UNIT IV

Index

1.Introduction to Object Oriented Programming in Python

Object Oriented Programming is a way of computer programming using the idea of "objects" to represents data and methods. It is also, an approach used for creating neat and reusable code instead of a redundant one.

2. Difference between Object-Oriented and Procedural Oriented Programming

Object-Oriented Programming (OOP)	Procedural-Oriented Programming (Pop)
It is a bottom-up approach	It is a top-down approach
Program is divided into objects	Program is divided into functions
Makes use of Access modifiers 'public', private', protected'	Doesn't use Access modifiers
It is more secure	It is less secure
Object can move freely within member functions	Data can move freely from function to function within programs
It supports inheritance	It does not support inheritance

3. What are Classes and Objects?

A class is a collection of objects or you can say it is a blueprint of objects defining the common attributes and behaviours. Now the question arises, how do you do that?

Class is defined under a "Class" Keyword.

Example:

```
class class1(): // class 1 is the name of the class
```

Creating an Object and Class in python:

Example:

```
class employee():
    def    init     (self,name,age,id,salary):     //creating a
        function  self.name = name // self is an instance of a
        class
        self.age = age
        self.salary = salary
        self.id = id

emp1 = employee("harshit",22,1000,1234) //creating objects
emp2 = employee("arjun",23,2000,2234)
print(emp1.    dict     )//Prints dictionary
```

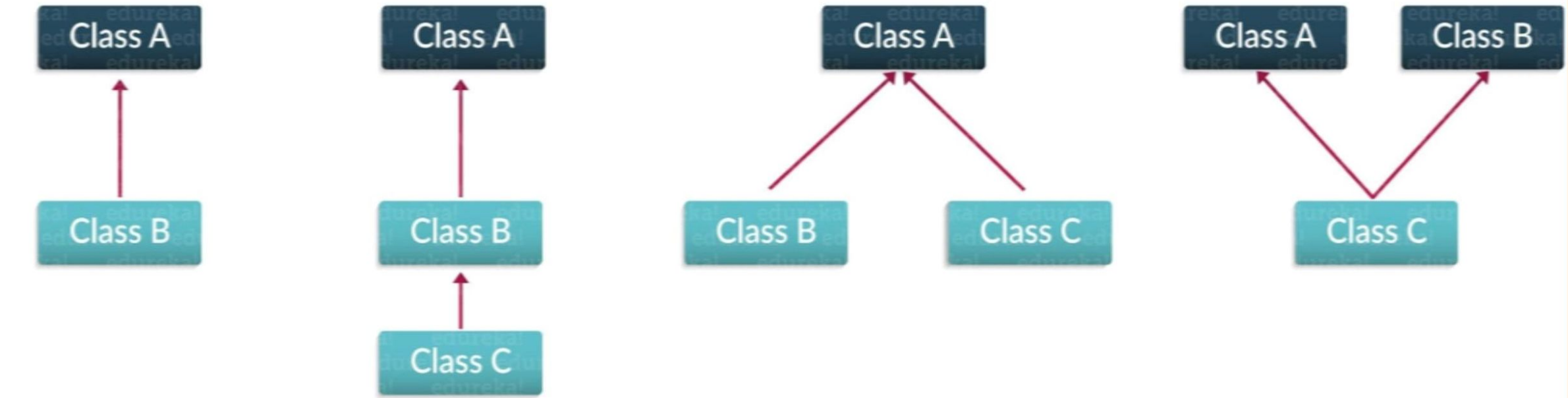
4. Object-Oriented Programming methodologies:

- Inheritance
- Polymorphism
- Encapsulation
- Abstraction

Inheritance:

- **Ever heard of this dialogue from relatives "you look exactly like your father/mother" the reason behind this is called 'inheritance'.**
- **From the Programming aspect, It generally means "inheriting or transfer of characteristics from parent to child class without any modification".**
- **The new class is called the derived/child class and the one from which it is derived is called a parent/base class.**

Types Of Inheritance



Single Inheritance

Multilevel Inheritance

Hierarchical Inheritance

Multiple Inheritance

Single Inheritance:

Single level inheritance enables a derived class to inherit characteristics from a single parent class.

