedded
    We can use python in any other languages .
    Python we can use in java called Jpython
11. Extensive library:
    python provides rich set of inbuilt libraries.
    Applications of Python
    1. Desktop applications (calculators, Excel etc) standalone applications
2. Web Applications (Django in python to develop web applications)
3. Database applications.
4. Machine learning
5. Networking applications
6. Games
7. Data analysis
8. Artificial Intelligence
9. IOT applications
10. Mobile based applications.

Currently following companies using python
- Google, Youtube, Dropbox, NASA, stock exchange, national security.

Variables, Identifiers, Datatypes
Variables: reserved memory location where values stored
Identifiers: names used to identify a variable, function, class. Rules to follow for identifiers
1. Identifiers can be in lower case or uppercase
2. Digits can be used but should not start with digit
3. Underscore is allowed
4. Keywords not defined as identifiers
5. Identifiers are case sensitive cash is different from Cash
6. Identifiers should not start with $, #,% and numbers(0-9)

Literal constants: the values are defined directly in expression without assigning to variable is called literal constant.
ex 7+5- 7 and 5 are literal constants
Note: identifiers starts with __ two underscore symbols are also valid

Reserved words or key words

Words which are defined by python has some specific meaning and functionality are called reserved words or key words.

Python provides total 33 keywords

To find total keywords present in python run the following statements

```
import  keyword
keyword.kwlist
```

Comments in python
- # used to comment single line
- ‘’’ triple quoted string is used to comment multiple lines
  example
  ‘’’ this is a python programming ‘’’

Data types in python

Python does not require to specify the type explicitly. Based on value provide, the type will be assigned automatically. Hence python is called dynamically typed language.
There are 14 data types in python
1. int
2. float
3. str
4. complex
5. bool
6. bytes
7. bytesarray
8. range
9. list
10. tuple
11. set
12. dict

13. frozenset
14. None

int, float, bool, complex and str are fundamental data types in python

Built in functions in python

Several built in functions in python few are

- type(): gives type of the variable
- id(): address of a variable
- print(): prints a value

1. int data type:

stores the integral values Ex x=10

type(x) # <class 'int'>

- int values can also be represented in following ways

decimal(base-10)

a=10

binary form (base-2)

- represents numbers in 0 or 1
- binary numbers prefix with 0B or 0b Ex a=0B0001111/0b0001111

Octal(base-8)

- represents numbers from 0 to 7
- numbers prefix with zero followed by Big oh(O) or small oh(o) Ex x=0O3456 or 0o234

hexadecimal(base-16)

represents numbers from 0 to 15

numbers prefix with zero followed by capital X or smal x x= 0Xa1ab23 or 0xbb1122

- To convert one base to another python provides built in functions

bin(): converts any base to binary number

bin(15) # '0b1111'

oct(): converts any base to octal

hex(): converts any bae to hexadecimal.

float data type:

represents float values ex f=12.89

type(f) # <class 'float'>

we can represent float values in exponential form also f=1.2e3 #1.2×103

- we can also use capital E

Complex data type

to represent complex numbers

a+bj

x=2+30j y=3+4.3j

Complex data types has some built in attributes to retrieve real and imaginary part

- x=10+20j

x.real #10

x.imag # 20

bool data type:

stores True or False True stores value 1 False stores value 0

x=True # <class 'bool'>

y=False # <class 'bool'>

str data type:

- allows strings there is no char data type in python even single character treated as string data type

a string can be defined with single quote or double quotes

Ex: x=‘Durga Devi’
x=“Durga Devi”
triple single quotes(‘’’) or triple double quotes(“”’) are used to represent multiple lines

Slicing of strings:

[:] operator is called slice operator used to retrieve part of the string

Input and output statements
To read data dynamically from keyboard two methods are used
raw_input() # in python2
input() #python2 and python3
Example
#add two numbers read them from keyboard x=int(input(“enter x value”);
y= int(intput(“ente y value”);
Print (“adding two numbers=“, x+y)
Read multiple values from the keyboard at a time
Use split method Example
a,b,c=[int(x) for x in input(“Enter two numbers”).split()]
or
#we can also use eval() function
a,b,c=[eval(x) for x in input(“enter two numbers”).split()] print (“the sum is=“, a+b+c)
print() method
print() method is used to output the data
Example:
A,b,c=10,20,30
Print(“the values are”,abc,sep=’:’)
Output 10:20:30

print(1,2,3,4)
# Output: 1 2 3
4
print(1,2,3,4,se
p='*') #
Output:
1*2*3*4

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

Formatting strings are same as your c programming
x=89.1234567
print("x value is %3.2f" %x)
a,b,c=10,20,30
print("a,b,c values are %i %i %i" %(a,b,c))
s="CBIT" # string type l=[10,20,30,40] # list type print("Hello %s the list is %s" %(s,l))

Output Hello CBIT the list is [10,20,30,40]
you can replace format string using format function

- syntax str.format(arguments)

Example name="SreeRam" Rollno=99 branch="CSE"

print("Hello{0} your rollno is {1} and you belong to {2}
        branch".format(name,Rollno,branch))

Output

HelloSreeRam your rollno is 99 and you belong to CSE branch

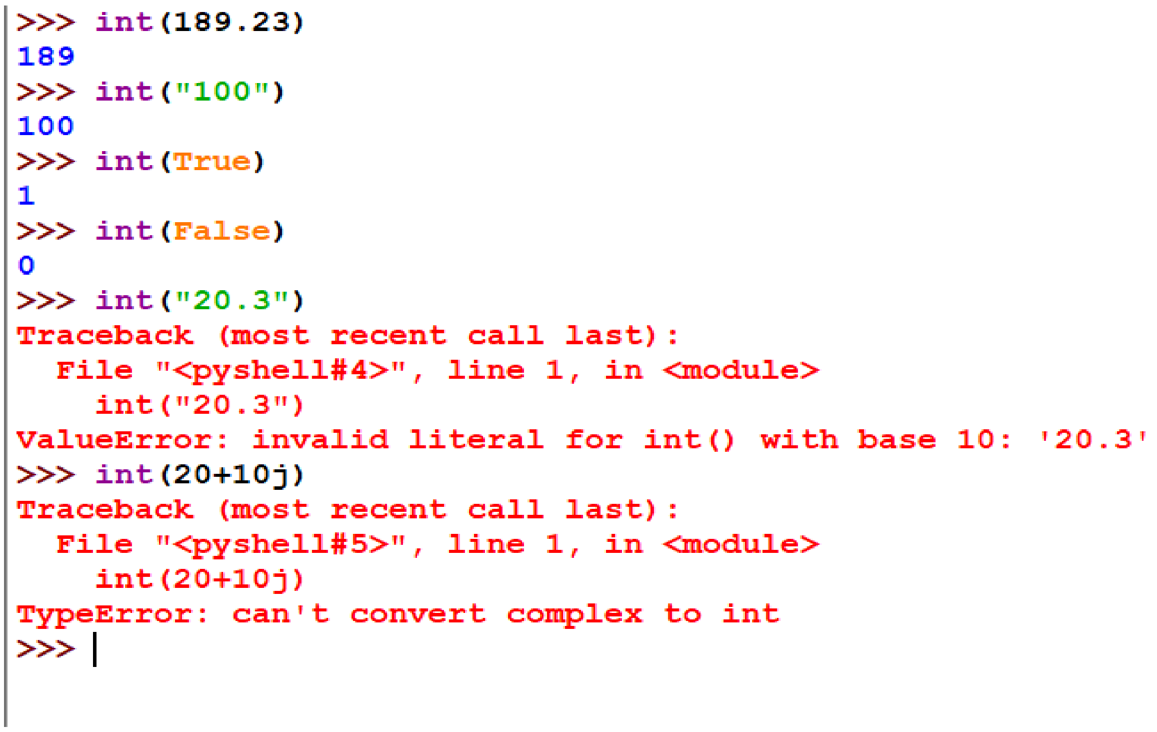
Type Casting

Type casting: converting one data type to another is called type casting or type coersion.

- Following are built in functions for type casting

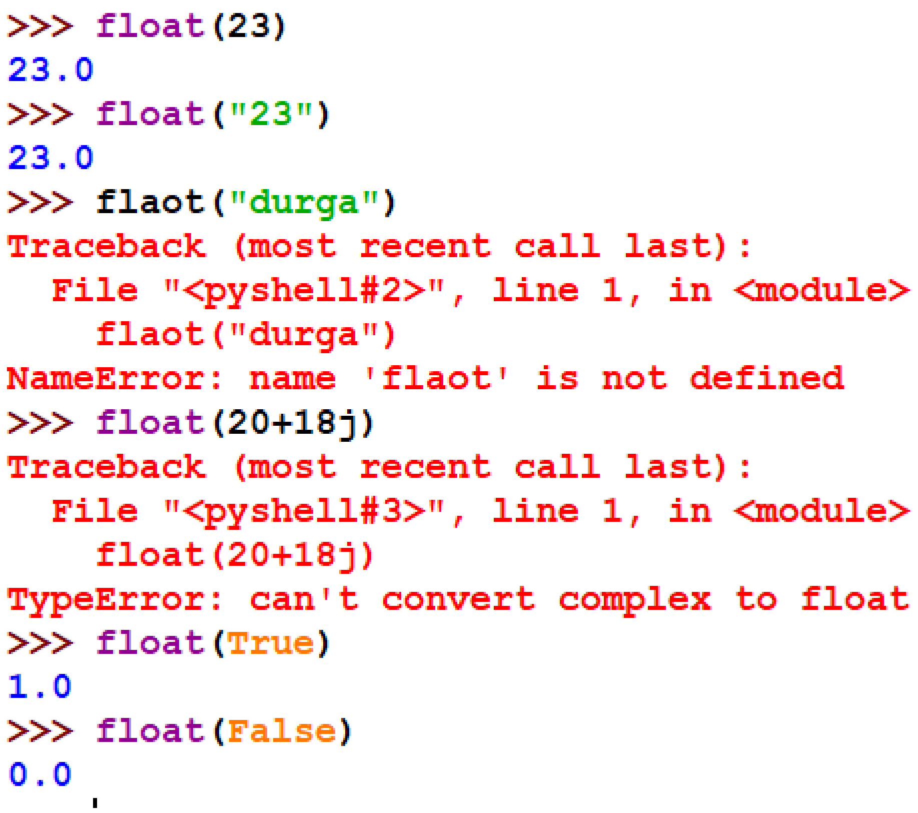
1. int():

- We can convert any type to int except the complex type.
- If str has base -10, str can also be converted into the int type



```
>>> int(189.23)
189
>>> int("100")
100
>>> int(True)
1
>>> int(False)
0
>>> int("20.3")
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    int("20.3")
ValueError: invalid literal for int() with base 10: '20.3'
>>> int(20+10j)
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    int(20+10j)
TypeError: can't convert complex to int
>>>
```

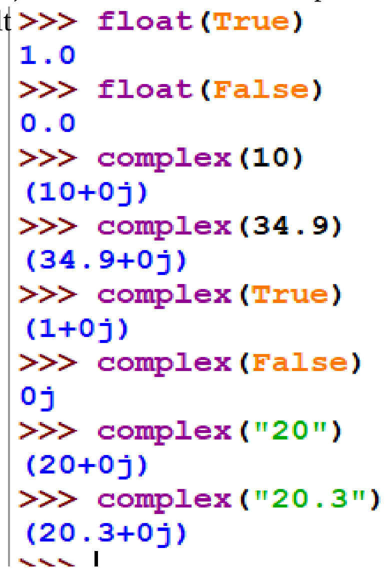
2. float(): converts other type value to float value



```
>>> float(23)
23.0
>>> float("23")
23.0
>>> flaot("durga")
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    flaot("durga")
NameError: name 'flaot' is not defined
>>> float(20+18j)
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    float(20+18j)
TypeError: can't convert complex to float
>>> float(True)
1.0
>>> float(False)
0.0
```

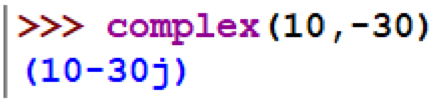
Note: complex type cannot convert to float

3. complex(x): converts the x into complex real part and imaginary part is zero default



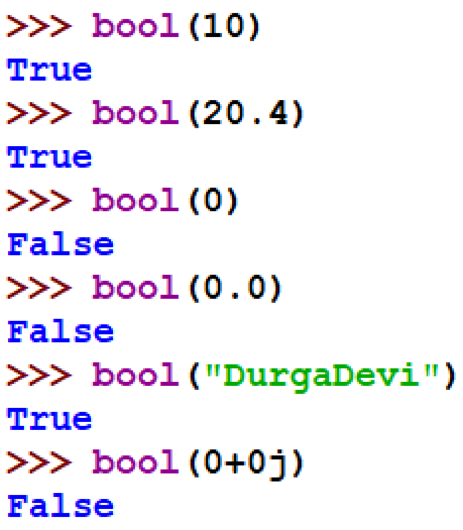
```
>>> float(True)
1.0
>>> float(False)
0.0
>>> complex(10)
(10+0j)
>>> complex(34.9)
(34.9+0j)
>>> complex(True)
(1+0j)
>>> complex(False)
0j
>>> complex("20")
(20+0j)
>>> complex("20.3")
(20.3+0j)
```

- complex(x,y): converts x into real part and y in imaginary part

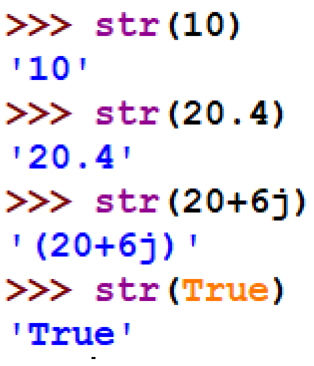


```
>>> complex(10,-30)
(10-30j)
```

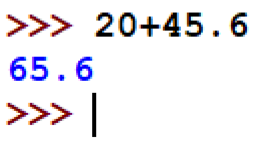
4. bool(): converts to bool type value


```
>>> bool(10)
True
>>> bool(20.4)
True
>>> bool(0)
False
>>> bool(0.0)
False
>>> bool("DurgaDevi")
True
>>> bool(0+0j)
False
```

5. str(): convert to string


```
>>> str(10)
'10'
>>> str(20.4)
'20.4'
>>> str(20+6j)
'(20+6j)'
>>> str(True)
'True'
```

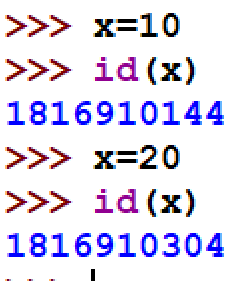
Type coersion: convert type to other type implicitly during the compile or run time


```
>>> 20+45.6
65.6
>>> |
```

All fundamental data types are immutable
What is immutable?

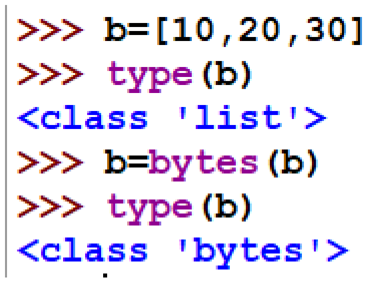
Once you create an object , the value of the object cannot be changed. If you are trying to change

- to change a new object is created and old object will be destroyed is called immutable.
- All fundamental data types like int,float,bool,str,complex are immutable.
- Ex


```
>>> x=10
>>> id(x)
1816910144
>>> x=20
>>> id(x)
1816910304
```

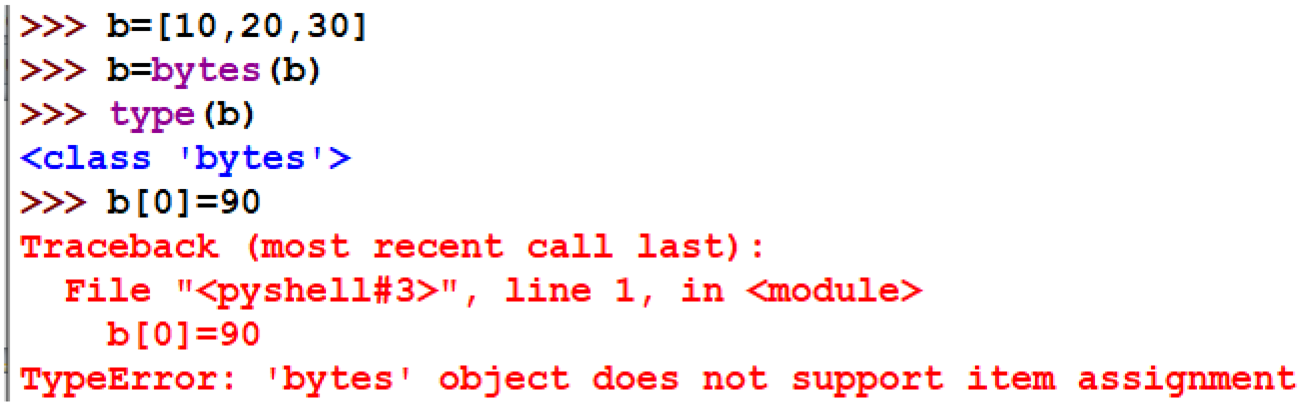
6. byte data type: represents byte numbers like an array

- Use built in function bytes() to convert into byte data type
- Allows only numbers from 0 to 255



```
>>> b=[10,20,30]
>>> type(b)
<class 'list'>
>>> b=bytes(b)
>>> type(b)
<class 'bytes'>
```

- Once byte data type created value cannot be changed.



```
>>> b=[10,20,30]
>>> b=bytes(b)
>>> type(b)
<class 'bytes'>
>>> b[0]=90
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    b[0]=90
TypeError: 'bytes' object does not support item assignment
```

8. range(): built in method used to generate sequence
syntax: range(1,n): generate number from 1 to n-1 range(n): generate 0 to n-1

for x in range(1,10):
print(x)# prints 1 to 9 for x in range(10):
print(x) # prints0 to 9

List data type

- insertion order is preserved
- Duplicate values are allowed
- Null values are not allowed in the list
- Stores any type of values
- List of values enclosed in []

- to access list elements we can use either +ve or –ve index

Example

list=[10,20,30,40,’b’]

if you want to know the type of the data use, type(variablename)

Example

type(l)

output is <class 'list'>

index value of list starts with 0

ex l[0]   gives 10

Methods in list data type

- list.append(x) : append value at end of the list
- list.insert(i, x): insert element x in specified position **i**
- list.remove(x) : removes first item matched with value x else returns error.
- list.pop(i): removes item from specified position and returns the removed item.
- list.count(x): returns no of times the elements appeared
- list.clear(): removes all the elements from the list

tuple data type

- tuple is immutable but list is mutable
- Tuple is represented in rounded parenthesis

t=(10,20,30,40,’durga’,True)

Advantages of tuple

Since tuple are immutable, iterating through tuple is faster than with list. So there is a slight performance boost.

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

Creating tuples

```
t1=(1,2,”isha”)

Print(t1)

#nested tuple

print("\n nested tuples");

t2 = ("mouse", [8, 4, 6], (1, 2, 3))
```

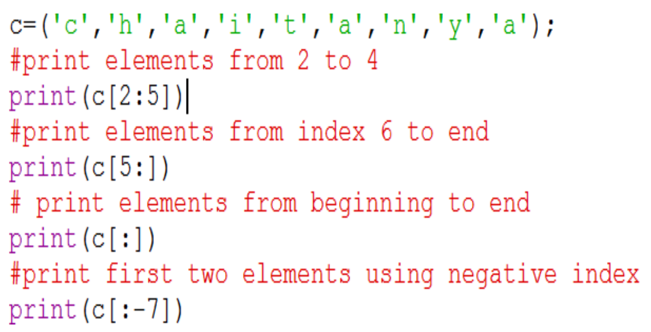
print(t2)

# tuples can be created without parenthesis is called tuple
packing print("\n tuple packing")

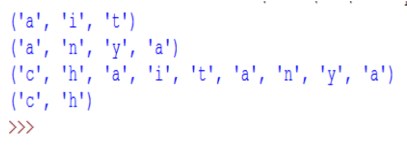
Accessing tuples

t2 = ("mouse", [8, 4, 6], (1, 2, 3))

- Accessing elements through negative index allows to access from last element
- -1 indicates last element



```
c=('c','h','a','i','t','a','n','y','a');
#print elements from 2 to 4
print(c[2:5])
#print elements from index 6 to end
print(c[5:])
# print elements from beginning to end
print(c[:])
#print first two elements using negative index
print(c[:-7])
```



```
('a', 'i', 't')
('a', 'n', 'y', 'a')
('c', 'h', 'a', 'i', 't', 'a', 'n', 'y', 'a')
('c', 'h')
>>>
```

Method	Meaning
all	Return True if all elements of the tuple are true (or if the tuple is empty).
count(x)	count no of elements that are equal to x
index(x)	Finds index of a specified element x
len	returns no of elements
max()	largest element in tuple
min()	smallest element in tuple
sum()	sum of all the elements of tuple
sorted()	returns new sorted tuple (does not sort tuple itself)

Built-in methods

Slicing operator

- We can access range of elements in a tuple by using the slicing operator -

Set Data Type

- Order does not preserved
- Duplicates not allowed
- Set will be represented in { }
- Does not support indexing
- It can have any number of items and they may be of different types (integer, float, tuple, string etc.). But a set cannot have a mutable element, like list, set or dictionary , as its element.

Properties of set data type

- no indexing
- no slicing
- no order
- no duplicates
- it is mutable
- growable.
- Allows mixed data types except mutable elements

  Ex. s={1,2,3,"CBIT",(10,20,30)}

To create empty set use set() function

x=set()

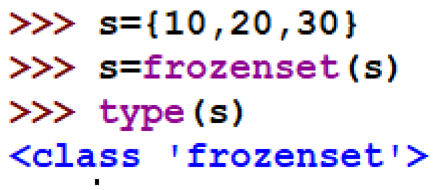
Method	Meaning
add(x)	Add element x
discard(x)	discard element x from set
clear()	removes all the elements
remove(x)	same as discard(), removes specified element
pop()	removes first element and returns removed element
copy()	Copies elements
union	union of two sets as new set
difference	Difference of two sets as new set
symmetric_difference	symetric_ difference of two sets as new set

Methods in set

frozen set data type:

- same as set data type but it is immutable
- can not use add or remove functions

- Example:

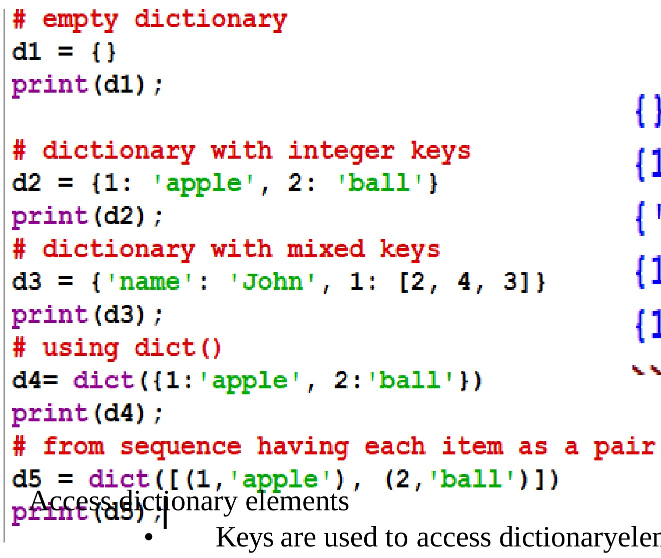


```
>>> s={10,20,30}
>>> s=frozenset(s)
>>> type(s)
<class 'frozenset'>
```

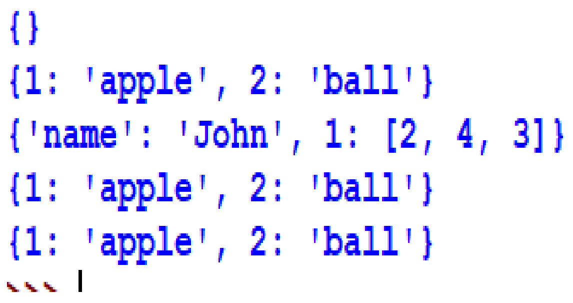
Dictionary datatype

1. A dictionary has a key : value pair
2. Order does not preserve
3. Values are accessed through key.
4. Keys in dictionary should be immutable (string, number or tuple with immutable elements) and must be unique.
5. Elements are placed in { } separated by comma
6. Key and its corresponding value separated with :
7. We can create a dictionary by using built in function dict()

Example:



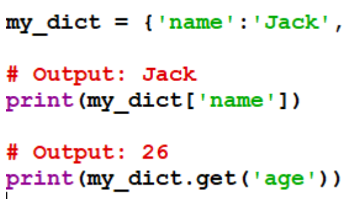
```
# empty dictionary
d1 = {}
print(d1);

# dictionary with integer keys
d2 = {1: 'apple', 2: 'ball'}
print(d2);
# dictionary with mixed keys
d3 = {'name': 'John', 1: [2, 4, 3]}
print(d3);
# using dict()
d4= dict({1:'apple', 2:'ball'})
print(d4);
# from sequence having each item as a pair
d5 = dict([(1,'apple'), (2,'ball')])
print(d5);
```



```
{}
{1: 'apple', 2: 'ball'}
{'name': 'John', 1: [2, 4, 3]}
{1: 'apple', 2: 'ball'}
{1: 'apple', 2: 'ball'}
```

Access dictionary elements

- Keys are used to access dictionaryelements
- Key can be used either within the square brackets or with get() method.
- When get() method is used it returnsnone when key element is not found.

