**Object Oriented Programming**

Object Oriented Programming (OOP) tends to be one of the major obstacles for beginners when they are first starting to learn Python.

Even though Pyton is object oriented language it does not force the programmers to write programs in completed object oriented way Features of object oriented Programming system are

1. Classess and Objects
2. Encapsulation
3. Abstraction
4. Inheritence
5. Polymorphism

For this lesson we will construct our knowledge of OOP in Python by building on the following topics:

* Objects
* Using the *class* keyword
* Creating class attributes
* Creating methods in a class
* Learning about Inheritance
* Learning about Polymorphism
* Learning about Special Methods for classes

Remember how we could call methods on a list?

lst.count(2)

Out[2]:

1

What we will basically be doing in this lecture is exploring how we could create an Object type like a list. We've already learned about how to create functions. So let's explore Objects in general:

**Objects**

In Python, *everything is an object*. Remember from previous lectures we can use type() to check the type of object something is:

So we know all these things are objects, so how can we create our own Object types? That is where the class keyword comes in.

**class**

User defined objects are created using the class keyword. The class is a blueprint that defines the nature of a future object. From classes we can construct instances. An instance is a specific object created from a particular class. For example, above we created the object lst which was an instance of a list object.

In Python, a class definition has the following format Although you should note that you can mix the order of the definition of attributes, and methods as required within a single class.

class nameOfClass(SuperClass):

**init**

attributes

methods

Let see how we can use class:

*# Create a new object type called Sample*

c**++**

**class** Patientclass{

private:

string patname;

int patid

int patage; *#attributes*

string patdishistory

string consuldoc

public:

Patientclass(){} **--** constructor

Patientclass(int,int)

Patientclass(float,int)*#methods -constructor and initilise*

string get\_name(){}

int get\_age(){}

}

​

​

​

sampleobj **=** new Patientclass()

sampleobj.get\_name()

​

​

**class** Patient:

allocate the memory \_\_new\_\_() **+** initialzw **with** intial values \_\_init\_\_()

**def** \_\_init\_\_(self,patname,patage): **//**instance methods \_\_new\_\_ **+**\_\_init\_\_ **=** construtor

self.name **=** patname **//**instance variables

self.age **=** patage

**def** display(self): **///**instan**--**ce methode

print(self.name,self.age)

​

patobj1 **=** Patient("ABC",30) **--->**Patinet(**&**patobj1,"ABC",30)

patobj2 **=** Patinet("XYZ",40) **--->**Patient(**&**Patobj2,"XYZ",40)

​

patobj1.display()

patobj2.display()

​

​

**class** Patient(object):

*#instance variables and instance methods*

*#attributes*

*#instance methods*

**def** \_\_init\_\_(self,name,age,gen,heathstat): *#\_\_init\_\_ is a constructor - \_\_new\_\_ + \_\_init\_\_ \_\_doc\_\_*

self.patname **=** name

self.age **=** age

self.gender **=** gen

self.healthstaus**=** healthstat

self.specialfeature **=** ""

**def** operation(self,surgery**=** address):

self.operation **=** heartsurgery

**pass**

**def** generalchecking(self):

self.checking **=** true

​

PatientObj1 **=** Patient("XYX",30,"Male",poor)

1000 **->** PatientObj1 **=** Patient("XYX",30,"Male",poor) **--->** Patient.operation(PatientObj1)

PatientObj1.specialfeature **=** "swimmer"

print(PatientObj1.falseceiling) **=** wooden **pass**

​

PatientObj1.operation() **--->**Patient.operation(PatientObj1)

​

2000 **->** PatientObj2 **=** Patient("ABC",20,"Female",good)

*#PatientObj2.operation() -->Patient.operation(PatientObj1)*

PatientObj2.generalchecking()

PatientObj2.operation() **--->**Patient.operation(PatientObj2)

print(PatientObj2.specialfeature) error

​

​

​

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

**class** Patinet:

**def** \_\_init\_\_(self,patname,patage):

self.name **=**patname

self.age **=** patage

**def** display(self):

print(self.name,self.age)

Obj1 **=** Patinet("ABC",30)

Obj2 **=** Patinet("XYZ",40)

​

Obj1.display()

Obj2.display()

​

ABC 30

XYZ 40

The following code is an example of a class definition: Although this is not a hard and fast rule, it is common to define a class in a file named after that class.

For example, the above code would be stored in a file called Sample.py; this makes it easier to find the code associated with a class.

By convention we give classes a name that starts with a capital letter. Note how x is now the reference to our new instance of a Sample class. In other words, we **instantiate** the Sample class.

Inside of the class we currently just have pass. But we can define class attributes and methods.

An **attribute** is a characteristic of an object. A **method** is an operation we can perform with the object.

For example, we can create a class called Dog. An attribute of a dog may be its breed or its name, while a method of a dog may be defined by a .bark() method which returns a sound.

