

UNIT IV: Python classes and objects

There are five important features related to Object Oriented Programming System. They are:

1. **Classes and objects**
2. **Encapsulation**
3. **Abstraction**
4. **Inheritance**
5. **Polymorphism**

1. Class and Objects in Python?

- **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.

Objects have two characteristics: They have **states and behaviors** (object has attributes and methods attached to it) Attributes represent its state, and methods represent its behavior. Using its methods, we can modify its state.

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**.

Python is an Object-Oriented Programming language, so everything in Python is treated as **an object**. An object is a real-life entity. It is the collection of various data and functions that operate on those data.

For example, if we design a class based on the states and behaviors of a Person, then **States** can be represented as **instance variables** and **behaviors** as **class methods**.

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: roshani

- **State:**
 - Name: roshani
 - 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: sunil

- **State:**
 - Name: sunil

- Sex: Male
- Profession: Doctor

Behavior:

- Working: He is working as a doctor
- Study: He studies 5 hours a day

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

Create a Class in Python

In Python, class is defined by using the **class** keyword. The syntax to create a class is given below.

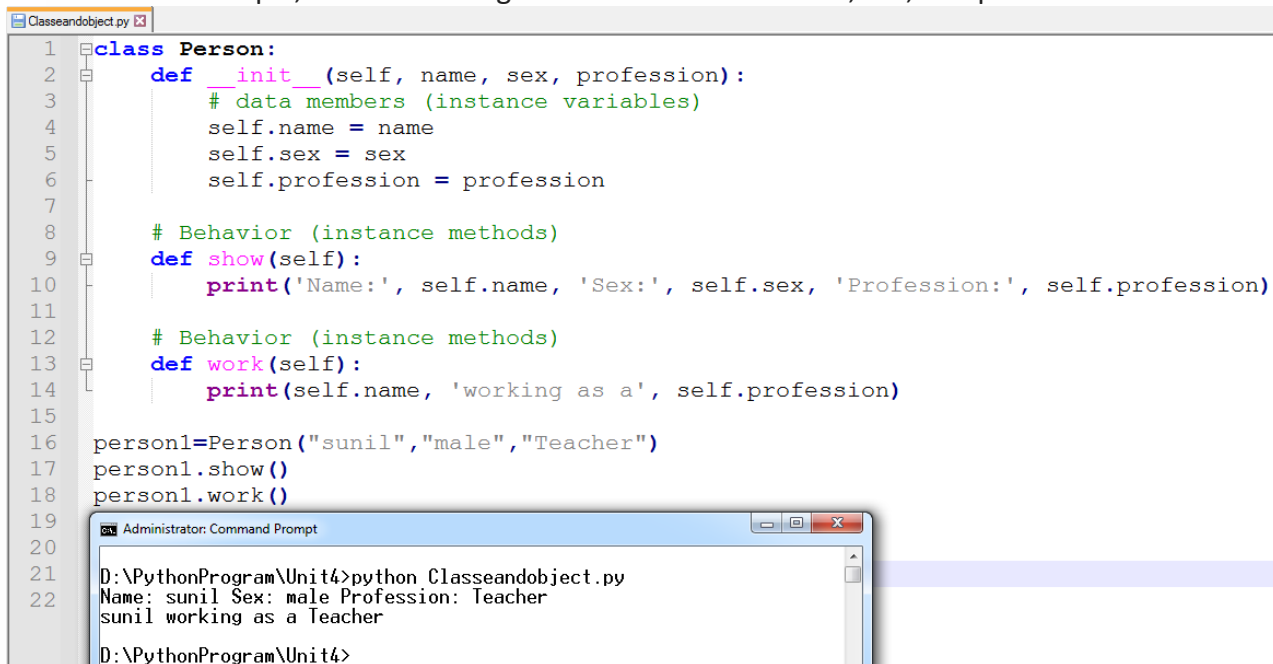
Syntax

```
class class_name:
    '''This is a docstring. I have created a new class'''
    <statement 1>
    <statement 2>
    .
    .
    <statement N>
```

- **class_name**: It is the name of the class
- **Docstring**: It is the first string inside the class and has a brief description of the class. Although not mandatory, this is highly recommended.
- **statements**: Attributes and methods

Example: Define a class in Python

- In this example, we are creating a Person Class with name, sex, and profession instance variables.



```
1 class Person:
2     def __init__(self, name, sex, profession):
3         # data members (instance variables)
4         self.name = name
5         self.sex = sex
6         self.profession = profession
7
8         # Behavior (instance methods)
9     def show(self):
10        print('Name:', self.name, 'Sex:', self.sex, 'Profession:', self.profession)
11
12        # Behavior (instance methods)
13    def work(self):
14        print(self.name, 'working as a', self.profession)
15
16 person1=Person("sunil","male","Teacher")
17 person1.show()
18 person1.work()
19
20
21
22
```

Administrator: Command Prompt

```
D:\PythonProgram\Unit4>python Classeandobject.py
Name: sunil Sex: male Profession: Teacher
sunil working as a Teacher
D:\PythonProgram\Unit4>
```

Create Object of a Class

An object is essential to work with the class attributes. The object is created using the class name. When we **create an object of the class, it is called instantiation**. The object is also called the instance of a class.

A [constructor](#) is a special method used to **create and initialize** an object of a class. This method is defined in the class.

In Python, Object creation is divided into two parts in **Object Creation** and **Object initialization**

- Internally, the `__new__` is the method that creates the object
- And, using the `__init__()` method we can implement **constructor to initialize the object**.

Syntax

<object-name> = <class-name>(<arguments>)

Below is the code to create the object of a Person class

```
sunil = Person('sunil', 'male', 'Teacher')
```

Class Attributes

When we design a class, we use instance variables and class variables.

In Class, attributes can be defined into two parts:

- [Instance variables](#): The instance variables are attributes attached to an instance of a class. We define instance variables in the constructor (the `__init__()` method of a class).
- [Class Variables](#): A class variable is a variable that is declared inside of class, but outside of any instance method or `__init__()` method.

Objects do not share instance attributes. Instead, every object has its copy of the instance attribute and is unique to each object.

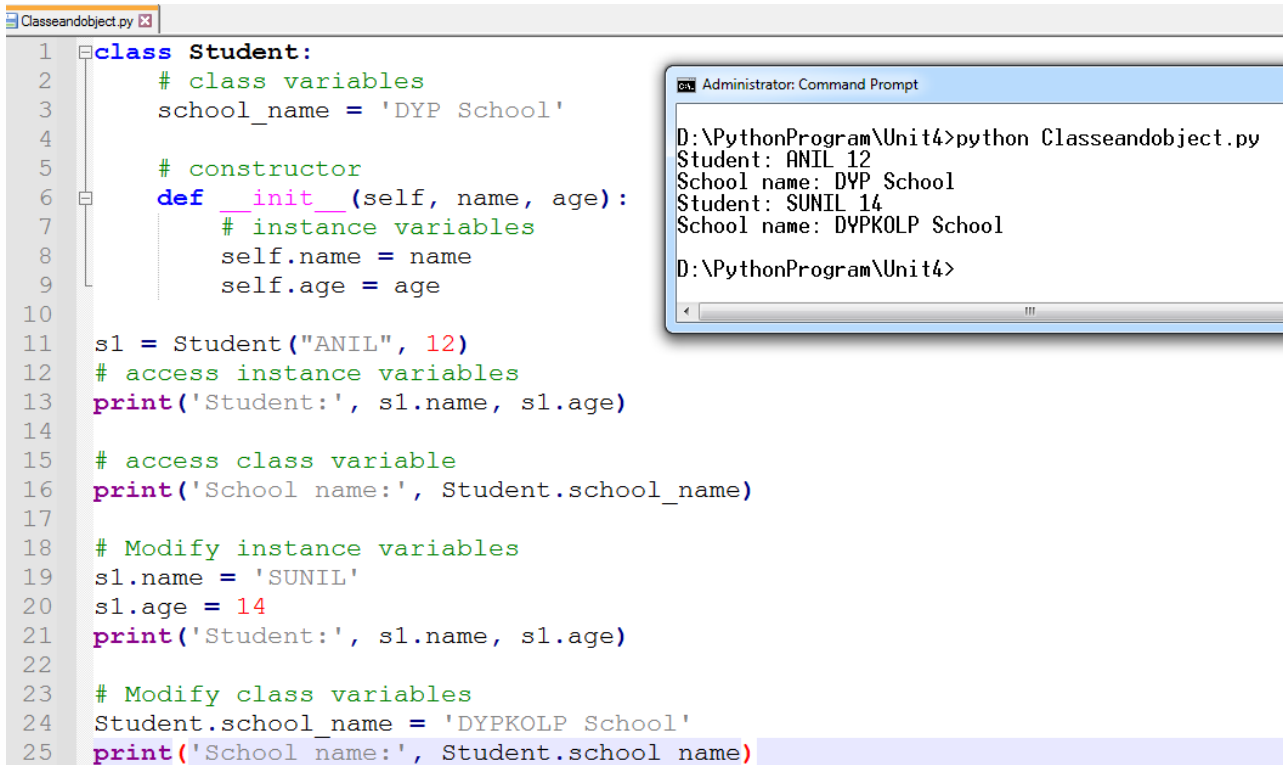
All instances of a class share the **class variables**. However, unlike instance variables, the value of a class variable is not varied from object to object.

Only one copy of the **static variable** will be created and shared between all objects of the class.

Accessing properties and assigning values

- An instance attribute can be **accessed or modified by using the dot notation: `instance_name.attribute_name`**.
- A class variable is **accessed or modified using the class name**

Example



The screenshot shows a Python script named 'Classeandobject.py' and its execution output in a Command Prompt window.

```

1 class Student:
2     # class variables
3     school_name = 'DYP School'
4
5     # constructor
6     def __init__(self, name, age):
7         # instance variables
8         self.name = name
9         self.age = age
10
11 s1 = Student("ANIL", 12)
12 # access instance variables
13 print('Student:', s1.name, s1.age)
14
15 # access class variable
16 print('School name:', Student.school_name)
17
18 # Modify instance variables
19 s1.name = 'SUNIL'
20 s1.age = 14
21 print('Student:', s1.name, s1.age)
22
23 # Modify class variables
24 Student.school_name = 'DYPKOLP School'
25 print('School name:', Student.school_name)

```

The Command Prompt output shows the execution of the script:

```

D:\PythonProgram\Unit4>python Classeandobject.py
Student: ANIL 12
School name: DYP School
Student: SUNIL 14
School name: DYPKOLP School
D:\PythonProgram\Unit4>

```

Class Methods

In [Object-oriented programming](#), Inside a Class, we can define the following three types of methods.

- **Instance method**: Used to **access or modify the object state**. If we use [instance variables](#) inside a method, such methods are called instance methods.
- **Class method**: Used to **access or modify the class state**. In method implementation, if we use only [class variables](#), then such type of methods we should declare as a class method.
- **Static method**: It is a general utility method that performs a task in isolation. Inside this method, we don't use instance or class variable because this **static method doesn't have access to the class attributes**.
 - Instance methods work on the **instance level (object level)**. For example, if we have two objects created from the student class, **They may have different names, marks, roll numbers, etc. Using instance methods, we can access and modify the instance variables.**
 - **A class method is bound to the class** and not the object of the class. It can access only class variables.
 - **Example**: Define and call an instance method and class method

```

1  # class methods demo
2  class Student:
3      # class variable
4      school_name = 'DYP School'
5      # constructor
6      def __init__(self, name, age):
7          # instance variables
8          self.name = name
9          self.age = age
10     # instance method
11     def show(self):
12         # access instance variables and class variables
13         print('Student:', self.name, self.age, Student.school_name)
14     # instance method
15     def change_age(self, new_age):
16         # modify instance variable
17         self.age = new_age
18     # class method
19     @classmethod
20     def modify_school_name(cls, new_name):
21         # modify class variable
22         cls.school_name = new_name
23 s1 = Student("SUNIL", 12)
24 # call instance methods
25 s1.show()
26 s1.change_age(14)
27 # call class method
28 Student.modify_school_name('DYPKOLP School')
29 # call instance methods
30 s1.show()

```

Administrator: Command Prompt

```

D:\PythonProgram\Unit4>python Classeandobject.py
Student: SUNIL 12 DYP School
Student: SUNIL 14 DYPKOLP School
D:\PythonProgram\Unit4>

```

Class Naming Convention

Naming conventions are essential in any programming language for better readability. If we give a sensible name, it will save our time and energy later. Writing readable code is one of the guiding principles of the Python language.

