Jawaharlal Nehru Technological University Hyderabad

S C D E, AIML Certification Program 2023

Kukatpally, Hyderabad - 500 085, Telangana, India

Object Oriented Programming using Python part 1

Session 12 , 27 May 23

Dr N V Ganapathi Raju
Professor and HOD of IT
Gokaraju Rangaraju Institute of Eng and Tech

Introduction to Object Oriented Programming

- The fundamental differences between **object-oriented systems** and their **traditional counterparts** is the way in which we approach problems.
- Most traditional development methodologies are either **algorithm centric or data centric**.
- In an **algorithmic-centric methodology**, we think of an algorithm that can accomplish the task, then build data structures for that algorithm to use.
- In a **data-centric methodology**, we think how to structure the data, then build the algorithm around that structure.
- In an **Object-Oriented methodology** the algorithm and data structures are packed together as an object, which has a set of attributes and properties.

Object and Class

- Everything in the universe is an object. (Any real world entity; either tangible / intangible)
- Every object has three characteristics: **Identity, State and Behavior**.

Identity : Name of the object

State : Properties / Attributes

Behavior : actions / Functions / Methods/ Responsibilities

```
class class_name:
    def …

     ..
```

- In general object is called as combination of data members and members functions
- A class is a user defined datatype, is a collection of objects; all objects share same properties and behaviors.
- In programming terms: Instance of a class is an object.

Variables

A class can contain have the following variables

- **Local variables** − Variables defined inside methods, or blocks are called local variables. The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.
- **Instance variables** − Instance variables are variables within a class but outside any method. These variables are initialized when the class is instantiated. Instance variables can be accessed from inside any method, constructor or blocks of that particular class.
- **Class variables** − Class variables are variables declared within a class, outside any method, with the **static** keyword.

Methods/functions

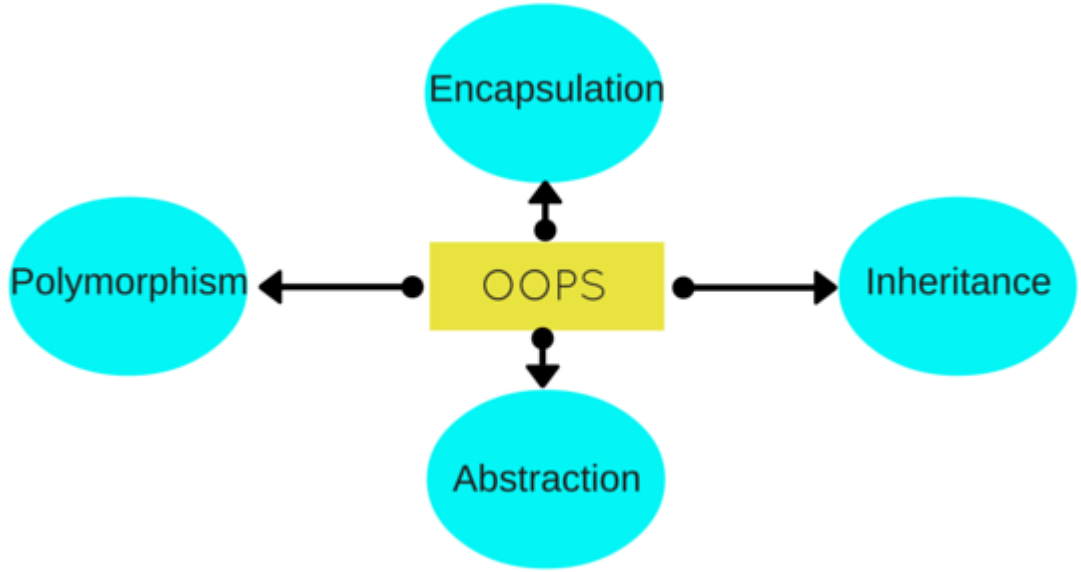
- The syntax of function definition as follows :

```
def function_name(parameters):
    statement(s)
```

- def keyword: This marks the beginning of the function header.
- function_name: This is a unique name that identifies the function.
- parameters or arguments: Values are passed to the function by enclosing them in parentheses (). optional.
- The colon (:) marks the end of the function header.
- statement(s):There must be one or more valid statements that make up the body of the function. Notice that the statements are indented (typically tab or four spaces).
- An optional return statement to return a value from the function.