**Attributes**

The syntax for creating an attribute is:

self.attribute = something

There is a special method called:

\_\_init\_\_()

This method is used to initialize the attributes of an object. For example:

*#Class - is a blue print for creating instance of the class*

*#Instance - each unique object that we create using our class will be an instance of that class*

**class** Employee:

**def** \_\_init\_\_(self,first,last,pay): *#initializer in python special methods dunder methods*

self.firstname **=** first *#attributes*

self.lastname **=** last

self.pay **=** pay *#can be same as that ofthe arguments*

**def** fullnamedisplay(self): *# methods - here self is emp\_1 ,emp\_2 so calling of self.firstname is equal to emp\_1.firstname etc*

print('{} {} has the pay of {}'.format(self.firstname ,self.lastname,self.pay))

​

​

​

*#To initilise values for each object then for each object one by one we need to set the values*

*# emp\_1 = Employee()*

*# emp\_2 = Employee()*

​

​

​

*#if many objects are there then storing indivually will not serve the purpose of the class*

*# emp\_1.first ='User1firstname'*

*# emp\_1.last = 'User1lastname'*

*# emp\_1.pay = 1000*

*#instead of printing for all the objects seperately , we can have one common method fullname , were self will be*

*#each object passed*

​

*# print('{} {} has the pay of {}'.format(emp\_1.firstname ,emp\_1.lastname,emp\_1.pay))*

*# print('{} {} has the pay of {}'.format(emp\_2.firstname ,emp\_2.lastname,emp\_2.pay))*

​

​

​

*# instead use \_\_init\_\_ method to initilise the value for each object created*

​

emp\_1 **=** Employee("ASha","Talari",1000)

*# emp\_1.fullnamedisplay() #-->Employee.fullname(emp\_1)*

emp\_2 **=** Employee("Dheeraj","Mahendiran",2000)

*# emp\_2.fullnamedisplay() #-->Employee.fullname(emp\_2)*

emp\_3 **=** Employee("Mmonica","Bedi",3000)

emp\_2.fullnamedisplay()

​

​

​

*# print(emp\_1)*

*# print(emp\_2) # are objects with different memory address*

​

​

​

*# we are not passing the self value explicity and it carries the memory address of the object invoked*

*#it is equivivalent to Classname.fullname(object)*

*# emp\_1.fullname() #->Employee.fullname(emp\_1)*

*# emp\_2.fullname() #->Employee.fullname(emp\_2)*

​

*# Employee.fullname(emp\_1)*

​

*#if self is not used in the function then it will not know on which object it needs to work and it will be expecting one argument*

*#that is the address of the object as the below st is equvilent to Employee.fullname(emp\_1)*

*#so we are passing one argument but in defination we are specifying that it takes argument hence error*

*# emp\_1.fullname()*

Mmonica Bedi has the pay of 3000

**class** Dog:

**def** \_\_init\_\_(self,breed): *#initilizerr of the values*

*#constructor = \_\_new\_\_ + \_\_init\_\_*

self.breed **=** breed

**def** display(self):

print("Breed is {}".format(self.breed))

sam **=** Dog(breed **=**'Lab')

sam.display()

print(sam.breed)

print(sam)

​

frank **=** Dog(breed**=**'Huskie')

frank.display()

​

​

Breed is Lab

Lab

<\_\_main\_\_.Dog object at 0x06620170>

Breed is Huskie

Lets break down what we have above.The special method

\_\_init\_\_()

is called automatically right after the object has been created:

def \_\_init\_\_(self, breed):

Each attribute in a class definition begins with a reference to the instance object. It is by convention named self. The breed is the argument. The value is passed during the class instantiation.

self.breed = breed

Now we have created two instances of the Dog class. With two breed types, we can then access these attributes like this:

sam.breed

Out[6]:

'Lab'

frank.breed

Out[7]:

'Huskie'

Note how we don't have any parentheses after breed; this is because it is an attribute and doesn't take any arguments.

In Python there are also *class object attributes*. These Class Object Attributes are the same for any instance of the class. For example, we could create the attribute *species* for the Dog class. Dogs, regardless of their breed, name, or other attributes, will always be mammals. We apply this logic in the following manner:

**class** Dog:

*# Class Object Attribute*

species **=** 'mammal' *#class atrribute*

**def** \_\_init\_\_(self,breed,name): *#instance methods*

self.breedname **=** breed *#instance variables*

self.dogname **=** name

sam **=** Dog('Lab','Sam')

​

frank **=** Dog("some","new")

sam.breedname

sam.dogname

​

print(frank.breedname)

print(frank.dogname)

some

new

Note that the Class Object Attribute is defined outside of any methods in the class. Also by convention, we place them first before the init.

sam.species **=**'Human' *#changed the class variable at instance level , so only in that instance the change is seen*

*#if it is changed at class level then all the instances can see the changes*

print(sam.species)

*# frank.species = 'Human'*

print(frank.species)

Human

Human

*#Assignment*

p1 **=** Dog('Lab', 'john')

p2 **=** Dog('Nab','henry')

px **=** p1 *#here the address holded by p1 is given to px , so both p1 and px will be pointing to same object*

​

print(id(p1))

print(id(px))

p1**=** p2 *#if we assign p1 to some other object then p1 will point to new object where as px remain unaffcted*

print(id(p1))

print(id(p2))

print(id(px))

15098416

15098416

15100080

15100080

15098416

*#if we print the objects*

print(p1)

print(p2)

​

<\_\_main\_\_.Dog object at 0x00E665B0>

<\_\_main\_\_.Dog object at 0x00E665B0>

What this is showing is the name of the class (in this case Dog) and a hexadecimal number indicates where it is held in memory. Neither of which is particularly useful and certainly doesn’t help us in knowing what information p1 and p2 are holding

​

**Methods**

Methods are functions defined inside the body of a class. They are used to perform operations with the attributes of our objects. Methods are a key concept of the OOP paradigm. They are essential to dividing responsibilities in programming, especially in large applications.

