Top of Form

## ****Building your 1st Class and Object****

### Theory:

* A **class** is a blueprint/template.
* An **object** is an instance of a class.
* type() shows what type something is.
* When you create an object using ClassName(), Python internally calls the class constructor.

class chai:

    pass   # empty class

class chaitime():

    pass   # another empty class

print(type(chai))

# <class 'type'> -> means 'chai' itself is a class

ginger\_chai = chai()

# creates an object (instance) of class chai

print(type(ginger\_chai))

# <class '\_\_main\_\_.chai'> -> means this object is from chai class

print(type(ginger\_chai) is chai)

# True -> confirms ginger\_chai belongs to chai

print(type(ginger\_chai) is chaitime)

# False -> because ginger\_chai is not an instance of chaitime

## ****59. Class and Object Namespace****

### Theory:

* **Class attributes** → shared by all objects.
* **Object attributes** → unique to each object.
* You can add new attributes dynamically to both class and object.

class chai:

    origin = "india"   # class attribute

print(chai.origin)   # Accessing class attribute -> india

chai.is\_hot = True   # Adding new class attribute dynamically

print(chai.is\_hot)   # True

# creating object from class chai

masala = chai()

print(f"Masala {masala.origin}")   # inherits from class -> india

print(f"Masala {masala.is\_hot}")   # inherits from class -> True

print("-----------------------------------------")

masala.is\_hot = False  # creates a new object attribute (shadowing class attr)

print("Class: ",chai.is\_hot)       # Class attribute -> True

print(f"Masala {masala.is\_hot}")   # Object attribute -> False

masala.flavour = "masala"  # adding new attribute only to this object

print(masala.flavour)      # masala

## ****60. Attribute Shadowing****

### Theory:

* If an attribute exists in both **class and object**, Python will first check the **object’s namespace**.
* If deleted, Python falls back to the **class attribute**.

class chai:

    temprature = "hot"   # class attribute

    strenght = "strong"

cutting = chai()

print(cutting.temprature)  # "hot" (from class)

cutting.temprature = "Mild"   # shadows the class attribute

cutting.cup = "small"         # new object attribute

print("After Changing",cutting.temprature)  # "Mild"

print("cup size is",cutting.cup)            # small

print("Direct look into the class",chai.temprature)  # still "hot"

del cutting.temprature  # removes object attribute

del cutting.cup

print(cutting.temprature)  # falls back to class attribute -> "hot"

print(cutting.cup)         # Error (object doesn’t have it, class also doesn’t)

## ****61. Self Argument****

### Theory:

* self refers to the current instance (object).
* Methods must explicitly take self as their first parameter.
* You can call method in two ways:
  + obj.method()
  + Class.method(obj)

class chaicup:

    size = 150   # class attribute

    def describe(self):

        # self -> current object calling this method

        return f"A {self.size} ml chai cup "

cup = chaicup()

print(cup.describe())         # via object

print(chaicup.describe(cup))  # via class

cup\_two = chaicup()

cup\_two.size = 100   # object attribute shadows class attribute

print(chaicup.describe(cup\_two))  # A 100 ml chai cup

## ****62. Constructors (****\_\_init\_\_****)****

### Theory:

* \_\_init\_\_ is the constructor.
* It runs automatically when an object is created.
* Used to initialize instance attributes.

class chaiorder:

    def \_\_init\_\_(self, type\_, size):

        self.type = type\_   # instance attribute

        self.size = size

    def summary(self):

        return f"{self.size} ml of {self.type} chai"

order = chaiorder("Masala", 2000)

print(order.summary())  # 2000 ml of Masala chai

order1 = chaiorder("Lemon", 1000)

print(order1.summary()) # 1000 ml of Lemon chai

## ****Coding Exercise 26: Smart Home Device Tracker****

✅ Your solution is correct!

