Python  Programming

Day 10: Classes and Objects (2)

Classes and Objects

Instance	Methods
Property	OOP

Color and symbol meaning

Hint

Preferred

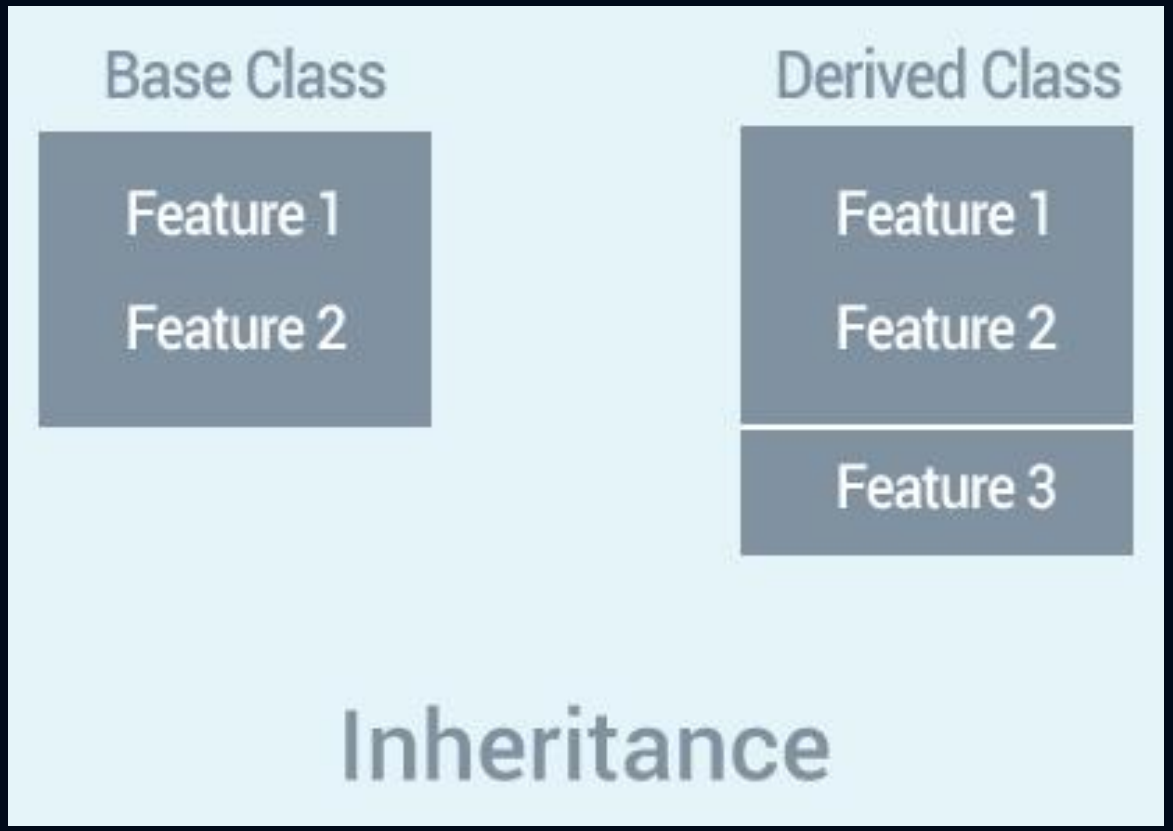
Student's activity

Practice code

	Keyword
	In-built functions
	Strings
	Output

Class Inheritance

Instead of starting from scratch, you can create a class by deriving it from a preexisting class by listing the parent class in parentheses after the new class name.



Class Inheritance

The original class is called a **base class** or a **superclass**. The new class is called a **derived class** or a **subclass**. When a class is created via inheritance, it "inherits" the attributes defined by its base classes.

However, a derived class may redefine any of these attributes and add new attributes of its own.

Class Inheritance

A child class can also **override** data members and methods from the parent.

Inheritance is often used to redefine the behaviour of existing methods.