Example:

```
class employee1()://This is a parent class
    def    init     (self, name, age, salary):
        self.name = name
        self.age = age
        self.salary = salary

class childemployee(employee1)://This is a child class
    def    init     (self, name, age, salary,id):
        self.name = name
        self.age = age
        self.salary = salary
        self.id = id
emp1 = employee1('harshit',22,1000)
print(emp1.age)
```

Output: 22

Multilevel Inheritance:

Multi-level inheritance enables a derived class to inherit properties from an immediate parent class which in turn inherits properties from his parent class.

Example:

```
class employee()://Super class
    def init    (self,name,age,salary):
        self.name = name
        self.age = age
        self.salary = salary
class childemployee1(employee)://First child class
    def init    (self,name,age,salary):
        self.name = name
        self.age = age
        self.salary = salary
```

```
class childemployee2(childemployee1)://Second child class
    def  init  (self, name, age, salary):
        self.name = name
        self.age = age
        self.salary = salary
emp1 = employee('harshit',22,1000)
emp2 = childemployee1('arjun',23,2000)
print(emp1.age)
print(emp2.age)
```

- Output: 22,23

Hierarchical Inheritance:

Hierarchical level inheritance enables more than one derived class to inherit properties from a parent class.

Example:

```
class employee():
    def     init     (self, name, age,
        salary):  self.name = name
        self.age = age
        self.salary = salary
```

```
class childemployee1(employee):
    def     init
        (self,name,age,salary):
        self.name = name
        self.age = age
        self.salary = salary

class childemployee2(employee):
    def     init     (self, name, age,
        salary):  self.name = name
        self.age = age
        self.salary = salary
emp1 = employee('harshit',22,1000)
emp2 = employee('arjun',23,2000)
```

Multiple Inheritance:

Multiple level inheritance enables one derived class to inherit properties from more than one base class.

Example:

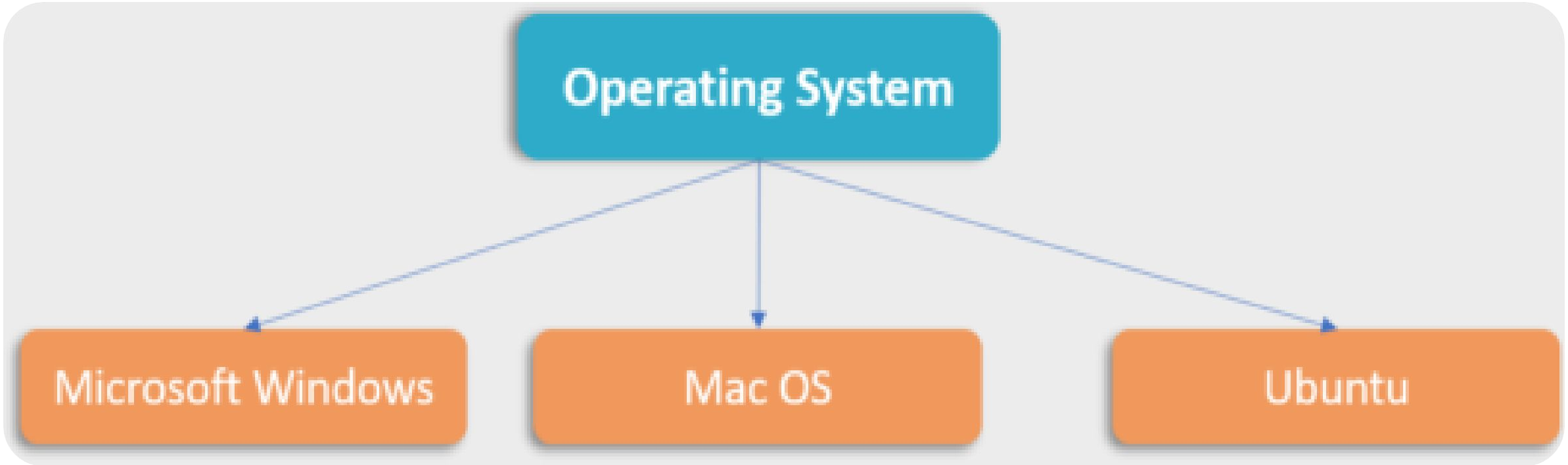
```
class employee1(): //Parent class
    def   init    (self, name, age, salary):
        self.name = name
        self.age = age
        self.salary = salary
```

```
class employee2(): //Parent class
    def init     (self,name,age,salary,id):
     self.name = name
     self.age = age
     self.salary = salary
     self.id = id

class childemployee(employee1,employee2):
    def init     (self, name, age, salary,id):
    self.name = name
     self.age = age
     self.salary = salary
     self.id = id

emp1 = employee1('harshit',22,1000)
emp2 = employee2('arjun',23,2000,1234)
```