We should follow specific rules while we are deciding a name for the class in Python.

Naming Styles

Type	Naming Convention	Examples
Class	Start each word with a capital letter. Do not separate words with underscores. This style is called camel case.	Model , MyClass
Method	Use a lowercase word or words. Separate words with underscores to improve readability.	class_method , method

Python's built-in classes are typically **lowercase words**

pass Statement in Class

In Python, the `pass` is a null statement. Therefore, nothing happens when the `pass` statement is executed. The `pass` statement is used to have an empty block in a code because the empty code is not allowed in loops, function definition, class definition. Thus, the `pass` statement will result in no operation (NOP). Generally, we use it as a placeholder when we do not know what code to write or add code in a future release.

For example, suppose we have a class that is not implemented yet, but we want to implement it in the future, and they cannot have an empty body because the interpreter gives an error. So use the `pass` statement to construct a body that does nothing.

Example

class Demo:

pass

In the above example, we defined class without a body. To avoid errors while executing it, we added the `pass` statement in the class body.

Object Properties

Every object has properties with it. In other words, we can say that object property is an association between **name** and **value**.

For example, a car is an object, and its properties are car **color, sunroof, price, manufacture, model, engine**, and so on. Here, color is the name and red is the **value**. Object properties are nothing but **instance variables**.

Modify Object Properties

Every object has properties associated with them. We can set or modify the object's properties after object initialization by calling the property directly using the dot operator.

Obj.PROPERTY=value

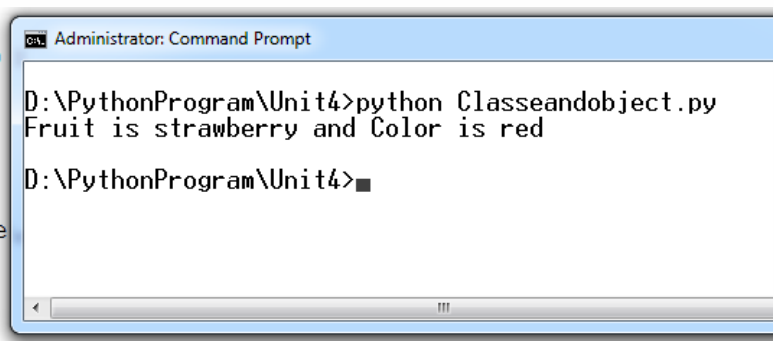
```
class Fruit:
    def __init__(self, name, color):
        self.name = name
        self.color = color

    def show(self):
        print("Fruit is", self.name)
```

```
# creating object of the class
obj = Fruit("Apple", "red")
```

```
# Modifying Object Properties
obj.name = "strawberry"
```

```
# calling the instance method using the object obj
obj.show()
# Output Fruit is strawberry and Color is red
```



What is Constructor in Python?

In [object-oriented programming](#), A constructor is a special method used to create and initialize an object of a [class](#). This method is defined in the class.

- The constructor is **executed automatically at the time of object creation**.
- The primary use of a constructor is **to declare and initialize data member/ [instance variables](#) of a class**. The constructor contains a **collection of statements (i.e., instructions) that executes at the time of object creation to initialize the attributes of an object**.

For example, when we execute `obj = Sample()`, Python gets to know that obj is an object of class `Sample` and calls the constructor of that class to create an object.

Note: In Python, internally, the `__new__` is the method that creates the object, and `__del__` method is called to destroy the object when the reference count for that object becomes zero.

In Python, Object creation is divided into two parts in **Object Creation** and **Object initialization**

- Internally, the `__new__` is the method that creates the object
- And, using the `__init__()` method we can implement constructor to initialize the object.

Syntax of a constructor

```
def __init__(self):  
    # body of the constructor
```

Where,

- `def`: The keyword is used to define function.
- `__init__()` Method: It is a reserved method. This method gets called as soon as an object of a class is instantiated.
- `self`: The first argument `self` refers to the **current object**. It binds the instance to the `__init__()` method. It's usually named `self` to follow the naming convention.

Note: The `__init__()` method arguments are optional. We can define a constructor with any number of arguments.

Example: Create a Constructor in Python

In this example, we'll create a Class **Student** with an instance variable student name. we'll see how to use a constructor to initialize the student name at the time of object creation.

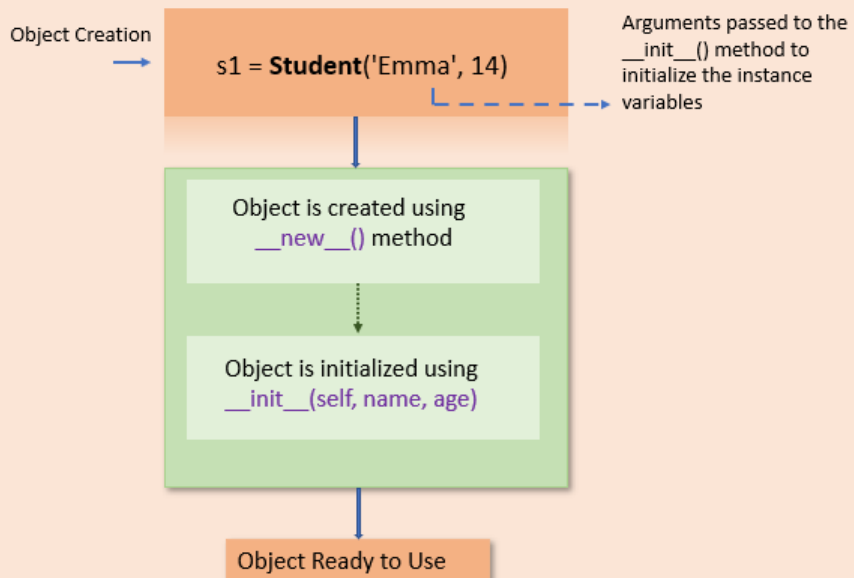
```
1 class Student:
2     # constructor
3     # initialize instance variable
4     def __init__(self, name):
5         print('Inside Constructor')
6         self.name = name
7         print('All variables initialized')
8
9     # instance Method
10    def show(self):
11        print('Hello, my name is', self.name)
12
13    # create object using constructor
14    s1 = Student('SUNIL')
15    s1.show()
16
```

Administrator: Command Prompt

```
D:\PythonProgram\Unit4>python constructor.py
Inside Constructor
All variables initialized
Hello, my name is SUNIL
D:\PythonProgram\Unit4>
```

- In the above example, an object `s1` is created using the constructor
- While creating a Student object `name` is passed as an argument to the `__init__()` method to initialize the object.
- Similarly, various objects of the Student class can be created by passing different names as arguments.

Object Creation in Python



Note:

- For every object, the constructor will be executed only once. For example, if we create four objects, the constructor is called four times.
- In Python, every class has a constructor, but it's not required to define it explicitly. Defining constructors in class is optional.
- Python will provide a default constructor if no constructor is defined.

Types of Constructors

In Python, we have the following three types of constructors.

- Default Constructor
- Non-parametrized constructor
- Parameterized constructor

Default Constructor

Python will provide a default constructor if no constructor is defined. Python adds a default constructor when we do not include the constructor in the class or forget to declare it. **It does not perform any task but initializes the objects.** It is an empty constructor without a body.

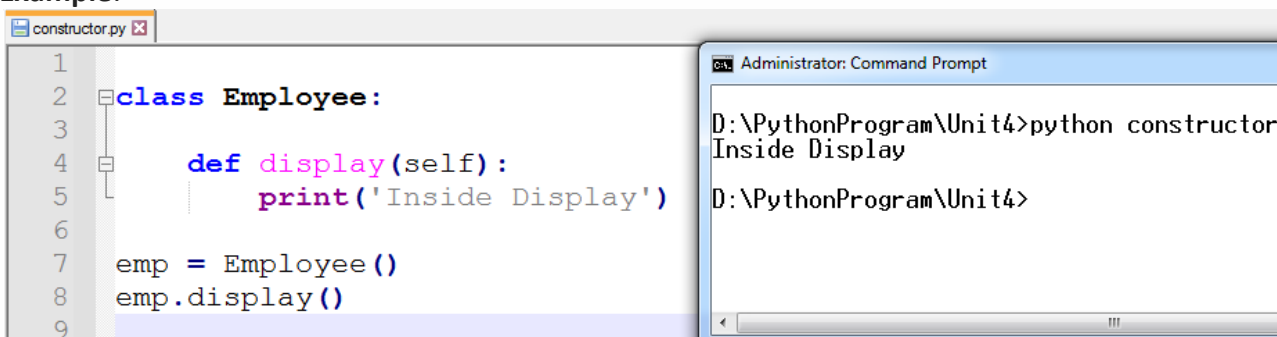
If you do not implement any constructor in your class or forget to declare it, the Python inserts a **default constructor** into your code on your behalf. This constructor is known as the default constructor.

It does not perform any task but initializes the objects. It is an empty constructor without a body.

Note:

- The default constructor is not present in the source py file. It is inserted into the code during compilation if not exists. See the below image.
- If you implement your constructor, then the default constructor will not be added.

Example:



```
1
2 class Employee:
3
4     def display(self):
5         print('Inside Display')
6
7 emp = Employee()
8 emp.display()
9
```

```
D:\PythonProgram\Unit4>python constructor
Inside Display
D:\PythonProgram\Unit4>
```

As you can see in the example, we do not have a constructor, but we can still create an object for the class because Python added the default constructor during a program compilation.

Non-Parametrized Constructor

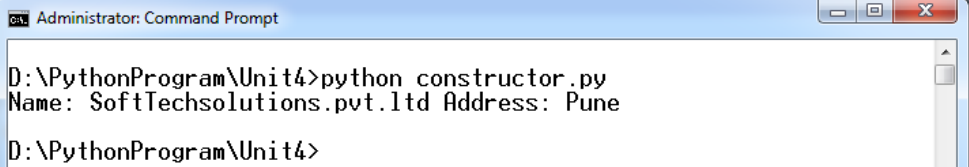
A constructor without any arguments is called a non-parameterized constructor. This type of constructor is used to initialize each object with default values.

This constructor doesn't accept the arguments during object creation. Instead, it initializes every object with the same set of values.

```

1 class Company:
2     # no-argument constructor
3     def __init__(self):
4         self.name = "SoftTechsolutions.pvt.ltd"
5         self.address = "Pune"
6
7     # a method for printing data members
8     def show(self):
9         print('Name:', self.name, 'Address:', self.address)
10
11 # creating object of the class
12 cmp = Company()
13
14 # calling the instance method using the object
15 cmp.show()

```



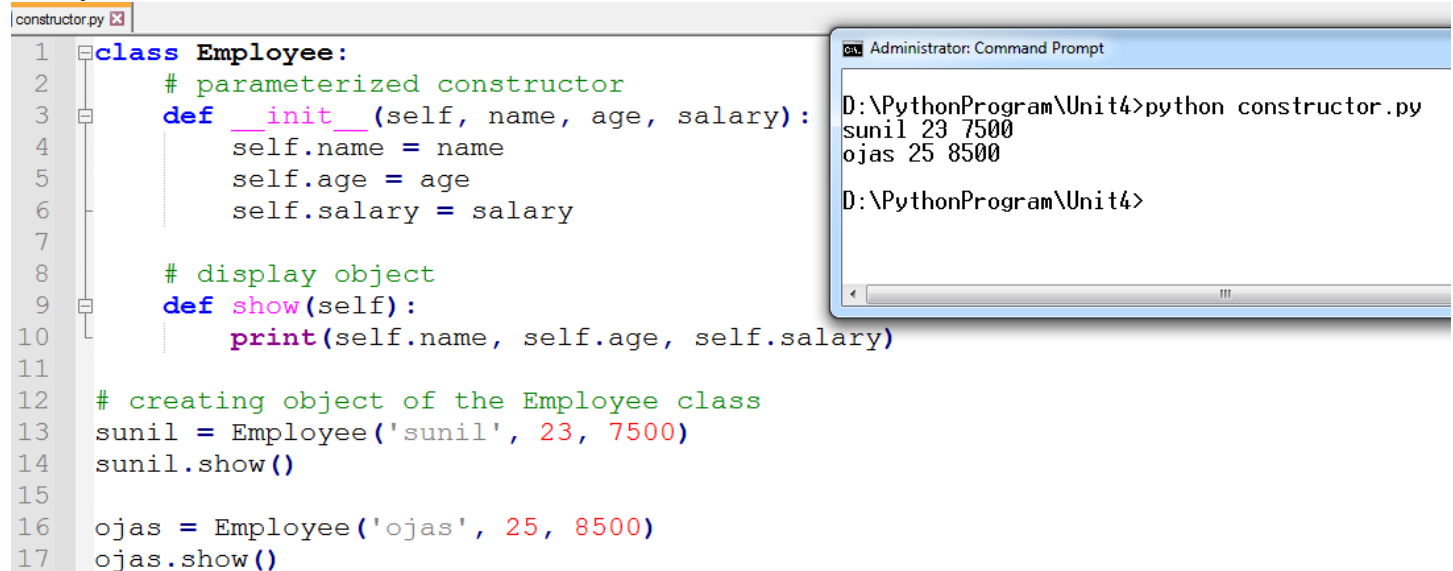
As you can see in the example, we do not send any argument to a constructor while creating an object.

