Object Oriented Programming (OOP)

Object-oriented programming (OOP) is a programming paradigm based on the concept of "**objects**".

The object contains both data and code:

- Data in the form of **properties** (often known as attributes or state),
- Code in the form of **methods** (actions an object can perform or behavior).

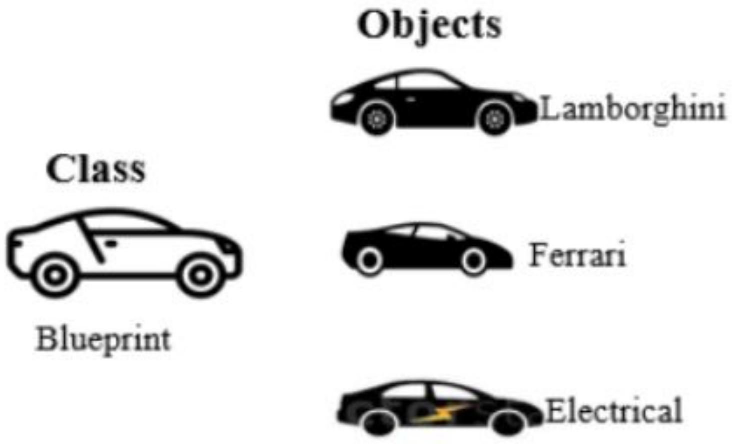
One important aspect of OOP in Python is to create **reusable code** using the concept of inheritance.

This concept is also known as DRY (Don't Repeat Yourself).

Class and Objects

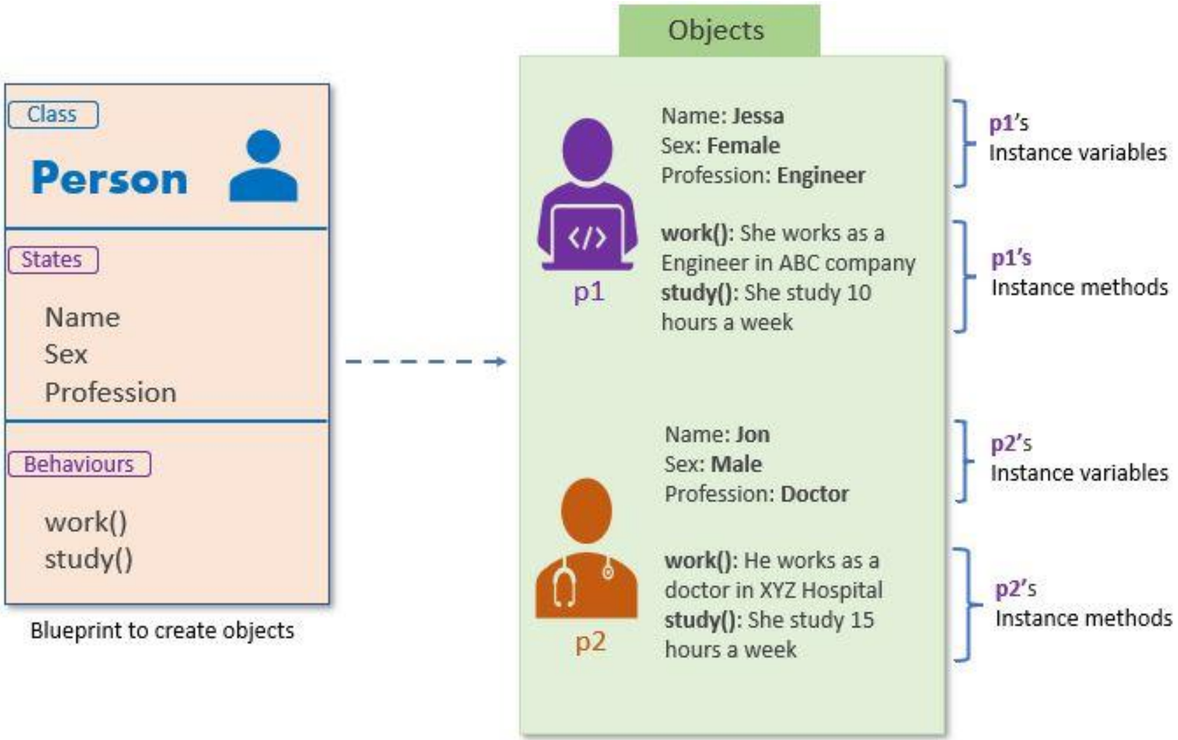
In Python, everything is an object.