Principles of Object Oriented Programming

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism



Abstraction

- Data Abstraction is a process of providing only the essential details to the outside world and hiding the internal details, i.e., representing only the essential details in the program.
- Data Abstraction is a programming technique that depends on the separation of the interface and implementation details of the program.
- Class definition is best example for abstraction in Java/C++/Python.

Encapsulation

- Encapsulation is the mechanism that binds together code and data it manipulates and keeps both safe from outside interference and misuse.
- We can create a fully encapsulated class by making all the data members of the class private.
- Now we can use setter and getter methods to set and get the data in it.
- By providing only setter or getter method, you can make the class read-only or write-only.

Inheritance

- **Inheritance** is a mechanism in which one object acquires all the properties and behaviors of a parent object.
- The idea behind inheritance is that you can create new classes that are built upon existing classes.
- When you inherit from an existing class, you can reuse methods and fields of the parent class.
- Moreover, you can add new methods and fields in your current class also.

Polymorphism

- **Polymorphism,** we can perform a *single action in different ways*.
- Polymorphism : "poly" means many and "morphs" means forms. So polymorphism means many forms.
- There are two types of polymorphism : compile-time polymorphism and runtime polymorphism.
- We can perform polymorphism by method overloading and method overriding.
- Method overloading is an example of compile time and method overriding is an example of runtime polymorphism.

Python is objected-oriented

•Python is an object-oriented programming language, key notion is that of an object.

•An object consists of two things: ***data*** and ***functions*** (called methods) that work with that data.

- **strings** in Python are objects.
- •The data of the string object is the actual characters that make up that string.
- •The methods are things like lower, replace, and split.

• In Python, everything is an object.

- That includes not only strings and lists, but also integers, floats, and even functions themselves.

Creating own classes: Steps

•A class is a template for objects. It contains the code for all the object's methods.

• To define a class in Python, you can use the ***class*** keyword, followed by the ***class_name*** and a ***colon***.

•Inside the class, an **__init__** method has to be defined with ***def***.

•The underscores indicate that it is a special kind of method.

•It is called a constructor, and it is automatically called when someone creates a new object from your class.

•The constructor is usually used to set up the class's variables

•**__init__** must always be present. It takes one argument: ***self***, which refers to the object itself.

• To instantiate an object, type the class name, followed by two brackets. We can assign this to a variable.

Note: **self** in Python is equivalent to **this** in C++ or Java.

Creating objects

•Object refers to a particular instance of a class where the object contains variables and methods defined in the class.

•Class objects support two kinds of operations: attribute references and instantiation.

•The term attribute refers to any name (variables or methods) following a dot.

•The act of creating an object from a class is called *instantiation.*

Attribute References

•The names in a class are referenced by objects called ***attribute references.***

•*There are* two kinds of ***attribute references:***

data attributes and ***method attributes***.

•Variables defined within the methods are called ***instance variables*** *and are used to store data values.*

•*New* instance variables are associated with each of the objects that are created for a class. These instance variables are also called ***data attributes.***

•**Method attributes** *are methods inside a* class and are referenced by ***objects of a class.***

•**Attribute references use the standard dot notation syntax as supported in Python.**

Accessing attributes

•The syntax to access data attribute is,

object_name.data_attribute_name

•The syntax to assign value to data attribute is,

object_name.data_attribute_name = value

where value can be of integer, float, string types, or another object itself.

•The syntax to call method attribute is,

object_name.method_attribute_name()

Class instantiation

object_name = ClassName(argument_1, argument_2, ….., argument_n)

- The syntax creates a new object for the class ClassName and assigns object to the variable object_name.
- We can specify any number of arguments during instantiation of the class object.
- Note:
- Method definitions have ***self*** as the first parameter.
- This is because **whenever we call a method using an object, the object itself is automatically passed in as the first parameter to the self parameter variable**.

Example for a class and object

```
class Person:

    def __init__(self):
        print("In Constructor method")

    def receive(self):
        print("Receive method")

    def send(self):
        print("Send method")

p = Person()

p.receive()
p.send()
```

- The first parameter in each method must be ***self***.

- *When self is used, it is just a variable name to which the* object that was created based on a class is assigned.

