GOVERNMENT POLYTECHNIC AMRAVATI
DIPLOMA PROGRAMME IN COMPUTER ENGINEERING

COURSE : PROGRAMMING WITH PYTHON

COURSE CODE :CM5461

Unit 6

OOP in Python

RATIONALE

- Python is used for developing desktop GUI applications, websites and web applications. Also, as a high level programming language it allows you to focus on core functionality of the application by taking care of common programming tasks. This course is designed to help the students to understand fundamental syntactic information about ‘Python’. Also it will help the students to apply the basic concepts, program structure and principles of ‘Python’ programming paradigm to build given application. The course is basically designed to create a base to develop foundation skills of programming language.

COURSE OUTCOMES (COs)

At the end of this course, student will be able to: -

- Write and execute simple ‘Python’ programs.
- Write ‘Python’ programs using arithmetic expressions and control structure.
- Develop ‘Python’ programs using List, Tuples and Dictionary.
- Develop/Use functions in Python programs for modular programming approach.
- Develop ‘Python’ programs using File Input/output operations.
- Write ‘Python’ code using Classes and Objects.

UNIT 4 :- CONTENTS

1. Opening file in different modes.
2. Accessing file Contents using standard library functions.
3. Closing a file.
4.
5.

OOP in Python

- **Object Oriented Programming**
- Python is a multi-paradigm programming language. It supports different programming approaches.
- One of the popular approaches to solve a programming problem is by creating objects. This is known as Object-Oriented Programming (OOP).

OOP in Python

- An object has two characteristics:
- attributes
- behavior
- Let's take an example:
- A parrot is an object, as it has the following properties:
- name, age, color as attributes
- singing, dancing as behavior
- The concept of OOP in Python focuses on creating reusable code. This concept is also known as DRY (Don't Repeat Yourself).
- In Python, the concept of OOP follows some basic principles:

OOP in Python

- An object has two characteristics:
- attributes
- behavior
- Let's take an example:
- A parrot is an object, as it has the following properties:
- name, age, color as attributes
- singing, dancing as behavior
- The concept of OOP in Python focuses on creating reusable code. This concept is also known as DRY (Don't Repeat Yourself).
- In Python, the concept of OOP follows some basic principles:

Class

- A class is a blueprint for the object.
- We can think of class as a sketch of a parrot with labels. It contains all the details about the name, colors, size etc. Based on these descriptions, we can study about the parrot. Here, a parrot is an object.
- The example for class of parrot can be :
- class Parrot: passHere, we use the class keyword to define an empty class Parrot. From class, we construct instances. An instance is a specific object created from a particular class.

Object

- An object (instance) is an instantiation of a class. When class is defined, only the description for the object is defined. Therefore, no memory or storage is allocated.
- The example for object of parrot class can be:
- obj = Parrot() Here, obj is an object of class Parrot.
- Suppose we have details of parrots. Now, we are going to show how to build the class and objects of parrots.

Object

```
class Parrot:
 # class attribute species = "bird“
# instance attribute
def __init__(self, name, age):
self.name = name
self.age = age
# instantiate the Parrot class
blu = Parrot("Blu", 10)
woo = Parrot("Woo", 15)
```

Object

```
print("Blu is a {}".format(blu.__class__.species))

print("Woo is also a {}".format(woo.__class__.species))

 # access the instance attributes

print("{} is {} years old".format( blu.name, blu.age))

print("{} is {} years old".format( woo.name, woo.age))
```

Blu is a bird

Woo is also a bird

Blu is 10 years old

Woo is 15 years old

Object

- In the above program, we created a class with the name Parrot. Then, we define attributes. The attributes are a characteristic of an object.
- These attributes are defined inside the __init__ method of the class. It is the initializer method that is first run as soon as the object is created.

Object

- Then, we create instances of the Parrot class. Here, blu and woo are references (value) to our new objects.
- We can access the class attribute using __class__.species. Class attributes are the same for all instances of a class. Similarly, we access the instance attributes using blu.name and blu.age. However, instance attributes are different for every instance of a class.