You can basically think of methods as functions acting on an Object that take the Object itself into account through its *self* argument.

Let's go through an example of creating a Circle class:

**class** Circle:

pi **=** 3.14 *#class variable*

​

*# Circle gets instantiated with a radius (default is 1)*

**def** \_\_init\_\_(self, radius**=**1): *#instance methods*

self.radius **=** radius *#instance attributes*

self.area **=** radius **\*** radius **\*** Circle.pi

*# i=10*

*# print(i)*

​

*# Method for resetting Radius*

**def** setRadius(self, new\_radius): *#instance methods*

self.radius **=** new\_radius

self.area **=** new\_radius **\*** new\_radius **\*** self.pi

​

*# Method for getting Circumference*

**def** getCircumference(self): *#instance methods*

print(self.radius)

**return** self.radius **\*** self.pi **\*** 2

**def** displayatclasslevel(self):

Circle.pi **=** 4.14

print("The value of pi in Class level is ",Circle.pi)

**def** displayatinstancelevel(self):

self.pi **=** 5.14

print("The value of pi in instance level",self.pi)

**def** display(self):

print("The value of pi",self.pi)

​

​

​

c1 **=** Circle()

*# c1.diameter = 10*

​

c1.displayatclasslevel()

c1.display()

c1.displayatinstancelevel()

c1.display()

​

c2 **=** Circle()

c2.display()

print(c1.diameter)

*# print(c2.diameter)*

​

​

​

*# c3 = Circle()*

*# c3.display()*

​

*# print('Radius is: ',c1.radius)*

*# print('Area is: ',c1.area)*

*# print('Circumference is: ',c1.getCircumference())*

The value of pi in Class level is 4.14

The value of pi 4.14

The value of pi in instance level 5.14

The value of pi 5.14

The value of pi 4.14

10

pi **=** 3.14 className.pi **=**4.14 **--** classlevel

obj1**->**radius**=**10 obj2**->**radius**=**30 obj3**->**radius**=**10

print(obj3.self.pi)**=**4.14

print(obj1**->**self.pi)obj2**->**self.pi**=**5.14

print(obj2.self.pi)5.14

4.14

In the \_*init\_* method above, in order to calculate the area attribute, we had to call Circle.pi. This is because the object does not yet have its own .pi attribute, so we call the Class Object Attribute pi instead.  
In the setRadius method, however, we'll be working with an existing Circle object that does have its own pi attribute. Here we can use either Circle.pi or self.pi.  
  
Now let's change the radius and see how that affects our Circle object:

c.setRadius(2)

​

print('Radius is: ',c.radius)

print('Area is: ',c.area)

print('Circumference is: ',c.getCircumference())

Radius is: 2

Area is: 12.56

Circumference is: 12.56

print(dir(Circle))

['\_\_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\_\_', 'getCircumference', 'pi', 'setRadius']

*#class variable*

**class** Employee:

*#class variable , which can be accessed via classname or instance , its namespace will be through out the class*

*#if it is modified at the class namespace then its effect will be seen in all the instances , where as if it is*

*#modified at instance level then its change will not be seen at all instances*

raise\_amount **=** 1.04

no\_emp **=**0

**def** \_\_init\_\_(self,first,last,pay):

self.firstname **=** first *#attributes*

self.lastname **=** last

self.pay **=** pay *#can be same as that ofthe arguments*

*#self.no\_emp +=1 # increemnted every time when emp obj is created , if we use self. then the increment happens*

*# in that instance only and not at other ,*

Employee.no\_emp **+=**1

**def** fullname(self): *# methods -*

print('{} {} has the pay of {} {} {}'.format(self.firstname ,self.lastname,self.pay,Employee.raise\_amount,self.raise\_amount))

*#raise\_amount can not be accessed without instance or classname as it may give error*

**def** no\_employees(self):

print("The number of employees are {}".format(Employee.no\_emp)) *#So while accessing through instance ,*

*# we can see the change*

*# at instance level not at class level*

emp\_1 **=** Employee("User1firstname","User1lastname",1000)

emp\_2 **=** Employee("User2firstname","User2lastname",2000)

​

emp\_1.fullname()

emp\_2.no\_employees()

emp\_1.no\_employees()

User1firstname User1lastname has the pay of 1000 1.04 1.04

The number of employees are 2

The number of employees are 2

​

In the previous section we printed out information from the instances of class

Person by accessing the attributes name and age.

However, we now needed to know the internal structure of the class Person to

print out its details. That is, we need to know that there are attributes called name

and age available on this class.

It would be much more convenient if the object itself knew how to convert its

self into a string to be printed out!

In fact we can make the class Person do this by defining a method that can be

used to convert an object into a string for printing purposes.

This method is the \_\_str\_\_ method. The method is expected to return a string

which can be used to represent appropriate information about a class.

The signature of the method is

​

def \_\_str\_\_(self)

​

We can add this method to our class Person and see how that affects the output

generated when using the print() function.

