Module 14

OOP in Python.

Object Oriented Programming System (OOPS)

- Class
- Object
- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

Classes and Objects

A class is a user-defined blueprint or prototype from which objects are created. Classes provide a means of bundling data and functionality together. Creating a new class creates a new type of object, allowing new instances of that type to be made. Each class instance can have attributes attached to it to maintain its state. Class instances can also have methods (defined by their class) for modifying their state.

The class creates a user-defined data structure, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object.

An object is an instance of a class. A class is like a blueprint while an instance is a copy of the class with actual values. An object consists of the following.

- State : It is represented by the attributes of an object. It also reflects the properties of an object.
- Behavior : It is represented by the methods of an object. It also reflects the response of an object to other objects.
- Identity : It gives a unique name to an object and enables one object to interact with other objects.

The principles of OOPS are based on 4 pillars apart from class and objects.

Abstraction

- Abstraction means displaying only essential information and hiding the details.
- Example : Car will accelerate, but internal working of engine not exposed.

Encapsulation

- Bundling of data into a single unit
- Bundling of all the methods that can act on an object of that data
- Car can have colors, brake, engine and a lot of other methods.

Inheritance

- When a class derives from another class
- Any new car company can inherit all the information from car class

Polymorphism

- The word 'polymorphism' means having many forms
- Same class method can work differently for different objects.

Creating classes

- It's a blueprint to create objects
- The class keyword is used to create classes
- By general convention, we start name with Capital letter
- Example : int, float, string, etc.

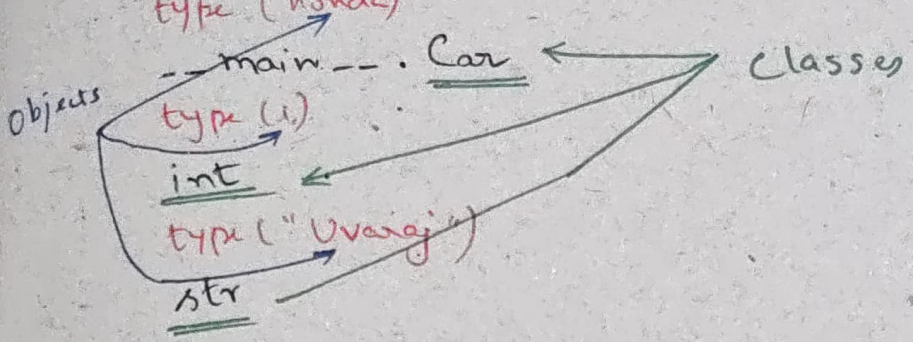
```
class Car :
    pass
```

Objects

- objects are instances or entities of a class
- It has all the properties of its class
- Example : 1, 2 are objects of int class.

honda = Car()

type(honda)



Constructor

- A constructor is a special method used to create and initialize an object of a class
- This method is defined in the class
- The constructor is executed automatically at the time of object creation

```
class Human:
    # I want some properties to be with every Human objects
    def __init__(self):
        print("Always execute")

uvaraj = Human()
Always execute
emma = Human()
Always execute
print(type(uvaraj))
<class '__main__.Human'>
```

Double Underscore (or) Dunder

```
class Human:
    def __init__(self, name, age, hobby):
        self.name = name
        self.age = age
        self.hobby = hobby

uvaraj = Human("Uvaraj", 35, "Travelling")
emma = Human("Emma", 32, "Acting")
```

```
uvaraj.name
'Uvaraj'
uvaraj.hobby
'Travelling'
emma.age
32
emma.hobby
'Acting'
```

Class Methods

- Apart from special methods, you can make your custom methods
- Example: Humans can speak, greet, etc.,

```
class Human:
    # constructor
    def __init__(self, name, age, hobby):
        # These are the attributes of an object
        self.name = name
        self.age = age
        self.hobby = hobby

    # Class Methods
    def greet(self):
        print(f"Hey, my name is {self.name}. Good morning!!")

uvaraj = Human("Uvaraj", 35, "Travelling")
uvaraj.name
'Uvaraj'
uvaraj.greet()
Hey, my name is Uvaraj. Good morning!!
emma = Human("Emma", 32, "Acting")
emma.greet()
Hey, my name is Emma. Good morning!!
```

```
uvaraj.nationality = "Indian"
uvaraj.nationality
'Indian'
emma.nationality
AttributeError: 'Human' object has no attribute 'nationality'
emma.hobby
Acting
emma.hobby = "Social Servie"
emma.hobby
'Social Servie'
uvaraj.hobby
'Travelling'
```

Class Variables

- These are common to the class
- Example: Population of Human class is common to all objects.

```
class Human:
    # class variables
    population = 0

    # Constructor
    def __init__(self, name, age):
        # Attributes of an object
        self.name = name
        self.age = age

        # increment population for every new Human object
        Human.population += 1

    # class methods
    def greet(self):
        print(f"Hey, my name is {self.name}. Good Eve!!")
```

```
Uvaraj = Human("Uvaraj", 35)
Human.population
1
Emma = Human("Emma", 32)
Human.population
2
```

```
class Human:
    # class variables
    population = 0
    data = []

    # constructor
    def __init__(self, name, age):
        # Attributes of an object
        self.name = name
        self.age = age
        # increment population for every new Human object
        Human.population += 1
        # append the name of every new Human object
        Human.data.append(self.name)

    # class methods
    def greet(self):
        print(f"Hey, My name is {self.name}. Good Eve!!")

Uvaraj = Human("Uvaraj", 35)
Emma = Human("Emma", 32)
Human.population
2
Human.data
['Uvaraj', 'Emma']
```

```
class Human :
    # class variables
    population = 0
    data = []
    # constructor
    def __init__(self, name, age, alive = True):
        # Attributes of an object
        self.name = name
        self.age = age
        self.alive = alive
        # increment population for every new Human object
        Human.population += 1
        # append the name of every new Human object
        Human.data.append(self.name)

    # class methods
    def greet(self):
        print(f"Hey, My name is {self.name}.
                                Good evening !!")

    def dead(self):
        if self.alive:
            print(self.name, "is no more now.")
            Human.population -= 1
            Human.data.remove(self.name)
            self.alive = False
        else:
            Print("This person is already dead")
```

```
h1 = Human("H1", 70)
h2 = Human("H2", 60)
h3 = Human("H3", 50)
h4 = Human("H4", 40)
Human.population
4
h3.dead()
H3 is no more now.
h3.dead()
This person is already dead.
```

```
Human.population
3
Human.data
['H1', 'H2', 'H4']
h5 = Human("H5", 30)
Human.population
4
Human.data
['H1', 'H2', 'H4', 'H5']
```

