19 Intricacies of Classes & Objects

"It's the detail that matters..."

Contents

KanNotes

Identifier Naming Convention

- We have created identifiers for many things—normal variables, functions, classes, instance data, instance methods, class data and class methods.
- It is a good idea to follow the following convention while creating identifiers:

  (a) All variables and functions not belonging to a class - Start with a lowercase alphabet.
  Example: real, imag, name, age, salary, printit( ), display( )

  (b) Variables which are to be used and discarded - Use _.
  Ex: for _ in [10, 20, 30, 40] : print(_)

  (c) Class names - Start with an uppercase alphabet.
  Example: Employee, Fruit, Bird, Complex, Tool, Machine

  (d) Private identifiers, i.e. identifiers which we want should be accessed only from within the class in which they are declared - Start with two leading underscores.
  Example: __name, __age, __get_errors( )

  (e) Protected identifiers, i.e. identifiers which we want should be accessed only from within the class in which they are declared or from the classes that are derived from the class using a concept called inheritance (discussed in Chapter 20) - Start with one leading underscore.
  Example: _address, _maintain_height( )

  (f) Public identifiers, i.e. identifiers which we want should be accessed only from within the class or from outside it - Start with a lowercase alphabet.
  Example: neighbour, displayheight( )

  (g) Language-defined special names - Start and end with two __.
  Example: __init__( ), __del__( ), __add__( ), __sub__( )

  Don't call these methods. They are the methods that Python calls.

(h) Unlike C++ and Java, Python does not have keywords private, protected or public to mark the attributes. So if above conventions are followed diligently, the identifier name itself can convey how you wish it to be accessed.

Calling Functions and Methods

- Consider the program given below. It contains a global function **printit( )** which does not belong to any class, an instance method called **display( )** and a class method called **show( )**.

```
def printit( ) :                        # global function
    print('Opener')

class Message :
    def display(self, msg) :            # instance method
        printit( )
        print(msg)

    def show( ) :                       # class method
        printit( )
        print('Hello')
        # display( )                    # this call will result in an error

printit( )                              # call global function
m = Message( )
m.display('Good Morning')               # call instance method
Message.show( )                         # call class method
```

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

```
Opener
Opener
Good Morning
Opener
Hello
```

- Class method **show( )** does not receive **self**, whereas instance method **display( )** does.
- A global function **printit( )** can call a class method **show( )** and instance method **display( )**.

- A class method and instance method can call a global function **printit( )**.
- A class method **show( )** cannot call an instance method **display( )** since **show( )** doesn't receive a **self** argument. In absence of this argument **display( )** will not know which object is it supposed to work with.
- A class method and instance method can also be called from a method of another class. The syntax for doing so remains same:

```
m2 = Message( )
m2.display('Good Afternoon')
Message.show('Hi')
```

Operator Overloading

- Since **Complex** is a user-defined class, Python doesn't know how to add objects of this class. We can teach it how to do it, by overloading the + operator as shown below.

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

    def __add__(self, other) :
        z = Complex( )
        z.__real = self.__real + other.__real
        z.__imag = self.__imag + other.__imag
        return z

    def __sub__(self, other) :
        z = Complex( )
        z.__real = self.__real - other.__real
        z.__imag = self.__imag - other.__imag
        return z

    def display(self) :
        print(self.__real, self.__imag)

c1 = Complex(1.1, 0.2)
c2 = Complex(1.1, 0.2)
c3 = c1 + c2
c3.display( )
```

```
c4 = c1 - c2
c4.display( )
```

- To overload the + operator we need to define **__add__( )** function within the **Complex** class.
- Likewise, to overload the - operator we need to define **__sub__( )** function for carrying out subtraction of two **Complex** objects.
- In the expression **c3 = c1 + c2**, **c1** becomes available in **self**, whereas, **c2** is collected in **other**.

Which Operators to Overload?

- Given below is the list of operators that we can overload and their function equivalents that we need to define.

