Python Classes / Object

A class is a user-defined blueprint or prototype from which objects are created. Classes provide a means of bundling data and functionality together. Creating a new class creates a new type of object, allowing new instances of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by its class) for modifying its state.

"Class creates a user-defined data structure, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object."

Some points on Python class:

- Classes are created by keyword class.
- Attributes are the variables that belong to class.
- Attributes are always public and can be accessed using dot (.) operator. Eg.: Myclass.Myattribute

Class Definition Syntax:

```
class ClassName:
    # Statement-1
     .
     .
     .
     # Statement-N
```

Example :

```
class Student:
    # A simple class
    # attribute
    attr1 = "mamal"
    attr2 = "dog"

    # A sample method
    def fun(self):
        print("I'm a", self.attr1)
        print("I'm a", self.attr2)
```

Class Object :

An Object is an instance of a Class. A class is like a blueprint while an instance is a copy of the class with actual values. It's not an idea anymore, it's an actual dog, like a dog of breed pug who's seven years old. You can have many dogs to create many different instances, but without the class as a guide, you would be lost, not knowing what information is required.

An object consists of :

State : It is represented by attributes of an object. It also reflects the properties of an object.

Behavior : It is represented by methods of an object. It also reflects the response of an object with other objects.

Identity : It gives a unique name to an object and enables one object to interact with other objects.

Declaring Objects (Also called instantiating a class)

When an object of a class is created, the class is said to be instantiated. All the instances share the attributes and the behavior of the class. But the values of those attributes, i.e. the state are unique for each object. A single class may have any number of instances.

Example:

```
class Dog:
    # A simple class
    # attribute
    attr1 = "mamal"
    attr2 = "dog"

    # A sample method
    def fun(self):
        print("I'm a", self.attr1)
        print("I'm a", self.attr2)

# Driver code
# Object instantiation
Rodger = Dog()

# Accessing class attributes
# and method through objects
print(Rodger.attr)
Rodger.fun()
```

OutPut :

```
mamal
I'm a mamal
I'm a dog
```

In the above example, an object is created which is basically a dog named Rodger. This class only has two class attributes that tell us that Rodger is a dog and a mammal.

The self

Class methods must have an extra first parameter in method definition. We do not give a value for this parameter when we call the method, Python provides it.

If we have a method which takes no arguments, then we still have to have one argument.

This is similar to this pointer in C++ and this reference in Java.

When we call a method of this object as myobject.method(arg1, arg2), this is automatically converted by Python into MyClass.method(myobject, arg1, arg2) – this is all the special self is about.

__init__ method

The __init__ method is similar to constructors in C++ and Java. Constructors are used to initialize the object's state. Like methods, a constructor also contains collection of statements(i.e. instructions) that are executed at time of Object creation. It is run as soon as an object of a class is instantiated. The method is useful to do any initialization you want to do with your object.

Example:

```
# A Sample class with init method
    class Person:

        # init method or constructor
        def __init__(self, name):
            self.name = name

        # Sample Method
        def say_hi(self):
            print('Hello, my name is', self.name)

    p = Person('Nikhil')
    p.say_hi()
```

OutPut : Hello, my name is Nikhil

Class and Instance Variables

Instance variables are for data unique to each instance and class variables are for attributes and methods shared by all instances of the class. Instance variables are variables whose value is assigned inside a constructor or method with self whereas class variables are variables whose value is assigned in the class. Defining instance varibale using constructor.

Example:

```
# Python program to show that the variables with a value
# assigned in class declaration, are class variables and
# variables inside methods and constructors are instance
# variables.

# Class for Computer Science Student
class Dog:

    # Class Variable
    animal = 'dog'

    # The init method or constructor
    def __init__(self, breed, color):

        # Instance Variable
        self.breed = breed
        self.color = color

# Objects of CSStudent class
Rodger = Dog("Pug", "brown")
Buzo = Dog("Bulldog", "black")

print('Rodger details:')
print('Rodger is a', Rodger.animal)
print('Breed: ', Rodger.breed)
print('Color: ', Rodger.color)

print('\nBuzo details:')
```

```
print('Buzo is a', Rodger.animal)
print('Breed: ', Buzo.breed)
print('Color: ', Buzo.color)

# Class variables can be accessed using class
# name also
print("\nAccessing class variable using class name")
print(Dog.animal)
```

OutPut:

```
Rodger details:
Rodger is a dog
Breed:    Pug
Color:    brown

Buzo details:
Buzo is a dog
Breed:    Bulldog
Color:    black

Accessing class variable using class name
dog
```

Defining instance variable using normal method.