- **Class**:
  - The class is a user-defined data structure that binds the data members and methods into a single unit.
  - Class is a blueprint or code template for object creation.
  - Using a class, you can create as many objects as you want.
- **Object**:
  - An **object is an instance of a class**.
  - It is a collection of attributes (variables) and methods.
  - We use the object of a class to perform actions.



In short, Every object has the following property

- **Identity:** Every object must be uniquely identified.
- **State:** An object has an attribute that represents a state of an object, and it also reflects the property of an object.
- **Behavior**: An object has methods that represent its behavior to modify its state or etc.



A real-life example of class and objects.

Class: Person

- **State**: Name, Sex, Profession
- **Behavior**: Working, Study

Using the above class, we can create multiple objects that depict different states and behavior.

Object 1: Jessa

- **State**:
  - Name: Jessa
  - Sex: Female
  - Profession: Software Engineer
- **Behavior**:
  - Working: She is working as a software developer at ABC Company
  - Study: She studies 2 hours a day

Object 2: Jon

- **State**:
  - Name: Jon
  - Sex: Male
  - Profession: Doctor
- **Behavior**:
  - Working: He is working as a doctor
  - Study: He studies 5 hours a day

As you can see, Jessa is female, and she works as a Software engineer. On the other hand, Jon is a male, and he is a doctor. Here, both **objects are created from the same class, but they have different states and behaviors**.

Now we will see some real example to start of classes and objects with variables.

There are several kinds of variables in Python:

- **Instance variables** in a class: these are called fields or attributes of an object
- **Local Variables**: Variables in a method or block of code
- **Parameters**: Variables in method declarations
- **Class variables:** This variable is shared between all objects of a class

Now when we design a class,
we use instance variables and class variables.

- Instance variables are declared inside a method using the `self` keyword

```python
class Car:
    # Class variable
    manufacturer = 'BMW'

    def __init__(self, model, price):
        # instance variable
        self.model = model
        self.price = price

# create Object
car = Car('x1', 2500)
print(car.model, car.price, Car.manufacturer)
```

Now we will see some real example to start of classes and objects.

```
#Syntax of defining a class

class class_name(object):
    #body of class
    #attributes
    #methods
```

Syntax:

```
reference_variable = classname()
```

The example for object of Person class can be:

```
obj = Person()
```

Here, `obj` is an `object` of class Person.

```
[12]: # Creating a class

      class Person:
          pass
      print(Person)
```

