Object Oriented Programming- I

OOPS stands for Object Oriented Programming System and any programming language that supports this system is called Object Oriented Programming Language.

What is an Object?

In simple words, an object can be anything that we see in real life. Essentially an object have state and behaviour. For example : a book, a table, people around you .These all are considered as object.

What is a class?

It is a blueprint of an object. It's just a plan which does not consume any memory.

For example: If you wish to build a house on a piece of land, you would require a map for your house. Since using same map, you can create several houses and that map will not consume any memory.

How to create class and object?

To create a class, we follow the simple syntax mentioned below

```
class  classname :
    <statement 1>
    <statement 2>
    .
    .
    <statement n >
```

In order to create an object, we follow the syntax mentioned below:

```
objectname = classname()
```

Consider an example where we want to create a class named “Cart” and it should contain one variable maxsize and one function named “printcartsize” and we want to create an object named “P1” of that class and call the function using the object.

The syntax for the above is mentioned below:

```
class Cart:
    maxsize=50
    def printmaxsize(self): #self keyword we will learn afterwards
        print(self.maxsize)s
P1 = Cart()
P1.printmaxsize()
```

Output : 50

What are attributes?

In Python, attributes are the characteristics or properties associated with an object. They define the state and behaviour of the object.
There are two types of attributes in Python : instance and class attribute.

1. **Instance Attributes**

   Instance attributes are specific to an instance of a class. Each instance of a class can have its own set of instance attributes. These attributes are defined within the methods of a class and are prefixed with the **self** keyword.

   Example:

```
class Car:
    def __init__(self, color, model):
        self.color = color
        self.model = model

my_car = Car("red", "SUV")
print(my_car.color)  # Output: red
print(my_car.model)  # Output: SUV
```

   In the above example, **color** and **model** are instance attributes specific to each instance of the Car class.

2. Class attributes

Class attributes are shared among all instances of a class. They are defined outside of any method in a class. Class attributes are accessed using the class name itself or an instance of the class.

Example

```
class Car:
    wheels = 4  # Class attribute

    def __init__(self, color, model):
        self.color = color
        self.model = model

my_car = Car("red", "SUV")
print(my_car.wheels)  # Output: 4

another_car = Car("blue", "sedan")
print(another_car.wheels)  # Output: 4

print(Car.wheels)  # Output: 4
```

In the above example, **wheels** is a class attribute shared by all instances of the Car class.

Access modifiers and encapsulation

In Python, access modifiers and encapsulation are concepts that help control the visibility and accessibility of attributes and methods within a class. Although Python does not have strict access modifiers like some other programming languages (e.g., Java), there are conventions and techniques that can be used to achieve encapsulation.

1. Public Access

   By default, all attributes and methods in Python classes are public, meaning they can be accessed from anywhere within the program. Public attributes and methods can be accessed using dot notation.

   Example:

```
class MyClass:
    def __init__(self):
        self.public_attribute = 10

    def public_method(self):
        return "This is a public method"

obj = MyClass()
print(obj.public_attribute)  # Output: 10
print(obj.public_method())   # Output: This is a public
method
```

2. Protected Access

In Python, attributes and methods that are intended for internal use within a class or its subclasses can be marked as protected by convention. This is done by prefixing the attribute or method name with a single underscore _. Although the attribute or method can still be accessed from outside the class, it indicates that it's for internal use only.

Example

```
class MyClass:
    def __init__(self):
        self._protected_attribute = 20

    def _protected_method(self):
        return "This is a protected method"

obj = MyClass()
print(obj._protected_attribute)    # Output: 20
print(obj._protected_method())         #  Output:  This  is  a
protected method
```

3. Private Access

Python does not have true private access modifiers like some other languages, but by convention, attributes and methods that are intended to be private can be prefixed with a double underscore __. This triggers name mangling, which changes the name of the attribute or method to make it harder to access from outside the class. It's a way of indicating that the attribute or method should not be accessed or overridden directly.

Example

```
class MyClass:
    def __init__(self):
        self.__private_attribute = 30

    def __private_method(self):
        return "This is a private method"

obj = MyClass()
# Accessing private attribute and method using mangled names
print(obj._MyClass__private_attribute)    # Output: 30
print(obj._MyClass__private_method())     # Output: This is a
private method
```

Encapsulation refers to the practice of bundling related attributes and methods together within a class and controlling their access to maintain the integrity and consistency of the object's state. By using access modifiers and following naming conventions, encapsulation in Python can be achieved. It helps in hiding the internal implementation details and providing a clean interface for interacting with the objects.

Decorators

Decorators in Python are a powerful feature that allows you to modify the behaviour of functions or classes without directly changing their source code. Decorators provide a way to add functionality to existing functions classes by wrapping them with another function. They are denoted by the @ symbol followed by the name of the decorator function.

