https://www.linkedin.com/in/pothuri-harshith-varma-4508991b2/

- **Follow me on Linkedin for such more notes.**

Creating Classes

A class is a blueprint for creating objects (a particular data structure), providing initial values for state (member variables or attributes), and implementations of behavior (member functions or methods).

In Python, a class is defined using the `class` keyword, followed by the name of the class. By convention, the name of the class should start with a capital letter.

```
# Creating a Class
class ClassName:
    pass

obj = ClassName()
print(obj)
# Output: <__main__.ClassName object at 0x7feb29dc8e10>
```

Pillars of OOP

The pillars of Object-Oriented Programming (OOP) are:

1. Inheritance: allows new classes to be derived from existing classes, creating a hierarchy of classes that share common attributes and methods.
2. Polymorphism: the ability of objects of different classes to be treated as objects of the same class, allowing for functions and operations to work with objects of multiple classes.
3. Encapsulation: the mechanism of wrapping data and functions within an object, making it inaccessible to the outside world and protected from accidental corruption.
4. Abstraction: the process of hiding complex implementation details and exposing only necessary information to the user.

Inheritance

Inheritance is the mechanism of creating a new class from an existing class, allowing new classes to reuse, extend, or modify the attributes and behaviors defined in the existing class. The new class is called a derived class or a subclass, and the existing class is called a base class or a superclass.

```
class Animal:
    def __init__(self, name):
```

```
        self.name = name

    def show_name(self):
        print(f"Name: {self.name}")

class Dog(Animal):
    pass

dog = Dog("dog")
dog.show_name()
# Output: Name: dog
```

Polymorphism

Polymorphism is the ability of objects of different classes to be treated as objects of the same class, allowing for functions and operations to work with objects of multiple classes.

```
class Circle:
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius * self.radius

class Square:
    def __init__(self, side):
        self.side = side

    def area(self):
        return self.side * self.side

shapes = [Circle(10), Square(20)]
for shape in shapes:
    print(shape.area())
# Output:
# 314.0
# 400
```

Encapsulation

Encapsulation is the mechanism of wrapping data and functions within an object, making it inaccessible to the outside world and protected from accidental corruption. In Python, this is achieved by using the `private` access specifier, which is denoted by a double underscore `__` prefix.

```
class ClassName:
    def __init__(self):
        self.__attribute = 0

    def set_attribute(self, value):
        self.__attribute = value

    def get_attribute(self):
        return self.__attribute
```

```
obj = ClassName()
obj.set_attribute(10)
print(obj.get_attribute())
# Output: 10
```

Abstraction

Abstraction is the process of hiding complex implementation details and exposing only necessary information to the user. In Python, this can be achieved by using abstract classes and abstract methods.

An abstract class is a class that cannot be instantiated, but can be subclassed. An abstract method is a method that is declared in an abstract class but has no implementation. Subclasses of the abstract class are required to provide an implementation for the abstract method.

```
from abc import ABC, abstractmethod

class AbstractClass(ABC):
    @abstractmethod
    def abstract_method(self):
        pass

class ConcreteClass(AbstractClass):
    def abstract_method(self):
        print("Implementation of abstract method")

obj = ConcreteClass()
obj.abstract_method()
# Output: Implementation of abstract method
```

Class Methods and Static Methods

Class methods and static methods are methods that are defined within a class, but they don't receive any reference to the instance of the class.

Class methods are defined using the `@classmethod` decorator and receive a reference to the class itself as the first argument.

Static methods are defined using the `@staticmethod` decorator and don't receive any reference to the class or the instance of the class.

```
class ClassName:
    @classmethod
    def class_method(cls):
        print("Class method")

    @staticmethod
    def static_method():
        print("Static method")

ClassName.class_method()
# Output: Class method
```

```
ClassName.static_method()
# Output: Static method
```

Special (Magic/Dunder) Methods

Special methods in Python, also known as magic or dunder (short for "double underscore") methods, are methods that have a special syntax and are used to define the behavior of objects in Python. Some of the most commonly used special methods are:

- `__init__`: The constructor method that is called when an object is created from a class.
- `__str__`: The method that returns a string representation of the object.
- `__len__`: The method that returns the length of the object.

```
class ClassName:
    def __init__(self):
        self.attribute = 0

    def __str__(self):
        return f"ClassName object with attribute value {self.attribute}"

    def __len__(self):
        return 1

obj = ClassName()
print(obj)
# Output: ClassName object with attribute value 0

print(len(obj))
# Output: 1
```

Property Decorators - Getters, Setters, and Deletes

Property decorators are used to define getters, setters, and deletes for class attributes.

- Getters: methods that are used to retrieve the value of an attribute.
- Setters: methods that are used to set the value of an attribute.
- Deletes: methods that are used to delete an attribute.

```
class ClassName:
    def __init__(self):
        self.__attribute = 0

    @property
    def attribute(self):
        return self.__attribute

    @attribute.setter
    def attribute(self, value):
        self.__attribute = value

    @attribute.deleter
```

```
    def attribute(self):
        del self.__attribute

obj = ClassName()
obj.attribute = 10
print(obj.attribute)

#Output: 10
del obj.attribute

#AttributeError: attribute '_ClassName__attribute' is not defined
```