



**Department of Computer Science and Engineering
PES University, Bangalore, India**

Lecture Notes

Python for Computational Problem Solving

UE23CS151A

Lecture #99 & #100
Classes and Objects

By,
Prof. Sindhu R Pai,
Anchor, PCPS - 2023
Assistant Professor
Dept. of CSE, PESU
&
Dr. Ramya C
Associate Professor
Dept. of CSE, PESU

Many Thanks to
Dr. Shylaja S S (Director, CCBD and CDSAML Research Centers, Former
Chairperson, CSE, PES University)
Prof. Chitra G M (Asst. Prof, Dept. of CSE, PCPS Anchor – 2022)

Introduction

In the last lecture, we discussed the features of Object Oriented Programming – Encapsulation, Data hiding, Polymorphism, Inheritance and etc. To implement these features, **classes and objects** are necessary.

The keyword - class in python

A class is a **methodology to create an entity**. It is a **model or a blueprint** of very specific part of reality, reflecting properties and activities found in the real world. **A class specifies the set of instance variables and methods(same as functions but inside a type) that are “bundled together” for defining a user defined type.**

Objects in python

Python supports different kinds of data .The data can be integer, floating point number, string, lists, dictionaries, tuples etc. **An object is an instance of a type/class. Each object has the following:**

- A name that uniquely identifies it.
- A set of individual properties which makes it original or unique or outstanding.
- A set of abilities, called methods to perform specific activities, able to change the object itself, or some of the other objects.

Examples of Objects and types:

- Number **157** is an instance of type/class **int**
- **“hello”** is an instance of type/class **str**

Let’s consider a real world example - Snoopy is a fat dog which eats all day

Objectname: Snoopy

Class: Dog

Attribute: Weight(fat)

Activity: Eat

Creation of a type and its objects:

A language has only a few **types of predefined classes namely strings, numbers, lists, tuples, sets and dictionaries**. It cannot provide all possible types we may want to have like employee, book, baby, library, college, person, table, computer, dog etc. Python provides a mechanism to make our **own type using the class keyword**. **Classes allow you to define the attributes and behaviors that characterize anything you want to model in your program i.e. a class by itself is a type and implementation.**

Syntax: The simplest form of class definition looks like this.

```
class class_name:
    #Attributes
    #Methods          #Function associated with the class.
```

Following the class keyword is the name of the class. The name of the class should be followed by a colon.

Let us create a class type called Student who has attributes such as name, srn, phno, address and one of the behaviour of it is to display all the attributes.

```
class Student:
    name =
    phno =
    address =
    def display():
        pass
```

```
C:\Users\Dell>python testclass.py
File "C:\Users\Dell\testclass.py", line 2
    name =
    ^
SyntaxError: invalid syntax
```

But in the above code, there is no value associated with any of the attributes. So the definition of a type is incomplete. **Results in SyntaxError**. Let us give some values to these attributes inside the class directly. No error. But these are class attributes [Later section]

```
class Student:
    name = "SINDHU"
    phno = 87656742
    address = "Jayanagar"
    def display():
        pass
```

```
C:\Users\Dell>python testclass.py
C:\Users\Dell>
```

```
print(Student, type(Student))
```

Now, if you add this in the code outside the class, observe the output showing the type of created class.

```
C:\Users\Dell>python testclass.py
<class '__main__.Student'> <class 'type'>
```

We did not create any objects yet. Just created a class/type. The existence of a class does not mean that any of the compatible objects will automatically be created. The class itself isn't able to create an object – you have to create it yourself and Python allows you to do this.

To create an object for the type, we need to **use the function name which has the same name as the class.**

Object_name = className() #This calls the default constructor function. __init__()

Let us try to create three objects of the student class/type.

```
s1 = Student()
s2 = Student()
s3 = Student()
print(s1, s2, s3)
print(type(s1), type(s2), type(s3))
```

```
C:\Users\Dell>python testclass.py
<__main__.Student object at 0x000001E22E074590> <__main__.Student object at 0x000001E22E074510> <__main__.Student object at 0x000001E22E0745D0>
<class '__main__.Student'> <class '__main__.Student'> <class '__main__.Student'>
```

Let us access name, phno and address using these objects.

```
print(s1.name, s1.phno, s1.address)
print(s2.name, s2.phno, s2.address)
print(s3.name, s3.phno, s3.address)
```

```
C:\Users\Dell>python testclass.py
SINDHU 87656742 Jayanagar
SINDHU 87656742 Jayanagar
SINDHU 87656742 Jayanagar
```

Observe that all objects are having the same details. Think about the creation of objects with different values for instance variables.