Polymorphism:

You all must have used GPS for navigating the route, Isn't it amazing how many different routes you come across for the same destination depending on the traffic, from a programming point of view this is called 'polymorphism'.

It is one such OOP methodology where one task can be performed in several different ways. To put it in simple words, it is a property of an object which allows it to take multiple forms.


Operating System
Microsoft Windows
Mac OS
Ubuntu

Polymorphism is of two types:

- **Compile-time Polymorphism**
- **Run-time Polymorphism**

Compile-time Polymorphism:

A compile-time polymorphism also called as static polymorphism which gets resolved during the compilation time of the program. One common example is "method overloading"

Example:

```
class employee1():
    def name(self):
        print("Harshit is his name")
    def salary(self):
        print("3000 is his salary")
    def age(self):
        print("22 is his age")

class employee2():
    def name(self):
        print("Rahul is his name")
    def salary(self):
        print("4000 is his salary")
    def age(self):
        print("23 is his age")
```

```
def func(obj)://Method Overloading
    obj.name()
    obj.salary()
    obj.age()

obj_emp1 = employee1()
obj_emp2 = employee2()
func(obj_emp1)
func(obj_emp2)
```

Output:

```
Harshit is his name
3000 is his salary
22 is his age
Rahul is his name
4000 is his salary
23 is his age
```

Run-time Polymorphism:

A run-time Polymorphism is also, called as dynamic polymorphism where it gets resolved into the run time. One common example of Run-time polymorphism is "method overriding".

Example:

```
class employee():
    def     init
        (self,name,age,id,salary):
        self.name = name
        self.age = age
        self.salary = salary
        self.id = id
    def earn(self):
        pass

class childemployee1(employee):
    def earn(self): //Run-time polymorphism
        print("no money")
```

```
class childemployee2(employee):
    def earn(self):
        print("has money")

c = childemployee1
c.earn(employee)
d = childemployee2
d.earn(employee)
```