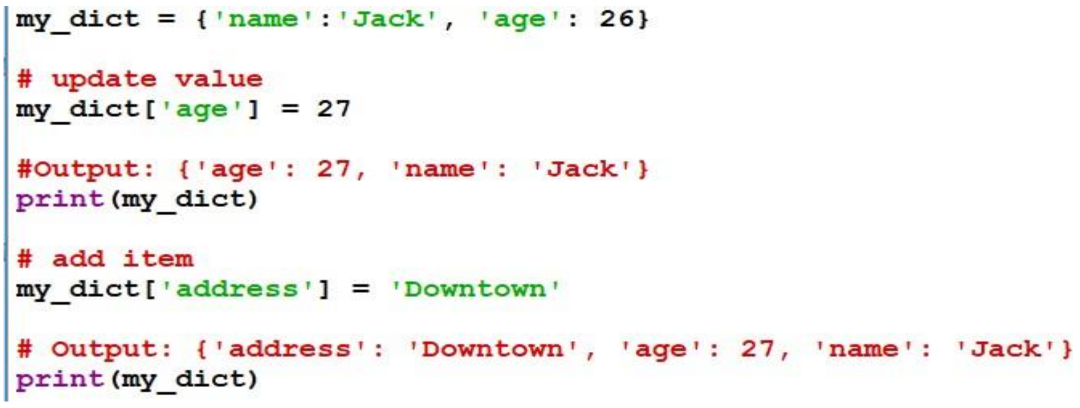


```
my_dict = {'name':'Jack', 'age': 26}

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

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

Change or add elements in dictionary

- Dictionaries are mutable so that we can add or change elementsin dictionary

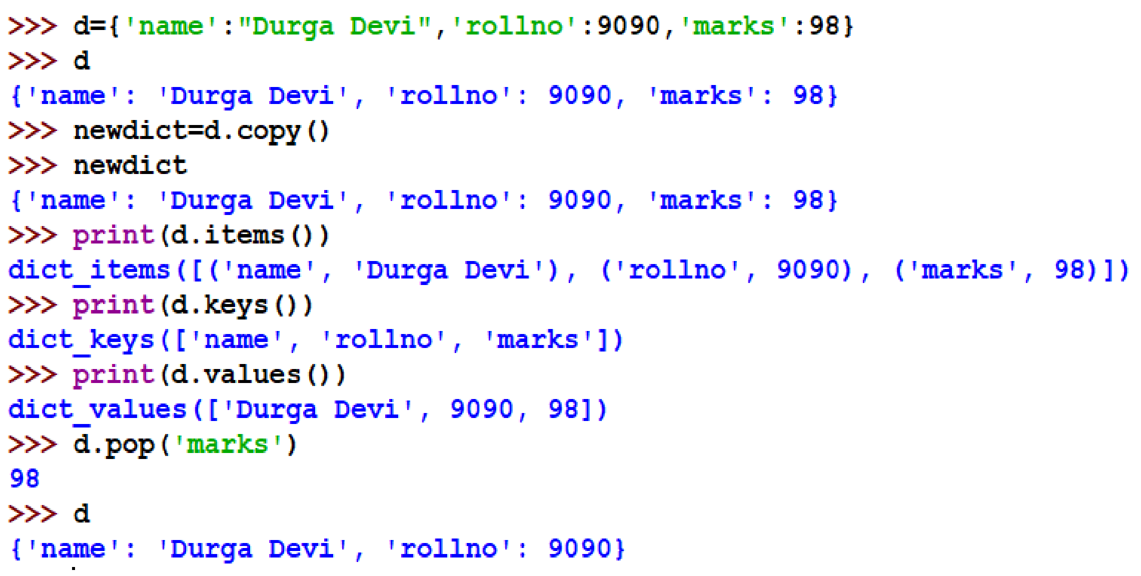
Example



```
my_dict = {'name':'Jack', 'age': 26}

# update value
my_dict['age'] = 27

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

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

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

Built-in Functions

name	Purpose
copy()	Copies the dictionary values and return new dictionary
items()	Return a new view of the dictionary's items (key, value).
keys()	Return a new view of the dictionary's keys.
values()	Return a new view of the dictionary's values
pop(key):	deletes specified key value and returns the value.
popitem():	deletes and return arbitrary element.
del()	delete particular element or entire dictionary



```
>>> d={'name':"Durga Devi",'rollno':9090,'marks':98}
>>> d
{'name': 'Durga Devi', 'rollno': 9090, 'marks': 98}
>>> newdict=d.copy()
>>> newdict
{'name': 'Durga Devi', 'rollno': 9090, 'marks': 98}
>>> print(d.items())
dict_items([('name', 'Durga Devi'), ('rollno', 9090), ('marks', 98)])
>>> print(d.keys())
dict_keys(['name', 'rollno', 'marks'])
>>> print(d.values())
dict_values(['Durga Devi', 9090, 98])
>>> d.pop('marks')
98
>>> d
{'name': 'Durga Devi', 'rollno': 9090}
```

Operators and expressions

Operator is symbol that performs certain operations

Types of operators:

Arithmetic operators

Relationalor comparison operators

Logical operators

Bitwise operators

Assignment operators

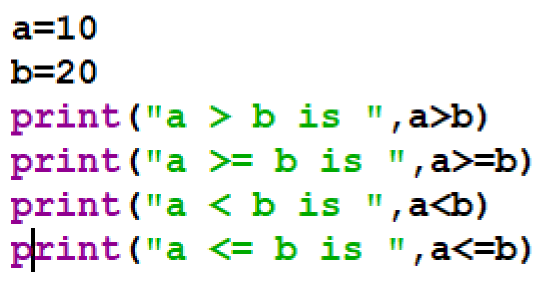
Special operators

Arithmetic Operators

operator	Description	Example	Output
+	**Addition**	**print(a+b)**	**30**
-	**Substraction**	**print(a-b)**	**-10**
*****	**Multiplication**	**print(a*b)**	**200**
/	**Division gives output in float**	**print(a/b)**	**0.5**
//	**floor division** **If both inputs are int output is int . If any in float result in float**	**print(a//b)**	**0**
%	**Modulus**	**print(a%b)**	**10**
**	**Exponential**	**print(a**b)**	**a=2** **b=3** **a**b=8**

Relational operators

- compare the objectvalues
- >,<,<=,>=, != ,==
- result into True orFalse



```
a=10
b=20
print("a > b is ",a>b)
print("a >= b is ",a>=b)
print("a < b is ",a<b)
print("a <= b is ",a<=b)
```

Logical operator

- and, or andnot
- Booleantype
  - and -> if both arguments are True then result is True
  - or -> if at least one argument is True then result is True
- Non- booleantype
  - 0- False
  - non zero-True
  - empty string is always treatedas False

- x and y: if any one of the argument is false that argument will return. ex: 0 and 20 #0

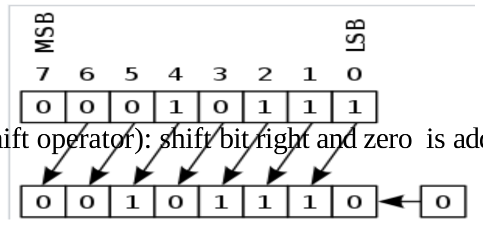
- if x and y are true second argumentwill return

ex: 10 and 20 #20

Bitwise Operator

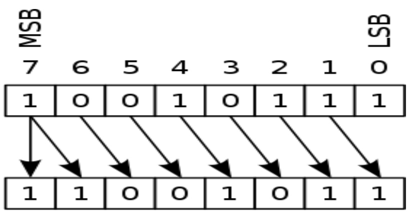
- we can apply bitwise operators on int and bool type only
- &,|,^,~,<<,>>
- &: if both bits are 1 then result is 1 else 0
- | : at least one bit is 1 result is 1 else 0
- ^: if both the bits are different then result is 1 otherwise result is 0
- ~: 0->1, 1->0
- <<(left shift operator): shift bit left and zero is added at the least significant bit position.

ex: print(2<<2)# 8 shift two bits to left when any number to be multiply by 2 then left shift operator to be used



- >>(right shift operator): shift bit right and zero is added at most significant bit position ex: print(4>>1)# 2

number is divided with the 2 when we use right shift operator.



Assignment Operator

- assigns value to a variable
- Wecan combine asignment operator with some other operator to form compound assignment operator.

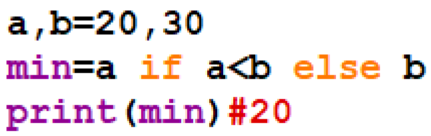
+=, -=, *=, /=, %=, //=, **= &=, |=, ^=, >>=, <<=

1. Ternary operator
2. Identity operator
3. Membership operator

Ternary Operator:

a= x if condition else y

If condition is true x will be assigned to a else y will be assigned.



```
a,b=20,30
min=a if a<b else b
print(min)#20
```

- Program to find minimum of three numbers a=int(input("Enter First Number:")) b=int(input("Enter Second Number:"))

c=int(input("Enter Third Number:"))

max=a if a>b and a>c else b if b>c else c

print("Maximum Value:",max)

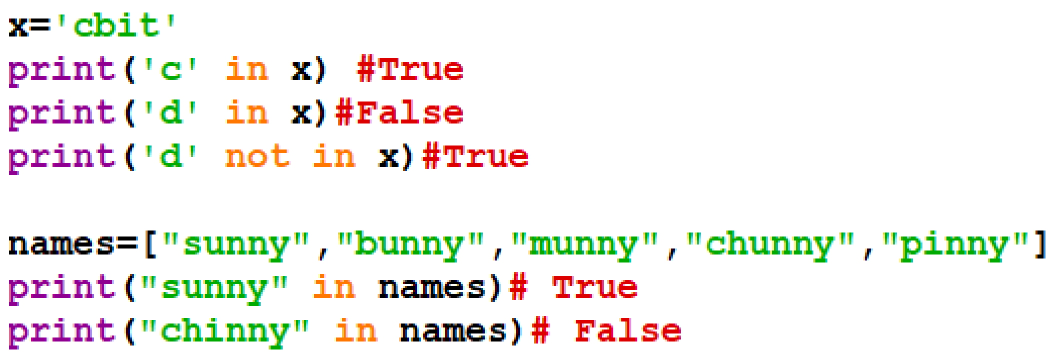
Identity Operator

- Uses is, is not keywords
- Used to compare address of 2 operands or objects

A=10
B=20
Print(a is b) #False

Membership Operator:

- To check whether the given object present in the give collection or not
- Collection can be (string, list, tuple, set or dict)
- Uses in and not in as keywords



```
x='cbit'
print('c' in x) #True
print('d' in x)#False
print('d' not in x)#True

names=["sunny","bunny","munny","chunny","pinny"]
print("sunny" in names)# True
print("chinny" in names)# False
```

Operator precedence chart
()
**
~, - unary operators
***/,%,//**
+, -
&
^ XOR
|
Relational operator
assignment operator
Is, is not

Essay Type questions

8. What are the immutable data types in python . Give example for each
9. Explain about basic data types in python.
10. What is the difference between mutable and immutable data types
11. List any built in functions in string and give example for each
12. What is a list object? What are the built in functions in list object
13. How do define a set object?what are the built in functions in set object
14. What are the built in functions in tuple and dictionary objects.
15. What is the use of tuple in python? Is tuple mutable? How to create tuple in python? Explain with an example.
16. Explain about programming paradigms ?

17. Explain about generation of programming languages
18. What are the built in methods for the type conversion?
19. State about logical operators in python.
20. Write about floor and division operator in python
21. What is command line arguments?
22. Relate strings and list
23. How to slice a list in python?
24. Write a python code to find largest of three numbers using ternary operator
25. Analyze string slicing. Explain with an example in python.

UNIT-II

Lecture-No	Topic	Learning Objective
01	Selection/Conditional Branching-I	Able to get insight into branching and conditional statements for writing complex programs
02	Loop Control Structures	Able to understand the usage of loop structures to write efficient programs.
06	Uses of functions, Function definition, function call	Able to understand the way of writing methods to modularize a large sized program.
07	Variable scope and Lifetime, Recursion	Able to understand the concept of recursion and its effect on time complexity.
08	Lambda functions, map, reduce, filter built-in functions	Able to understand the built-in functions and lambda functions to achieve a task done in lesser time.
09	Modules, Packages	Able to understand how to reuse a code using packages.

Introduction

A *control statement* is a statement that determines the control flow of a set of instructions, i.e., it decides the sequence in which the instructions in a program are to be executed.

Types of Control Statements are

1. Sequential Control Statements: A Python program is executed sequentially from the first line of the program to its last line.
2. Selection Control Statements: To execute only a selected set of statements.
3. Iterative Control Statements: To execute a set of statements repeatedly.

Decision making is anticipation of conditions occurring while execution of the program and specifying actions taken according to the conditions. Decision structures evaluate multiple expressions which produce TRUE or FALSE as outcome. You need to determine which action to take and which statements to execute if outcome is TRUE or FALSE otherwise.

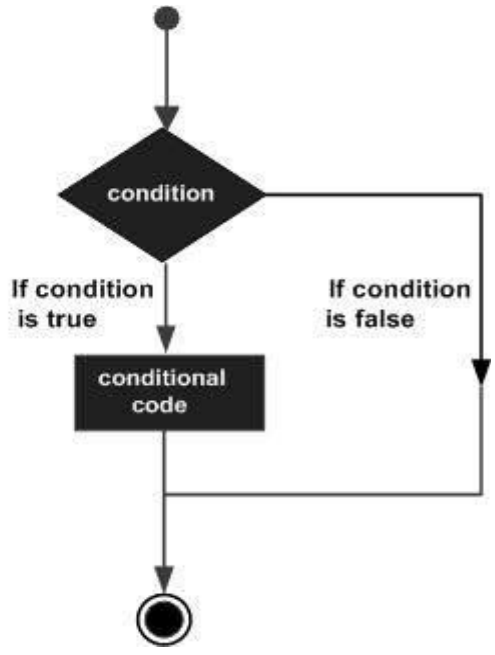
<u>Selection /conditional branching statements:</u>

If statement

The if statement contains a logical expression using which data is compared and a decision is made based on the result of the comparison.

```
SYNTAX OF IF STATEMENT

if test_expression:
     statement1
       ………..
    Statement n
statement x;
```

If the boolean expression evaluates to TRUE, then the block of statement(s) inside the if statement is executed. If boolean expression evaluates to FALSE, then the first set of code after the end of the if statement(s) is executed.



Example:

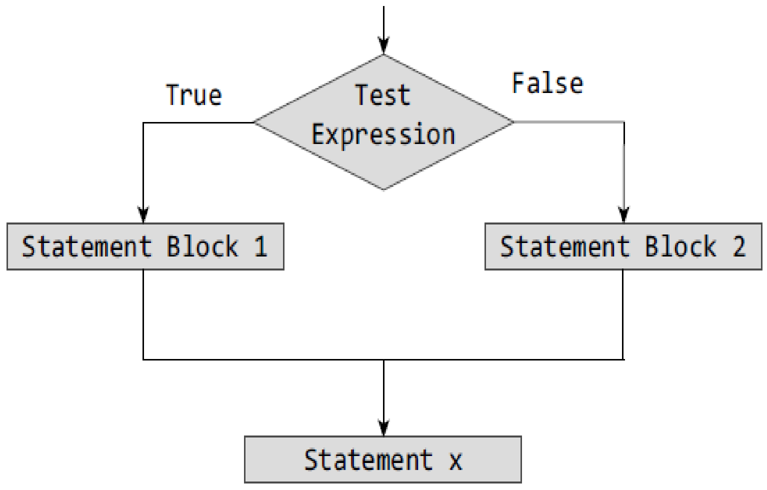
```
x = 10       #Initialize the value of x
if(x>0):     #test the value of x
   x = x+1     #Increment the value of x if it is > 0
print(x)      #Print the value of x

OUTPUT
x = 11
```

Else statement:

An else statement can be combined with an if statement. An else statement contains the block of code that executes if the conditional expression in the if statement resolves to 0 or a FALSE value.

The else statement is an optional statement and there could be at most only one else statement following if.

```
SYNTAX OF IF-ELSE STATEMENT

if (test expression):
        statement block 1
else:
       statement block 2
statement x;
```



Example

```
age = int(input("Enter the age : "))
if(age>=18):
   print("You are eligible to vote")
else:
       yrs = 18 - age
       print("You have to wait for another " + str(yrs) +" years to cast your vote")

OUTPUT
Enter the age : 10
You have to wait for another 8 years to cast your vote
```

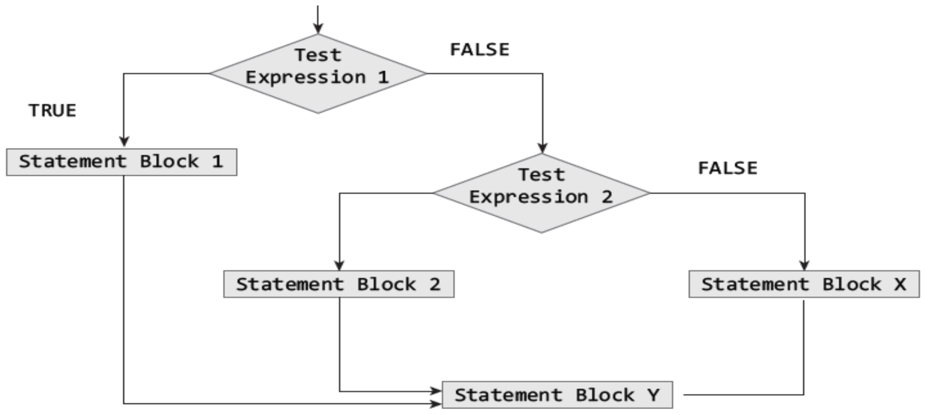
The elif Statement

The elif statement allows you to check multiple expressions for TRUE and execute a block of code as soon as one of the conditions evaluates to TRUE.

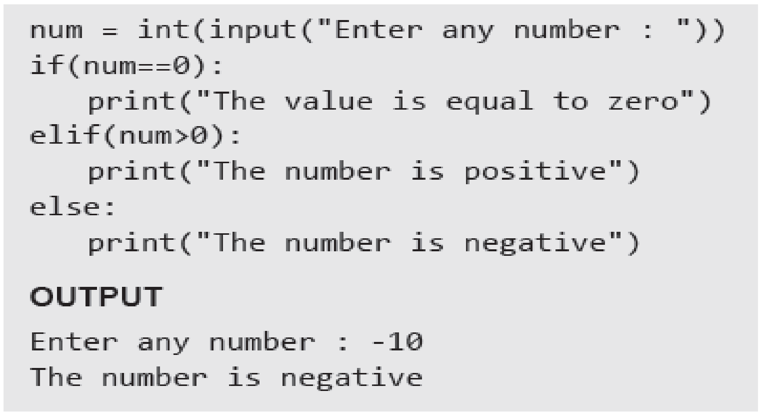
Similar to the else, the elif statement is optional. However, unlike else, for which there can be at most one statement, there can be an arbitrary number of elif statements following an if.

syntax

```
if expression1:
  statement(s)
elif expression2:
  statement(s)
elif expression3:
  statement(s)
else:
  statement(s)
```



Example



```
num = int(input("Enter any number : "))
if(num==0):
   print("The value is equal to zero")
elif(num>0):
   print("The number is positive")
else:
   print("The number is negative")
```

OUTPUT

```
Enter any number : -10
The number is negative
```

Nested if statements:

There may be a situation when you want to check for another condition after a condition resolves to true. In such a situation, you can use the nested if construct.

A statement that contains other statements is called a *compound statement.* To perform more complex checks, if statements can be nested, that is, can be placed one inside the other. In such a case, the inner if statement is the statement part of the outer one. Nested if statements are used to check if more than one conditions are satisfied.

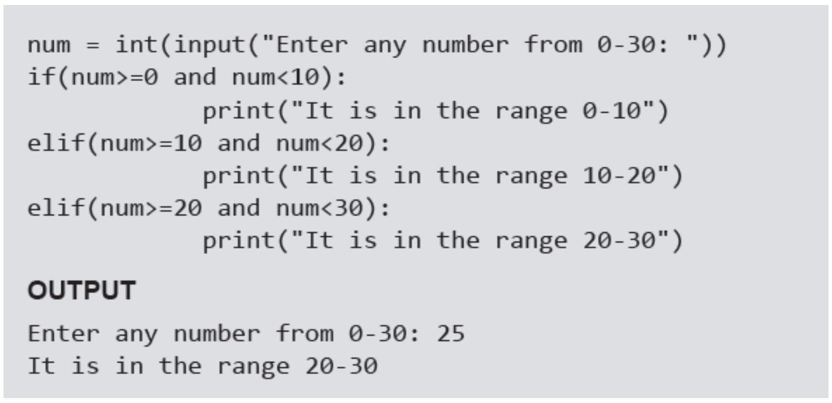
In a nested if construct, you can have an if...elif...else construct inside another if...elif...else construct.

Syntax

The syntax of the nested if...elif...else construct may be −

```
if expression1:
   statement(s)
   if expression2:
      statement(s)
   elif expression3:
      statement(s)
   elif expression4:
      statement(s)
   else:
      statement(s)
else:
   statement(s)
```

Example:



```
num = int(input("Enter any number from 0-30: "))
if(num>=0 and num<10):
            print("It is in the range 0-10")
elif(num>=10 and num<20):
            print("It is in the range 10-20")
elif(num>=20 and num<30):
            print("It is in the range 20-30")
```

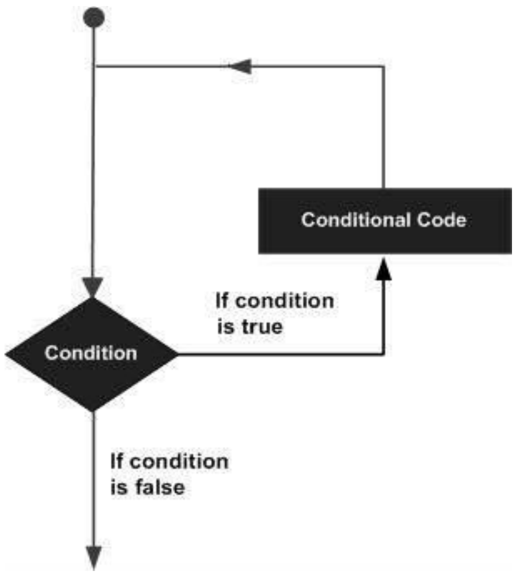
OUTPUT

```
Enter any number from 0-30: 25
It is in the range 20-30
```

<u>Basic loop control structures:</u>

In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on. There may be a situation when you need to execute a block of code several number of times.

Programming languages provide various control structures that allow for more complicated execution paths. A loop statement allows us to execute a statement or group of statements multiple times. The following diagram illustrates a loop statement.



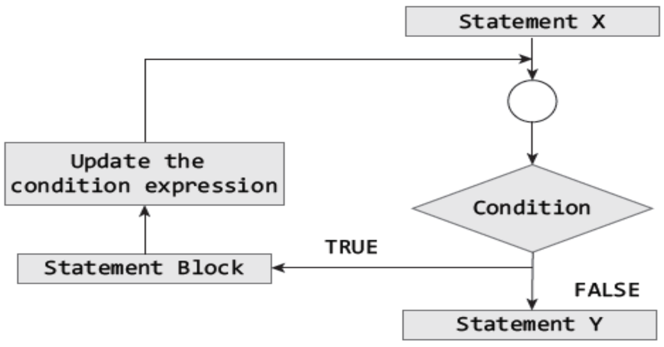
While loop:

A while loop statement in Python programming language repeatedly executes a target statement as long as a given condition is true.

```
SYNTAX OF WHILE LOOP
statement x
while (condition):
           statement block
statement y
```

Here, statement(s) may be a single statement or a block of statements. The condition may be any expression, and true is any non-zero value. The loop iterates while the condition is true.

When the condition becomes false, program control passes to the line immediately following the loop.



In Python, all the statements indented by the same number of character spaces after a programming construct are considered to be part of a single block of code. Python uses indentation as its method of grouping statements.

