RV College of Engineering®

Autonomous Institution Affiliated to Visvesvaraya Technological University, Belagavi

Approved by AICTE New Delhi

Introduction to Python Programming

UNIT 5

Prof. Narasimha Swamy S
Department of AIML
RV College of Engineering
Bengaluru-59

Go, Change the World

Outline

→ **Software Objects**

- Defining a Class
- Defining Method
- Instantiating an Object
- Invoking a Methods
- Using Constructor
- Using Class Attributes and Static Methods
- Understanding Object Encapsulation

→ **Object-Oriented Programming**

- Using Inheritance to Create New Classes
- Creating a Base Class
- Inheriting from a Base Class
- Extending a Derived Class
- Using the Derived Class
- Extending a Class through Inheritance
- Understanding Polymorphism

Python Classes

- A class is considered as a blueprint of objects
- Think of the class as a sketch (blueprint) of a house
- It contains all the details about the floors, doors, windows, etc.
- Based on these descriptions we build the house
  - House is the object
- Since many houses can be made from the same description, same way many objects can be created from a class

Define Python Class

- We use the **class** keyword to create a class in Python
- For example

```
class ClassName:
    # class definition
```

- Here, we have created a class named **ClassName**
- Let's see an example,

```
class Bike:
        name = ""
        gear = 0
```

Here,

- **Bike** - the name of the class
- **name/gear** - variables inside the class with default values "" and 0 respectively

→ Note: The variables inside a class are called attributes

Python Objects and Instantiation of the Objects

- An object is called an instance of a class
- For example, suppose Bike is a class then we can create objects like bike1, bike2, etc. from the class
- Here's the syntax for object instantiation

```
objectName = ClassName()
```

- Let's see an example,

```
# create class
class Bike:
    name = ""
    gear = 0
# Instate the  objects of class
bike1 = Bike()
```

- Here, bike1 is the object of the class. Now, we can use this object to access the class attributes

Create Multiple Objects of Python Class

- We can also create multiple objects from a single class
- For example

```
# define a class
class Employee:
        employee_id = 0 # define an attribute
# create two objects of the Employee class
employee1 = Employee()
employee2 = Employee()
# access attributes using employee1
employee1.employeeID = 1001
print(f"Employee ID: {employee1. employee_id }")
# access attributes using employee2
employee2.employeeID = 1002
print(f"Employee ID: {employee2. employee_id }")
```

- In the above example, we have created two objects employee1 and employee2 of the Employee class

Access Class Attributes Using Objects

- The '.' Notation is used to access the attributes of a class
- For Example

```
# define a class
class Bike:
    name = ""
    gear = 0

# create object of class
bike1 = Bike()

# access attributes and assign new values
bike1.gear = 5
bike1.name = “Royal Enfield"
print(f"Name: {bike1.name}, Gears: {bike1.gear} ")
```

Python Methods

- A Python Function defined inside a class is called a method
- Let's see an Example

```
# create a class
class Room:
    length = 0.0
    breadth = 0.0

  # method to calculate area (Function Defined  Inside the Class)
    def calculate_area(self):
        print("Area of Room =", self.length * self.breadth)

# create object of Room class
study_room = Room()

# assign values to all the attributes
study_room.length = 42.5
study_room.breadth = 30.8

# access method inside class
study_room.calculate_area()
```

- Self parameter is the reference to the current instance of the class and used to access the variables of the class

Invoking a Methods

- The ‘.’ Notation is used to Invoke the method
- In the pervious example, we have created a class named Room with Attributes: length and breadth and Method:

  calculate_area() and object named study_room
- Now we have used the object to call the method inside the class

  study_room.calculate_area()
- We have used the ‘.’ notation to call the method
- Finally, the statement inside the method will execute

Constructor in Python

- A constructor is a special method in a class used to create and initialize an object of a class
- The constructor is invoked automatically when an object of a class is created
- Syntax of Python Constructor

```
def __init__(self):        # initializations
```

- The __init__ is one of the reserved functions in Python
- Rules of Python Constructor
  - It starts with the def keyword, like all other functions in Python.
  - It is followed by the word init, which is prefixed and suffixed with double underscores with a pair of brackets, i.e., __init__()
  - It takes an argument called self, assigning values to the variables