Methods

Methods are functions defined inside the body of a class. They are used to define the behaviors of an object.

```
class Parrot:

 # instance attributes

def __init__(self, name, age):

self.name = name

self.age = age
```

Methods

```
def sing(self, song):

 return "{} sings {}".format(self.name, song)

def dance(self):

return "{} is now dancing". format(self.name)

blu = Parrot("Blu", 10)

print(blu.sing("'Happy'"))

print(blu.dance())
```

Object Methods

Objects can also contain methods. Methods in objects are functions that belong to the object. Let us create a method in the Person class:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age
  def myfunc(self):
    print("Hello my name is " + self.name)
p1 = Person("John", 36)
p1.myfunc()
```

The self Parameter

The self parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class. It does not have to be named self you can call it whatever you like, but it has to be the first parameter of a function in the class:

```
class Person:
    def __init__(myobject, name, age):
      myobject.name = name
      myobject.age = age
    def myfunc(abc):
      print("Hello my name is " + abc.name)
p1 = Person("John", 36)
p1.myfunc()
```

Inheritance

- Inheritance is a way of creating a new class for using details of an existing class without modifying it. The newly formed class is a derived class (or child class). Similarly, the existing class is a base class (or parent class).

Inheritance

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()
```

Inheritance

Create a class named Student, which will inherit the properties and methods from the Person class:

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

Use the Student class to create an object, and then execute the printname method:

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

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).

Inheritance

Create a class named Student, which will inherit the properties and methods from the Person class:

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

Use the Student class to create an object, and then execute the printname method:

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

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).

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

Inheritance

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

```
class Student(Person):
  def __init__(self, fname, lname):
    super().__init__(fname, lname)
    self.graduationyear = 2019
```

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

x = Student("Mike", "Olsen", 2019)
```

Inheritance

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

```
class Student(Person):
  def __init__(self, fname, lname):
    super().__init__(fname, lname)
    self.graduationyear = 2019
```

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

x = Student("Mike", "Olsen", 2019)
```