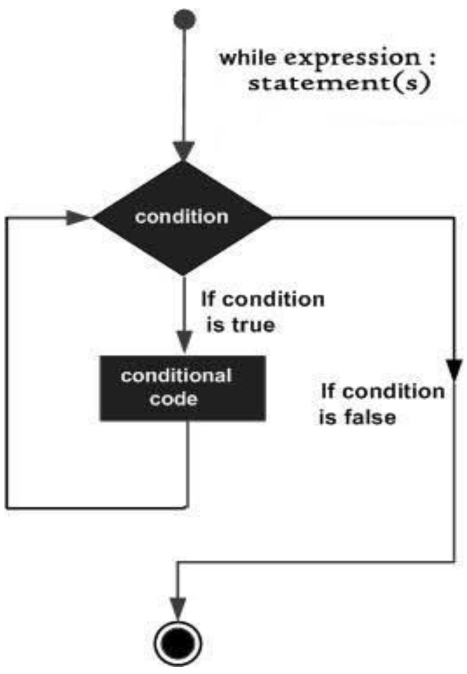
Here, key point of the while loop is that the loop might not ever run. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example:

```
i = 0
while(i<=10):
    print(i,end=" ")
    i = i+1
```

OUTPUT

```
0 1 2 3 4 5 6 7 8 9 10
```



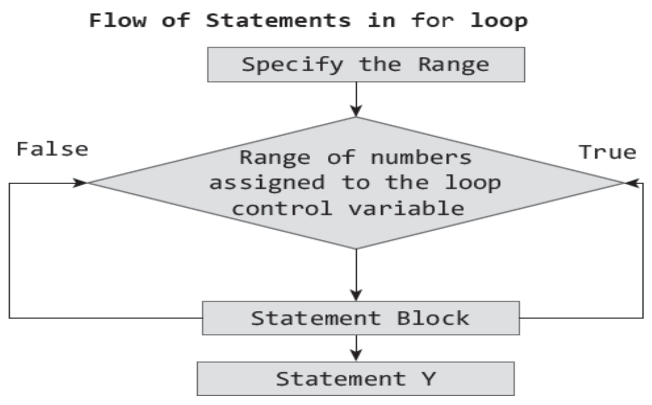
Here, key point of the while loop is that the loop might not ever run. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

For loop:

For loop provides a mechanism to repeat a task until a particular condition is True. It is usually known as a *determinate or definite loop* because the programmer knows exactly how many times the loop will repeat.

The for...in statement is a looping statement used in Python to iterate over a sequence of objects.

```
Syntax of for Loop

for loop_contol_var in sequence:
   statement block
```

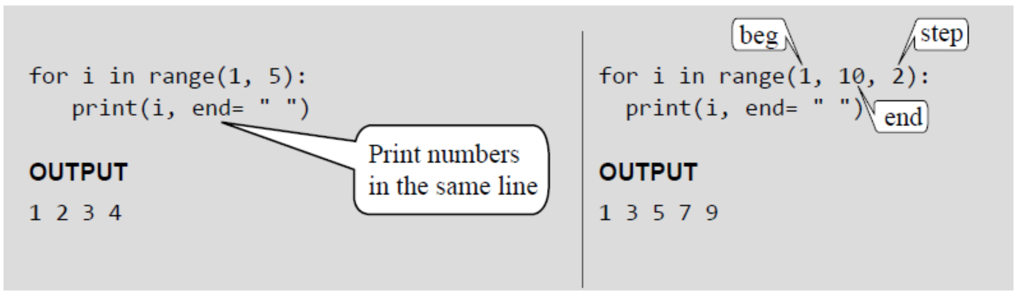

Flow of Statements in for loop
Specify the Range
False
True
Range of numbers assigned to the loop control variable
Statement Block
Statement Y

Range() Function

The range() function is a built-in function in Python that is used to iterate over a sequence of numbers. The syntax of range() is range(beg, end, [step])

The range() produces a sequence of numbers starting with beg (inclusive) and ending with one less than the number end. The step argument is option (that is why it is placed in brackets). By default, every number in the range is incremented by 1 but we can specify a different increment using step. It can be both negative and positive, but not zero.

Example:



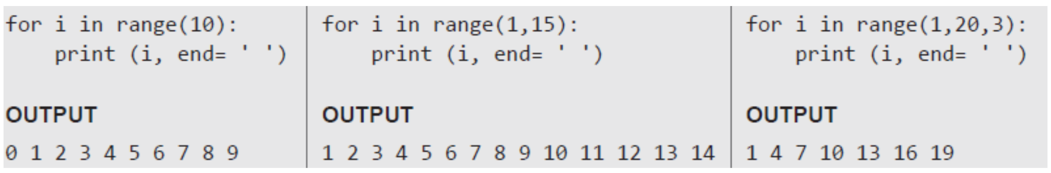
```
for i in range(1, 5):
   print(i, end= " ")

OUTPUT
1 2 3 4
```

```
for i in range(1, 10, 2):
  print(i, end= " ")

OUTPUT
1 3 5 7 9
```

If range() function is given a single argument, it produces an object with values from 0 to argument-1. For example: range(10) is equal to writing range(0, 10).

I. If range() is called with two arguments, it produces values from the first to the second. For example, range(0,10).

II. If range() has three arguments then the third argument specifies the interval of the sequence produced. In this case, the third argument must be an integer. For example, range(1,20,3).

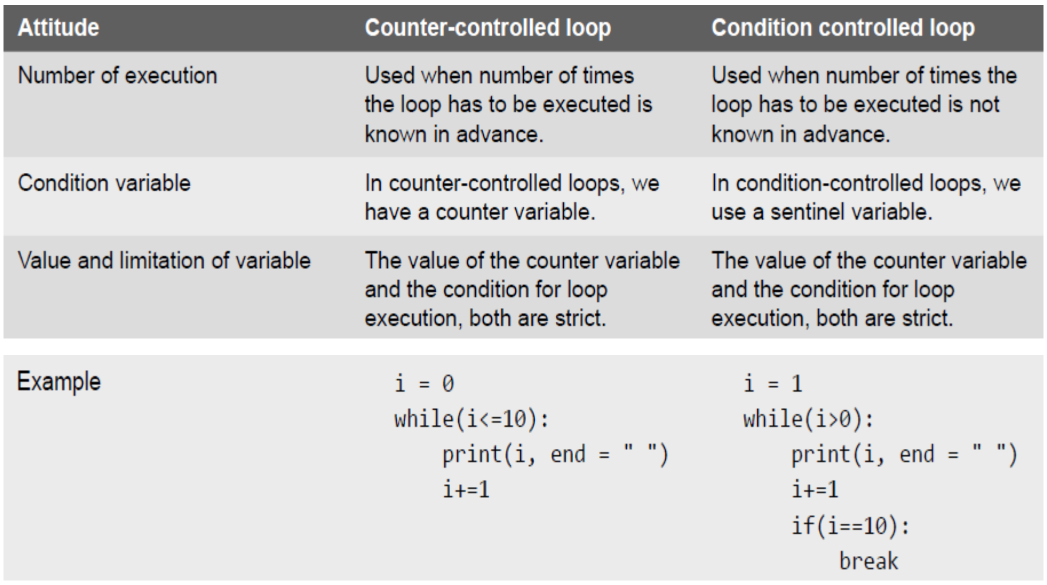


```
for i in range(10):
    print (i, end= ' ')

OUTPUT
0 1 2 3 4 5 6 7 8 9
```

```
for i in range(1,15):
    print (i, end= ' ')

OUTPUT
1 2 3 4 5 6 7 8 9 10 11 12 13 14
```

```
for i in range(1,20,3):
    print (i, end= ' ')

OUTPUT
1 4 7 10 13 16 19
```

Condition-controlled and Counter-controlled Loops



Attitude	Counter-controlled loop	Condition controlled loop
Number of execution	Used when number of times the loop has to be executed is known in advance.	Used when number of times the loop has to be executed is not known in advance.
Condition variable	In counter-controlled loops, we have a counter variable.	In condition-controlled loops, we use a sentinel variable.
Value and limitation of variable	The value of the counter variable and the condition for loop execution, both are strict.	The value of the counter variable and the condition for loop execution, both are strict.

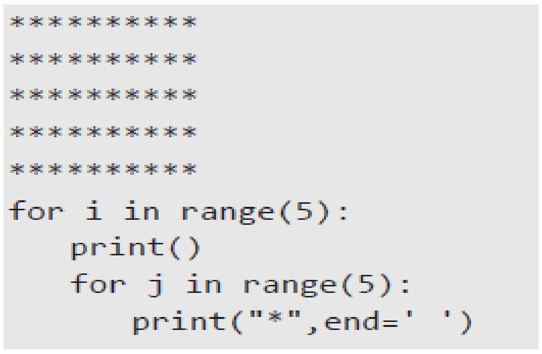
Example	`i = 0`
	`while(i<=10):`
	`    print(i, end = " ")`
	`    i+=1`
		`i = 1`
		`while(i>0):`
		`    print(i, end = " ")`
		`    i+=1`
		`    if(i==10):`
		`        break`

Nested Loops

Python allows its users to have nested loops, that is, loops that can be placed inside other loops. Although this feature will work with any loop like while loop as well as for loop.

A for loop can be used to control the number of times a particular set of statements will be executed. Another outer loop could be used to control the number of times that a whole loop is repeated. Loops should be properly indented to identify which statements are contained within each for statement.

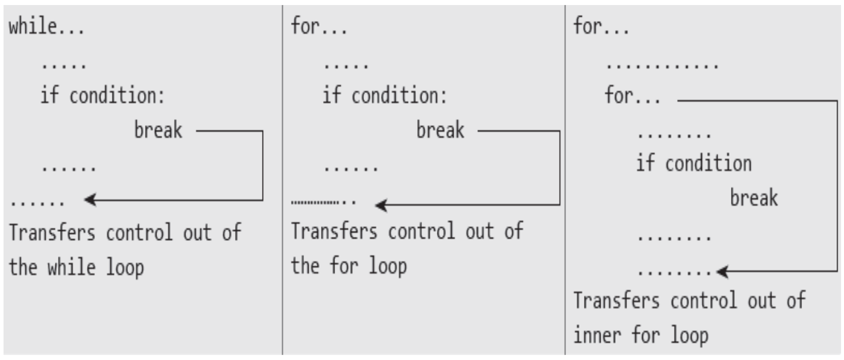
Example:



```
**********
**********
**********
**********
**********
for i in range(5):
    print()
    for j in range(5):
        print("*",end=' ')
```

The Break Statement

The *break* statement is used to terminate the execution of the nearest enclosing loop in which it appears. The break statement is widely used with for loop and while loop. When compiler encounters a break statement, the control passes to the statement that follows the loop in which the break statement appears.



```
while...
   .....
   if condition:
            break
   ......
......
Transfers control out of
the while loop
```

```
for...
   .....
   if condition:
            break
   ......
..........
Transfers control out of
the for loop
```

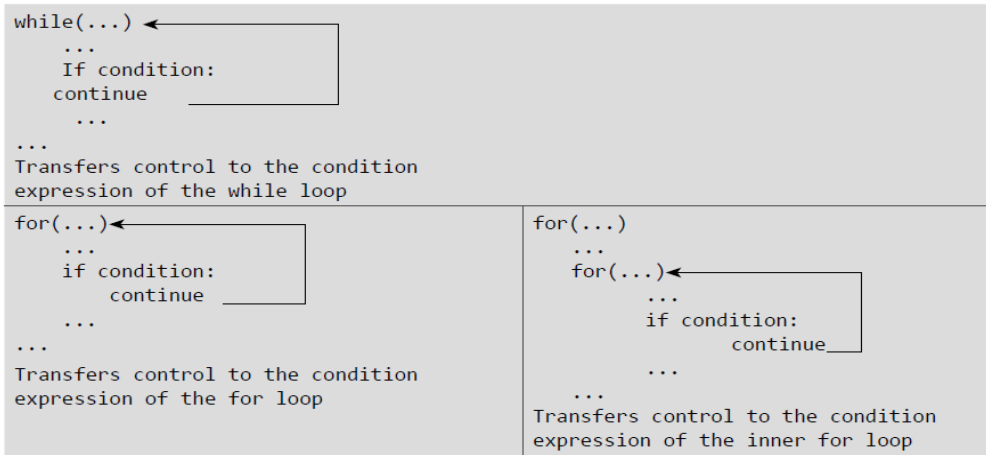
```
for...
   ...........
   for...
      ........
      if condition
               break
      ........
      ........
Transfers control out of
inner for loop
```

Example:

```
i = 1
while i <= 10:
        print(i, end=" ")
        if i==5:
                break
        i = i+1
print("\n Done")

OUTPUT
1 2 3 4 5
Done
```

The Continue Statement

Like the break statement, the continue statement can only appear in the body of a loop. When the compiler encounters a continue statement then the rest of the statements in the loop are skipped and the control is unconditionally transferred to the loop-continuation portion of the nearest enclosing loop.



```
while(...)
    ...
    If condition:
   continue
     ...
...
Transfers control to the condition
expression of the while loop
```

```
for(...)
    ...
    if condition:
        continue
    ...
...
Transfers control to the condition
expression of the for loop
```

```
for(...)
   ...
   for(...)
         ...
         if condition:
                continue
         ...
   ...
Transfers control to the condition
expression of the inner for loop
```

Example:

```
for i in range(1,11):
   if(i==5):
      continue
   print(i, end=" ")
print("\n Done")
```

OUTPUT

```
1 2 3 4 6 7 8 9 10
Done
```

The Pass Statement

Pass statement is used when a statement is required syntactically but no command or code has to be executed. It specified a *null* operation or simply No Operation (NOP) statement. Nothing happens when the pass statement is executed.

Difference between comment and pass statements In Python programming, pass is a null statement. The difference between a comment and pass statement is that while the interpreter ignores a comment entirely, pass is not ignored. Comment is not executed but pass statement is executed but nothing happens.

Example:

```
for letter in "HELLO":
        pass     #The statement is doing nothing
        print("Pass : ", letter)
print("Done")
```

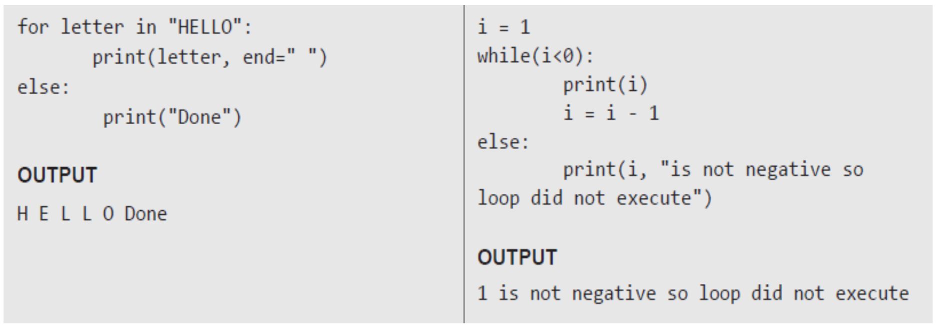
OUTPUT

```
Pass :  H
Pass :  E
Pass :  L
Pass :  L
Pass :  O
Done
```

The Else Statement Used With Loops

Unlike C and C++, in Python you can have the *else* statement associated with a loop statements. If the else statement is used with a *for* loop, the *else* statement is executed when the loop has completed iterating. But when used with the *while* loop, the *else* statement is executed when the condition becomes false.

Example:



```
for letter in "HELLO":
       print(letter, end=" ")
else:
        print("Done")
```

OUTPUT

```
H E L L O Done
```

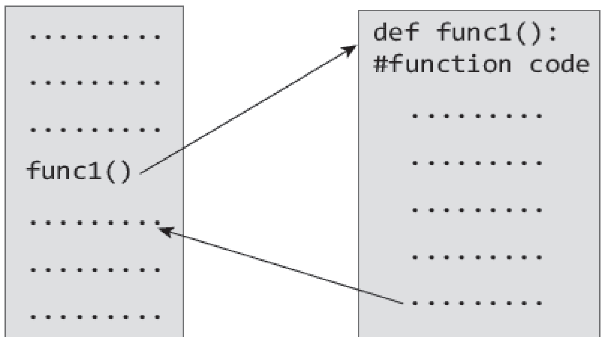
```
i = 1
while(i<0):
        print(i)
        i = i - 1
else:
        print(i, "is not negative so
loop did not execute")
```

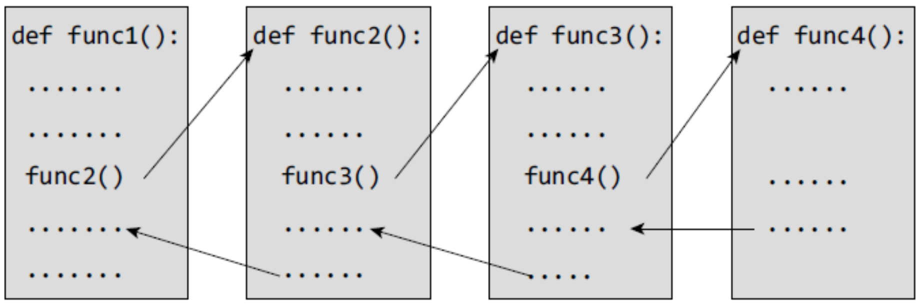
OUTPUT

```
1 is not negative so loop did not execute
```

Functions

Python enables its programmers to break up a program into segments commonly known as functions, each of which can be written more or less independently of the others. Every function in the program is supposed to perform a well-defined task.





A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

Need for Functions

Each function to be written and tested separately.

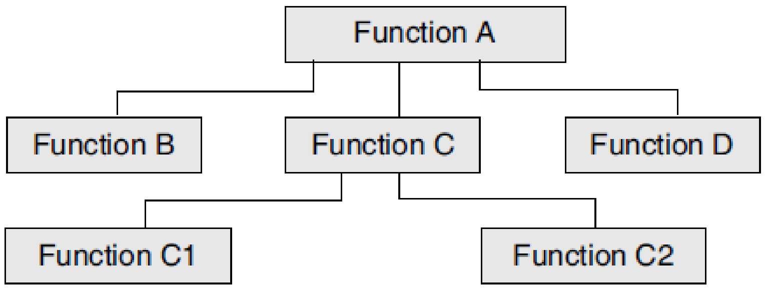
Understanding, coding and testing multiple separate functions is far easier.

Without the use of any function, then there will be countless lines in the code and maintaining it will be a big mess.

Programmers use functions without worrying about their code details. This speeds up program development, by allowing the programmer to concentrate only on the code that he has to write.

Different programmers working on that project can divide the workload by writing different functions.

Like Python libraries, programmers can also make their functions and use them from different **point in the main program or any other program that needs its functionalities.**



As you already know, Python gives you many built-in functions like print(), etc. but you can also create your own functions. These functions are called *user-defined functions.*

Function Declaration and Definition

A function, f that uses another function g, is known as the *calling function* and g is known as the *called function*.

The inputs that the function takes are known as *arguments/parameters*

When a called function returns some result back to the calling function, it is said to return that result.

The calling function may or may not pass parameters to the called function. If the called function accepts arguments, the calling function will pass parameters, else not.

Function declaration is a declaration statement that identifies a function with its name, a list of arguments that it accepts and the type of data it returns.

Function definition consists of a function header that identifies the function, followed by the body of the function containing the executable code for that function.

Defining a Function

You can define functions to provide the required functionality. Here are simple rules to define a function in Python.

- Function blocks begin with the keyword **def** followed by the function name and parentheses (( )).
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
- The first statement of a function can be an optional statement - the documentation string of the function or *docstring*.
- The code block within every function starts with a colon (:) and is indented.
- The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

Syntax:

```
def functionname( parameters ):
   "function_docstring"
   function_suite
   return [expression]
```

By default, parameters have a positional behavior and you need to inform them in the same order that they were defined.

Example 1

The following function takes a string as input parameter and prints it on standard screen.

```
def printme( str ):
   "This prints a passed string into this function"
   print str
   return
```

Example 2

```
def diff(x,y):          # function to subtract two numbers
        return x-y
a = 20
b = 10
operation = diff        # function name assigned to a variable
print(operation(a,b))   # function called using variable name
```

OUTPUT

```
10
```

Calling a Function

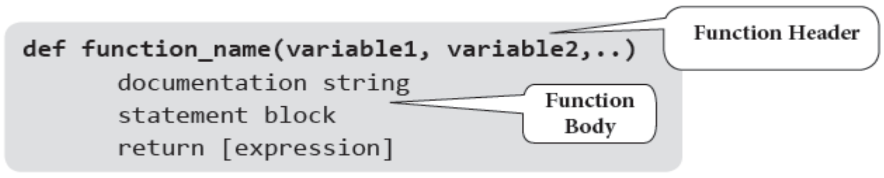
Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code. The function call statement invokes the function. When a function is invoked the program control jumps to the called function to execute the statements that are a part of that function. Once the called function is executed, the program control passes back to the calling function.

Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt. Following is the example to call printme() function.

Function Parameters

A function can take parameters which are nothing but some values that are passed to it so that the function can manipulate them to produce the desired result. These parameters are normal variables with a small difference that the values of these variables are defined (initialized) when we call the function and are then passed to the function.

Function name and the number and type of arguments in the function call must be same as that given in the function definition. If the data type of the argument passed does not matches with that expected in function then an error is generated.



```
def function_name(variable1, variable2,..)
      documentation string
      statement block
      return [expression]
```

```
# Function definition is here
def printme( str ):
   "This prints a passed string into this function"
   print str
   return;

# Now you can call printme function
printme("I'm first call to user defined function!")
printme("Again second call to the same function")
```

Examples:

```
def total(a,b):     # function accepting parameters
    result = a+b
    print("Sum of ", a, " and ", b, " = ", result)

a = int(input("Enter the first number : "))
b = int(input("Enter the second number : "))
total(a,b) #function call with two arguments
```

OUTPUT

```
Enter the first number : 10
Enter the second number : 20
Sum of 10 and 20 = 30
```

```
def func(i):
        print("Hello World", i)
func(5+2*3)
```

OUTPUT

```
Hello World 11
```

```
def func(i):              # function definition header accepts a variable with name i
print("Hello World", i)
j = 10
func(j)                   # Function is called using variable j
```

OUTPUT

```
Hello World 10
```

Pass by reference vs value

All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function. For example

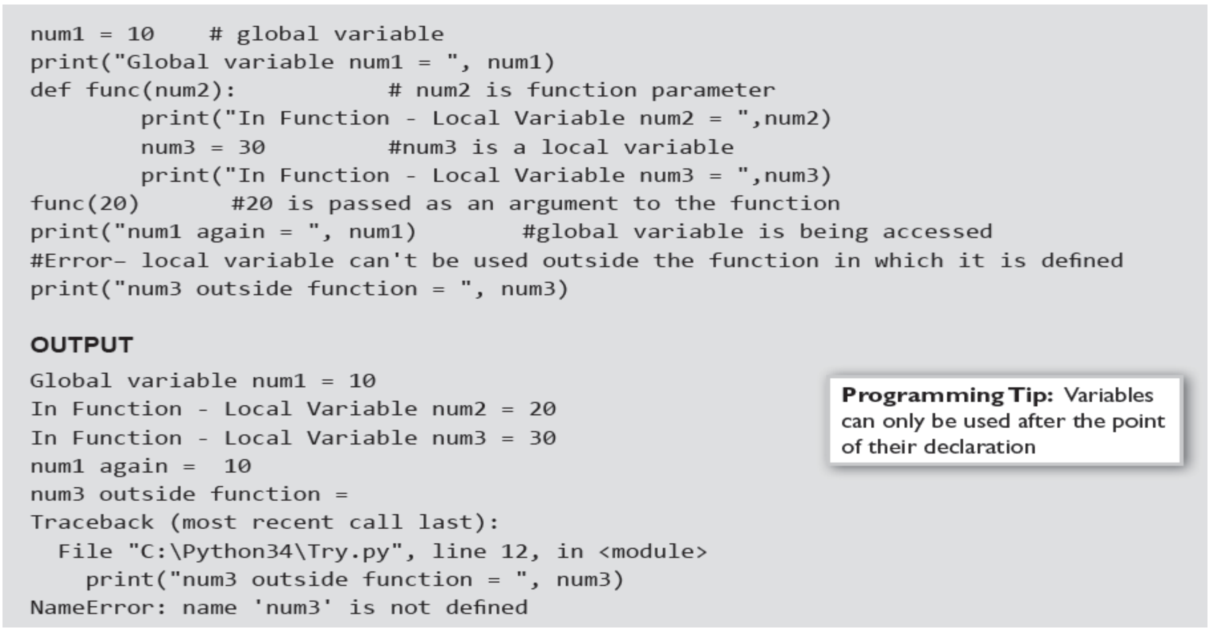
```
# Function definition is here
def changeme( mylist ):
   "This changes a passed list into this function"
   mylist.append([1,2,3,4]);
   print "Values inside the function: ", mylist
   return

# Now you can call changeme function
mylist = [10,20,30];
changeme( mylist );
print "Values outside the function: ", mylist
```

Local and Global Variables

A variable which is defined within a function is *local* to that function. A local variable can be accessed from the point of its definition until the end of the function in which it is defined. It exists as long as the function is executing. Function parameters behave like local variables in the function. Moreover, whenever we use the assignment operator (=) inside a function, a new local variable is created.

