Python Encapsulation Concept:

- *The wrapping up of data and function together, into a single unit is called as encapsulation.*

  Data **+ Function ----->> Class**
- This feature keeps the data safe from outside interface and misuse.
- This led to a concept of data hiding.
- In an object oriented python program, you can restrict access to methods and variables.
- This can prevent the data from being modified by accident and is known as encapsulation.
- Encapsulation restricted accesss to methods or variables.

What is Encapsulation?

- When working with classes and dealing with sensitive data, providing global access to all the variables used within the program is not a good choice.
- Encapsulation offers a way for us to access the required variables without providing the program full-fledged access to any of those variables.
- Updating, modifying, or deleting data from variables can be done through the use of methods that are defined specifically for the purpose.
- The benefit of using this approach to programming is improved control over the input data and better security.

What is Encapsulation in Python?

- The concept of encapsulation is the same in all object-oriented programming languages.
- The difference is seen when the concepts are applied to particular languages.
- Compared to languages like Java that offer access modifiers (public or private) for variables and methods, Python provides access to all the variables and methods globally.
- Check the below demonstration of how variables can easily be accessed.

Example:

```
class Person:
    def __init__(self, name, age=0):
        self.name = name
        self.age = age
```

```
    def display(self):
        print("Person Name is :",self.name)
        print("Person Age is :",self.age)


person = Person('Srinivas', 30)
# accessing using class method
person.display()
# accessing directly from outside
print("Person Name :",person.name)
print("Person Age :",person.age)
```

Output:

Person Name is : Srinivas

Person Age is : 30

Person Name : Srinivas

Person Age : 30

Note: Since we do not have access modifiers in Python, we will use a few different ways to control the access of variables within a Python program.

Preocedure(different ways) to Control Access :

- There are multiple methods(ways) that are offered by Python to limit variable and method access across the program. Let's go over the methods in detail.

Using Single Underscore :

- A common Python programming convention to identify a protected variable is by prefixing it with an underscore.
- Now, this doesn't really make any difference on the compiler side of things.
- The variable is still accessible as usual. But being a convention that programmers have picked up on, it tells other programmers that the variables or methods have to be used only within the scope of the class.

Example:

```
class Person:
    def __init__(self, name, age=0):
        self.name = name
        self._age = age

    def display(self):
        print("Name is :", self.name)
        print("Age is :", self._age)
```

```
person = Person('Srinivas', 30)

# accessing using class method
person.display()

# accessing directly from outside
print("Name is :", person.name)
print("Age is :", person._age)
```

Output:

Name is : Srinivas

Age is : 30

Name is : Srinivas

Age is : 30

Note: It's clear that the variable access is unchanged. But can we do anything to really make it private? Let's have a look further.

Using Double Underscores:

- If you want to make class members i.e. methods and variables private, then you should prefix them with double underscores.
- But Python offers some sort of support to the private modifier. This mechanism is called "Name mangling".
- With this, it is still possible to access the class members from outside it.

Name Mangling:

In Python, any identifier with **__Var** is rewritten by a python interpreter as **_Classname__Var**, and the class name remains as the present class name.
This mechanism of changing names is called "**Name Mangling"** in Python.
In the below example, in Class person, the age variable is changed and it's prefixed by leading double underscores.

Syntax: objectName._className__variableName

Example:

```
class Person:
    def __init__(self, name, age=0):
        self.name = name
        self.__age = age

    def display(self):
        print("Person Name is :",self.name)
```

```
        print("Person Age is :",self.__age)


person = Person('Srinivas', 30)
# accessing using class method
person.display()
# accessing directly from outside
print('Trying to access variables from outside the class ')
print("Person Name is :",person.name)
print("Person Age is : ",person.__age)   # error
# print("Person Age is : ",person._Person__age) # Name Mangling Concept
```

Output:

Person Name is : Srinivas

Person Age is : 30

Trying to access variables from outside the class

Person Name is : Srinivas

AttributeError: 'Person' object has no attribute '__age'

Note: You can observe that variables are still be accessed using methods, which is a part of the class. But you cannot access age directly from outside, as it is a private variable.

