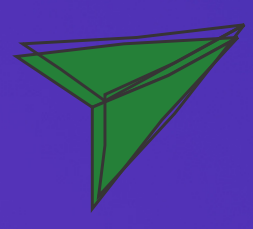
# OOPS

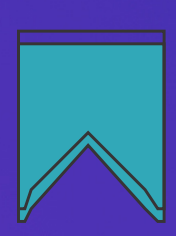
## USING

# PYTHON'

CREATED BY

UDAYABHANU NAYAK





# By Udayabhanu Nayak CSE Grad 2024

# OOPs using Python

## Class and Object

**class is a factory that produces objects ,It is a blueprint to create a object.**

**Object is the instance of a class**

## Class Attribute : shared by all instance of class
## obj._dict_

## Instance Attribute : belong to specific instance of a class
## class._dict_

In [1]:
```
class Emp:
    inc=1.5
    def __init__(self,name):
        self.name=name
    def increase(self):
        pass

obj = Emp('udaya')
print(obj.name,"  call by Object ")

print(Emp.inc,"  call by Class ")

print(obj.inc,"  call by Object ")
print(obj.increase(),"  call by Object ")
```

```
udaya   call by Object
1.5   call by Class
1.5   call by Object
None   call by Object
```

In [2]:
```
#Instance Attribute

obj.__dict__
```

Out[2]:
```
{'name': 'udaya'}
```

In [3]:
```
# Class Attribute

Emp.__dict__
```

Out[3]:
```
mappingproxy({'__module__': '__main__',
              'inc': 1.5,
              '__init__': <function __main__.Emp.__init__(self,
name)>,
              'increase': <function __main__.Emp.increase(self)
>,
              '__dict__': <attribute '__dict__' of 'Emp' object
s>,
              '__weakref__': <attribute '__weakref__' of 'Emp'
objects>,
              '__doc__': None})
```

# Methods

**Methods are functions that belongs to the class**

## Types of methods / Decorators

## 1. Class method

**do not need an instance to execute**

**use cls**

In [4]:
```
class class_method:
    no=2
    @classmethod
    def check(cls):
        print("Class method")

class_method.check()
```

```
Class method
```

### 1.1. As alternate constructor

In [5]:
```
class alternate_constructor:
    def __init__(self, name):
        self.name = name

    @classmethod
    def check(cls, age):
        return age

# Create an instance of the class
obj = alternate_constructor("Mansi")
print(obj)

# Call the check method by passing age as an argument
clss = alternate_constructor.check(21)
print(clss)
```

```
<__main__.alternate_constructor object at 0x000001B75565AF10>
21
```

### 1.2 Factory and Singletone pattern

In [ ]:
```
```

## 2. Static method

**act as regular functions but declared inside class**

**they don't operate on instance of class**

**no need to pass cls or self argument**

In [6]:
```
class static_method:
    x=3
    @staticmethod
    def checking(y):
        return y*static_method.x

static_method.checking(6)
```

Out[6]:
```
18
```

## 3. Instance method

### declared inside class

### use self, and bound to instance of class

In [7]:
```
class static_method:
    def __init__(self,name):
        self.name=name
    def checked(self):
        print(self.name)

obj=static_method('udaya')
obj.checked()
```

```
udaya
```

# 4 pillers of OOP

### 1. Inheritance

### 2. pollymorphism

### 3. Encapsulation

### 4. Abstraction

# Inheritance

### Inheriting attributes and methods from base class to a derived class called inheritance

## 1.1 Single Inheritance

### Inheriting from single parent class

In [8]:
```
class Animal:
    def __init__(self,types):
        self.types=types
    def sound(self):
        pass
class dog(Animal):
    def name(self):
        print(self.types)

obj = dog("Dogesh ")
obj.name()
```

```
Dogesh
```

## 1.2 Multiple Inheritance

### One sub class Inheriting from multiple parent class

In [9]:
```
class Animal:
    def __init__(self,name):
        self.name=name
    def swim(self):
        print("Animal can't swim ")
class Flies:
    def __init__(self,name):
        self.name=name
    def fly(self):
        print("Flies can fly ")
class check(Animal,Flies):
    pass

obj = check("kuku")
obj.swim()
```

```
Animal can't swim
```

## 1.2.1 Diamond Problem

### it arrises when a class inherits from two or more classes having same ansestors

In [10]:
```
class A:
    def method(self):
        print("Method from class A")

class B(A):
    def method(self):
        print("Method from class B")

class C(A):
    def method(self):
        print("Methos from class C")
class D(B,C):
    pass

obj = D()
obj.method()
```

```
Method from class B
```

## 1.3 Multilevel Inheritance

### Inheriting from parent class of parent class

### A-->B-->C

In [11]:
```
class A:
    def method(self):
        print("A")
class B(A):
    def method1(self):
        print("B")
class C(B):
    def method2(self):
        print("C")

obj=C()
obj.method()
```

```
A
```