Global variables are those variables which are defined in the main body of the program file. They are visible throughout the program file. As a good programming habit, you must try to avoid the use of global variables because they may get altered by mistake and then result in erroneous output.

Example:



```
num1 = 10    # global variable
print("Global variable num1 = ", num1)
def func(num2):            # num2 is function parameter
        print("In Function - Local Variable num2 = ",num2)
        num3 = 30          #num3 is a local variable
        print("In Function - Local Variable num3 = ",num3)
func(20)       #20 is passed as an argument to the function
print("num1 again = ", num1)        #global variable is being accessed
#Error- local variable can't be used outside the function in which it is defined
print("num3 outside function = ", num3)
```

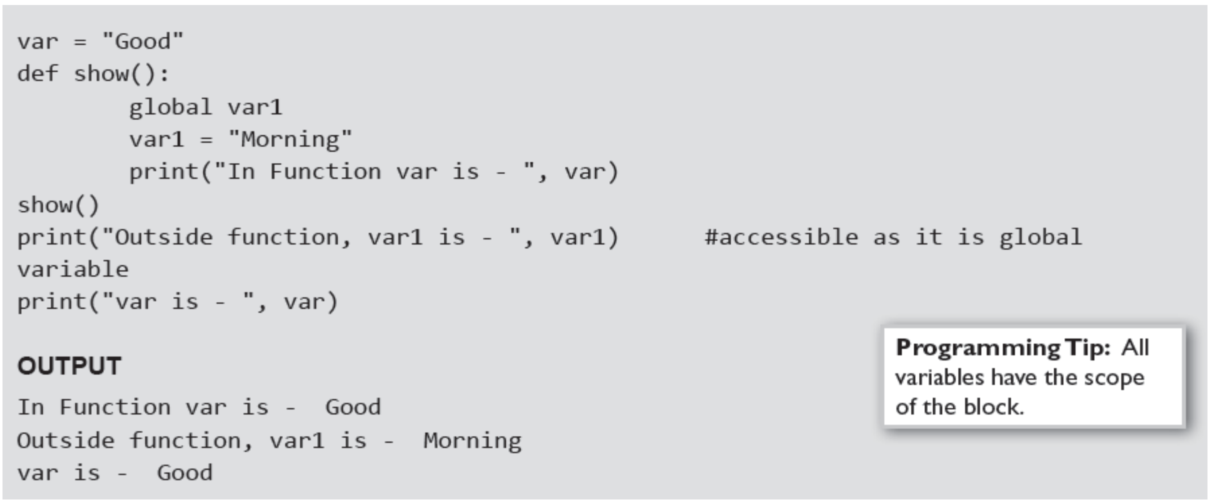
OUTPUT

```
Global variable num1 = 10
In Function - Local Variable num2 = 20
In Function - Local Variable num3 = 30
num1 again =  10
num3 outside function =
Traceback (most recent call last):
  File "C:\Python34\Try.py", line 12, in <module>
    print("num3 outside function = ", num3)
NameError: name 'num3' is not defined
```

Programming Tip: Variables can only be used after the point of their declaration

Using the Global Statement

To define a variable defined inside a function as global, you must use the global statement. This declares the local or the inner variable of the function to have module scope.



```
var = "Good"
def show():
        global var1
        var1 = "Morning"
        print("In Function var is - ", var)
show()
print("Outside function, var1 is - ", var1)      #accessible as it is global
variable
print("var is - ", var)
```

OUTPUT

```
In Function var is -  Good
Outside function, var1 is -  Morning
var is -  Good
```

Programming Tip: All variables have the scope of the block.

Resolution of names

Scope defines the visibility of a name within a block. If a local variable is defined in a block, its scope is that particular block. If it is defined in a function, then its scope is all blocks within that function.

When a variable name is used in a code block, it is resolved using the nearest enclosing scope. If no variable of that name is found, then a NameError is raised. In the code given below, str is a global string because it has been defined before calling the function.

The Return Statement

The syntax of return statement is,

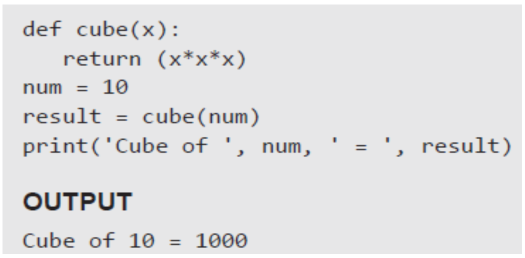
return [expression]

The expression is written in brackets because it is optional. If the expression is present, it is evaluated and the resultant value is returned to the calling function. However, if no expression is specified then the function will return None.

The return statement is used for two things.

• Return a value to the caller

• To end and exit a function and go back to its caller



```
def cube(x):
   return (x*x*x)
num = 10
result = cube(num)
print('Cube of ', num, ' = ', result)
```

OUTPUT

```
Cube of 10 = 1000
```

Function Arguments

You can call a function by using the following types of formal arguments

1. Required arguments

2. Keyword arguments
3. Default arguments
4. Variable-length arguments

Required arguments

Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition.

Example:

```
def display():
   print "Hello"
display("Hi")

OUTPUT
TypeError: display() takes
no arguments (1 given)
```

```
def display(str):
   print str
display()

OUTPUT
TypeError: display() takes
exactly 1 argument (0
given)
```

```
def display(str):
   print str
str ="Hello"
display(str)

OUTPUT
Hello
```

Keyword arguments

Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.

This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters.

Example:

```
def display(str, int_x, float_y):
    print("The string is : ",str)
    print("The integer value is : ", int_x)
    print("The floating point value is : ", float_y)
display(float_y = 56789.045, str = "Hello", int_x = 1234)

OUTPUT
The string is:  Hello
The integer value is:  1234
The floating point value is:  56789.045
```

Variable-length Arguments

In some situations, it is not known in advance how many arguments will be passed to a function. In such cases, Python allows programmers to make function calls with arbitrary (or any) number of arguments.

When we use arbitrary arguments or variable length arguments, then the function definition use an asterisk (*) before the parameter name. The syntax for a function using variable arguments can be given as,

```
def functionname([arg1, arg2,.... ] *var_args_tuple ):
    function statements
    return [expression]
```

Example:

```
def func(name, *fav_subjects):
    print("\n", name, " likes to read ")
    for subject in fav_subjects:
        print(subject)
func("Goransh", "Mathematics", "Android Programming")
func("Richa", "C", "Data Structures", "Design and Analysis of Algorithms")
func("Krish")
```

OUTPUT

```
Goransh  likes to read  Mathematics Android Programming
Richa likes to read  C Data Structures Design and Analysis of Algorithms
Krish  likes to read
```

Default Arguments

Python allows users to specify function arguments that can have default values. This means that a function can be called with fewer arguments than it is defined to have. That is, if the function accepts three parameters, but function call provides only two arguments, then the third parameter will be assigned the default (already specified) value.

The default value to an argument is provided by using the assignment operator (=). Users can specify a

default value for one or more arguments.

Example:

```
def display(name, course = "BTech"):
    print("Name : " + name)
    print("Course : ", course)
display(course = "BCA", name = "Arav") # Keyword Arguments
display(name = "Reyansh")              # Default Argument for course
```

OUTPUT

```
Name : Arav
Course :  BCA
Name : Reyansh
Course :  BTech
```

Lambda Functions Or Anonymous Functions

Lambda or anonymous functions are so called because they are not declared as other functions using the def keyword. Rather, they are created using the lambda keyword. Lambda functions are throw-away functions, i.e. they are just needed where they have been created and can be used anywhere a function is required. The lambda feature was added to Python due to the demand from LISP programmers.

Lambda functions contain only a single line. Its syntax can be given as,

```
lambda arguments: expression
```

Example:

```
sum = lambda x, y: x + y
print("Sum = ", sum(3, 5))

OUTPUT
Sum = 8
```

These functions are called anonymous because they are not declared in the standard manner by using the *def* keyword. You can use the *lambda* keyword to create small anonymous functions.

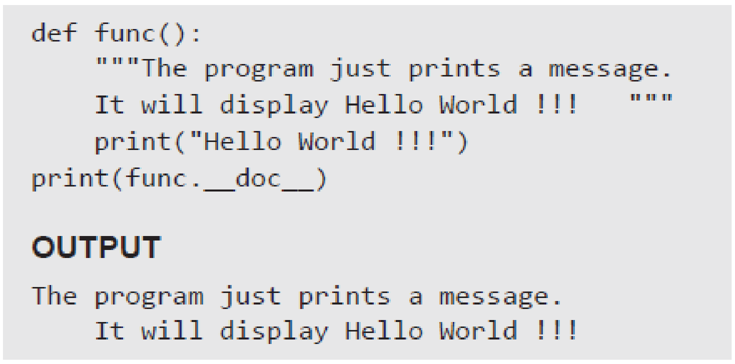
1. Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
2. An anonymous function cannot be a direct call to print because lambda requires an expression
3. Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.
4. Although it appears that lambda's are a one-line version of a function, they are not equivalent to inline statements in C or C++, whose purpose is by passing function stack allocation during invocation for performance reasons.

Documentation Strings

Docstrings (documentation strings) serve the same purpose as that of comments, as they are designed to explain code. However, they are more specific and have a proper syntax.

```
def functionname(parameters):
   "function_docstring"
   function statements
   return [expression]
```

Example:



```
def func():
    """The program just prints a message.
    It will display Hello World !!!    """
    print("Hello World !!!")
print(func.__doc__)
```

OUTPUT

```
The program just prints a message.
    It will display Hello World !!!
```

Recursive Functions

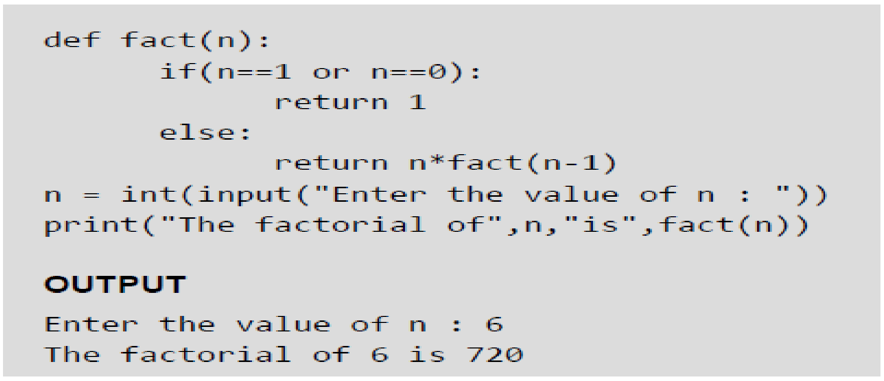
A recursive function is defined as a function that calls itself to solve a smaller version of its task until a final call is made which does not require a call to itself. Every recursive solution has two major cases, which are as follows:

base case, in which the problem is simple enough to be solved directly without making any further calls to the same function.

recursive case, in which first the problem at hand is divided into simpler sub parts.

Recursion utilized divide and conquer technique of problem solving.

Example:



```
def fact(n):
      if(n==1 or n==0):
            return 1
      else:
            return n*fact(n-1)
n = int(input("Enter the value of n : "))
print("The factorial of",n,"is",fact(n))
```

OUTPUT

```
Enter the value of n : 6
The factorial of 6 is 720
```

The from…import Statement

A module may contain definition for many variables and functions. When you import a module, you can use any variable or function defined in that module. But if you want to use only selected variables or functions, then you can use the from...import statement. For example, in the aforementioned program you are using only the path variable in the sys module, so you could have better written from sys import path.

Example:

```
from math import pi
print("PI = ", + pi)
```

OUTPUT

```
3.141592653589793
```

To import more than one item from a module, use a comma separated list. For example, to import the value of pi and sqrt() from the math module you can write,

```
from math import pi, sqrt
```

Modules and Namespaces

A namespace is a container that provides a named context for identifiers. Two identifiers with the same name in the same scope will lead to a name clash. In simple terms, Python does not allow programmers to have two different identifiers with the same name. However, in some situations we need to have same name identifiers. To cater to such situations, namespaces is the keyword. Namespaces enable programs to avoid potential name clashes by associating each identifier with the namespace from which it originates.

Example:

```
# module1
def repeat_x(x):
    return x*2

# module2
def repeat_x(x):
    return x**2

import module1
import module2
result = repeat_x(10)     # ambiguous reference for identifier repeat_x
```

Local, Global, and Built-in Namespaces

During a program's execution, there are three main namespaces that are referenced- the built-in namespace, the global namespace, and the local namespace. The built-in namespace, as the name suggests contains names of all the built-in functions, constants, etc that are already defined in Python. The global namespace contains identifiers of the currently executing module and the local namespace has identifiers defined in the currently executing function (if any).

When the Python interpreter sees an identifier, it first searches the local namespace, then the global namespace, and finally the built-in namespace. Therefore, if two identifiers with same name are defined in more than one of these namespaces, it becomes masked.

Example:

```
def max(numbers):      # global namespace
    print("USER DEFINED FUNCTION MAX.....")
    large = -1       # local namespace
    for i in numbers:
        if i>large:
            large = i
    return large
```

The globals() and locals() functions are used to return the names in the global and local namespaces (In Python, each function, module, class, package, etc owns a “namespace” in which variable names are identified and resolved). The result of these functions is of course, dependent on the location from where they are called. For example,

If locals() is called from within a function, names that can be accessed locally from that function will be returned.

If globals() is called from within a function, all the names that can be accessed globally from that function is returned.

Reload()- When a module is imported into a program, the code in the module is executed only once. If you want to re-execute the top-level code in a module, you must use the reload() function. This function again imports a module that was previously imported.

filter() Function

The filter() function constructs a list from those elements of the list for which a function returns True. The

syntax of the filter() function is given as, filter(function, sequence)

As per the syntax, the filter() function returns a sequence that contains items from the sequence for which the function is True. If sequence is a string, Unicode, or a tuple, then the result will be of the same type;

otherwise, it is always a list.

Example:

```
def check(x):
    if (x % 2 == 0 or x % 4 == 0):
        return 1
# call check() for every value between 2 to 21
evens = list(filter(check, range(2, 22)))
print(evens)
```

OUTPUT

```
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

Programming Tip: Do not add or remove elements from the list during iteration.

map() Function

The map() function applies a particular function to every element of a list. Its syntax is same as the filter function

```
map(function, sequence)
```

After applying the specified function on the sequence, the map() function returns the modified list. The map() function calls function(item) for each item in the sequence and returns a list of the return values.

Example:

```
def add_2(x):
    x += 2
    return x
num_list = [1,2,3,4,5,6,7]
print("Original List is : ", num_list)
new_list = list(map(add_2, num_list))
print("Modified List is : ", new_list)
```

OUTPUT

```
Original List is :  [1, 2, 3, 4, 5, 6, 7]
Modified List is :  [3, 4, 5, 6, 7, 8, 9]
```

reduce() Function

The reduce() function with syntax as given below returns a single value generated by calling the function

on the first two items of the sequence, then on the result and the next item, and so on.

```
reduce(function, sequence)
```

Example: Program to calculate the sum of values in a list using the reduce() function

```
import functools #functools is a module that contains the function reduce()
def add(x,y):
    return x+y
num_list = [1,2,3,4,5]
print("Sum of values in list = ")
print(functools.reduce(add, num_list))
```

OUTPUT

```
Sum of values in list =  15
```

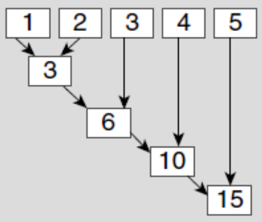


Figure 8.4 reduce() function

Module Private Variables

In Python, all identifiers defined in a module are public by default. This means that all identifiers are accessible by any other module that imports it. But, if you want some variables or functions in a module to be privately used within the module, but not to be accessed from outside it, then you need to declare those identifiers as private.

In Python identifiers whose name starts with two underscores (__) are known as private identifiers. These identifiers can be used only within the module. In no way, they can be accessed from outside the module. Therefore, when the module is imported using the import * form modulename, all the identifiers of a module's namespace is imported except the private ones (ones beginning with double underscores). Thus, private identifiers become inaccessible from within the importing module.

Packages in Python

A package is a hierarchical file directory structure that has modules and other packages within it. Like modules, you can very easily create packages in Python.

Every package in Python is a directory which must have a special file called __init__.py. This file may not even have a single line of code. It is simply added to indicate that this directory is not an ordinary directory and contains a Python package. In your programs, you can import a package in the same way as you import any module.

For example, to create a package called MyPackage, create a directory called MyPackage having the module MyModule and the __init__.py file. Now, to use MyModule in a program, you must first import it. This can be done in two ways.

import MyPackage.MyModule

or

from MyPackage import MyModule

Questions

1. When should we use 'nested if' statements? Illustrate your answer with an example. [CO2][BT1]
2. Write short note on conditional branching statements supported by python. [CO2][BT2]

3. Explain the syntax of for loop with an example. [CO1][BT2]
4. Is it necessary for every if block to be accompanied with an else block. Comment on this with an example.[CO2][BT4]
5. Explain utility of Range() with the help of an example. [CO2][BT5]
6. For loop is usually known as a definite or determine loop. Justify the statement with the help of an example. [CO2][BT4]
7. Explain the utility of break statement with the help of an example. [CO1][BT2]
8. Differentiate between counter controlled loops and sequential controlled loops. [CO2][BT4]
9. Write short note on nested loops. [CO2][BT2]
10. Explain the utility of continue statement with the help of an example. [CO1][BT2]
11. Differentiate between pass and continue. [CO1][BT4]
12. Write short note on elif statement with an example. [CO1][BT2]
13. Write short note on iterative statements. [CO1][BT2]
14. Define function with a an example and give its advantages. [CO2][BT1]
15. What are user defined functions ? Give an example. [CO2][BT1]
16. What do you understand by the term arguments? How do you pass them to a function. [CO2][BT2]
17. Can a function call another function ? justify your answer with an example. [CO1][BT5]
18. ‘The return statement is optional’. Justify this statement with an example. [CO1][BT5]
19. Differentiate local and global variables. [CO1][BT1]
20. Define a function that calculates the sum of all numbers from 0 to its argument. [CO2][BT1]
21. When you can have a variable with the same name as that of a global variable in the program, how is the name resolved in python? [CO2][BT3]
22. Explain the use of return statement. [CO1][BT2]
23. What are the modules. How do you use them in your programs ? [CO2][BT3]
24. What are the packages in python ? [CO2][BT2]
25. Write short notes on [CO2][BT2]
    a) Keyword arguments
    b) Default arguments
    c) Lambda functions

Unit-III

Lecture-No	Topic	Learning Objective
01	Introduction to Classes and Objects	Able to understand the basics of Class and Object
02	init method, Class variables, and Object variables	Able to manipulate the variables with proper knowledge
03	Public and Private Data members	Able to understand the usage of data members
04	calling methods from other methods	Get to know the working of methods in Class
05	built-in class attributes, garbage collection	Able to understand the extra features that are inbuilt in Python related to Class
06	class methods	Able to understand the methods related to a Class
07	static methods	Able to understand the purpose of static methods in Class.

➔ **Classes and objects**

Class: class is a plan or blue print to create objects.

Object: physical existence in memory

Class contains properties(attributes) and actions(methods)

Properties represented as variables

Actions as methods.

Classes are implemented in python my using **class** keyword.

Syntax for class:

```
class classname:
            variable#instance variables, static and local variables
            methods # instance, static and class methods
```

➔ **Object creation**

Reference variable =classname()

Members of a class accessed through objects

- objectname.membername

Ex.

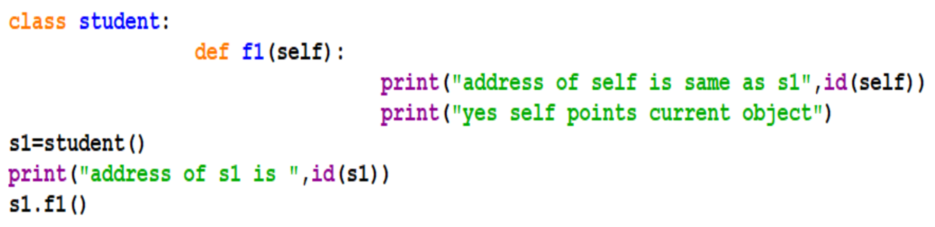
class student:

```
        def info(self):
                self.name=”Isha”
                self.rollno=11
                self.marks=94
                print(self.name,self.rollno,self.marks)
s=student()
s.info()
```

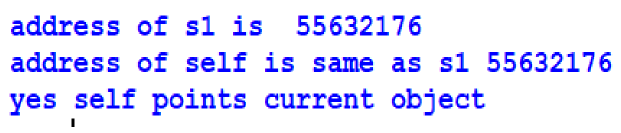
➔ **self variable:**

the reference variable pointing to current object is called self varible.

- Self is used to access instance variables and instance methods.
- Self should be first parameter for instance method and constructor.
- Example



```
class student:
        def f1(self):
                print("address of self is same as s1",id(self))
                print("yes self points current object")
s1=student()
print("address of s1 is ",id(s1))
s1.f1()
```



```
address of s1 is  55632176
address of self is same as s1 55632176
yes self points current object
```

➔ **Constructor**

- Constructor is a special method used to initialize instance variable when the objects are created.
- Constructor defined with __init__(two underscores)
- Constructor is called when the objects are created.
- Constructor takes first argument as self.

Ex.

```
class student:
        def __init__(self,name,rollno,marks):
                self.name=name
                self.rollno=rollno
                self.marks=marks
        def display(self):
                print(“name is {0} rollno is {1} marks is
{2}”.format(self.name,self.rollno,self.marks))
        s=student(“Isha”,10,90)
        s.display()
        s1.student(“XYZ”,11,95)
```

```
s1.display()
O/P: name is Isha rollno is 10 marks is 90
----same---
```

method	constructor
Method name can be any valid identifier	name must be __init__(self)
method has business logic	Initialize variables
invoked through object name and method name	Invoked automatically when objects are created

➔ **Types of variables**

1. Instance variables(object level variables)
2. Static variables(class level variables)
3. Local variables(method level variables)

I. Instance variables

- Instance variable: value of the variable changed from object to object.
- A separate copy of the variable is created for each object.

Where instance variables are declared?

1. Inside constructor using self
2. Inside instance method using self
3. Out side class using reference variable.

a. Inside constructor using self:

➢ Instance variables are defined inside a constructor using self key word.
➢ Instance variables are initialized when the objects are created.
➢ Instance variables are added to object when objects are created

Ex.

```
Class student:
        def __init__(self):
                self.name="Isha"
                self.rollno=90
s=student()
print(s.__dict__)
```

O/P: {'name': 'Isha Padhy','rollno':90}

b. Inside instance method using self

➢ Instance variables defined inside instance method using self keyword.

- Instance variables added to object once the method is called.

```
Class student:
        def __init__(self):
                self.name="Isha Padhy"
                self.rollno=90
        def addmarks(self):
                self.marks=100
s=student()
print(s.__dict__)
s.addmarks()
print(s.__dict__)
```

c. Outside class using object reference variable

- Instance variables added to a particular object

Ex.

```
class student:
        def __init__(self):
                self.name="Isha"
                self.rollno=20
        def addmarks(self):
                self.marks=100
s=student()
print(s.__dict__)
s.addmarks()
print(s.__dict__)
s.section="CSE"
print(s.__dict__)
```

Deletion of instance variables

- Delete inside class

  del self.variable_name

- Delete outside class

  del Objectreferencevariable.variablename

- Instance variables which are deleted from one object will not effect to another object.

If we change the values of instance variables of one object then those changes won't be reflected to the remaining objects, because for every object we are separate copy of instance variables are available.

Ex.

```
Class Test:
        def __init__(self):
                self.a=10
                self.b=15
t1=Test()
t2=Test()
t1.a=100
t2.b=200
print("t1:",t1.a,t1.b)
print("t2:",t2.a,t2.b)
```

II. Static variables

Static variables are also called class variables.

Variables which are defined in a class but outside of a methods are called static variables.

Static variable a copy will be created that copy is shared by all the objects created for that class.

Static variables are accessed by either class name or object name.

When the static variables are modified using self or object name those changes will not affect other objects

To affect changes to all the objects, access static variables using class name.

Accessing static variables

a. Inside constructor: by using either self or class name
b. Inside instance method: by using either self or class name
c. Inside class method: by using either class variable or class name
d. Inside static method: by using class name
e. From outside of class: by using either object reference or class name

Ex.