Class Inheritance - Syntax

Derived classes are declared much like their parent class; however, a list of base classes to inherit from is given after the class name.

```
class SubClassName (ParentClass1[, ParentClass2, ...]):
  'Optional class documentation string'
  class_suite
```


Class Inheritance

Sample Code

```
class Parent:        # define parent class
   parentAttr = 100
   def __init__(self):
      print ("Calling parent constructor")

   def parentMethod(self):
      print ('Calling parent method')

   def setAttr(self, attr):
      Parent.parentAttr = attr

   def getAttr(self):
      print ("Parent attribute :", Parent.parentAttr)
```


Class Inheritance

Sample Code

```
class Child(Parent): # define child class
  def __init__(self):
    print ("Calling child constructor")

  def childMethod(self):
    print ('Calling child method')

c = Child()          # instance of child
c.childMethod()      # child calls its method
c.parentMethod()     # calls parent's method
c.setAttr(200)       # again call parent's method
c.getAttr()          # again call parent's method
```


Class Inheritance

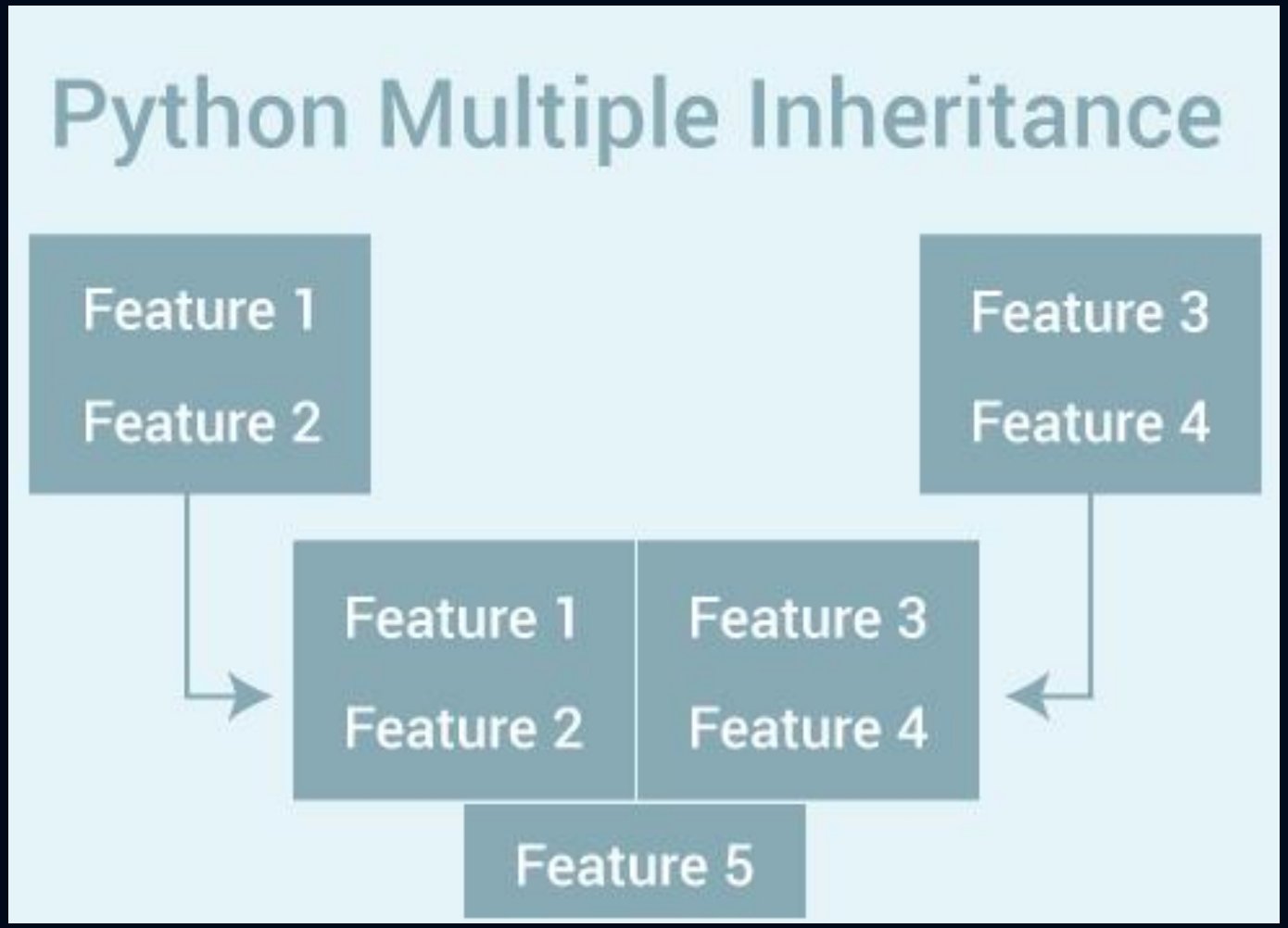
Sample Code

When the above code is executed, it produces the following result

```
Calling child constructor
Calling child method
Calling parent method
Parent attribute : 200
```


Class Inheritance - Multiple parent

Similar way, you can derive a class from **multiple parent** classes as follows



Class Inheritance - Multiple parent

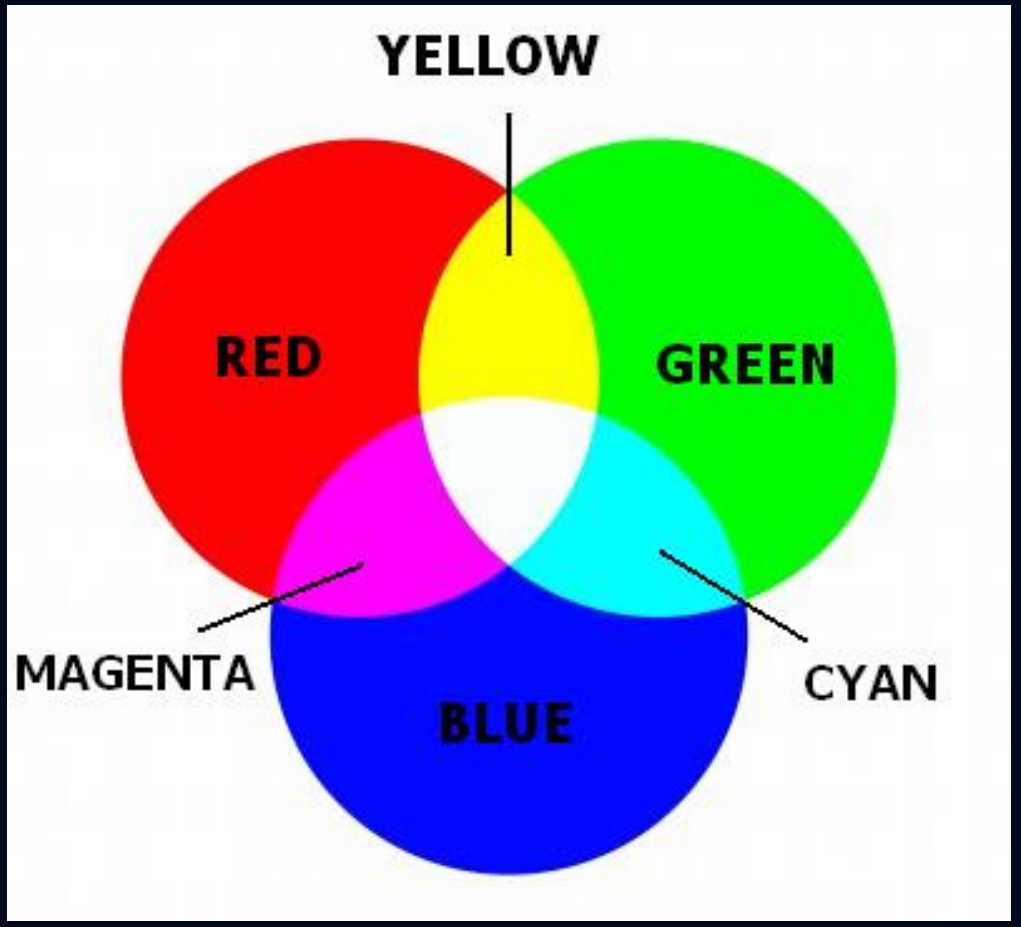
```
class A:        # define your class A
.....

class B:        # define your class B
.....

class C(A, B):   # subclass of A and B
.....
```


Class Activity 1

Create a class named ‘primaryColor’ with 3 methods that return the name of the 3 primary colors. Then create a derived class named ‘secondaryColor’ with 3 methods that combines any 2 primary colors from ‘primaryColor’ base class to return the resultant secondary color.



