Introduction to Object Oriented programming with Python

Python Classes and Objects

- Object-oriented programming is a programming paradigm that provides a means of structuring programs so that properties and behaviors are bundled into individual **objects**
- For instance, an object could represent a person with **properties** like a name, age, and address and **behaviors** such as walking, talking, breathing, and running
- It could represent an email with properties like a recipient list, subject, and body and behaviors like adding attachments and sending

Classes vs Instances

- Classes are used to create user-defined data structures. Classes define functions called **methods**, which identify the behaviors and actions that an object created from the class can perform with its data
- A class is a blueprint for how something should be defined. It doesn't actually contain any data
- While the class is the blueprint, an instance is an object that is built from a class and contains real data.

How to Define a Class

- All class definitions start with the class keyword, which is followed by the name of the class and a colon. Any code that is indented below the class definition is considered part of the class's body.
- class class_name:

  Data members

  Methods

```
Class person:
        name
         age
```

__init__()

- dunder methods: Start and end with __
- __init__() sets the initial state of the object by assigning the values of the object's properties. That is, .__init__() initializes each new instance of the class.
- You can give .__init__() any number of parameters, but the first parameter will always be a variable called self. When a new class instance is created, the instance is automatically passed to the self parameter in .__init__() so that new attributes can be defined on the object.
- class Person:
-     def __init__(self, name, age):
-         self.name = name
-         self.age = age

Cont...

- Attributes created in .__init__() are called instance attributes
- An instance attribute's value is specific to a particular instance of the class.
- All Person objects have a name and an age, but the values for the name and age attributes will vary depending on the Person instance.
- On the other hand, class attributes are attributes that have the same value for all class instances. You can define a class attribute by assigning a value to a variable name outside of .__init__().
- class Person:
-     # Class attribute
-     region = "Asia"

-     def __init__(self, name, age):
-         self.name = name
-         self.age = age

Instance Methods

- Instance methods are functions that are defined inside a class and can only be called from an instance of that class. Just like .__init__(), an instance method’s first parameter is always self.

```
class Person:
    region = “Asia”

    def __init__(self, name, age):
        self.name = name
        self.age = age

    # Instance method
    def description(self):
        return f"{self.name} is {self.age} years old"

    # Another instance method
    def speak(self, language):
        return f"{self.name} says {language}"
```

Inner Classes

- A class defined in another class is known as inner class or nested class. If an object is created using child class means inner class then the object can also be used by parent class or root class. A parent class can have one or more inner class

```
class Car:
  type = “Four wheel”

  def __init__(self, company_name, model):
    self.name = name
    self.age = age
class  engine:
    def __init__(self,eng_type,eng_mfg_year)
       self. eng_ type =eng_ type
       self.eng_mfg_year=eng_mfg_no


C1=car(“ford”,”MT2020”)
E1=c1.engine(“BS6”,”2020”)
```

Encapsulation

- Encapsulation is one of the fundamental concepts in object-oriented programming (OOP). 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. To prevent accidental change, an object's variable can only be changed by an object's method. Those types of variables are known as private variable.
- A class is an example of encapsulation as it encapsulates all the data that is member functions, variables, etc.

Cont...

- Protected members : the convention by prefixing the name of the member by a single underscore "_"
- 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. In Python, there is no existence of Private instance variables that cannot be accessed except inside a class. However, to define a private member prefix the member name with double underscore "__".

Inheritance

- Inheritance allows us to define a class that inherits all the methods and properties from another class.
- Parent class is the class being inherited from, also called base class.
- Child class is the class that inherits from another class, also called derived class.

- Person and student

Types of Inheritance

- Single Inheritance
- Multiple Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance

Overriding Super Class Constructors and Methods, super()

- Overriding is the property of a class to change the implementation of a method provided by one of its base classes.
- The super() function is used to give access to methods and properties of a parent or sibling class.
- The super() function returns an object that represents the parent class.

Method Resolution Order

- Method Resolution Order(MRO) it denotes the way a programming language resolves a method or attribute. Python supports classes inheriting from other classes. The class being inherited is called the Parent or Superclass, while the class that inherits is called the Child or Subclass.
- In python, method resolution order defines the order in which the base classes are searched when executing a method. First, the method or attribute is searched within a class and then it follows the order we specified while inheriting. This order is also called Linearization of a class and set of rules are called MRO(Method Resolution Order).

Polymorphism

- Polymorphism is taken from the Greek words Poly (many) and morphism (forms). It means that the same function name can be used for different types.