Output: no money, has money

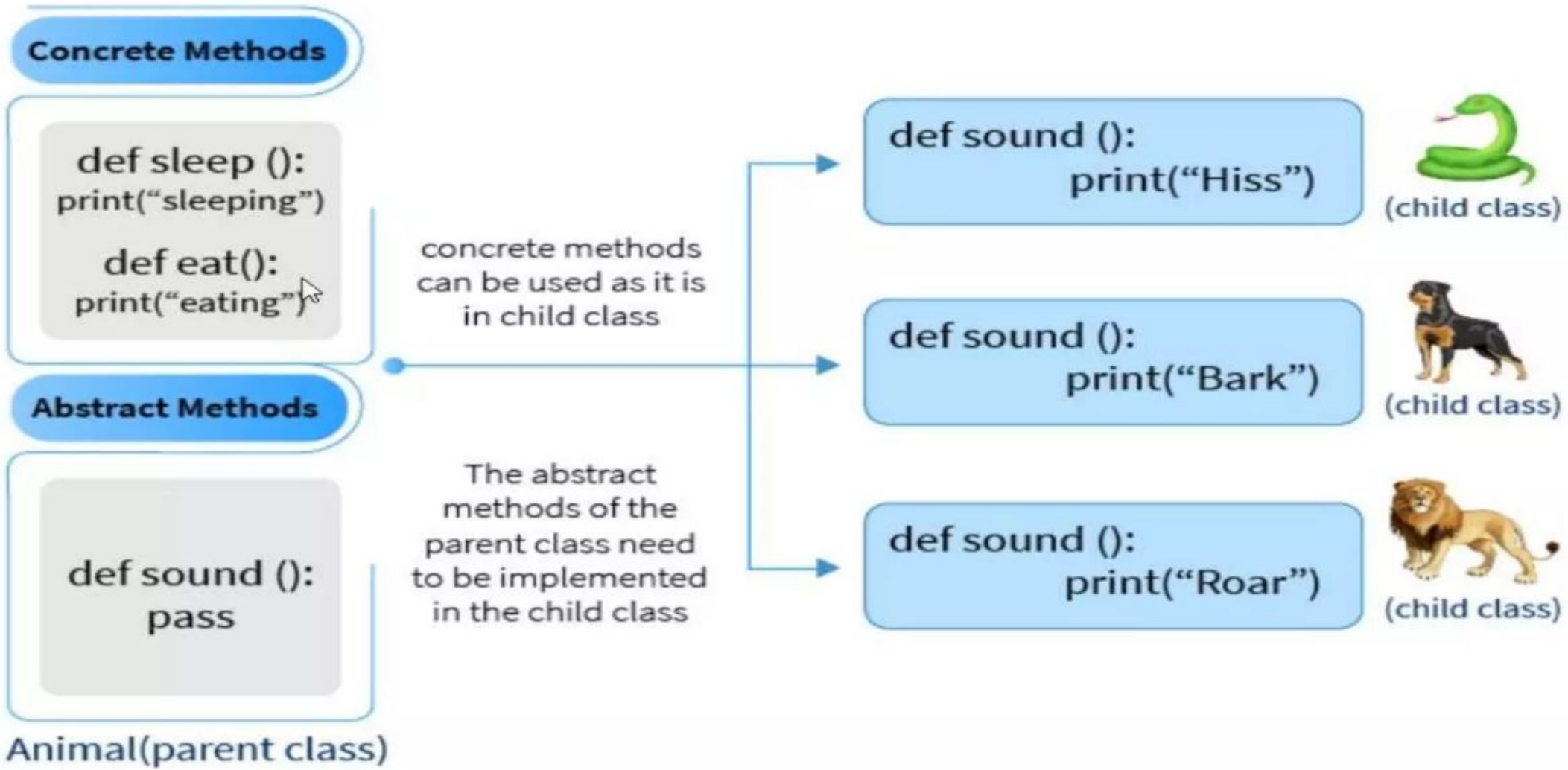
Abstraction:

- A class that consists of one or more abstract method is called the abstract class. Abstract methods do not contain their implementation
- Abstract class can be inherited by the subclass and abstract method gets its definition in the subclass. Abstraction classes are meant to be the blueprint of the other class.

Syntax

1.from abc **import** ABC
2.class ClassName(ABC):
We import the ABC class from the **abc** module.