```
class Test:
        count=0
        def __init__(self):
                Test.count=Test.count+1
                print("no of objects created are",self.count)
t1=Test()
t2=Test()
t3=Test()
```

III. **Local variables**

Local variables: variables which are declared inside of a method.

Local variables scope and life time will be within the method

Outside of a method local variables cannot be accessed.

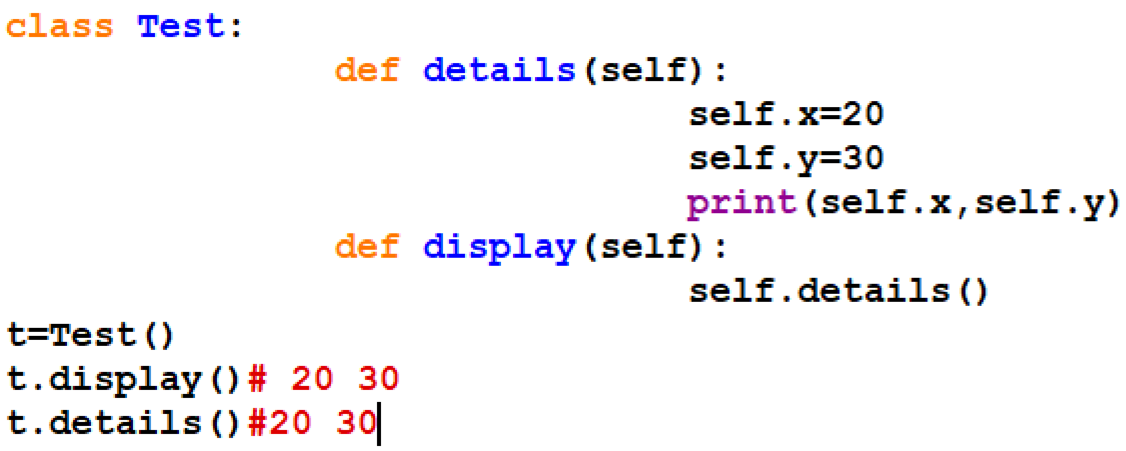
➔ **Types of methods**

1. Instance method
2. Class method
3. Static method

1. Instance method: if method takes self as first argument is called instance method and uses the instance variables.

- def functionname(self):

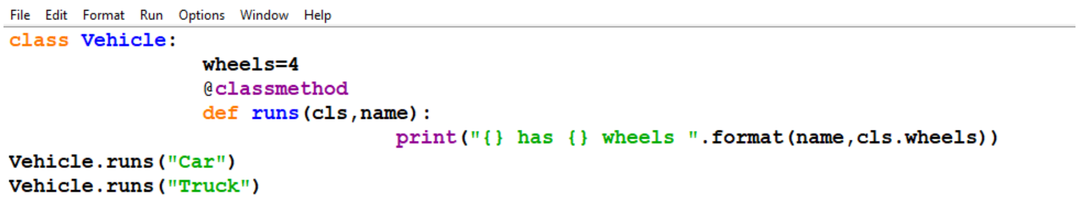
- Instance methods are called with self inside a class and outside the class instance methods are called with object reference variable.



```
class Test:
                def details(self):
                                self.x=20
                                self.y=30
                                print(self.x,self.y)
                def display(self):
                                self.details()
t=Test()
t.display()# 20 30
t.details()#20 30
```

2. Class methods

- Class methods: if a method uses only class variables such methods are declared as class methods.
- Class methods are declared as explicitly using @classmethod decorator.
- For class methods keyword cls is to be used as first argument rather than self.
- Class methods to be called by using class name or object reference variable.

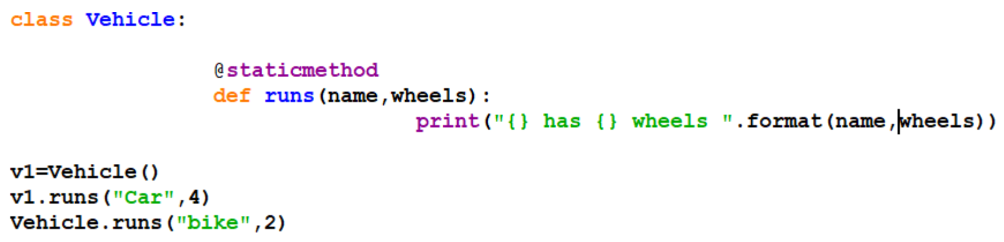


```
class Vehicle:
                wheels=4
                @classmethod
                def runs(cls,name):
                                print("{} has {} wheels ".format(name,cls.wheels))
Vehicle.runs("Car")
Vehicle.runs("Truck")
```

3. Static method

- Static method: the method does not use any instance or class variables is called static method
- Static methods does not use any argument of type such as self and cls.
- Static methods explicitly defined as @staticmethod.

- Static methods are defined like a normal function.
- Static methods called with class name or reference variable.



```
class Vehicle:

            @staticmethod
            def runs(name,wheels):
                        print("{} has {} wheels ".format(name,wheels))

v1=Vehicle()
v1.runs("Car",4)
Vehicle.runs("bike",2)
```

We can access members of one class from other class

Ex.

```
class Employee:
            def __init__(self,eno,ename,esal):
                        self.eno=eno
                        self.ename=ename
                        self.esal=esal
            def display(self):
                        print("Employee Number:",self.eno)
                        print("Employee Name:",self.ename)
                        print("Employee Salary:",self.esal)
class Test:
            def change(emp):
                        emp.esal=emp.esal+1000
                        emp.display()
e=Employee(100,"Isha",1000)
Test.change(e)
```

Create a list of objects

```
#from collections import Counter
class Student:
   def __init__(self,var):
      self.var=var
object=[]
for i in range(10):
   object.append(Student(i))
#print(Counter(object))
for obj in object:
    print(obj.var)
```

When object variable and class variable has same name then, object variable has more preference.

```
class Abc:
    even=0
    def __init__(self,num):
        if num%2==0:
            self.even=1
    def pr(self):
        if self.even==1:
            print("even")
        else:
            print("odd")
obj1=Abc(4)
obj1.pr()
obj2=Abc(3)
obj2.pr()
```

```
>>>even
odd
```

Built-in Function

Functions are used with the attribute of a class

1. **getattr(obj,name[,default]):**

The above syntax is equivalent to:

object.name

obj - object whose named attribute's value is to be returned

name - string that contains the attribute's name

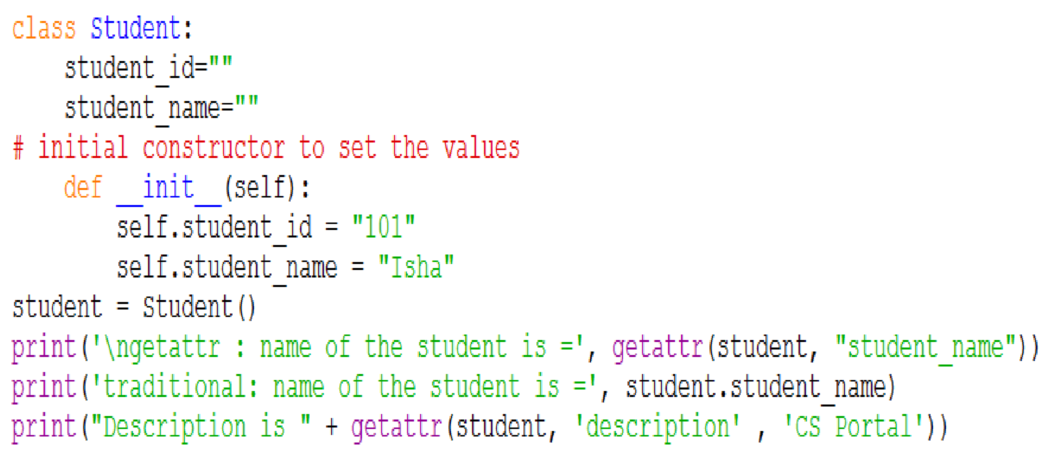
default (Optional) - value that is returned when the named attribute is not found

The getattr() method returns:

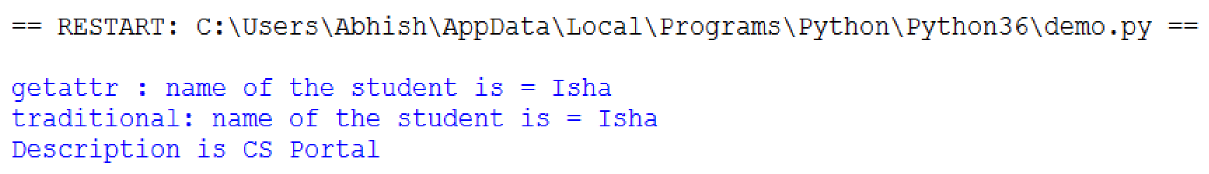
- value of the named attribute of the given object

- default, if no named attribute is found

- AttributeError exception, if named attribute is not found and default is not defined



```
class Student:
    student_id=""
    student_name=""
# initial constructor to set the values
    def __init__(self):
        self.student_id = "101"
        self.student_name = "Isha"
student = Student()
print('\ngetattr : name of the student is =', getattr(student, "student_name"))
print('traditional: name of the student is =', student.student_name)
print("Description is " + getattr(student, 'description' , 'CS Portal'))
```



```
== RESTART: C:\Users\Abhish\AppData\Local\Programs\Python\Python36\demo.py ==

getattr : name of the student is = Isha
traditional: name of the student is = Isha
Description is CS Portal
```

2. **hasattr(obj,name):**

object - object whose named attribute is to be checked

name - name of the attribute to be searched

The hasattr() method returns:

True, if object has the given named attribute

False, if object has no given named attribute

```
class Employee:
    id = 0
    name = ''
    def __init__(self, i, n):
        self.id = i
        self.name = n
d = Employee(10, 'isha')
if hasattr(d, 'name'):
    print(getattr(d, 'name'))
```

O/P: isha

Note:

print(hasattr(student,"description"))

O/P: False

```
class Employee:
    id = 0
    name = ''
    def __init__(self, i, n):
        self.id = i
        self.name = n
d = Employee(10, 'isha')
if hasattr(d, 'name'):
    print(getattr(d, 'name'))
```

O/P: isha

3**. setattr(*object, attribute, value*)**

The setattr() function sets the value of the specified attribute of the specified object.

obj : Object whose which attribute is to be assigned.

var : object attribute which has to be assigned.

val : value with which variable is to be assigned.

```
class Demo:
    name = 'Hello'
obj = Demo()
print("Before setattr name : ", obj.name)
setattr(obj, 'name', 'HelloWorld')
print("After setattr name : ", obj.name)
```

O/p:

```
Before setattr name :  Hello
After setattr name :  HelloWorld
```

```
class Data:
    pass
d = Data()
attr_name = input('Enter the attribute name:\n')
attr_value = input('Enter the attribute value:\n')
setattr(d, attr_name, attr_value)
print('Data attribute =', attr_name, 'and its value =', getattr(d, attr_name))
```

```
Enter the attribute name:
hj
Enter the attribute value:
45
Data attribute = hj and its value = 45
```

4. **The delattr() deletes an attribute from the object (if the object allows it).**

delattr(object, name)

object - the object from which name attribute is to be removed

name - a string which must be the name of the attribute to be removed from the object

```
>>> class Demo :
...     def __init__(self, x=0):
```

```
...        self.x = x
...
>>> f = Demo(10)
>>> f.x
10
>>> delattr(f, 'x')
>>> f.x
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
AttributeError: Demo instance has no attribute 'x'
```

Calling methods from other methods

```
class Student:
    def __init__(self,var):
        self.var=var
    def display(self):
        print(self.var + "\n" + "yes I am")
    def adding(self):
        self.var='Am I not '+self.var+"?"
        self.display()
obj= Student("genious")
obj.adding()
```

```
== RESTART: C:\Users\Abhish\AppData\Local\Programs\Python\Python36\demo.py ==
Am I not genious?
yes I am
>>>
```

```
class Student:
    def __init__(self,func):
        self.var="hello"
        Student.show=func
class Faculty:
    def __init__(self,met):
        self.var="hi"
        self.show=met
def method():
    print("CBIT")
obj= Student(method)
Student.show()
```

```
#obj.show()
obj1= Faculty(method)
obj1.show()
```

== RESTART: C:\Users\Abhish\AppData\Local\Programs\Python\Python36\demo.py ==

CBIT

CBIT

Python object functions

-> __str__()

- returns the string representation of the object.
- Method is called when print() or str() function is invoked on an object.
- This method must return the String object.
- If we don't implement __str__() function for a class, then built-in object implementation is used that actually calls __repr__() function.

-> __repr__()

- returns the object representation. It could be any valid python expression such as tuple, dictionary, string etc.
- This method is called when repr() function is invoked on the object, in that case, __repr__() function must return a String

```
class Student:
def __init__(self, name, roll):
        self.name = name
        self.roll=roll
p = Student('isha', 34)
 print(p.__str__())
 print(p.__repr__())
__main__.Person object at 0x10ff22470> <__main__.Person object at 0x10ff22470>
class Student:
  def __init__(self,name,roll):
     self.name=name
     self.roll=roll
  def __repr__(self):
     return {'name':self.name,'roll':self.roll}
  def __str__(self):
     return 'name is'+self.name+'and roll is'+str(self.roll)
s=Student('isha',23)
print(s.__str__())                    #name is isha and roll is 23
print(s.__repr__())
print(repr(s))     #error as repr is returning a dictinary instead of string
Print(str(s))      #if __str__() definition is not there,
                                    Traceback (most recent call last):
 File C:\Users\Abhish\AppData\Local\Programs\Python\Python36\demo.py", line 13, in <module>
    TypeError: __str__ returned non-string (type dict)
```

- #change repr returning a string
- **__cmp__()**

-> cmp() is an in-built function in Python, it is used to compare two objects and it returns negative, zero or positive value based on the given input i.e. will return **negative (-1)** if obj1<obj2, **zero (0)** if obj1=obj2, **positive (1)** if obj1>obj2.

```
-> def __cmp__(self,other):
            if( __lt__(self.value,other.value))
                        return -1
            elif self.value > other.value:
                        return 1
             else:
                        return 0
```

- **__len__()**

-> len(obj) is a builtin function which returns the length of an object. The *obj* argument can be either a sequence (lists, tuple or string) or a mapping (dictionary).

-> len() is the public interface we use to get the length of an object. The __len__ method is the implementation that an object that supports the concept of length is expected to implement.

- **__call__()**
- **__del__()**
- Class Point:

```
            def __init__(self, x, y):
                        self.x = x
                        self.y = y
```

- Python will complain about Point(0, 1) < Point(1, 0)
- since it doesn’t know how to compare.
- TypeError: '<' not supported between instances of 'Point' and 'Point'

➔ **Garbage Collection**

- Objects used in programs occupy some space in memory so after its use the space need to be freed.
- Not freeing memory leads to:-

-> **Memory leaks** : part of memory no longer in use is not released.

Object is stored in memory but cannot be accessed by running code

-> **Dangling pointer**

- Automatic memory management:

1. Reference Counting: keeping track of all the references to an object.

→ Disad: needs extra memory to keep the references.

→ It gets created when object is created. Count gets incremented whenever the object is referenced and gets decremented when dereferenced. When count becomes 0 memory is deallocated.

→ **sys.getrefcount(obj)**

```
stud_roll = [1, 2 ,3, 4, 5, 6, 7, 8, 9]           #Reference count of some_list = 1
other_list = stud_roll                           #Reference count = 2
 list_total = sum(stud_roll)                     #if we pass the object as an assignment
 list_of_list = [stud_roll, stud_roll, stud_roll] # If we put the object in a list, reference count will also increase
 #Let's check the reference count of object "stud_roll"
>>> import sys
>>> sys.getrefcount(stud_roll)
```

O/P: 6

-> **Generalized garbage collection:**

→ Garbage collector keeps track of all objects in memory.

→ 3 generations are maintained once the garbage collection process starts.

→ Concept of threshold is used: the garbage collector module has a threshold number of objects.If

→ number of objects increases than threshold then collection process starts

→ Here the GC(garbage collector) module is used

→ methods to be used:

1. **gc.get_threshold():** check the configured thresholds of the garbage collector
2. **gc.get_count():** number of objects in each generation

```
import gc
class Student:
    def __init__(self,name,roll):
        self.name=name
        self.roll=roll
s=Student('isha',23)
print(gc.get_threshold())
print(gc.get_count()
```

3. Python creates a number of objects by default before even start executing the program. We can trigger a manual garbage collection process by using the **gc.collect()** method:

4. **gc.get_objects()** : to see all the objects

5. **gc.garbage()**: A list of objects which the collector found to be unreachable but could not be freed (uncollectable objects).

6. **gc.set_threshold(*threshold0*[, *threshold1*[, *threshold2*]]):** Set the garbage collection thresholds (the collection frequency). Setting *threshold0* to zero disables collection.

Questions

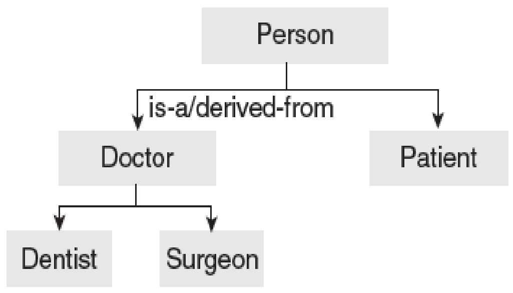
1. Explain about classes and objects in Python programming with an example. [BT1][CO1]
2. What are the different variable defined in class? Explain about purpose of each type. [BT3][CO1][CO3]
3. How instance variables differ from class variables? [BT2][CO1][CO3]
4. Explain about the access modifiers that are supported by the Python. [BT5][CO1][CO3]
5. What is a constructor? How do you define constructor in Python? What is the purpose of using constructor? [BT4][CO3]
6. Compare constructor and method. [BT5][CO3]
7. What is a self variable? [BT2][CO3]
8. Discuss the different python object related methods with examples. [BT6][CO3]
9. What is garbage collection? [BT1][CO1][CO3]
10. Elaborate the different attribute related methods. [BT2][CO3]

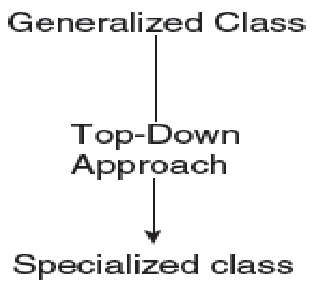
UNIT-IV

Lecture-No	Topic	Learning Objective
01	Introduction to Inheritance	Able to get insight into inheritance
02	Polymorphism and method overloading	Able to understand the way to achieve polymorphism
06	Composition or Containership	Able to understand the concept of composition
07	Abstract classes and inheritance	Able to understand the abstract classes and their implementation
08	Introduction, Implementation of Operator Overloading, Overriding	Able to understand how operators can behave differently in different situation
09	File types, opening and closing files, reading and writing files	Able to understand files and their working
10	file positions, Regular Expressions.	Able to use REs in real scenarios

Inheritance

The technique of creating a new class from an existing class is called *inheritance*. The old or existing class is called the *base class* and the new class is known as the *derived class* or *sub-class*. The derived classes are created by first inheriting the data and methods of the base class and then adding new specialized data and functions in it. In this process of inheritance, the base class remains unchanged. The concept of inheritance is used to implement the is-a relationship. For example, teacher IS-A person, student IS-A person; while both teacher and student are a person in the first place, both also have some distinguishing features. So all the common traits of teacher and student are specified in the Person class and specialized features are incorporate in two separate classes- Teacher and Student. Inheritance which follows a top down approach to problem solving. In top-down approach, generalized classes are designed first and then specialized classes are derived by inheriting/extending the generalized classes.





Inheriting Classes in Python

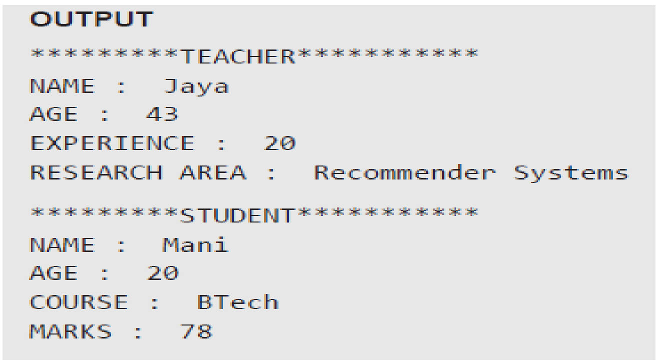
The syntax to inherit a class can be given as,

class DerivedClass(BaseClass): body_of_derived_class

Example:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def display(self):
        print("NAME : ", self.name)
        print("AGE : ", self.age)
class Teacher(Person):
    def __init__(self, name, age, exp, r_area):
        Person.__init__(self, name, age)
        self.exp = exp
        self.r_area = r_area
    def displayData(self):
        Person.display(self)
        print("EXPERIENCE : ", self.exp)
        print("RESEARCH AREA : ", self.r_area)
class Student(Person):
    def __init__(self, name, age, course, marks):
        Person.__init__(self, name, age)
        self.course = course
        self.marks = marks
    def displayData(self):
        Person.display(self)
        print("COURSE : ", self.course)
        print("MARKS : ", self.marks)
```

```
print("*********TEACHER***********")
T = Teacher("Jaya", 43, 20, "Recommender Systems")
T.displayData()
print("*********STUDENT***********")
S = Student("Mani", 20, "BTech", 78)
S.displayData()
```

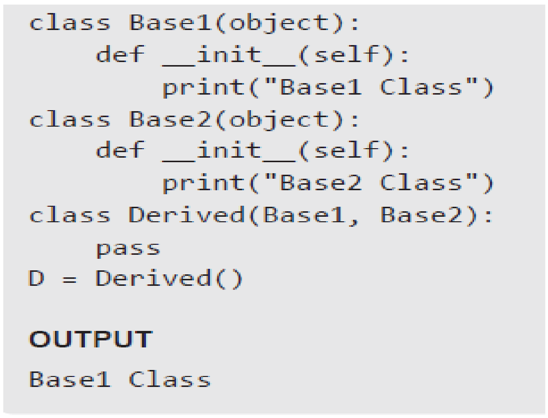


```
OUTPUT
*********TEACHER***********
NAME :  Jaya
AGE :  43
EXPERIENCE :  20
RESEARCH AREA :  Recommender Systems
*********STUDENT***********
NAME :  Mani
AGE :  20
COURSE :  BTech
MARKS :  78
```

Polymorphism and Method Overriding

Polymorphism refers to having several different forms. It is one of the key features of OOP. It enables the programmers to assign a different meaning or usage to a variable, function, or an object in different contexts. While inheritance is related to classes and their hierarchy, polymorphism, on the other hand, is related to methods. When polymorphism is applied to a function or method depending on the given parameters, a particular form of the function can be selected for execution. In Python, method overriding is one way of implementing polymorphism.