We will return a string from the \_\_str\_\_ method that provides and the name

and age of the person:

​

**class** Person:

**def** \_\_init\_\_(self, name1, age1):

self.name **=** name1

self.age **=** age1

*# def display(self):*

*# print(self.name,self.age)*

**def** \_\_str\_\_(self):

**return** self.name **+** ' is ' **+** str(self.age)

*# print(dir(Person))*

Obj1 **=** Person("Asha",10)

​

Obj2 **=** Person("Dheeraj",20)

*# Obj1.display()*

*# Obj2.display()*

​

print(Obj1.age)

​

print(Obj1.name)

print(Obj1)

​

print(Obj2)

10

Asha

Asha is 10

Dheeraj is 20

Note that in the **str** method we access the name and age attributes using the self parameter passed into the method by Python. Also note that it is necessary to convert the age number attribute into a string. This is because the '+' operator will do string concatenation unless one of the operands (one of the sides of the '+') is a number; in which case it will try and do arithmetic addition which of course will not work if the other operand is a string if we try to print p1 and p2

p1 **=** Person('john',36)

p2 **=** Person('Henry',46)

print(p1)

print(p2)

​

​

john is 36

Henry is 46

*# print(Circle.\_\_dict\_\_)*

a**=**'10'

b**=**'20'

**if** a**>**b:

**pass**

print(dir(a))

['\_\_add\_\_', '\_\_class\_\_', '\_\_contains\_\_', '\_\_delattr\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_getitem\_\_', '\_\_getnewargs\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_iter\_\_', '\_\_le\_\_', '\_\_len\_\_', '\_\_lt\_\_', '\_\_mod\_\_', '\_\_mul\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_rmod\_\_', '\_\_rmul\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'format\_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']

*#Regular Methods*

*#class variables*

​

**class** Employee:

'''This i example to show class variables and intance members'''

raise\_amount **=** 1.04

no\_emp **=**0 *#class variables*

**def** \_\_init\_\_(self,first,last,pay):

self.firstname **=** first *#attributes*

self.lastname **=** last

self.pay **=** pay *#can be same as that ofthe arguments*

self.no\_emp **+=**1 *# increemnted every time when emp obj is created , if we use self. then the increment happens*

*# in that instance only and not at other ,*

*# Employee.no\_emp +=1*

**def** fullname(self): *# methods -*

print('{} {} has the pay of {} {} {}'.format(self.firstname ,self.lastname,self.pay,Employee.raise\_amount,self.raise\_amount))

*#raise\_amount can not be accessed without instance or classname as it may give error*

**def** no\_employees(self):

print("The number of employees are {}".format(self.no\_emp)) *#So while accessing through instance , we can see the change*

*# at instance level not at class level , here instead of self use classnmae Employee then it will be 0 and for each object it will be 1*

emp\_1 **=** Employee("User1firstname","User1lastname",1000)

emp\_2 **=** Employee("User2firstname","User2lastname",2000)

​

emp\_1.no\_employees()

​

​

The number of employees are 1

print(Employee.\_\_doc\_\_) *#to get the doument string included in the class*

This i example to show class variables and intance members

The del Keyword Having at one point created an object of some type (whether that is a bool, an int or a user defined type such as Person) it may later be necessary to delete that object. This can be done using the keyword del. This keyword is used to delete objects which allows the memory they are using to be reclaimed and used by other parts of your program.

p1 **=** Person('John', 36)

print(p1)

**del** p1

​

print(p1) *#after deleting if we try to print once again then it display as error*

John is 36

**---------------------------------------------------------------------------**

**NameError** Traceback (most recent call last)

**<ipython-input-61-b8cca1c454d3>** in <module>

3 **del** p1

4

**----> 5** print**(**p1**)** **#after deleting if we try to print once again then it display as error**

**NameError**: name 'p1' is not defined

After the del statement the object held by p1 will no longer be available and any attempt to reference it will generate an error. You do not need to use del as setting p1 above to the None value (representing nothingness) will have the same effect. In addition, if the above code was defined within a function or a method then p1 will cease to exist once the function or method terminates and this will again have the same effect as deleting the object and freeing up the memory.

Intrinsic Attributes

Every class (and every object) in Python has a set of intrinsic attributes set up by

​

the Python runtime system. Some of these intrinsic attributes are given below for

​

classes and objects.

​

Classes have the following intrinsic attributes:

  • \_\_name\_\_ the name of the class

  • \_\_module\_\_ the module (or library) from which it was loaded

  • \_\_bases\_\_ a collection of its base classes (see inheritance later in this course)

  • \_\_dict\_\_ a dictionary (a set of key-value pairs) containing all the attributes

(including methods)

  • \_\_doc\_\_ the documentation string.

For objects:

  • \_\_class\_\_ the name of the class of the object

  • \_\_dict\_\_ a dictionary containing all the object’s attributes.

Notice that these intrinsic attributes all start and end with a double underbar—

this indicates their special status within Python.

