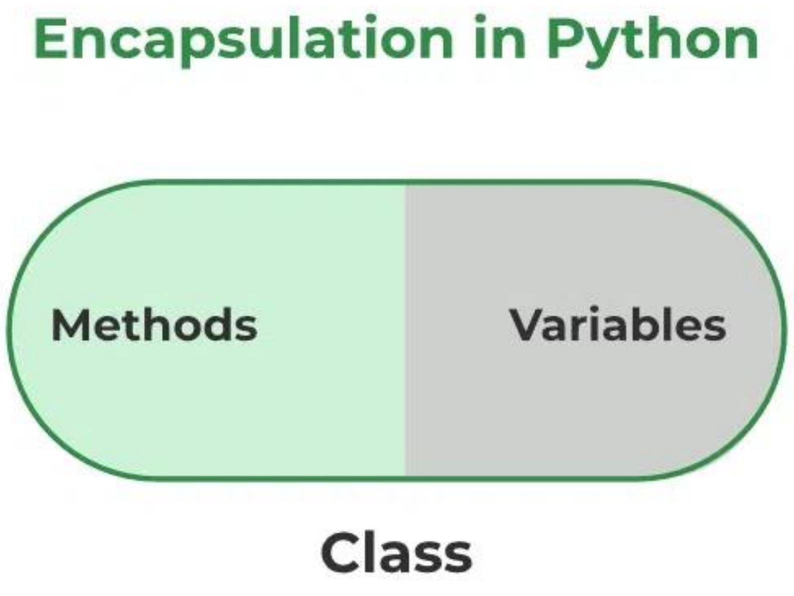
Encapsulation

- Encapsulation is one of the fundamental concepts in object-oriented programming (OOP).
- It describes the idea of wrapping data and the methods that work on data within one unit. This puts restrictions on accessing variables and methods directly and can prevent the accidental modification of data.
- To prevent accidental change, an object's variable can only be changed by an object's method. Those types of variables are known as **private variables.**
- A class is an example of encapsulation as it encapsulates all the data that is member functions, variables, etc.
- The goal of information hiding is to ensure that an object's state is always valid by controlling access to attributes that are hidden from the outside world.



❖ Protected members

- Protected members are those members of the class that cannot be accessed outside the class but can be accessed from within the class and its subclasses.
- To accomplish this in Python, just follow **the convention** by prefixing the name of the member by a **single underscore "_"**.
- Although the protected variable can be accessed out of the class as well as in the derived class (modified too in derived class), it is customary(convention not a rule) to not access the protected out the class body.

Note: The __init__ method is a constructor and runs as soon as an object of a class is instantiated.

Example:

```
class MyClass:
    def __init__(self):
        self._protected_var = 42  # This is a protected variable

    def _protected_method(self):
        print("This is a protected method")

# Outside the class
obj = MyClass()

# Accessing the protected variable
print(obj._protected_var)

# Calling the protected method
obj._protected_method()
```

Output:

```
42

This is a protected method
```

❖ Private members

- Private members are similar to protected members, the difference is that the class members declared private should neither be accessed outside the class nor by any base class.
- In Python, there is no existence of **Private** instance variables that cannot be accessed except inside a class.
- However, to define a private member prefix the member name with double underscore "__".

Note: Python's private and protected members can be accessed outside the class through python name mangling.

Example:

```
class MyClass:
    def __init__(self):
        self.public_member = "I'm a public member"
        self.__private_member = "I'm a private member"

obj = MyClass()

print(obj.public_member)
print(obj._MyClass__private_member)
```

Output

```
I'm a public member

I'm a private member
```

➔ 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 to understand by clearly separating the public interface from the implementation details.
- **Better control over class properties:** Encapsulation allows developers to have better control over the properties and methods of a class, by making some properties private or protected and only allowing them to be accessed by specific methods.
- **Better class design:** Encapsulation encourages developers to design classes that are cohesive and have a single responsibility, which makes the code more modular and easier to maintain.