## 1.4 Hierarchical Inheritance

### multiple sub classes inheriting from same parent class

### A-->B A-->C

In [12]:
```
class A:
    def method(self):
        print("A")
class B(A):
    def check(self):
        pass
class C(A):
    def checking(self):
        pass

obj = B()
print(obj.method())

obj2 = C()
print(obj2.method())
```

```
A
None
A
None
```

## 1.4.1 MRO ( Method resolution order )

In [13]:
```
class A:
    pass

class B(A):
    pass

class C(A):
    pass

class D(B, C):
    pass

print(D.mro())

obj = D()
obj
```

```
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.
C'>, <class '__main__.A'>, <class 'object'>]
```

Out[13]:
```
<__main__.D at 0x1b7556844d0>
```

## 1.4.2 Dunder Method

**Dunder methods, short for "double underscore methods," are special methods in Python that are surrounded by double underscores (e.g., init, repr).**

**These methods provide functionality to customize the behavior of objects in various ways,**

**such as operator overloading, object initialization, representation, and more.**

In [14]:
```
class MyClass:
    def __init__(self, value):
        self.value = value

    def __repr__(self):
        return f"MyClass({self.value})"

    def __add__(self, other):
        return MyClass(self.value + other.value)

    def __eq__(self, other):
        return self.value == other.value

obj1 = MyClass(5)
obj2 = MyClass(10)

print(obj1)
print(obj1 + obj2)
print(obj1 == obj2)
```

```
MyClass(5)
MyClass(15)
False
```

# 2. Pollymorphism

**pollymorphism means ability to take various forms**

**Same object having different behavious**

**print(5+5) --> 10**

**print('5'+'5') --> '55'**

**object is 5 but have different forms**

# Types

## 2.1 Compile Time Pollymorphism

**resolved during compile time**

**It is also known as static or early binding**

### 2.1.1 Method Overloading

**when a class has multiple methods having same name and different parameters**

**Python doesn't support it**

In [15]:
```
class MO:
    def add(self,a,b):
        return a+b
    def add(self,a,b,c):
        return a+b+c

obj = MO()
print(obj.add(1,2,3))
print(obj.add(1,3))
```

```
6

---------------------------------------------------------------
------------
TypeError                                 Traceback (most recen
t call last)
Cell In[15], line 9
      7 obj = MO()
      8 print(obj.add(1,2,3))
----> 9 print(obj.add(1,3))

TypeError: MO.add() missing 1 required positional argument: 'c'
```

### 2.1.2 Operator Overloading

**Operator overloading is a feature in Python that allows you to define custom behavior for operators when applied to objects of user-defined classes.**

**This means you can use standard operators such as +, -, *, /, ==, <, etc., with objects of your own classes**

In [16]:
```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    def __sub__(self, other):
        return Vector(self.x - other.x, self.y - other.y)

    def __mul__(self, scalar):
        return Vector(self.x * scalar, self.y * scalar)

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y


v1 = Vector(2, 3)
v2 = Vector(4, 5)


result_add = v1 + v2
result_sub = v1 - v2
result_mul = v1 * 3
print(result_add)
print(result_sub)
print(result_mul)
print(v1 == v2)
```

```
<__main__.Vector object at 0x000001B755B86B90>
<__main__.Vector object at 0x000001B755B84650>
<__main__.Vector object at 0x000001B755B87D10>
False
```

## 2.2 Run Time Pollymorphism

**Known as late binding**

**it is resolved during runtime**

## 2.2.1 Method Overridding

**When we write methods name with same signature in parent and child class called `Method Overridding` .**

In [17]:
```
class A:
    def show(self):
        return 'A'
class B(A):
    def show(self):
        return 'B'

obj = B()
obj.show()
```

Out[17]: 'B'

### 2.2.1.1 Super keyword

**used to deal with method overridding**

In [18]:
```
class A:
    def show(self):
        return 'A'

class B(A):
    def show(self):
        result_from_A = super().show()
        return f'{result_from_A} B'
obj = B()
print(obj.show())
```

```
A B
```

# 3.Encapsulation

**Python provides access to all variable and methods globally**

**By using encapsulation we can restrict the variable and methods access globally by making private and protected**

**__ protected**

**_ private**

In [19]:
```
class A:
    _a=10 # protected variable
    __b=20 # can't use outside class as protected
    def show(self):
        pass
class B(A):
    def check(self):
        print(self._a)

obj = B()
print(obj._a)
print(obj.check())
```

```
10
10
None
```

**N.B : In python allmost all are Encapsulated**

# 4. Abstraction

**It is a process of hiding implementation action details of object and expose relavent features or behaviour to the user .**

In [ ]:

## 4.1 Abstract class

**A class is called abstract class if it has one or more abstract methods .**

**Obj of abstract class can't created**

**abc module used to work with it**

**@abstractmethod**

In [20]:
```
from abc import ABC, abstractmethod

class car(ABC):
    def show(self):
        print("Let's show ")

    @abstractmethod
    def speed(self):
        pass

class maruti(car):
    def speed(self):
        print("Maruti")

obj = maruti()
obj.show()
obj.speed()
```

```
Let's show
Maruti
```

# Thank you from Udaya