OBJECT ORIENTED PROGRAMMING in **Python**

Things to learn about,

* Classes
* Objects
* Attributes
* Methods
* Self
* STATIC METHOD
* Constructor
  + \_\_new\_\_()
  + \_\_init\_\_()
* Pillars of OOP
  + Encapsulation
    - Data Binding
    - Data Hiding
    - Access Specifiers
    - Getters
    - Setters
  + Abstraction
    - Metaclass
  + Inheritance
    - Single inheritance
    - Multiple inheritance
    - Multilevel inheritance
    - Hierarchical inheritance
    - Hybrid inheritance
    - Super()
  + Polymorphism

**Class**

A Class is a template or a Blueprint or a skeleton of an Object

class Animal:

   pass

It is advisable to name a class Capitalized.

**Object : Anything that has (properties and some behaviours) n-number of functions or variables or both**

In Python almost everything is an Object, it is created from a Class, carbon copies of a class. By the process of **object instantiation. That is an instance of the class.**

tiger=Animal()

Here we created an Object Called tiger from the class Animal, This Tiger has all the properties of the Class Animal

You can create n-number of objects from a single class

**Attributes**

An Object can have values or information or variables with data, the data can either be assigned to them, and can also be passed through class functions called **Instance Attributes.**

The value can be used by 🡪 object.attribute 🡪 The Object name followed by Dot notation and the name of the attribute.

tiger=Animal()

tiger.color='orange'

print(tiger.color)

orange

or given to them by the class. That is an **Class Attribute.**

class Animal:

    thing='Living Organism'

tiger=Animal()

print(tiger.thing)

Living Organism

The Attributes of an Object and the Attribute of a Class is Mutable

class Animal:

    thing='Living Organism'

tiger=Animal()

print(tiger.thing)

tiger.thing='inorganic'

print(tiger.thing)

print(Animal.thing)

Animal.thing='Organic'

print(Animal.thing)

Living Organism

inorganic

Living Organism

Organic

Now things are going to get a little interesting, Changing the Value of the Class Attribute changes the attribute across all the objects of that class

class Animal:

    thing='Living Organism'

tiger=Animal()

print(tiger.thing)

Animal.thing='Organic'

print(tiger.thing)

Living Organism

Organic

But, when we change the value of the Object’s attribute by assigning it a value, Even if we change the value of the Class Attribute, the object’s attribute remains the same.

class Animal:

    thing='Living Organism'

tiger=Animal()

tiger.thing='ELECTRIC'

print(tiger.thing)

Animal.thing='Organic'

print(tiger.thing)

ELECTRIC

ELECTRIC

Now we do some reference semantics for two classes, if I change the value of one attribute, it changes the value of the attribute in the other class.

class red:

    color='red'

class blue:

    color='blue'

blue=red

print(blue.color)

red.color='green'

print(blue.color)

blue.color='yellow'

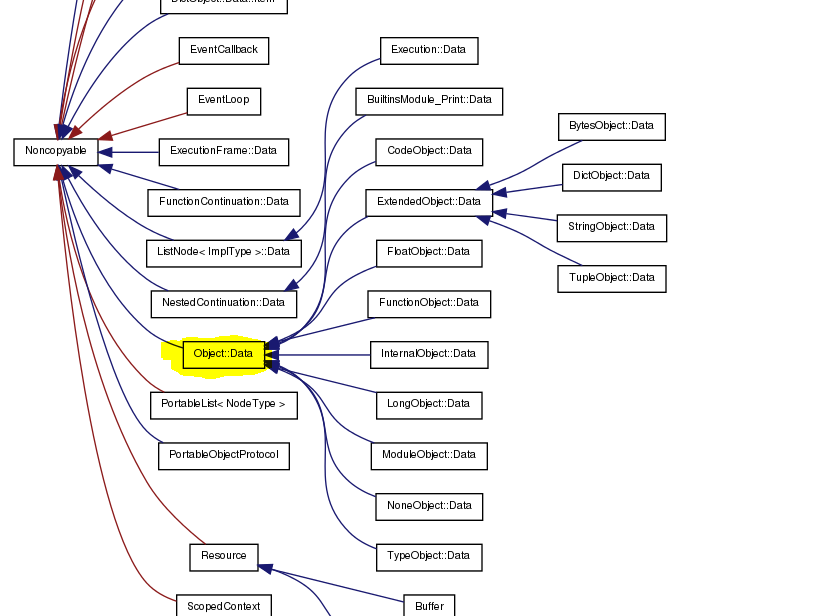
print(red.color)

red

green

yellow

For properties that should have a similar value per instance of a class, use class attributes. For properties that differ per instance, use instance attributes.