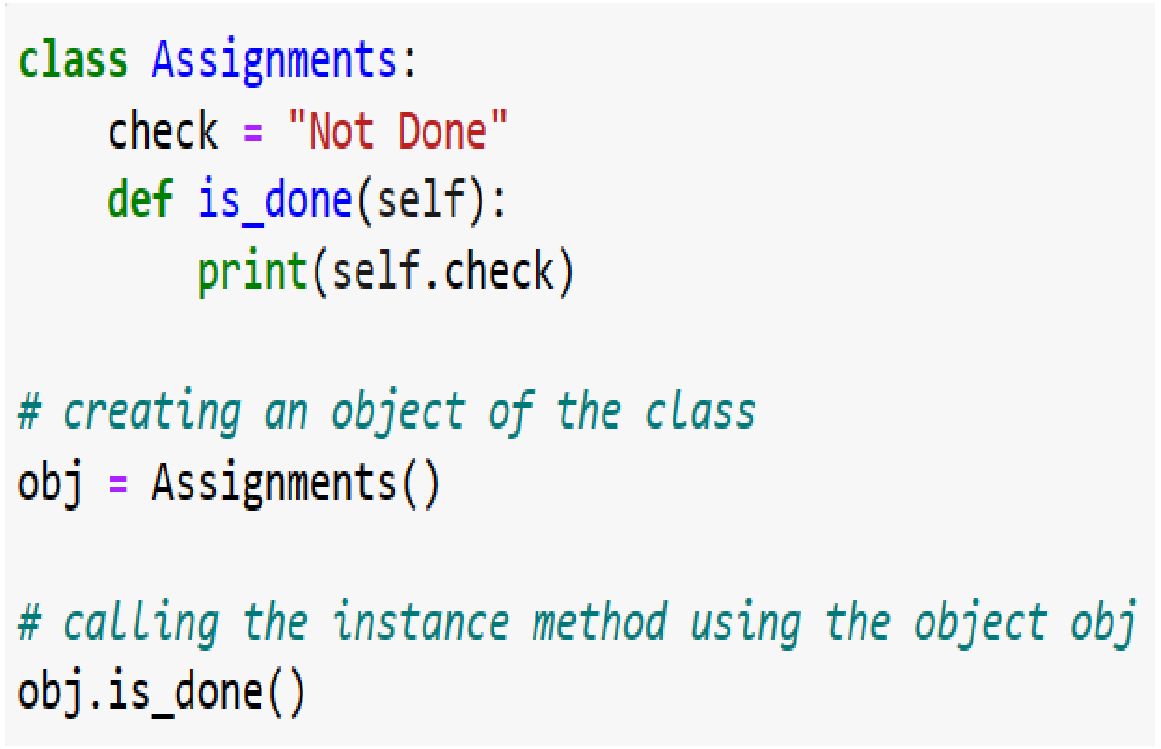
Invoking Default Constructor in Python

- When you do not write the constructor in the class created, Python itself creates a constructor during the compilation of the program
- It generates an empty constructor that has no code in it. Let's see an example:

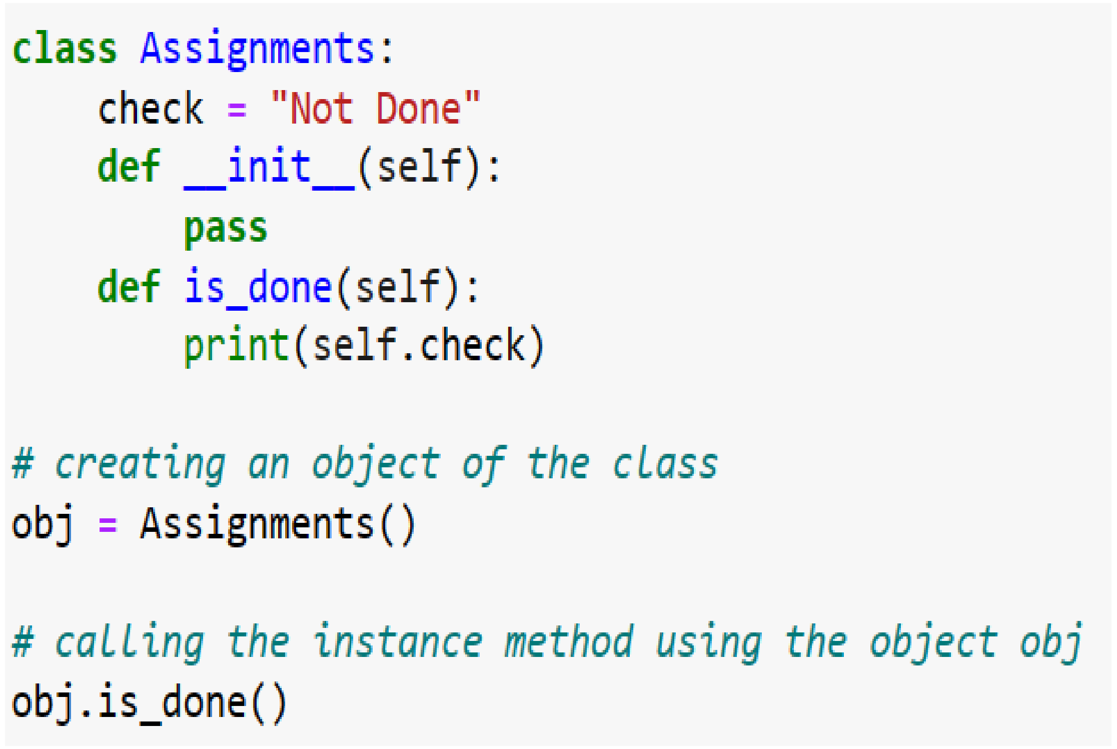
```
class Assignments:
        check= "not done"
        # a method
        def is_done(self):
                print(self.check)

# creating an object of the class
obj = Assignments()
# calling the instance method using the object obj
obj.is_done()
```