Example:

```
# Python program to show that we can create
# instance variables inside methods

# Class for Computer Science Student
class Dog:

    # Class Variable
    animal = 'dog'
    # The init method or constructor
    def __init__(self, breed):
        # Instance Variable
        self.breed = breed

    # Adds an instance variable
    def setColor(self, color):
        self.color = color

    # Retrieves instance variable
    def getColor(self):
        return self.color

# Driver Code
Rodger = Dog("pug")
Rodger.setColor("brown")
print(Rodger.getColor())
```

OutPut: brown

Method Overloading In Python:

Like other languages (for example method overloading in C++) do, python does not supports method overloading. We may overload the methods but can only use the latest defined method.

```
# First product method.
# Takes two argument and print their
# product
def product(a, b):
    p = a * b
    print(p)

# Second product method
# Takes three argument and print their
# product
def product(a, b, c):
    p = a * b*c
    print(p)

# Uncommenting the below line shows an error
# product(4, 5)

# This line will call the second product method
product(4, 5, 5)
```

OutPut : 100

In the above code we have defined two product method, but we can only use the second product method, as python does not supports method overloading. We may define many method of same name and different argument but we can only use the latest defined method. Calling the other method will produce an error. Like here calling **product(4,5)** will produce an error as the latest defined product method takes three arguments.

However we may use other implementation in python to make the same function work differently i.e. as per the arguments.

Example:

```
# Function to take multiple arguments
def add(datatype, *args):
    # if datatype is int
    # initialize answer as 0
    if datatype =='int':
        answer = 0

    # if datatype is str
    # initialize answer as ''
    if datatype =='str':
        answer =''

    # Traverse through the arguments
    for x in args:

        # This will do addition if the
```

```
        # arguments are int. Or concatenation
        # if the arguments are str
        answer = answer + x
    print(answer)
# Integer
add('int', 5, 6)
# String
add('str', 'Hi ', 'Geeks')
```

Output:

```
11
Hi Geeks
```

Operator Overloading in Python

Operator Overloading means giving extended meaning beyond their predefined operational meaning. For example operator + is used to add two integers as well as join two strings and merge two lists. It is achievable because '+' operator is overloaded by int class and str class. You might have noticed that the same built-in operator or function shows different behavior for objects of different classes, this is called Operator Overloading.

Example:

```
# Python program to show use of
# + operator for different purposes.
print(1 + 2)
# concatenate two strings
print("ABC"+"For")
# Product two numbers
print(3 * 4)
# Repeat the String
print("ABC"*4)
```

Output:

```
3
ABCFor
12
ABCABCABCABC
```

How to overload the operators in Python?

Consider that we have two objects which are a physical representation of a class (user-defined data type) and we have to add two objects with binary '+' operator it throws an error, because compiler don't know how to add two objects. So we define a method for an operator and that process is called operator overloading. We can overload all existing operators but we can't create a new operator. To perform operator overloading, Python provides some special function or magic function that is automatically invoked when it is associated with that particular operator. For example, when we use + operator, the magic method __add__ is automatically invoked in which the operation for + operator is defined.

Overloading binary + operator in Python :

When we use an operator on user defined data types then automatically a special function or magic function associated with that operator is invoked. Changing the behavior of operator is as simple as changing the behavior of method or function. You define methods in your class and operators work according to that behavior defined in methods. When we use + operator, the magic method __add__ is automatically invoked in which the operation for + operator is defined. There by changing this magic

method's code, we can give extra meaning to the + operator.

