III B.Tech.

Computer Science & Engineering

CSE304: PYTHON PROGRAMMING WITH WEB FRAMEWORKS

UNIT-III: Classes and Objects

By

Mrs. S. KAMAKSHI, AP-III / CSE

School of Computing

Object Oriented Programming

- OOP groups related variables and functions into data structures called objects
- A class is a template or blueprint from which objects are created
- Attributes of a class define the type of data that an object can store
- Methods of a class provide a way to access and / or modify the attributes of the object
- Once an object is created from a class the object has an identity(ID: a unique address), a state(the data that it stores), and behavior (the methods that it contains)
- An object is an instance of a class
- All methods must take a reference to the object (self) as their first parameter

Defining Classes & Objects

- **Defining class**

```
class classname:
    def __init__(self, [v1], …):
        [self.v1 = v1    ]
        statements…
    def method(self, ….):
        statements …
    ……….
    def __str__(self):
        strrep = “Some String”
        return strrep
```

- **Creating Object**

```
obj = classname(pars)
```

Using Classes & Objects

- To reuse a class defined in another module
  - from module_name import class_name …
- To create object of a class
  - object_name = class_name([pars])
- To refer the attributes of an object
  - object_name.attribute_name = value
- To call a method of an object
  - object_name.method_name(pars)
- All objects need not have same set of attributes
  - Runtime attribute bindings are also allowed

    object_name.newattr = value
  - Dynamic method bindings are also allowed

    def fn():

        statements

    object_name.newmeth = fn

Constructors & String Representations

Constructor:

```
def __init__(self, [pars]):
    statements for initialization ..
```

String Representation:

```
def __str__(self):
    return string_value
def __repr__(self):
    return string_value
```

Example

```
class Product:
    def __init__(self, name, price, discountpercent):
        self.name = name
        self.price = price
        self.discountpercent = discountpercent

    def getDiscountAmount(self):
        return self.price * self.discountpercent / 100

    def getDiscountPrice(self):
        return self.price - self.getDiscountAmount()

    def __str__(self):
        return "Product Name: " + self.name + "\nProduct Price: " + str(self.price) +
                        "\nReduced Price: " + str(self.getDiscountPrice())
```

Example

```
p1 = Product("AAA", 1490, 20)
print(p1)
```

Output:

```
Product Name: AAA
Product Price: 1490
Reduced Price: 1192.0
```

Creating Objects with Runtime Attributes


```
class A:
    def __str__(self):
        srep = ""
        for x in self.__dict__:
            srep += "\n"+str(x)+" : " +
                        str(self.__dict__[x])
        return srep
```

```
a1 = A()
a1.name = "AAA"
a1.regno = 12345
a1.marks = [80, 95, 79]
a2 = A()
a2.name = "BBB"
a2.desig = "Manager"
a2.salary = 50000
print(a1)
print(a2)
```

Object Composition

- Placing an object as an attribute in another object
- It is a way to create complex objects from simple objects

Example

```
class person:
    def __init__(self):
        self.name = 'AKASH'
        self.db = Dob(10, 3, 2000)

    def __str__(self):
        return 'NAME: '+ self.name
            + "\nDoB: " + str(self.db)
```

```
class Dob:
    def __init__(self, dd, mm, yy):
        self.dd = dd
        self.mm = mm
        self.yy = yy
    def __str__(self):
        return '{}/{}/{}'.format(
                self.dd, self.mm, self.yy)

# creating person class object
p = person()
print(p)
```

Data Encapsulation

- To hide the data attributes of an object from other codes that uses the object
- Public attributes of an object can be accessed directly by the code that uses the object
- Private attributes can be accessed indirectly through public methods and properties
- Property is a special type of method that can be used to provide access to the hidden (private) attribute
- An annotation is a line that begins with @ symbol and used for special purposes (for getting and setting [mutator] private attribute)
- A private attribute that has only getter method is called read-only property
- A private attribute that has only setter method is called write-only property

Private Attributes

```
class A:
    def __init__(self):
        self.__x = 10
A1 = A()
print(A1.__x)
```

Output:

AttributeError: 'A' object has no attribute '__x'

Accessing Private Attributes (Not Recommended)

```
print(A1._A__x)
```

Properties and Annotations

```
class A:
    def __init__(self):
        self.__x = 0
        self.__y = 0
    def  get_x(self):
        return self.__x
    def set_x(self, a):
        self.__x = a
    def  get_y(self):
        return self.__y
    def set_y(self, b):
        self.__y = b
    x = property(get_x, set_x)
    y = property(get_y, set_y)
O1 = A()
O1.x = 45
O1.y = 20
print (O1.x, O1.y)
```

```
class A:
    def __init__(self):
        self.__x = 0
        self.__y = 0
    @property                 # getter method for x
    def x(self):
        return self.__x
    @x.setter                 # setter method for x
    def x(self, a):
        self.__x = a
    @property                 # getter method for y
    def y(self):
        return self.__y
    @y.setter                 # setter method for y
    def y(self, b):
        self.__y = b
O1 = A()
O1.x = 45
O1.y = 20
print (O1.x, O1.y)
```

Class Attributes

```
class C:
    x = 10
C1 = C()
C2 = C()
print("C's x value: ", C.x)
print("C1's x value: ", C1.x)
print ("C2's x value: ", C2.x)
```

- Output
  - C's x value: 10
  - C1's x value: 10
  - C2's x value: 10

```
C.x = 30
print("C1's x value: ", C1.x)
print ("C2's x value: ", C2.x)
```

- Output
  - C1's x value: 30
  - C2's x value: 30

```
C2.x = 40
print("C1's x value: ", C1.x)
print ("C2's x value: ", C2.x)
```

- Output
  - C1's x value: 30
  - C2's x value: 40

Inheritance

```
class A:
    def __init__(self, a):
        self.a = a
class B(A):
    def __init__(self, x, y):
        A.__init__(self, x)
        self.b = y
    def __str__(self):
        return "a= "+ str(self.a) +  "\nb = "+str(self.b)
B1=B(10, 20)
print(B1)
```

Method Overriding

```
class A:
    def mymethod(self):
        print("A's method")
class B(A):
    def mymethod(self):
        print("B's method")
class C(A):
    def mymethod(self, a):
        print("C's method")
class D(A):
    pass
O1 = A()
O2 = B()
O3 = C()
O4 = D()
```

O1.mymethod()
Output: A's method
O2.mymethod()
Output: B's method
O3.mymethod()
Output: C's method
O4.mymethod()
Output: A's method
isinstance(O1,A)
Output: True
isinstance(O2,B)
Output: True
isinstance(O2,A)
Output: True

Types of Inheritance

- Single Inheritance

  class A: pass

  class B(A): pass
- Multi-level Inheritance

  class C(B): pass
- Multiple Inheritance

  class D: pass

  class E(A,D): pass
- Diamond Inheritance

  class F(A):pass

  class G(B,F): pass