```
# Arithmetic operators
+       __add__(self, other)
-       __sub__(self, other)
*       __mul__(self, other)
/       __truediv__(self, other)
%       __mod__(self, other)
**      __pow__(self, other)
//      __floordiv__(self, other)

# Comparison operators
<       __lt__(self, other)
>       __gt__(self, other)
<=      __le__(self, other)
>=      __ge__(self, other)
==      __eq__(self, other)
!=      __ne__(self, other)

# Compound Assignment operators
=       __isub__(self, other)
+=      __iadd__(self, other)
*=      __imul__(self, other)
/=      __idiv__(self, other)
//=     __ifloordiv__(self, other)
%=      __imod__(self, other)
**=     __ipow__(self, other)
```

- Unlike many other languages like C++, Java, etc., Python does not support function overloading. It means function names in a program, or method names within a class should be unique. If we define two functions or methods by same name we won't get an error message, but the latest version would prevail.

Everything is an Object

- In python every entity is an object. This includes int, float, bool, complex, string, list, tuple, set, dictionary, function, class, method and module.
- When we say **x = 20**, a nameless object of type **int** is created containing a value 20 and address (location in memory) of the object is stored in **x**. **x** is called a reference to the **int** object.
- Same object can have multiple references.

```
i = 20
j = i          # another reference for same int object referred to by i
k = i          # yet another reference for same object
k = 30
print (k)      # will print 30, as k now points to a new int object
print (i, j)   # will print 20 20 as i, j continue to refer to old object
```

- In the following code snippet **x** and **y** are referring to same object. Changing one doesn't change the other. Same behavior is shown for **float, complex, bool** and **str** types.

```
x = 20
y = 20         # x and y point to same object
x = 30         # x now points to a new object
```

- In the following code snippet **x** and **y** are referring to different objects. Same behavior is shown for list, tuple, set, dictionary, etc.

```
x = Sample(10, 20)
y = Sample(10, 20)
```

- Some objects are mutable, some are not. Also, all objects have some attributes and methods.

- The **type( )** function returns type of the object, whereas **id( )** function returns location of the object in memory.

```
import math
class Message :
    def display(self, msg):
        print(msg)

def fun( ) :
    print('Everything is an object')

i = 45
a = 3.14
c = 3 + 2j
city = 'Nagpur'
lst = [10, 20, 30]
tup = (10, 20, 30, 40)
s = {'a', 'e', 'i', 'o', 'u'}
d = {'Ajay' : 30, 'Vijay' : 35, 'Sujay' : 36}

print(type(i), id(i))
print(type(a), id(a))
print(type(c), id(c))
print(type(city), id(city))
print(type(lst), id(lst))
print(type(tup), id(tup))
print(type(s), id(s))
print(type(d), id(d))
print(type(fun), id(fun))
print(type(Message), id(Message))
print(type(math), id(math))
```

On execution of this program we get the following output:

```
<class 'int'> 495245808
<class 'float'> 25154336
<class 'complex'> 25083752
<class 'str'> 25343392
<class 'list'> 25360544
<class 'tuple'> 25317808
<class 'set'> 20645208
<class 'dict'> 4969744
```

```
<class 'function'> 3224536
<class 'type'> 25347040
<class 'module'> 25352448
```

Imitating a Structure

- In C if we wish to keep dissimilar but related data together we create a structure to do so.
- In Python too, we can do this by creating a class that is merely a collection of attributes (and not methods).
- Moreover, unlike C++ and Java, Python permits us to add/delete/ modify these attributes to a class/object dynamically.
- In the following program we have added 4 attributes, modified two attributes and deleted one attribute, all on the fly, i.e. after creation of **Bird** object.