Example: *Using Name Mangling Concept Accessing the Private variables*

```
class Person:
    def __init__(self, name, age=0):
        self.name = name
        self.__age = age

    def display(self):
        print("Person Name is :",self.name)
        print("Person Age is :",self.__age)
person = Person('Srinivas', 30)
# accessing using class method
person.display()
# accessing directly from outside
print('Trying to access variables from outside the class ')
print("Person Name is :",person.name)
# print("Person Age is : ",person.__age)  # error
print("Person Age is : ",person._Person__age) # Name Mangling Concept
```

Output:

Person Name is : Srinivas

Person Age is : 30

Trying to access variables from outside the class

Person Name is : Srinivas

Person Age is : 30

Using Getter and Setter methods to access private variables:

- If you want to access and change the private variables, accessor (getter) methods and mutators(setter methods) should be used, as they are part of Class.

Example:

```
class Person:
    def __init__(self, name, age=0):
        self.name = name
        self.__age = age

    def display(self):
        print("Person Name is :",self.name)
        print("Person Age is :",self.__age)

    def setAge(self, age):
        self.__age = age

    def getAge(self):
        print("Age is :",self.__age)

person = Person('Srinivas', 30)
#accessing using class method
person.display()
#changing age using setter
person.setAge(35)
person.getAge()
```

Output:

Person Name is : Srinivas

Person Age is : 30

Age is : 35

Practice Examples:

Q. Creating name and age variables as a normal variables and accessing from child class

```
class Person:
    def create(self,name, age):
        self.name = name
        self.age = age

class Employee(Person):
    def display(self):
        print("Employee name is :", self.name)
        print("Employee Age is :", self.age)

employee = Employee()
```

```
employee.create('Sachin', 50)
employee.display()
```

Output:

Employee name is : Sachin

Employee Age is : 50

Q. Creating name and age variables as a Private variables and accessing from Current class method But not accessing from child class method.

```
class Person:
    def create(self, name, age):
        self.__name = name
        self.__age = age

    def read(self):
        print("Employee name is :", self.__name)
        print("Employee Age is :", self.__age)

class Employee(Person):
    def display(self):
        print("Employee name is :", self.__name)
        print("Employee Age is :", self.__age)

employee = Employee()
employee.create('Sachin', 50)
employee.read()
employee.display()
```

```
Output:
Employee name is : Sachin
Employee Age is : 50
AttributeError: 'Employee' object has no attribute '_Employee__name'
```

Q.Creating name and age variables as a Private variables and read() method as a Private method , accessing from Current class private method and also accessing from child class method using Name Mangling concept.

```
class Person:
    def create(self,name, age):
        self.__name = name
        self.__age = age

    def __read(self):
        print("Employee name is :", self.__name)
        print("Employee Age is :", self.__age)

class Employee(Person):
    def display(self):
        print("Employee name is :", self._Person__name)
        print("Employee Age is :", self._Person__age)

employee = Employee()
employee.create('Virat',40)
```

```
# employee.__read()  # AttributeError: 'Employee' object has no attribute
'__read'
employee._Person__read()  ## Name mangling concept

employee.display()
```

Output:

```
Employee name is : Virat
Employee Age is : 40
Employee name is : Virat
Employee Age is : 40
```

Q. Creating name and age variables as a Private variables and creating read() method as Private method. Accessing private method read() from Child class method display() using Name Mangling concept.

```
class Person:
    def create(self,name, age):
        self.__name = name
        self.__age = age

    def __read(self):
        print("Employee name is :", self.__name)
        print("Employee Age is :", self.__age)

class Employee(Person):
    def display(self):
        print("I am from display() method")
        super()._Person__read()
        # super().__read()
        print("I am from display() method again")

employee = Employee()
employee.create('Virat',40)
employee.display()
```

Output:

```
I am from display() method
Employee name is : Virat
Employee Age is : 40
I am from display() method again
```

Benefits of Encapsulation in Python:

- Encapsulation not only ensures better data flow but also protects the data from outside sources.
- The concept of encapsulation makes the code self-sufficient.
- It is very helpful in the implementation level, as it prioritizes the ‘how’ type questions, leaving behind the complexities.

