CBS Department

OOP Principles: Encapsulation and Abstraction

Soma Soleiman Zadeh

Object-Oriented Programming (CBS 215)

Fall 2025 - 2026

Week 8

November 26-27, 2025

Outline

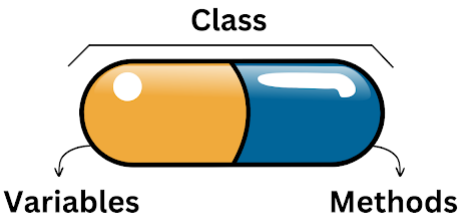
- **OOP Concepts (Encapsulation, Abstraction, Inheritance, Polymorphism)**
- **Encapsulation**
- **Abstraction**
  - **Public** Attributes/Methods
  - **Private** Attributes/Methods
- **getters** and **Setters**

Encapsulation

- **Encapsulation** is a core principle of Object-Oriented Programming.
- **Encapsulation** is about bundling **data** and **methods** that operate on the data within one unit, such as a **class.**

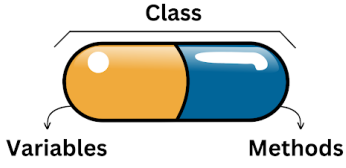
```
class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def setName(self, newName):
        self.name = newName
```



Encapsulation

- What is the benefit of **encapsulation**?
  - Hiding a class internal details and only exposing what's necessary.
  - Protecting a class's important data from being changed by other classes and functions.



Abstraction

- **Encapsulation** implements the concept of **abstraction**.
- **Abstraction** focuses on hiding implementation details and exposing only the essential functionality of an object.
  - ➢ Simplifying complex concepts.
- User of an object should only "see" the public interface of the object; all the internal details are hidden.

A Real-World Example of Abstraction

Simple interface to perform tasks, without knowing what happens inside the ATM machine (Hiding unnecessary details)

Public Variables vs. Private Variables

Each class can have **public** or **private** attributes/methods.

- Public attributes/methods are accessed by

Public Attributes/Methods

- **Python attributes and methods are public by default.**

  - **public attributes:** public attributes can be viewed and changed by any other class or function.

  - **public methods:** public methods can be called by any other class or function.

Can I Make Attributes/Methods Private?

- It is possible to make attributes/methods **private** in a class.
- Add __ (two underscores) to the beginning of the attribute or method name, and it becomes private!

name and **age** are public attributes.

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

    def setName(self,newName):
        self.name = newName
```

name and **age** are private attributes.

```
class student:
    def __init__(self, name, age):
        self.__name = name
        self.__age = age

    def setName(self,newName):
        self.__name = newName
```

Private Attributes/Methods

- **Private attributes** can only be accessed by methods defined in the class.
- **Private methods** can only be called by other methods defined in the class.

- Both **name** and **age** are private attributes.
- **setName( )** is public method.

```
class Student:
    def __init__(self, name, age):
        self.__name = name
        self.__age = age

    def setName(self, newName):
        self.__name = newName
```

- Both **name** and **age** are private attributes.
- **setName( )** is a private method.

```
class Student:
    def __init__(self, name, age):
        self.__name = name
        self.__age = age

    def __setName(self, newName):
        self.__name = newName
```

Getter and Setter Methods

- In Python, **getter** and **setter** methods are used to access and change private attributes safely.
- **Getter Methods** → Methods that return an attribute value.
- **Setter Methods** → Methods that set(change) an attribute value.

Getter and Setter Methods

- Instead of accessing private data directly, **getter** and **setter** methods provide controlled access, allowing you to:
  - Read data using a **getter** method.
  - Update data using a **setter** method.
- **getter** and **setter** methods hide the internal data of your class.
- You can make all attributes **private**, and use **getter** and **setter** methods to access or modify them safely.

Class Example

- Create a class named **Person**, with the following details:
  - Attributes:
    - Name (**public** attribute)
    - Age (**private** attribute)
  - Methods:
    - getAge (**public** method)
    - setAge (**public** method)

setAge( ) method doesn't change the age if its value is negative.

Class Example

```
############ Creating Person Class #############
class Person:
    def __init__(self, name, age):
        self.name = name
        self.__age = age

    def getAge(self):
        return self.__age

    def setAge(self, newAge):
        if newAge <= 0:
            print('Invalid Age Value!')
        else:
            self.__age = newAge
```

```
########## Creating One Person Object #############
p1 = Person('Kate', 24)

################ Accessing Attributes Directly ####################

print(p1.name)        # Public Attribute. Name is shown --> 'Kate'.
print(p1.__age)       # Private Attribute. Age is not shown.


######################### Calling Methods ##########################
print(p1.getAge())    # Getter Method. Accessing Age attribute
                      # through getter. Age is shown --> 24

p1.setAge(25)         # Setter Method. Changing Age attribute
                      # through setter. Age is changed to 25.

print(p1.getAge())    # Getter Method. Accessing Age attribute
                      # through getter. New Age is shown --> 25
```

Instance Attributes vs. Class Attributes

- **Instance Attributes**
  - Variables that belong to one object;
  - They are **unique to each object** and are not shared between other class objects.
- **Class Attributes**
  - Variables that belong to a class,
  - They are **shared between all class objects**.

Instance Attributes vs. Class Attributes

- **Instance Attributes**
  - **Instance attributes** are created inside **__init__( )** method.
  - If you change an instance attribute, only one object is affected.
- **Class Attributes**
  - **Class attributes** are created outside **__init__( )** method.
  - If you change a class attribute, the value is changed for all objects.

Student Class – Class Attributes vs. Instance Attributes

```
############### Creating Student Class ###############
class Student:
    department = 'Cybersecurity'
    def __init__(self, student_id, name, gpa):
        self.student_id = student_id
        self.name = name
        self.gpa = gpa

########## Creating Two Student Objects #############
s1 = Student(1, 'Kate', 2.6)
s2 = Student(2, 'Mike', 3.1)
```

Student Class – Class Attributes vs. Instance Attributes

```
print(s1.name, "with gpa =", s1.gpa, "studies in", s1.department, "department!")
print(s2.name, "with gpa =", s2.gpa,"studies in", s2.department, "department!")

# Assigning new value to an instance attribute
s1.gpa = 3.4
print(s1.name, "with gpa =", s1.gpa, "studies in", s1.department, "department!")
print(s2.name, "with gpa =", s2.gpa,"studies in", s2.department, "department!")

# Assigning new value to a class attribute
Student.department = "IT"
print(s1.name, "with gpa =", s1.gpa, "studies in", s1.department, "department!")
print(s2.name, "with gpa =", s2.gpa,"studies in", s2.department, "department!")
```

department value is Cybersecurity for all objects.

gpa is an instance attribute, so it is changed only for one object.

department is a class attribute, so it is changed for the class (all objects).

```
Kate with gpa = 2.6 studies in Cybersecurity department!
Mike with gpa = 3.1 studies in Cybersecurity department!


Kate with gpa = 3.4 studies in Cybersecurity department!
Mike with gpa = 3.1 studies in Cybersecurity department!


Kate with gpa = 3.4 studies in IT department!
Mike with gpa = 3.1 studies in IT department!
```