- Class person:
-             def avg_execise()
- Class cricketer:

            def avg_execise()

Duck typing philosophy of Python

- Duck Typing is a type system used in dynamic languages. For example, Python, Perl, Ruby, PHP, Javascript, etc. where the type or the class of an object is less important than the method it defines. Using Duck Typing, we do not check types at all. Instead, we check for the presence of a given method or attribute.
- “If it looks like a duck and quacks like a duck, it’s a duck”

```
class Bird:
    def fly(self):
        print("fly with wings")

class Airplane:
    def fly(self):
        print("fly with fuel")
```

```
class Fish:
    def fly(self):
        print("fish can not fly")

# Attributes having same name are
# considered as duck typing
for obj in Bird(), Airplane(), Fish():
    obj.fly()
```

Operator Overloading

- Operator Overloading means giving extended meaning beyond their predefined operational meaning. For example operator + is used to add two integers as well as join two strings and merge two lists. It is achievable because '+' operator is overloaded by int class and str class. You might have noticed that the same built-in operator or function shows different behavior for objects of different classes, this is called Operator Overloading.

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

     # adding two objects
    def __add__(self, other):
        return self.a + other.a, self.b + other.b
```

```
    def __str__(self):
        return self.a, self.b

Ob1 = complex(1, 2)
Ob2 = complex(2, 3)
Ob3 = Ob1 + Ob2
print(Ob3)
```

Python magic methods or special functions for operator overloading

Operator	Magic Method
+	__add__(self, other)
–	__sub__(self, other)
*	__mul__(self, other)
/	__truediv__(self, other)
//	__floordiv__(self, other)
%	__mod__(self, other)
**	__pow__(self, other)
>>	__rshift__(self, other)
<<	__lshift__(self, other)
&	__and__(self, other)
|	__or__(self, other)
^	__xor__(self, other)

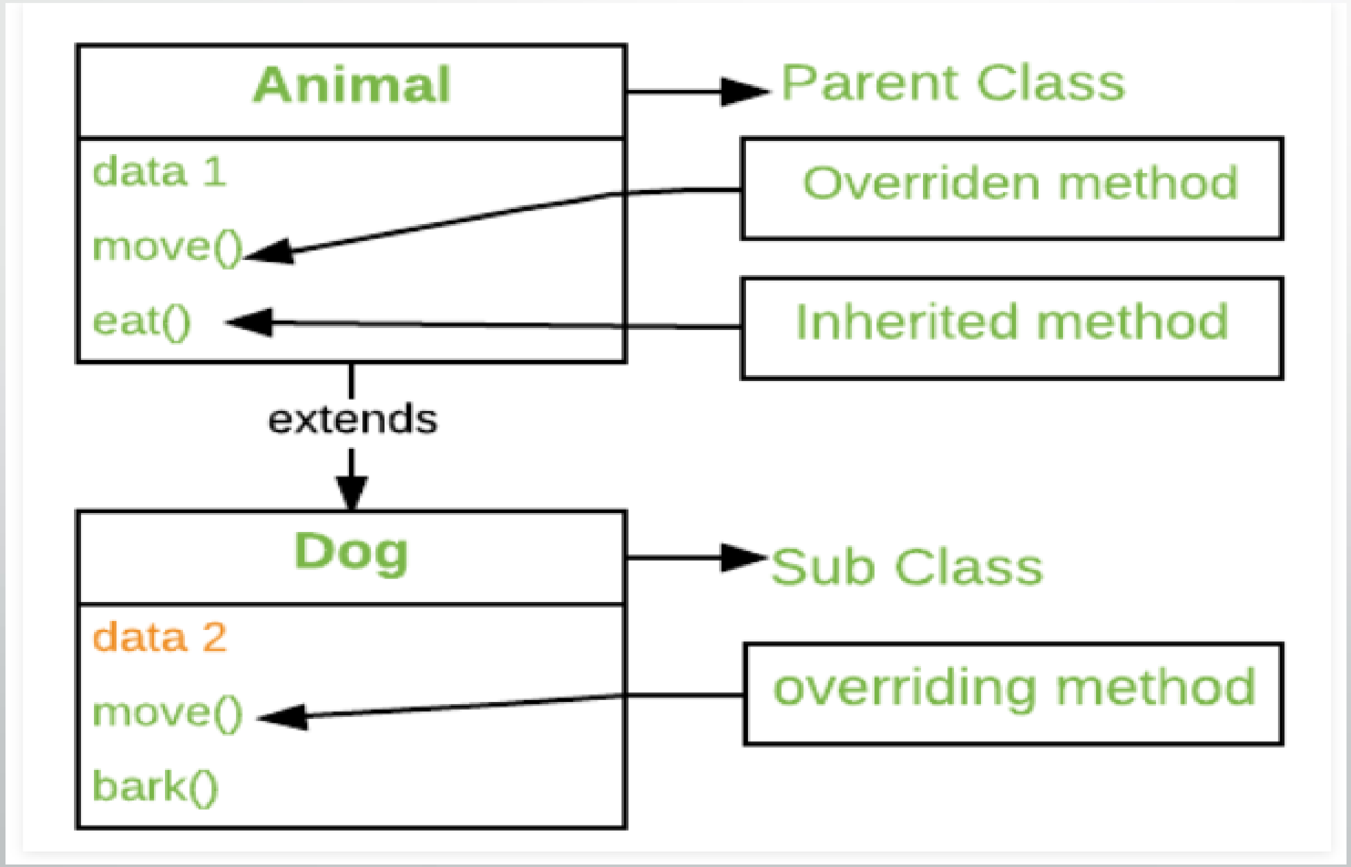
Method Overloading

- Python does not support method overloading by default. But there are different ways to achieve method overloading in Python.