Example of Data Abstraction



Program:

```
# Python program demonstrate
# abstract base class work
from abc import ABC, abstractmethod
class Car(ABC):
    def mileage(self):
        pass

class Tesla(Car):
    def mileage(self):
        print("The mileage is 30kmph")
class Suzuki(Car):
    def mileage(self):
        print("The mileage is 25kmph ")
class Duster(Car):
     def mileage(self):
          print("The mileage is 24kmph ")
```

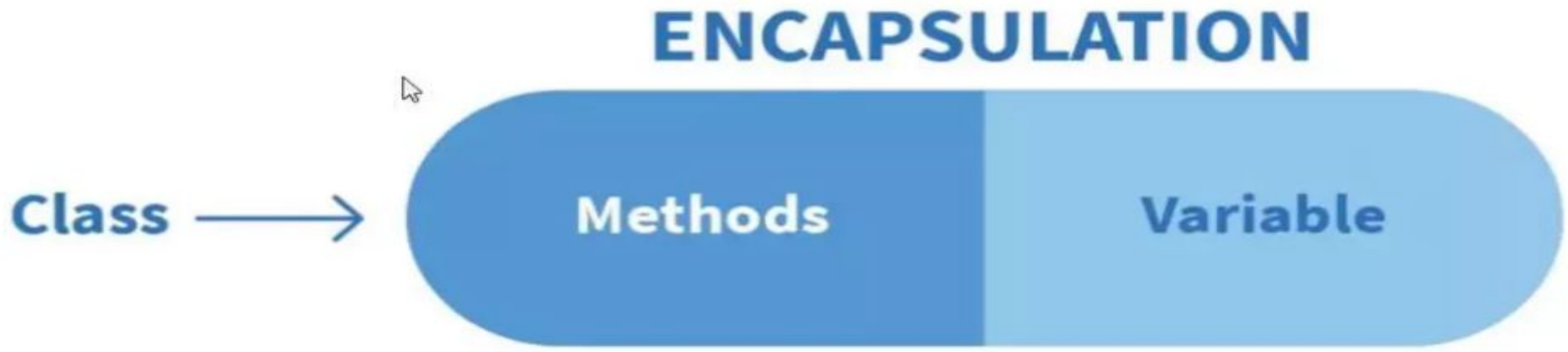
```
class Renault(Car):
    def mileage(self):
            print("The mileage is 27kmph ")

# Driver code
t= Tesla ()
t.mileage()

r = Renault()
r.mileage()

s = Suzuki()
s.mileage()
d = Duster()
d.mileage()
```