Parameterized Constructor

A constructor with defined parameters or arguments is called a parameterized constructor. We can pass different values to each object at the time of creation using a parameterized constructor.

The first parameter to constructor is `self` that is a reference to the being constructed, and the rest of the arguments are provided by the programmer. A parameterized constructor can have any number of arguments. For example, consider a company that contains thousands of employees. In this case, while creating each employee object, we need to pass a different name, age, and salary. In such cases, use the parameterized constructor.

Example:



```

1 class Employee:
2     # parameterized constructor
3     def __init__(self, name, age, salary):
4         self.name = name
5         self.age = age
6         self.salary = salary
7
8     # display object
9     def show(self):
10        print(self.name, self.age, self.salary)
11
12 # creating object of the Employee class
13 sunil = Employee('sunil', 23, 7500)
14 sunil.show()
15
16 ojas = Employee('ojas', 25, 8500)
17 ojas.show()

```

Output from Command Prompt:

```

D:\PythonProgram\Unit4>python constructor.py
sunil 23 7500
ojas 25 8500
D:\PythonProgram\Unit4>

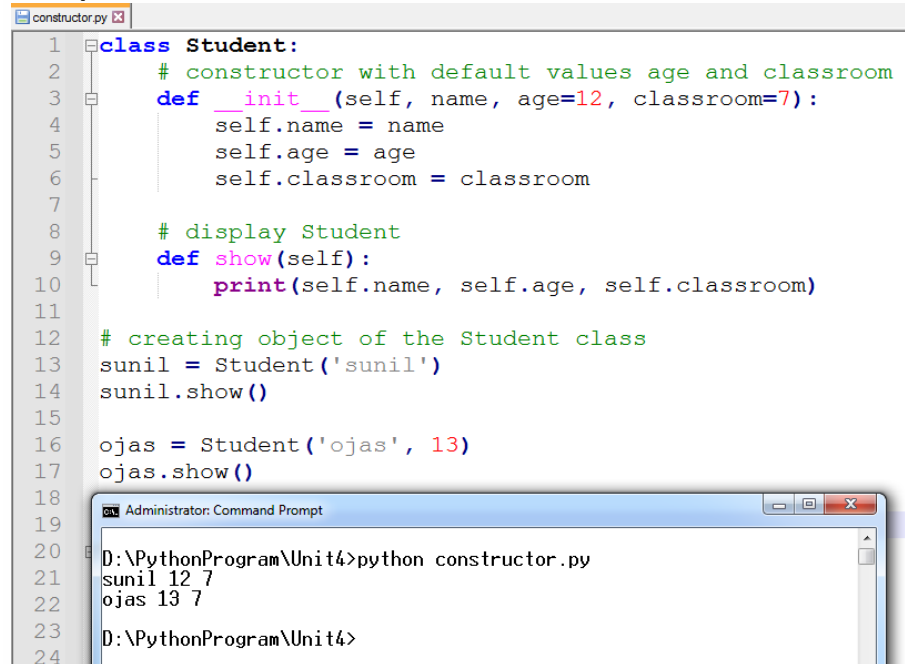
```

Constructor with Default Values

Python allows us to define a **constructor with default values**. The default value will be used if we do not pass arguments to the constructor at the time of object creation.

The following example shows how to use the default values with the constructor.

Example



The image shows a code editor window titled 'constructor.py' containing the following Python code:

```
1 class Student:
2     # constructor with default values age and classroom
3     def __init__(self, name, age=12, classroom=7):
4         self.name = name
5         self.age = age
6         self.classroom = classroom
7
8     # display Student
9     def show(self):
10        print(self.name, self.age, self.classroom)
11
12 # creating object of the Student class
13 sunil = Student('sunil')
14 sunil.show()
15
16 ojas = Student('ojas', 13)
17 ojas.show()
18
19
20
21
22
23
24
```

Below the code editor is a screenshot of a Windows Command Prompt window titled 'Administrator: Command Prompt'. It shows the execution of the script:

```
D:\PythonProgram\Unit4>python constructor.py
sunil 12 7
ojas 13 7
D:\PythonProgram\Unit4>
```

As you can see, we didn't pass the age and classroom value at the time of object creation, so default values are used.

Self Keyword in Python

As you all know, the class contains instance variables and methods. Whenever we define instance methods for a class, we use `self` as the first parameter. Using `self`, we can access the [instance variable](#) and [instance method](#) of the object.

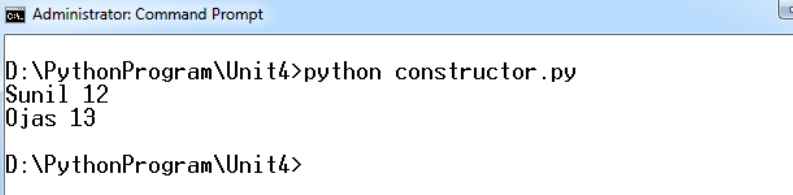
The first argument `self` refers to the current object.

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.

Example

```
1 class Student:
2     # constructor
3     def __init__(self, name, age):
4         self.name = name
5         self.age = age
6
7     # self points to the current object
8     def show(self):
9         # access instance variable using self
10        print(self.name, self.age)
11
12    # creating first object
13    sunil = Student('Sunil', 12)
14    sunil.show()
15
16    # creating Second object
17    ojas = Student('Ojas', 13)
18    ojas.show()
19
20
21
22
23
24
25
```



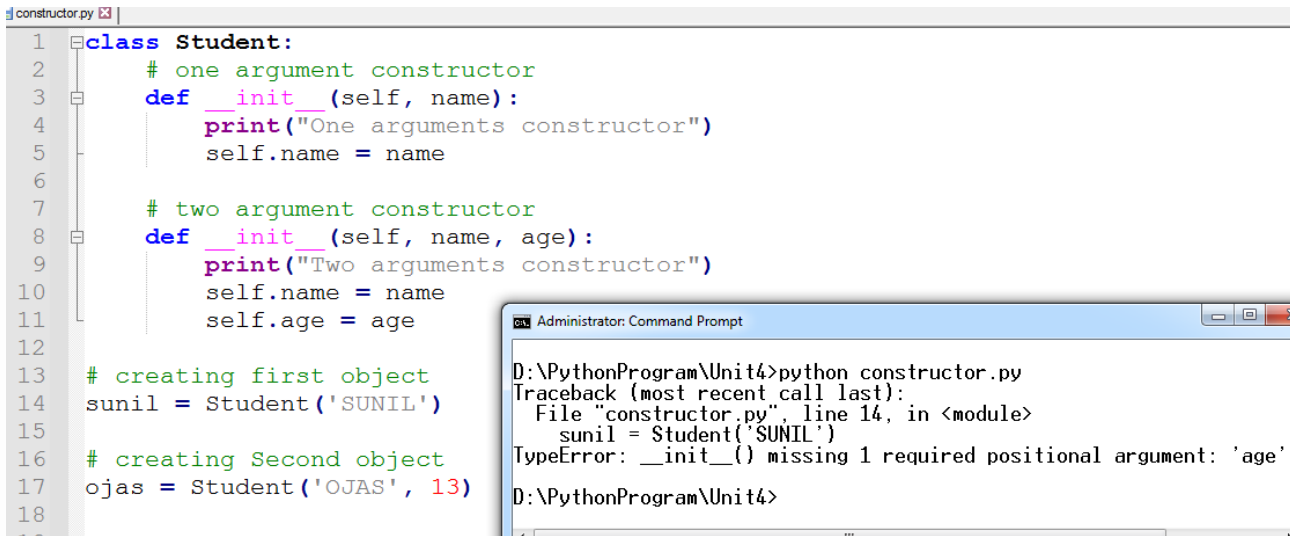
Constructor Overloading

Constructor overloading is a concept of **having more than one constructor with a different parameters** list in such a way so that each constructor can perform different tasks.

For example, we **can create a three constructor** which accepts a different set of parameters

Python does not support constructor overloading. If we define multiple constructors then, the interpreter will considers only the last constructor and throws an error if the sequence of the arguments doesn't match as per the last constructor. The following example shows the same.

Example



```
1 class Student:
2     # one argument constructor
3     def __init__(self, name):
4         print("One arguments constructor")
5         self.name = name
6
7     # two argument constructor
8     def __init__(self, name, age):
9         print("Two arguments constructor")
10        self.name = name
11        self.age = age
12
13 # creating first object
14 sunil = Student('SUNIL')
15
16 # creating Second object
17 ojas = Student('OJAS', 13)
18
```

```
Administrator: Command Prompt
D:\PythonProgram\Unit4>python constructor.py
Traceback (most recent call last):
  File "constructor.py", line 14, in <module>
    sunil = Student('SUNIL')
TypeError: __init__() missing 1 required positional argument: 'age'
D:\PythonProgram\Unit4>
```

- As you can see in the above example, we defined multiple constructors with different arguments.
- At the time of object creation, the interpreter executed the second constructor because Python always considers the last constructor.
- Internally, the object of the class will always call the last constructor, even if the class has multiple constructors.
- In the example when we called a constructor only with one argument, we got a type error.

Constructor Chaining

Constructors are used for instantiating an object. The task of the constructor is to assign value to data members when an object of the class is created.

Constructor chaining is the process of calling one constructor from another constructor. Constructor chaining is useful when you want to **invoke multiple constructors, one after another, by initializing only one instance.**

In Python, **constructor chaining is convenient when we are dealing with [inheritance](#).** When an instance of a child class is initialized, the constructors of all the parent classes are first invoked and then, in the end, the constructor of the child class is invoked.

Using the `super()` method we can invoke the parent class constructor from a child class.

Example

```

1 class Vehicle:
2     # Constructor of Vehicle
3     def __init__(self, engine):
4         print('Inside Vehicle Constructor')
5         self.engine = engine
6
7 class Car(Vehicle):
8     # Constructor of Car
9     def __init__(self, engine, max_speed):
10        super().__init__(engine)
11        print('Inside Car Constructor')
12        self.max_speed = max_speed
13
14 class Electric_Car(Car):
15     # Constructor of Electric Car
16     def __init__(self, engine, max_speed, km_range):
17         super().__init__(engine, max_speed)
18         print('Inside Electric Car Constructor')
19         self.km_range = km_range
20
21 # Object of electric car
22 ev = Electric_Car('1500cc', 240, 750)
23 print(ev.engine, ev.max_speed, ev.km_range)

```

Administrator: Command Prompt

```

D:\PythonProgram\Unit4>python constructor.py
Inside Vehicle Constructor
Inside Car Constructor
Inside Electric Car Constructor
1500cc 240 750

D:\PythonProgram\Unit4>

```

Counting the Number of objects of a Class

The constructor executes when we create the object of the class. For every object, the constructor is called only once. So for counting the number of objects of a class, we can add a counter in the constructor, which increments by one after each object creation.

Example

```

1 class Employee:
2     count = 0
3     def __init__(self):
4         Employee.count = Employee.count + 1
5
6
7
8 # creating objects
9 e1 = Employee()
10 e2 = Employee()
11 e2 = Employee()
12 print("The number of Employee:", Employee.count)

```

Administrator: Command Prompt

```

D:\PythonProgram\Unit4>python constructor.py
The number of Employee: 3

D:\PythonProgram\Unit4>

```

Constructor Return Value

In Python, the constructor does not return any value. Therefore, while declaring a constructor, we don't have anything like return type. Instead, a constructor is implicitly called at the time of object instantiation. Thus, it has the sole purpose of initializing the instance variables.