Inheritance

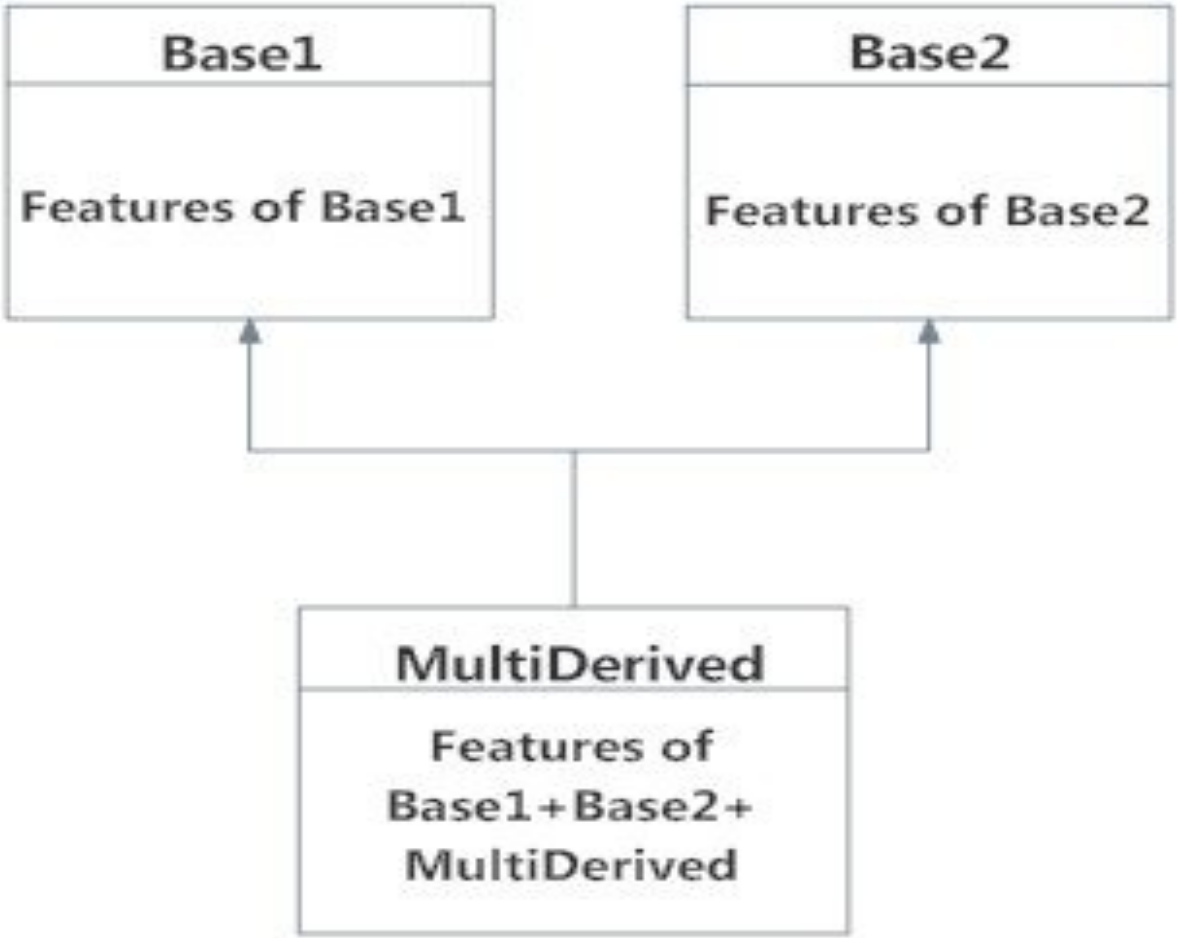
```
class Student(Person):
  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.graduationyear = year

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

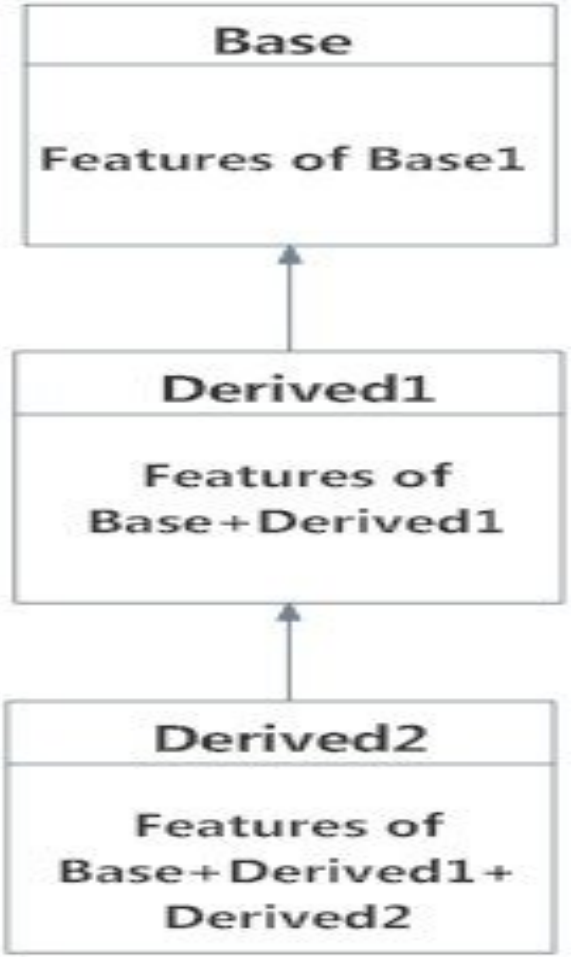
Inheritance

```
class Base1:
    pass
class Base2:
    pass
Class MultiDerived(Base1, Base2):
pass class.
```

Inheritance



Inheritance



Operator Overloading

- Python operators work for built-in classes. But the same operator behaves differently with different types. For example, the + operator will perform arithmetic addition on two numbers, merge two lists, or concatenate two strings.
- This feature in Python that allows the same operator to have different meaning according to the context is called operator overloading.
- So what happens when we use them with objects of a user-defined class? Let us consider the following class, which tries to simulate a point in 2-D coordinate system.

Operator Overloading

```
class Point:

 def __init__(self, x=0, y=0):

 self.x = x self.y = y

 p1 = Point(1, 2)

p2 = Point(2, 3)

print(p1+p2)
```

1. Here, we can see that a TypeError was raised, since Python didn't know how to add two Point objects together.
2. However, we can achieve this task in Python through operator overloading. But first, let's get a notion about special functions.