An example of printing these attributes out for the class Person and a instance

of the class are shown below

​

​

​

print('Class attributes')

print(Person.\_\_name\_\_)

print(Person.\_\_module\_\_)

print(Person.\_\_doc\_\_)

print(Person.\_\_dict\_\_)

print('Object attributes')

print(p1.\_\_class\_\_)

print(p1.\_\_dict\_\_)

print(p2.\_\_class\_\_)

print(p2.\_\_dict\_\_)

Class attributes

Person

\_\_main\_\_

None

{'\_\_module\_\_': '\_\_main\_\_', '\_\_init\_\_': <function Person.\_\_init\_\_ at 0x00E48E40>, '\_\_str\_\_': <function Person.\_\_str\_\_ at 0x00E48270>, '\_\_dict\_\_': <attribute '\_\_dict\_\_' of 'Person' objects>, '\_\_weakref\_\_': <attribute '\_\_weakref\_\_' of 'Person' objects>, '\_\_doc\_\_': None}

Object attributes

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

{'name': 'John', 'age': 36}

*#Types of variables*

​

​

*#instance variable*

*#if the values are modified at instance level then only the instances can see the changes*

**class** Sample:

**def** \_\_init\_\_(self):

self.x **=**10 *# instance vairables*

*#instance method*

**def** modify(self):

self.x**+=**1

*#create 2 instances*

s1 **=** Sample()

s2 **=** Sample()

print('x in s1 =',s1.x)

print('x in s2 =',s2.x)

​

*#after modifucation*

s1.modify()

​

print('x in s1 =',s1.x)

print('x in s2 =',s2.x)

x in s1 = 10

x in s2 = 10

x in s1 = 11

x in s2 = 10

In Python classes can also have attributes; these are referred to as class variables or attributes (as opposed to instance variables or attributes). In Python variables defined within the scope of the class, but outside of any methods, are tied to the class rather than to any instance and are thus class variables.

It is also possible to define behaviour that is linked to the class rather than an individual object; this behaviour is defined in a class method. Class methods are written in a similar manner to any other method but are decorated with @classmethod and take a first parameter which represents the class rather than an individual instance. This decoration is written before the method declaration. An example of a class method is shown below:

# Why Class-Side Methods?

It may at first seem unclear what should normally go in an instance method as

opposed to what should go in a class method. After all, they are both defined in the

class. However, it is important to remember that

• Instance methods define the behaviour of the instance or object.

• Class methods define the behaviour of the class.

Class-side methods should only perform one of the following roles:

• Instance creation This role is very important as it is how you can use a class as a

factory for objects and can help hide a whole load of set up and instantiation

work.

• Answering enquiries about the class This role can provide generally useful

objects, frequently derived from class variables. For example, they may return

the number of instances of this class that have been created.

• Instance management In this role, class-side methods control the number of

instances created. For example, a class may only allow a single instance of the

class to be created; this is termed a singleton class. Instance management

methods may also be used to access an instance (e.g. randomly or in a given

state).

• Examples Occasionally, class methods are used to provide helpful examples

which explain the operation of a class. This can be very good practice.

• Testing Class-side methods can be used to support the testing of an instance of a

class. You can use them to create an instance, perform an operation and compare

the result with a known value. If the values are different, the method can report

an error. This is a very useful way of providing regression tests.

• Support for one of the above roles.

Any other tasks should be performed by an instance method.

*#class variables*

*#class methods*

**class** Sample:

*# this is a class var*

x**=**10

​

*# class method builtin decorator , class method will have first parameter cls as default parameter*

*# which can access class variable*

*#to access the class variable outside the classwe can use classname.variable*

​

**def** \_\_init\_\_(self,number):

*#when ever initilization is needed init can be written*

self.number **=** number

self.y **=** number

*#this is class method*

@classmethod *#builtin decorator*

**def** modify(cls,number):

cls.x**+=**number

cls.y **=** number**+**10 *#instance variable modified at class level and the changes seen at all objects*

**def** modifyself(self): *#instance method*

self.x**+=**1 *#calss variable modified at instance level , change seen only at object level*

*#create 2 instance*

s1**=** Sample(10)

s2**=** Sample(20)

*# print(s2.number)*

print('x in s1 =',s1.x)

print('x in s2 =',s2.x)

print('y in s1 =',s1.y)

​

*#modify x in s1*

s1.modify(10) *#class method can be invoked either by class name or instance name*

print('x in s1 =',s1.x)

print('x in s2 =',s2.x)

print('y in s2 =',s2.y)

​

*# Sample.modifyself() #can not invoke instance methods with class name as instance methods needs address of the object*

​

*# Sample.modify() #can invoke class method with class name.*

s1.modifyself()

print('x in s1 =',s1.x)

print('x in s2 =',s2.x)

print('y in s2 =',s2.y)

​

x in s1 = 10

x in s2 = 10

y in s1 = 10

x in s1 = 20

x in s2 = 20

y in s2 = 20

x in s1 = 21

x in s2 = 20

x in s2 = 20

**def** add(n1,n2):

**def** wrap():

x**=**use()

x**=**x**+**10

**return** wrap

@add

**def** use():

n4 **=** n3**+**10

**return** n4

*#Namespaces*

*#modifying class variable at class namespace level*

**class** Sample:

*#this is class var*

n**=**10

*#access class var in the class namespace using class name*

print(Sample.n)

Sample.n**+=**1 *#modify class var in class namespace*

print(Sample.n)

​

*#modified value availble to all instances since it is modifed all class level*

s1 **=** Sample()

s2 **=** Sample()

print(s1.n)

print(s2.n)

