18 Classes and Objects

"World is OO, you too should be..."

Contents

KanNotes

Programming Paradigms

- Paradigm means the principle according to which a program is organized to carry out a given task.
- Python supports three programming paradigms—Structured programming, Functional Programming and Object-oriented programming (OOP). We had a brief introduction to these paradigms in Chapter 1.

What are Classes and Objects?

- World is object oriented. It is full of objects like Sparrow, Rose, Guitar, Keyboard, etc.
- Each object is a specific instance of a class. For example, Sparrow is a specific instance of a Bird class or Rose is a specific instance of a Flower class.
- More examples of classes and objects in real life:

  Bird is a class. Sparrow, Crow, Eagle are objects of Bird class.
  Player is a class. Sachin, Rahul, Kapil are objects of Player class.
  Flower is a class. Rose, Lily, Gerbera are objects of Flower class.
  Instrument is a class. Sitar, Flute are objects of Instrument class.
- A class describes two things—the form an object created from it will take and functionality it will have. For example, a Bird class may specify the form in terms of weight, color, number of feathers, etc. and functionality in terms of flying, hopping, chirping, eating, etc.
- The form is often termed as properties and the functionality is often termed as methods. A class lets us bundle data and functionality together.
- When objects like Sparrow or Eagle are created from the Bird class the properties will have values. The methods can either access or manipulate these values. For example, the property weight will have value 250 grams for a Sparrow object, but 10 Kg for an Eagle object.
- Thus class is generic in nature, whereas an object is specific in nature.

- Multiple objects can be created from a class. The process of creation of an object from a class is called instantiation.

Classes and Objects in Programming

- In Python every type is a class. So **int**, **float**, **complex**, **bool**, **str**, **list**, **tuple**, **set**, **dict** are all classes.
- A class has a name, whereas objects are nameless. Since objects do not have names, they are referred using their addresses in memory.
- When we use a simple statement **num = 10**, a nameless object of type **int** is created in memory and its address is stored in **num**. Thus **num** refers to or points to the nameless object containing value 10.
- However, instead of saying that **num** refers to a nameless **int** object, often for sake of convenience, it is said that **num** is an **int** object.
- More programmatic examples of classes and objects:

```
a = 3.14                # a is an object of float class
s = 'Sudesh'            # s is an object of str class
lst = [10, 20, 30]      # lst is an object of list class
tpl = ('a', 'b', 'c')   # tpl is an object of tuple class
```

- Different objects of a particular type may contain different data, but same methods. Consider the code snippet given below.

```
s1 = 'Rupesh'           # s1 is object of type str
s2  = 'Geeta'           # s2 is object of type str
```

  Here **s1** and **s2** both are **str** objects containing different data, but same methods like **upper( )**, **lower( )**, **capitalize( )**, etc.

- The specific data in an object is often called **instance data** or **properties** of the object or **state** of the object or **attributes** of the object. Methods in an object are called **instance methods**.

User-defined Classes

- In addition to providing ready-made classes like **int**, **str**, **list**, **tuple**, etc., Python permits us to define our own classes and create objects from them.

- The classes that we define are called user-defined data types. Rules for defining and using a user-defined class and a standard class are same.

- Let us define a user-defined class **Employee**.

```
class Employee :
    def set_data(self, n, a, s) :
        self.name = n
        self.age = a
        self.salary = s

    def display_data(self) :
        print(self.name, self.age, self.salary)

e1 = Employee( )
e1.set_data('Ramesh', 23, 25000)
e1.display_data( )
e2 = Employee( )
e2.set_data('Suresh', 25, 30000)
e2.display_data( )
```

- The **Employee** class contains two methods **set_data( )** and **display_data( )** which are used to set and display data present in objects created from Employee class.

- Two nameless objects get created through the statements:

```
e1 = Employee( )
e2 = Employee( )
```

  Addresses of the nameless objects are stored in **e1** and **e2**.

- In principle both the nameless objects should contain instance data **name**, **age**, **salary** and instance methods **set_data( )** and **display_data( )**.

