OOPs (Contd.)

Topics:

- Modules in Python
- OOPs Principle
    - Encapsulation (tomorrow)
    - Inheritance
    - Abstraction (tomorrow)
    - Polymorphism (tomorrow)

We can create a module/package (fancy name for folder). Any folder which has ***__init__.py*** file then the folder becomes a module/package.

IN a vehicle folder, we will have car.py, truck.py, abc.py. if this folder also has the __init__.py file, then this will become a module or a package folder and we can import this module/package.

Create a folder vehicle. Then create a __init__.py. once it is done, this becomes a module.

IN **main.py**

```
from vehicle import Car

if __name__ == "__main__":

    car = Car()
```

IN **vehicle.py**

```
class Car:

    def __init__(self):
        print("Car")
```

OOP's Concept.

class Animal:

attributes:	methods:
teeth	run
legs	eat
tail	
eyes	

Let’s create two class, class Dog and class Cat. They will have attribute of class Animals. But other than this, they will have their own attributes as well, like a dog can bark, but a cat can meow.

We can understand that class Animal is the parent class for both ***class Dog*** and ***class Cat.***

This representation in code is called **Inheritance**.

Just as we receive certain common attributes from our parents who have received them from his parents. So, in CODE:

```
class Animal:

    def __init__(self, name, no_of_legs, has_tail):
        """
        Animal: constructor for the class Animal
        """
        print("Animal constructor is called.")

        self.name = name
        self.no_of_legs = no_of_legs
        self.has_tail = has_tail

    def eat(self):
        """
        eat: takes no argument and will make the
        animal eat food.
        """
        print(f"{self.name} is eating")
    def walk(self):
        """
        walk: animal will walk
        """
        print(f"{self.name} is walking")
```

IN main.py

```
from animals.dog import Dog

if __name__ == "__main__":

    bruno = Dog("bruno", 4, True)
    bruno.bark()
```

Creating a class Dog

Whenever we say, a = Animal (“ABC”, 4, True), the def __init__ will be called first.

In class Dog (Animal), which has its own constructor (def __init__) will take its own attributes. In this, if we call dog, the dog constructor would be called. Then, who would call the class Animal?

So, we have to call class Animal as well. When we are doing inheritance we should called the parent of the class Dog as well.

```
from animals.animal import Animal

class Dog(Animal):

    def __init__(self, name, no_of_legs, has_tail):
        """
        Dog: constructor for the dog class. Here it
        is calling the parents class constructor so that the attributes of the
        parent can be initialized.
        """
        print("Dog constructor is called")
        super().__init__(name, no_of_legs, has_tail)

    def bark(self):
        """
        bark: bark is a method by which our dog will bark. Since dog is a
        type of animal so it will have all the attributes of animal. ie self.name is
        accessible in Dog class.
        """
        print(f"{self.name} is barking")
```

So, when we call bruno = Dog (“Bruno”,4, True). It will call the Animal constructor and will create a dog.

OUTPUT:

Dog constructor is called
Animal constructor is called.
bruno is barking

real_life_example:

OOPs University:

The first to model would be *class University*. Prior to this, we will create a ***main.py*** file. In this main.py file,

```
if __name__ == "__main__":
    attainu = university("attainu", "noida-128")
```

let's create a class now,

class University

```
class University:
    def __int__(self, name, location):
        self.name  = name
        self.location = location
```

we have created a class University. AttainU, DU Madhya Pradesh University is also a type of university. Can you see inheritance?

Let's create another *class Attainu* and will import it. If your using pycharm, then use Alt + Enter, and things would be automatically imported. So, the code would look like,

```
from university.university import University
class Attainu:
    pass
```

We will make a constructor for it, (self, name, location, logo). We will add logo to attainu and then we will add, self.logo = logo for *class Attainu*.

```
from Sessions.Week_04.day2.university.university import University


class Attainu:
    def __init__(self, name, location, logo):
        super().__init__(name,location)
```

We are using super () to call the parent constructor as well. We can add the logo now to the *class Attainu*.

```
from Sessions.Week_04.day2.university.university import University
class Attainu:
```

```
    def __init__(self, name, location, logo):
        super().__init__(name,location)

        self.logo = logo
```

What will be the methods of *class Attainu*?

```
    def get_isa(self):
        print(f"{self.name}:getting isa for batch")

    def make_batch(self, batch):
        print(f"{self.name}: making {batch} for the student")
```

IN **main.py**, we will import the *class Attainu*. We can pass the name, location and logo for Attainu.

```
from university.attainu import Attainu

if __name__ == "__main__":

    attainu = Attainu("Attainu", "Noida-128", "A")
    attainu.make_batch("Subramanyam")
    attainu.get_isa()
```

OUTPUT:

attainu: making subramanyam batch for the student
attainu: getting isa for batch

Now, we can create a new package as **batch** in which we will create a new file *batch.py*

```
class Batch:

    def __init__(self, name):

        self.name = name
        self.no_of_students = no_of_students
        self.mentors = list()
        self.instructors = list()
```

Attainu university can have multiple batches, so it can have another attribute as batches.

```
from university.university import University

class Attainu:

    def __init__(self, name, location, logo):
        super().__init__(name,location)

        self.logo = logo
```

```
        self.batch = list()

    def get_isa(self):
        print(f"{self.name}:getting isa for batch")

    def make_batch(self, batch):
        print(f"{self.name}: making {batch} for the student")
```

We will change make_batch to add_batch and will give the instruction as *self.batches.append()*

```
from university.university import University

class Attainu:

    def __init__(self, name, location, logo):
        super().__init__(name,location)
        self.logo = logo
        self.batch = list()

    def get_isa(self):
        print(f"{self.name}:getting isa for batch")

    def add_batch(self, batch):
        self.batches.append(batch)
```

we can add batches to attainu. If we create def print_all_batches, it will create all batches.

```
    def print_all_batches(self):
        for batch in self.batches:
            print(batch.name)
```

IN **main.py**

```
from university.attainu import Attainu

if __name__ == "__main__":
    attainu = Attainu("Attainu", "Noida-128", "A")

    Subramanyam = Batch("Subramanyam", 100)
    CV_Raman = Batch("CV_Raman", 50)
    Aryabhata = Batch("Aryabhata", 10)
```

Now, in Subramanyam, we can add,

```
    attainu.add_batch(Subramanyam)
    attainu.add_batch(CV_Raman)
    attainu.add_batch(Aryabhata)

    attainu.print_all_batches()
```

if you ***RUN***

Subramanyam
CV_Raman
Aryabhata