Code 1:

```
# Python Program illustrate how
# to overload an binary + operator
class A:
   def __init__(self, a):
      self.a = a

   # adding two objects
   def __add__(self, o):
      return self.a + o.a
ob1 = A(1)
ob2 = A(2)
ob3 = A("ABC")
ob4 = A(" For")

print(ob1 + ob2)
print(ob3 + ob4)
```

OutPut :

```
3
ABC For
```

Code 2 :

```
# Python Program to perform addition
# of two complex numbers using binary
# + operator overloading.
class complex:
   def __init__(self, a, b):
      self.a = a
      self.b = b

    # adding two objects
   def __add__(self, other):
      return self.a + other.a, self.b + other.b

   def __str__(self):
      return self.a, self.b

Ob1 = complex(1, 2)
Ob2 = complex(2, 3)
Ob3 = Ob1 + Ob2
print(Ob3)
```

OutPut:

```
(3,5)
```

Overloading comparison operators in Python :

Example:

```
# Python program to overload
# a comparison operators

class A:
```

```
def __init__(self, a):
  self.a = a
 def __gt__(self, other):
   if(self.a>other.a):
      return True
  else:
     return False
ob1 = A(2)
ob2 = A(3)
if(ob1>ob2):
      print("ob1 is greater than ob2")
else:
      print("ob2 is greater than ob1")
```

OutPut :

```
ob2 is greater than ob1
```

Overloading equality and less than operators :

Example: # Python program to overload equality

```
# and less than operators

class A:
   def __init__(self, a):
      self.a = a
   def __lt__(self, other):
      if(self.a<other.a):
         return "ob1 is lessthan ob2"
      else:
         return "ob2 is less than ob1"
   def __eq__(self, other):
      if(self.a == other.a):
         return "Both are equal"
      else:
         return "Not equal"

ob1 = A(2)
ob2 = A(3)
print(ob1 < ob2)

ob3 = A(4)
ob4 = A(4)
print(ob1 == ob2)
```

Output :

```
ob1 is lessthan ob2
Not equal
```

Python magic methods or special functions for operator overloading

Binary Operators:

OPERATOR	MAGIC METHOD
+	__add__(self, other)
–	__sub__(self, other)
*	__mul__(self, other)
/	__truediv__(self, other)
//	__floordiv__(self, other)
%	__mod__(self, other)
**	__pow__(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)

Unary Operators :

OPERATOR	MAGIC METHOD
–	__neg__(self, other)
+	__pos__(self, other)

~	__invert__(self, other)

Python Inheritance :

Inheritance allows us to define a class that inherits all the methods and properties from another class.
Parent class is the class being inherited from, also called base class.
Child class is the class that inherits from another class, also called derived class.

Create a Parent Class

Any class can be a parent class, so the syntax is the same as creating any other class:

Example
Create a class named Person, with firstname and lastname properties, and a printname method:

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
     self.lastname = lname

  def printname(self):
     print(self.firstname, self.lastname)

#Use the Person class to create an object, and then execute the printname method:
x = Person("John", "Doe")
x.printname()
```

OutPut: John Doe

Create a Child Class

To create a class that inherits the functionality from another class, send the parent class as a parameter when creating the child class:
Example
Create a class named Student, which will inherit the properties and methods from the Person class:

```
class Student(Person):
    pass
```

Note: Use the `pass` keyword when you do not want to add any other properties or methods to the class.

Now the Student class has the same properties and methods as the Person class.

```
x = Student("Mike", "Olsen")
x.printname()
```

OutPut : Mike Olsen

Add the __init__() Function

So far we have created a child class that inherits the properties and methods from its parent.
We want to add the __init__() function to the child class (instead of the pass keyword).
Note: The __init__() function is called automatically every time the class is being used to create a new object.

Example

Add the __init__() function to the Student class:

```
class Student(Person):
    def __init__(self, fname, lname):
    #add properties etc.
```

When you add the __init__() function, the child class will no longer inherit the parent's __init__() function.

Note: The child's __init__() function overrides the inheritance of the parent's __init__() function.

To keep the inheritance of the parent's __init__() function, add a call to the parent's __init__() function:

Example:

```
class Student(Person):
  def __init__(self, fname, lname):
    Person.__init__(self, fname, lname)
```

Now we have successfully added the __init__() function, and kept the inheritance of the parent class, and we are ready to add functionality in the `__init__()` function.

Use the super() Function

Python also has a super() function that will make the child class inherit all the methods and properties from its parent:

Example:

```
class Student(Person):
  def __init__(self, fname, lname):
    super().__init__(fname, lname)
```

By using the `super()` function, you do not have to use the name of the parent element, it will automatically inherit the methods and properties from its parent.

Add Properties:

Example : Add a property called `graduationyear` to the `Student` class:

```
class Student(Person):
    def __init__(self, fname, lname):
        super().__init__(fname, lname)
        self.graduationyear = 2020
OutPut : 2020
```

In the example below, the year `2019` should be a variable, and passed into the `Student` class when creating student objects. To do so, add another parameter in the __init__() function:

Example

Add a year parameter, and pass the correct year when creating objects:

```
class Student(Person):
    def __init__(self, fname, lname, year):
        super().__init__(fname, lname)
        self.graduationyear = year

x = Student("Mike", "Olsen", 2020)
```

Add Methods:

Example

Add a method called welcome to the Student class:

```
class Student(Person):
    def __init__(self, fname, lname, year):
        super().__init__(fname, lname)
        self.graduationyear = year

def welcome(self):
  print("Welcome", self.firstname, self.lastname, "to the class of", self.graduationyear)
```

OutPut:

Welcome Mike Olsen to the class of 2020

If you add a method in the child class with the same name as a function in the parent class, the inheritance of the parent method will be overridden.

Class or Static Variables in Python

Class or static variables are shared by all objects. Instance or non-static variables are different for different objects (every object has a copy of it).

For example, let a Computer Science Student be represented by class CSStudent. The class may have a static variable whose value is “cse” for all objects. And class may also have non-static members like name and roll.

The Python approach is simple, it doesn’t require a static keyword. All variables which are assigned a value in class declaration are class variables. And variables which are assigned values inside class methods are instance variables.

Example:

```
# Python program to show that the variables with a value
# assigned in class declaration, are class variables
# Class for Computer Science Student

class CSStudent:
   stream = 'cse'                # Class Variable
   def __init__(self,name,roll):
      self.name = name           # Instance Variable
      self.roll = roll           # Instance Variable

# Objects of CSStudent class
a = CSStudent('ABC', 1)
b = CSStudent('Nerd', 2)
print(a.stream)   # prints "cse"
print(b.stream)   # prints "cse"
print(a.name)     # prints "ABC"
print(b.name)     # prints "Nerd"
print(a.roll)     # prints "1"
print(b.roll)     # prints "2"

# Class variables can be accessed using class
# name also
print(CSStudent.stream) # prints "cse"
```

Output:

```
cse
cse
ABC
Nerd
1
2
cse
```

Class Method

The @classmethod decorator, is a builtin function decorator that is an expression that gets evaluated after your function is defined. The result of that evaluation shadows your function definition. A class method receives the class as implicit first argument, just like an instance method receives the instance

Syntax:

```
class C(object):
    @classmethod
    def fun(cls, arg1, arg2, ...):
        ....
```

fun: function that needs to be converted into a class method
returns: a class method for function.

- A class method is a method which is bound to the class and not the object of the class.
- They have the access to the state of the class as it takes a class parameter that points to the class and not the object instance.
- It can modify a class state that would apply across all the instances of the class. For example it can modify a class variable that will be applicable to all the instances.

Static Method :

A static method does not receive an implicit first argument.

Syntax:

```
class C(object):
    @staticmethod
    def fun(arg1, arg2, ...):
        ...returns: a static method for function fun.
```

- A static method is also a method which is bound to the class and not the object of the class.
- A static method can't access or modify class state.
- It is present in a class because it makes sense for the method to be present in class.

Example :

```
# Python program to demonstrate
# use of class method and static method.
from datetime import date

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

  # a class method to create a Person object by birth year.
  @classmethod
```

```
    def fromBirthYear(cls, name, year):
        return cls(name, date.today().year - year)

    # a static method to check if a Person is adult or not.
    @staticmethod
    def isAdult(age):
        return age > 18

person1 = Person('mayank', 21)
person2 = Person.fromBirthYear('mayank', 1996)

print person1.age
print person2.age

# print the result
print Person.isAdult(22)
```

Output:

```
21
21
True
```