1. Decorators Basics

- A decorator is a function that takes another function as an argument and extends its functionality.
- It returns a new function, which typically adds some code before or after the original function.
- The decorator function is defined above the function it decorates, and the decorator is applied using the @ symbol.

2. Decorators syntax

```
def decorator_func(original_func):
    def wrapper_func(*args, **kwargs):
        # Code to execute before the original function
        result = original_func(*args, **kwargs)
        # Code to execute after the original function
        return result
    return wrapper_func

@decorator_func
def my_function():
    # Function code
    pass
```

3. Multiple Decorators

- You can apply multiple decorators to a single function by stacking them using the @ syntax.
- Decorators are applied from bottom to top, meaning the decorator listed at the bottom is executed first.

4. Decorators for classes

- Decorators can also be applied to classes to modify their behaviour or add functionality.
- The decorator function should take the class as an argument and return a modified or subclassed version of the class.

Example

```
def decorator_class(original_class):
    class ModifiedClass(original_class):
        # Additional code or modifications
        pass
    return ModifiedClass

@decorator_class
class MyClass:
    # Class code
     pass
```

Static Methods

In Python, a static method is a method that belongs to a class rather than an instance of the class.Static methods do not have access to the instance or class state and are independent of the specific instance being created. They are defined using the @staticmethod decorator and can be called directly
from the class without creating an instance of the class.

Here are some key points about static methods in Python:

1. **Declaration**
    - Static methods are defined within a class using the @staticmethod decorator.
    - They do not take any special arguments like self or cls.
    - They can be defined below the class declaration or within the class body.

2. **Usage**
    - Static methods can be called directly from the class without instantiating it.
    - They are not bound to any instance or class attributes and cannot access or modify them directly.
    - They are typically used for utility functions or operations that don't require access to the instance or class state.

3. **Example**

```
class MyClass:
    @staticmethod
    def static_method():

MyClass.static_method() #calling the static method
```

4. Accessing static methods

- Static methods can be accessed by both the class and its instances.
- They are shared among all instances of the class and can be called using either the class name or an instance.

Example

```
class MyClass:
    @staticmethod
    def static_method():
        print("This is a static method")

# Calling static method using class name
MyClass.static_method()

# Calling static method using an instance
obj = MyClass()
obj.static_method()
```

5. Benefits of static methods

- They improve code organisation by grouping related functionality within a class.
- They provide a way to encapsulate utility functions that are closely associated with the class but do not depend on instance or class-specific data.
- Static methods can be easily overridden in subclasses if needed.

Class and Instance Methods

In Python, class methods and instance methods are two types of methods that can be defined within a class. They serve different purposes and have different ways of accessing and manipulating data.

1. **Instance Methods**
    - Instance methods are the most common type of method in Python classes.
    - They are defined within a class and take the self parameter as the first argument, which refers to the instance of the class.
    - Instance methods can access and modify the instance attributes and call other instance methods.
    - They are typically used to operate on the specific instance and its data.

Example:

```
class MyClass:
    def instance_method(self):
        # Access instance attributes
        print(self.attribute)

    def update_attribute(self, new_value):
        # Modify instance attributes
        self.attribute = new_value

obj = MyClass()
obj.attribute = "value"
obj.instance_method()     # Output: value
obj.update_attribute("new value")
obj.instance_method()     # Output: new value
```

2. Class Methods

- Class methods are defined using the @classmethod decorator.
- They take the cls parameter as the first argument, which refers to the class itself.
- Class methods can access and modify class-level attributes and call other class methods.
- They are commonly used for operations that involve the class as a whole, rather than individual instances.

Example

```
class MyClass:
    class_attribute = "class value"

    @classmethod
    def class_method(cls):
        # Access class attributes
        print(cls.class_attribute)

    @classmethod
    def update_class_attribute(cls, new_value):
        # Modify class attributes
        cls.class_attribute = new_value

MyClass.class_method()       # Output: class value
MyClass.update_class_attribute("new class value")
MyClass.class_method()       # Output: new class value
```

Destructors

In Python, destructors are special methods that are automatically called when an object is about to be destroyed or garbage collected. The destructor method is named __del__ and it is used to perform any necessary cleanup actions before an object is removed from memory.
The destructor method is defined within a class using the __del__ special method name. It takes only the self parameter and does not accept any other arguments.

Example:

```
class MyClass:
    def __del__(self):
        # Cleanup actions

obj = MyClass()  # Create an object
del obj          # Destroy the object and trigger the destructor
```

The destructor is automatically invoked when an object is no longer referenced or explicitly deleted using the del keyword. The exact timing of when the destructor is called and the object is removed from memory is determined by the garbage collector.