Invoking Default Constructor in Python

- When you do not write the constructor in the class created, Python itself creates a constructor during the compilation of the program
- It generates an empty constructor that has no code in it
- Let's see an example, below, in both the output is" Not Done"



```python
class Assignments:
    check = "Not Done"
    def is_done(self):
        print(self.check)

# creating an object of the class
obj = Assignments()

# calling the instance method using the object obj
obj.is_done()
```



```python
class Assignments:
    check = "Not Done"
    def __init__(self):
        pass
    def is_done(self):
        print(self.check)

# creating an object of the class
obj = Assignments()

# calling the instance method using the object obj
obj.is_done()
```

Invoking Method by Parameterized Constructor in Python

- When the constructor accepts arguments along with self, it is known as parameterized constructor
- These arguments can be used inside the class to assign the values to the data members
- Let's see an example:

```
class Family:
    # Constructor - parameterized
    members=‘’
    def __init__(self, count):
        print("This is parametrized constructor")
        self.members = count
    def show(self):
        print("No. of members is", self.members)

object = Family(10)
object.show()
```

Explanation

- An object of the class Family is created. It has a variable known as members.
- When the object is created, a parameter (here it is 10) is passed as arguments.
- This parameter (10 as in the given example) is taken up by the constructor as the object is created.
- The number 10 is assigned to the variable count, which is further assigned to self.members.
- The self.members can be used within the class to print the data

Methods in Python

- There are two types of the Method in Python
- Class Method
  - The purpose of the class methods is to set or get the details (status) of the class. That is why they are known as class methods
  - They can’t access or modify specific instance data. They are bound to the class instead of their objects
  - Two important things about class methods:
    - In order to define a class method, you have to specify that it is a class method with the help of the @classmethod decorator
    - Class methods take one default parameter- cls

Methods in Python

```
class My_class:
  @classmethod
  def class_method(cls):
          return "This is a class method."
```

- We'll create the instance of this My_class as well and try calling this class_method():

```
obj = My_class()
obj.class_method()
```

- We can access the class methods with the help of a class instance/object
- we can also access the class methods directly without creating an instance or object of the class.

```
Class_name.Method_name().
i.e My_class.class_method()
```

Methods in Python

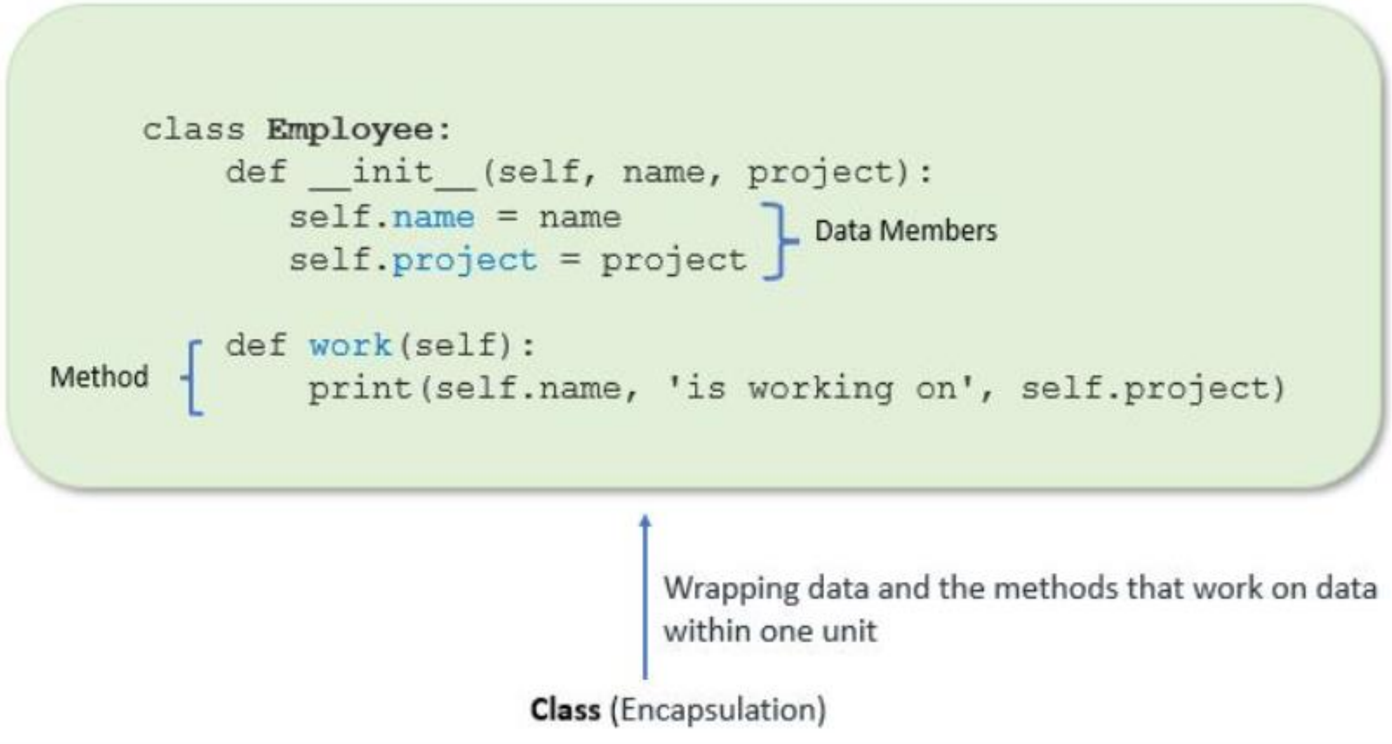
- A static method is bound to a class rather than the objects for that class
- This means that a static method can be called without an object for that class
- In order to define a static method, we can use the @staticmethod decorator
- Example

```
 class Calculator:
     # create addNumbers static method
     @staticmethod
     def addNumbers(x, y):
         return x + y
print('Product:', Calculator.addNumbers(15, 110))
```