The `__init__()` is required to return None. We can not return something else. If we try to return a non-None value from the `__init__()` method, it will raise `TypeError`.

Example

```

1
2 class Test:
3
4     def __init__(self, i):
5         self.id = i
6         return True
7
8 d = Test(10)
9
Administrator: Command Prompt
D:\PythonProgram\Unit4>python constructor.py
Traceback (most recent call last):
  File "constructor.py", line 8, in <module>
    d = Test(10)
TypeError: __init__() should return None, not 'bool'
D:\PythonProgram\Unit4>

```

Conclusion and Quick recap

- A constructor is a **unique method used** to initialize an object of the class.
- Python will provide a **default constructor** if no constructor is defined.
- Constructor **is not a method** and doesn't return anything. it returns None
- In Python, we have three types of constructor **default, Non-parametrized, and parameterized constructor**.
- Using self, we can access the **instance variable and instance method** of the object. The first argument self refers to the current object.
- **Constructor overloading** is not possible in Python.
- If the parent class **doesn't have a default constructor**, then the compiler would not insert a default constructor in the child class.
- A child class constructor can also invoke the parent class constructor using the `super()` method.

What is Destructor in Python?

In object-oriented programming, A **destructor is called when an object is deleted or destroyed**. Destructor is used to perform **the clean-up activity before destroying the object**, such as **closing database connections or filehandle**.

Python has a **garbage collector** that handles **memory management automatically**. For example, it cleans up the memory when an object goes out of scope.

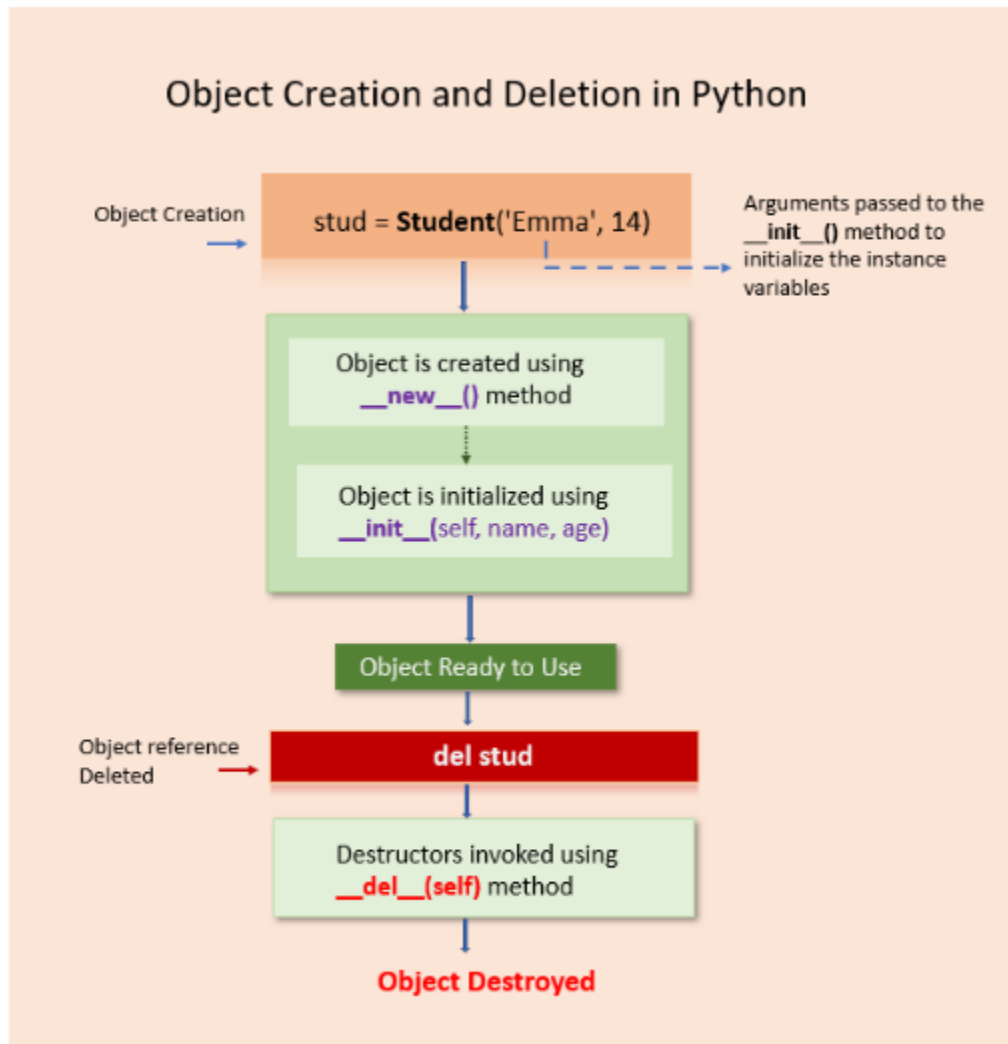
But it's not just memory that has to be freed when an object is destroyed. We **must release or close the other resources object were using**, such as open files, database connections, cleaning up the buffer or cache. **To perform all those cleanup tasks we use destructor** in Python.

The destructor is the reverse of the [constructor](#). The constructor is used to initialize objects, while the destructor is used to delete or destroy the object that releases the resource occupied by the object.

In Python, destructor is not called manually but completely automatic. **destructor gets called in the following two cases**

- When an object goes out of scope or
- The reference counter of the object reaches 0.

In Python, The special method `__del__()` is used to define a destructor. For example, when we execute `del object_name` destructor gets called automatically and the object gets garbage collected.



Python destructor to destroy an object

Create Destructor using the `__del__()` Method

The magic method `__del__()` is used as the destructor in Python. The `__del__()` method will be implicitly invoked when all references to the object have been deleted, i.e., is when an object is eligible for the garbage collector.

This method is automatically called by Python when the instance is about to be destroyed. It is also called a finalizer or (improperly) a destructor.

Syntax of destructor declaration

```
def __del__(self):
    # body of a destructor
```

Where,

- **def:** The keyword is used to define a method.
- **`__del__()` Method:** It is a reserved method. This method gets called as soon as all references to the object have been deleted

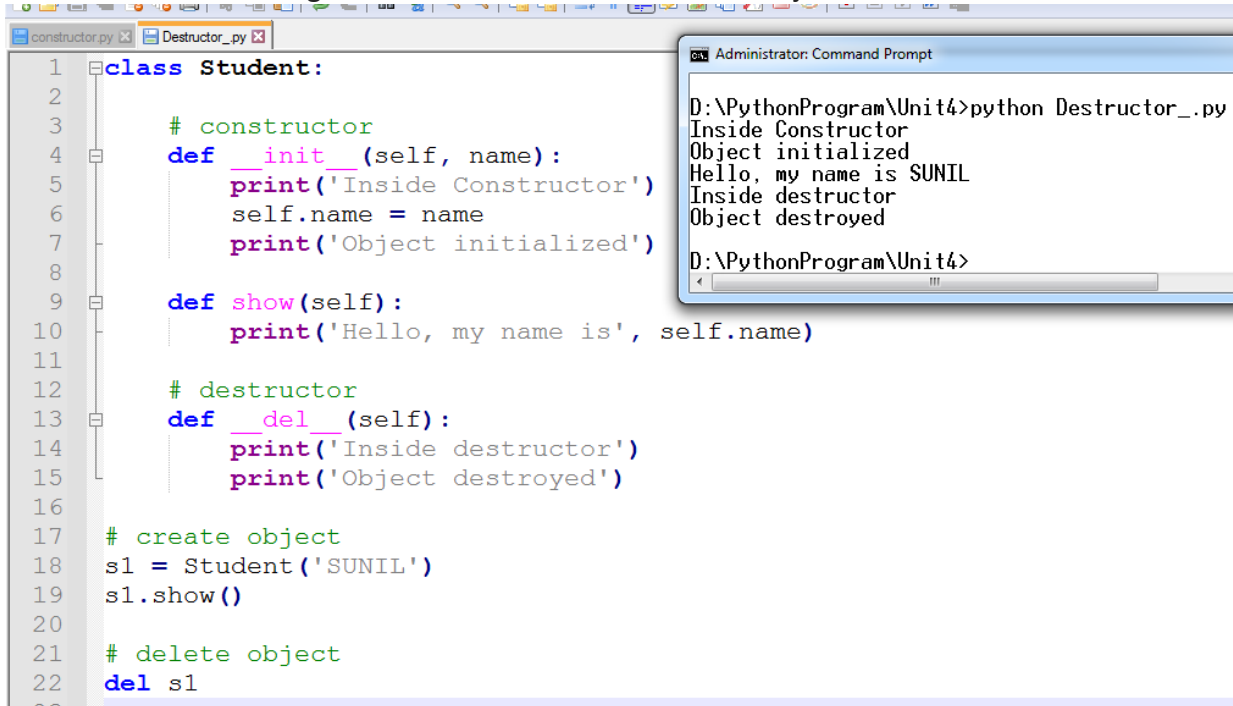
- **self**: The first argument **self** refers to the current object.

Note: The `__del__()` method arguments are optional. We can define a destructor with any number of arguments.

Example

Let's see how to create a destructor in Python with a simple example. In this example, we'll create a Class Student with a destructor. We'll see: –

- How to implement a destructor
- how destructor gets executed when we delete the object.
- How destructor gets executed when we delete the object.



```

1 class Student:
2
3     # constructor
4     def __init__(self, name):
5         print('Inside Constructor')
6         self.name = name
7         print('Object initialized')
8
9     def show(self):
10        print('Hello, my name is', self.name)
11
12    # destructor
13    def __del__(self):
14        print('Inside destructor')
15        print('Object destroyed')
16
17    # create object
18    s1 = Student('SUNIL')
19    s1.show()
20
21    # delete object
22    del s1
  
```

Administrator: Command Prompt

```

D:\PythonProgram\Unit4>python Destructor_.py
Inside Constructor
Object initialized
Hello, my name is SUNIL
Inside destructor
Object destroyed
D:\PythonProgram\Unit4>
  
```

Note:

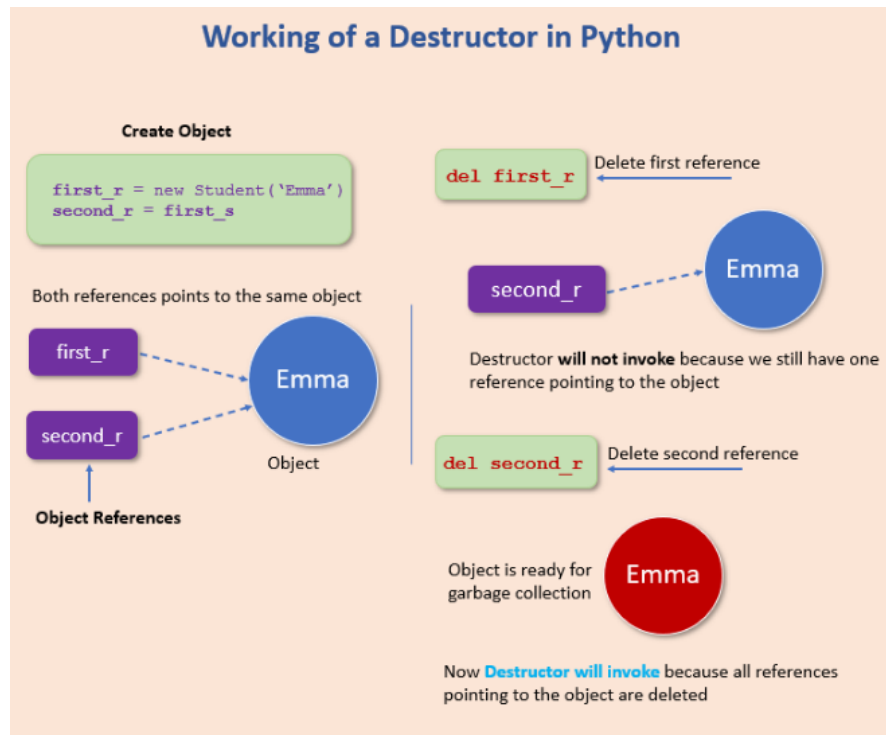
As you can see in the output, the `__del__()` method get called automatically is called when we deleted the object reference using `del s1`.

In the above code, we created one object. The `s1` is the reference variable that is pointing to the newly created object.