- In practice each object has its own instance data **name**, **age** and **salary**, whereas the methods **set_data( )** and **display_data( )** are shared amongst objects.

- Instance data is not shared since instance data values would be different from one object to another (Refer Figure 18.1).

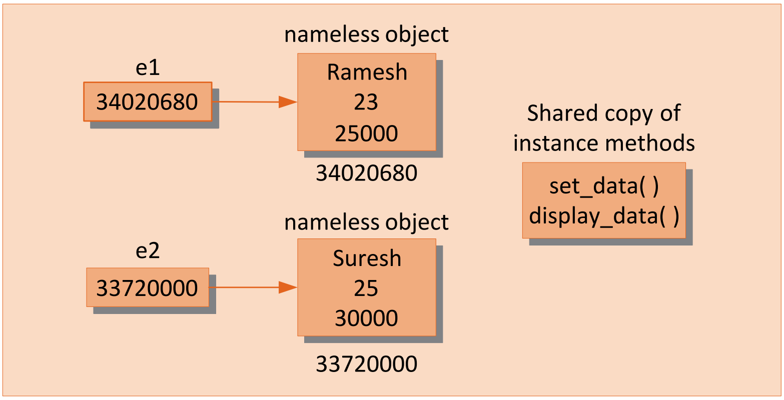

Figure 18.1

- The syntax to call an object's method is **object.method( )**, as in **e1.display_data( )**.
- Whenever we call an instance method using an object, address of the object gets passed to the method implicitly. This address is collected by the instance method in a variable called **self**.
- Thus, when **e1.set_data('Ramesh', 23, 25000)** calls the instance method **set_data( )**, first parameter passed to it is the address of object, followed by values 'Ramesh', 23, 25000.
- Within **set_data( ) self** contains the address of first object. Likewise, when **set_data( )** is called using **e2**, **self** contains address of the second object.
- Using address of the object present in **self** we indicate which object's instance data we wish to work with. To do this we prepend the instance data with **self.**, as **in self.name, self.age** and **self.salary**.
- **self** is like **this** pointer of C++ or **this** reference of Java. In place of **self** any other variable name can be used.

Access Convention

- We have accessed instance methods **set_data( )** and **display_data( )** from outside the class. Even instance data name, age and salary are accessible from outside the class. Thus, following statements would work:

```
e3 = Employee( )
```

```
e3.name = 'Rakesh'
e3.age = 25
```

- However, it is a good idea to keep data in a class inaccessible from outside the class and access it only through member functions of the class.
- There is no mechanism or keyword available in Python to enforce this. Hence a convention is used to start the instance data identifiers with two leading underscores (often called dunderscore, short for double underscore). Example: **__name**, **__age** and **__salary**.

Object Initialization

- There are two ways to initialize an object:

  Method 1 : Using methods like **get_data( )** / **set_data( )**.
  Method 2 : Using special method **__init__( )**

- **get_data( )** can receive data from keyboard into instance data variables. **set_data( )** can set up instance data with a values that it receives. The benefit of this method is that the data remains protected from manipulation from outside the class.
- The benefit of initializing an object using the special method **__init__( )** is that it guarantees initialization, since **__init__( )** is always called when an object is created.
- Following program illustrates both these methods:

```
class Employee :
    def set_data(self, n, a, s) :
        self.__name = n
        self.__age = a
        self.__salary = s

    def display_data(self) :
        print(self.__name, self.__age, self.__salary)

    def __init__(self, n = ' ', a = 0, s = 0.0) :
        self.__name = n
        self.__age = a
        self.__salary = s

    def __del__(self) :
        print('Deleting object' + str(self))

e1 = Employee( )
e1.set_data('Suresh', 25, 30000)
e1.display_data( )
e2 = Employee('Ramesh', 23, 25000)
e2.display_data( )
e1 = None
e2 = None
```

On execution of this program, we get the following output:

```
Ramesh 23 25000
Suresh 25 30000
Deleting object<__main__.Employee object at 0x013F6810>
Deleting object<__main__.Employee object at 0x013F65B0>
```

- The statements

```
e1 = Employee( )
e2 = Employee('Ramesh', 23, 25000)
```

  create two objects which are referred by **e1** and **e2**. In both cases **__init__( )** is called.

