Introduction to Object-Oriented Programming

`Concept` : Object-Oriented Programming (OOP) is a programming paradigm that uses "objects" to design applications and programs. Objects can be seen as real-world entities, having attributes (data) and methods (functions)

`Features` :

1 `Inheritance`

2. `Polymorphism`

3. `Encapsulation`

4. `Abstractions`

`Scenario` : Imagine a car rental system. Each car is an object with attributes like make, model, year, and methods like start, stop, and rent

In [ ]:
```
# Class Insrtrument:

#     def how_to_play():
#     def keyOrStrings()

# piano=
# guitar=
```

Defining a `Class`

`Concept` : A class is a blueprint for creating objects. It defines a set of attributes and methods that the objects created from the class can have.

`Keyword` :

1. `__init__`

In Python, the **init** method is a special method known as a constructor. It is automatically called when an instance (object) of the class is created. The primary purpose of the **init** method is to initialize the instance's attributes with values provided during the creation of the object.

Initialization: The **init** method allows you to initialize the object's attributes with initial values.

2. `Self` : The first parameter of **init** is always self, which refers to the instance being created.

In [ ]:
```python
class Car:

    def __init__(self, make: str, model: str, year: str):
        self.make = make
        self.model = model
        self.year = year

    def start(self):
        print(f"{self.make} {self.model} is starting.")

    def stop(self):
        print(f"{self.make} {self.model} is stopping.")
```

In [ ]:
```python
# class Human:

#     def __init__(self, name, age, gender):
#         self.name = name
#         self.age = age
#         self.gender = gender

# students=Human()
# officeWorker=Human()
# Parents=Human()
```

Creating `Objects`

`Concept` : An object is an instance of a class. You can create multiple objects from the same class.

`Scenario` : Let's create two car objects.

In [ ]:
```
car1 = Car("Toyota", "Camry", 2020)
car2 = Car("Honda", "Accord", 2019)

car1.start()
car1.stop()

car2.start()
car2.stop()
```

```
Toyota Camry is starting.
Toyota Camry is stopping.
Honda Accord is starting.
Honda Accord is stopping.
```

Attributes and Methods

`Concept` : Attributes are the variables that belong to a class, and methods are the functions that belong to a class.

`Scenario` : Let's add a new attribute is_rented and a method rent to our Car class.

In [ ]:
```
class Car:
    def __init__(self, make, model, year,is_rented=False):
        self.make = make
        self.model = model
        self.year = year
        self.is_rented = is_rented

    def start(self):
        print(f"{self.make} {self.model} is starting.")

    def stop(self):
```

```
        print(f"{self.make} {self.model} is stopping.")

    def charge(self,cap):
        print(f"{self.make} {self.model} charges at {cap}Kw.")

    def rent(self):
        if not self.is_rented:
            self.is_rented = True
            print(f"{self.make} {self.model} has been rented.")
        else:
            print(f"{self.make} {self.model} is already rented.")


car1 = Car("Toyota", "Camry", 2020, is_rented=True)

#car1.rent()
car1.charge(cap=510)
```

```
Toyota Camry charges at 510Kw.
```

Inheritance

Concept : Inheritance allows a class to inherit attributes and methods from another class. This promotes code reuse.

Scenario : Imagine we have a new type of car, an ElectricCar, which has an additional attribute battery_capacity.

Keyword

1. super()

The super() function in Python is used to give you access to methods and properties of a parent or sibling class. The super() function returns an object that represents the parent class. It is particularly useful in inheritance, where you want to call a method from a parent class in a child class.