- When we need some functionality not w.r.t an Object but w.r.t the complete class, we make a method static
- This is pretty much advantageous when we need to create Utility methods as they aren't tied to an object lifecycle usually

Encapsulation in Python

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



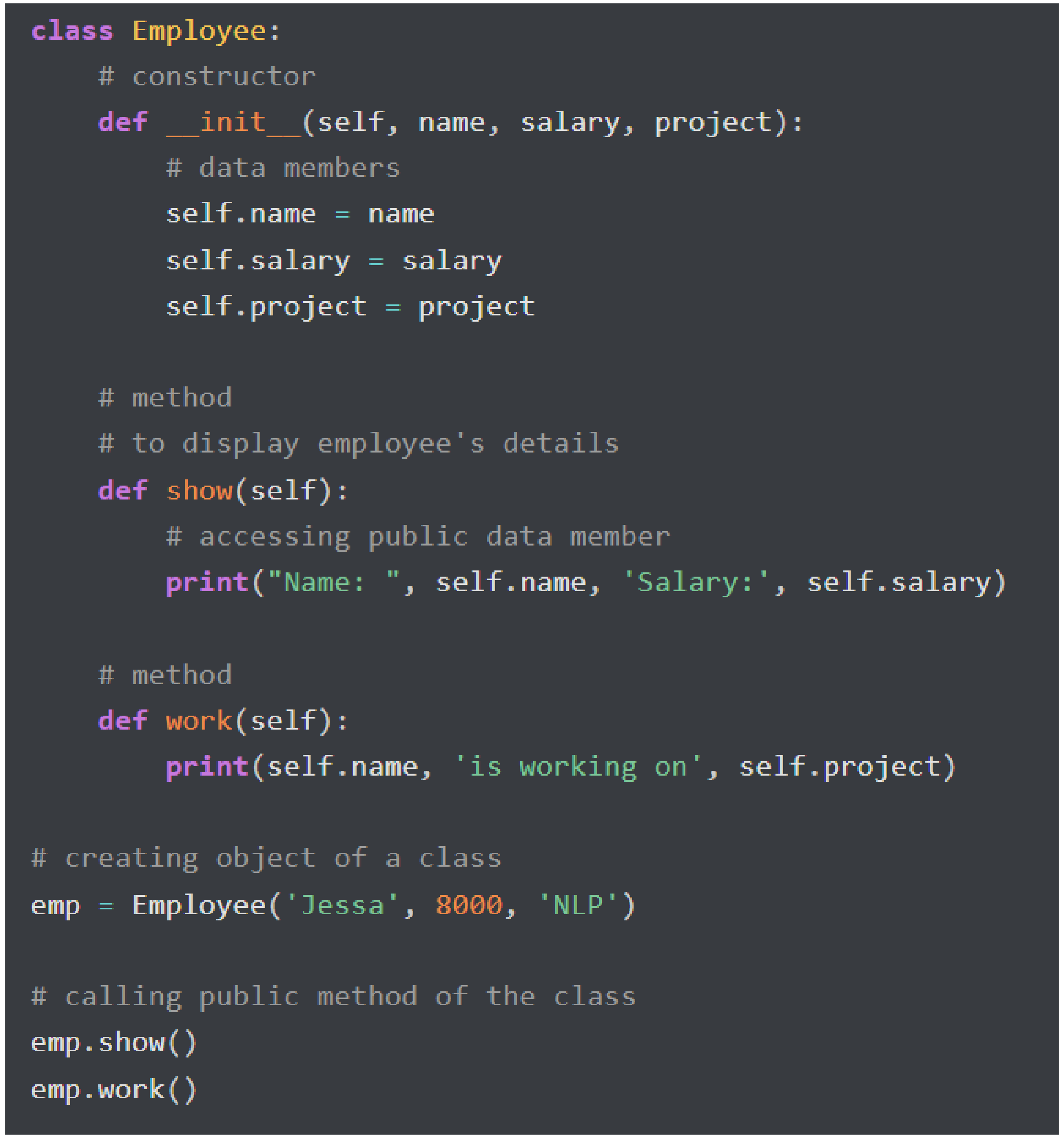
```
class Employee:
    def __init__(self, name, project):
        self.name = name
        self.project = project

    def work(self):
        print(self.name, 'is working on', self.project)
```

Data Members
Method
Wrapping data and the methods that work on data within one unit
Class (Encapsulation)

Implement encapsulation using a class

Encapsulation Example

In this example, we created an Employee class by defining employee attributes such as name and salary as an instance variable and implementing behaviour using work() and show() instance methods.