Class Inheritance

You can use issubclass() or isinstance() functions to check a relationships of two classes and instances.

The **issubclass**(sub, sup) boolean function returns true if the given subclass **sub** is indeed a subclass of the superclass **sup**.

The **isinstance**(obj, Class) boolean function returns true if **obj** is an instance of class **Class** or is an instance of a subclass of **Class**

Overriding Methods

Sample Code

You can always **override** your parent class methods. One reason for overriding parent's methods is because you may want special or different functionality in your subclass.

```
class Parent:       # define parent class
  def myMethod(self):
    print ('Calling parent method')

class Child(Parent): # define child class
  def myMethod(self):
    print ('Calling child method')

c = Child()         # instance of child
c.myMethod()   # child calls overridden method
```

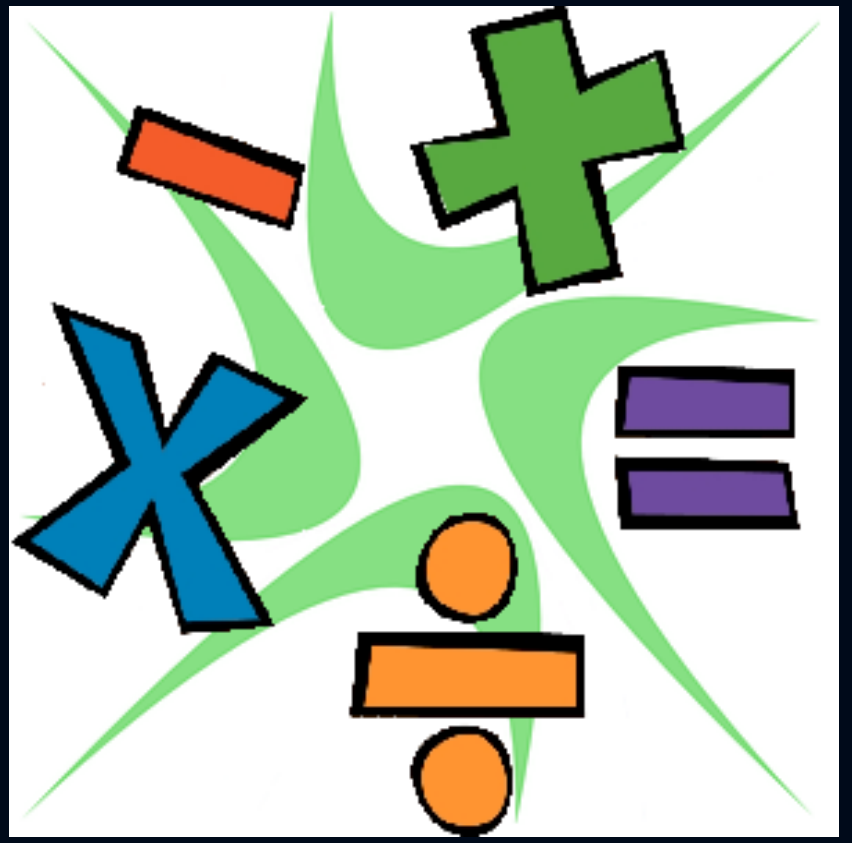

Base Overloading Methods

SN	Method, Description & Sample Call
1	**__init__ (self [,args...])** Constructor (with any optional arguments) Sample Call : *obj = className(args)*
2	**__del__(self)** Destructor, deletes an object Sample Call : *del obj*
3	**__repr__(self)** Evaluatable string representation Sample Call : *repr(obj)*
4	**__str__(self)** Printable string representation Sample Call : *str(obj)*
5	**__cmp__ (self, x)** Object comparison Sample Call : *cmp(obj, x)*

The displayed table lists some generic functionality that you can override in your own classes

Overloading Operators

Suppose you have created a Vector class to represent two-dimensional vectors, what happens when you use the plus operator to add them?