Consider the type Student once again by adding the init() method within it. **The init() is a constructor method called during the object creation. Called automatically when the function name is same as the class name.**

```
class Student:
    def __init__(self):
        print("I am in init")
s1 = Student()
s2 = Student()
s3 = Student()
```

```
C:\Users\Dell\B\Unit4>python oop1.py
I am in init
I am in init
I am in init
```

Instance variables:

Attributes / Variables associated with the instance/object of the class. This is used to make sure that **attributes of each student are allocated memory separately in an object.**

Consider the Student type having the attributes – Name, age, Phno and Address. Each object must be created with these details. This is possible by making use of function which has the same as the class name.

class Student: #class creation

```
def __init__(self, name, age, addr):
    #instance variables/ attributes/properties
    self.name = name
    self.age = age
    self.addr = addr
    print("i m in init")
def display(self):
    print(self.name+ " "+str(self.age)+ " "+self.addr)
```

#object creation and demo display()

```
s1 = Student("Sindhu", 20, "jayanar")
s2 = Student("Indhu", 56, "JP Nagara")
s3 = Student("Bindu", 34, "silkboard")
s1.display()
s2.display()
s3.display()
```

```
C:\Users\Dell\B\Unit4>python oop1.py
i m in init
i m in init
i m in init
Sindhu 20 ,jayanar
Indhu 56 JP Nagara
Bindu 34 silkboard
```

Note:

- **The self is not a keyword.** It is just a good naming convention to indicate that the object itself is invoking the methods. Self refers to the current object. This is passed

implicitly to all the instance methods of the class. Self is a **parameter that refers to the object that is used to access the class attributes and methods.**

- It is possible to add the attributes and values to objects on the fly outside the class using the object name.
- All instance variable are public by default. Accessible outside the class using the dot(.) operator with the object name.

Constructor:

When an object is created, a special function of the class is called for initializing the attributes. This method is called as constructor. The name of the constructor in Python is `__init__`. If a class has a constructor, it is invoked automatically and implicitly when the object of the class is instantiated.

“Constructor in Python is used to define the attributes of an instance and assign values to them”

Types of Constructors

- **Non-Parameterized constructor: Only self is the parameter**

```
class Person:
    def __init__(self):
        print("without parameters")
p = Person()
# Person.__init__(p) internal modifications done by the runtime
```

```
C:\Users\Dell\B\Unit4>python oop1.py
without parameters
```

- **Parameterized constructor: Arguments are sent in the function that have same name as the class name.**

```
class Person:
    def __init__(self,name,age):
        self.name = name
        self.age = age
        print("in init - parameterized")
p = Person("THAKSH",30) # Person.__init__(p, "THAKSH",30)
```

```
C:\Users\Dell\B\Unit4>python oop1.py
in init - parameterized
C:\Users\Dell\B\Unit4>
```

Example_code1: Printing the object to understand better.

```
class Person:
    def __init__(self,name,age):
        print(self)
        self.name = name
        self.age = age
        print("in init - parameterized")
p = Person("THAKSH",30)
print(p)
```

```
C:\Users\Dell\B\Unit4>python oop1.py
<__main__.Person object at 0x000001C7E3FF4610>
in init - parameterized
<__main__.Person object at 0x000001C7E3FF4610>
```

Hence by looking at the output, we can make out that an object of class Person has got created at a particular location, 0x00001C7...4610.

Destructors:

Destructors are called when an object gets destroyed. In Python, destructors are not needed as much as in C++ because Python has a garbage collector that handles memory management automatically. The `__del__()` method is known as a destructor method in Python. It is called when all references to the object have been deleted i.e when an object is garbage collected. **A reference to objects is also deleted when the object goes out of reference or when the program ends.**

```
class Person:
    def __init__(self,name,age):
        self.name = name
        self.age = age
        print("i am in constructor")
    def __del__(self):
        print("i am in destructor")
p1 = Person("SINDHU",30)
p2 = Person("SHYAMA",35)
p3 = Person("JAYA", 45)
```

```
C:\Users\Dell>python testdel.py
i am in constructor
i am in constructor
i am in constructor
i am in destructor
i am in destructor
i am in destructor
```

Points to note:

- Destructors are not called explicitly. If you want to call , it is possible using either del operator or `__del__` function
- Sequence of destructor calls need not be same as the constructor calls. It might differ.