10

11

11

11

*#Namespaces*

*#modifying at instance level*

**class** Sample:

*#this is class variable*

n**=**10

*#access class var in s1 instance variable*

s1**=**Sample()

s2 **=** Sample()

print(s1.n)

print(s2.n)

s1.n**+=**1 *#modifying at instance level*

print(s1.n)

print(s2.n) *#modification not availble , since only instance s1 is modified , even the variable is class variable but at what name space it is modified shows where we can see the output*

10

10

11

10

*#Instace Method - in this there are 2 types of methods one is accessor method called as getmethods , mutator methods called as*

*#set methods*

*#mutator and Accessor methods*

**class** Student:

*#mutator method*

**def** setName(self,name):

self.name **=** name

*#accessor method*

**def** getName(self):

**return** self.name

**def** setMarks(self,marks):

self.marks **=** marks

**def** getMarks(self):

**return** self.marks

​

n **=** int(input('How many students:'))

i**=**0

**while**(i**<**n):

*#create student instance*

s**=**Student()

name **=** input("Enter the name")

s.setName(name)

marks **=** int(input("Enter the marks"))

s.setMarks(marks)

print("The student name is",s.getName())

print("The marks are",s.getMarks())

i **=**i**+**1

print('--------------------')

​

How many students:2

Enter the namexxx

Enter the marks80

The student name is xxx

The marks are 80

--------------------

Enter the nameyyy

Enter the marks100

The student name is yyy

The marks are 100

--------------------

*#Class methods*

**class** Bird:

wings **=**2

*#this is a class method*

@classmethod

**def** fly(cls,name): *# the variables inside the class are accessed using cls instead of self*

cls.name **=** name *#directly name can be used with assigning it to cls varibale*

print('{} flies with {} wings'.format(cls.name,cls.wings))

**def** display(self):

print(self.wings)

*#display information*

Bird.fly('Sparrow')

Bird.fly('Pigeon') *#class methods can be accessed via class name*

b **=** Bird()

b.fly("Bird") *#Bird.fly(b)#class methods accessed via object name*

b.display()

*# Bird.display() #will get error , becuase self needs the address of ojbect and here there is no object instead only class address*

Sparrow flies with 2 wings

Pigeon flies with 2 wings

Bird flies with 2 wings

2

'''

Class variables are shared among all instances

instance varaibles are unique for each instance where as class variable are common for all the instances

'''

​

**class** Employee:

*#classvariable*

raiseamount **=** 1.04

​

**def** \_\_init\_\_(self,fname,lname,pay):

self.firstname **=** fname

self.lastname **=** lname

self.pay **=** pay

​

**def** fullname(self):

print('{} {} has pay of {}'.format(self.firstname,self.lastname,self.pay))

​

**def** apply\_raise(self):

*#self.pay = int(self.pay \* 1.04) #here the raise is constant*

self.pay **=** int(self.pay **\*** self.raiseamount) *#or through Employee.raiseamount*

​

​

​

​

emp\_1 **=** Employee('Asha','Talari',1000)

emp\_2 **=** Employee('Dheeray','Talari',2000)

​

print(emp\_1.pay)

emp\_1.apply\_raise() *#instead of literal we can give it to a variable and can be common to all the instances*

print(emp\_1.pay) *#like emp\_1.raiseamount or Employee.raiseamount*

​

*#we can access classvariable either through class name or through instance name .*

*#as first it will check whether it is present in instance if not it will check whether it is present in the class*

print(Employee.raiseamount)

print(emp\_1.raiseamount)

print(emp\_2.raiseamount)

​

*#to check this where raise amount is present we can print*

print(emp\_1.\_\_dict\_\_) *#it does not have the attributes*

print(Employee.\_\_dict\_\_) *# it has raise amount*

​

*#Namescape of the classvariable applicable for all the instances*

*#Employee.raiseamount = 1.05*

print(Employee.raiseamount)

print(emp\_1.raiseamount)

print(emp\_2.raiseamount)

*#applicable to only that instance*

emp\_1.raiseamount **=** 1.05

print(Employee.raiseamount)

print(emp\_1.raiseamount)

print(emp\_2.raiseamount)

​

print(emp\_1.\_\_dict\_\_) *#now if we check emp\_1 has raise amount i.e it will check in its instance since it is there it will*

*#update instead of class variable*

​

​

1000

1040

1.04

1.04

1.04

{'firstname': 'Asha', 'lastname': 'Talari', 'pay': 1040}

{'\_\_module\_\_': '\_\_main\_\_', 'raiseamount': 1.04, '\_\_init\_\_': <function Employee.\_\_init\_\_ at 0x02C34420>, 'fullname': <function Employee.fullname at 0x02C342B8>, 'apply\_raise': <function Employee.apply\_raise at 0x02C344F8>, '\_\_dict\_\_': <attribute '\_\_dict\_\_' of 'Employee' objects>, '\_\_weakref\_\_': <attribute '\_\_weakref\_\_' of 'Employee' objects>, '\_\_doc\_\_': None}

1.04

1.04

1.04

1.04

1.05

1.04

{'firstname': 'Asha', 'lastname': 'Talari', 'pay': 1040, 'raiseamount': 1.05}

​

*#Static methods*

*# we need static methods when we process at the class level but we need not involve the class or instances .*

*#static methods are used when processing related to class but does nto need class or instances to perform the work like*

*#setting environment varibales , counting the number of instances , modifying attribure of another class*