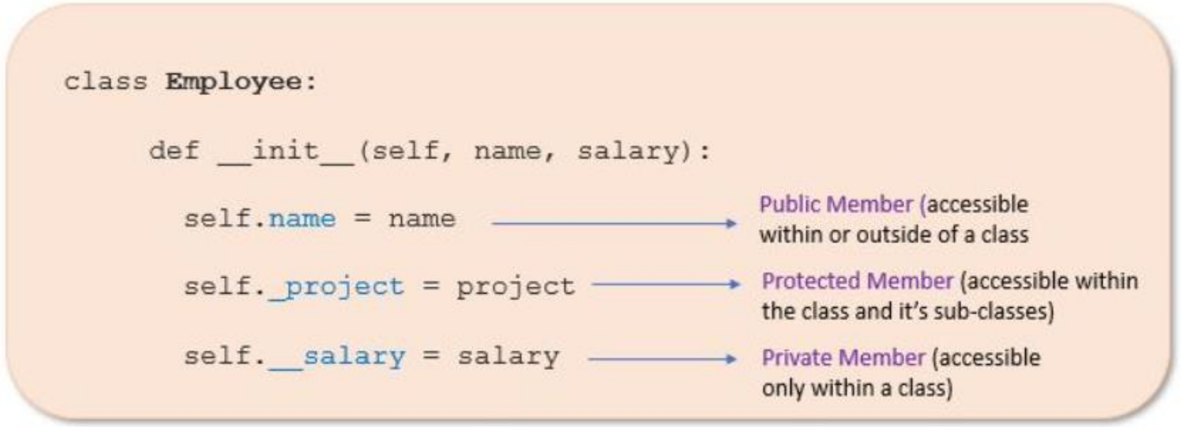


```python
class Employee:
    # constructor
    def __init__(self, name, salary, project):
        # data members
        self.name = name
        self.salary = salary
        self.project = project

    # method
    # to display employee's details
    def show(self):
        # accessing public data member
        print("Name: ", self.name, 'Salary:', self.salary)

    # method
    def work(self):
        print(self.name, 'is working on', self.project)

# creating object of a class
emp = Employee('Jessa', 8000, 'NLP')

# calling public method of the class
emp.show()
emp.work()
```

Access Modifiers in Python

- Encapsulation can be achieved by declaring the data members and methods of a class either as private or protected
- This can be achieved by using single underscore and double underscores
- Access modifiers limit access to the variables and methods of a class
- Python provides three types of access modifiers private, public, and protected

  - Public Member: Accessible anywhere from outside class.
  - Private Member(__): Accessible within the class
  - Protected Member(_): Accessible within the class and its sub-classes



```
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

Public Member

- Public data members are accessible within and outside of a class. All member variables of the class are by default public.

```
class Employee:
    # constructor
    def __init__(self, name, salary):
        # public data members
        self.name = name
        self.salary = salary

    # public instance methods
    def show(self):
        # accessing public data member
        print("Name: ", self.name, 'Salary:’,
                                    self.salary)
```

```
# creating object of a class
emp = Employee('Jessa', 10000)

# accessing public data members
print("Name: ", emp.name, 'Salary:', emp.salary)

# calling public method of the class
emp.show()
```

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

```python
# base class
class Company:
    def __init__(self):
        # Protected member
        self._project = "NLP"

# child class
class Employee(Company):
    def __init__(self, name):
        self.name = name
        Company.__init__(self)

    def show(self):
        print("Employee name :", self.name)
        # Accessing protected member in child class
        print("Working on project :", self._project)

c = Employee("Jessa")
c.show()

# Direct access protected data member
print('Project:', c._project)
```

Private Member

- To define a private variable add two underscores as a prefix at the start of a variable name
- The Private members are accessible only within the class, and we can’t access them directly from the class objects

```
class Employee:
   # constructor
   def __init__(self, name, salary):
      # public data member
      self.name = name
      # private member
      self.__salary = salary