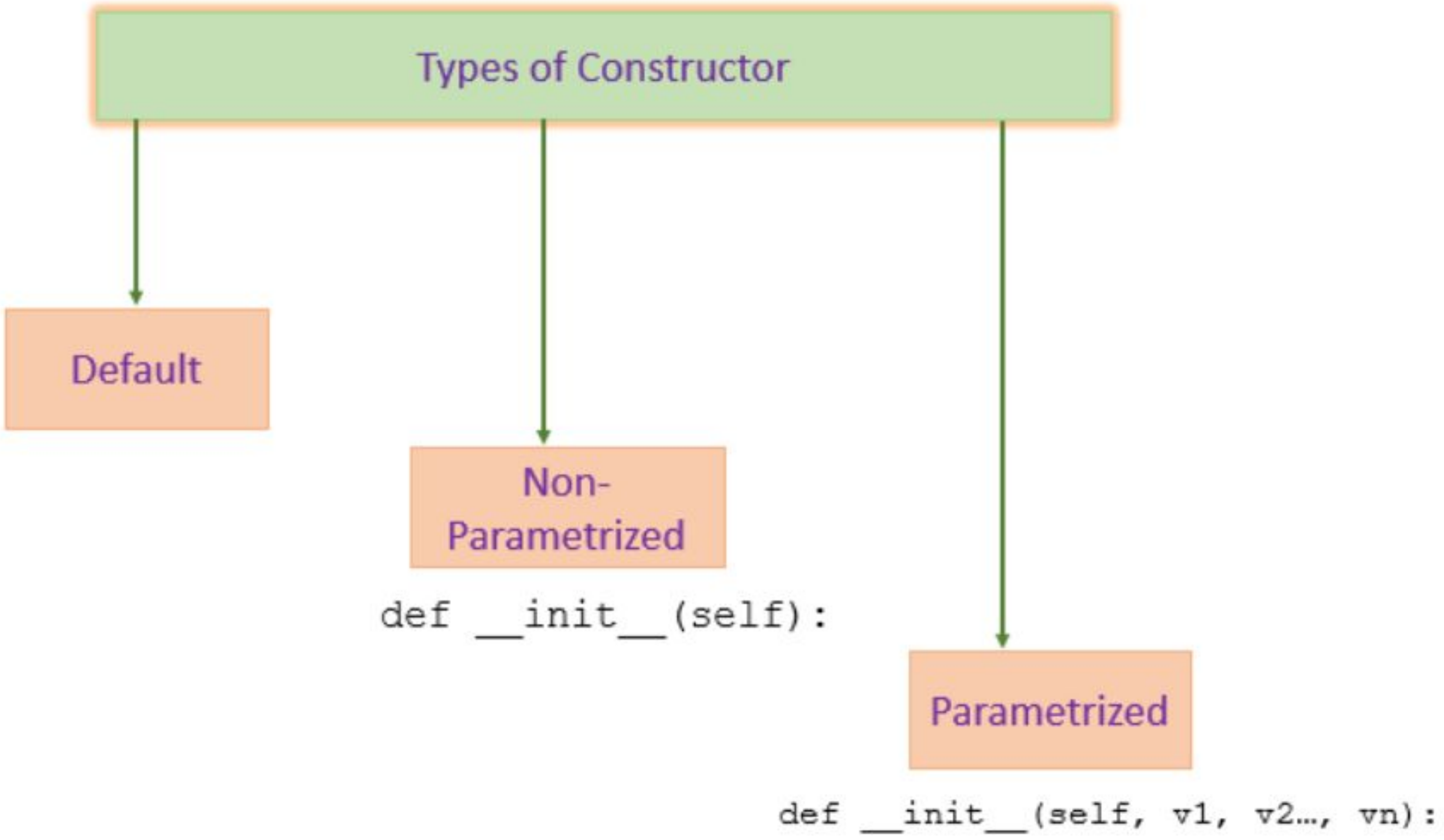
```
<class '__main__.Person'>
```

```
[13]: # Example 1: We can create an object by calling the class

      p = Person()
      print(p)
```

```
<__main__.Person object at 0x0000020D18929B20>
```

Instance Variable	Class Variable
Instance variables are not shared by objects. Every object has its own copy of the instance attribute	Class variables are shared by all instances and object.
Instance variables are declared inside the constructor i.e., the `__init__()` method.	Class variables are declared inside the class definition but outside any of the instance methods and constructors.
It is gets created when an instance of the class is created.	It is created when the program begins to execute.
Changes made to these variables through one object will not reflect in another object.	Changes made in the class variable will reflect in all objects.