​

*# Static methods are defined within a class but are not tied to either the class nor*

*# any instance of the class; they do not receive the special first parameter representing*

*# either the class (cls for class methods) or the instances (self for instance methods).*

*# They are in effect, the same as free standing functions but are defined within a*

*# class often for convenience or to provide a way to group such functions together.*

*# A static method is a method that is decorated with the @staticmethod decorator.*

​

**class** Myclass:

*# class var*

n**=**0

counter **=**0

*#constructor that increments n when an instance is created ,*

**def** \_\_init\_\_(self):

Myclass.n **=** Myclass.n**+**1

@staticmethod *#inbuilt decorator to specify static method*

**def** noOjbects():

Myclass.counter **=** Myclass.counter**+**1

print('No of instances created',Myclass.n,Myclass.counter)

*#create 3 instances*

obj1 **=** Myclass()

obj2 **=** Myclass()

obj3 **=** Myclass()

​

Myclass.noOjbects()

Myclass.noOjbects()

Myclass.noOjbects()

​

obj1.noOjbects() *#can even call them through objects*

​

No of instances created 3 1

No of instances created 3 2

No of instances created 3 3

No of instances created 3 4

'''

regular methods or instance methods automatically takes self as first argument

class method takes cls as first argument

static method do not take either self or cls

'''

**import** datetime

**class** Employee:

​

raise\_amount **=** 0

**def** \_\_init\_\_(self,fname,lname,pay):

self.firstname **=** fname

self.lastname **=** lname

self.pay **=** pay

​

**def** fullname(self):

print('{} {} has pay of {}'.format(self.firstname,self.lastname,self.pay))

​

**def** apply\_raise(self):

self.pay **=** int(self.pay **\*** self.raiseamount)

​

@classmethod

**def** set\_raise\_amount(cls,amount):

cls.raiseamount **=** amount

​

​

*#when object is returned as class object then it is like creating obj and using it for invoking methods*

@classmethod

**def** from\_string(cls,emp\_str):

firstname, lastname, epay **=** emp\_str.split('-')

pay **=** int(epay)

**return** cls(firstname,lastname,pay)

​

@staticmethod

**def** is\_workday(day):

**if** day.weekday() **==** 5 **or** day.weekday() **==** 6:

**return** **False**

**return** **True**

​

​

​

​

*# emp\_1 = Employee('Asha','Talari',1000)*

*# emp\_2 = Employee('Dheeraj','Talari',2000)*

*#*

*# #for entire class ie all instances the value will get changed*

*# Employee.set\_raise\_amount(1.05)*

*# Employee.apply\_raise(emp\_1)*

*#*

*#*

*# emp\_1.fullname()*

*# emp\_2.fullname()*

*#*

​

*#*

​

*#classmethod is alternative method of creating class*

​

emp\_1\_str **=** 'Asha-Talari-1000'

emp\_2\_str **=**'Dheeraj-Talari-2000'

​

*# firstname ,lastname ,pay = emp\_1\_str.split('-')*

*# emp\_1 = Employee(firstname,lastname,pay)*

*# firstname,lastname,pay = emp\_2\_str.split('-')*

*# emp\_2 = Employee(firstname,lastname,pay)*

​

*# emp\_1.fullname()*

*# emp\_2.fullname()*

​

*#instead of this we can write a class method and return the class object*

​

emp\_1 **=** Employee.from\_string(emp\_1\_str)

emp\_2 **=** Employee.from\_string(emp\_2\_str)

​

​

*# print(Employee.\_\_dict\_\_) #when we have still not used class variable it will not be displayed yet*

*# print(dir(Employee))*

​

Employee.set\_raise\_amount(1.06)

emp\_1.apply\_raise()

emp\_2.apply\_raise()

​

emp\_1.fullname()

emp\_2.fullname()

​

*#import the datetime module and create date object and check whether the day is weekday or weekend*

my\_date **=** datetime.date(2020,5,17)

print(my\_date.year) *#just to show*

​

print(Employee.is\_workday(my\_date))

​

Employee.function()

​

*# print(emp\_1.\_\_dict\_\_)*

*# print(emp\_2.\_\_dict\_\_)*

*# print(dir(emp\_1)) # raise amount was used as instance variable also hence it is getting display at instance level*

*# print(Employee.\_\_dict\_\_) #here already class variable raise amount is used and it will get displayed*

Asha Talari has pay of 1060

Dheeraj Talari has pay of 2120

2020

False

**---------------------------------------------------------------------------**

**AttributeError** Traceback (most recent call last)

**<ipython-input-17-200ae1c13830>** in <module>

93 print**(**Employee**.**is\_workday**(**my\_date**))**

94

**---> 95** Employee**.**function**()**

96

97 **# print(emp\_1.\_\_dict\_\_)**

**AttributeError**: type object 'Employee' has no attribute 'function'

**class** BankAccount:

**def** \_\_init\_\_(self,accno,name,acctype,openBal):

self.Acc**=**accno

self.CustName**=**name

self.AccType**=**acctype

self.Obal**=**openBal

**def** display(self):

print(f"Account Number is {self.Acc} and the name of the person is {self.CustName} and Acctype is {self.AccType}" )

​

​

**def** \_\_str\_\_(self):

**return** (f"Account Number is {self.Acc} and the name of the person is {self.CustName} and Acctype is {self.AccType}")

**def** BalCheck(self):

print(f"Current Balance is {self.Obal}")