# creating object of a class
emp = Employee('Jessa', 10000)

# accessing private data members
print(‘Name :', emp.name)
print('Salary :', emp.__salary)
```

$\longrightarrow$ *When you Run The Code : The Error Occurs*

Access Private Member

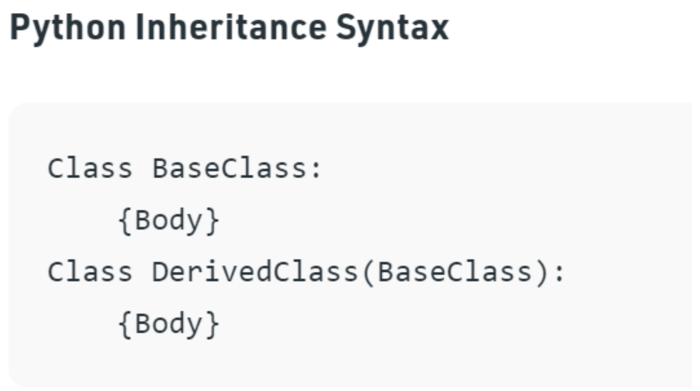
- Access Private member outside of a class using an instance method

```
class Employee:
    # constructor
    def __init__(self, name, salary):
        # public data member
        self.name = name
        # private member

        self.__salary = salary
    # public instance methods
    def show(self):
        # private members are accessible from a class
        print("Name: ", self.name, 'Salary:', self.__salary)
# creating object of a class
emp = Employee('Jessa', 10000)
# calling public method of the class
emp.show()
```

Using Inheritance to Create New Classes

- Inheritance allows us to define a class that inherits all the methods and properties from another class
- **Parent class** is the class being inherited from, also called **base class**
- **Child class** is the class that inherits from another class, also called **derived class**
- Any class can be a parent class, so the syntax is the same as creating any other class

```
class Person:


  def __init__(self, fname, lname):
          self.firstname = fname
          self.lastname = lname

  def printname(self):
          print(self.firstname, self.lastname)

#Use the Person class to create an object, and then execute the print name method:

x = Person("John", "Doe")
x.printname()
```


Python Inheritance Syntax

```
Class BaseClass:
    {Body}
Class DerivedClass(BaseClass):
    {Body}
```

Using Inheritance to Create New Classes

Creating a Child Class & Inheriting from a Base Class/Parent Class

To create a class that inherits the functionality from another class, **send the parent class as a parameter when creating the child class**.

- Create a class named Student, which will inherit the properties and methods from the Person class.

Example:

```
class Student(Person):
  pass
```

<u>**Note:**</u> Use the **pass** keyword when you do not want to add any other properties or methods to the class.

Now the Student class has the same properties and methods as the Person class.

Use the Student class to create an object, and then execute the **printname** method,**this printname method was created in parent class.**

```
x = Student(“varun", “sharma")
x.printname()
```

Using Inheritance to Create New Classes

- Creating a Child Class & Inheriting from a Base Class/Parent class
- Example

```
class Person:
  def __init__(self, fname, lname):
        self.firstname = fname
        self.lastname = lname

  def printname(self):
        print(self.firstname, self.lastname)


class Student(Person):
        pass


x = Student("Varun","Sharma")
x.printname()
```

Output

```
Varun Sharma
```

Using Inheritance to Create New Classes

- Using the Derived[child] Class, extending a parent Class through Inheritance
- Extending a Derived Class method

```
# 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')

# Create object of Car
car = Car()

# access Vehicle's info using car object
car.Vehicle_info()
car.car_info()
```

Output

```
Inside Vehicle class
Inside Car class
```

Polymorphism

- It refers to the use of a single type entity (method, operator or object) to represent different types in different scenarios

- Example

For integer data types, + operator is used to perform arithmetic addition operation, Similarly, for string data types, + operator is used to perform concatenation.

```
num1 = 1
num2 = 2
print(num1+num2)