## ****63. Inheritance and Composition****

### Theory:

* **Inheritance** → One class inherits from another (child → parent).
* **Composition** → One class contains an object of another class (like "has-a" relationship).

class BaseChai:

    def \_\_init\_\_(self,type\_):

        self.type = type\_

    def prepare(self):

        print(f"preparing {self.type} chai ....")

class MasalaChai(BaseChai):   # Inheritance

    def add\_spices(self):

        print("Adding cardamom, ginger, cloves")

class ChaiShop:

    chai\_cls = BaseChai   # composition -> ChaiShop has a Chai object

    def \_\_init\_\_(self):

        self.chai = self.chai\_cls("Regular")  # Shop contains Chai

    def serve(self):

        print(f"serving {self.chai.type} chai in shop")

        self.chai.prepare()

class FansyChaiShop(ChaiShop):

    chai\_cls = MasalaChai   # composition with a different class

shop = ChaiShop()

fansy = FansyChaiShop()

shop.serve()     # Uses BaseChai

fansy.serve()    # Uses MasalaChai

fansy.chai.add\_spices()  # Extra method only available in MasalaChai

🔑 **Composition here:**

* ChaiShop doesn’t directly **inherit** from BaseChai.
* Instead, it **creates an instance** of chai\_cls (BaseChai or MasalaChai) → That’s **composition** ("has-a" relationship).

## ****64. 3 Ways to Access Base Class****

### Theory:

When creating a subclass, you need to call the base class constructor so attributes get initialized.  
There are **3 ways**:

1. **Code duplication** → re-write base attributes (bad practice).
2. **Explicit call** → BaseClass.\_\_init\_\_(self, ...)
3. **super()** → recommended, automatically finds the parent class using **MRO (Method Resolution Order)**.

class Chai:

    def \_\_init\_\_(self,type\_,strength):

        self.type = type\_

        self.strength = strength

# ❌ BAD: Code duplication

# class GingerChai(Chai):

#     def \_\_init\_\_(self, type\_, strength, spice\_level):

#         self.type = type\_

#         self.strength = strength

#         self.spice\_level = spice\_level

# ✅ Option 2: Explicit call

# class GingerChai(Chai):

#     def \_\_init\_\_(self, type\_, strength, spice\_level):

#         Chai.\_\_init\_\_(self, type\_, strength)  # calling base class

#         self.spice\_level = spice\_level

# ✅ BEST: super()

class GingerChai(Chai):

    def \_\_init\_\_(self, type\_, strength, spice\_level):

        super().\_\_init\_\_(type\_, strength)  # automatically finds base class

        self.spice\_level = spice\_level

ginger = GingerChai("Herbal", "Medium", "High")

print(ginger.\_\_dict\_\_)

# {'type': 'Herbal', 'strength': 'Medium', 'spice\_level': 'High'}

## ****65. Method Resolution Order (MRO)****

### Theory:

* Python uses **MRO** to decide which class method/attribute is used when multiple inheritance is involved.
* You can check with Class.\_\_mro\_\_.

class A:

    label = "A: Base class"

class B(A):

    label = "B: Masala blend"

class C(A):

    label = "C: Herbal Blend"

class D(B,C):

    pass   # inherits from B first, then C

cup = D()

print(cup.label)     # Output: B: Masala blend

print(D.\_\_mro\_\_)

# Shows: (<class '\_\_main\_\_.D'>, <class '\_\_main\_\_.B'>, <class '\_\_main\_\_.C'>, <class '\_\_main\_\_.A'>, <class 'object'>)

🔑 **Rule**: Python looks **left to right** in inheritance. Since D(B, C), it chooses B first.

## ****66. Static Methods****

### Theory:

* Use @staticmethod when the method does **not depend on class or object attributes**.
* It works like a normal function but lives inside the class for organization.

class chaiutils:

    @staticmethod

    def clean\_ingridients(text):

        # split by comma and strip spaces

        return [item.strip() for item in text.split(",")]

raw = "water,  milk,  ginger,  honey"

cleaned = chaiutils.clean\_ingridients(raw)

print(cleaned)   # ['water', 'milk', 'ginger', 'honey']

## ****67. Classmethod vs Staticmethod****

## 

### Theory:

* **@classmethod**
  + First parameter is cls (refers to class).
  + Used to create objects in alternative ways (factory methods).
* **@staticmethod**
  + No self or cls.
  + Utility functions related to the class.

class orderchai:

    def \_\_init\_\_(self,tea\_type,sweetness,size):

        self.tea\_type = tea\_type

        self.sweetness = sweetness

        self.size = size

    @classmethod

    def from\_dict(cls,order\_data):   # factory method

        return cls(

            order\_data["tea\_type"],

            order\_data["sweetness"],

            order\_data["size"],

        )

    @classmethod

    def from\_string(cls, order\_string):  # another factory method

        tea\_type, sweetness, size = order\_string.split("-")

        return cls(tea\_type, sweetness, size)

class ChaiUtils:

