Chapter 17 - OOP

Dr. Raza Ul Mustafa Khokhar
American University
Computer Science Department - CSC-148

OOP

- Python is an object-oriented programming language. That means it provides features that support object-oriented programming (OOP).
- In Python, every value is actually an object. Whether it be a turtle, a list, or even an integer, they are all objects.

An example

An object is any entity that has attributes and behaviors. For example, a parrot is an object. It has

attributes - name, age, color, etc.

behavior - dancing, singing, etc.

Similarly, a class is a blueprint for that object.

Syntax

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

Parrot example

```
class Parrot:
    # class attribute
    name = ""
    age = 0
# create parrot1 object
parrot1 = Parrot()
parrot1.name = "Blu"
parrot1.age = 10
# create another object parrot2
parrot2 = Parrot()
parrot2.name = "Woo"
parrot2.age = 15
# access attributes
print(f"{parrot1.name} is {parrot1.age} years old")
print(f"{parrot2.name} is {parrot2.age} years old")
```

The Python __init__ Method

- The __init__ method is similar to constructors in C++ and Java.
- 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

```
class Dog:
   # class attribute
    attr1 = "mammal"
   # Instance attribute
    def __init__(self, name):
        self.name = name
 # Driver code
# Object instantiation
Rodger = Dog("Rodger")
# Accessing instance attributes
print("My name is {}".format(Rodger.name))
```

Class Dog - Methods

```
class Dog:
    # class attribute
    default_name = "ABC dog"
    default_sound = "Some sound ABC"
    # Instance attribute
    def __init__(self):
        self.name = 'Tommy'
        self.sound= 'Tommy sounds like poodle'

    def speak(self):
        print(f"My name is {self.name} and sound is {self.sound}")
obj   = Dog()
# Accessing class methods
obj.speak()
print(obj.default_name)
```

Inheritance - Parent Class

- Inheritance is the capability of one class to derive or inherit the properties from another class. The class that derives properties is called the derived class or child class and the class from which the properties are being derived is called the base class or parent class.

```
class Person():
     def __init__(self, name, idnumber):
        self.name = name
        self.idnumber = idnumber
     def display(self):
        print(self.name)
        print(self.idnumber)
    def details(self):
        print("My name is {}".format(self.name))
        print("IdNumber: {}".format(self.idnumber))
```

Child class - Employee

```
class Employee(Person):
    def __init__(self, name, idnumber, salary, post):
        self.salary = salary
        self.post = post
        # invoking the __init__ of the parent class
        Person.__init__(self, name, idnumber)
    def details(self):
        print("My name is {}".format(self.name))
        print("IdNumber: {}".format(self.idnumber))
        print("Post: {}".format(self.post))
```

Instantiating both Classes

```
main_class_obj = Person('Khokhar','852')
main_class_obj.display()
print('----')
# creation of an object variable or an instance
a = Employee('Raza', 886012, 200000, "Teacher")
# calling a function of the class Person using its instance
a.display()
a.details()
```

Method overloading

- Function overloading in Python works differently from some other programming languages like C++ or Java.
- In Python, you cannot create multiple functions with the same name but different parameter lists like you would in those languages.
- However, you can achieve similar behavior using default argument values and variable-length argument lists.

Example to get concept of overloading

```
class MathOperations:
    def add(self, a, b=None):
        if b is None:
            return a
        return a + b

# Create an instance of the MathOperations class
math_ops = MathOperations()

# Test the add() method with different numbers of arguments
print(math_ops.add(5))         # Output: 5
print(math_ops.add(5, 10))     # Output: 15
```

Polymorphism

- Polymorphism simply means having many forms
- Consider class Bird
- Two inherited classes are: Sparrow and Ostrich
- In Python, Polymorphism lets us define methods in the child class that have the same name as the methods in the parent class. In inheritance, the child class inherits the methods from the parent class. However, it is possible to modify a method in a child class that it has inherited from the parent class.

Bird class

- Important where: This is particularly useful in cases where the method inherited from the parent class doesn't quite fit the child class. It has special name, see next slide)

```
class Bird:

    def intro(self):
        print("There are many types of birds.")

    def flight(self):
        print("Most of the birds can fly but some cannot.")
```

Inherited classes But! Many forms

- This process of re-implementing a method in the child class is known as Method Overriding

```
class sparrow(Bird):

    def flight(self):
        print("Sparrows can fly.")

class ostrich(Bird):

    def flight(self):
        print("Ostriches cannot fly.")
```

Now let’s instantiate

```
obj_bird = Bird()
obj_spr = sparrow()
obj_ost = ostrich()

obj_bird.intro()
obj_bird.flight()

obj_spr.intro()
obj_spr.flight()

obj_ost.intro()
obj_ost.flight()
```

Another example - polymorphism in Python using inheritance and method overriding:

```
class Animal:
    def speak(self):
        return "Sound"
class Dog(Animal):
    def speak(self):
        return "Woof!"
class Cat(Animal):
    def speak(self):
        return "Meow!"
# Create a list of Animal objects
animals = [Animal(), Dog(), Cat()]

# Call the speak method on each object
for animal in animals:
    print(animal.speak())
```

Encapsulation

- It describes the idea of wrapping data and the methods that work on data within one unit.
- This puts restrictions on accessing variables and methods directly and can prevent the accidental modification of data.
- Those types of variables are known as **private variables.**
- **Let's see an example**

Consider a story!

- A company A has different departments, Accounts, Finance, Sales, etc
- Each responsible – object manages their departments, However!
- Issue came ? and now Finance needs all the data and information of Sale!

- Here in this case, he is not allowed to directly access all the data. He needs to contact object of Sales department to have access.

This is what encapsulation is.

Protected members - Encapsulation

- To accomplish this in Python, just follow the convention by prefixing the name of the member by a single **underscore “_”.**

```
class Base:
    def __init__(self):

        # Protected member
        self._a = 2
```

Protected members- Derived Class - Encapsulation

```
class Derived(Base):
    def __init__(self):

        # Calling constructor of
        # Base class
        Base.__init__(self)
        print("Calling protected member of base class: ",
              self._a)

        # Modify the protected variable:
        self._a = 3
        print("Calling modified protected member outside class: ",
              self._a)
```

Protected members - Instantiating – Encapsulation example

```
obj1 = Derived()

obj2 = Base()

# Calling protected member
# Can be accessed but should not be done due to convention
print("Accessing protected member of obj1: ", obj1._a)

# Accessing the protected variable outside
print("Accessing protected member of obj2: ", obj2._a)
```

Private members

- Private members are similar to protected members, the difference is that the class members declared private should neither be accessed outside the class nor by any base class.
- However, to define a private member prefix the member name with double underscore “__”.

```
class Base:
    def __init__(self):
        self.a = "Nazim"
        self.__c = "Khokhar But! Private"
```

Calling private member in base class

```
class Derived(Base):
    def __init__(self):

        # Calling constructor of
        # Base class
        Base.__init__(self)
        print("Calling private member of base class: ")
        print(self.__c)
```

Instantiating

```
# Driver code
obj1 = Base()
print(obj1.a)

# Uncommenting print(obj1.c) will
# raise an AttributeError
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_4744\3496939942.py in <module>
     17 # Driver code
     18 obj1 = Base()
---> 19 print(obj1.c)
     20
     21 # Uncommenting print(obj1.c) will

AttributeError: 'Base' object has no attribute 'c'
```

Slides & Material

- razaulmustafa.us/cs148/
- Canvas
- Email: rmustafa@american.edu