- Whenever an object is created, space is allocated for it in memory and **__init__( )** is called. So address of object is passed to **__init__( )**.

- **__init__( )**'s parameters can take default values. In our program they get used while creating object **e2**.

- **__init__( )** doesn't return any value.

- If we do not define **__init__( )**, then Python inserts a default **__init__( )** method in our class.

- **__init__( )** is called only once during entire lifetime of an object.

- A class may have **__init__( )** as well as **set_data( )**.

  __init__( ) – To initialize object.
  set_data( ) – To modify an already initialized object.

- **__del__( )** method gets called automatically when an object goes out of scope. Cleanup activity, if any, should be done in **__del__( )**.

- **__init__( )** method is similar to constructor function of C++ / Java.
- **__del__( )** is similar to destructor function of C++.

Class Variables and Methods

- If we wish to share a variable amongst all objects of a class, we must declare the variable as a **class variable** or **class attribute**.
- To declare a class variable, we have to create a variable without prepending it with **self**.
- Class variables do not become part of objects of a class.
- Class variables are accessed using the syntax **classname.varname**.
- Contrasted with instance methods, **class methods** do not receive a **self** argument.
- Class methods can be accessed using the syntax **classname.methodname( )**.
- Class variables can be used to count how many objects have been created from a class.
- Class variables and methods are like static members in C++ / Java.

vars( ) and *dir( )* Functions

- There are two useful built-in functions **vars( )** and **dir( )**. Of these, **vars( )** returns a dictionary of attributes and their values, whereas **dir( )** returns a list of attributes.
- Given below is the sample usage of these functions:

```
import math              # standard module
import functions         # some user-defined module
a = 125
s = 'Spooked'

print(vars( ))           # prints dict of attributes in current module
                         # including a and s
print(vars(math))        # prints dict of attributes in math module
print(vars(functions))   # prints dict of attributes in functions module

print(dir( ))            # prints list of attributes in current module
                         # including a and s
```

```
print(dir(math))         # prints list of attributes in math module
print(dir(functions))    # prints list of attributes in functions module
```

More *vars( )* and *dir( )*

- Both the built-in functions can be used with a class as well as an object as shown in the following program.

```
class Fruit :
    count = 0

    def __init__(self, name = ' ', size = 0, color = ' ') :
        self.__name = name
        self.__size = size
        self.__color = color
        Fruit.count += 1

    def display( ) :
        print(Fruit.count)

f1 = Fruit('Banana', 5, 'Yellow')
print(vars(Fruit))
print(dir(Fruit))
print(vars(f1))
print(dir(f1))
```

On execution of this program, we get the following output:

```
{... ... ... , 'count': 0, '__init__': <function Fruit.__init__>,
'display': <function Fruit.display at 0x7f290a00f598>, ... ... ... }
[ ... ... ... '__init__', 'count', 'display']
{'_name': 'Banana', '_size': 5, '_color': 'Yellow'}
[... ... ... '__init__', '_color', '_name', '_size', 'count', 'display']
```

- When used with class, **vars( )** returns a dictionary of the class's attributes and their values. On the other hand the **dir( )** function merely returns a list of its attributes.

- When used with object, **vars( )** returns a dictionary of the object's attributes and their values. In addition, it also returns the object's class's attributes, and recursively the attributes of its class's base classes.

- When used with object, **dir( )** returns a list of the object's attributes, object's class's attributes, and recursively the attributes of its class's base classes.