Getters and Setters:

The main purpose of using getters and setters in object-oriented programs is to ensure **data encapsulation**. But private variables in python are not actually hidden/private fields like in other object oriented languages. They are only private like variables if you start the variable with `__` while defining the instance variables or class variables [Discussed in the next section]. Getters and Setters in python are often used when you want to add validation logic around getting and setting a value.

Getter: A function which returns the value of an instance variable.

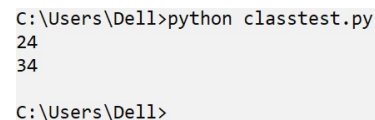
Setter: A function which sets the value to an instance variable.

Usually for all the instance variables, good to write getters and setters.

Private like variable is defined using `__`(2 underscores) in the beginning of the variable creation. They are accessible outside the class using the classname. -> **`_className__variable`**

Example_code_2: Accessing the private like variable outside the class

```
class A:
    def __init__(self):
        self.__a = 34
        self.b = 24
a1 = A()
print(a1.b)
#print(a1.__a) #Error
print(a1._A__a)
```



```
C:\Users\Dell>python classtest.py
24
34
C:\Users\Dell>
```

Example_code_3: Usage of setters and getters

```
class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def display(self):
        print(self.get_name(), self.get_age()) #usage of getter in display
    def set_name(self, name): #setter for name field
        self.name = name
    def set_age(self, age): #setter for age field
        self.age = age
#getters for name and age are as below.
```



```

def get_name(self):
    return self.name
def get_age(self):
    return self.age
s1= Student("abc", 56)
s1.display()
s2 = Student("pqr", 88)
s2.display()
s1.set_name("def")
print("modified s1 name-----", s1.get_name())
s2.set_age(100) #this can also be set without using the setter in python
#s2.age(100)
print("modified s2 age-----", s2.get_age())
s1.display()
s2.display()

```

```

C:\Users\Dell>python classtest.py
abc 56
pqr 88
modified s1 name----- def
modified s2 age----- 100
def 56
pqr 100

```

Class variables:

Attributes and methods shared by all instances(objects) of the class. Ideally used with the classname and a dot operator and the variable/method.

```

class Sample:
    a = 0
    def __init__(self):
        a = a+1
s1 = Sample()
print(s1.a)

```

```

C:\Users\Dell>python classtest.py
Traceback (most recent call last):
  File "C:\Users\Dell\classtest.py", line 5, in <module>
    s1 = Sample()
    ~~~~
  File "C:\Users\Dell\classtest.py", line 4, in __init__
    a = a+1
    ~
UnboundLocalError: cannot access local variable 'a' where it is not associated with a value

```

Use a class variable a with the class name. -> Sample.a

```

class Sample:
    a = 0
    def __init__(self):
        Sample.a = Sample.a+1
s1 = Sample();print(s1.a)
s1.a = 67
print(s1.a)
s2 = Sample()
print(s2.a)
s2.a = 36
print(s2.a)

```

```

C:\Users\Dell>python classtest.py
1
67
2
36

```

If we consider the student type, requirement is to increment the count of student objects created as and when the students are registering for a given course. Decrement the count when students are willing to join for some other course.

Think! -> Can we create the count variable as an instance variable? If we do this, count will be created for every object. We want to allocate memory once and update that memory location every time student object created or deleted.

Example_code_4: Usage of class variable for the above requirement.

```
class Student:
    count = 0
    def __init__(self, name, age):
        self.age = age
        self.name = name
        Student.count += 1
s1 = Student("abc",66)
print("count of students =",s1.count)
s2 = Student("abcd",61)
print("count of students =",s1.count)
```

```
C:\Users\Dell>python classtest.py
count of students = 1
count of students = 2
```

But in the above code, count is a class variable. Conventionally, the variable count must be accessed using the class name only. So, replacing s1.count by Student.count in both the places, output remains the same. What if we want to add a method to access this count? Output remains the same.