Example:



```
class Base1(object):
    def __init__(self):
        print("Base1 Class")
class Base2(object):
    def __init__(self):
        print("Base2 Class")
class Derived(Base1, Base2):
    pass
D = Derived()

OUTPUT
Base1 Class
```

Types of Inheritence

Multiple Inheritance

When a derived class inherits features from more than one base class, it is called *multiple inheritance*. The derived class has all the features of both the base classes and in addition to them can have additional new features. The syntax for multiple inheritance is similar to that of single inheritance and can be given as:

class Base1:

  statement block

class Base2:

  statement block

class Derived (Base1, Base2):

statement block

Example:

```
class Base1(object):          # First Base Class
  def __init__(self):
    super(Base1, self).__init__()
    print("Base1 Class")
class Base2(object):          # Second Base Class
  def __init__(self):
    super(Base2, self).__init__()
    print("Base2 Class")
class Derived(Base1, Base2):    # Derived Class derived from Base1 and Base2
  def __init__(self):
    super(Derived, self).__init__()
    print("Derived Class")
D = Derived()
```

OUTPUT

```
Base2 Class
Base1 Class
Derived Class
```

Multi-Level Inheritance

The technique of deriving a class from an already derived class is called *multi - level inheritance.*

The syntax for multi-level inheritance can be given as,

class Base:

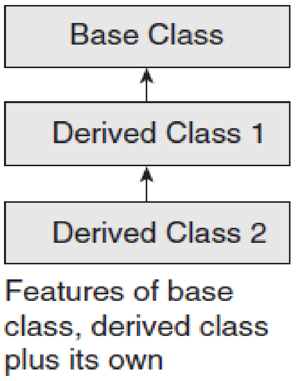
pass

class Derived1(Base):

pass

class Derived2(Derived1):

Pass



Example:

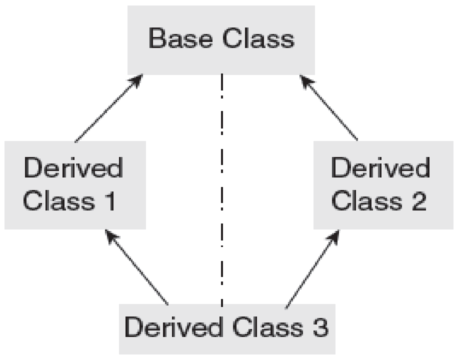
```
class Person:                # Base class
    def name(self):
      print('Name...')
class Teacher(Person):       # Class derived from Person
   def Qualification(self):
      print('Qualification...Ph.D must')
class HOD(Teacher):    # Class derived from Teacher, now hierarchy is Person-
>Teacher->HOD
    def experience(self):
        print('Experience......at least 15 years')
hod = HOD()
hod.name()
hod.Qualification()
hod.experience()
```

OUTPUT

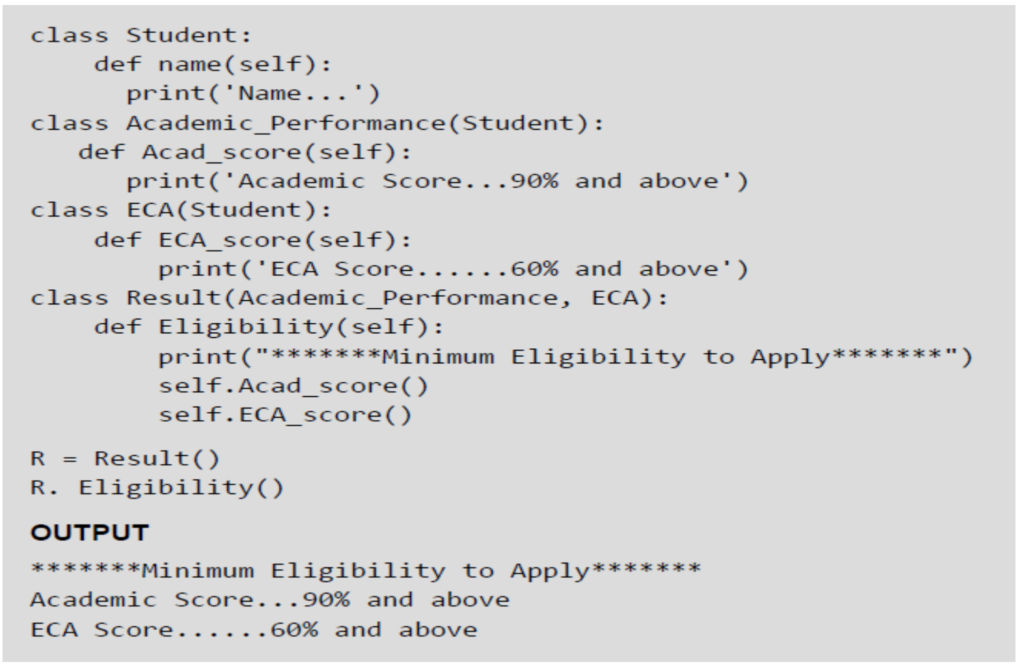
```
Name...
Qualification...Ph.D must
Experience......at least 15 years
```

Multi-path Inheritance

Deriving a class from other derived classes that are in turn derived from the same base class is called *multi-path inheritance*.



Example:



```
class Student:
    def name(self):
      print('Name...')
class Academic_Performance(Student):
   def Acad_score(self):
      print('Academic Score...90% and above')
class ECA(Student):
    def ECA_score(self):
        print('ECA Score......60% and above')
class Result(Academic_Performance, ECA):
    def Eligibility(self):
        print("*******Minimum Eligibility to Apply*******")
        self.Acad_score()
        self.ECA_score()

R = Result()
R. Eligibility()
```

OUTPUT

```
*******Minimum Eligibility to Apply*******
Academic Score...90% and above
ECA Score......60% and above
```

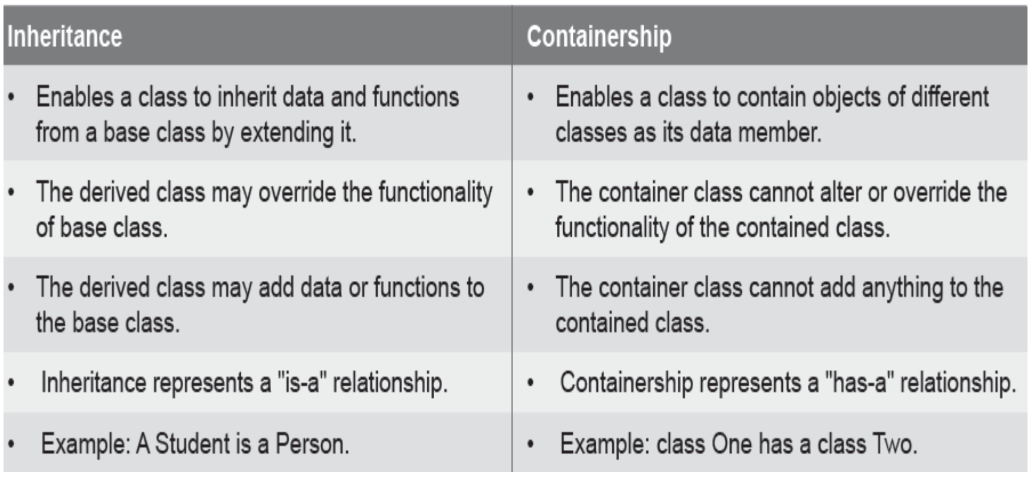
Composition or Containership or Complex Objects

Complex objects are objects that are built from smaller or simpler objects. For example, a car is built using a metal frame, an engine, some tyres, a transmission, a steering wheel, and several other parts. Similarly, a computer system is made up of several parts such as CPU, motherboard, memory, and so on. This process of building complex objects from simpler ones is called *composition or containership.*

In object-oriented programming languages, object composition is used for objects that have a has-a relationship to each other. For example, a car has-a metal frame, has-an engine, etc. A personal computer has-a CPU, a motherboard, and other components.

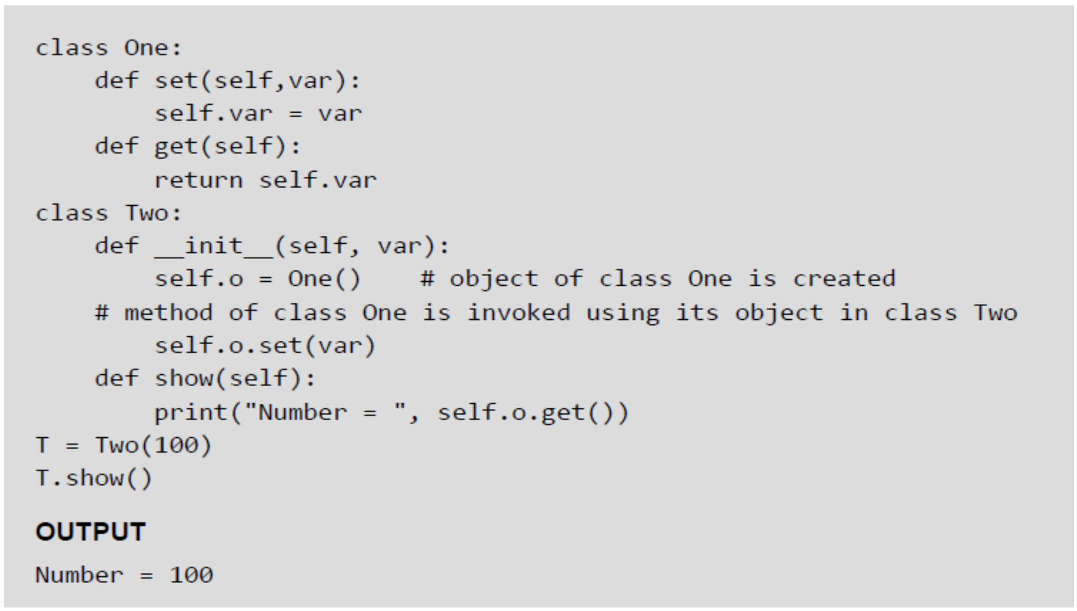
Until now, we have been using classes that have data members of built-in type. While this worked well for simple classes, for designing classes that simulate real world applications, programmers often need data members that belong to other simpler classes.

Inheritance vs Composition



Inheritance	Containership
• Enables a class to inherit data and functions from a base class by extending it.	• Enables a class to contain objects of different classes as its data member.
• The derived class may override the functionality of base class.	• The container class cannot alter or override the functionality of the contained class.
• The derived class may add data or functions to the base class.	• The container class cannot add anything to the contained class.
• Inheritance represents a "is-a" relationship.	• Containership represents a "has-a" relationship.
• Example: A Student is a Person.	• Example: class One has a class Two.

Example:



```
class One:
    def set(self,var):
        self.var = var
    def get(self):
        return self.var
class Two:
    def __init__(self, var):
        self.o = One()    # object of class One is created
    # method of class One is invoked using its object in class Two
        self.o.set(var)
    def show(self):
        print("Number = ", self.o.get())
T = Two(100)
T.show()
```

OUTPUT

```
Number = 100
```

Abstract Classes and Interfaces

It is possible to create a class which cannot be instantiated. This means that that you cannot create objects of that class. Such classes could only be inherited and then an object of the derived class was used to access the features of the base class. Such a class was known as the abstract class.

An abstract class corresponds to an abstract concept. For example, a polygon may refer to a rectangle, triangle or any other closed figure. Therefore, an abstract class is a class that is specifically defined to lay a foundation for other classes that exhibits a common behavior or similar characteristics. It is primarily used only as a base class for inheritance.

Since an abstract class, is an incomplete class, users are not allowed to create its object. To use such a class, programmers must derive it and override the features specified in that class.

An abstract class just serves as a *template* for other classes by defining a list of methods that the classes must implement. In Python, we use the NotImplementedError to restrict the instantiation of a class. Any class that has the NotImplementedError inside method definitions cannot be instantiated.

Example:

```
class Fruit:
    def taste(self):
        raise NotImplementedError()
    def rich_in(self):
        raise NotImplementedError()
    def colour(self):
        raise NotImplementedError()
class Mango(Fruit):
    def taste(self):
        return "Sweet"
    def rich_in(self):
        return "Vitamin A"
    def colour(self):
        return "Yellow"
class Orange(Fruit):
    def taste(self):
        return "Sour"
    def rich_in(self):
        return "Vitamin C"
    def colour(self):
        return "Orange"
Alphanso = Mango()
print(Alphanso.taste(), Alphanso.rich_in(), Alphanso.colour())
Org = Orange()
print(Org.taste(), Org.rich_in(), Org.colour())
```

OUTPUT

```
Sweet Vitamin A Yellow
Sour Vitamin C Orange
```

Programming Tip: Super falls apart if the methods of subclasses do not take the same arguments.

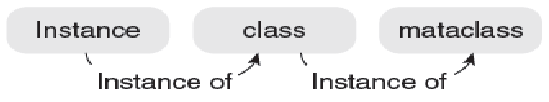
Metaclass

A *metaclass* is the class of a class. While a class defines how an instance of the class behaves, a metaclass, on the other hand, defines how a class behaves. Every class that we create in Python, is an instance of a metaclass.

For example, type is a metaclass in Python. It is itself a class, and it is its own type. Although, you cannot make an exact replica of something like type, but Python does allow you to create a metaclass by making a subclass type.

A *metaclass* is most commonly used as a class-factory. It allows programmers to create an instance of the class by calling the class,

Python allows you to define normal methods on the metaclass which are like classmethods, as they can be called on the class without an instance. You can even define the normal magic methods, such as __add__, __iter__ and __getattr__, to implement or change how the class behaves.



Polymorphism

Python allows programmers to redefine the meaning of operators when they operate on class objects. This feature is called *operator overloading*. Operator overloading allows programmers to extend the meaning of existing operators so that in addition to the basic data types, they can be also applied to user defined data types.

Another form of Polymorphism

Like function overloading, operator overloading is also a form of compile-time polymorphism. Operator overloading, is therefore less commonly known as operator *ad hoc polymorphism* since different operators have different implementations depending on their arguments. Operator overloading is generally defined by the language, the programmer, or both.

Example for Implementing Operator Overloading

```
class Complex:
    def __init__(self):
        self.real = 0
        self.imag = 0
    def setValue(self, real, imag):
        self.real = real
        self.imag = imag
    def display(self):
        print("(", self.real, " + ", self.imag, "i)")
C1 = Complex()
C1.setValue(1,2)
C2 = Complex()
C2.setValue(3,4)
C3 = Complex()
C3 = C1 + C2
C3.display()
```

OUTPUT

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

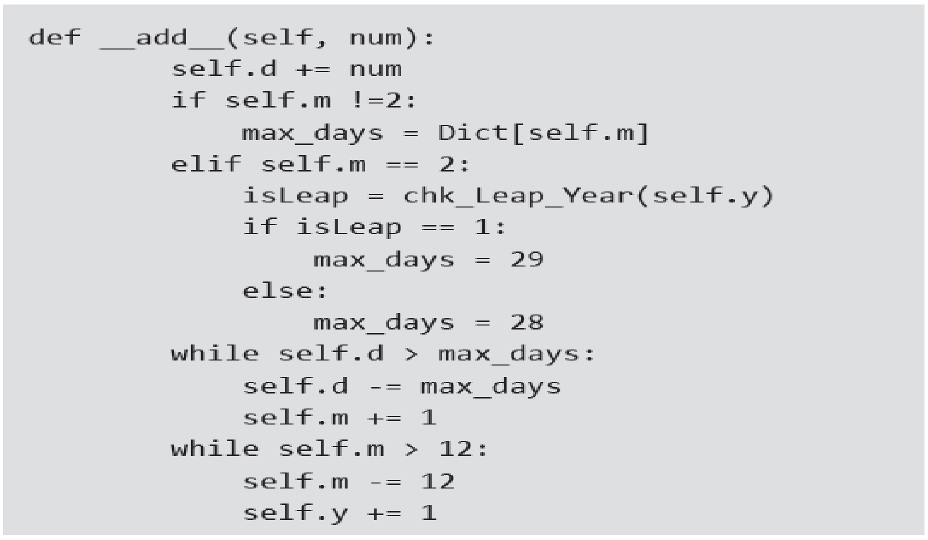
```
  File "C:\Python34\Try.py", line 15, in <module>
    C3 = C1 + C2
TypeError: unsupported operand type(s) for +: 'instance' and 'instance'
```

Operators and Their Corresponding Function Names

Operator	Function Name	Operator	Function Name
+	`__add__`	+=	`__iadd__`
-	`__sub__`	-=	`__isub__`
*	`__mul__`	*=	`__imul__`
/	`__truediv__`	/=	`__idiv__`
**	`__pow__`	**=	`__ipow__`
%	`__mod__`	%=	`__imod__`
>>	`__rshift__`	>>=	`__irshift__`
&	`__and__`	&=	`__iand__`
|	`__or__`	|=	`__ior__`
^	`__xor__`	^=	`__ixor__`
~	`__invert__`	~=	`__iinvert__`
<<	`__lshift__`	<<=	`__ilshift__`
>	`__gt__`	<=	`__le__`
<	`__lt__`	==	`__eq__`
>=	ge	!=	ne

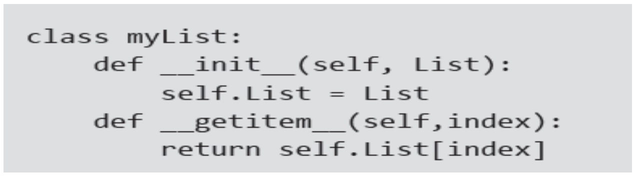
Reverse Adding

In operator overloading functions, we can add a basic data type on a user defined object by writing *user_defined_object + basic_data_type_var* but cannot do the reverse. However, to provide greater flexibility, we should also be able to perform the operation in reverse order, that is, adding a non-class object to the class object. For this, Python provides the concept of reverse adding.

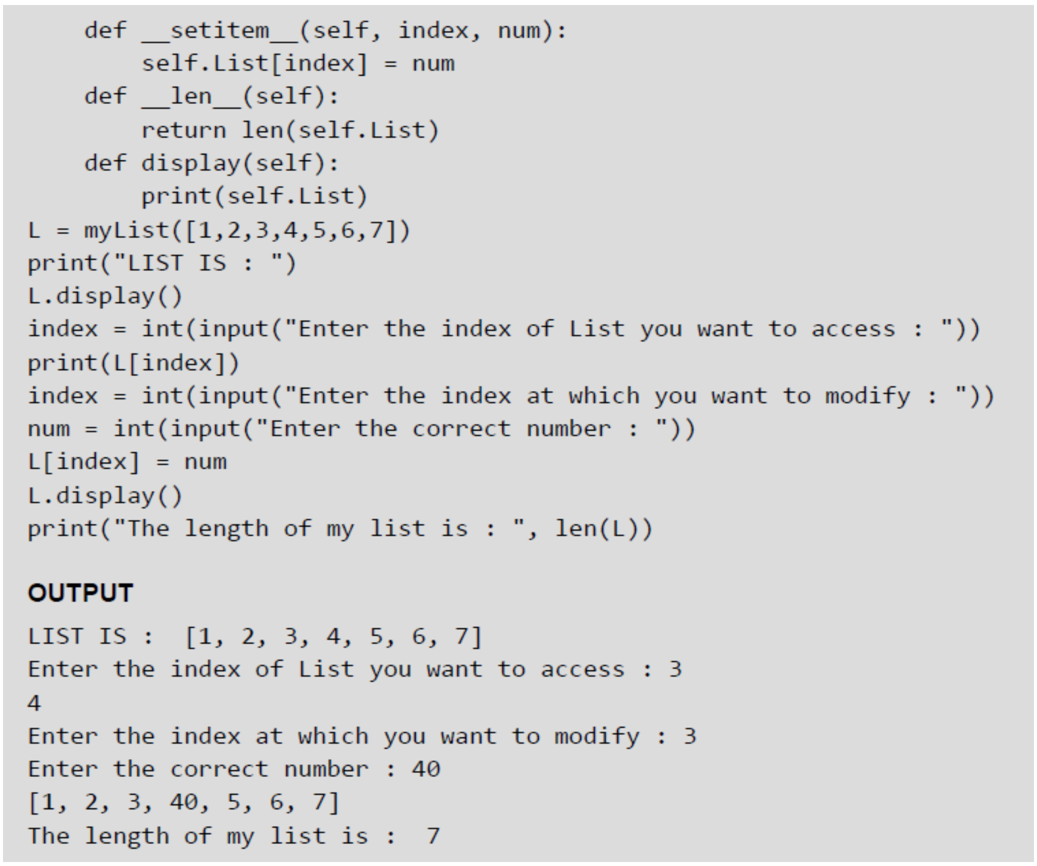


```
def __add__(self, num):
        self.d += num
        if self.m !=2:
            max_days = Dict[self.m]
        elif self.m == 2:
            isLeap = chk_Leap_Year(self.y)
            if isLeap == 1:
                max_days = 29
            else:
                max_days = 28
        while self.d > max_days:
            self.d -= max_days
            self.m += 1
        while self.m > 12:
            self.m -= 12
            self.y += 1
```

Example for Overriding __getitem__() and __setitem__()



```
class myList:
    def __init__(self, List):
        self.List = List
    def __getitem__(self,index):
        return self.List[index]
```



```
    def __setitem__(self, index, num):
        self.List[index] = num
    def __len__(self):
        return len(self.List)
    def display(self):
        print(self.List)
L = myList([1,2,3,4,5,6,7])
print("LIST IS : ")
L.display()
index = int(input("Enter the index of List you want to access : "))
print(L[index])
index = int(input("Enter the index at which you want to modify : "))
num = int(input("Enter the correct number : "))
L[index] = num
L.display()
print("The length of my list is : ", len(L))
```

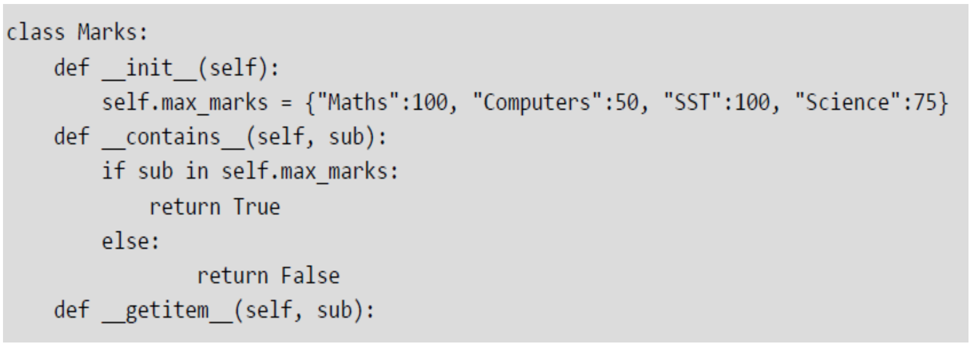
OUTPUT

```
LIST IS :  [1, 2, 3, 4, 5, 6, 7]
Enter the index of List you want to access : 3
4
Enter the index at which you want to modify : 3
Enter the correct number : 40
[1, 2, 3, 40, 5, 6, 7]
The length of my list is :  7
```

Overriding the in **Operator**

in is a membership operator that checks whether the specified item is in the variable of built-in type or not We can overload the same operator to check whether the given value is a member of a class variable or not. To overload the in operator we have to use the function __contains__().

Example:



```
class Marks:
    def __init__(self):
        self.max_marks = {"Maths":100, "Computers":50, "SST":100, "Science":75}
    def __contains__(self, sub):
        if sub in self.max_marks:
            return True
        else:
                return False
    def __getitem__(self, sub):
        return self.max_marks[sub]
    def __str__(self):
       return "The Dictionary has name of subjects and maximum marks allotted to them"
M = Marks()
print(str(M))
sub = input("Enter the subject for which you want to know extra marks : ")
if sub in M:
    print("Social Studies paper has maximum marks alloted = ", M[sub])
```

OUTPUT

```
The Dictionary has name of subjects and maximum marks allotted to them
Enter the subject for which you want to know extra marks : Computers
Social Studies paper has maximum marks alloted =  50
```

Overriding the __call __() Method

The *__call__() method* is used to overload call expressions. The __call__ method is called automatically when an instance of the class is called. It can be passed any positional or keyword arguments. Like other functions, __call__() also supports all of the argument-passing modes.

Example:

```
class Mult:
  def __init__(self, num):
    self.num = num
  def __call__(self, O):
    return self.num * O
x = Mult(10)
print(x(5))
```

OUTPUT

```
50
```

Regular Expressions

Regular Expressions are a powerful tool for various kinds of string manipulation. These are basically a special text string that is used for describing a search pattern to extract information from text such as code, files, log, spreadsheets, or even documents.

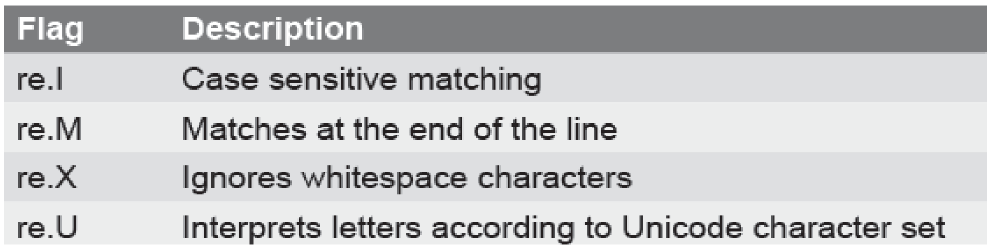
Regular expressions are a *domain specific language* (DSL) that is present as a library in most of the modern programming languages, besides Python. A *regular expression* is a special sequence of characters that helps to match or find strings in another string. In Python, regular expressions can be accessed using the re module which comes as a part of the Standard Library

The Match Function

As the name suggest, the match() function matches a pattern to string with optional flags. The syntax of match() function is,

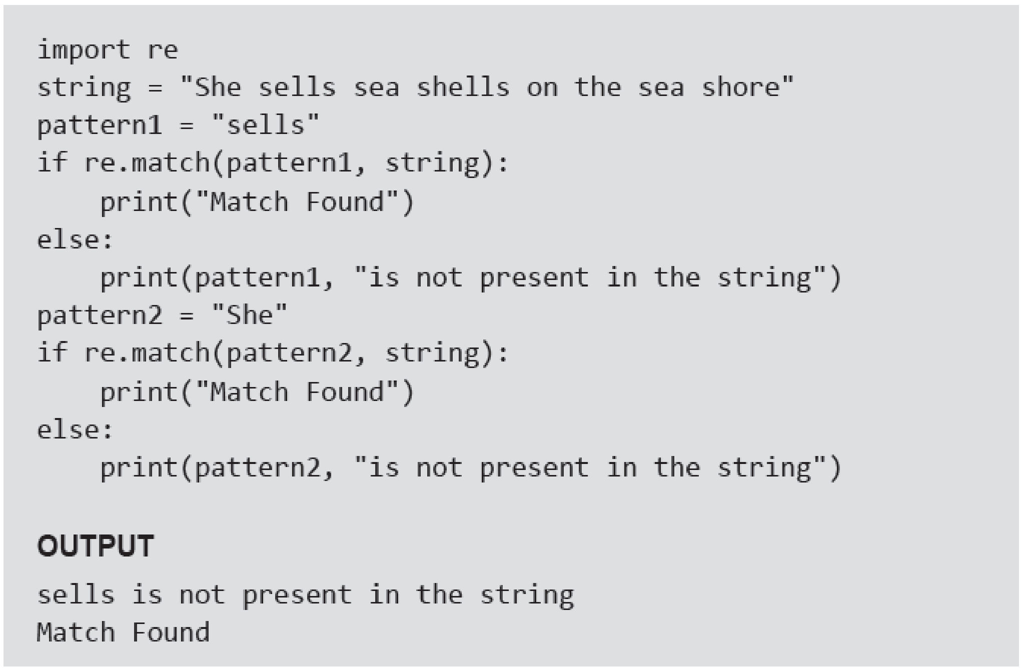
re.match (pattern, string, flags=0)

Match() Function



Flag	Description
re.I	Case sensitive matching
re.M	Matches at the end of the line
re.X	Ignores whitespace characters
re.U	Interprets letters according to Unicode character set

Example



```
import re
string = "She sells sea shells on the sea shore"
pattern1 = "sells"
if re.match(pattern1, string):
    print("Match Found")
else:
    print(pattern1, "is not present in the string")
pattern2 = "She"
if re.match(pattern2, string):
    print("Match Found")
else:
    print(pattern2, "is not present in the string")
```

OUTPUT

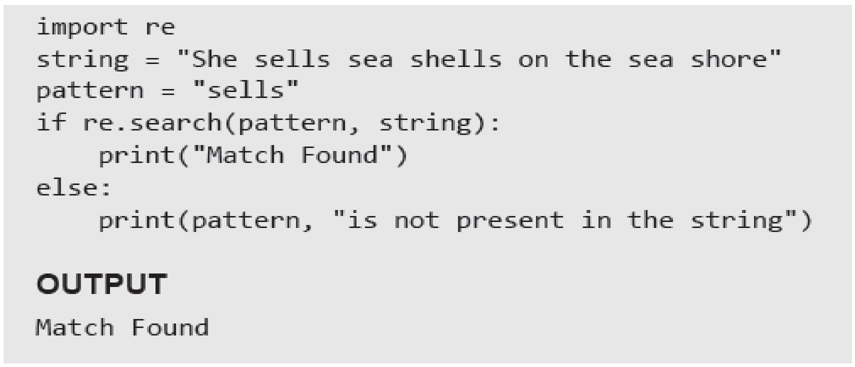
```
sells is not present in the string
Match Found
```

The search Function

The search() function in the re module searches for a pattern anywhere in the string. Its syntax of can be given as, re.search(pattern, string, flags=0)

The syntax is similar to the match() function. The function searches for first occurrence of *pattern* within a *string* with optional *flags*. If the search is successful, a *match* object is returned and None otherwise.