In Constructor method

Receive method

Send method

Passing arguments to functions

```
class Addition:

    def __init__(self, a, b):
        self.a = a
        self.b = b

    def add(self):
        return self.a + self.b

a = Addition (10, 20)
print(a.add())
```

- To create a class, we use the **class statement.**
- Most classes will have a method called **__init__.**
- The first argument to every method in your class is a special variable called **self.**
- Every time your class refers to one of its variables or methods, it must precede them by self.
- To create a new object from the class, you call the class name along with any values that you want to send to the constructor.
- To use the object's methods, use the dot operator,

Constructor methods

•Python uses a special method called a constructor method. Python allows you to define only one constructor per class.

•Also known as the __init__() method, it will be the first method definition of a class.

```
def __init__(self, parameter_1, parameter_2, …., parameter_n):
    statements
```

•The __init__() method defines and initializes the instance variables.

•It is invoked as soon as an object of a class is instantiated

Example for constructor

```
class Person:
    def __init__(self, name):
        self.person_name = name

    def receive(self):
        print(f"Receive method {self.person_name} Person")

    def send(self):
        print(f"Send method {self.person_name} Person")

p = Person("Ram")
p.receive()
p.send()
```

```
Receive method Ram Person
Send method Ram Person
```

Working with multiple objects

- Multiple objects for a class can be created while attaching a unique copy of data attributes and methods of the class to each of these objects.

```
class Person:
    def __init__(self, name):
        self.person_name = name

    def receive(self):
        print(f"Receive method {self.person_name} Person")

    def send(self):
        print(f"Send method {self.person_name} Person")

p = Person("Ram")
p.receive()
p.send()

p2 = Person("Krishna")
p2.receive()
p2.send()
```

```
Receive method Ram Person
Send method Ram Person

Receive method Krishna Person
Send method Krishna Person
```

Deleting objects

- We can delete the properties of the object or object itself by using the del keyword.

```
class Person:
    def __init__(self, name):
        self.person_name = name

    def receive(self):
        print(f"Receive method {self.person_name} Person")

    def send(self):
        print(f"Send method {self.person_name} Person")

p = Person("Ram")
p.receive()
p.send()

del p
p.receive()
p.send()
```

```
Receive method Ram Person
Send method Ram Person

---------------------------------------------------------------
NameError                                 Traceback
<ipython-input-9-7dcb3f4869cf> in <module>
     14
     15 del p
---> 16 p.receive()
     17 p.send()

NameError: name 'p' is not defined
```

Object assignment

- We can assign one object to another using the assignment **operator in python**.

```
class Person:

    def __init__(self):
        print("In Constructor method")

    def receive(self):
        print("Receive method")

    def send(self):
        print("Send method")

p = Person()
p.receive()

p1 = p
p1.send()
```

```
class Person:

    def __init__(self):
        print("In Constructor method")

    def receive(self):
        print("Receive method")

    def send(self):
        print("Send method")

p = Person()

p.receive()

p1 = p

p1.send()
```

```
In Constructor method
Receive method
Send method
```

Checking IDs of objects

```
class Person:
    def __init__(self):
        print("In Constructor method")
    def receive(self):
        print("Receive method")
    def send(self):
        print("Send method")

p = Person()
p.receive()

p1 = p
p1.send()

print(id(p))
print(id(p1))
print(id(p)==id(p1))
```

```
In Constructor method
Receive method
Send method
654462957320
654462957320
True
```

Objects as arguments

- **An object can be passed to a calling function as an argument.**

- **When we pass an object of a class as an argument, we have access to all the data attributes attached to that object.**

Objects as arguments and returned values

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

class Person:
    def __init__(self, student):
        self.student = student

    def showStudent(self):
        print("Student Name: ",self.student.name)

    def getStudent(self):
        return self.student

s1 = Student("Ram")
p1 = Person(s1)
p1.showStudent()

