

Chapter 2

- Mrs. Apurwa A. Barve

Fundamentals of OOP

- ❖ OOP - Object Oriented Programming
- ❖ Everything revolves around class, objects and methods invoked on those objects
- ❖ **Class** - It is user defined data structure which acts as a blueprint combining data(attributes) and behaviour (methods). Class definitions, like function definitions ([def](#) statements) must be executed before they have any effect.
- ❖ **Object** - It is the real world implementation of class. One class can have multiple objects. Each object has data(state/attributes), behaviour(functionality/methods) and identity (unique id).
- ❖ **Method** - It defines the purpose of the class in which it is defined. It is possible to complete the business functionality through methods of a given class. Method is invoked by the objects of that class.

Activity 2.1

1. Define a class **City** using following syntax:

class City:

- Definition starts with the keyword **class**
- Class name begins with a **capital letter** as per naming convention in Python
- : follows the class name to mark the beginning of the class

def __init__(self, name, population):

self.name = name

self.population = population

- Built in method
- Always executed when the class is being initiated.
- Used to assign values to object properties and operations

- Assigning values to respective ATTRIBUTES of the given class.

mumbai = City("Mumbai", 1000000)

- Creating OBJECT (mumbai) of the class City.
- Mumbai object will have the values Mumbai and 1000000 for name and population attributes respectively.

Activity 2.1

2. Add a method, describeCity() to the class using the following syntax

`def describeCity(self):` 

- User defined function for business logic
- Can have the first argument as self if the function needs to access instance variables