```
class sumNum:
    def sum(self, a=None, b=None, c=None):
        if a!=None and b!=None and c!=None:
            print("Sum of three number :", a+b+c)
        elif a!=None and b!=None:
            print("Sum of two number :", a+b)
        else:
            print("Two or three arguments are expected")
```

Method Overriding

- Method overriding is an ability of any object-oriented programming language that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes.



Calling the Parent’s method within the overridden method

- Using Classname : Parent’s class methods can be called by using the Parent classname.method inside the overridden method
- Using Super() : Python super() function provides us the facility to refer to the parent class explicitly. It is basically useful where we have to call superclass functions. It returns the proxy object that allows us to refer parent class by ‘super’

Method Overriding Cont…

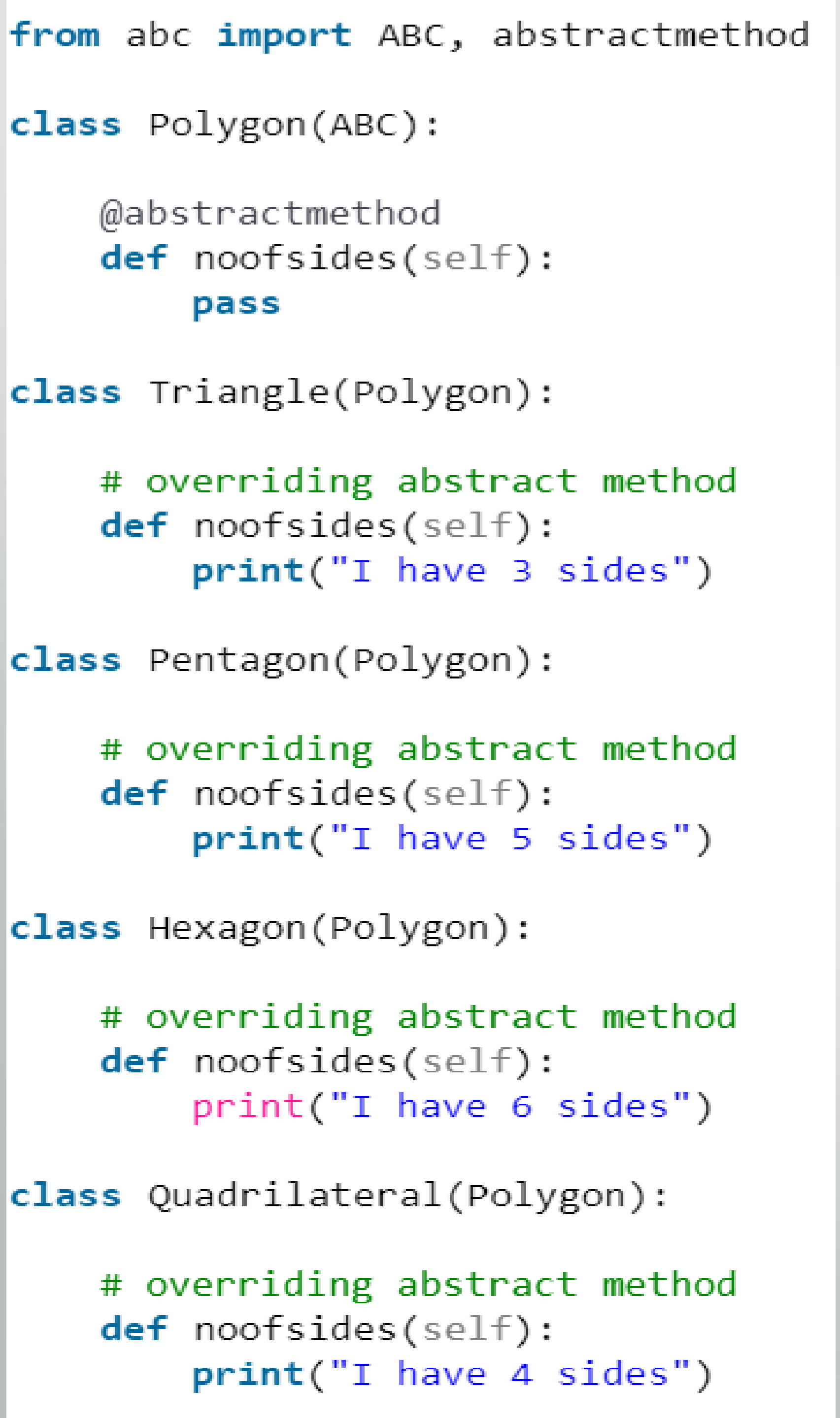
```
class person:
  __name=None
  __age=None
  def __init__(self,name,age):
    self.__name=name
    self.__age=age