old_s1 = p1.getStudent()
print("Student Name : ",old_s1.name)
```

Student Name: Ram

Student Name : Ram

Instance, Class, and Static Methods

- **Python offers three types of methods namely**

  instance,

  ***class* and**

  static methods.

Instance Methods

- Instance method receives the instance of the class as the first argument, which by convention is called self, and points to the instance of class. However it can take any number of arguments.
- Using the self parameter, we can access the other attributes and methods on the same object and can change the object state.
- Also, using the self.__class__ attribute, we can access the class attributes, and can change the class state as well.
- Therefore, instance methods gives us control of changing the object as well as the class state.

Class Methods

- A class method accepts the class as an argument.
- It take the class parameter, instead of the object of it.
- It is declared with the @classmethod decorator.
- Class methods are bound to the class and not to the object of the class.
- **They can alter the class state that would apply across all instances of class but not the object state.**

Static Methods

- A static method is marked with a @staticmethod decorator to flag it as static.
- It does not receive an implicit first argument (neither self nor cls).
- It can also be put as a method that “does’t know its class”.
- Hence a static method is merely attached for convenience to the class object.
- Hence static methods can neither modify the object state nor class state.
- They are primarily a way to namespace our methods.

Example Instance, Class, and Static Methods

```
from datetime import date

class Person:

    def __init__(self, name, age):
        self.name = name
        self.age = age

    @classmethod
    def from_birth_year(cls, name, year):
        return cls(name, date.today().year - year)

    @staticmethod
    def is_adult(age):
        return age > 18
```

```
person1 = Person('Venkat', 45)

person2 = Person.from_birth_year('Surya', 1975)

print(person1.name, person1.age)

print(person2.name, person2.age)

print(Person.is_adult(25))
```

```
Venkat 45
Surya 45

True
```

Points to remember

- **Instance methods can access the instance through self as well as the class via self.__class__ attribute.**
- **Class methods can’t access the instance (self) but they have access to the class itself via cls.**
- **Static Methods work like regular functions but belong to the class’s namespace. Static methods don’t have access to cls or self.**

Design a class called RestaurantCheck

It should have the following:

- Fields called check_number, sales_tax_percent, subtotal, total amount need to calculate.
- A constructor that sets the values of all four fields
- A method called calculate_total that takes no arguments (besides self) and returns the total bill including sales tax.
- A method called print_check that writes to a file called check###.txt, where ### is the check number and writes information about the check to that file, formatted like below:

  Check Number: 123
  Sales tax: 8.0%
  Subtotal: $XXXX
  Total: $XXXX

- Test the class by creating a RestaurantCheck object and calling the print_check() method.

Class RestaurantCheck

```
class RestaurantCheck:
    def __init__(self, check_number, sales_tax_percent, subtotal):
        self.check_number = check_number
        self.sales_tax_percent = sales_tax_percent
        self.subtotal = subtotal

    def calculate_total(self):
        return self.subtotal * (1+self.sales_tax_percent/100)

    def print_check(self):
        output_file = open('check' + str(self.check_number) + '.txt', 'w')
        print('Check Number:', self.check_number, file=output_file)
        print('Sales tax: ', self.sales_tax_percent, '%', sep='', file=output_file)
        print('Subtotal: {:.2f}'.format(self.subtotal), file=output_file)
        print('Total: {:.2f}'.format(self.calculate_total()), file=output_file)
        output_file.close()

check = RestaurantCheck(123, 8, 500)
check.print_check()
```

```
Check Number: 123
Sales tax: 8%
Subtotal: 500.00
Total: 540.00
```

Class and Instance variables

- Objects combine code and data in a single entity.
- The basis for an object is a *class*, which defines the object's attributes, methods, and events.
- Generally speaking:
  - instance variables are for data unique to each instance and
  - class variables are for attributes and methods shared by all instances of the class.

Demonstrate Class members and Instance members

```
class student:
    college = "conduira online"  # class variable shared by all instances

    def __init__(self, name):
      self.name= name         # instance variable unique to each instance

stud1 = student('Rama')

stud2 = student('Krishna')

print(student.college)

print(stud1.college)          # shared by all students

print(stud2.college)          # shared by all students

print(stud1.name)             # unique to stud1

print(stud2.name)             # unique to stud1
```

```
conduira online
conduira online
conduira online
Rama
Krishna
```

Demonstrate Instance members