Example:



```
import re
string = "She sells sea shells on the sea shore"
pattern = "sells"
if re.search(pattern, string):
    print("Match Found")
else:
    print(pattern, "is not present in the string")
```

OUTPUT

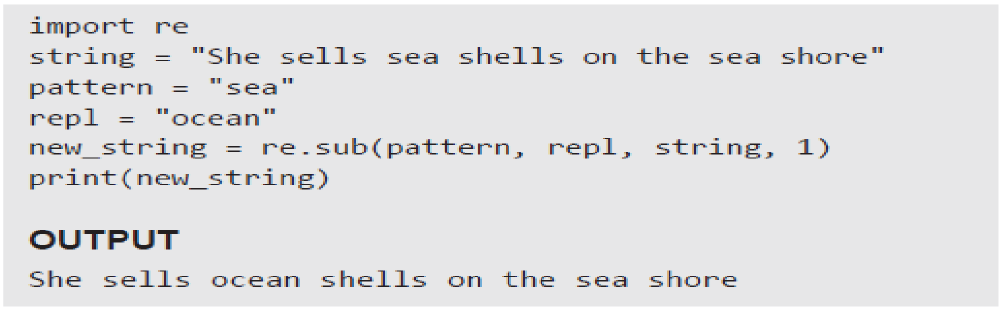
```
Match Found
```

The sub() Function

The sub() function in the re module can be used to search a pattern in the string and replace it with another pattern. The syntax of sub() function can be given as, re.sub(pattern, repl, string, max=0)

According to the syntax, the sub() function replaces all occurrences of the pattern in string with repl, substituting all occurrences unless any max value is provided. This method returns modified string.

Example:



```
import re
string = "She sells sea shells on the sea shore"
pattern = "sea"
repl = "ocean"
new_string = re.sub(pattern, repl, string, 1)
print(new_string)
```

OUTPUT

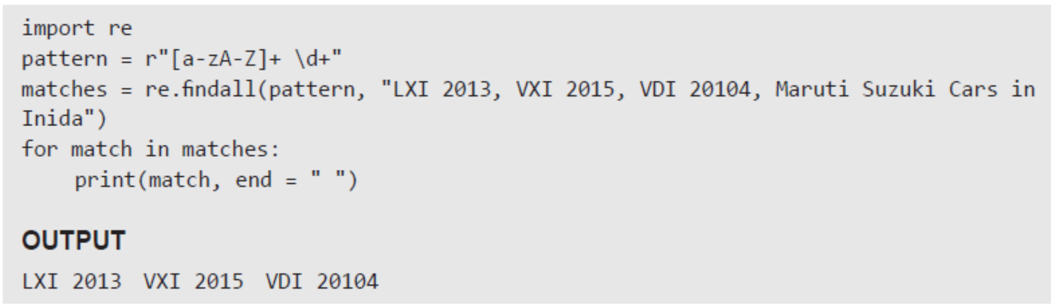
```
She sells ocean shells on the sea shore
```

The findall() and finditer() Function

The findall() function is used to search a string and returns a list of matches of the pattern in the string. If no match is found, then the returned list is empty. The syntax of match() function can be given as,

matchList = re.findall(pattern, input_str, flags=0)

Example:



```
import re
pattern = r"[a-zA-Z]+ \d+"
matches = re.findall(pattern, "LXI 2013, VXI 2015, VDI 20104, Maruti Suzuki Cars in
Inida")
for match in matches:
    print(match, end = " ")
```

OUTPUT

```
LXI 2013  VXI 2015  VDI 20104
```

Flag Options

The search(), findall() and match() functions of the module take options to modify the behavior of the pattern match. Some of these flags are:

re.I or re.IGNORECASE — Ignores case of characters, so "Match", "MATCH", "mAtCh", etc are all same

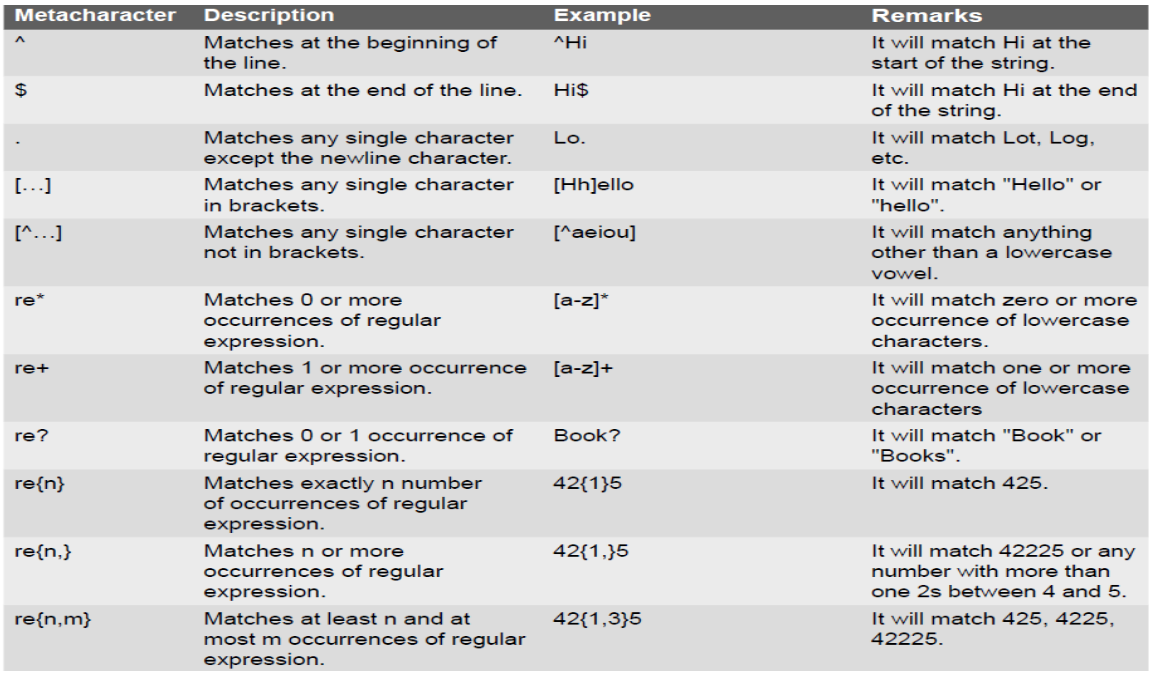
re.S or re.DOTALL — Enables dot (.) to match newline. By default, dot matches any character other than the newline character.

re.M or re.MULTILINE — Makes the ^ and $ to match the start and end of each line. That is, it matches even after and before line breaks in the string. By default, ^ and $ matches the start and end of the whole string.

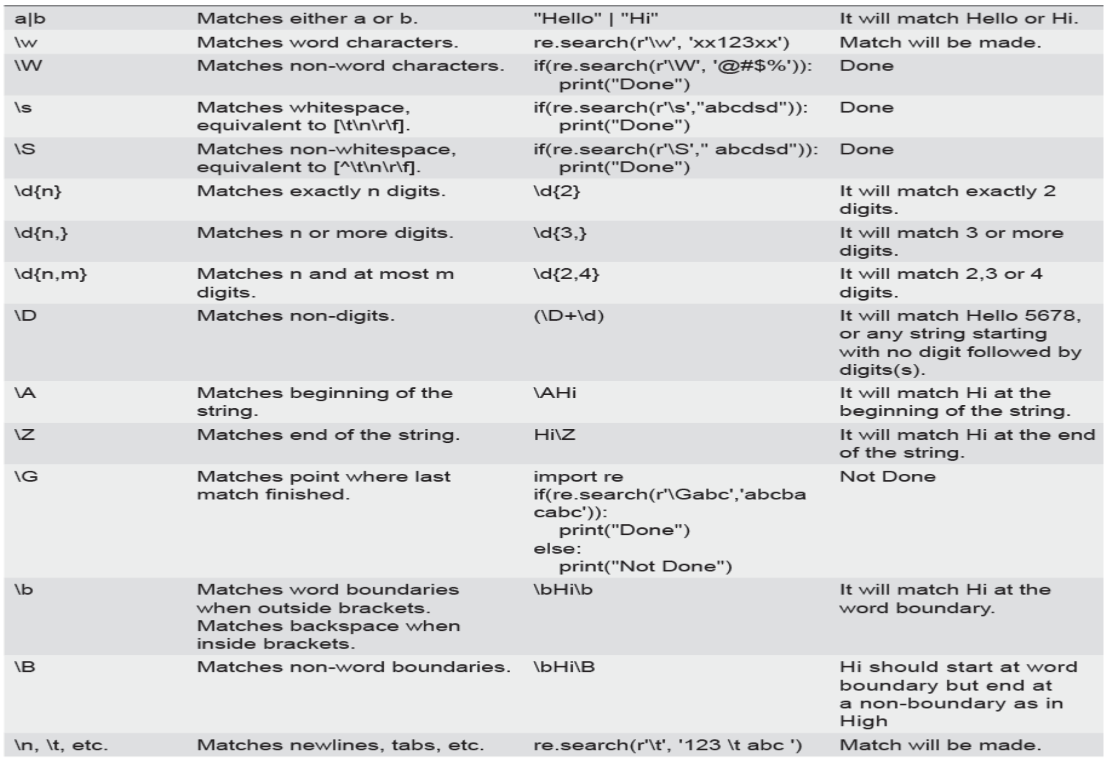
re.L or re.LOCALE- Makes the flag \w to match all characters that are considered letters in the given current locale settings.

re.U or re.UNICODE- Treats all letters from all scripts as word characters.

Metacharacters in Regular Expression



Metacharacter	Description	Example	Remarks
^	Matches at the beginning of the line.	^Hi	It will match Hi at the start of the string.
$	Matches at the end of the line.	Hi$	It will match Hi at the end of the string.
.	Matches any single character except the newline character.	Lo.	It will match Lot, Log, etc.
[...]	Matches any single character in brackets.	[Hh]ello	It will match "Hello" or "hello".
[^...]	Matches any single character not in brackets.	[^aeiou]	It will match anything other than a lowercase vowel.
re*	Matches 0 or more occurrences of regular expression.	[a-z]*	It will match zero or more occurrence of lowercase characters.
re+	Matches 1 or more occurrence of regular expression.	[a-z]+	It will match one or more occurrence of lowercase characters
re?	Matches 0 or 1 occurrence of regular expression.	Book?	It will match "Book" or "Books".
re{n}	Matches exactly n number of occurrences of regular expression.	42{1}5	It will match 425.
re{n,}	Matches n or more occurrences of regular expression.	42{1,}5	It will match 42225 or any number with more than one 2s between 4 and 5.
re{n,m}	Matches at least n and at most m occurrences of regular expression.	42{1,3}5	It will match 425, 4225, 42225.

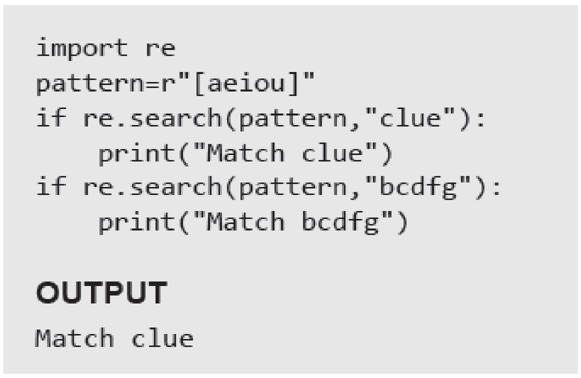


a|b	Matches either a or b.	"Hello" | "Hi"	It will match Hello or Hi.
\w	Matches word characters.	re.search(r'\w', 'xx123xx')	Match will be made.
\W	Matches non-word characters.	if(re.search(r'\W', '@#$%')): print("Done")	Done
\s	Matches whitespace, equivalent to [\t\n\r\f].	if(re.search(r'\s',"abcdsd")): print("Done")	Done
\S	Matches non-whitespace, equivalent to [^\t\n\r\f].	if(re.search(r'\S'," abcdsd")): print("Done")	Done
\d{n}	Matches exactly n digits.	\d{2}	It will match exactly 2 digits.
\d{n,}	Matches n or more digits.	\d{3,}	It will match 3 or more digits.
\d{n,m}	Matches n and at most m digits.	\d{2,4}	It will match 2,3 or 4 digits.
\D	Matches non-digits.	(\D+\d)	It will match Hello 5678, or any string starting with no digit followed by digits(s).
\A	Matches beginning of the string.	\AHi	It will match Hi at the beginning of the string.
\Z	Matches end of the string.	Hi\Z	It will match Hi at the end of the string.
\G	Matches point where last match finished.	import re if(re.search(r'\Gabc','abcba cabc')): print("Done") else: print("Not Done")	Not Done
\b	Matches word boundaries when outside brackets. Matches backspace when inside brackets.	\bHi\b	It will match Hi at the word boundary.
\B	Matches non-word boundaries.	\bHi\B	Hi should start at word boundary but end at a non-boundary as in High
\n, \t, etc.	Matches newlines, tabs, etc.	re.search(r'\t', '123 \t abc ')	Match will be made.

Character Classes

When we put the characters to be matched inside square brackets, we call it a character class. For example, [aeiou] defines a character class that has a vowel character.

Example:



```
import re
pattern=r"[aeiou]"
if re.search(pattern,"clue"):
    print("Match clue")
if re.search(pattern,"bcdfg"):
    print("Match bcdfg")
```

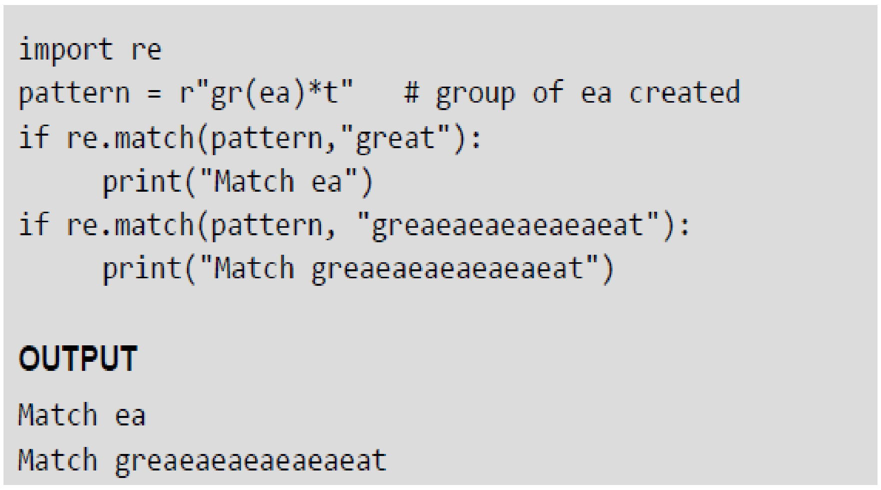
OUTPUT

```
Match clue
```

Groups

A group is created by surrounding a part of the regular expression with parentheses. You can even give group as an argument to the metacharacters such as * and ?.

Example:



```
import re
pattern = r"gr(ea)*t"   # group of ea created
if re.match(pattern,"great"):
     print("Match ea")
if re.match(pattern, "greaeaeaeaeaeaeat"):
     print("Match greaeaeaeaeaeaeat")
```

OUTPUT

```
Match ea
Match greaeaeaeaeaeaeat
```

The content of groups in a match can be accessed by using the group() function. For example,

group(0) or group() returns the whole match.

group(n), where n is greater than 0, returns the nth group from the left.

group() returns all groups up from 1.

File

A file is a collection of data stored on a secondary storage device like hard disk.

A file is basically used because real-life applications involve large amounts of data and in such situations the console oriented I/O operations pose two major problems:

1. First, it becomes cumbersome and time consuming to handle huge amount of data through terminals.
2. Second, when doing I/O using terminal, the entire data is lost when either the program is terminated or computer is turned off. Therefore, it becomes necessary to store data on a permanent storage (the disks) and read whenever necessary, without destroying the data.

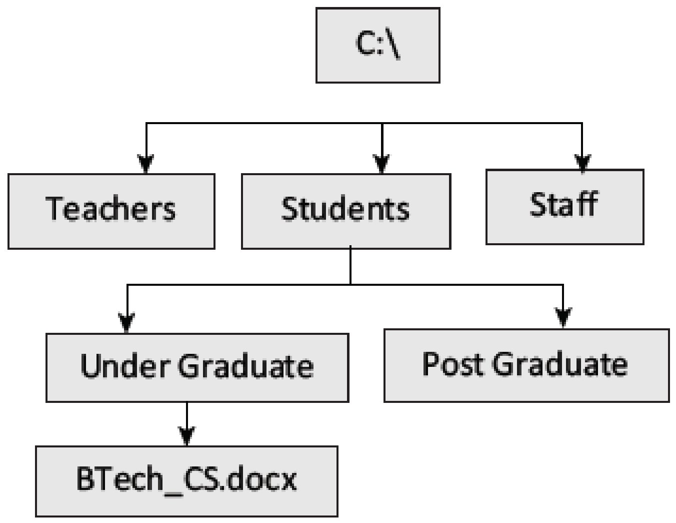
File Path

Files that we use are stored on a storage medium like the hard disk in such a way that they can be easily retrieved as and when required.

Every file is identified by its path that begins from the root node or the root folder. In Windows, C:\ (also known as C drive) is the root folder but you can also have a path that starts from other drives like D:\, E:\, etc. The file path is also known as *pathname.*

Relative Path and Absolute Path

A file path can be either *relative* or *absolute*. While an absolute path always contains the root and the complete directory list to specify the exact location the file, relative path needs to be combined with another path in order to access a file. It starts with respect to the current working directory and therefore lacks the leading slashes. For example, C:\Students\Under Graduate\BTech_CS.docx but Under Graduate\BTech_CS.docx is a relative path as only a part of the complete path is specified.



ASCII Text Files

A *text file* is a stream of characters that can be sequentially processed by a computer in forward direction. For this reason a text file is usually opened for only one kind of operation (reading, writing, or appending) at any given time. Because text files can process characters, they can only read or write data one character at a time. In Python, a text stream is treated as a special kind of file.

Depending on the requirements of the operating system and on the operation that has to be performed (read/write operation) on the file, the newline characters may be converted to or from carriage-return/linefeed combinations. Besides this, other character conversions may also be done to satisfy the storage requirements of the operating system. However, these conversions occur transparently to process a text file. In a text file, each line contains zero or more characters and ends with one or more characters

Another important thing is that when a text file is used, there are actually two representations of data- internal or external. For example, an integer value will be represented as a number that occupies 2 or 4 bytes of memory internally but externally the integer value will be represented as a string of characters representing its decimal or hexadecimal value.

Binary Files

A *binary file* is a file which may contain any type of data, encoded in binary form for computer storage and processing purposes. It includes files such as word processing documents, PDFs, images, spreadsheets, videos, zip files and other executable programs. Like a text file, a binary file is a collection of bytes. A binary file is also referred to as a character stream with following two essential differences.

• A binary file does not require any special processing of the data and each byte of data is transferred to or from the disk unprocessed.

• Python places no constructs on the file, and it may be read from, or written to, in any manner the programmer wants.

While text files can be processed sequentially, binary files, on the other hand, can be either processed sequentially or randomly depending on the needs of the application.

The Open() Function

Before reading from or writing to a file, you must first open it using Python's built-in open() function. This function creates a file object, which will be used to invoke methods associated with it. The syntax of open() is:

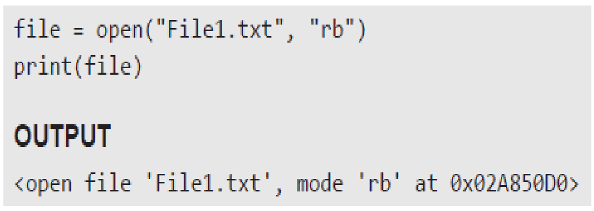
fileObj = open(file_name [, access_mode])

Here,

file_name is a string value that specifies name of the file that you want to access.

access_mode indicates the mode in which the file has to be opened, i.e., read, write, append, etc.