`print(f'The name of the city is {self.name}')`

`print(f'The population of the city is {self.population}')` 

- Accessing individual instance variables using DOT operator and printing them

`mumbai = City("Mumbai",1000000)`

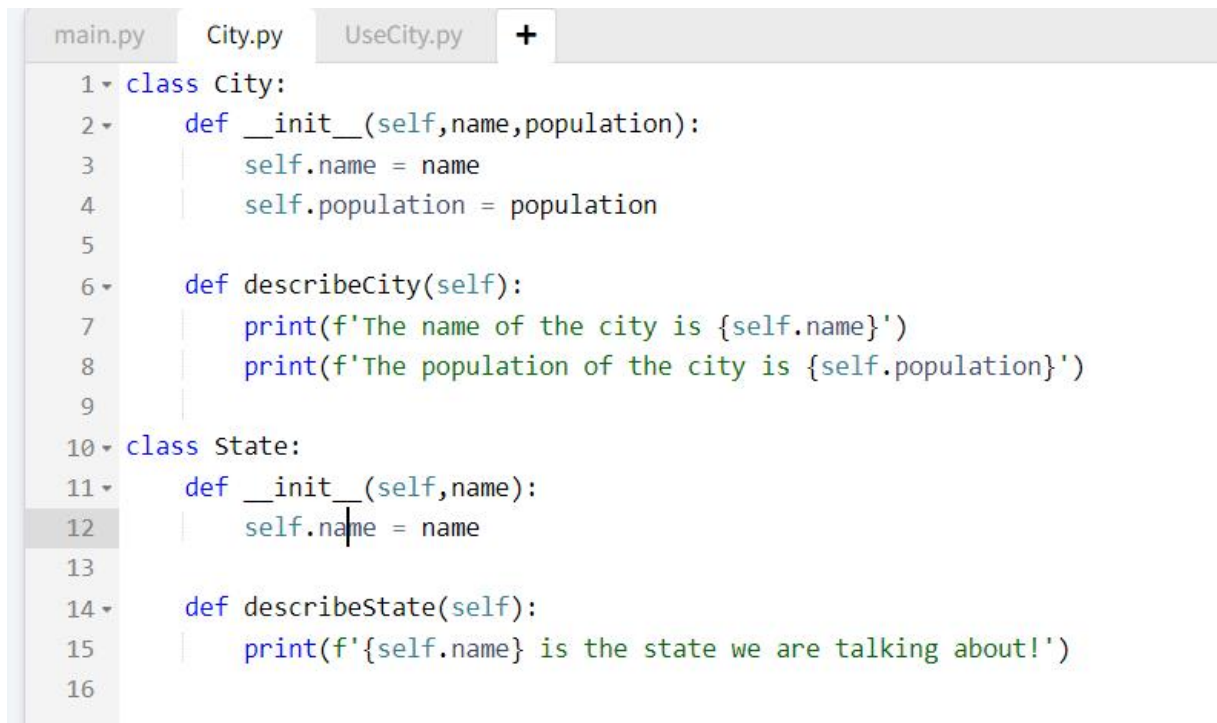
`mumbai.describeCity()` 

- Creating OBJECT (mumbai) of the class City.
- mumbai object will have the values Mumbai and 1000000 for name and population attributes respectively.
- mumbai object is used to invoke the method describeCity using DOT operator

Activity 2.1

3. To the existing file add one more class State having attribute name and method describeState()

Final code should look like:



The screenshot shows a code editor with three tabs: 'main.py', 'City.py', and 'UseCity.py'. A '+' button is visible to the right of the tabs. The 'City.py' tab is active, displaying the following Python code:

```
1 class City:
2     def __init__(self, name, population):
3         self.name = name
4         self.population = population
5
6     def describeCity(self):
7         print(f'The name of the city is {self.name}')
8         print(f'The population of the city is {self.population}')
9
10 class State:
11     def __init__(self, name):
12         self.name = name
13
14     def describeState(self):
15         print(f'{self.name} is the state we are talking about!')
16
```

Class in Python

❖ Accessing members outside the class -

- import statement allows us to use previously defined readymade classes or user defined classes in our current class / file.
- Syntax to use a class from other file:

```
from nameOfFile import nameOfClass
```

- ## ❖ Naming Convention -
- It is recommended naming convention for Python classes is the **PascalCase**, where each word is capitalized.

Activity 2.2

1. Using the syntax given in the previous slide, import classes City and State into another python file. Create their respective objects and invoke describeCity() and describeState() methods on the respective objects.

Final code should look like:

A screenshot of a code editor with three tabs: 'main.py', 'City.py', and 'UseCity.py'. The 'UseCity.py' tab is active. The code in the editor is as follows:

```
1 from City import City
2 from City import State
3
4 mumbai = City("Mumbai",1000000)
5 mumbai.describeCity()
6
7 mah = State("Maharashtra")
8 mah.describeState()
```

The cursor is at the end of line 8.

getters and setters in Python

- ❖ **Getter** is a method that fetches the value of the attribute for you.
- ❖ **Setter** is a method that assigns/set the value of the attribute for you.
- ❖ In OOP, public attributes should be used only when you're sure that such values will not impact the business logic. If there are values which on mutating/computing/modifying can alter the business log then they should be made PRIVATE and accessed through getters and setters.
- ❖ Implementing the getter and setter pattern requires:
 1. Making your attributes non-public
 2. Writing getter and setter methods for each attribute

Activity 2.3

Write a class Circle with following specifications:

1. a non-public attribute radius and a public attribute area.
2. Define the `__init__` which calls the setter for radius
3. Define the getter and setter for non-public attribute radius
4. Define a method, `calculateArea` which calculates the area of the circle using the pi constant from 'math' library. Use the getter method to fetch the radius value. Return the calculated area
5. Create an object of the Circle class and assign it a radius value. Call `calculateArea()` on this object and print its area.

Activity 2.3

The final code should look like this:

```
import math

class Circle:
    _radius = 0 #what is the access modifier here?
    area = 0.0 #what is the access modifier here?

    def __init__(self,rad):
        self.set_radius(rad)
    def set_radius(self,rd):
        self._radius = rd
        print("Radius is set")
    def get_radius(self):
        return self._radius
    def computeArea(self):
        rdtmp = self.get_radius()
        area = math.pi*rdtmp*rdtmp
        return area

myCir = Circle(2)
print(f'{myCir.computeArea()} is the area of the circle with radius {myCir.get_radius()}')
```

Inheritance in Python

- ❖ Mechanism by which we can create a hierarchy of classes sharing common properties with each other and having their own specific properties and behaviour is called **Inheritance**.
- ❖ Inheritance allows one class to derive from another class thus avoiding code redundancy and encouraging code reusability.
- ❖ Python supports MULTIPLE inheritance.
- ❖ Syntax :

```
class ParentClass:
```

Body of the parent class

```
class ChildClass(ParentClass):
```

Body of the child class

Inheritance in Python - Example

Let us understand some basic concepts of Inheritance such as **creating class hierarchies, using super in constructors and method overriding** in Python using an example of a base class, Mammal and child class, Bat.