Types of Constructor
Default
Non-
Parametrized
def __init__(self):
Parametrized
def __init__(self, v1, v2…, vn):

Default Constructor

```python
class Employee:

    def display(self):
        print('Inside Display')

emp = Employee()
emp.display()
```

Non-Parametrized Constructor

```python
class Company:

    # no-argument constructor
    def __init__(self):
        self.name = "PYnative"
        self.address = "ABC Street"

    # a method for printing data members
    def show(self):
        print('Name:', self.name, 'Address:', self.address)

# creating object of the class
cmp = Company()

# calling the instance method using the object
cmp.show()
```

Parametrized Constructor

```
class Employee:
    # parameterized constructor
    def __init__(self, name, age, salary):
        self.name = name
        self.age = age
        self.salary = salary

    # display object
    def show(self):
        print(self.name, self.age, self.salary)

# creating object of the Employee class
emma = Employee('Emma', 23, 7500)
emma.show()

kelly = Employee('Kelly', 25, 8500)
kelly.show()
```

Constructor with default values (age and classroom)

```python
class Student:
    # constructor with default values age and classroom
    def __init__(self, name, age=12, classroom=7):
        self.name = name
        self.age = age
        self.classroom = classroom

    # display Student
    def show(self):
        print(self.name, self.age, self.classroom)

# creating object of the Student class
emma = Student('Emma')
emma.show()

kelly = Student('Kelly', 13)
kelly.show()
```

Self Keyword in Python

We use **self** as the first parameter

- Using self, **we can access the instance variable and instance method of the object.**

- Whenever we call an instance method through an object,

  the Python compiler implicitly passes object reference as the first argument commonly known as self.

- It is not mandatory to name the first parameter as a `self`. We can give any name whatever we like, but it has to be the first parameter of an instance method.

Ways to Access Instance Variable

There are two ways to access the instance variable of class:

- Within the class in instance method by using the **object reference** (`self`) or

  name_object.name_Instance_Variable

- Using `getattr()` method

```python
class Student:
    # constructor
    def __init__(self, name, age):
        # Instance variable
        self.name = name
        self.age = age

# create object
stud = Student("Jessa", 20)
# First Way
print(stud.name)
# Use getattr instead of stud.name
print('Name:', getattr(stud, 'name'))
print('Age:', getattr(stud, 'age'))
```

task 1: OOP Building a Simple Statistics Class in Python

Task Overview:

Create a class called `my_statistics` that calculates basic statistics from a list of numbers.
The class should include **4 methods**:

- `count()` → Returns the number of elements
- `sum()` → Returns the total sum of elements
- `mean()` → Returns the average value
- `median()` → Returns the middle value

For example task 1:

```
ages = [31, 26, 34, 37, 27, 26, 32, 32, 26, 27]
Calc = my_statistics(ages)
Calc.count()  ➝ returns 10
Calc.sum()    ➝ returns 298
Calc.mean()   ➝ returns 29.8
Calc.median() ➝ returns 29.0   >>> sorted()
```

task 2: OOP Voting System

Create a class to simulate a simple voting system with the following functions:

Function Name	Description
`add_candidate(name)`	Add a candidate to the election
`show_candidates()`	Display the list of all candidates
`vote(name)`	Add a vote to a candidate by name
`get_results(name)`	Return the total votes for a specific candidate
`display_winner()`	Show the candidate with the highest number of votes

Notes:

- If the candidate doesn't exist, the system prints an error message.
- Voting results and winner are displayed clearly.
- You can expand the logic by adding voter ID tracking or duplicate vote prevention.

For example task 2 in a next slide

For example

```
voting = voting_system()

voting.add_candidate('Baraa')
voting.add_candidate('Sama')
voting.add_candidate('Saif')

voting.show_candidates()
# Output: ['Baraa', 'Sama', 'Saif']

# Voting process
voting.vote_for_candidate('Baraa')
voting.vote_for_candidate('Baraa')
voting.vote_for_candidate('Baraa')

voting.vote_for_candidate('Sama')
voting.vote_for_candidate('Sama')

voting.vote_for_candidate('ssss')
# Output: Candidate (ssss) is not on the list

# Get individual results
voting.get_result_of_candidate('Baraa')  # Output: Number of votes for (Baraa) are: 3
voting.get_result_of_candidate('Sama')   # Output: Number of votes for (Sama) are: 2
voting.get_result_of_candidate('sss')    # Output: No such candidate!

# Display winner
voting.display_winner()
# Output: The winner is Baraa with 3 votes.
```

There are four Pillars of Object Oriented Programming:

- Abstraction
- **Encapsulation**
- Inheritance
- Polymorphism

Lets try to understand each of them in a most easiest way!