**Object**, The Object is a Global Variable. That is the root class for all data types.  


**Methods**

The Methods in Python are the same thing as the methods used by the Data Types. Every Data type in Python is a Class and the methods are it’s Class Methods.

Methods are defined and can be given to an object by the class, The methods are function Definitions inside a Class called **Instance Methods.**

class Animal:

    def greet(*self*):

        print('Hello')

#self will be explained

example:

class Animal:

    health='Good'

    def injury(*self*):

*self*.health='Poor'

#self will be explained

tiger=Animal()

print(tiger.health)

tiger.injury()

print(tiger.health)

Good

Poor

Passing arguments for the object methods,

class Animal:

    name='Living Organism'

    def greet(*self*,*color*,*size*):

        print('Hello',*size*,*color*,*self*.name,'')

tiger=Animal()

tiger.greet('orange','large')

Hello large orange Living Organism

NOTE:

Unlike attributes, Methods can be only given by Class and cannot be redefined, but can be assigned another class with the New Method

class Animal:

    thing='Living Organism'

    health='Good'

    def injury(*self*):

*self*.health='Poor'

tiger=Animal()

print(tiger.health)

tiger.injury()

print(tiger.health)

class Animal:

    thing='Living Organism'

    health='Good'

    def injury(*self*):

*self*.health='INJURED'

tiger=Animal()

print(tiger.health)

tiger.injury()

print(tiger.health)

Good

Poor

Good

INJURED

**Self**

Self is the First Argument to pass inside a Function Definition inside a Class.

class Animal:

    def greet(*self*):

        print('Hello')

Inside a function definition of a class, Self should be prefixed with a Dot notation to Access an attribute of the same Class

class Animal:

    name='Living Organism'

    def greet(*self*):

        print('Hello',*self*.name)

tiger=Animal()

tiger.greet()

Hello Living Organism

BUT why Self ?

class Animal:

    def greet(*self*):

        print('Hello')

tiger=Animal()

tiger.greet()

In the above example, you would have noticed that the Function greet has an Argument – self , but in the object methods there were no arguments passed inside the parentheses and no error was raised.

But if you leave the Argument- Self from the Class function, it returns an error.

class Animal:

    def greet(): #🡪 Removed the self and not going to give any Argument

        print('Hello')

tiger=Animal()

tiger.greet()

Traceback (most recent call last):

  File "d:\Dinesh\OOP & OOD\OOP\_in\_Python.py", line 30, in <module>

    tiger.greet()

TypeError: Animal.greet() takes 0 positional arguments but 1 was given

If you look at the Message, it says the greet function in the class Animal takes Zero arguments,

But 1 was given.

Consider calling method greet of the object tiger, that gives a string with the attributes name and color of the object tiger.

class Animal:

    def greet(*self*):

        print('Hello',*self*.color,*self*.name)

tiger=Animal()

tiger.name='tiger'

tiger.color='orange'

tiger.greet()

Hello orange tiger

The Class Function greet is being called when the object method greet gets called in the code,

So the Object tiger is Directly passed as a Variable to the self-argument in the class Animal’s function greet().

class Animal:

    def greet(*self*):

        print('Hello',*self*.color,*self*.name)

tiger=Animal()

tiger.name='tiger'

tiger.color='orange'

tiger.greet() **🡪** Animal.greet(tiger) 🡪 (‘Hello’,tiger.color,tiger.name)

The above representation clearly states that the Object is being passed as a variable to the class function.

**STATIC METHODS**: Methods without self. That can be called without creating a variable for that class

class Selfless:

    @staticmethod

    def stat\_meth():

        print("Look no self was passed")

Selfless.stat\_meth()

Look no self was passed

This is done using the Decorator @staticmethod

Or by passing the Object root class.

class Selfless(object):

    def stat\_meth():

        print("Look no self was passed")

Selfless.stat\_meth()

Look no self was passed

Or use the staticmethod(), builtin function to call a function from a class without self

class Selfless:

    def no\_self():

        print('Look No self was passed')

stat\_meth = staticmethod(Selfless.no\_self)

stat\_meth()

Look No self was passed

**CONSTRUCTOR**

**\_\_new\_\_()** is the Constructor, that is automatically passed when creating an Object.

class Animal:

    def \_\_new\_\_(*cls*,\**args*,\*\**kwargs*):

        print("new method")

        print(*cls*)

        print(*args*)

        print(*kwargs*)

# create our object and return it

        obj = super().\_\_new\_\_(*cls*)

        return obj

tiger=Animal()

This is Automatically passed, when creating an object.

new method

<class '\_\_main\_\_.Animal'>

()

{}

Remember \_\_init\_\_() is not a constructor.

