1. What is a Class?

A **class** in Python is a blueprint for creating objects (a particular data structure). Classes encapsulate data for the object (attributes) and functions (methods) to operate on that data.

```
class MyClass:
    # Class attribute (shared across all instances)
    class_attribute = "I am a class attribute"


    # Constructor (initializes object attributes)
    def __init__(self, attribute_value):
        self.instance_attribute = attribute_value  # instance attribute (unique to each object)


    # Method (function inside a class)
    def my_method(self):
        return f"Instance attribute is {self.instance_attribute}"
```

3. **Creating an Object (Instance) from a Class:**

```
# Create an instance (object) of MyClass
my_object = MyClass("Hello, World!")
# Accessing attributes and methods
print(my_object.instance_attribute)  # Output: Hello, World!
print(my_object.my_method())  # Output: Instance attribute is Hello, World!
```

4. Class Attributes vs. Instance Attributes:

- **Class attributes** are shared across all instances of a class.
- **Instance attributes** are specific to the object and set in the __init__ method.

```
class Example:
    class_attr = "Shared value"  # Class attribute

    def __init__(self, instance_attr_value):
        self.instance_attr = instance_attr_value  # Instance attribute

obj1 = Example("Instance 1")
obj2 = Example("Instance 2")

print(obj1.class_attr)  # Shared value
print(obj2.class_attr)  # Shared value

print(obj1.instance_attr)  # Instance 1
print(obj2.instance_attr)  # Instance 2
```

5. Methods in Classes:

- **Instance methods**: Operate on object instances. They can access and modify instance attributes. Defined with self.
- **Class methods**: Operate on the class itself, not instances. Defined using @classmethod and take cls as the first parameter.
- **Static methods**: Don’t modify object state or class state. Defined using @staticmethod and take no mandatory parameters.

```
class MyClass:
    # Class method
    @classmethod
    def class_method(cls):
        return "This is a class method"

    # Static method
    @staticmethod
    def static_method():
        return "This is a static method"

# Calling methods
print(MyClass.class_method())  # Class method can be called using the class name
print(MyClass.static_method())  # Static method can be called using the class name
```

6. Encapsulation (Public vs. Private):

- By convention, attributes or methods starting with a single underscore _ are treated as **protected** (shouldn't be accessed directly outside the class).
- Attributes or methods with two underscores __ are **private** (name-mangled to avoid direct access from outside).

```
class MyClass:
    def __init__(self):
        self._protected_attribute = "This is protected"
        self.__private_attribute = "This is private"

    def access_private(self):
        return self.__private_attribute

obj = MyClass()
print(obj._protected_attribute)  # Works, but not recommended
# print(obj.__private_attribute)  # This will raise an AttributeError
print(obj.access_private())  # Accessing private via method
```

7. Inheritance:

- **Inheritance** allows a class (child class) to inherit attributes and methods from another class (parent class).
- The super() function is used to call methods of the parent class.

```
class ParentClass:
    def __init__(self, name):
        self.name = name

    def greet(self):
        return f"Hello, {self.name}!"

class ChildClass(ParentClass):
    def __init__(self, name, age):
        super().__init__(name)  # Inherit from ParentClass
        self.age = age

    def show_age(self):
        return f"I am {self.age} years old"

child = ChildClass("Alice", 20)
print(child.greet())  # Hello, Alice!
print(child.show_age())  # I am 20 years old
```

8. Polymorphism:

Polymorphism allows different classes to be treated the same way by defining common methods

9. Dunder (Double Underscore) Methods (Magic Methods):

- Special methods with double underscores are used to define behavior for built-in operations (e.g., __init__, __str__, __len__)

```
class MyClass:
    def __init__(self, value):
        self.value = value

    def __str__(self):
        return f"MyClass with value {self.value}"

obj = MyClass(10)
print(obj)  # Output: MyClass with value 10
```