Python programming language

Table of Contents

Output

Compound Operator	Sample Expression	Expanded Form
+=	a+=2	a=a+2
-=	a-=6	a=a-6
*=	a*=7	a=a*7
/=	a/=4	a=a/4
%=	a%=9	a=a%9
**=	a**=3	a=a**3
//=	a//=2	a=a//2

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

>=	**Greater than or equal to operator**

Operator	**Description**
&	**Bitwise AND**
|	**Bitwise OR**
^	**Bitwise XOR**
~	**Bitwise NOT**
<<	**Left shift**
>>	**Right shift**

Date Time Functions

The strftime() function is used to convert date and time objects to their string representation.

- **datetime.now(): Returns the current date and time.**
- **datetime.date(): Returns date object of today's date.**
- **datetime.time(): Returns the current time.**
- **datetime.datetime(): Returns the current date and time as a datetime object.**
- **datetime.timedelta(days=0, seconds=0, microseconds=0, milliseconds=0, minutes=0, hours=0, weeks=0): Represents the difference between two date or time values.**

Syntax :

strftime (format)

List of format codes

Directive	Description	Example
%A	Weekday full name	Monday
%a	Weekday short name	Mon
%d	Day of month (1-31)	26
%b	Month of short name	Dec
%B	Month of full name	December
%Y	Year of full version , without century	2022
%y	Year of short version	22
%w	Weekday as a number (0-Sun , 1-Mon , 2-Tue , 3-Wed , 4-Thu , 5-Fri , 6-Sat)	1(Monday)
%W	Week number of Year (Monday as the first day of week (00-53))	48
%m	Month as a Number (1(Jun) - 12(Dec))	12(Dec)
%H	Hours (00-23)	15
%M	Minute (00-59)	50

%S	Second (00-59)	23
%p	PM / AM	PM
%c	Local version of date and time	Mon Dec 26 15 : 50 : 23 2022
%X	Local version of time	15:50:23
%x	Local version of date	12/26/22

Class & object

A class is a blueprint or serves as a template from which individual objects are created

Object is an instance of a class which consists of methods and properties

Syntax of Class:

```
class Class_Name :
    # statements
```

Example of Class:

```
class student :
    name = " Tutor joe's "
    age = 30
```

Syntax of Object:

```
object_name = class_name ( arguments )
```

Example of Object:

```
s = student ( )
```

```
class car():
    pass



a = 10
print(type(a))
```

```
print(type(car))
swift=car()


print(isinstance(swift,car))
print(isinstance(a,int))
print(type(swift))
```

Output

```
<class 'int'>
<class 'type'>
True
True
<class '__main__.car'>
```

Class Attributes

Class attributes belong to the class itself they will be shared by all the instances. Such attributes are defined in the class body parts usually at the top, for legibility.

```
class Student():
    name = "Ram Kumar"
    age = 25



''' This is Class Attributes '''


# getattr method
print(getattr(Student, 'name'))
print(getattr(Student, 'age'))
print(getattr(Student, 'gender', 'No Such Attribute Found'))


# Dot Notation
print(Student.name)
print(Student.age)


# setattr
setattr(Student, 'name', 'Tutor Joes')
print(Student.name)


setattr(Student, 'gender', 'Male')
```

```
print(Student.gender)


Student.city = "Salem"

print(Student.city)


print(Student.__dict__)

delattr(Student,"city")

print(Student.__dict__)

del Student.gender

print(Student.__dict__)
```

Output

```
Ram Kumar

25

No Such Attribute Found

Ram Kumar

25

Tutor Joes

Male

Salem

{'__module__': '__main__', 'name': 'Tutor Joes', 'age': 25, '__dict__': <attribute '__dict__' of
'Student' objects>, '__weakref__': <attribute '__weakref__' of 'Student' objects>, '__doc__':
None, 'gender': 'Male', 'city': 'Salem'}

{'__module__': '__main__', 'name': 'Tutor Joes', 'age': 25, '__dict__': <attribute '__dict__' of
'Student' objects>, '__weakref__': <attribute '__weakref__' of 'Student' objects>, '__doc__':
None, 'gender': 'Male'}

{'__module__': '__main__', 'name': 'Tutor Joes', 'age': 25, '__dict__': <attribute '__dict__' of
'Student' objects>, '__weakref__': <attribute '__weakref__' of 'Student' objects>, '__doc__':
None}
```

Instance Attributes

```
class user:

    course = 'Java'



o = user()

print(user.__dict__)

print(user.course)  # Print Class attribute

print(o.__dict__)

print(o.course)   → instance attribute
```

```
o.course = "C++"

print(o.__dict__)

print(o.course)


o2 = user()

print(o2.course)
```