```
class student:

    def __init__(self):
        self.sub1=0
        self.sub2=0

    def Compute_Avg(self):
        res = (self.sub1 + self.sub2)/2;
        print(f"Result is {res}")

stud1 = student()
stud1.Compute_Avg()

stud2 = student()
stud2.sub1=88
stud2.sub2=90

stud2.Compute_Avg()
```

Result is 0.0
Result is 89.0

Abstraction and Encapsulation

- Abstraction is a process where you show only “relevant” variables that are used to access data and “hide” implementation details of an object from the user.
- Encapsulation is the process of combining variables that store data and methods that work on those variables into a single unit called class.
- In Encapsulation, the variables are not accessed directly; it is accessed through the methods present in the class.
- Encapsulation ensures that the object’s internal representation (its state and behavior) are hidden from the rest of the application.
- Thus, encapsulation makes the concept of data hiding possible.

Abstraction student example

```
class student:
    college = "conduira online"

    def __init__(self, no,name, sub1, sub2):
        self.no= no
        self.name = name
        self.sub1=sub1
        self.sub2=sub2

    def Compute_Avg(self):
        res = (self.sub1 + self.sub2)/2;
        print(f"College is  {student.college}")
        print(f"Number is {self.no}")
        print(f"Name is {self.name}")
        print(f"Average is {res}")
```

```
stud3 = student(100,'Rama', 70, 80)
stud3.Compute_Avg()
print()
stud4 = student(101,'Krishna', 50,60);
stud4.Compute_Avg()
```

```
College is  conduira online
Number is 100
Name is Rama
Average is 75.0

College is  conduira online
Number is 101
Name is Krishna
Average is 55.0
```

Abstraction bank example

```
class Bank:

    def __init__(self, name):
        self.user_name = name
        self.balance = 0.0

    def show_balance(self):
        print(f"{self.user_name} has balanceRs.{self.balance} ")

    def withdraw_money(self, amount):
        if amount > self.balance:
            print("You don't have sufficient funds")
        else:
            self.balance -= amount
            print(f"{self.user_name} withdrawn Rs.{self.balance}")
    def deposit_money(self, amount):
            self.balance += amount
            print(f"{self.user_name} deposited  Rs.{self.balance} ")
```

```
acc = Bank("Conduira")
acc.deposit_money(1000)
acc.show_balance()
acc.withdraw_money(500)
acc.show_balance()
```

```
Conduira deposited  Rs.1000.0
Conduira has balanceRs.1000.0
Conduira withdrawn Rs.500.0
Conduira has balanceRs.500.0
```

Encapsulation

```
class Addition:

    def __init__(self, a, b):
        self.a = a
        self.b = b

    def add(self):
        return self.a + self.b

add_obj = Addition(3,4)
print(add_obj.add())
```

7

Encapsulation

Abstraction

Encapsulation

- **Getters**:- These are the methods used in Object-Oriented Programming (OOPS) which helps to access the private attributes from a **class**.
- **Setters**:- These are the methods used in OOPS feature which helps to set the value to private attributes in a **class**.
- **EncTest** has three methods.
- **__init__**:- It is used to initialize the attributes or properties of a **class**.
  - **__a**:- It is a private attribute.
- **get_a**:- It is used to get the values of private attribute **a**.
- **set_a**:- It is used to set the value of **a** using an **object** of a class.
- You are not able to access the private variables directly in **Python**. That's why you implemented the **getter** method.

Encapsulation

```
class EncTest:

    def __init__(self, a):
        self.__a = a

    def get_a(self):
        return self.__a

    def set_a(self, a):
        self.__a = a

e = EncTest(10)
print(e.get_a())

e.set_a(50)
print(e.get_a())
```

```
10
50
```

Default arguments

```
class student:

    def __init__(self, name= "Conduira Online", test1= 80 , test2 = 70):
        self.name=name
        self.test1=test1
        self.test2=test2

    def Compute_Avg(self):
        res = (self.test1 + self.test2)/2;
        print(f"Name is {self.name}")
        print(f"Test1 is {self.test1}")
        print(f"Test2 is {self.test2}")
        print(f"Result is {res}")
```

```
stud1 = student()
stud1.Compute_Avg()

print()

stud2 = student("Conduira Offline", 40,50)
stud2.Compute_Avg()
```

```
Name is Conduira Online
Test1 is 80
Test2 is 70
Result is 75.0

Name is Conduira Offline
Test1 is 40
Test2 is 50
Result is 45.0
```