4Py-OOP

1. Properties/ attributes – units inside objects / nature of the object

2. Methods – functions inside objects/ what it can do

# 4 pillars of object oriented programming

polymorphism, encapsulation, Inheritance, Abstraction.

Encapsulation : Concept where, attributes inside object instead of passing them as parameters. Reduce complexity, increase re-usablity

Abstraction : Technique where, An object with a few little properties instead of An Object with too many attributes, reduce impact of change, simpler interface.

Inheritance : Mechanism, DRY . Do NOT REPEAT YOURSELF.

Polymorphism : Technique, implementing methods within objects to make them unique.

++++++++++++

Classes are used to make templates or skeletons,  
A class called User is created

class User: /// classes are advised to Capitalize

+++++++++++

Create an Object called user\_1 from the class User,  
Create a variable and assign the class as a function.

class User:  
 pass

user\_1=User() /// user\_1 – object, User() – class

++++++++++++++++

/// Modules : write separate programs for the classes.

Now give attributes to the Object.  
attributes are variables with the Object name and a Dot notation used as Prefix assigned with a value, like -> object.attribute=value

class User:  
 pass

user\_1=User()

user\_1.name = ‘Duke’  
user\_1.id= ‘1’

SHELL

print(user1\_name)

OUTPUT  
Duke

++++++++++++++++

Creating another object from the same class.

class User:  
 pass

user\_1 = User()

user\_1.name = ‘Duke’  
user\_1.id= ‘1’

user\_2 = User()

user\_2.name = ‘Billy’  
user\_2.id = ‘2’

SHELL

print(user1\_name)  
print(user2\_name)

OUTPUT

Duke  
Billy

+++++++++++++++++

Instead of assigning attributes individually to each of the object, we can assign the attributes directly to the class

class User:  
 def \_\_init\_\_(self): /// METHOD, that constructs Objects, also known as THE CONSTRUCTOR  
 self.name = ‘Duke’  
 self.id=’1’

user\_1 = User()

/// now user\_1 has the name ‘Duke’ and id ‘1’

SHELL

print(user1\_name)

OUTPUT

Duke

+++++++++++++++++++

Now the previous example shows how the data from class is carried to the object. But to assign COMMON properties with UNIQUE values to different objects

When the object is being passed we can pass arguments respective to the class parameter.

class User:

    def \_\_init\_\_(*self*, *name*, *id*):

*self*.name = *name*

*self*.id = *id*

user\_1 = User('Duke','1')

user\_2 = User('Billy','2')

print(user\_1.name)

OUTPUT:

Duke

+++++++++++++++++++++++++

Now we are going to give both types of properties to the objects, 1. That is predefined in the class, and 2. That is Unique to every Object but is a common property.

class User:

    def \_\_init\_\_(*self*, *name*, *id*):

*self*.name = *name*

*self*.id = *id*

*self*.species = 'human'

user\_1 = User('Duke','1')

user\_2 = User('Billy','2')

print(user\_1.name)

print(user\_1.id)

print(user\_1.species)

print(user\_2.name)

print(user\_2.id)

print(user\_2.species)

OUTPUT:

Duke

1

human

Billy

2

human

++++++++++++++++++++++++++++++++++++++++++++

To import class to main file,

from file\_name import Class

object=Class()

++++++++++++++++++++++++++++++++++++++++++++

‘’’

We can create functions inside a class. Calling these functions as methods of a Class, which are normal methods like we use in python for specific data types. Since the Data types are actually Classes.

Here we create a class for An Army department,   
and a transfer method to transfer the units between multiple departments.

‘’’

class Arms:

    def \_\_init\_\_(*self*, *barracks*, *units*):

*self*.barracks= *barracks*

*self*.units= *units*

    def transfer(*self*, *department*, *recruits*):

*self*.units -= *recruits*

*department*.units += *recruits*

# WHICH IS ATUALLY,

class Class\_1:

    def \_\_init\_\_(*self*, *attribute\_1*, *attribute\_2*):

*self*.attribute\_1 = *attribute\_1*

*self*.attribute\_2 = *attribute\_2*

    def method\_1(*self*, *object\_placeholder*, *attribute\_effector*):

*self*.attribute\_1 -= *attribute\_effector*

*object\_placeholder*.attribute\_1 += *attribute\_effector*

Inside the Class, In method\_1, there are two syntax that affects two different Objects.

The first Syntax,

self.units -= recruits

the self refers to the object .

The second Syntax,

department.units += recruits

the department is the placeholder for the other object that is to be affected.

And While calling these functions

Here we create two Arms Departments(objects) border\_security and state\_gun\_squad with different barracks and recruits amount.

border\_security=Arms(2, 1000)

state\_gun\_squad=Arms(10, 50000)

Here we call the method to transfer a number of recruits from one department to another department (one object to another object)

state\_gun\_squad.transfer(border\_security, 10000)

We can check the number of units in each department by calling the specific attribute for units

print(border\_security.units)

print(state\_gun\_squad.units)

OUTPUT

11000

40000

Methods in the Objects can be called with or without arguments based on the object method definition,

For example

class Arms:

    def \_\_init\_\_(*self*, *barracks*, *units*):

*self*.barracks= *barracks*

*self*.units= *units*

IN HERE THERE IS A SELF BUT STILL, WE CAN CALL IT BY

    def shout(*self*):

        print('To Arms!')

We can call it by

air\_force=Arms()

air\_force.shout()

NOTE THE ABSENCE OF ARGUMENTS PASSED IN THE FUNCTION

OUTPUT

To Arms!

++++++++++++++++++++++++++++++++++++++++++++

Now we can create multiple objects using for loop

departments=['navy','guards','militia','sentry','air\_force']

#Since objects except functions/Classes do not have \_\_name\_\_ in Python.

#it maybe somewhat essential to give names to the objects created from a Class.

arms=[]

#The above list is Going to contain all the objects to be generated

#Class Creation

class Arms:

        def \_\_init\_\_(*self*, *name*, *barracks*, *units*):

*self*.name= *name*

*self*.barracks= *barracks*

*self*.units= *units*

        def seige(*self*, *losses*):

*self*.units -= *losses*

        def recruit(*self*, *recruits*):

*self*.units += *recruits*

        def shout(*self*):

                print('To Arms!')

        def transfer(*self*, *reception*, *recruits*):

                if *recruits*> *self*.units:

                        print('Transfer not possible, Not enough units')

                elif *self*.units > 0:

*self*.units -= *recruits*

*reception*.units += *recruits*

#Object Creation

for department in departments:

        print(department)

        department=Arms(department, input('Enter the no.of.barracks: '),input('Enter the no.of.units: '))

        arms.append(department)

Checking into further application

Class