The destructor has called when the reference to the object is deleted or the reference count for the object becomes zero

Important Points to Remember about Destructor

- The `__del__` method is called for any object **when the reference count for that object becomes zero**.
- The reference count for that object becomes zero when the application ends, or we delete all references manually using the `del` keyword.
- The destructor will not invoke when **we delete object reference**. It will only **invoke when all references to the objects get deleted**.



Working of destructor

Example:

Let's understand the above points using the example.

- First create object of a student class using `s1 = student('Emma')`
- Next, create a new object reference `s2` by assigning `s1` to `s2` using `s2=s1`
- Now, both reference variables `s1` and `s2` point to the same object.
- Next, we deleted reference `s1`
- Next, we have added 5 seconds of sleep to the main thread to understand that destructors only invoke when all references to the objects get deleted.

2. Encapsulation

Encapsulation is a mechanism where the **data (variables)** and the **code (methods)** that act on the data will bind together.

For example, if we take a class, we write the **variables and methods inside the class**. Thus, class is binding them together. So class is an example for encapsulation.

For example, we can write a Student class with **'id' and 'name' as attributes** along with the **display()** method that displays this data. This Student class becomes an example for encapsulation.

The screenshot shows a Notepad++ window with the following Python code:

```

1 #example of encapsulation
2 class Student:
3     def __init__(self):
4         self.id = 10
5         self.name = 'Raju'
6     #display students details
7     def display(self):
8         print(self.id)
9         print(self.name)
10
11 Stul=Student()
12 Stul.display()
13

```

Overlaid on the right is a Windows Command Prompt window titled "Administrator: Command Prompt". It shows the execution of the script:

```

D:\PythonProgram\Unit4>python Enapuslation.py
10
Raju
D:\PythonProgram\Unit4>

```

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

```

class Employee:

    def __init__(self, name, salary):

        self.name = name
        self._project = project
        self.__salary = salary

```

Public Member (accessible within or outside of a class)

Protected Member (accessible within the class and it's sub-classes)

Private Member (accessible only within a class)

↑
Data Hiding using Encapsulation

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:

The screenshot shows a Notepad++ window with a Python script named 'Enapuslation.py'. The script defines a class 'Employee' with a constructor '__init__' and a public instance method 'show'. It creates an object 'emp' and calls the 'show' method. A Command Prompt window shows the output of running the script, displaying the name and salary of the employee.

```

1 # program to demonstrate public members
2 class Employee:
3     # constructor
4     def __init__(self, name, salary):
5         # public data members
6         self.name = name
7         self.salary = salary
8
9     # public instance methods
10    def show(self):
11        # accessing public data member
12        print("Name: ", self.name, 'Salary:', self.salary)
13
14    # creating object of a class
15    emp = Employee('Sunilkumbhar', 10000)
16
17    # accessing public data members
18    print("Name: ", emp.name, 'Salary:', emp.salary)
19
20    # calling public method of the class
21    emp.show()
22

```

```

D:\PythonProgram\Unit4>python Enapuslation.p
Name: Sunilkumbhar Salary: 10000
Name: Sunilkumbhar Salary: 10000
D:\PythonProgram\Unit4>

```

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:

The screenshot shows a Notepad++ window with a Python script named 'Enapuslation.py'. The script defines a class 'Employee' with a constructor '__init__' and a private instance method '__salary'. It creates an object 'emp' and attempts to access the private attribute 'emp.__salary'. A Command Prompt window shows the output of running the script, displaying a traceback error because the attribute is not accessible from outside the class.

```

1 #Private members are accessible only within the class
2 class Employee:
3     # constructor
4     def __init__(self, name, salary):
5         # public data member
6         self.name = name
7         # private member
8         self.__salary = salary
9
10    # creating object of a class
11    emp = Employee('sunil kumbhar', 50000)
12
13    # accessing private data members
14    print('Salary:', emp.__salary)
15

```

```

D:\PythonProgram\Unit4>python Enapuslation.py
Traceback (most recent call last):
  File "Enapuslation.py", line 14, in <module>
    print('Salary:', emp.__salary)
AttributeError: 'Employee' object has no attribute '__salary'
D:\PythonProgram\Unit4>

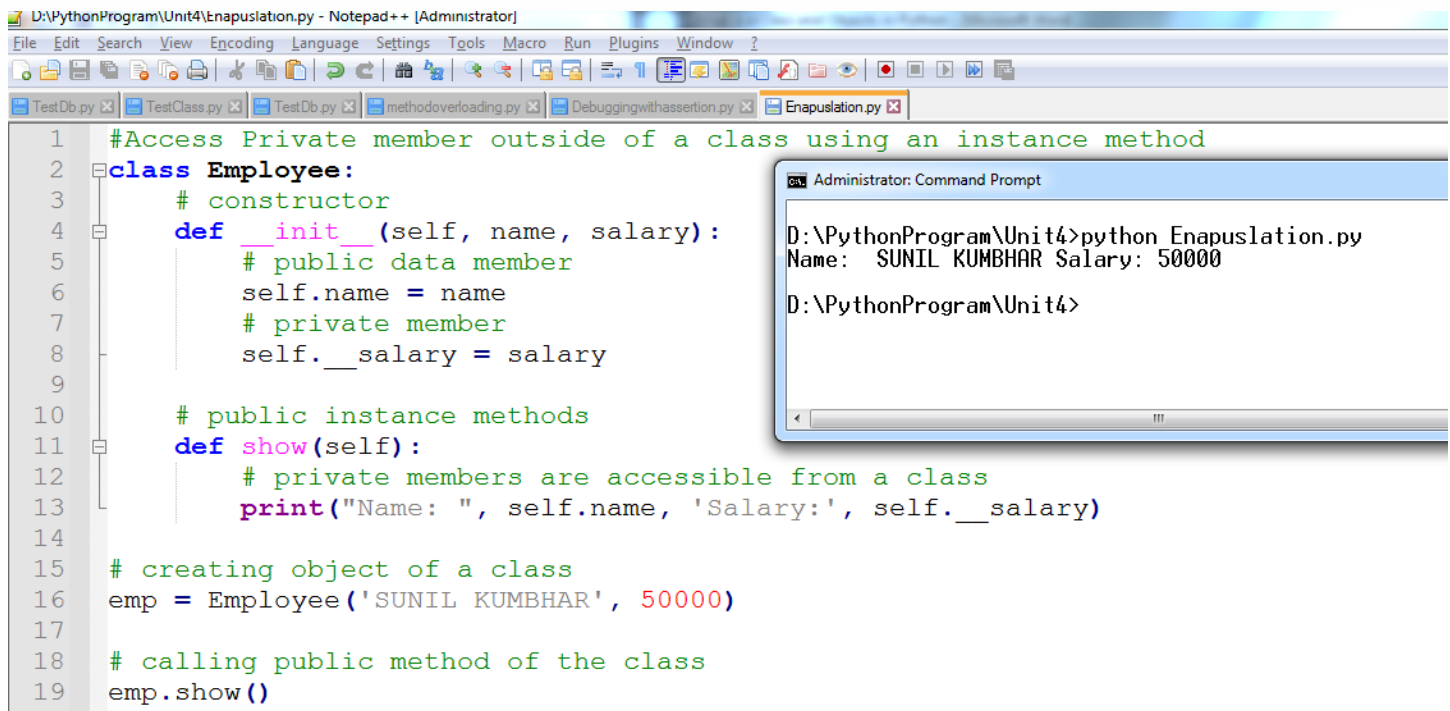
```

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

Public method to access private members

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



```

1  #Access Private member outside of a class using an instance method
2  class Employee:
3      # constructor
4      def __init__(self, name, salary):
5          # public data member
6          self.name = name
7          # private member
8          self.__salary = salary
9
10     # public instance methods
11     def show(self):
12         # private members are accessible from a class
13         print("Name: ", self.name, 'Salary:', self.__salary)
14
15     # creating object of a class
16     emp = Employee('SUNIL KUMBHAR', 50000)
17
18     # calling public method of the class
19     emp.show()
  
```

Administrator: Command Prompt

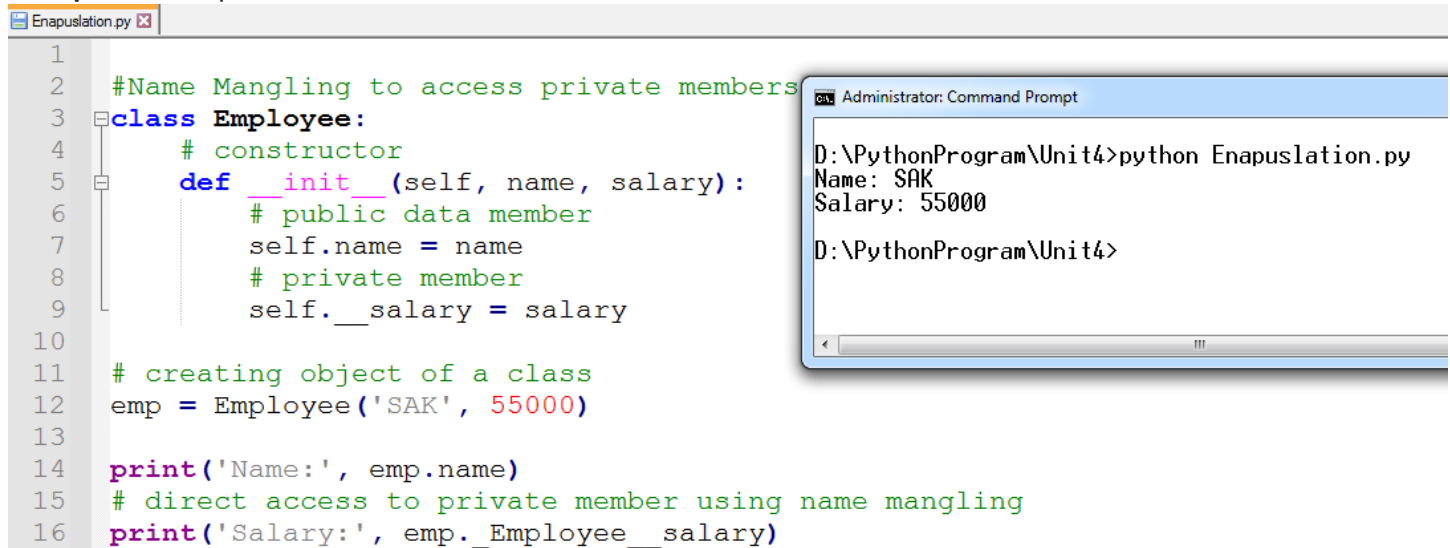
```

D:\PythonProgram\Unit4>python Enapuslation.py
Name:  SUNIL KUMBHAR Salary: 50000
D:\PythonProgram\Unit4>
  
```

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



```

1  #Name Mangling to access private members
2  class Employee:
3      # constructor
4      def __init__(self, name, salary):
5          # public data member
6          self.name = name
7          # private member
8          self.__salary = salary
9
10     # creating object of a class
11     emp = Employee('SAK', 55000)
12
13     print('Name:', emp.name)
14     # direct access to private member using name mangling
15     print('Salary:', emp._Employee__salary)
  