**def** withdraw(self):

wamt**=**0

wamt**=**int(input("Enter the amount to be withdrawn :"))

**if** wamt **<** self.Obal:

self.Obal**=**self.Obal**-**wamt

**else**:

print("Insufficient funds")

@classmethod

**def** interestrate(cls):

cls.intrate **=** 6.5

**def** applyinterest(self):

self.Obal **=** self.Obal **\*** BankAccount.intrate

bankobject1**=**BankAccount(6565,"Jack","Current",1000)

bankobject2**=**BankAccount(1234,"Kerry","Deposit",3000)

bankobject3**=**BankAccount(7879,"Mary","Investment",30000)

print(bankobject1)

print(bankobject2)

print(bankobject3)

bankobject1.BalCheck()

bankobject1.withdraw()

bankobject1.BalCheck()

BankAccount.interestrate()

bankobject1.applyinterest()

bankobject1.BalCheck()

​

*#bankobject1.display()*

*#bankobject2.display()*

*#bankobject3.display()*

Account Number is 6565 and the name of the person is Jack and Acctype is Current

Account Number is 1234 and the name of the person is Kerry and Acctype is Deposit

Account Number is 7879 and the name of the person is Mary and Acctype is Investment

Current Balance is 1000

Enter the amount to be withdrawn :500

Current Balance is 500

Current Balance is 3250.0

*#inner class*

**class** Person:

**def** \_\_init\_\_(self):

self.name **=**'charles'

self.db **=** self.Dob()

**def** display(self):

print('name =',self.name)

*#inner class*

**class** Dob:

**def** \_\_init\_\_(self):

self.mm **=**10

self.yy **=**1988

self.dd **=** 5

**def** display(self):

print('Dob = {}/{}/{}'.format(self.dd,self.mm,self.yy))

​

p **=** Person()

p.display()

​

*#to create inner class onject*

x **=** p.db

x.display()

name = charles

Dob = 5/10/1988

*#create a class circle and take input from the user in the constructor*

*#create methods area and perimeter and find area and perimeter and return to main ptogram*

**from** math **import** pi

**class** Circle:

**def** \_\_init\_\_(self):

self.radius **=** int(input("enter the value for radius"))

**def** Area\_circle(self):

area **=** pi**\***self.radius**\*\***2

**return** area

**def** perimeter\_cicle(self):

perimeter **=** 2 **\***pi**\***self.radius

**return** perimeter

cirObj **=** Circle()

print("The Area is",cirObj.Area\_circle())

​

print("The perimeter is",cirObj.perimeter\_cicle())

*#write a program to take input from the user any name and keep on appending name in a list*

*# till user enters DONE*

*#once list is created in a seperate method create a list of only those names which starts from a or A or conatains any vowel*

*# this search has to be case insensitive*

*#this final list user should not be able to modify in main program*

​

*#Asha, Amit, Dheeraj,sky (should not have sky)*

​

**class** NameCreation:

**def** \_\_init\_\_(self):

self.names **=** []

name **=** 'Amit'

**while** name.upper()**!=**'DONE':

name **=** input("Enter the names to be inserted")

**if** name.upper()**!=**'DONE':

self.names.append(name)

**def** letter\_checking(self):

self.lst **=**[]

print(self.names)

**for** name **in** self.names:

**if** name[0]**==**'A' **or** name[0]**==**'a':

self.lst.append(name)

**for** i **in** name:

**if** i.lower() **in** ['a','e','i','o','u']:

**if** name **not** **in** self.lst:

self.lst.append(name)

print(self.lst)

obj **=** NameCreation()

obj.letter\_checking()

​

*#create a program to take length and breadth of rectangle in constructor of base class*

*#in derived class calculate area and perimeter and print it*

​

​

**class** Get\_Input:

**def** \_\_init\_\_(self):

self.length **=** int(input("Enter the length"))

self.breadth **=** int(input("Enter the breadth"))

**class** Rectangle(Get\_Input):

**def** \_\_init\_\_(self):

super().\_\_init\_\_()

**def** Calculate(self):

self.area **=** self.length **\*** self.breadth

self.perimeter **=** 2 **\*** self.length **+** self.breadth

**def** \_\_str\_\_(self):

**return** "The area and perimeter is {} {}".format(self.area,self.breadth)

obj **=** Rectangle()

obj.Calculate()

print(obj)

*# in a funtion take an input from user roll number and name of student in a dictionary*

*# keep on taking name from user till user enters -999 for roll no. return this dictionary to main*

​

*# in another function take input roll number and print name of student*

*# if roll no is not found trap and exception and print not found*

*# do not use if else statements*

​

**import** sys

**def** getinput():

rollno **=** 0

dt **=** {}

**while** rollno**!=-**999:

rollno **=** int(input("Enter the roll number"))

**if** rollno **!=-**999:

stuname **=** input("Enter the student name")

dt[rollno]**=**stuname

**return** dt

**def** Checking(dt):

**try**:

checkroll **=** int(input("Enter the roll no to be checked"))

print("The name of the student is ",dt[checkroll])

**except**:

print("The student soes not exisit",sys.exc\_info()[0],sys.exc\_info()[1])

dt **=** getinput()

Checking(dt)