Output

```
{'__module__': '__main__', 'course': 'Java', '__dict__': <attribute '__dict__' of 'user' objects>, '__weakref__': <attribute '__weakref__' of 'user' objects>, '__doc__': None}

Java

{}

Java

{'course': 'C++'}

C++

Java
```

Class Method

```
# Class Methods

class Student:

    name = "Tutor Joes"

    age = 25


    def printall():

        print("Name : ", Student.name)

        print("Age  : ", Student.age)



Student.printall()

print(Student.__dict__)


print(getattr(Student, "printall"))

getattr(Student, "printall")()


Student.__dict__['printall']()
```

Output

```
Name :  Tutor Joes

Age  :  25
```

```
{'__module__': '__main__', 'name': 'Tutor Joes', 'age': 25, 'printall': <function Student.printall at 0x000001F08DD5B5E0>, '__dict__': <attribute '__dict__' of 'Student' objects>>, '__weakref__': , '__doc__': None}


Name :  Tutor Joes
Age  :  25
Name :  Tutor Joes
Age  :  25
```

Instance method

```
# instance Methods
class Student:
    name = "Tutor Joes"
    age = 25

    def printall(self,gender):     → used self keyword means, it is instance method and directliy call by object
        print("Name : ", Student.name)
        print("Age  : ", Student.age)
        print("Gender  : ", gender)

o=Student()
"""
o.printall()
Student.printall(o)
"""
o.printall("Male")
Student.printall(o,"Male")
```

Output

```
Name :  Tutor Joes
Age  :  25
Gender  :  Male
Name :  Tutor Joes
Age  :  25
Gender  :  Male
```

Init Method

```
# init method in Python


class user:

    def __init__(self, name):

        print("Call When new Instance Created")

        self.name = name          →instance attribute


    def printall(self):

        print("Name : ", self.name)



o1 = user("Tutor Joes")


o1.printall()

print(o1.__dict__)

o2 = user("Joes")

o2.printall()

print(o2.__dict__)

print(user.__dict__)
```

Output

```
Call When new Instance Created

Name :  Tutor Joes

{'name': 'Tutor Joes'}

Call When new Instance Created

Name :  Joes

{'name': 'Joes'}

{'__module__': '__main__', '__init__': <function user.__init__ at 0x000002485E95B5E0>, 'printall':
<function user.printall at 0x000002485E95B670>, '__dict__': <attribute '__dict__' of 'user'
objects>, '__weakref__': <attribute '__weakref__' of 'user' objects> '__doc__': None}
```

Property Decorator

```
# Property Decorator


class user:

    def __init__(self, name, age):

        self.name = name
```

```
        self.age = age
# self.msg = self.name + " is " + str(self.age) + " years old"


    @property
    def msg(self):
        return self.name + " is " + str(self.age) + " years old"



o = user("Tutor Joes", 25)
print(o.name)
print(o.age)
print(o.msg)
o.age = 45
print(o.msg)
```

Output

```
Tutor Joes
25
Tutor Joes is 25 years old
Tutor Joes is 45 years old
```

Property Decorator Getter Setter

In Python, property decorators are used to define getter, setter, and deleter methods for class properties. They allow for the encapsulation of data, by controlling access to the underlying data. Property decorators are applied to methods and define how a property value can be retrieved, set, or deleted.

```
# Property Decorators Getter Setter


class student:
    def __init__(self, total):
        self._total = total


    def average(self):
        return self._total / 5.0


    @property
    def total(self):
        return self._total
```

```
    @total.setter
    def total(self, t):
        if t < 0 or t > 500:
            print("Invalid Total and can't Change")
        else:
            self._total = t



o = student(450)
print("Total   : ", o.total)
print("Average : ", o.average())
o.total = 550
print("Total   : ", o.total)
print("Average : ", o.average())
```

Output

```
Total   :  450
Average :  90.0
Invalid Total and can't Change
Total   :  450
Average :  90.0
```

Property Method

```
# Property Method


class student:
    def __init__(self, total):
        self._total = total

    def average(self):
        return self._total / 5.0

    def getter(self):
        return self._total

    def setter(self, t):
        if t < 0 or t > 500:
```

```
            print("Invalid Total and can't Change")
        else:
            self._total = t

    total = property(getter, setter)



o = student(450)
print("Total   : ", o.total)
print("Average : ", o.average())
o.total = 350
print("Total   : ", o.total)
print("Average : ", o.average())
```