```

Administrator: Command Prompt

```

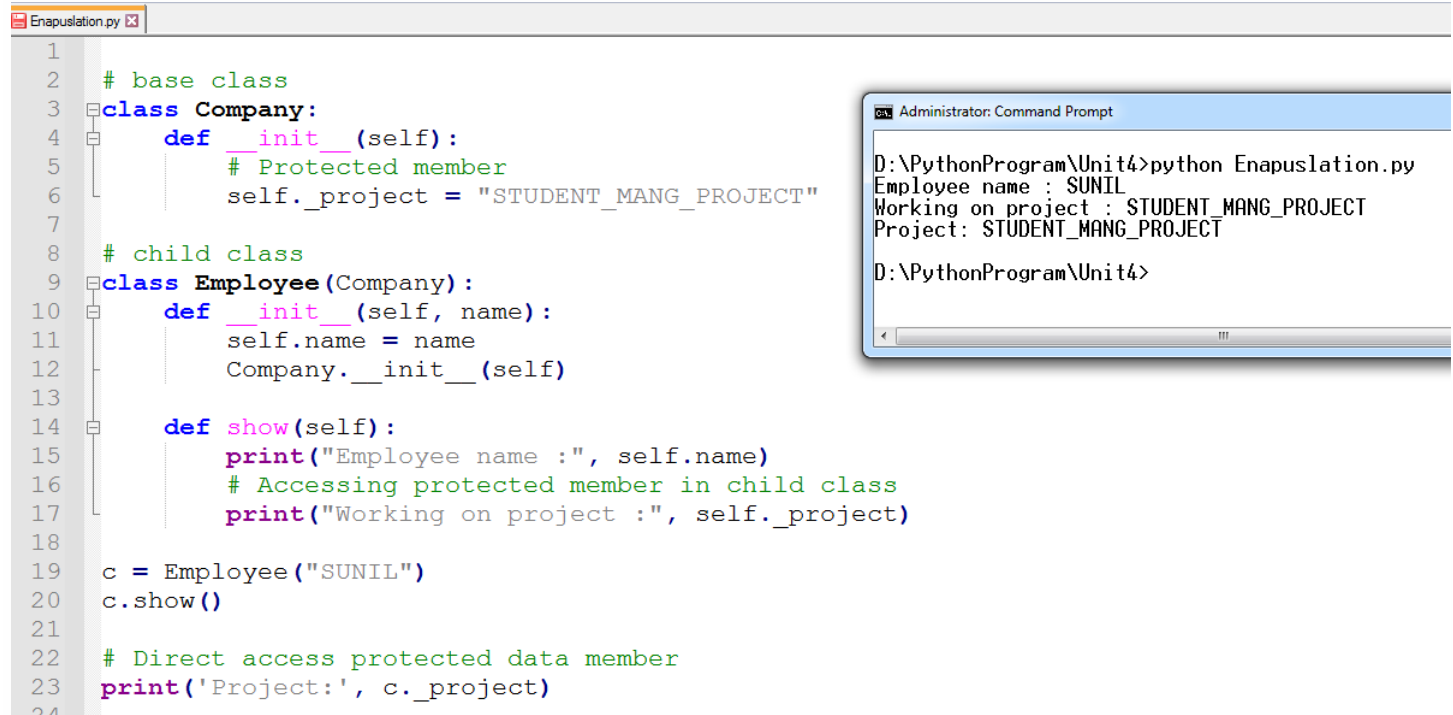
D:\PythonProgram\Unit4>python Enapuslation.py
Name: SAK
Salary: 55000
D:\PythonProgram\Unit4>
  
```

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: Protected member in inheritance.



The screenshot shows a Python script named `Enapuslation.py` and its execution output in a Command Prompt window.

```

1
2 # base class
3 class Company:
4     def __init__(self):
5         # Protected member
6         self._project = "STUDENT_MANG_PROJECT"
7
8 # child class
9 class Employee(Company):
10    def __init__(self, name):
11        self.name = name
12        Company.__init__(self)
13
14    def show(self):
15        print("Employee name :", self.name)
16        # Accessing protected member in child class
17        print("Working on project :", self._project)
18
19 c = Employee("SUNIL")
20 c.show()
21
22 # Direct access protected data member
23 print('Project:', c._project)
24

```

The Command Prompt window shows the following output:

```

D:\PythonProgram\Unit4>python Enapuslation.py
Employee name : SUNIL
Working on project : STUDENT_MANG_PROJECT
Project: STUDENT_MANG_PROJECT
D:\PythonProgram\Unit4>

```

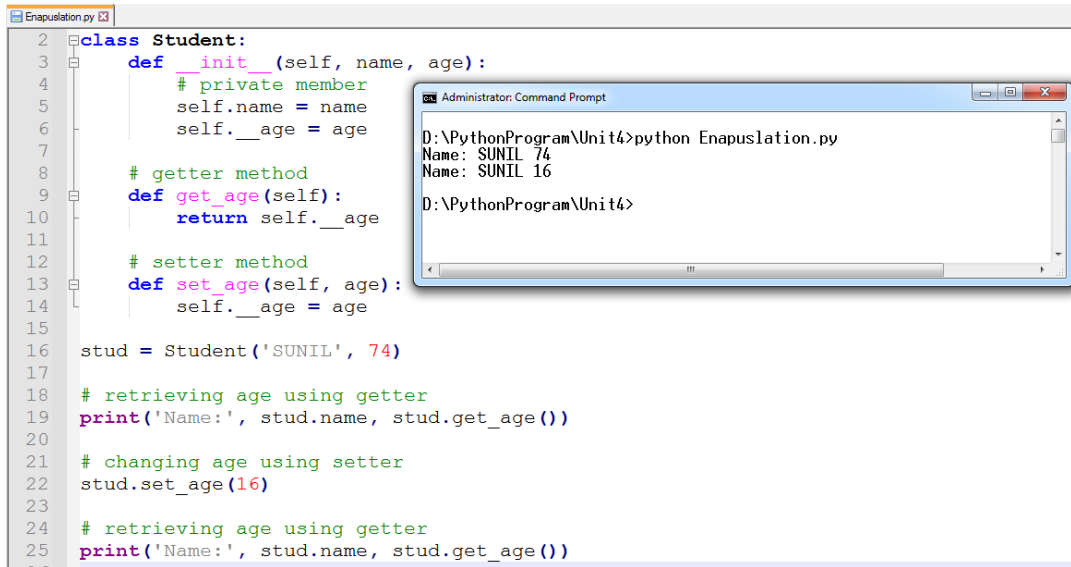
Getters and Setters in Python

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



The screenshot shows a Python script named 'Enapsulation.py' (note the typo) and its execution in a Windows Command Prompt. The script defines a 'Student' class with private attributes 'name' and 'age', and methods 'get_age' and 'set_age'. It creates a 'stud' object, prints its name and age, changes the age, and prints it again.

```

2 class Student:
3     def __init__(self, name, age):
4         # private member
5         self.name = name
6         self.__age = age
7
8     # getter method
9     def get_age(self):
10        return self.__age
11
12    # setter method
13    def set_age(self, age):
14        self.__age = age
15
16 stud = Student('SUNIL', 74)
17
18 # retrieving age using getter
19 print('Name:', stud.name, stud.get_age())
20
21 # changing age using setter
22 stud.set_age(16)
23
24 # retrieving age using getter
25 print('Name:', stud.name, stud.get_age())

```

Command Prompt Output:

```

D:\PythonProgram\Unit4>python Enapsulation.py
Name: SUNIL 74
Name: SUNIL 16
D:\PythonProgram\Unit4>

```

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

3. Abstraction

There may be a lot of data, a class contains and the user does not need the entire data. The user requires only some part of the available data. In this case, we **can hide the unnecessary data from the user** and expose only that **data that is of interest to the user**. This is called abstraction.

Abstraction in Python

In languages like Java, we have keywords like private, protected and public to implement various levels of abstraction. These keywords are called access specifiers.

In Python, such words are not available. **Everything written in the class will come under public.** That means everything written in the class is available outside the class to other people. Suppose, we do not want to make a variable available outside the class or to other members inside the class, we can write the variable with two double scores before it as: **__var**. This is like a private variable in Python. In the following example, **'y' is a private variable since it is written as: __y**.

Class Myclass:

#this is constructor.

```
def __init__(self):
    self.__y = 3 #this is private variable
```

Now, it is not possible to access the variable from within the class or out of the class as:

```
m = Myclass()
```

```
print(m.y) #error
```

The preceding print() statement displays error message as: **AttributeError: 'Myclass' object has no attribute 'y'**. Even though, we cannot access the private variable in this way, it is possible to access it in the format:

```
instancename._Classname__var.
```

That means we are using **Classname differently to access the private variable**. This is called **name mangling**. In name mangling, we have to use **one underscore before the classname and two underscores after the classname**. Like this, using the names differently to access the private variables is called name mangling.

4. Inheritance in Python

The **process of inheriting the properties of the parent class into a child class is called inheritance**. The existing class is called a base class or parent class and the new class is called a subclass or child class or derived class.

In this Python lesson, you will learn inheritance, method overloading, method overriding, types of inheritance, and MRO (Method Resolution Order).

In Object-oriented programming, inheritance is an important aspect. The main purpose of inheritance is the **reusability** of code because we can use the existing [class](#) to create a new class instead of creating it from scratch.

In inheritance, the child class acquires all the data members, properties, and functions from the parent class. Also, a child class can also provide its specific implementation to the methods of the parent class.

For example, In the real world, Car is a sub-class of a Vehicle class. We can create a Car by inheriting the properties of a Vehicle such as Wheels, Colors, Fuel tank, engine, and add extra properties in Car as required.

Syntax

```
class BaseClass:
```

```
    Body of base class
```

```
class DerivedClass(BaseClass):
```

```
    Body of derived class
```

Types Of Inheritance

In Python, based upon the number of child and parent classes involved, there are five types of inheritance. The type of inheritance are listed below:

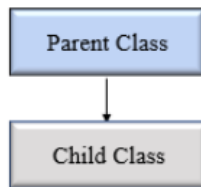
1. Single inheritance
2. Multiple Inheritance
3. Multilevel inheritance
4. Hierarchical Inheritance
5. Hybrid Inheritance

Now let's see each in detail with an example.

1. Single Inheritance

In single inheritance, a child class inherits from a single-parent class. Here is one child class and one parent class.

Python Single Inheritance



Python Single Inheritance

example

Let's create one parent class called **ClassOne** and one child class called **ClassTwo** to implement single inheritance.

The screenshot shows a Python script named `inheritance.py` and its execution in a command prompt. The script defines a base class `Vehicle` with a method `Vehicle_info` and a child class `Car` that inherits from `Vehicle` and has its own method `car_info`. It then creates an instance of `Car` and calls both methods.

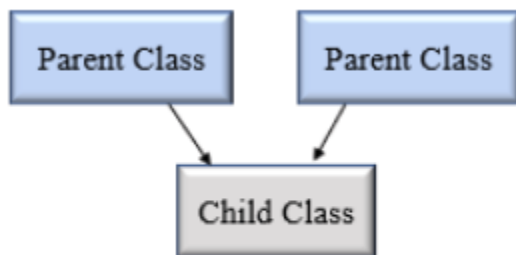
```
1 # Base class
2 class Vehicle:
3     def Vehicle_info(self):
4         print('Inside Vehicle class')
5
6 # Child class
7 class Car(Vehicle):
8     def car_info(self):
9         print('Inside Car class')
10
11 # Create object of Car
12 car = Car()
13
14 # access Vehicle's info using car object
15 car.Vehicle_info()
16 car.car_info()
```

The command prompt shows the execution of `python inheritance.py`, which outputs:

```
D:\PythonProgram\Unit4>python inheritance.py
Inside Vehicle class
Inside Car class
D:\PythonProgram\Unit4>
```

2. Multiple Inheritance

In multiple inheritances, one child class can inherit from multiple parent classes. So here is one child class and multiple parent classes.



Python Multiple Inheritance

Example

```
# Parent class 1
class Person:
    def person_info(self, name, age):
        print('Inside Person class')
        print('Name:', name, 'Age:', age)

# Parent class 2
class Company:
    def company_info(self, company_name, location):
        print('Inside Company class')
        print('Name:', company_name, 'location:', location)

# Child class
class Employee(Person, Company):
    def Employee_info(self, salary, skill):
        print('Inside Employee class')
        print('Salary:', salary, 'Skill:', skill)

# Create object of Employee
emp = Employee()

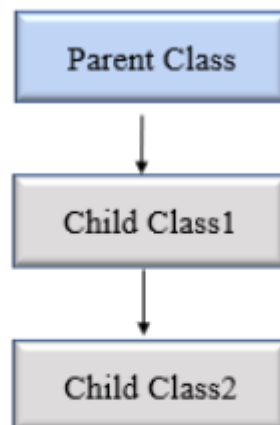
# access data
emp.person_info('sunil', 28)
emp.company_info('TCS', 'pune')
emp.Employee_info(40000, 'Data Science')
```

Administrator: Command Prompt

```
D:\PythonProgram\Unit4>python MultipleIn.py
Inside Person class
Name: sunil Age: 28
Inside Company class
Name: TCS location: pune
Inside Employee class
Salary: 40000 Skill: Data Science
D:\PythonProgram\Unit4>
```

3. Multilevel inheritance

In multilevel inheritance, a class inherits from a child class or derived class. Suppose three classes A, B, C. A is the superclass, B is the child class of A, C is the child class of B. In other words, we can say a **chain of classes** is called **multilevel inheritance**.



