**Encapsulation in Python**

Updated on: August 28, 2021 | [11 Comments](https://pynative.com/python-encapsulation/#llc_comments)

Encapsulation is one of the fundamental concepts in [object-oriented programming](https://pynative.com/python/object-oriented-programming/) (OOP), including abstraction, inheritance, and polymorphism. This lesson will cover what encapsulation is and how to implement it in Python.

**After reading this article, you will learn:**

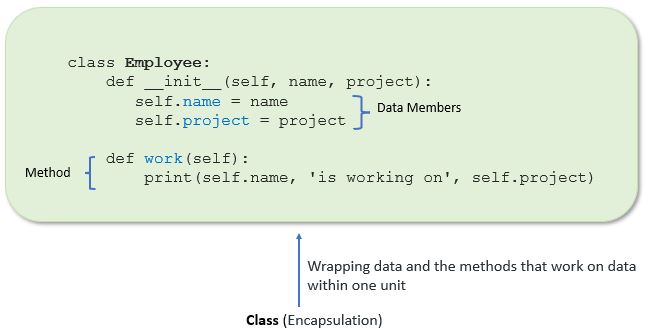
* Encapsulation in Python
* Need for Encapsulation
* Data Hiding using public, protected, and private members
* Data Hiding vs. Encapsulation
* Getter and Setter Methods
* Benefits of Encapsulation

**Table of contents**

* [What is Encapsulation in Python?](https://pynative.com/python-encapsulation/#h-what-is-encapsulation-in-python)
* [Access Modifiers in Python](https://pynative.com/python-encapsulation/#h-access-modifiers-in-python)
* [Public Member](https://pynative.com/python-encapsulation/#h-public-member)
* [Private Member](https://pynative.com/python-encapsulation/#h-private-member)
  + [Public method to access private members](https://pynative.com/python-encapsulation/#h-public-method-to-access-private-members)
  + [Name Mangling to access private members](https://pynative.com/python-encapsulation/#h-name-mangling-to-access-private-members)
* [Protected Member](https://pynative.com/python-encapsulation/#h-protected-member)
* [Getters and Setters in Python](https://pynative.com/python-encapsulation/#h-getters-and-setters-in-python)
* [Advantages of Encapsulation](https://pynative.com/python-encapsulation/#h-advantages-of-encapsulation)

**What is Encapsulation in Python?**

[Encapsulation](https://en.wikipedia.org/wiki/Encapsulation_(computer_programming)) in Python describes the concept of **bundling data and**[**methods**](https://pynative.com/python-instance-methods/)**within a single unit**. So, for example, when you create a [class](https://pynative.com/python-classes-and-objects/), it means you are implementing encapsulation. A class is an example of encapsulation as it binds all the data members ([instance variables](https://pynative.com/python-instance-variables/)) and methods into a single unit.

Implement encapsulation using a class

**Example**:

In this example, we create an Employee class by defining employee attributes such as name and salary as an instance variable and implementing behavior using work() and show() instance methods.

**class** Employee:

# constructor

**def** \_\_init\_\_(self, name, salary, project):

# data members

self.name = name

self.salary = salary

self.project = project

# method

# to display employee's details

**def** show(self):

# accessing public data member

**print**("Name: ", self.name, 'Salary:', self.salary)

# method

**def** work(self):

**print**(self.name, 'is working on', self.project)

# creating object of a class

emp = Employee('Jessa', 8000, 'NLP')

# calling public method of the class

emp.show()

emp.work()

**Output**:

Name: Jessa Salary: 8000

Jessa is working on NLP

Using encapsulation, we can hide an object’s internal representation from the outside. This is called information hiding.

Also, encapsulation allows us to restrict accessing variables and methods directly and prevent accidental data modification by creating private data members and methods within a class.

Encapsulation is a way to can restrict access to methods and variables from outside of class. Whenever we are working with the class and dealing with sensitive data, providing access to all variables used within the class is not a good choice.

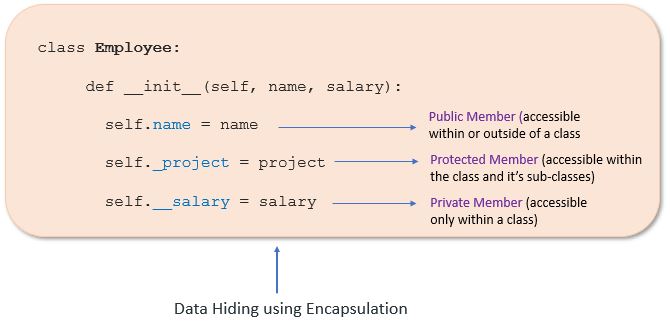
For example, Suppose you have an attribute that is not visible from the outside of an object and bundle it with methods that provide read or write access. In that case, you can hide specific information and control access to the object’s internal state. Encapsulation offers a way for us to access the required variable without providing the program full-fledged access to all variables of a class. This mechanism is used to protect the data of an object from other objects.

**Access Modifiers in Python**

Encapsulation can be achieved by declaring the data members and methods of a class either as private or protected. But In Python, we don’t have direct access modifiers like public, private, and protected. We can achieve this by using single **underscore** and **double** **underscores**.

Access modifiers limit access to the variables and methods of a class. Python provides three types of access modifiers private, public, and protected.

* **Public Member**: Accessible anywhere from otside oclass.
* **Private Member**: Accessible within the class
* **Protected Member**: Accessible within the class and its sub-classes

Data hiding using access modifiers

**Public Member**

Public data members are accessible within and outside of a class. All member variables of the class are by default public.