Output

```
Total   :  450
Average :  90.0
Total   :  350
Average :  70.0
```

Class Method Decorator

```
class student:
    count = 0

    def __init__(self, name, age):
        self.name = name
        self.age = age
        student.count += 1

    def printDetail(self):
        print("Name  : ", self.name, "  Age : ", self.age)

    @classmethod
    def total(cls):
        return cls.count



o = student("Joes", 25)
```

```
o.printDetail()
a = student("Raja", 45)
a.printDetail()

print("Total Admission :", student.total())
print("Total Admission :", o.total())
```

Output

```
Name  :  Joes   Age :  25
Name  :  Raja   Age :  45
Total Admission : 2
Total Admission : 2
```

Static Method

```
# Static Method in Python

class student:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def printDetail(self):
        print("Name : ", self.name, "  Age : ", self.age)

    @staticmethod
    def welcome():
        print("Welcome to our Institution")


s1 = student("Joes", 25)
s1.printDetail()
s1.welcome()


s2 = student("Raja", 45)
s2.printDetail()
s2.welcome()
```

Output

```
Name :  Joes   Age :  25
Welcome to our Institution
Name :  Raja   Age :  45
Welcome to our Institution
```

Abstraction and Encapsulation

Data abstraction

- Data abstraction refers to providing only essential information to the outside world hiding their background details
- To present the needed information in program without presenting the details

Data encapsulation

- Encapsulation is a process of wrapping code and data together into a single unit

```
# Abstraction and Encapsulation in Python


class Library:
    def __init__(self, books):
        self.books = books

    def list_books(self):
        print("Available Books")
        for book in self.books:
            print(book)

    def borrow_book(self, borrow_book):
        if borrow_book in self.books:
            print("Get Your Book Now")
            self.books.remove(borrow_book)
        else:
            print("Book not Available")

    def receive_book(self, receive_book):
        print("You have returned the book")
        self.books.append(receive_book)
```

```
books = ['C', 'C++', 'Java']
o = Library(books)

msg = """
    1.Display Book
    2.Borrow Book
    3.Return Book
"""
while True:
    print(msg)
    ch = int(input("Enter Your Choice : "))
    if ch == 1:
        o.list_books()
    elif ch == 2:
        book = input("Enter Book Name To Borrow : ")
        o.borrow_book(book)
    elif ch == 3:
        book = input("Enter Book Name To Return : ")
        o.receive_book(book)
    else:
        print("Thank You come again")
        quit()
```

Output

```
    1.Display Book
    2.Borrow Book
    3.Return Book

Enter Your Choice : 1
Available Books
C
C++
Java

    1.Display Book
    2.Borrow Book
```

```
    3.Return Book

Enter Your Choice : 2
Enter Book Name To Borrow : C
Get Your Book Now

    1.Display Book
    2.Borrow Book
    3.Return Book

Enter Your Choice : 1
Available Books
C++
Java

    1.Display Book
    2.Borrow Book
    3.Return Book

Enter Your Choice : 3
Enter Book Name To Return : Python
You have returned the book

    1.Display Book
    2.Borrow Book
    3.Return Book

Enter Your Choice : 1
Available Books
C++
Java
Python

    1.Display Book
    2.Borrow Book
    3.Return Book
```

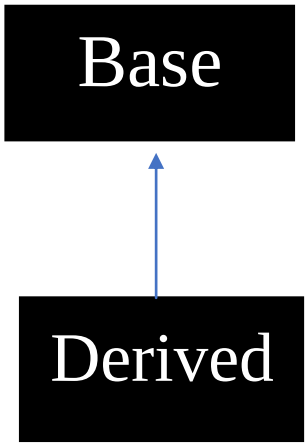
```
Enter Your Choice : 4
Thank You come again
```

Inheritance

Inheritance is a process in which one object acquires all the properties and behavior of its parent object automatically

Single inheritance

It is defined as the inheritance in which a derived class is inherited from the only one base class



Syntax :

class base1 :
body of base class
class derived(base1) :
body of derived class

```
class Nokia:
    company = "Nokia India"
    webiste = "www.nokia-india.com"

    def contact_details(self):
        print("Address : Cherry Road,Near Bus Stand ,Salem")



class Nokia1100(Nokia):
    def __init__(self):
        self.name = "Nokia 1100"
        self.year = 1998

    def product_details(self):
        print("Name     : ", self.name)
        print("Year     : ", self.year)
        print("Company  : ", self.company)
        print("Website  : ", self.webiste)
```

```
mobile = Nokia1100()
mobile.product_details()
mobile.contact_details()
```