```
class Bird :
    pass

b = Bird( )

# create attributes dynamically
b.name = 'Sparrow'
b.weight = 500
b.color = 'light brown'
b.animaltype = 'Vertebrate'

# modify attributes dynamically
b.weight = 450
b.color = 'brown'

# delete attributes dynamically
del b.animaltype
```

Type Conversion

- There are two types of conversions that we may wish to perform. These are:
  (a) Conversion between different built-in types
  (b) Conversion between different built-in types and container types
  (c) Conversion between built-in and user-defined types

- We are already aware of first two types of conversions, some examples of which are given below:

```
a = float(25)              # built-in to built-in conversion
b = tuple([10, 20, 30])    # container to container conversion
c = list('Hello')          # built-in to container conversion
d = str([10, 20, 30])      # container to built-in conversion
```

- Conversion between built-in and user-defined types:

  Following program illustrates how a user-defined **String** type can be converted to built-in type **int**. **__int__( )** has been overloaded to carry out conversion from **str** to **int**.

```
class String :
    def __init__(self, s = '') :
        self.__str = s

    def display(self) :
        print(self.__str)

    def __int__(self) :
        return int( self.__str )

s1 = String(123)      # conversion from int to String
s1.display( )
i = int(s1)           # conversion from string to int
print(i)
```

P</> Programs

Problem 19.1

Write a Python program that displays the attributes of integer, float and function objects. Also show how these attributes can be used.

Program

```
def fun( ) :
    print('Everything is an object')

print(dir(55))
print(dir(-5.67))
```

```
print(dir(fun))
print((5).__add__(6))
print((-5.67).__abs__( ))
d = globals( )
d['fun'].__call__( )        # calls fun( )
```

Output

```
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', ...]
['__abs__', '__add__', '__bool__', '__class__', '__delattr__', ...]
['__annotations__', '__call__', '__class__', '__closure__', ... ]
11
5.67
Everything is an object
```

Tips

- Output shows incomplete list of attributes of **int**, **float** and **function** objects.
- From this list we have used the attributes **__add__( )** to add two integers, **__abs__( )** to get absolute value of float and **__call__( )** to call the function **fun( )**.
- **globals( )** return a dictionary representing the current global symbol table. From this dictionary we have picked the object representing the **fun** function and used it to call **__call__( )**. This results into call to **fun( )**.

Problem 19.2

Create a class **Date** that has a list containing day, month and year attributes. Define an overloaded **==** operator to compare two **Date** objects.

Program

```
class Date :
    def __init__(self, d, m, y) :
        self.__day, self.__mth, self.__yr = d, m, y

    def __eq__(self, other) :
```

```
        if self.__day == other.__day and self.__mth == other.__mth and
            self.__yr == other.__yr :
            return True
        else :
            return False

d1 = Date(17, 11, 98)
d2 = Date(17, 11, 98)
d3 = Date(19, 10, 92)
print(id(d1))
print(id(d2))
print(d1 == d3)
```

Output

```
44586224
44586256
False
```

Tips

- ids of the two objects referred by **d1** and **d2** are different. This means that they are referring to two different objects.
- To overload the == operator in the **Date** class, we need to define the function **__eq__( )**.

Problem 19.3

Create a class **Weather** that has a list containing weather parameters. Define an overloaded **in** operator that checks whether an item is present in the list.

Program

```
class Weather :
    def __init__(self) :
        self.__params = [ 'Temp', 'Rel Hum', 'Cloud Cover', 'Wind Vel']
    def __contains__(self, p) :
        return True if p in self.__params else False

w = Weather( )
```

```
if 'Rel Hum' in w :
    print('Valid weather parameter')
else :
    print('Invalid weather parameter')
```

Output

```
Valid weather parameter
```

Tips

- To overload the **in** operator we need to define the function **__contains__( )**.

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

(a) A global function can call a class method as well as an instance method.

(b) In Python a function, class, method and module are treated as objects.

(c) Given an object, it is possible to determine its type and address.

(d) It is possible to delete attributes of an object during execution of the program.

(e) Arithmetic operators, Comparison operators and Compound assignment operators can be overloaded in Python.

(f) The + operator has been overloaded in the classes **str**, **list** and **int**.

[B] Answer the following questions:

(a) Which functions should be defined to overload the +, -, / and // operators?

(b) How many objects are created by **lst = [10, 10, 10, 30]**?