Inheritance

- When a class derives every attributes and methods from other class, it gets access to all its methods and attributes.
- This helps in reusability of code

```
# Base class
class Human :

    # class variables
    population = 0
    data = []

    # constructor
    def __init__(self, name, age, alive = True):

        # Attributes of an object
        self.name = name
        self.age = age
        self.alive = alive

        # increment population for every new Human object
        Human.population += 1

        # append the name of every new Human object
        Human.data.append(self.name)

    # class methods
    def greet(self):
        print(f"My name is {self.name}")

    def dead(self):
        if self.alive:
            print(self.name, "is no more now")
            Human.population -= 1
            Human.data.remove(self.name)
            self.alive = False
        else:
            print("This person is already dead")
```

```
# Derived class
class Employee (Human):
    pass

uvaraj = Human ("Uvaraj", 35)
emma = Human ("Emma", 32)
Human.population
2
mark = Employee ("Mark", 35)
Human.population
3
Human.data
['Uvaraj', 'Emma', 'Mark']
mark.greet()
My name is Mark
```

Adding Attributes in Derived class

```
class Employee (Human):
    # re-initiate constructor
    def __init__ (self, name, age, company, post):
        super().__init__(name, age)
        # attributes for employee class
        self.company = company
        self.post = post

uvaraj = Human ("Uvaraj", 35)
emma = Human ("Emma", 32)
Human.population
2
rohit = Employee("Rohit", 26, "Google", "CEO")
Human.population
3
rohit.post
'CEO'
rohit.company
'Google'
```

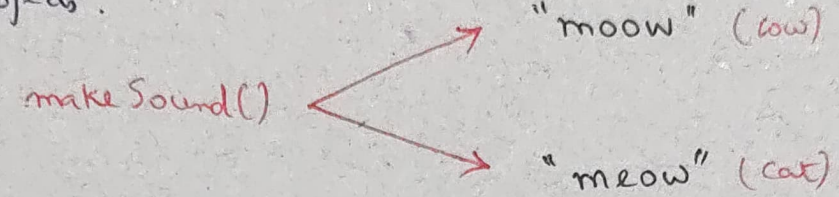
Adding Methods in Derived class

```
class Employee (Human):
    # re-initiate constructor
    def __init__(self, name, age, company, post):
        super().__init__(name, age)
        # attributes for employee class
        self.company = company
        self.post = post

    # Add some Attributes
    def hire(self, person):
        print(f"{person} has been hired in our company")
        Human.data.append(person)
        Human.population += 1

Uvaraj = Human("Uvaraj", 35)
emma = Human("Emma", 32)
Human.population
2
rohit = Employee("Rohit", 26, "Google", "CEO")
Human.population
3
rohit.post
'CEO'
Human.data
['Uvaraj', 'Emma', 'Rohit']
rohit.hire("Harry")
Harry has been hired in our company
Human.population
4
Human.data
['Uvaraj', 'Emma', 'Rohit', 'Harry']
ironman = Employee("IronMan", 40, "Stark", "MD")
Human.population
5
```

```
ironman.dead()
IronMan is no more now
Human.population
4
ironman.dead()
This person is already dead
```

Polymorphism

- The word 'polymorphism' means having many forms
- Same class method can work differently for different objects.



operator's level polymorphism

- Same operator can work differently for different objects

```
a = 2
b = 4
a + b
6
```

+ operator performs Addition

```
2 + 4
6
```

+ operator performs Concatenation

```
"2" + "4"
'24'
```

+ operator throws TypeError

```
2 + "4"
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

```
2.0 * 2
4.0
2 * 2
4
"uvaraj" * 3
'uvarajuvarajuvaraj'
"uvaraj" * "uvaraj"
```

* operator works differently for different objects.

TypeError: Can't multiply sequence by non-int of type 'str'

Function level Polymorphism

- Here, a same function behaves differently.

```
l = [1, 2, 3, 4]
len(l)
4
len("Vivaraj")
5
len(234)
TypeError : object of type 'int' has no len()
```

'len()' function behaves differently

```
sum(l)
10
sum(2324)
TypeError : 'int' object is not iterable
```

sum() function behaves differently

```
def mul(*args):
    total = 1
    for i in args:
        total *= i
    return total

mul()
1
mul(2)
2
mul(2, 3)
6
mul(2, 3, 4)
24
```

Custom function behaves differently

```
class Human:
    def speak(self, language):
        print("I speak", language)

h1 = Human()
h2 = Human()
h1.speak("Hindi")
I speak Hindi
h2.speak("Tamil")
I speak Tamil
h1.speak("English")
I speak English
```

Same method behaves differently.