str1 = "Python"
str2 = "Programming"
print(“str1”+”str2”)
```

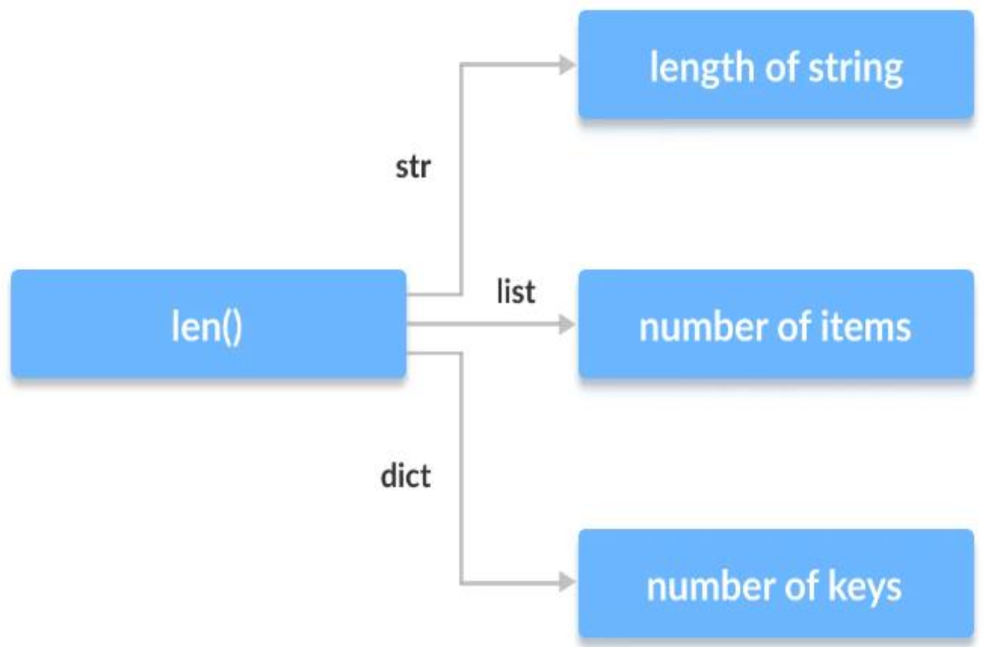
Function Polymorphism in Python

- There are some functions in Python which are compatible to run with multiple data types

- One such function is the len() function. It can run with many data types in Python. Let's look at some example use cases of the function

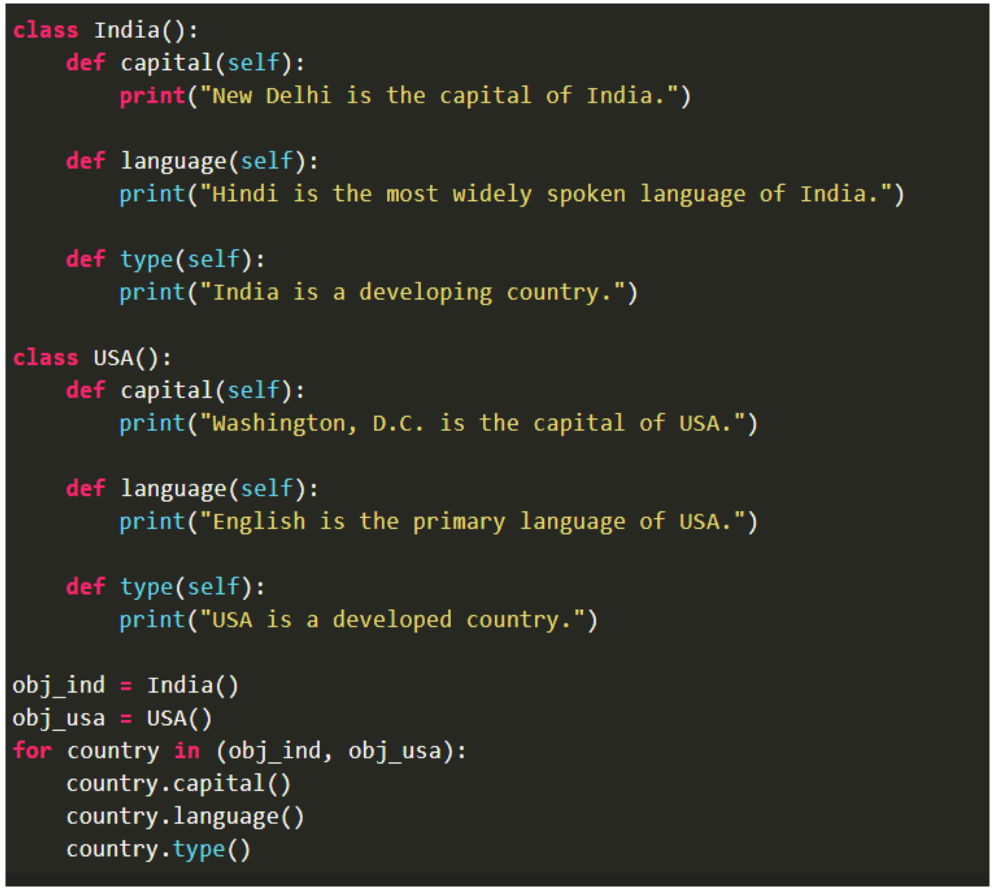
- Example

