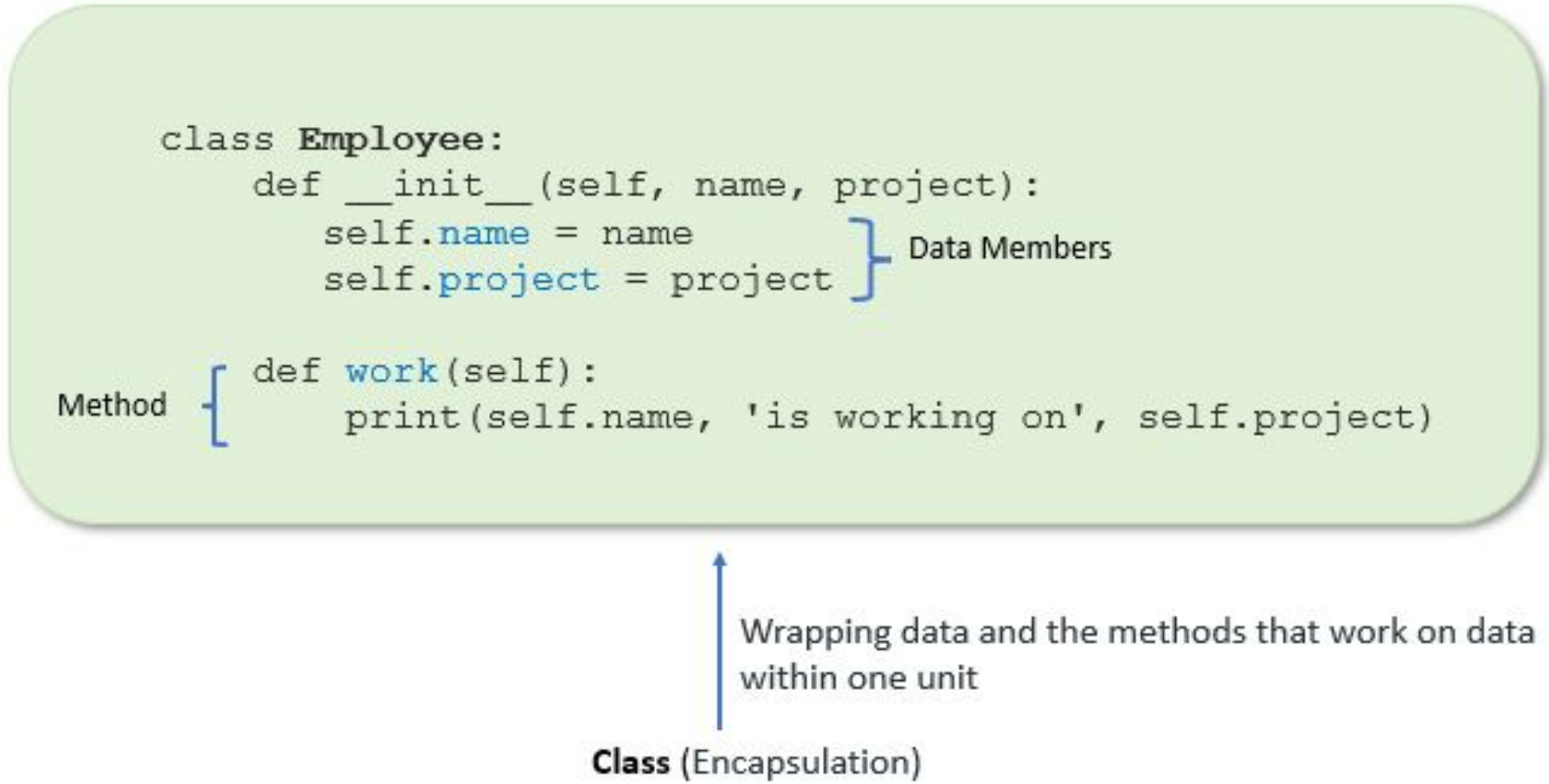
OOPs-3

What is Encapsulation in Python?

Encapsulation in Python describes the concept of bundling data and methods within a single unit. So, for example, when you create a class, it means you are implementing encapsulation. A class is an example of encapsulation as it binds all the data members (instance variables) and methods into a single unit.



