```
#oops
#Class and Objects

# Class --> Blueprints  --> set of attributes and methods
# Attribute are variables that belong to that class

# class MyClass:
#     x =2
# Attribute

# self The self parameter in methods refers to the instance of the
class. It's used to access the attributes and other methods of the
object.
```

- You can use any name instead of self, but it’s a convention to use self.

```
#self

class Car:
    def __init__(self, brand, model, year):
        self.brand = brand
        self.model = model
        self.year= year
     #Method (behavior)
    def start_engine(self):
        return f"{self.model} has started"

# Create the objects of the class

car1 = Car("Ford" , "civic" , 2020)
car2 = Car("Ford" , "model1" , 2020)

print(car1)
print(car1.model)
print(car2.model)

print(car1.start_engine())
```

__str__ ⇒ string representation of the object

```
class Car:
    def __init__(self, brand, model, year):
        self.brand = brand
        self.model = model
        self.year= year
     #Method (behavior)
    def start_engine(self):
        return f"{self.model} has started"
    def __str__(self):
        return f"{self.brand} and {self.model}"


# Create the objects of the class

car1 = Car("Ford" , "civic" , 2020)
car2 = Car("Ford" , "model1" , 2020)

print(car1)
```

```
1. Write a Python program to create a class representing a Circle.
Include method
```

4 Pillars
Encapsulation
Balance access modifiers
3 public protected private , __balance
Inheritance

Abstraction

Polymorphism

```
#1. Compile Time polymorphism(Method Overloading)
# 3 python does not allow multiple methods with same name , but diff
param

class Math:
 def add(self,a,b,c=0):
  return a+b+c
```

```python
math = Math()
print(math.add(2,3,5))
print(math.add(2,3))


class Math:
    def add(self, *args):
        return sum(args)

math = Math()
print(math.add(2, 3, 5))
print(math.add(2, 3))
print(math.add(2))


#STRING

# runtime polymorphism (method overriding)


class Animal:
   def make_sound(self):
      return "some sound"

class Dog(Animal):
   def make_sound(self):
      return "Bark"

animal = Dog()
print(animal.make_sound())



################################ ENCAPSULATION ############

#Bundling of data(attribute) and methods (function) within a class ,
restrict access to some components

#Types of Encapsulation
# Public Members : Accessible from anywhere
# Protected members: Accessible within the class and its subclass
#Private Members : Accessible only within class
```

```python
class Person:
    def __init__(self,name,age,salary):
      self.name = name
      self._age = age
      self.__salary = salary

    def display(self):
       """MEthod to display details"""
       print(f"Name is {self.name} Salary:{self.__salary}")

    def get_salary(self):
       return self.__salary


person = Person("Alice" , 30 , 50000)
print(person)
print(person.name)
# print(person.__salary) # this will raise an attribute error

print(person.get_salary())
print(person._age)

###abstraction
# hiding the implementation details and exposing only the necessary
part of the object
from abc import ABC, abstractmethod

class Animal(ABC):
   @abstractmethod
   def speak(self):
      pass
   @abstractmethod
   def greet(self):
      pass

class Dog(Animal):
   def speak(self):
      print("woof")

   def greet(self):
      pass

dog = Dog()
```

```python
dog.speak()



# from abc import ABC, abstractmethod

# class Payment(ABC):
#     @abstractmethod
#     def pay(self, amount):
#         pass
#     @abstractmethod
#     def message(self):
#        pass

# class CreditCardPayment(Payment):
#     def pay(self, amount):
#         print(f"Paid {amount} using Credit Card.")

# class PayPalPayment(Payment):
#     def pay(self, amount):
#         print(f"Paid {amount} using PayPal.")

# # Creating objects
# payment1 = CreditCardPayment()
# payment1.pay(1000)  # Output: Paid 1000 using Credit Card.

# payment2 = PayPalPayment()
# payment2.pay(500)   # Output: Paid 500 using PayPal.


from abc import ABC, abstractmethod

class UserAuth(ABC):
    @abstractmethod
    def authenticate(self, username, password):
        pass

class GoogleAuth(UserAuth):
    def authenticate(self, username, password):
        print("Authenticated using Google OAuth.")

class FacebookAuth(UserAuth):
    def authenticate(self, username, password):
        print("Authenticated using Facebook API.")

# User only calls authenticate(), they don’t see how passwords are
verified
auth = GoogleAuth()
auth.authenticate("user", "pass")  # Output: Authenticated using Google
OAuth.

######################################3Inheritance ###


#Single inheritance
# One child class inherits from one parent

class Parent:
   pass
class Child(Parent):
   pass


## Mulyiple Inheritance
# A child class inherits from more than one parent

class Parent1 :
   pass
class Parent2:
   pass

class child(Parent1,Parent2):
   pass


## Multilevel inheritance


class GrandParent1 :
   pass
class Parent1(GrandParent1):
   pass

class child(Parent1):
   pass
```

```
## Hierachial inheritance

class Parent1:
   pass

class child1(Parent1):
   pass

class child2(Parent1):
   pass



## Hybrid inheritance

class Parent:
    pass

class Child1(Parent):
    pass

class Child2(Parent):
    pass

class GrandChild(Child1, Child2):
    pass

class Animal:
   def __init__(self,name):
      self.name = name
   def makesound(self):
       print("Sound from parent")


class Dog(Animal):
   def makesound(self):
      print("sound from child")
   def __init__(self,name,breed):
      super().__init__(name)
      self.breed = breed
      super().makesound()
```

```
dog = Dog("Tom", "Rottweiller")
print(dog.name)
print(dog.breed)
```

Create an abstract class `Device` with an abstract method `turn_on()`.