P</> *Programs*

Problem 18.1

Write a class called **Number** which maintains an integer. It should have following methods in it to perform various operations on the integer:

```
set_number(self, n)        # sets n into int
get_number(self)           # return current value of int
print_number(self)         # prints the int
isnegative(self)           # checks whether int is negative
isdivisibleby(self, n)     # checks whether int is divisible by n
absolute_value(self)       # returns absolute value of int
```

Program

```
class Number :
    def set_number(self, n) :
        self.__num = n

    def get_number(self) :
        return self.__num

    def print_number(self) :
        print(self.__num)

    def isnegative(self) :
        if self.__num < 0 :
            return True
        else :
            return False ;

    def isdivisibleby(self, n) :
        if n == 0 :
            return False
        if self.__num % n == 0 :
            return True
        else :
            return False
    def absolute_value(self) :
        if self.__num >= 0 :
            return self.__num
        else :
            return -1 * self.__num

x = Number( )
x.set_number(-1234)
x.print_number( ) ;
if x.isdivisibleby(5) == True :
    print("5 divides ", x.get_number( ))
else :
    print("5 does not divide ", x.get_number( ))
print("Absolute Value of ", x.get_number( ), " is ", x.absolute_value( ))
```

Output

```
-1234
5 does not divide  -1234
Absolute Value of  -1234  is  1234
```

Problem 18.2

Write a program to create a class called Fruit with attributes size and color. Create multiple objects of this class. Report how many objects have been created from the class.

Program

```
class Fruit :
    count = 0

    def __init__(self, name = ' ', size = 0, color = ' ') :
        self.__name = name
        self.__size = size
        self.__color = color
        Fruit.count += 1

    def display( ) :
        print(Fruit.count)

f1 = Fruit('Banana', 5, 'Yellow')
f2 = Fruit('Orange', 4, 'Orange')
```

```
f3 = Fruit('Apple', 3, 'Red')
Fruit.display( )
print(Fruit.count)
```

Output

```
3
3
```

Tips

- **count** is a class attribute, not an object attribute. So it is shared amongst all **Fruit** objects.
- It can be initialized as **count = 0**, but must be accessed using **Fruit.count**.

Problem 18.3

Write a program that determines whether two objects are of same type, whether their attributes are same and whether they are pointing to same object.

Program

```
class Complex :
    def __init__(self, r = 0.0, i = 0.0) :
        self.__real = r
        self.__imag = i

    def __eq__(self, other) :
        if self.__real == other.__real and self.__imag == other.__imag :
            return True
        else :
            return False

c1 = Complex(1.1, 0.2)
c2 = Complex(2.1, 0.4)
c3 = c1
if c1 == c2 :
    print('Attributes of c1 and c2 are same')
else :
    print('Attributes of c1 and c2 are different')

if type(c1) == type(c3) :
    print('c1 and c3 are of same type')
else :
    print('c1 and c3 are of different type' )

if c1 is c3 :
    print('c1 and c3 are pointing to same object')
else :
    print('c1 and c3 are pointing to different objects' )
```

Output

```
Attributes of c1 and c2 are different
c1 and c3 are of same type
c1 and c3 are pointing to same object
```

Tips

- To compare attributes of two **Complex** objects we have overloaded the **==** operator, by defining the function **__eq__( )**. Operator overloading is explained in detail in Chapter 19.
- **type( ) is** used to obtain the type of an object. Types can be compared using the **==** operator.
- **is** keyword is used to check whether **c1** and **c3** are pointing to the same object.

Problem 18.4

Write a program to get a list of built-in functions.

Program

```
import builtins
print(dir(builtins))
print( )
print(vars(builtins))
```

Output

```
['ArithmeticError', 'AssertionError', 'AttributeError', ...
'__debug__', '__doc__', '__import__', '__loader__', '__name__', ...
'abs', 'all', 'any', 'ascii', 'bin', 'bool', 'bytearray', 'bytes', 'callable', 'chr', ...
'sum', 'super', 'tuple', 'type', 'vars', 'zip']

{'__name__': 'builtins', '__package__': '', '__loader__': <class
'_frozen_importlib.BuiltinImporter'>, 'abs': <built-in function abs>,
'all': <built-in function all>, 'any': <built-in function any>, ... 'False': False}
```

Tips

- In the output above only partial items of dictionary and list is being displayed. The actual output is much more exhaustive.

Problem 18.5

Suppose we have defined two functions **msg1( )** and **msg2( )** in main module. What will be the output of **vars( )** and **dir( )** on the current module? How will you obtain the list of names which are present in both outputs, those which are unique to either list?

