**Why do we need Encapsulation in Python?**

The advantages of Encapsulation in Python can be summed up as follows –

**1. Encapsulation provides well-defined, readable code**

The primary advantage of using Encapsulation in Python is that as a user, we do not need to know the architecture of the methods and the data and can just focus on making use of these functional, encapsulated units for our applications. This results in a more organized and clean code. The user experience also improves greatly and makes it easier to understand applications as a whole.

**2. Prevents Accidental Modification or Deletion**

Another advantage of encapsulation is that it prevents the accidental modification of the data and methods. Let’s consider the example of NumPy again, if I had access to edit the library, then I might make a mistake in the implementation of the mean function and then because of that mistake, thousands of projects using NumPy would become inaccurate.

**3. Encapsulation provides security**

Encapsulation in Python is achieved through the access modifiers. These access modifiers ensure that access conditions are not breached and thus provide a great user experience in terms of security.

**Access Modifiers in Python encapsulation**

Sometimes there might be a need to restrict or limit access to certain variables or functions while programming. That is where access modifiers come into the picture.

Now when we are talking about access, 3 kinds of access specifiers can be used while performing Encapsulation in Python. They are as follows :

1. Public Members
2. Private Members
3. Protected Members

**Encapsulation in Python using public members**

As the name suggests, the public modifier allows variables and functions to be accessible from anywhere within the class and from any part of the program. All member variables have the access modifier as public by default.

Now let’s check out how we can implement Encapsulation in Python using public methods –

# illustrating public members & public access modifier

class pub\_mod:

# constructor

def \_\_init\_\_(self, name, age):

self.name = name;

self.age = age;

def Age(self):

# accessing public data member

print("Age: ", self.age)

# creating object

obj = pub\_mod("Jason", 35);

# accessing public data member

print("Name: ", obj.name)

# calling public member function of the class

obj.Age()

**Encapsulation in Python using private members**

The private access modifier allows member methods and variables to be accessed only within the class. To specify a private access modifier for a member, we make use of the double underscore \_\_.

Let’s check out this example to understand how we can implement Encapsulation using private members –

# illustrating private members & private access modifier

class Rectangle:

\_\_length = 0 #private variable

\_\_breadth = 0#private variable

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

#constructor

self.\_\_length = 5

self.\_\_breadth = 3

#printing values of the private variable within the class

print(self.\_\_length)

print(self.\_\_breadth)

rect = Rectangle() #object created

#printing values of the private variable outside the class

print(rect.length)

print(rect.breadth)

5

3

Traceback (most recent call last) :

File "main.py", line 14, in <module>

print(rect.length)

AttributeError: 'Rectangle' object has no attribute 'length'

We can see that we have received an AttributeError in the output. Can you guess why?

Well, your thoughts should wander towards the private access modifier!

Since len is a private member and we have tried to access it outside the class, that is why we received the above error. We can access private members from outside of a class using the following two approaches

* Public method to access private members
* Name Mangling to access private members

**Public method to access private members**

# illustrating protected members & protected access modifier

class details:

\_name="Jason"

\_age=35

\_job="Developer"

class pro\_mod(details):

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

print(self.\_name)

print(self.\_age)

print(self.\_job)

# creating object of the class

obj = pro\_mod()

**Name Mangling to access private members**

We can directly access private and protected variables from outside of a class through name mangling. The name mangling is created on an identifier by adding two leading underscores and one trailing underscore, like this \_classname\_\_dataMember, where classname is the current class, and data member is the private variable name.

class details:

\_name="Jason"

\_age=35

\_job="Developer"

class pro\_mod(details):

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

print(self.\_name)

print(self.\_age)

print(self.\_job)

# creating object of the class

obj = pro\_mod()

# direct access of protected member

print("Name:",obj.\_name)

print("Age:",obj.\_age)

**Encapsulation in Python using protected members**

What sets protected members apart from private members is that they allow the members to be accessed within the class and allow them to be accessed by the sub-classes involved. In Python, we demonstrate a protected member by prefixing with an underscore \_ before its name.

As we know, if the members have a protected access specifier, it can also be referenced then within the class and the subsequent sub-clas

So now let’s see this concept in action –

# illustrating protected members & protected access modifier

class details:

\_name="Jason"

\_age=35

\_job="Developer"

class pro\_mod(details):

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

print(self.\_name)

print(self.\_age)

print(self.\_job)

# creating object of the class

obj = pro\_mod()

# direct access of protected member

print("Name:",obj.name)

print("Age:",obj.age)

It is quite clear from the output that the class pro\_mod was successfully able to inherit the variables from the class details and print them to the console, although they were protected variables. And when we tried to refer to them outside of their parent class and the sub-class, we got an AttributeError for the same.

**Advantages of Encapsulation**

* **Code reusability**: Encapsulation in Python allows developers to create reusable code by hiding the implementation details of an object or class and only exposing a public interface for interacting with it.
* **Data hiding**: Encapsulation helps to protect the internal state of an object or class from being accessed or modified by external code, which improves the security of the application.
* **Improved maintainability**: By encapsulating the implementation details of an object or class, developers can make changes to the internal state or behavior of the object without affecting external code that uses it.
* **Easier to understand**: Encapsulation makes the code more organized and easier