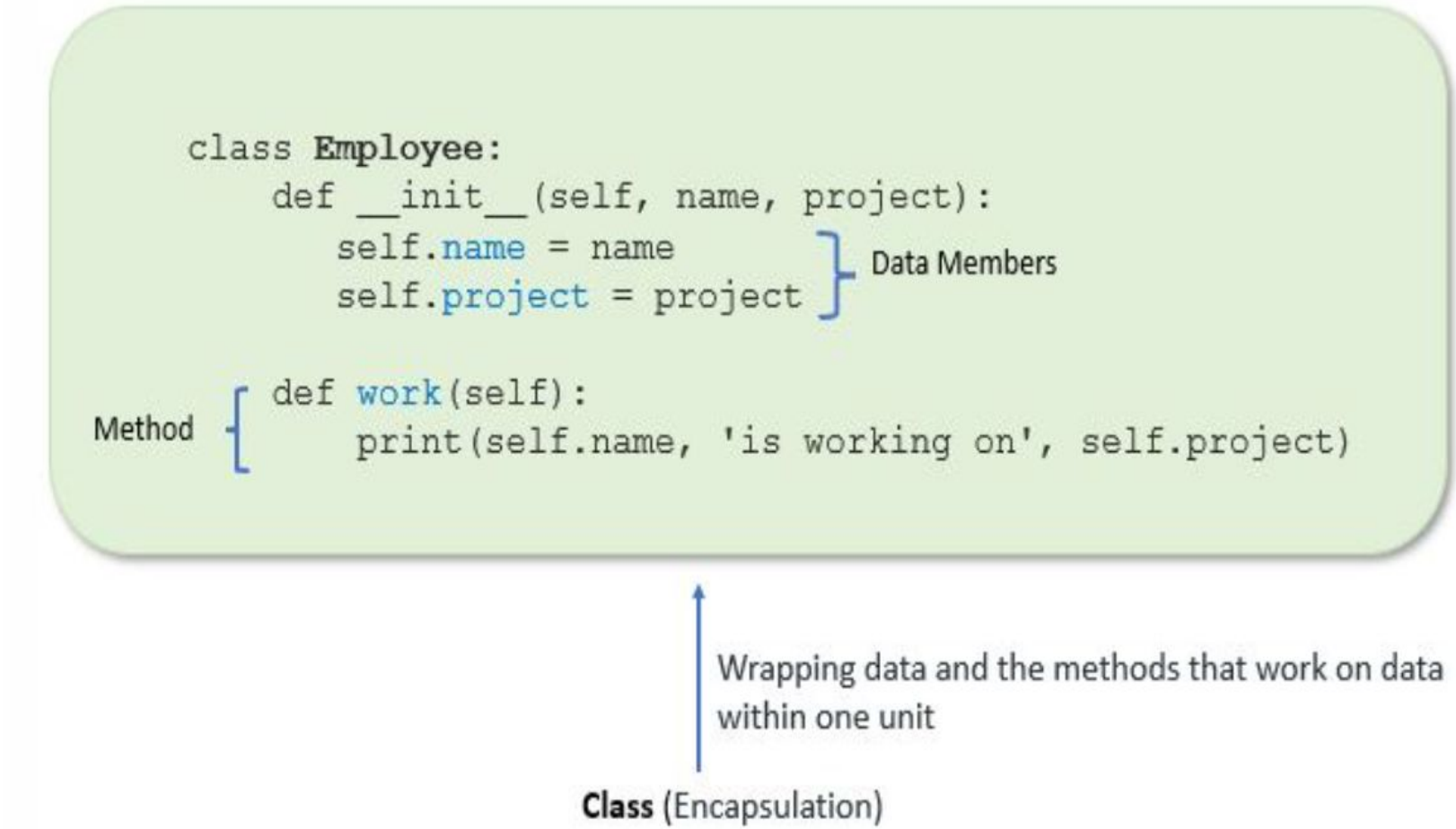
OUTPUT:

The mileage is 30kmph
The mileage is 27kmph
The mileage is 25kmph
The mileage is 24kmph

Encapsulation:

- Encapsulation in Python describes the concept of **bundling data and methods within a single unit**.





Implement encapsulation using a class

Access Modifiers in Python

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

Protected Members

```
Class Base1:
    def __init__(self):

        # the protected member
        self._p = 78

# here, we will create the derived class
class Derived1(Base):
    def __init__(self):

# now, we will call the constructor of Base class
        Base1.__init__(self)
        print ("We will call the protected member of base class: ",
            self._p)
```

```
# Now, we will be modifing the protected variable:
        self._p = 433
        print ("we will call the modified protected member outside the class: ",
            self._p)


obj_1 = Derived1()

obj_2 = Base1()

# here, we will call the protected member
# this can be accessed but it should not be done because of convention
print ("Access the protected member of obj_1: ", obj_1._p)

# here, we will access the protected variable outside
print ("Access the protected member of obj_2: ", obj_2._p)
```

OUTPUT:

```
We will call the protected member of base class: 78
we will call the modified protected member outside the class: 433
Access the protected member of obj_1: 433
Access the protected member of obj_2: 78
```

Private Members

- Private members are the same as protected members.

- The difference is that class members who have been declared private should not be accessed by anyone outside the class or any base classes.

- Python does not have Private instance variable variables that can be accessed outside of a class.

```
class Base1:
    def __init__(self):
        self.p = “Pythonpoint"
        self.__q = “Pythonpoint"

# Creating a derived class
class Derived1(Base1):
     def __init__(self):

# Calling constructor of
# Base class
    Base1.__init__(self)
    print("We will call the private member of base class: ")
    print(self.__q)


# Driver code
obj_1 = Base1()
print(obj_1.p)
```

OUTPUT:

PythonPoint