Program

```
def msg1( ) :
    print('Wright Brothers are responsible for 9/11 too')

def msg2( ) :
    print('Cells divide to multiply')

d = vars( )
l = dir( )
print(sorted(d.keys()))
print(l)
print(d.keys( ) - l)
print(l - d.keys( ))
```

Output

```
['__annotations__', '__builtins__', '__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__spec__', 'd', 'l', 'msg1',
'msg2']
['__annotations__', '__builtins__', '__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__spec__', 'd', 'msg1', 'msg2']
{'l'}
set( )
```

Tips

- **set( )** shown in the output means an empty set. It means there is nothing in **l** that is not present in **d**.

Problem 18.6

Is there any difference in the values returned by the functions **dir( )** and **vars(..).keys( )**? If yes, write a program to obtain that difference?

Program

```
s = set(dir(list)).difference(vars(list).keys( ))
print(s)
```

Output

```
{'__class__', '__setattr__', '__format__', '__init_subclass__',
'__subclasshook__', '__delattr__', '__dir__', '__reduce__',
'__reduce_ex__', '__str__'}
```

Tips

- **dir(list)** will return a list of attributes of **list** type.
- **vars(list).keys( )** returns a list of keys from the dictionary returned by **vars( )** for the **list** type.
- **differernce( )** returns the difference between the two lists.

Exercises

[A] State whether the following statements are True or False:

(a) Class attributes and object attributes are same.

(b) A class data member is useful when all objects of the same class must share a common item of information.

(c) If a class has a data member and three objects are created from this class, then each object would have its own data member.

(d) A class can have class data as well as class methods.

(e) Usually data in a class is kept private and the data is accessed / manipulated through object methods of the class.

(f) Member functions of an object have to be called explicitly, whereas, the **__init__( )** method gets called automatically.

(g) A constructor gets called whenever an object gets instantiated.

(h) The **__init__( )** method never returns a value.

(i) When an object goes out of scope, its **__del__( )** method gets called automatically.

(j) The **self** variable always contains the address of the object using which the method/data is being accessed.

(k) The **self** variable can be used even outside the class.

(l) The **__init__( )** method gets called only once during the lifetime of an object.

(m) By default, instance data and methods in a class are public.

(n) In a class two constructors can coexist—a 0-argument constructor and a 2-argument constructor.

[B] Answer the following questions:

(a) Which methods in a class act as constructor?

(b) How many object are created in the following code snippet?

```
a = 10
b = a
c = b
```

(c) What is the difference between variables, **age** and **__age**?

(d) What is the difference between the function **vars( )** and **dir( )**?

(e) In the following code snippet what is the difference between **display( )** and **show( )**?

```
class Message :
    def display(self, msg) :
        pass
    def show(msg) :
        pass
```

(f) In the following code snippet what is the difference between **display( )** and **show( )**?

```
m = Message( )
m.display('Hi and Bye' )
Message.show('Hi and Bye' )
```

(g) How many parameters are being passed to **display( )** in the following code snippet:

```
m = Sample( )
m.display(10, 20, 30)
```

[C] Attempt the following questions:

(a) Write a program to create a class that represents Complex numbers containing real and imaginary parts and then use it to perform complex number addition, subtraction, multiplication and division.

(b) Write a program that implements a **Matrix** class and performs addition, multiplication, and transpose operations on 3 x 3 matrices.

(c) Write a program to create a class that can calculate the surface area and volume of a solid. The class should also have a provision to accept the data relevant to the solid.

(d) Write a program to create a class that can calculate the perimeter / circumference and area of a regular shape. The class should also have a provision to accept the data relevant to the shape.

(e) Write a program that creates and uses a **Time** class to perform various time arithmetic operations.

(f) Write a program to implement a linked list data structure by creating a linked list class. Each node in the linked list should contain name of the car, its price and a link to the next node.

[D] Match the following pairs:

a. dir( )	1. Nested packages
b. vars( )	2. Identifiers, their type & scope
c. Variables in a function	3. Returns dictionary
d. import a.b.c	4. Local namespace
e. Symbol table	5. Returns list
f. Variables outside all functions	6. Global namespace