Output

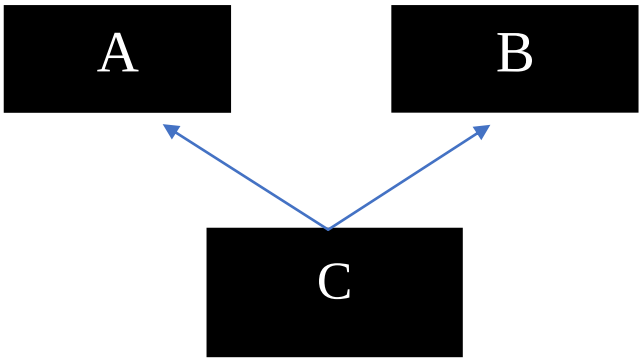
```
Name     :  Nokia 1100
Year     :  1998
Company  :  Nokia India
Website  :  www.nokia-india.com
Address : Cherry Road,Near Bus Stand ,Salem
```

Multiple inheritance

Multiple inheritance is a future of object oriented concept, where a class can inherit properties of more than one parent class

Syntax :

class Parent1 :
 # attributes and methods of Parent1
class Parent2 :
 # attributes and methods of Parent2
class Child(Parent1, Parent2) :
 # attributes and methods of Child




```
class Father:
    def fishing(self):
        print("Fishing in Rivers")

    def chess(self):
        print("Playing Chess From Father")


class Mother:
    def cooking(self):
        print("Cooking Food")

    def chess(self):
        print("Playing Chess From Mother")
```

```
class Son(Mother,Father):
    def ride(self):
        print("Riding Bicycle")
o = Son()
o.ride()
o.fishing()
o.cooking()
o.chess()
```

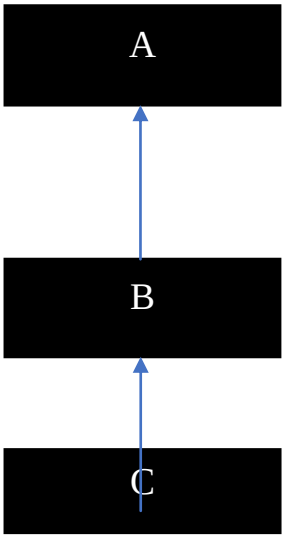
Output

```
Riding Bicycle
Fishing in Rivers
Cooking Food
Playing Chess From Mother
```

Multi level inheritance

Syntax :

class base1 :
body of base class
class derived1(base1) :
body of derived class
class derived2(derived1) :
body of derived class




```
# Multilevel Inheritance

class GrandFather:
    def ownHouse(self):
        print("Grandpa House")


class Father(GrandFather):
    def ownBike(self):
        print("Father's Bike")
```

```
class Son(Father):
    def ownBook(self):
        print("Son Have a Book")



o = Son()
o.ownHouse()
o.ownBike()
o.ownBook()
```

Output

Grandpa House

```
Father's Bike
Son Have a Book
```

Function Overriding

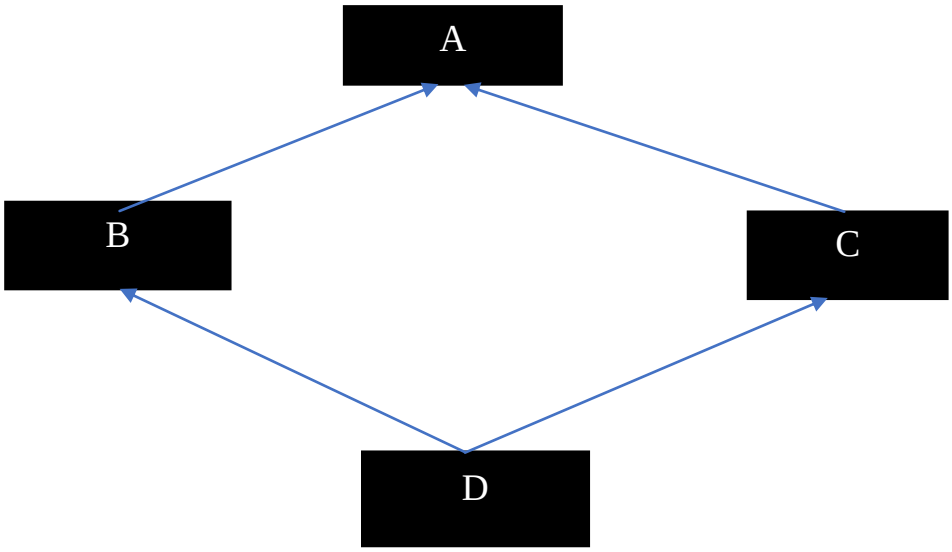
```
# Function Overriding
class Employee:
    def WorkingHrs(self):
        self.hrs = 50

    def printHrs(self):
        print("Total Working Hrs : ", self.hrs)



class Trainee(Employee):
    def WorkingHrs(self):
        self.hrs = 60

    def resetHrs(self):
        super().WorkingHrs()



employee = Employee()
employee.WorkingHrs()
employee.printHrs()
```

```
trainee=Trainee()
trainee.WorkingHrs()
trainee.printHrs()
# Reset Trainee Hrs
trainee.resetHrs()
trainee.printHrs()
```