**Example**:

**class** Employee:

# constructor

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

# public data members

self.name = name

self.salary = salary

# public instance methods

**def** show(self):

# accessing public data member

**print**("Name: ", self.name, 'Salary:', self.salary)

# creating object of a class

emp = Employee('Jessa', 10000)

# accessing public data members

**print**("Name: ", emp.name, 'Salary:', emp.salary)

# calling public method of the class

emp.show()

**Output**

Name: Jessa Salary: 10000

Name: Jessa Salary: 10000

**Private Member**

We can protect variables in the class by marking them private. To define a private variable add two underscores as a prefix at the start of a variable name.

Private members are accessible only within the class, and we can’t access them directly from the class objects.

**Example**:

**class** Employee:

# constructor

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

# public data member

self.name = name

# private member

self.\_\_salary = salary

# creating object of a class

emp = Employee('Jessa', 10000)

# accessing private data members

**print**('Salary:', emp.\_\_salary)

**Output**

AttributeError: 'Employee' object has no attribute '\_\_salary'

In the above example, the salary is a private variable. As you know, we can’t access the private variable from the outside of that class.

We can access private members from outside of a class using the following two approaches

* Create public method to access private members
* Use name mangling

Let’s see each one by one

**Public method to access private members**

**Example**: Access Private member outside of a class using an instance method

**class** Employee:

# constructor

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

# public data member

self.name = name

# private member

self.\_\_salary = salary

# public instance methods

**def** show(self):

# private members are accessible from a class

**print**("Name: ", self.name, 'Salary:', self.\_\_salary)

# creating object of a class

emp = Employee('Jessa', 10000)

# calling public method of the class

emp.show()

**Output**:

Name: Jessa Salary: 10000

**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.

**Example**: Access private member

**class** Employee:

# constructor

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

# public data member

self.name = name

# private member

self.\_\_salary = salary

# creating object of a class

emp = Employee('Jessa', 10000)

**print**('Name:', emp.name)

# direct access to private member using name mangling

**print**('Salary:', emp.\_Employee\_\_salary)

**Output**

Name: Jessa

Salary: 10000

**Protected Member**

Protected members are accessible within the class and also available to its sub-classes. To define a protected member, prefix the member name with a single underscore \_.

Protected data members are used when you implement [inheritance](https://pynative.com/python-inheritance/) and want to allow data members access to only child classes.

**Example**: Proctecd member in inheritance.

# base class

**class** Company:

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

# Protected member

self.\_project = "NLP"

# child class

**class** Employee(Company):

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

self.name = name

Company.\_\_init\_\_(self)

**def** show(self):

**print**("Employee name :", self.name)

# Accessing protected member in child class

**print**("Working on project :", self.\_project)

c = Employee("Jessa")

c.show()

# Direct access protected data member

**print**('Project:', c.\_project)

**Output**

Employee name : Jessa

Working on project : NLP

Project: NLP

**Getters and Setters in Python**

To implement proper encapsulation in Python, we need to use setters and getters. The primary purpose of using getters and setters in object-oriented programs is to ensure data encapsulation. Use the getter method to access data members and the setter methods to modify the data members.

In Python, private variables are not hidden fields like in other programming languages. The getters and setters methods are often used when:

* When we want to avoid direct access to private variables
* To add validation logic for setting a value

**Example**

**class** Student:

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

# private member

self.name = name

self.\_\_age = age

# getter method

**def** get\_age(self):

**return** self.\_\_age

# setter method

**def** set\_age(self, age):

self.\_\_age = age

stud = Student('Jessa', 14)

# retrieving age using getter

**print**('Name:', stud.name, stud.get\_age())

# changing age using setter

stud.set\_age(16)

# retrieving age using getter

**print**('Name:', stud.name, stud.get\_age())

**Output**

Name: Jessa 14

Name: Jessa 16

Let’s take another example that shows how to use encapsulation to implement information hiding and apply additional validation before changing the values of your object attributes (data member).

**Example**: Information Hiding and conditional logic for setting an object attributes

**class** Student:

**def** \_\_init\_\_(self, name, roll\_no, age):

# private member

self.name = name

# private members to restrict access

# avoid direct data modification

self.\_\_roll\_no = roll\_no

self.\_\_age = age

**def** show(self):

**print**('Student Details:', self.name, self.\_\_roll\_no)

# getter methods

**def** get\_roll\_no(self):

**return** self.\_\_roll\_no

# setter method to modify data member

# condition to allow data modification with rules

**def** set\_roll\_no(self, number):

**if** number > 50:

**print**('Invalid roll no. Please set correct roll number')

**else**:

self.\_\_roll\_no = number

jessa = Student('Jessa', 10, 15)

# before Modify

jessa.show()

# changing roll number using setter

jessa.set\_roll\_no(120)

jessa.set\_roll\_no(25)

jessa.show()

**Output**:

Student Details: Jessa 10

Invalid roll no. Please set correct roll number

Student Details: Jessa 25

**Advantages of Encapsulation**

* **Security**: The main advantage of using encapsulation is the security of the data. Encapsulation protects an object from unauthorized access. It allows private and protected access levels to prevent accidental data modification.
* **Data Hiding**: The user would not be knowing what is going on behind the scene. They would only be knowing that to modify a data member, call the setter method. To read a data member, call the getter method. What these setter and getter methods are doing is hidden from them.
* **Simplicity**: It simplifies the maintenance of the application by keeping classes separated and preventing them from tightly coupling with each other.
* **Aesthetics**: Bundling data and methods within a class makes code more readable and maintainable