```
print(len("Programiz"))
print(len(["Python", "Java", "C"]))
print(len({"Name": "John", "Address": "Nepal"}))
```



Class Polymorphism in Python

- Polymorphism while creating class methods as Python allows different classes to have methods with the same name.
- We can then later generalize calling these methods by disregarding the object we are working with.
- Let's look at an example



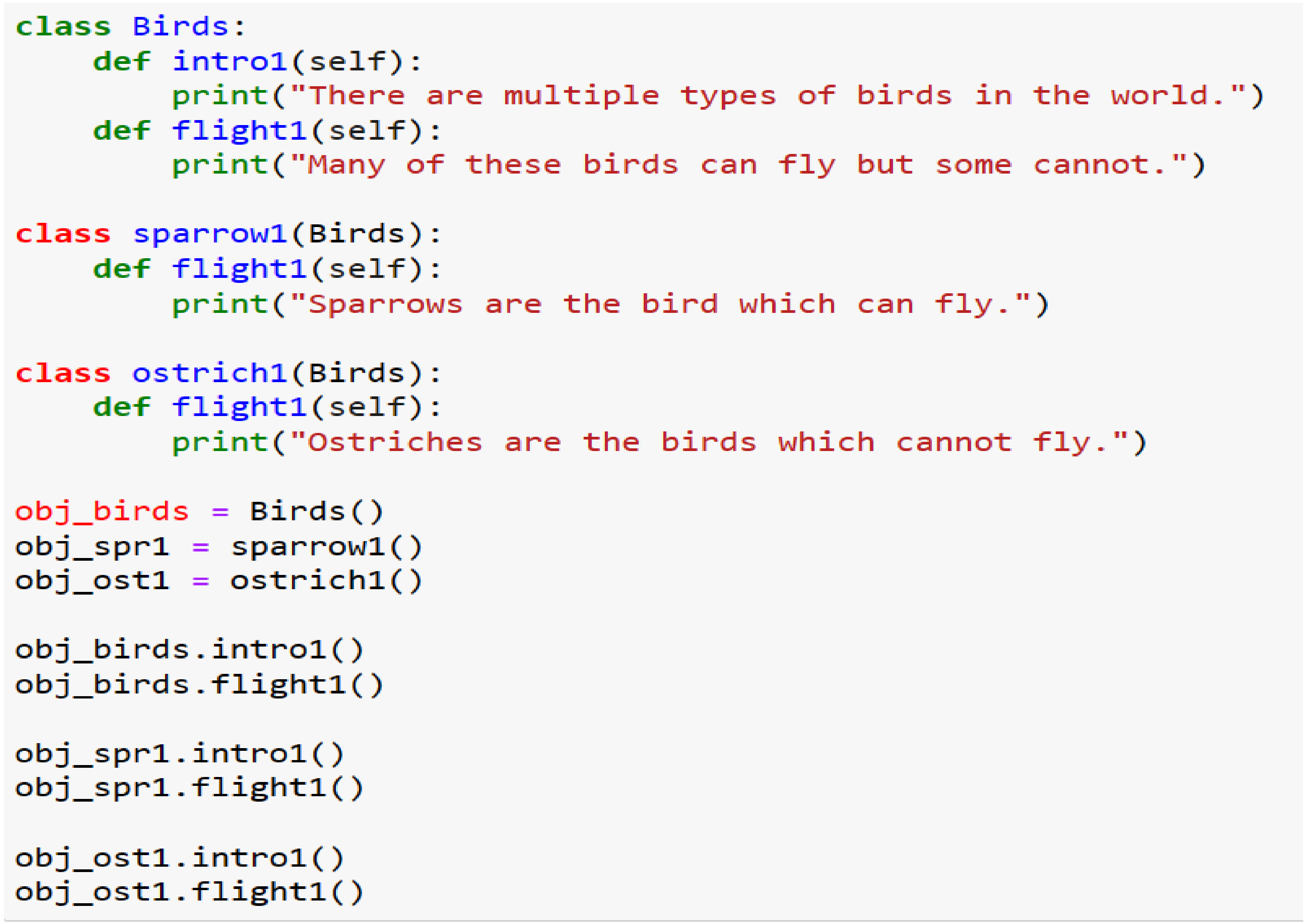
```
class India():
    def capital(self):
        print("New Delhi is the capital of India.")

    def language(self):
        print("Hindi is the most widely spoken language of India.")

    def type(self):
        print("India is a developing country.")

class USA():
    def capital(self):
        print("Washington, D.C. is the capital of USA.")

    def language(self):
        print("English is the primary language of USA.")

    def type(self):
        print("USA is a developed country.")

obj_ind = India()
obj_usa = USA()
for country in (obj_ind, obj_usa):
    country.capital()
    country.language()
    country.type()
```

- We have created two classes India and USA.
- They share a similar structure and have the same method names capital and language.
- However, notice that we have not created a common superclass or linked the classes together in any way.
- Even then, we can pack these two different objects into a tuple and iterate through it using a common country variable.
- It is possible due to polymorphism.

Polymorphism with Inheritance

- Polymorphism allows us to define methods in Python that are the same as methods in the parent classes
- In inheritance, the methods of the parent class are passed to the child class
- It is possible to change a method that a child class has inherited from its parent class
- This is especially useful when the method that was inherited from the parent doesn't fit the child's class
- We re-implement such methods in the child classes. This is Method Overriding

Polymorphism with Inheritance



```python
class Birds:
    def intro1(self):
        print("There are multiple types of birds in the world.")
    def flight1(self):
        print("Many of these birds can fly but some cannot.")

class sparrow1(Birds):
    def flight1(self):
        print("Sparrows are the bird which can fly.")

class ostrich1(Birds):
    def flight1(self):
        print("Ostriches are the birds which cannot fly.")

obj_birds = Birds()
obj_spr1 = sparrow1()
obj_ost1 = ostrich1()

obj_birds.intro1()
obj_birds.flight1()

obj_spr1.intro1()
obj_spr1.flight1()

obj_ost1.intro1()
obj_ost1.flight1()
```

Thank you