Mammal class specification:

1. Create a Mammal class with attributes, name, lifespan and habitat.
2. Add a constructor to initialise these attributes. Put a print statement in the constructor saying, "Mammal object created".
3. Add a method, describe() which will print the individual values of these attributes.

Inheritance in Python - Example

Bat class specification:

1. Create a Bat class extending Mammal class. Add an attribute, diet, to the Bat class
2. Add a constructor to initialise all the attributes from parent class and Bat class. Learn to use `super()`
Put a print statement in the constructor saying, "Bat object created".
3. Add a method, describe() which will print the individual values of Bat class attributes.

Inheritance in Python - Example

InheritanceEx.py specification

1. Import the necessary classes in this file.
2. Create object of Mammal class and call describe() of Mammal using this object.
3. Create object of Bat class and call describe() of Bat using this object.
4. Run the program and observe the sequence of output statements to understand which methods and called and in what order.

Inheritance in Python - super

- ❖ Super keyword is used to invoke parent class constructor or method invocation from a given overridden method.
- ❖ When a child class constructor is written, first line within in should be `super().__init__()` to invoke the parent class constructor. This helps in assigning the values to the parent class attributes if any of those are used in the child class.
- ❖ Take a look at the following activity to understand the importance of super.

Activity 2.4

1. In the describe method of Bat class in the previous example, make a call to super so as to call the parent class describe() first.
2. Make the following changes to your code and observe the output.

In Bat.py - comment the call to super class `__init__`

```
from Mammal import Mammal
class Bat(Mammal):
    diet = None
    def __init__(self,nm,lf,hb,dt):
        #super().__init__(nm,lf,hb)
        self.diet = dt
        print('Bat class object is created')
    def describe(self):
        print('The diet of a bat is:',self.diet)
```


Activity 2.4

Make the following changes to your code and observe the output.

In InheritanceEx.py - print batObj.name

```
mmObj = Mammal("Whale",70,"Aquatic")
mmObj.describe()
batObj = Bat("Bat",13,"Trees","Carnivorous")
batObj.describe()
print(batObj.name)
```

Observe the sequence of output and the value printed for batObj.name. Why is the name attribute not assigned its correct value, Bat? Why does it show None?

While creating child class objects it is important to make a call to parent class constructors if we want to use the attributes from the parent class(es). Else like in the above example, the attributes do not have the values which we want.

static and class methods

- ❖ Will be covered as a part of students' presentation

private attributes and Inheritance

- ❖ Consider the following changes made to Mammal class and InheritanceEx.py.

```
name = None  
lifespan = None  
habitat = None
```

```
__sound=None
```

```
def __init__(self,name,lifespan,habitat):  
    self.name = name  
    self.lifespan = lifespan  
    self.habitat = habitat  
    self.__sound="harsh"  
    print('Mammal object created')  
def describe(self):  
    print('The details of this mammal specimen are as follows:')  
    print('Name:',self.name)  
    print('Life span :',self.lifespan)  
    print('Habitat :',self.habitat)  
    print('Sound :',self.__sound)
```

private attributes and Inheritance

```
main.py  Mammal.py  Bat.py  InheritanceEx.py  +  
1  from Mammal import Mammal  
2  from Bat import Bat  
3  
4  mmObj = Mammal("Whale",70,"Aquatic")  
5  mmObj.describe()  
6  batObj = Bat("Bat",13,"Trees","Carnivorous")  
7  #batObj.describe()  
8  print(batObj.name)  
9  print(batObj.habitat)  
10  
11  print(mmObj.__sound)  
12  print(batObj.__sound)  
13
```

private attributes and Inheritance

- ❖ Run the above example and observe the output.
- ❖ When any attribute is made private / non-public by adding single or double underscores (_ / __) in front of its name then such attribute is NOT accessible to the sub class(es) of that class.
- ❖ Private attributes are also NOT accessible from outside the class.

Thank you