Python Multilevel
Inheritance

```
#multilevel inheritance
# Base class
class Vehicle:
    def Vehicle_info(self):
        print('Inside Vehicle class')

# Child class
class Car(Vehicle):
    def car_info(self):
        print('Inside Car class')

# Child class
class SportsCar(Car):
    def sports_car_info(self):
        print('Inside SportsCar class')

# Create object of SportsCar
s_car = SportsCar()

# access Vehicle's and Car info using SportsCar object
s_car.Vehicle_info()
s_car.car_info()
s_car.sports_car_info()
```

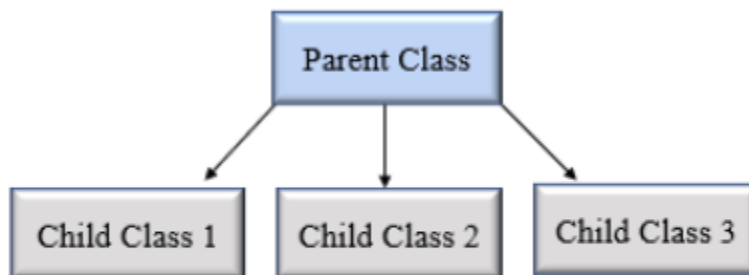
Administrator: Command Prompt

```
D:\PythonProgram\Unit4>python MultipleIn.py
Inside Vehicle class
Inside Car class
Inside SportsCar class

D:\PythonProgram\Unit4>
```

4. Hierarchical Inheritance

In Hierarchical inheritance, more than one child class is derived from a single parent class. In other words, we can say one parent class and multiple child classes.



Python hierarchical inheritance

Example

Let's create 'Vehicle' as a parent class and two child class 'Car' and 'Truck' as a parent class.

```
#Hierarchical inheritance

class Vehicle:
    def info(self):
        print("This is Vehicle")

class Car(Vehicle):
    def car_info(self, name):
        print("Car name is:", name)

class Truck(Vehicle):
    def truck_info(self, name):
        print("Truck name is:", name)

obj1 = Car()
obj1.info()
obj1.car_info('BMW')

obj2 = Truck()
obj2.info()
obj2.truck_info('Ford')
```

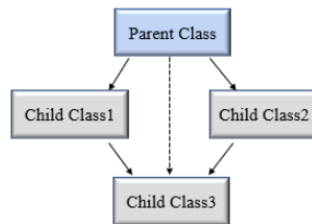
Administrator: Command Prompt

```
D:\PythonProgram\Unit4>python Inher.py
This is Vehicle
Car name is: BMW
This is Vehicle
Truck name is: Ford

D:\PythonProgram\Unit4>
```

5. Hybrid Inheritance

When inheritance consists of multiple types or a combination of different inheritance is called hybrid inheritance.



Python hybrid inheritance

Example

```
#HYBRID Inheritance
class Vehicle:
    def vehicle_info(self):
        print("Inside Vehicle class")

class Car(Vehicle):
    def car_info(self):
        print("Inside Car class")

class Truck(Vehicle):
    def truck_info(self):
        print("Inside Truck class")

# Sports Car can inherits properties of Vehicle
class SportsCar(Car, Vehicle):
    def sports_car_info(self):
        print("Inside SportsCar class")

# create object
s_car = SportsCar()

s_car.vehicle_info()
s_car.car_info()
s_car.sports_car_info()
```

```
Administrator: Command Prompt
D:\PythonProgram\Unit4>python Inher.py
Inside Vehicle class
Inside Car class
Inside SportsCar class
D:\PythonProgram\Unit4>
```

Python super() function

When a class inherits all properties and behavior from the parent class is called inheritance. In such a case, the inherited class is a subclass and the latter class is the parent class.

In child class, we can refer to parent class by using the **super()** function. The super function returns a temporary object of the parent class that allows us to call a parent class method inside a child class method.

Benefits of using the super() function.

1. We are not required to remember or specify the parent class name to access its methods.
2. We can use the **super()** function in both **single** and **multiple inheritances**.
3. The **super()** function support code **reusability** as there is no need to write the entire function

Example

```
#Python super() function
class Company:
    def company_name(self):
        return 'Google'

class Employee(Company):
    def info(self):
        # Calling the superclass method using super() function
        c_name = super().company_name()
        print("Sunil works at", c_name)

# Creating object of child class
emp = Employee()
emp.info()
```

```
Administrator: Command Prompt
D:\PythonProgram\Unit4>python Inher.py
Sunil works at Google
D:\PythonProgram\Unit4>
```

In the above example, we create a parent class **Company** and child class **Employee**. In **Employee** class, we call the parent class method by using a **super()** function.

issubclass()

In Python, we can verify whether a particular class is a subclass of another class. For this purpose, we can use Python built-in function `issubclass()`. This function returns **True** if the given class is the subclass of the specified class. Otherwise, it returns **False**.

Syntax

issubclass(class, classinfo)

Where,

- **class**: class to be checked.
- **classinfo**: a **class**, type, or a **tuple** of classes or data types.

Example

```
class Company:
```

```
    def fun1(self):  
        print("Inside parent class")
```

```
class Employee(Company):
```

```
    def fun2(self):  
        print("Inside child class.")
```

```
class Player:
```

```
    def fun3(self):  
        print("Inside Player class.")
```

```
# Result True
```

```
print(issubclass(Employee, Company))
```

```
# Result False
```

```
print(issubclass(Employee, list))
```

```
# Result False
```

```
print(issubclass(Player, Company))
```

```
# Result True
```

```
print(issubclass(Employee, (list, Company)))
```

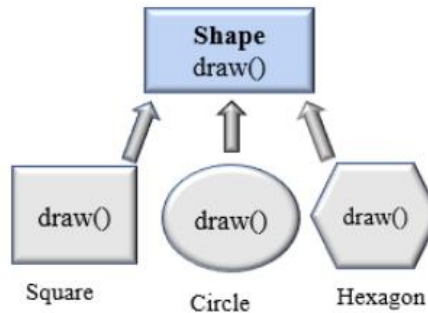
```
# Result True
```

```
print(issubclass(Company, (list, Company)))
```

Method Overriding

In inheritance, all members available in the parent class are by default available in the child class. If the child class does not satisfy with parent class implementation, then the child class is allowed to redefine that method by extending additional functions in the child class. This concept is called **method overriding**.

When a child class method has the same name, same parameters, and same return type as a method in its superclass, then the method in the child is said to **override** the method in the parent class.



Python method overriding

Example

```
class Vehicle:
    def max_speed(self):
        print("max speed is 100 Km/Hour")

class Car(Vehicle):
    # overridden the implementation of Vehicle class
    def max_speed(self):
        print("max speed is 200 Km/Hour")

# Creating object of Car class
car = Car()
car.max_speed()
```

Method Resolution Order in Python

In Python, Method Resolution Order(MRO) is the order by which **Python looks for a method or attribute**. First, the method or attribute is searched within a class, and then it follows the order we specified while inheriting.

This order is also called the Linearization of a class, and a set of rules is called MRO (Method Resolution Order). The **MRO plays an essential role in multiple inheritances as a single method may found in multiple parent classes**.

In multiple inheritance, the following search order is followed.

1. First, it searches in the current parent class if not available, then searches in the parents class specified while inheriting (that is left to right.)
2. We can get the MRO of a class. For this purpose, we can use either the `mro` attribute or the `mro()` method.

Example

```
class A:
    def process(self):
        print(" In class A")
```

```
class B(A):
    def process(self):
        print(" In class B")

class C(B, A):
    def process(self):
        print(" In class C")
# Creating object of C class
C1 = C()
C1.process()
print(C.mro())
```

Output:

```
# In class C
# [<class '__main__.C'>, <class '__main__.B'>, <class '__main__.A'>, <class 'object'>]
```

In the above example, we create three classes named **A**, **B** and **C**. Class **B** is inherited from **A**, class **C** inherits from **B** and **A**. When we create an object of the **C** class and calling the **process()** method, Python looks for the **process()** method in the current class in the **C** class itself.

Then search for parent classes, namely **B** and **A**, because **C** class inherit from **B** and **A**. that is, **C(B, A)** and always search in **left to right manner**

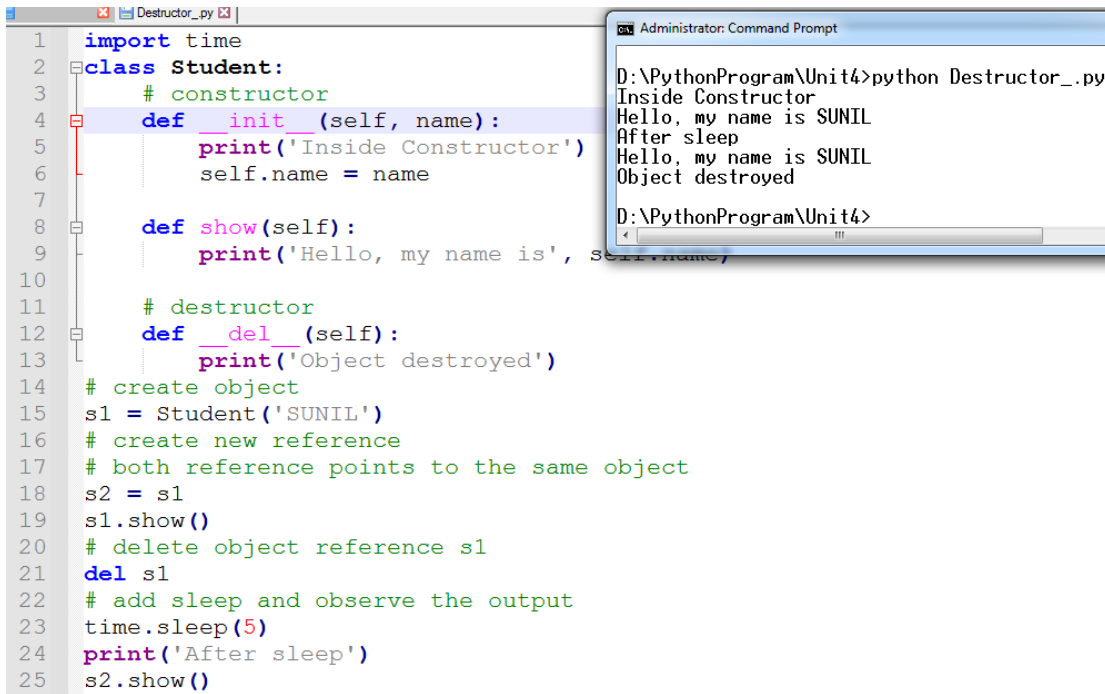
5. Polymorphism

The word 'Polymorphism' **came from two Greek words 'poly' meaning 'many' and 'morphos' meaning 'forms'**. Thus, polymorphism represents the **ability to assume several different forms**. In programming, if an object or method is exhibiting **different behavior in different contexts**, it is called polymorphic nature. Polymorphism provides flexibility in writing programs in such a way that the programmer uses same method call to perform different operations depending on the requirement.

#a function that exhibits polymorphism

```
def add(a, b):
    print(a+b)
#call add() and pass two integers
add(5, 10) #displays 15
#call add() and pass two strings
add("Core", "Python") #displays CorePython
```

When a function can perform different tasks, we can say that it is exhibiting polymorphism.



```

1 import time
2 class Student:
3     # constructor
4     def __init__(self, name):
5         print('Inside Constructor')
6         self.name = name
7
8     def show(self):
9         print('Hello, my name is', self.name)
10
11     # destructor
12     def __del__(self):
13         print('Object destroyed')
14 # create object
15 s1 = Student('SUNIL')
16 # create new reference
17 # both reference points to the same object
18 s2 = s1
19 s1.show()
20 # delete object reference s1
21 del s1
22 # add sleep and observe the output
23 time.sleep(5)
24 print('After sleep')
25 s2.show()

```

Administrator: Command Prompt

```

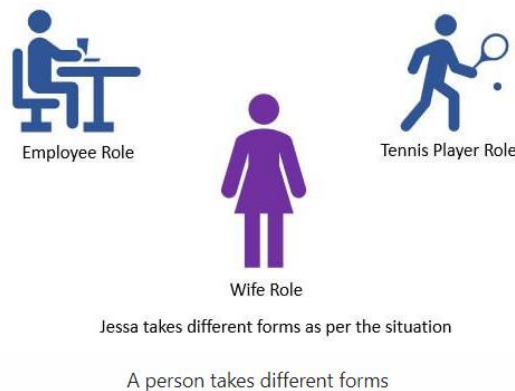
D:\PythonProgram\Unit4>python Destructor_.py
Inside Constructor
Hello, my name is SUNIL
After sleep
Hello, my name is SUNIL
Object destroyed
D:\PythonProgram\Unit4>

```

Polymorphism in Python

We can implement polymorphism using **function overloading**, **method overriding**, and **operator overloading**. Polymorphism in Python is the ability of an **object** to take many forms. In simple words, polymorphism allows us to perform the same action in many different ways.