- You should hide the data in the unit to make encapsulation easy and also to secure the data.

Conclusion:

- Encapsulation in Python is, the data is hidden outside the object definition.
- It enables developers to develop user-friendly experience.
- This is also helpful in securing data from breaches, as the code is highly secured and cannot be accessed by outside sources.

Car Example:

- We create a class Car which has two methods: drive() and updateSoftware().
- When a car object is created, it will call the private methods __updateSoftware().
- This function cannot be called on the object directly, only from within the class.

Example:

```
class Car:
    def __init__(self):
        self.__updateSoftware()

    def drive(self):
        print('driving')

    def  __updateSoftware(self):
        print('updating software')

redcar = Car()
redcar.drive()
# redcar.__updateSoftware()  # AttributeError: 'Car' object has no
attribute '__updateSoftware'. Did you mean: '_Car__updateSoftware'?

# Accessing using Name Mangling Concept
redcar._Car__updateSoftware()    #  updating software
```

Output:

updating software
driving
updating software

- Encapsulation prevents from accessing accidentally, but not intentionally.
- The private attributes and methods are not really hidden, they're renamed by adding _Car" in the beginning of their name.
- The method can actually be called using **redcar._Car__updateSoftware()**

Class with private variables:

- Variables can be private which can be useful on many occasions. A private variable can only be changed within a class method and not outside of the class.
- Objects can hold crucial data for your application and you do not want that data to be changeable from anywhere in the code.

An example:

```
class Car:
    __maxspeed = 0
    __name = ""

    def __init__(self, maxspeed, name):
        self.__maxspeed = maxspeed
        self.__name = name

    def drive(self):
        print('Car Name is : ' + str(self.__name))
        print('Car Driving-maxspeed :' + str(self.__maxspeed))

redcar = Car(200, "Super Car")
redcar.drive()
print()

redcar.__maxspeed = 10  # will not change variable because its private
print("Driving Speed is :",redcar.__maxspeed)
print()

redcar.drive()
print()

bluecar = Car(150, "Tata")
bluecar.drive()
```

Output:

```
Car Name is : Super Car
Car Driving-maxspeed :200

Driving Speed is : 10

Car Name is : Super Car
Car Driving-maxspeed :200

Car Name is : Tata
Car Driving-maxspeed :150
```

setter() methods

- If you want to change the value of a private variable, a setter method is used.
- This is simply a method that sets the value of a private variable.

Example:

```
class Car:
    __maxspeed = 0
    __name = ""

    def __init__(self):
        self.__maxspeed = 200
        self.__name = "Supercar"

    def drive(self):
        print('driving  maxspeed :' + str(self.__maxspeed))

    def setMaxSpeed(self,speed):
        self.__maxspeed = speed

redcar = Car()
redcar.drive()

redcar.setMaxSpeed(320)
redcar.drive()
```

Output:

```
driving  maxspeed :200
driving  maxspeed :320
```

- Why would you create them? Because some of the private values you may want to change after creation of the object while others may not need to be changed at all.

Encapsulation In Python FAQs :

1) What is Encapsulation in Python?

Encapsulation in Python is the process of wrapping up variables and methods into a single entity. In programming, a class is an example that wraps all the variables and methods defined inside it.

2) How can we achieve Encapsulation in Python?

In Python, Encapsulation can be achieved using Private and Protected Access Members.

For example, using single underscore and double underscore before the variables or methods.

_name = “Virat”

__name = “Rohit”

3) How can we define a variable as Private?

In Python, Private variables are preceded by using two underscores.

For example: __name = “JS.Rao”

4) How can we define a variable as Protected?

In Python, Protected variables are preceded by using a single underscore.

For example: _name = “Rohit”