    @staticmethod

    def is\_valid\_size(size):

        return size in ["Small", "Medium", "Large"]

print(ChaiUtils.is\_valid\_size("Medium"))   # True

# Creating objects in 3 ways

order1 = orderchai.from\_dict({"tea\_type": "masala", "sweetness": "medium", "size":"Large"})

order2 = orderchai.from\_string("Ginger-Low-Small")

order3 = orderchai("Large", "Low", "Large")

print(order1.\_\_dict\_\_)   # {'tea\_type': 'masala', 'sweetness': 'medium', 'size': 'Large'}

print(order2.\_\_dict\_\_)   # {'tea\_type': 'Ginger', 'sweetness': 'Low', 'size': 'Small'}

print(order3.\_\_dict\_\_)   # {'tea\_type': 'Large', 'sweetness': 'Low', 'size': 'Large'}

## ****68. Property Decorator – Getter & Setter****

### Theory:

* @property turns a method into an attribute (getter).
* @<property>.setter controls assignment (validation).
* Helps with **encapsulation** (control access to private attributes).

class tealeaf:

    def \_\_init\_\_(self,age):

        self.\_age = age   # private-like (by convention, not enforced)

    @property

    def age(self):   # getter

        return self.\_age + 2   # adding +2 to demonstrate transformation

    @age.setter

    def age(self,age):  # setter

        if 1 <= age <= 5:

            self.\_age = age

        else:

            raise ValueError("Tea age must be bw 1 and 5")

leaf = tealeaf(2)

print(leaf.age)   # 4 (because getter adds +2)

# leaf.age = 6  # ❌ raises ValueError

leaf.age = 3

print(leaf.age)  # 5

## ****Coding Exercise 27: Vehicle Rental System****

### Theory:

This exercise uses **all major OOP features**:

* **Composition** → Vehicle "has an" Engine.
* **Inheritance** → Car inherits from Vehicle.
* **Class attribute** → total\_vehicles shared across all.
* **Static method** → get\_vehicle\_type() utility.
* **Class method** → get\_total\_vehicles() uses class reference.
* **Property** → rental\_price with getter/setter validation.

class Engine:

    def \_\_init\_\_(self, horsepower):

        self.horsepower = horsepower

    def get\_engine\_info(self):

        # returns e.g. "150 HP Engine"

        return f"{self.horsepower} HP Engine"

class Vehicle:

    total\_vehicles = 0   # class attribute

    def \_\_init\_\_(self, brand, model, engine: Engine):

        if not isinstance(engine, Engine):

            raise TypeError("engine must be an Engine instance")

        self.brand = brand

        self.model = model

        self.engine = engine

        # increment class counter

        Vehicle.total\_vehicles += 1

        # default rental price

        self.\_rental\_price = 0

    def get\_details(self):

        return f"Brand: {self.brand}, Model: {self.model}, Engine: {self.engine.get\_engine\_info()}"

    @staticmethod

    def get\_vehicle\_type():

        return "Generic Vehicle"

    @classmethod

    def get\_total\_vehicles(cls):

        return cls.total\_vehicles

    @property

    def rental\_price(self):   # getter

        return self.\_rental\_price

    @rental\_price.setter

    def rental\_price(self, value):   # setter

        if not isinstance(value, (int, float)):

            raise TypeError("rental\_price must be a number")

        if value < 0:

            raise ValueError("Rental price cannot be negative.")

        self.\_rental\_price = value

class Car(Vehicle):   # inheritance

    def \_\_init\_\_(self, brand, model, engine: Engine, seats):

        super().\_\_init\_\_(brand, model, engine)

        self.seats = seats

    def get\_details(self):   # overriding parent method

        return f"{super().get\_details()}, Seats: {self.seats}"

# --------- Usage ---------

engine1 = Engine(150)

car1 = Car("Toyota", "Corolla", engine1, 5)

car1.rental\_price = 3000

print(car1.get\_details())       # Brand: Toyota, Model: Corolla, Engine: 150 HP Engine, Seats: 5

print("Rental Price:", car1.rental\_price)  # 3000

print("Vehicle Type:", Car.get\_vehicle\_type())  # Generic Vehicle

print("Total Vehicles:", Vehicle.get\_total\_vehicles())  # 1