For example, Jessa acts as an employee when she is at the office. However, when she is at home, she acts like a wife. Also, she represents herself differently in different places. Therefore, the same person takes different forms as per the situation.



In polymorphism, a method can **process objects differently depending on the class type or data type**. Let's see simple examples to understand it better.

1. Method Overloading

The process of calling the same method with different parameters is known as method overloading. **Python does not support method overloading.** Python considers **only the latest defined method** even if you overload the method. Python will raise a `TypeError` if you overload the method.

```
def addition(a, b):
    c = a + b
    print(c)
def addition(a, b, c):
    d = a + b + c
    print(d)
# the below line shows an error
# addition(4, 5)
# This line will call the second product method
addition(3, 7, 5)
```

To overcome the above problem, we can use different ways to achieve the method overloading. In Python, to **overload the class method**, we need to write the **method's logic so that different code** executes inside the function depending on the parameter passes.

For example, the built-in function [range\(\)](#) takes three parameters and produce different result depending upon the number of parameters passed to it.

```
for i in range(5): print(i, end=' ')
print()
for i in range(5, 10): print(i, end=' ')
print()
for i in range(2, 12, 2): print(i, end=' ')
```

Let's assume we have an [area\(\)](#) [method](#) to calculate the **area of a square and rectangle**. The method will calculate the area depending upon the number of parameters passed to it.

- If one parameter is passed, then the area of a square is calculated
- If two parameters are passed, then the area of a rectangle is calculated.

Example: User-defined polymorphic method

```
class Shape:
    # function with two default parameters
    def area(self, a, b=0):
        if b > 0:
            print('Area of Rectangle is:', a * b)
        else:
            print('Area of Square is:', a ** 2)
```

```
square = Shape()
square.area(5)
```

```
rectangle = Shape()
rectangle.area(5, 3)
```

Polymorphism in Built-in function len()

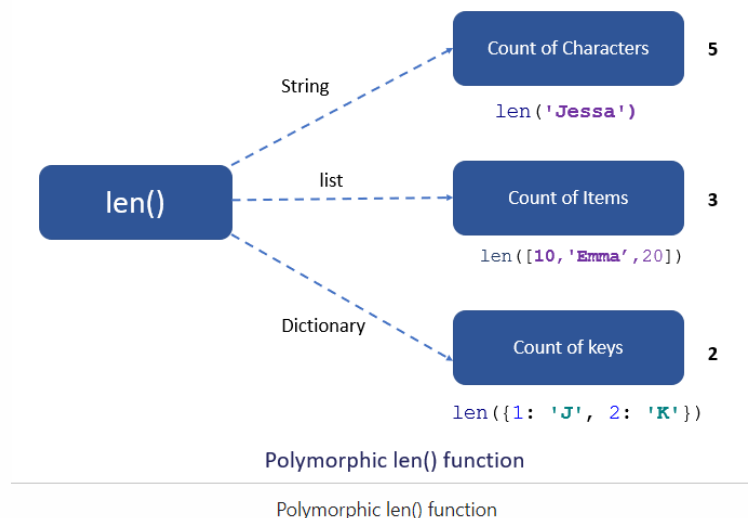
The built-in function `len()` calculates the length of an object depending upon its type. If an **object is a string**, it returns the **count of characters**, and If an object is a **list**, it returns the **count of items in a list**.

The `len()` method treats an object as per its class type.

Example:

```
students = ['Emma', 'Jessa', 'Kelly']
school = 'ABC School'
```

```
# calculate count
print(len(students))
print(len(school))
```



2. Operator Overloading in Python

Operator overloading means changing the **default behavior of an operator** depending on the operands (values) that we use. In other words, **we can use the same operator for multiple purposes**.

For example, the **+** operator will perform an **arithmetic addition operation** when used with **numbers**. Likewise, it will perform **concatenation** when used with **strings**.

The operator **+** is used to carry out different operations for distinct data types. This is one of the most simple occurrences of polymorphism in Python.

```
# add 2 numbers
print(100 + 200)
# concatenate two strings
print('Jess' + 'Roy')
# merger two list
print([10, 20, 30] + ['jessa', 'emma', 'kelly'])
```

Overloading + operator for custom objects

Suppose we have two objects, and we want to add these two **objects with a binary + operator**. However, it will throw an error if we perform addition because the compiler doesn't add two objects. See the following example for more details.

```
class Book:
    def __init__(self, pages):
        self.pages = pages
```

```
# creating two objects
```

```
b1 = Book(400)
```

```
b2 = Book(300)
```

```
# add two objects
```

```
print(b1 + b2)
```

output:

TypeError: unsupported operand type(s) for +: 'Book' and 'Book'

We can **overload + operator to work with custom objects** also. Python provides **some special or magic function that is automatically invoked when associated with that particular operator**.

For example, when we use the + operator, the magic **method __add__()** is automatically invoked. Internally + operator is implemented by using **__add__()** method. We have to override this method in our class if you want to add two custom objects.

```
class Book:
    def __init__(self, pages):
        self.pages = pages

    # Overloading + operator with magic method
    def __add__(self, other):
        return self.pages + other.pages
```

```
b1 = Book(400)
```

```
b2 = Book(300)
```

```
print("Total number of pages: ", b1 + b2)
```

Overloading the * Operator

The *** operator is used to perform the multiplication**. Let's see how to overload it to calculate the salary of an employee for a specific period. Internally * operator is implemented by using the **__mul__()** method.

```
class Employee:
    def __init__(self, name, salary):
        self.name = name
```

```

self.salary = salary

def __mul__(self, timesheet):
    print('Worked for', timesheet.days, 'days')
    # calculate salary
    return self.salary * timesheet.days

class TimeSheet:
    def __init__(self, name, days):
        self.name = name
        self.days = days

emp = Employee("Jessa", 800)
timesheet = TimeSheet("Jessa", 50)
print("salary is: ", emp * timesheet)

```

Operator Name	Symbol	Magic method
Addition	+	<code>__add__(self, other)</code>
Subtraction	-	<code>__sub__(self, other)</code>
Multiplication	*	<code>__mul__(self, other)</code>
Division	/	<code>__div__(self, other)</code>
Floor Division	//	<code>__floordiv__(self, other)</code>
Modulus	%	<code>__mod__(self, other)</code>
Power	**	<code>__pow__(self, other)</code>
Increment	+=	<code>__iadd__(self, other)</code>
Decrement	-=	<code>__isub__(self, other)</code>
Product	*=	<code>__imul__(self, other)</code>
Division	/+	<code>__idiv__(self, other)</code>
Modulus	%=	<code>__imod__(self, other)</code>
Power	**=	<code>__ipow__(self, other)</code>
Less than	<	<code>__lt__(self, other)</code>
Greater than	>	<code>__gt__(self, other)</code>
Less than or equal to	<=	<code>__le__(self, other)</code>
Greater than or equal to	>=	<code>__ge__(self, other)</code>
Equal to	==	<code>__eq__(self, other)</code>
Not equal	!=	<code>__ne__(self, other)</code>

3. Polymorphism with Inheritance

Polymorphism is mainly used with **inheritance**. In **inheritance**, child class inherits the **attributes and methods** of a parent class. The existing class is called a base class or parent class, and the new class is called a subclass or child class or derived class.

Using **method overriding** polymorphism allows us to define methods in the child [class](#) that have the **same name as the methods in the parent class**. This **process of re-implementing the inherited method in the child class** is known as Method Overriding.

Advantage of method overriding

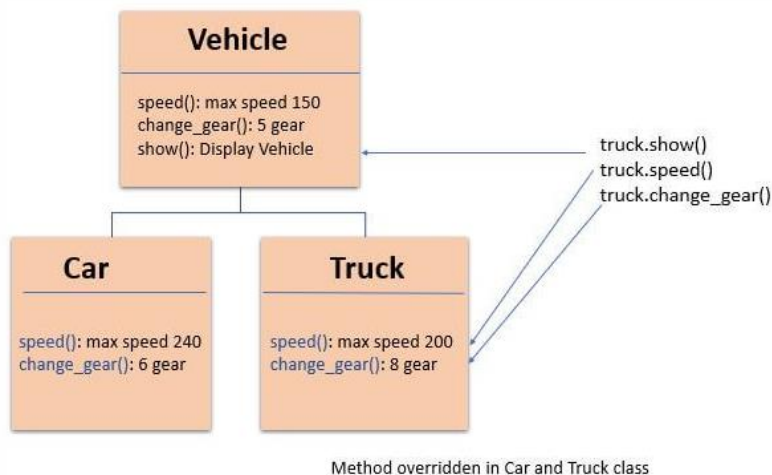
- It is effective when we want to extend the functionality by altering the inherited method. Or the **method inherited from the parent class** doesn't fulfill the need of a child class, so we need to re-implement the same method in the child class in a different way.
- Method overriding is useful when a parent class has multiple child classes, and one of that child classes wants to redefine the method. The other child classes can use the parent class method. Due to this, we don't need to modification the parent class code

In polymorphism, **Python first checks the object's class type and executes the appropriate method** when we call the method. For example, If you create the Car object, then Python calls the **speed()** method from a Car class.

Let's see how it works with the help of an example.

Example: Method Overriding

In this example, we have a vehicle class as a parent and a 'Car' and 'Truck' as its sub-class. But each vehicle can have a different seating capacity, speed, etc., so we can have the same [instance method](#) name in each class but with a different implementation. Using this code can be extended and easily maintained over time.



Polymorphism with Inheritance

class Vehicle:

```
def __init__(self, name, color, price):
    self.name = name
    self.color = color
    self.price = price

def show(self):
    print('Details:', self.name, self.color, self.price)

def max_speed(self):
    print('Vehicle max speed is 150')
```

```
def change_gear(self):
    print('Vehicle change 6 gear')

# inherit from vehicle class
class Car(Vehicle):
    def max_speed(self):
        print('Car max speed is 240')

    def change_gear(self):
        print('Car change 7 gear')

# Car Object
car = Car('Car x1', 'Red', 20000)
car.show()
# calls methods from Car class
car.max_speed()
car.change_gear()

# Vehicle Object
vehicle = Vehicle('Truck x1', 'white', 75000)
vehicle.show()
# calls method from a Vehicle class
vehicle.max_speed()
vehicle.change_gear():
```

python interpreter recognizes that the `max_speed()` and `change_gear()` methods are overridden for the car object. So, it uses the one defined in the child class (Car)

4. Polymorphism In Class methods

Polymorphism with **class methods** is useful when we **group different objects having the same method**. we can add them to **a list or a tuple**, and we don't need to check the object type before calling their methods. Instead, **Python will check object type at runtime and call the correct method**. Thus, we can call the methods without being concerned about which class type each object is. We assume that these methods exist in each class.

Python allows different classes to have methods with the same name.

- Let's design a different class in the same way by adding the same methods in two or more classes.
- Next, create an object of each class
- Next, add all objects in a [tuple](#).
- In the end, iterate the tuple using a [for loop](#) and call methods of a object without checking its class.

Example

In the below example, `fuel_type()` and `max_speed()` are the instance methods created in both classes.

```
class Ferrari:
    def fuel_type(self):
        print("Petrol")
```

```
def max_speed(self):
    print("Max speed 350")

class BMW:
    def fuel_type(self):
        print("Diesel")

    def max_speed(self):
        print("Max speed is 240")

ferrari = Ferrari()
bmw = BMW()

# iterate objects of same type
for car in (ferrari, bmw):
    # call methods without checking class of object
    car.fuel_type()
    car.max_speed()
```

output

```
Petrol
Max speed 350
Diesel
Max speed is 240
```

5. Polymorphism with Function and Objects

We can create **polymorphism with a function** that can **take any [object](#) as a parameter** and execute **its method without checking its class type**. Using this, we can call **object actions using the same function instead of repeating method calls**.

```
class Ferrari:
    def fuel_type(self):
        print("Petrol")

    def max_speed(self):
        print("Max speed 350")

class BMW:
    def fuel_type(self):
        print("Diesel")

    def max_speed(self):
        print("Max speed is 240")
```



```
# normal function
def car_details(obj):
    obj.fuel_type()
    obj.max_speed()

ferrari = Ferrari()
bmw = BMW()

car_details(ferrari)
car_details(bmw)
```