Ways to Accessing Class Variables

In Python, we can access the class variable in the following places

- Access inside the constructor by using either `self` parameter or class name.

- Access class variable inside instance method by using either self of class name
- Access from outside of class by using either object reference or class name.

```python
class Student:
    # Class variable
    school_name = 'ABC School '

    # constructor
    def __init__(self, name, roll_no):
        self.name = name
        self.roll_no = roll_no

    # Instance method
    def show(self):
        print('Inside instance method')
        # access using self
        print(self.name, self.roll_no, self.school_name)
        # access using class name
        print(Student.school_name)

# create Object
s1 = Student('Emma', 10)
s1.show()

print('Outside class')
# access class variable outside class
# access using object reference
print(s1.school_name)

# access using class name
print(Student.school_name)
```

Access Instance Variable From Another Class

In Python, we can access the class variable in the following places

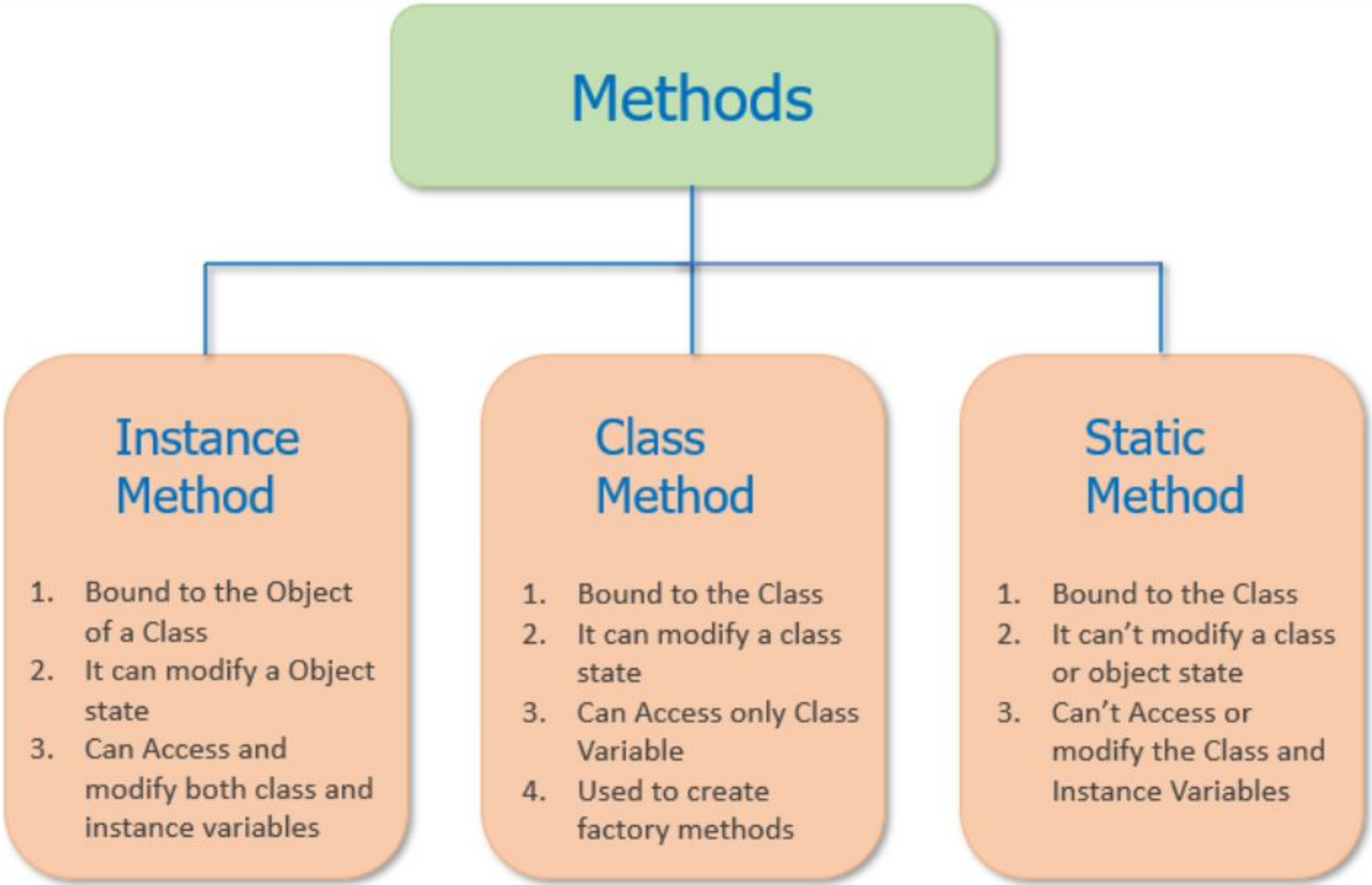
- Access inside the constructor by using either `self` parameter or class name.
- Access class variable inside instance method by using either self of class name
- Access from outside of class by using either object reference or class name.

```python
class Vehicle:
    def __init__(self):
        self.engine = '1500cc'

class Car(Vehicle):
    def __init__(self, max_speed):
        # call parent class constructor
        super().__init__()
        self.max_speed = max_speed

    def display(self):
        # access parent class instance variables 'engine'
        print("Engine:", self.engine)
        print("Max Speed:", self.max_speed)

# Object of car
car = Car(240)
car.display()
```