Implement encapsulation using a class

```python
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()
```

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.

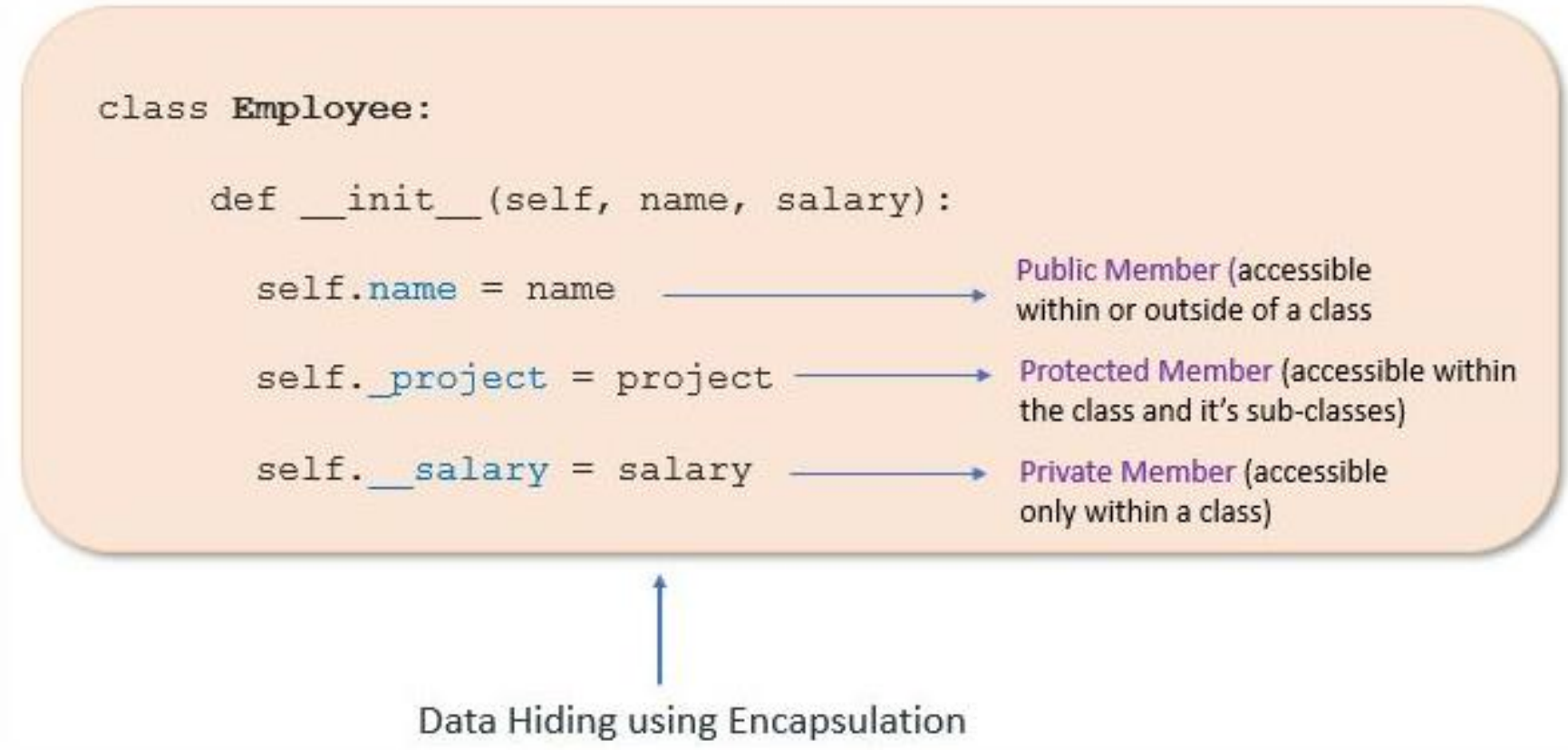
Access Modifiers in Python

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 outside class.
Private Member: Accessible within the class
Protected Member: Accessible within the class and its sub-classes

- **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()
```

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)
```

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 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)
```

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())
```

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