In [ ]:
```
#ElectricCar is a child class. Car is a parent class.
# Child always inherits from Parents.
```

```
#ElectricCar is inheriting attributes from the parent.

class ElectricCar(Car):
    def __init__(self, make, model, year, battery_capacity):
        super().__init__(make, model, year) #this set of attributes comes from parent class
        self.battery_capacity = battery_capacity #this attribute is exclusive to the child class

    def info(self): #this method is exclusive to my child class
        print(f"{self.make} {self.model} has {self.battery_capacity}KW")
```

In [ ]:
```
tesla=ElectricCar(make='tesla',model='Model S', year=2021, battery_capacity=510)
```

In [ ]:
```
tesla.info()
```

```
tesla Model S has 510KW
```

Polymorphism

Concept : Polymorphism allows methods to be used interchangeably between different classes that share the same method names.

Scenario : Both Car and ElectricCar have a start method, but they can be called on their respective instances.

In [ ]:
```python
class ElectricCar(Car):
    def __init__(self, make, model, year, battery_capacity):
        super().__init__(make, model, year)
        self.battery_capacity = battery_capacity

    def charge(self, is_charging=False):
        super().charge(cap=510)
        if not is_charging:
            is_charging=True
            print('My car is charging now!')
        print(f"{self.make} {self.model} is charging its {self.battery_capacity}kWh battery.")
```

In [ ]:
```python
#method overriding and method overloading
```

In [ ]:
```python
tesla=ElectricCar(make='tesla',model='Model S', year=2021, battery_capacity=510)
tesla.charge()
```

```
tesla Model S charges at 510Kw.
My car is charging now!
tesla Model S is charging its 510kWh battery.
```

Encapsulation

`Concept`: Encapsulation is the concept of `restricting access` to certain attributes and methods, usually by making them private.

`Scenario`: Let's make the is_rented attribute private and provide methods to access it.

In [ ]:
```python
class Car:
    def __init__(self, make, model, year):
        #setter function
        self.__make = make #private
        self.model = model #public attribute
        self.year = year # public attributes
```

```
        self.__is_rented = False #private attribute

    def start(self):
        print(f"{self.make} {self.model} is starting.")

    def stop(self):
        print(f"{self.make} {self.model} is stopping.")

    def rent(self):
        if not self.__is_rented:
            self.__is_rented = True
            print(f"{self.__make} {self.model} has been rented.")
        else:
            print(f"{self.__make} {self.model} is already rented.")

    #getter for rent
    def is_rented(self):
        return self.__is_rented
    #getter for make
    def getMake(self):
        return self.__make

car1 = Car("Toyota", "Camry", 2020)

car1.is_rented()
```

Out[ ]: False

In [ ]:
```
car1.rent()
```

```
Toyota Camry has been rented.
```

In [ ]:
```
#c#, Java

#getter: gets the value of the class attribute
#setter: Sets the value of the class parameter/attribute


#Parametres are variable that we pass in a function
#attributes are the parameters of the class
```

In [ ]:
```
car1.is_rented()
```

Out[ ]:
```
False
```

Abstraction

`Concept` : Abstraction in object-oriented programming is the concept of hiding the complex implementation details and showing only the necessary features of an object. This helps in reducing programming complexity and effort. It focuses on what an object does rather than how it does it.

In [ ]:
```
from abc import ABC, abstractmethod

#Parent Class
class Car(ABC):

    #Abstract Method
    @abstractmethod
    def color(self):
        pass

class Electric(Car):

    def color(self):
        print("Yellow")

class Diesel(Car):

    def color(self,a,b):
        _add=a+b
        print(_add)
        print("Red")
```

In [ ]:
```
# #java--> Interfaces

# Class MusicPlayer:
```

```
#     interface Iplayer_UI


# class sonyMusciplayer(MusicPlayer)

#     def Iplayer_UI:
#         print('xyz')
```

In [ ]:
```
e=Electric()
d=Diesel()

e.color()

d.color(a=4,b=7)
```

```
Yellow
11
Red
```

In [ ]:
```
from manipulateData import add,subtract,multiply

add_numbers=add(5,6)
print(add_numbers)

subtract_numbers=subtract(10,2)
print(subtract_numbers)
```

```
11
8
```

In [ ]:
```

```