In order to pass data as arguments for an object while creating it using a class other than the existing class attributes,

class Animal:

    living=True

    locomotion=True

    def \_\_init\_\_(*self*,*spec*,*food*,*color*):

*self*.species=*spec*

*self*.diet=*food*

*self*.color=*color*

tiger=Animal('Panthera','Carnivore','Orange')

print(tiger.living)

print(tiger.locomotion)

print(tiger.species)

print(tiger.diet)

print(tiger.color)

True

True

Panthera

Carnivore

Orange

Using the **\_\_init\_\_()** function defined in the class Animal, The Object tiger was given the attributes by passing the arguments in the parameter of the class.

print(dir(tiger))

['\_\_class\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_le\_\_', '\_\_lt\_\_', '\_\_module\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', '\_\_weakref\_\_', 'color', 'diet', 'living', 'locomotion', 'species']

**Encapsulation**

**Clumping or Bundling, Data and methods in a class. And providing data to the Object of the Class**

( Data Binding ), **meanwhile not permitting other objects to access for the data** (Data Hiding ).

Data Binding:

The application of value or reference semantics in a program

Static Data Binding: The act of data given from the class to the object  
Dynamic Data Binding: The act of passing data as an argument in either class using \_\_init\_\_ or as an argument in class method

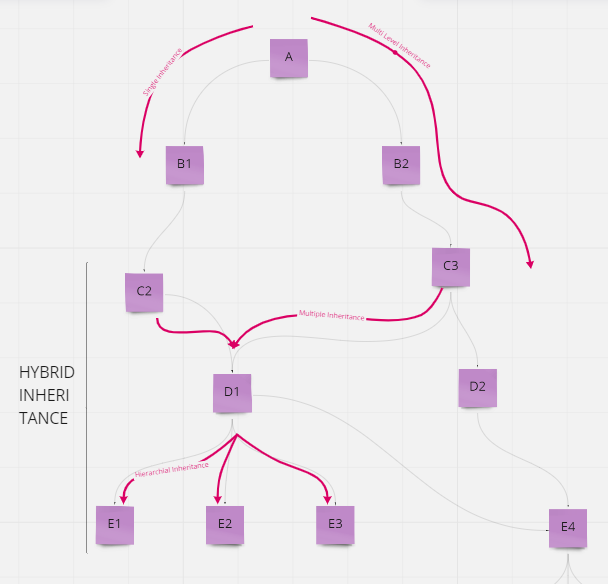
Data Hiding:

The act of isolating the members of a class or isolating the user from a part of program implementation.

**Inheritance**

The concept of inheriting Class attributes (Properties) and Class Functions (Methods) from one class (parent) by another class (child) is called Inheritance.

deriving a new class from an existing class



**Single Inheritance**

class Animal:       #Parent Class

    walks='yes'

    def walk(*self*,*name*):

*self*.name=*name*

        return f'The {*self*.name} is walking'

class Reptile(Animal):  #Parent Class passed as an argument for class Reptile

    eats='bugs'

lizard=Reptile()

print(lizard.eats)

print(lizard.walks)

print(lizard.walk('lizard'))

bugs

yes

The lizard is walking

The Lizard is an Object Instantiated from the Class Reptile, but the lizard calls the Instance Method lizard.walk() which was created as a function in the Class Animal. Thus the Object Instantiated from a Class can access the properties and methods from the Parent Class of the Class the object was Instantiated from.

**Multiple Inheritance**

More than 1 parent Class with 1 child Class

class parent\_1:

    pass

class parent\_2:

    pass

class child(parent\_1,parent\_2):

    pass

obj = child()

class Brands:               #parent\_class

    brand\_name\_1 = "Amazon"

    brand\_name\_2 = "Ebay"

    brand\_name\_3 = "OLX"

class Products:            #child\_class

    prod\_1 = "Online Ecommerce Store"

    prod\_2 = "Online Store"

    prod\_3 = "Online Buy Sell Store"

class Popularity(Brands,Products):

    prod\_1\_popularity = 100

    prod\_2\_popularity = 70

    prod\_3\_popularity = 60

obj\_1 = Popularity()          #Object\_creation

print(obj\_1.brand\_name\_1+" is an "+obj\_1.prod\_1)

print(obj\_1.brand\_name\_2+" is an "+obj\_1.prod\_2)

print(obj\_1.brand\_name\_3+" is an "+obj\_1.prod\_3)

Amazon is an Online Ecommerce Store

Ebay is an Online Store

OLX is an Online Buy Sell Store

**Multilevel Inheritance**

The Class properties and methods Inherited from a parent are properties methods that are the Class Attributes and Class Functions from another Parent Class.

class A:

    pass

class B(A):

    pass

class C(B):

    pass

obj = C()

**Hierarchical Inheritance**

**Hybrid Inheritance**

**super()**

The function super() g