Example:

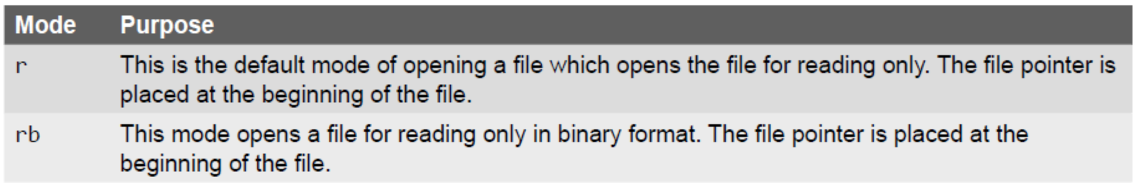


```
file = open("File1.txt", "rb")
print(file)
```

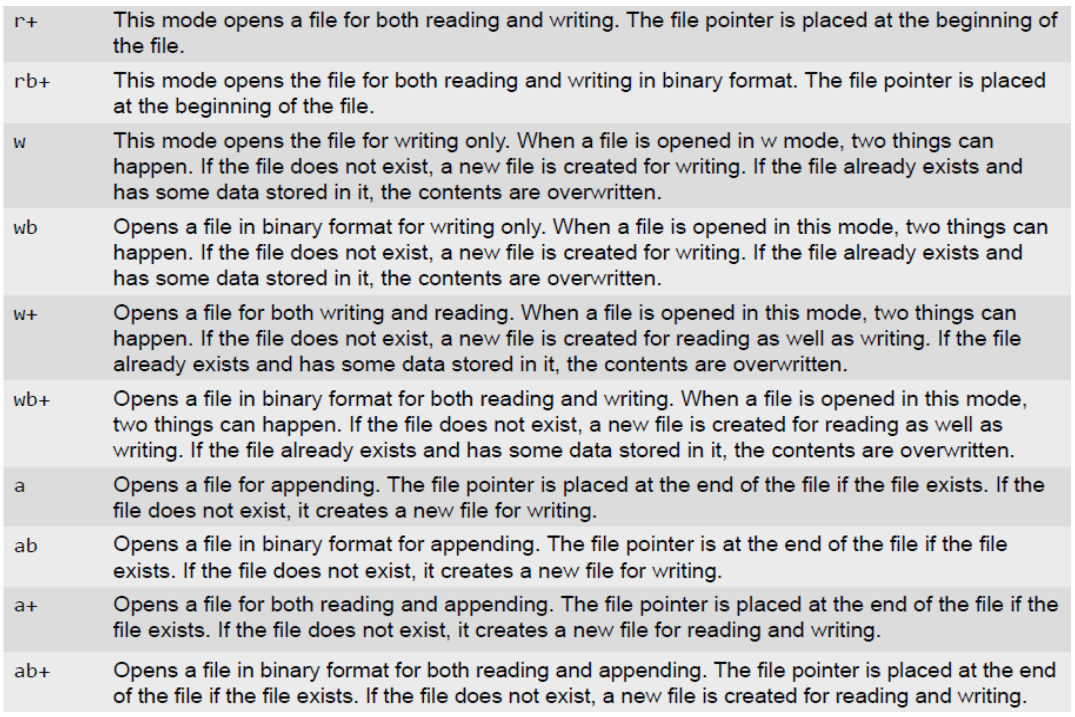
OUTPUT

```
<open file 'File1.txt', mode 'rb' at 0x02A850D0>
```

Access Modes:



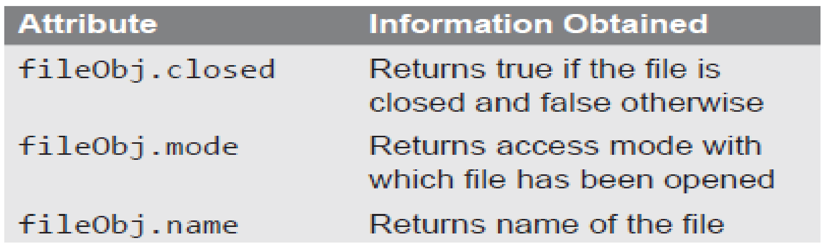
Mode	Purpose
r	This is the default mode of opening a file which opens the file for reading only. The file pointer is placed at the beginning of the file.
rb	This mode opens a file for reading only in binary format. The file pointer is placed at the beginning of the file.



r+	This mode opens a file for both reading and writing. The file pointer is placed at the beginning of the file.
rb+	This mode opens the file for both reading and writing in binary format. The file pointer is placed at the beginning of the file.
w	This mode opens the file for writing only. When a file is opened in w mode, two things can happen. If the file does not exist, a new file is created for writing. If the file already exists and has some data stored in it, the contents are overwritten.
wb	Opens a file in binary format for writing only. When a file is opened in this mode, two things can happen. If the file does not exist, a new file is created for writing. If the file already exists and has some data stored in it, the contents are overwritten.
w+	Opens a file for both writing and reading. When a file is opened in this mode, two things can happen. If the file does not exist, a new file is created for reading as well as writing. If the file already exists and has some data stored in it, the contents are overwritten.
wb+	Opens a file in binary format for both reading and writing. When a file is opened in this mode, two things can happen. If the file does not exist, a new file is created for reading as well as writing. If the file already exists and has some data stored in it, the contents are overwritten.
a	Opens a file for appending. The file pointer is placed at the end of the file if the file exists. If the file does not exist, it creates a new file for writing.
ab	Opens a file in binary format for appending. The file pointer is at the end of the file if the file exists. If the file does not exist, it creates a new file for writing.
a+	Opens a file for both reading and appending. The file pointer is placed at the end of the file if the file exists. If the file does not exist, it creates a new file for reading and writing.
ab+	Opens a file in binary format for both reading and appending. The file pointer is placed at the end of the file if the file exists. If the file does not exist, a new file is created for reading and writing.

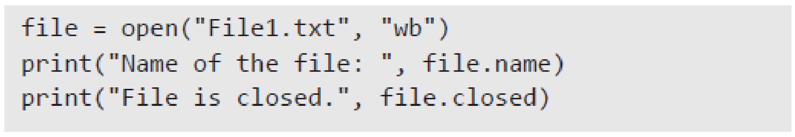
The File Object Attributes

Once a file is successfully opened, a *file* object is returned. Using this file object, you can easily access different type of information related to that file. This information can be obtained by reading values of specific attributes of the file**.**

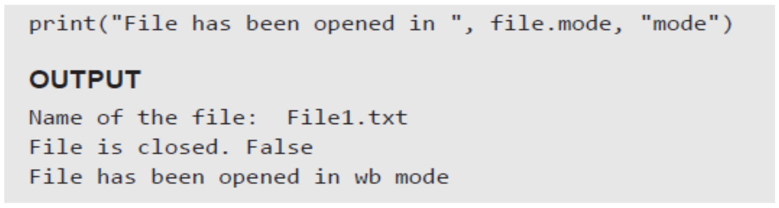


Attribute	Information Obtained
fileObj.closed	Returns true if the file is closed and false otherwise
fileObj.mode	Returns access mode with which file has been opened
fileObj.name	Returns name of the file

Example:



```
file = open("File1.txt", "wb")
print("Name of the file: ", file.name)
print("File is closed.", file.closed)
```



```
print("File has been opened in ", file.mode, "mode")
```

OUTPUT

```
Name of the file:  File1.txt
File is closed. False
File has been opened in wb mode
```

The close () Method

The *close() method* is used to close the file object. Once a file object is closed, you cannot further read from or write into the file associated with the file object. While closing the file object the close() flushes any unwritten information. Although, Python automatically closes a file when the reference object of a file is reassigned to another file, but as a good programming habit you should always explicitly use the close() method to close a file. The syntax of close() is fileObj.close()

The close() method frees up any system resources such as file descriptors, file locks, etc. that are associated with the file. Moreover, there is an upper limit to the number of files a program can open. If that limit is exceeded then the program may even crash or work in unexpected manner. Thus, you can waste lots of memory if you keep many files open unnecessarily and also remember that open files always stand a chance of corruption and data loss.

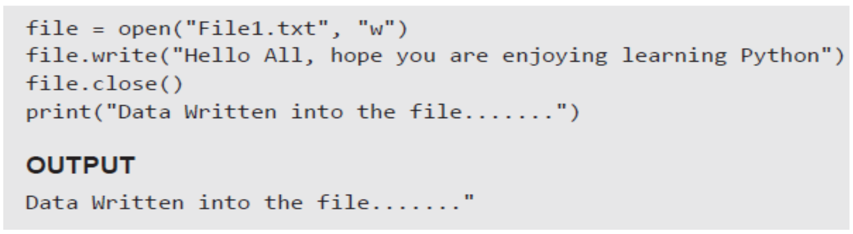
Once the file is closed using the close() method, any attempt to use the file object will result in an error.

The write() and writelines() Methods

The *write() method* is used to write a string to an already opened file. Of course this string may include numbers, special characters or other symbols. While writing data to a file, you must remember that the write() method does not add a newline character ('\n') to the end of the string. The syntax of write() method is: fileObj.write(string)

The *writelines() method* is used to write a list of strings.

Example:



```
file = open("File1.txt", "w")
file.write("Hello All, hope you are enjoying learning Python")
file.close()
print("Data Written into the file.......")
```

OUTPUT

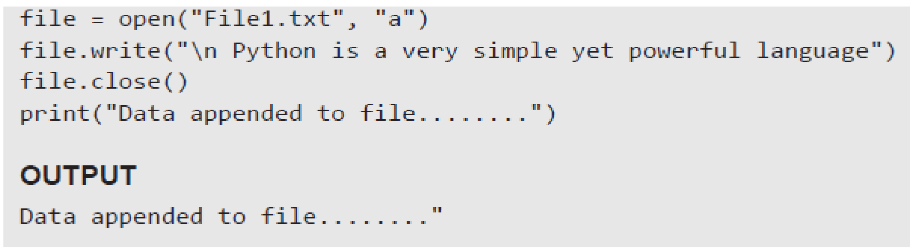
```
Data Written into the file......."
```

append() Method

Once you have stored some data in a file, you can always open that file again to write more data or append data to it. To append a file, you must open it using 'a' or 'ab' mode depending on whether it is a text file or a binary file. Note that if you open a file in 'w' or 'wb' mode and then start writing data into it, then its existing contents would be overwritten. So always open the file in 'a' or 'ab' mode to add more data to existing data stored in the file.

Appending data is especially essential when creating a log of events or combining a large set of data into one file.

Example:



```
file = open("File1.txt", "a")
file.write("\n Python is a very simple yet powerful language")
file.close()
print("Data appended to file........")
```

OUTPUT

```
Data appended to file........"
```

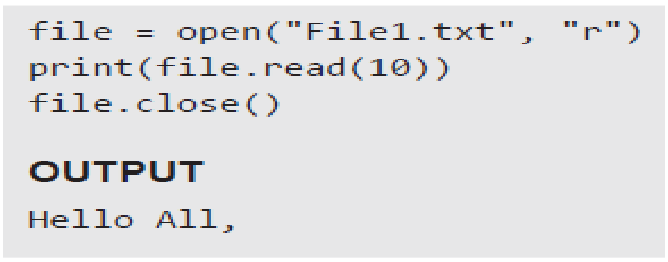
The read() and readline() Methods

The *read() method* is used to read a string from an already opened file. As said before, the string can include, alphabets, numbers, characters or other symbols. The syntax of read() method is fileObj.read([count])

In the above syntax, count is an optional parameter which if passed to the read() method specifies the number of bytes to be read from the opened file. The read() method starts reading from the beginning of the file and if *count* is missing or has a negative value then, it reads the entire contents of the file (i.e., till the end of file).

The *readlines() method* is used to read all the lines in the file

Example:



```
file = open("File1.txt", "r")
print(file.read(10))
file.close()
```

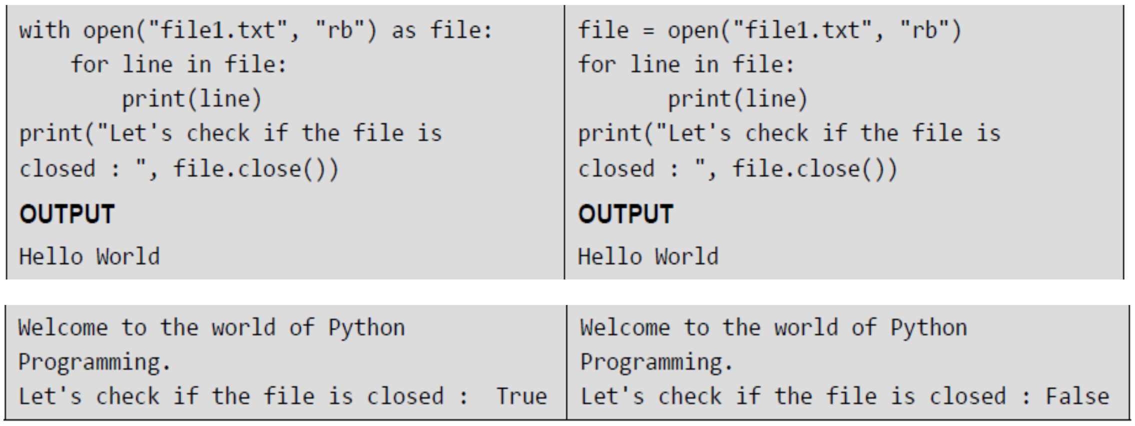
OUTPUT

```
Hello All,
```

Opening Files using "with" Keyword

It is good programming habit to use the with keyword when working with file objects. This has the advantage that the file is properly closed after it is used even if an error occurs during read or write operation or even when you forget to explicitly close the file.

Example:



```
with open("file1.txt", "rb") as file:
    for line in file:
        print(line)
print("Let's check if the file is
closed : ", file.close())
```

OUTPUT

```
Hello World
Welcome to the world of Python
Programming.
Let's check if the file is closed :  True
```

```
file = open("file1.txt", "rb")
for line in file:
       print(line)
print("Let's check if the file is
closed : ", file.close())
```

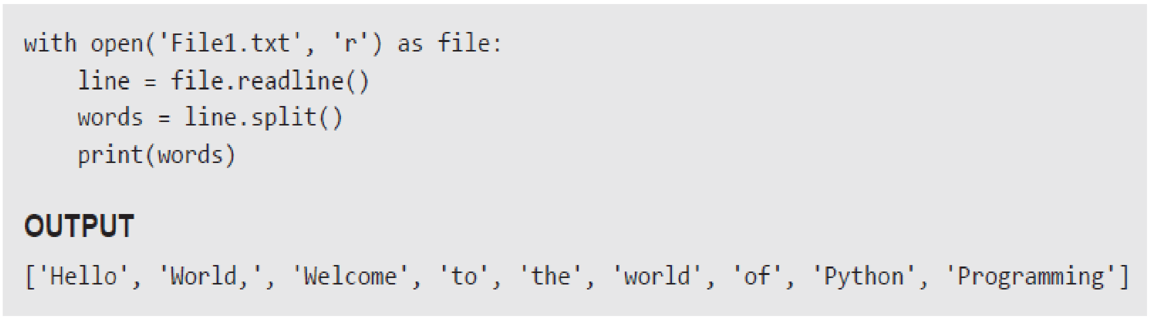
OUTPUT

```
Hello World
Welcome to the world of Python
Programming.
Let's check if the file is closed : False
```

Splitting Words

Python allows you to read line(s) from a file and splits the line (treated as a string) based on a character. By default, this character is space but you can even specify any other character to split words in the string.

Example:

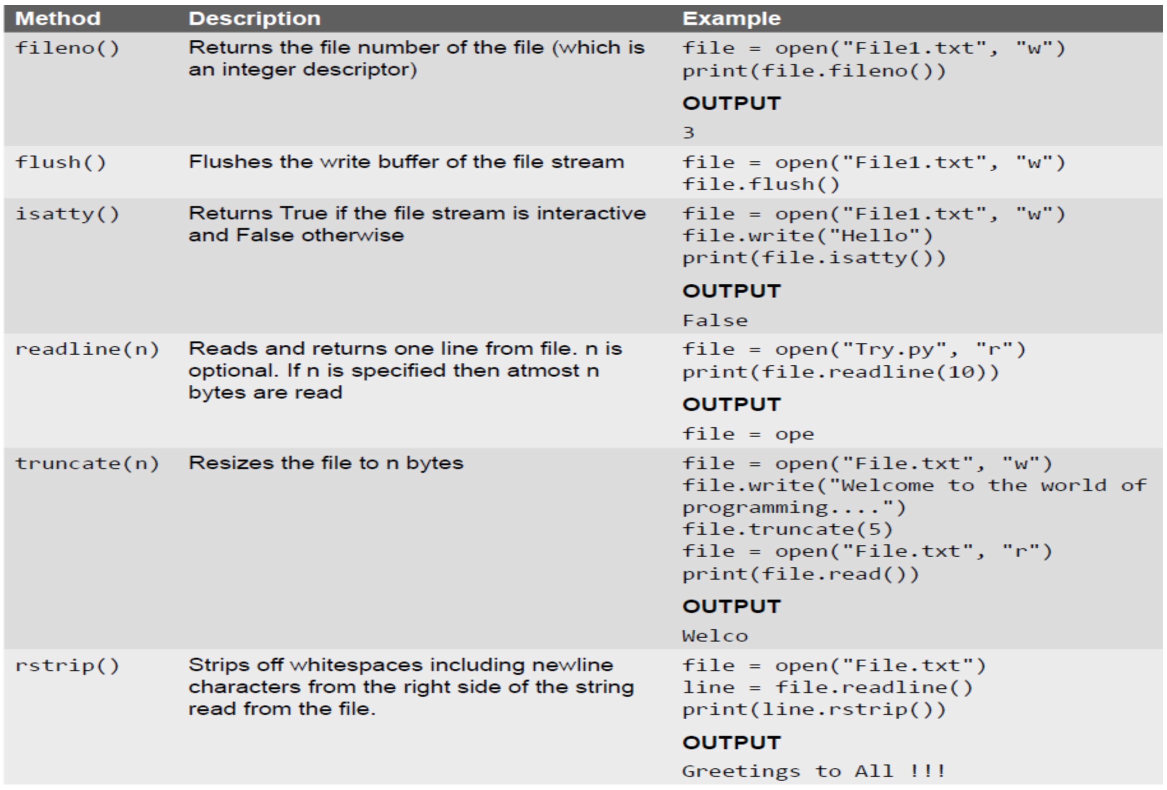


```
with open('File1.txt', 'r') as file:
    line = file.readline()
    words = line.split()
    print(words)
```

OUTPUT

```
['Hello', 'World,', 'Welcome', 'to', 'the', 'world', 'of', 'Python', 'Programming']
```

Some Other Useful File Methods



Method	Description	Example

`fileno()`	Returns the file number of the file (which is an integer descriptor)	`file = open("File1.txt", "w")`
		`print(file.fileno())`
		OUTPUT
		`3`

`flush()`	Flushes the write buffer of the file stream	`file = open("File1.txt", "w")`
		`file.flush()`

`isatty()`	Returns True if the file stream is interactive and False otherwise	`file = open("File1.txt", "w")`
		`file.write("Hello")`
		`print(file.isatty())`
		OUTPUT
		`False`

`readline(n)`	Reads and returns one line from file. n is optional. If n is specified then atmost n bytes are read	`file = open("Try.py", "r")`
		`print(file.readline(10))`
		OUTPUT
		`file = ope`

`truncate(n)`	Resizes the file to n bytes	`file = open("File.txt", "w")`
		`file.write("Welcome to the world of programming....")`
		`file.truncate(5)`
		`file = open("File.txt", "r")`
		`print(file.read())`
		OUTPUT
		`Welco`

`rstrip()`	Strips off whitespaces including newline characters from the right side of the string read from the file.	`file = open("File.txt")`
		`line = file.readline()`
		`print(line.rstrip())`
		OUTPUT
		`Greetings to All !!!`

File Positions

With every file, the file management system associates a pointer often known as *file pointer* that facilitate the movement across the file for reading and/ or writing data. The file pointer specifies a location from where the current read or write operation is initiated. Once the read/write operation is completed, the pointer is automatically updated.

Python has various methods that tells or sets the position of the file pointer. For example, the tell() method tells the current position within the file at which the next read or write operation will occur. It is specified as number of bytes from the beginning of the file. When you just open a file for reading, the file pointer is positioned at location 0, which is the beginning of the file.

The seek(offset[, from]) method is used to set the position of the file pointer or in simpler terms, move the file pointer to a new location. The offset argument indicates the number of bytes to be moved and the from argument specifies the reference position from where the bytes are to be moved.

Example:

```
file = open("File1.txt", "rb")
print("Position of file pointer before reading is : ", file.tell())
print(file.read(10))
print("Position of file pointer after reading is : ", file.tell())
print("Setting 3 bytes from the current position of file pointer")
file.seek(3,1)
print(file.read())
file.close()
```

OUTPUT

```
Position of file pointer before reading is : 0
Hello All,
Position of file pointer after reading is : 10
Setting 3 bytes from the current position of file pointer
pe you are enjoying learning Python
```

Renaming and Deleting Files

The os module in Python has various methods that can be used to perform file-processing operations like renaming and deleting files. To use the methods defined in the os module, you should first import it in your program then call any related functions.

The rename() Method: The *rename()* method takes two arguments, the current filename and the new filename. Its syntax is: os.rename(old_file_name, new_file_name)

The remove() Method: This method can be used to delete file(s). The method takes a filename (name of the file to be deleted) as an argument and deletes that file. Its syntax is: os.remove(file_name)

```
import os
os.rename("File1.txt", "Students.txt")
print("File Renamed")
```

OUTPUT

```
File Renamed
```

```
import os
os.remove("File1.txt")
print("File Deleted")
```

OUTPUT

```
File Deleted
```

Directory Methods

The mkdir() Method: The mkdir()method of the OS module is used to create directories in the current directory. The method takes the name of the directory (the one to be created) as an argument. The syntax of mkdir() is, os.mkdir("new_dir_name")

The getcwd() Method: The getcwd() method is used to display the current working directory (cwd).

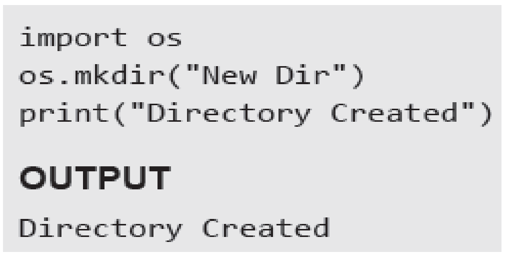
os.getcwd()

The chdir() Method: The chdir() method is used to change the current directory. The method takes the name of the directory which you want to make the current directory as an argument. Its syntax is

os.chdir("dir_name")

The rmdir() Method: The *rmdir()* method is used to remove or delete a directory. For this, it accepts name of the directory to be deleted as an argument. However, before removing a directory, it should be absolutely empty and all the contents in it should be removed. The syntax of remove() method is os.rmdir(("dir_name")

The makedirs() method: The method mkdirs() is used to create more than one folder.



```
import os
os.mkdir("New Dir")
print("Directory Created")
```

OUTPUT

```
Directory Created
```

Methods from the os Module

The os.path.abspath() method uses the string value passed to it to form an absolute path. Thus, it is another way to convert a relative path to an absolute path

The os.path.isabs(path) method accepts a file path as an argument and returns True if the path is an absolute path and False otherwise.

The os.path.relpath(path, start) method accepts a file path and a start string as an argument and returns a relative path that begins from the start. If start is not given, the current directory is taken as start.

The os.path.dirname(path) Method returns a string that includes everything specified in the path (passed as argument to the method) that comes before the last slash.

The os.path.basename(path) Method returns a string that includes everything specified in the path (passed as argument to the method) that comes after the last slash.

The os.path.split(path) Method: This method accepts a file path and returns its directory name as well as the . So it is equivalent to using two separate methods os.path.dirname() and os.path.basename()

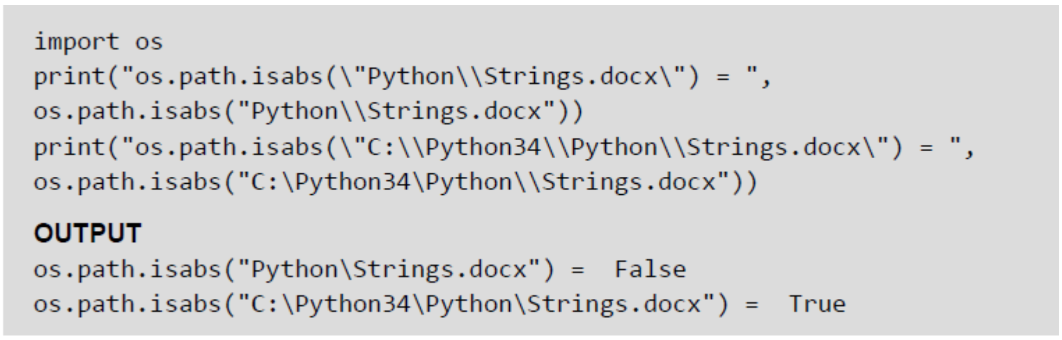
The os.path.getsize(path) Method: This method returns the size of the file specified in the path argument.

The os.listdir(path) Method: The method returns a list of filenames in the specified path.

The os.path.exists(path) Method: The method as the name suggests accepts a path as an argument and returns True if the file or folder specified in the path exists and False otherwise.

The os.path.isfile(path) Method: The method as the name suggests accepts a path as an argument and returns True if the path specifies a file and False otherwise.

The os.path.isdir(path) Method: The method as the name suggests accepts a path as an argument and returns True if the path specifies a folder and False otherwise.



```
import os
print("os.path.isabs(\"Python\\Strings.docx\") = ",
os.path.isabs("Python\\Strings.docx"))
print("os.path.isabs(\"C:\\Python34\\Python\\Strings.docx\") = ",
os.path.isabs("C:\Python34\Python\\Strings.docx"))
```

OUTPUT

```
os.path.isabs("Python\Strings.docx") =  False
os.path.isabs("C:\Python34\Python\Strings.docx") =  True
```

Questions

1. What is inheritance and explain types of inheritance. [CO1][BT1]
2. Differentiate between base class and derived class. [CO1][BT2]
3. Explain super() function with an example. [CO2][BT4]
4. Explain multiple inheritance with an example. [CO2][BT5]
5. How does inheritance allow users to reuse code ? [CO1][BT6]
6. What is multilevel inheritance ? [CO2][BT2]
7. What will happen when a call inherits from another class with the same attributes or methods? Will it override them ? [CO2][BT4]
8. Differentiate between the following[CO2][BT3]
    a) Simple, multiple & multilevel inheritance
    b) Inheritance and composition
    c) Containership and aggregation
9. What are abstract classes? Explain with an example.[CO2][BT2]
10. Define the term operator overloading with an example. [CO2][BT1]
11. List advantages of operator overloading. [CO1][BT4]
12. Differentiate between _add_, _radd_, and _iadd_ functions. [CO2][BT2]
13. When is the _call()_ method invoked. [CO2][BT2]
14. What are the files? Why do we need them ? [CO1][BT2]
15. Differentiate between a file and folder. [CO1][BT2]
16. Explain the utility of open() function. [CO2][BT3]
17. Differentiate between text and binary file. [CO1][BT2]
18. Give the significance of with keyword. [CO2][BT4]
19. Explain the syntax of read() method. [CO2][BT1]
20. Define file and explain types of files. [CO1][BT1]