  def display(self):
    print("Name:",self.__name,",Age:",self.__age)

class student(person):
  def __init__(self,sid,course,name,age):
    super().__init__(name,age)
    self.sid=sid
    self.course=course
  def display(self):
    super().display()
    print("ID:",self.sid,",Course:",self.course)



s1=student(1,"MCA",“Shubh",22)
s1.display()
```

Output:

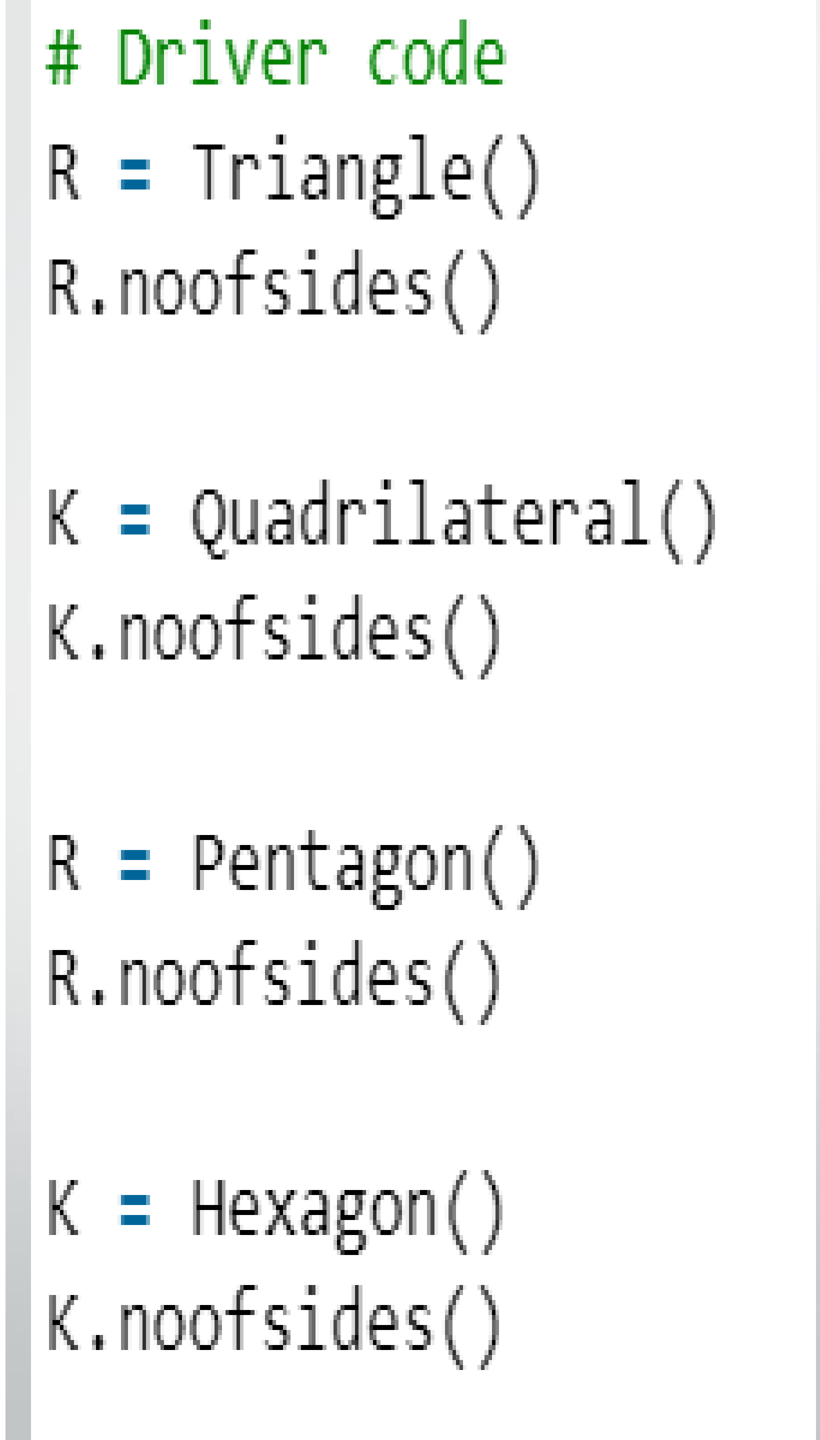
```
Name: Shubh ,Age: 22
ID: 1 ,Course: MCA
```

Abstract Classes & Methods

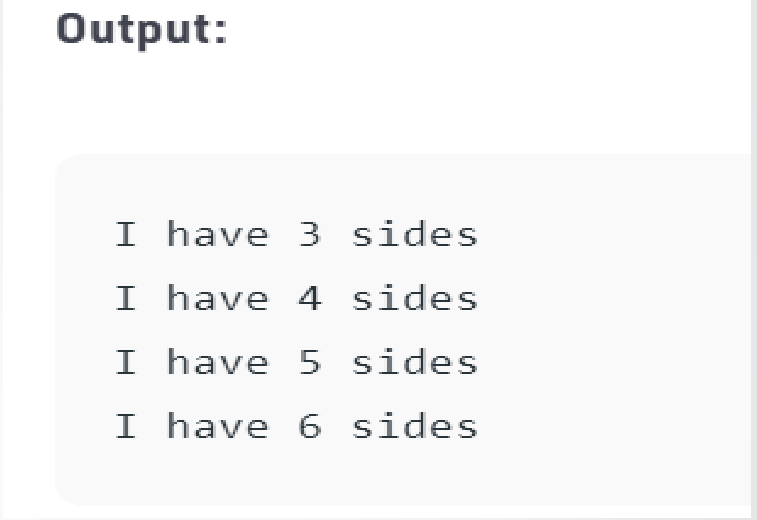
- An abstract class can be considered as a blueprint for other classes. It allows you to create a set of methods that must be created within any child classes built from the abstract class. A class which contains one or more abstract methods is called an abstract class.
- An abstract method is a method that has a declaration but does not have an implementation.
- By defining an abstract base class, you can define a common Application Program Interface(API) for a set of subclasses.
- A method becomes abstract when decorated with the keyword @abstractmethod.



```python
from abc import ABC, abstractmethod

class Polygon(ABC):

    @abstractmethod
    def noofsides(self):
        pass

class Triangle(Polygon):

    # overriding abstract method
    def noofsides(self):
        print("I have 3 sides")

class Pentagon(Polygon):

    # overriding abstract method
    def noofsides(self):
        print("I have 5 sides")

class Hexagon(Polygon):

    # overriding abstract method
    def noofsides(self):
        print("I have 6 sides")

class Quadrilateral(Polygon):

    # overriding abstract method
    def noofsides(self):
        print("I have 4 sides")
```



```python
# Driver code
R = Triangle()
R.noofsides()


K = Quadrilateral()
K.noofsides()


R = Pentagon()
R.noofsides()


K = Hexagon()
K.noofsides()
```



Output:

```
I have 3 sides
I have 4 sides
I have 5 sides
I have 6 sides
```

Why use Abstract Base Classes ?

- By defining an abstract base class, you can define a common Application Program Interface(API) for a set of subclasses.
- This capability is especially useful in situations where a third-party is going to provide implementations, such as with plugins, but can also help you when working in a large team or with a large code-base where keeping all classes in your mind is difficult or not possible.