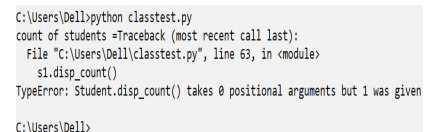
```
class Student:
    count = 0
    def __init__(self, name, age):
        self.age = age;    self.name = name;
        Student.count += 1
    def disp_count(self):
        print(Student.count)
s1 = Student("abc",66); print("count of students =", end = "")
s1.disp_count()
s2 = Student("abcd",61); print("count of students =",end = "")
s2.disp_count()
```

But, in the above code, `disp_count` is a function which takes `self` as the parameter, but never used inside a method? Can we avoid `self` and call the function using the class name? Output remains the same.

```
class Student:
    count = 0
    def __init__(self, name, age):
        self.age = age
        self.name = name
        Student.count += 1
    def disp_count(): #self removed
        print(Student.count)
s1 = Student("abc",66)
print("count of students =", end = "")
Student.disp_count()
s2 = Student("abcd",61)
print("count of students =",end = "")
Student.disp_count()
```

Now can we call the function using object i.e., `s1` and `s2`? Results in Error

```
class Student:
    count = 0
    def __init__(self, name, age):
        self.age = age
        self.name = name
        Student.count += 1
    def disp_count():
        print(Student.count)
s1 = Student("abc",66)
print("count of students =", end = "")
s1.disp_count()
s2 = Student("abcd",61)
print("count of students =",end = "")
s2.disp_count()
```



```
C:\Users\De11\python classtest.py
count of students =Traceback (most recent call last):
  File "C:\Users\De11\classtest.py", line 63, in <module>
    s1.disp_count()
TypeError: Student.disp_count() takes 0 positional arguments but 1 was given

C:\Users\De11>
```

To avoid error in the above code, we use `@staticmethod`. When function decorated with `@staticmethod` is called, runtime doesn't pass an instance of the class to it as it is normally done with methods. It means that the function is put inside the class but it cannot access the instance of that class.

A static method is a method which is **bound to the class and not the object of the class**. It is present in a class because it makes sense for the method to be present in class. **A static method does not receive an implicit first argument.**

Features:

- Allocated memory once when the object for the class is created for the first time.
- Created outside of methods but inside a class
- It can be accessed through a class but not directly with an instance if no decorator called
- Behavior doesn't change for every object.

Advantages:

- **Memory efficiency:** Since static variables are shared among all instances of a class, they can save memory by avoiding the need to create multiple copies of the same data.
- **Shared state:** Static variables can provide a way to maintain shared state across all instances of a class, allowing all instances to access and modify the same data.
- **Easy to access:** Static variables can be accessed using the class name itself, without needing an instance of the class. This can make it more convenient to access and modify the data stored in a static variable.
- **Initialization:** Static variables can be initialized when the class is defined, making it easy to ensure that the variable has a valid starting value.
- **Readability:** Static variables can improve the readability of the code, as they clearly indicate that the data stored in the variable is shared among all instances of the class.

```
class Student:
    count = 0
    def __init__(self, name, age):
        self.age = age;    self.name = name
        Student.count += 1
    @staticmethod
    def disp_count():
        print(Student.count)
```

```
s1 = Student("abc",66)
print("count of students =", end = "")
s1.disp_count()
s2 = Student("abcd",61)
print("count of students =",end = "")
s2.disp_count()
```

```
C:\Users\Dell>python classtest.py
count of students = 1
count of students = 2
```

Example_code_5: Using the collection as a class variable

```
class Animal:
    L1=[]
    def __init__(self,name):
        self.name = name
    def add(self,activity):
        self.L1.append(activity)
a=Animal('pinky')
b=Animal('snow')
a.add('roll over')
b.add('playdead')
print( a.L1 )
print(b.L1)
```

Output:

```
>>>
= RESTART: C:/Users/cramy/AppDat
y
['roll over', 'play dead']
['roll over', 'play dead']
>>>
```

Try this:

- Create a Rectangle type with two attributes - > length and breadth. Create few instances and find the area for every instance. Display the length, breadth and area. Add getters and setters. If user wants to modify only length or only breadth, use these functions in the client code accordingly.
- Create a complex class to perform addition, subtraction, multiplication and division of two complex numbers. Use real and imaginary part as its attributes.

-END-