Most likely Python will yell at you.

Overloading Operators

```
class Vector:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def __str__(self):
        return 'Vector (%d, %d)' % (self.a, self.b)

    def __add__(self, other):
        return Vector(self.a + other.a, self.b + other.b)


v1 = Vector(2, 10)
v2 = Vector(5, -2)
print(v1 + v2)
```

You could, however, define the **__add__** method in your class to perform vector addition and then the plus operator would behave as per expectation.

Overloading Operators

When the above code is executed, it produces the result below

Vector(7,8)

Now Try it without the __add__ method!

Data Hiding

Sample Code

Attributes with a **double underscore prefix** will not be directly visible to outsiders.

```
class JustCounter:
    __secretCount = 0

    def count(self):
        self.__secretCount += 1
        print(self.__secretCount)


counter = JustCounter()
counter.count()
counter.count()
print(counter.__secretCount)
```


Data Hiding

Python protects those members by internally changing the name to include the class name. You can access such attributes as **object._className__attrName**.

```
1
2
Traceback (most recent call last):
  File "test.py", line 12, in <module>
    print counter.__secretCount
AttributeError: JustCounter
instance has no attribute
'__secretCount'
```


Data Hiding

If you would replace your last line in the sample code with the line of code below then it works for you

print counter._JustCounter__secretCount

Super Method

super( ) returns a special object that lets you perform attribute lookups on the base classes.

Super Method

In Python, super() built-in has two major use cases:

- Allows us to avoid using base class explicitly
- Working with Multiple Inheritance

super() with Single Inheritance

```
class Mammal(object):
    def __init__(self, mammalName):
        print(mammalName, 'is a warm-blooded animal.')


class Dog(Mammal):
    def __init__(self):
        print('Dog has four legs.')
        super().__init__('Dog')


d1 = Dog()
```


Super Method

When you run the program, the output will be:

```
Dog has four legs.
Dog is a warm-blooded animal.
```

Here, we called __init__ method of the Mammal class (from the Dog class) using code.

super().__init__('Dog')

instead of

Mammal.__init__(self, 'Dog')

Super Method

Since, we do not need to specify the name of the base class if we use super(), we can easily change the base class for Dog method easily (if we need to).

```
# changing base class to CanidaeFamily
class Dog(CanidaeFamily):
  def __init__(self):
    print('Dog has four legs.')

    # no need to change this
    super().__init__('Dog')
```


super() with Multiple Inheritance

The super() built-in returns a proxy object, a substitute object that has ability to call method of the base class via delegation.

Open a new python file and enter the following block of codes

super() with Multiple Inheritance

```
class Animal:
   def __init__(self, animalName):
      print(animalName, 'is an animal.');

class Mammal(Animal):
   def __init__(self, mammalName):
      print(mammalName, 'is a warm-blooded animal.')
      super().__init__(mammalName)

class NonWingedMammal(Mammal):
   def __init__(self, NonWingedMammalName):
      print(NonWingedMammalName, "can't fly.")
      super().__init__(NonWingedMammalName)
```


super() with Multiple Inheritance

```
class NonMarineMammal(Mammal):
    def __init__(self, NonMarineMammalName):
        print(NonMarineMammalName, "can't swim.")
        super().__init__(NonMarineMammalName)

class Dog(NonMarineMammal, NonWingedMammal):
    def __init__(self):
        print('Dog has 4 legs.');
        super().__init__('Dog')

d = Dog()
print('')
bat = NonMarineMammal('Bat')
```


super() with Multiple Inheritance

When you run the program, the output will be:

Dog has 4 legs.
Dog can't swim.
Dog can't fly.
Dog is a warm-blooded animal.
Dog is an animal.

Bat can't swim.
Bat is a warm-blooded animal.
Bat is an animal.

Next Lecture ...

Python Programming

Day 11: Standard Library Module