- Derive `Smartphone` and `Laptop` classes from it and implement the `turn_on()` method.

Create a `BankAccount` class with a private attribute `_balance`.

- Implement methods `deposit()`, `withdraw()`, and `get_balance()` to modify and access `_balance`.

Create a `Bird` class with a method `make_sound()`.

- Derive `Parrot` and `Sparrow` classes that override `make_sound()` with their own implementation.

Create a function `play_sound()` that takes a `Bird` object and calls its `make_sound()` method.
Create an abstract class `Vehicle` with private attributes `_speed` and `_fuel`.

- Implement a method `drive()` and create subclasses `Car` and `Bike` to define how they are driven.

```
# Assignment: OOP Pillars - Abstraction, Encapsulation, and
Polymorphism

# Q1: Abstraction
# Create an abstract class `Device` with an abstract method
`turn_on()`.
# - Derive `Smartphone` and `Laptop` classes from it and implement the
`turn_on()` method.

from abc import ABC, abstractmethod
```

```python
class Device(ABC):
    @abstractmethod
    def turn_on(self):
        pass

class Smartphone(Device):
    def turn_on(self):
        print("Smartphone is turning on...")

class Laptop(Device):
    def turn_on(self):
        print("Laptop is booting up...")

smartphone = Smartphone()
laptop = Laptop()
smartphone.turn_on()
laptop.turn_on()


# Q2: Encapsulation
# Create a `BankAccount` class with a private attribute `_balance`.
# Implement methods `deposit()`, `withdraw()`, and `get_balance()` to
modify and access `_balance`.

class BankAccount:
    def __init__(self, initial_balance):
        self._balance = initial_balance

    def deposit(self, amount):
        if amount > 0:
            self._balance += amount
            print(f"Deposited: ${amount}")
        else:
            print("Invalid deposit amount")

    def withdraw(self, amount):
        if 0 < amount <= self._balance:
            self._balance -= amount
            print(f"Withdrawn: ${amount}")
        else:
            print("Insufficient balance or invalid amount")

    def get_balance(self):
        return self._balance

account = BankAccount(500)
account.deposit(200)
account.withdraw(100)
print("Current Balance:", account.get_balance())


# Q3: Polymorphism
# Create a `Bird` class with a method `make_sound()`.
# Derive `Parrot` and `Sparrow` classes that override `make_sound()`
with their own implementation.

class Bird:
    def make_sound(self):
        print("Bird is making a sound")

class Parrot(Bird):
    def make_sound(self):
        print("Parrot says: Hello!")

class Sparrow(Bird):
    def make_sound(self):
        print("Sparrow chirps: Chirp Chirp!")

parrot = Parrot()
sparrow = Sparrow()
parrot.make_sound()
sparrow.make_sound()


# Q4: Using Polymorphism in Functions
# Create a function `play_sound()` that takes a `Bird` object and calls
its `make_sound()` method.

def play_sound(bird):
    bird.make_sound()

play_sound(Parrot())
play_sound(Sparrow())


# Q5: Combining Encapsulation and Abstraction
# Create an abstract class `Vehicle` with private attributes `_speed` 
and `_fuel`.
# Implement a method `drive()` and create subclasses `Car` and `Bike` 
to define how they are driven.

class Vehicle(ABC):
    def __init__(self, speed, fuel):
        self._speed = speed
        self._fuel = fuel

    @abstractmethod
    def drive(self):
        pass

class Car(Vehicle):
    def drive(self):
        if self._fuel > 0:
            print(f"Car is driving at {self._speed} km/h")
        else:
            print("Car is out of fuel")

class Bike(Vehicle):
    def drive(self):
        if self._fuel > 0:
            print(f"Bike is moving at {self._speed} km/h")
        else:
            print("Bike is out of fuel")

car = Car(60, 10)
bike = Bike(40, 0)
car.drive()
bike.drive()
```

```python
# DECORATORS , MODIFY AND EXTEND THE BEHAVIOUS OF FUNCTIONS AND 
METHODS, WITHOUT CHANGING THE ACTUAL CODE

# ,ALLOWS YOU TO WRAP ANOTHER FUNCTION
# FUNCTIONS CAN BE PASSED AS AN aGRUMENT TO ANOTHER FUNCTION

# WRAPPER FUNCTION --> a DECORATOR USES INNER FUNCTION TO MODIFY THE
BEHAVIOUR

def decorator_function(original_function):
    def wrapper_function():
        print("Function is being decorated")
        return original_function()
    return wrapper_function




@decorator_function
def display():
    print("display function is being called")


display()



# The decorator that accepts an argument
import time

# The decorator that accepts an argument

def time_decorator(log_message):
    def decorator_function(original_function):
        def wrapper_function():
            start_time = time.time()  # Record the start time
            result = original_function()  # Call the original function
            print
            end_time = time.time()  # Record the end time
            execution_time = end_time - start_time
            print(f"{log_message} - Function
{original_function.__name__} took {execution_time:.4f} seconds to
execute.")
            return result
        return wrapper_function
    return decorator_function
```

```python
# Using the decorator with a custom message
# TIME DEC = (MSG)(SLOW FUNC)
@time_decorator("Execution Time Log")
def slow_function():
    time.sleep(2)  # Simulate a slow function by sleeping for 2 seconds
    print("Slow function executed.")

slow_function()
```