Methods
Instance Method
1. Bound to the Object of a Class
2. It can modify a Object state
3. Can Access and modify both class and instance variables
Class Method
1. Bound to the Class
2. It can modify a class state
3. Can Access only Class Variable
4. Used to create factory methods
Static Method
1. Bound to the Class
2. It can't modify a class or object state
3. Can't Access or modify the Class and Instance Variables

```python
Employee(object):
epart_name = "Computer"
ef __init__(self, name, salary, project_name):
    self.name = name
    self.salary = salary
    self.project_name = project_name

lassmethod ## Class methods can only access and modify class attributes and not instance attributes.
ef change_Department(cls, depart_name):
    cls.depart_name = depart_name
    print('New Department', cls.depart_name )

taticmethod
ef gather_requirement(project_name):
    if project_name == 'ABC Project':
        requirement = ['task_1', 'task_2', 'task_3']
    else:
        requirement = ['task_1']
    return requirement

instance method
ef work(self):
    # call static method from instance method
    requirement = self.gather_requirement(self.project_name)
    for task in requirement:
        print('Completed', task)

instance method
ef show(self):
    print(self.name,self.salary,self.project_name,Employee.depart_name)
```

```python
emp = Employee('Kelly', 12000, 'ABC Project')
emp.work()
emp.gather_requirement('ABC Project')
emp.show()

Employee.change_Department('CCC')

emp.show()
emp2 = Employee('baraa', 4566, 'Project Management')
emp2.show()
```

```
Completed task_1
Completed task_2
Completed task_3
Kelly 12000 ABC Project Computer
New Department CCC
Kelly 12000 ABC Project CCC
baraa 4566 Project Management CCC
```

Access modifiers in Python Programming

To encapsulation

Access Specifiers	Same Class	Same Package	Derived Class
Public	Yes	Yes	Yes
Protected	Yes	Yes	Yes
Private	Yes	No	No

```
#defining class Student
class Student:
    #constructor is defined
    def __init__(self, name, age, salary):
        self.age = age                  # public Attribute
        self._name = name               # protected Attribute
        self.__salary = salary          # private Attribute

    def _funName(self):                 # protected method
        pass

    def __funName(self):                # private method
        pass

# object creation
obj = Student("Baraa",25,555555)
```

To implement proper encapsulation in Python,

we need to use setters and getters.

The getters and setters methods are often used when:

- **When we want to avoid direct access to private variables**
-

```python
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())
```

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**
-

```python
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()
```

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