Output

```
Total Working Hrs :  50
Total Working Hrs :  60
Total Working Hrs :  50
```

Handling Diamond Problem in Python




```
class A:
    def display(self):
        print("I am the display of Class A")


class B(A):
    def display(self):
        print("I am the display of Class B")


class C(A):
    def display(self):
```

```
        print("I am the display of Class C")


class D(B, C):
    def display(self):
        print("I am the display of Class D")


o = D()
o.display()
```

Output

```
I am the display of Class D
```

Operator Overloading

```
"""
a = 10
b = 20
print(a + b)

a = "Tutor"
b = "Joes"
print(a + b)

"""


class Addition:
    def __init__(self, a):
        self.a = a

    def __add__(o1, o2):
        return o1.a + o2.a

    def __sub__(o1, o2):
        return o1.a - o2.a
```

```
o1 = Addition(10)
o2 = Addition(20)

print("Total      : ", (o1 + o2))
print("Difference : ", (o1 - o2))

"""
Operator       Magic Method
+       __add__(self, other)
-       __sub__(self, other)
*       __mul__(self, other)
/       __truediv__(self, other)
//      __floordiv__(self, other)
%       __mod__(self, other)
**      __pow__(self, other)
>>      __rshift__(self, other)
<<      __lshift__(self, other)
&       __and__(self, other)
|       __or__(self, other)
^       __xor__(self, other)

Comparison Operators :
Operator       Magic Method
<       __LT__(SELF, OTHER)
>       __GT__(SELF, OTHER)
<=      __LE__(SELF, OTHER)
>=      __GE__(SELF, OTHER)
==      __EQ__(SELF, OTHER)
!=      __NE__(SELF, OTHER)

Assignment Operators :
Operator       Magic Method
-=      __ISUB__(SELF, OTHER)
+=      __IADD__(SELF, OTHER)
*=      __IMUL__(SELF, OTHER)
/=      __IDIV__(SELF, OTHER)
```

```
//=     __IFLOORDIV__(SELF, OTHER)
%=      __IMOD__(SELF, OTHER)
**=     __IPOW__(SELF, OTHER)
>>=     __IRSHIFT__(SELF, OTHER)
<<=     __ILSHIFT__(SELF, OTHER)
&=      __IAND__(SELF, OTHER)
|=      __IOR__(SELF, OTHER)
^=      __IXOR__(SELF, OTHER)

Unary Operators :
Operator       Magic Method
-       __NEG__(SELF, OTHER)
+       __POS__(SELF, OTHER)
~       __INVERT__(SELF, OTHER)

"""
```

Output

```
Total      :  30
Difference :  -10
```

Abstract Base Class

```
from abc import ABC, abstractmethod



class Bank(ABC):
    @abstractmethod
    def loan(self): pass

    @abstractmethod
    def credit(self): pass

    @abstractmethod
    def debit(self): pass

class HDFC(Bank):
    def loan(self):
        print("We can Provide 7.5% Interest Loan")
```

```
    def credit(self):
        print("HDFC Provide Credit")

    def debit(self):
        print("HDFC Provide Debit")

    def card(self):
        print("HDFC Provide Credit Card")
o=HDFC()
o.loan()
o.credit()
o.debit()
o.card()
```

Output

```
We can Provide 7.5% Interest Loan
HDFC Provide Credit
HDFC Provide Debit
HDFC Provide Credit Card
```

File HANDLING

File handling is an important part of any web application. Python has several functions for creating, reading, updating, and deleting files

Open a File

```
 try:
    f=open("ram.txt",'w')
#f=open("data.txt",'a')
#f=open("data.txt",'r')
#print(f.read())
#print(f.readline())
    #print(f.readlines())
    """
    for line in f:
```

```
        print(line)
    """
    f.write("\nThis is New Line")
except FileNotFoundError:
    print("File not Found")
else:
    print("Thank You")
    f.close()
```

Output

```
This is a New Line
Thank You
```

Delete a file

```
import os


if os.path.exists("data.txt"):
    os.remove("data.txt")
else:
    print("File Not Found")
```