Object-Oriented Python I

OA Fakinlede

Theory of OOP

- Object-Oriented Programming is based on three major Pillars:
  1. Data Abstraction
     Type Formation
     Abstract Data Typing in such a way that the data and behavior are tightly bound into a single unit called the class from which objects are instantiated.
  2. Inheritance
     Code reuse without altering the original code: Adding extra functionality while preserving the original.
  3. Polymorphism.
     The behavior of class hierarchies allowing related classes to live like a family

Teaching for Skill

- While the theory is taught, the only way you will be examined is a demonstration of the theory in working programs.
- It matters little to me that you know these concepts. ALL I care about is that the concepts come out in the WORKING programs that you develop.
- In fact, each concept will be demonstrated with programs that work. Understand the code, write your own and make them work, then you understand the concept. Anything short of that is a waste of time.

Only the programming SKILLS WILL BE TESTED

- A students that intends to succeed in this course, by now shall have done the following:
  1. Copied, executed EVERY example code that was given in this course. That is the first line of duty. Asking me any question when you have not already done that is foolishness. You will find this out sooner or later. THIS COURSE is a Programming Skills course: No more, no less.
  2. Tested EVERY principle with simple code fragments taken through a debugger and understand how the Python interpreter treats your code.
  3. Extended the given code to do other things.

Today’s class continues the same trend

- You cannot hope to do well in this course by reading through the night before the exam.
- You will write a program as your final exam! You have lost 40% of the grade if your program does not run.
- The first thing we do is to run the code you submit. When that does not run, you are fighting for 60 % of the marks!
- Do yourself a favor: Practice! Practice!! Practice!!!

Functions, Methods & Operators

In this course, we have assumed the elementary concepts of functions and operators.

Is there any difference between a function and an operator? Are they the same?

If they are different, how are they so?

- Can we replace operations by functions?
- Is the converse possible?
- How do we decide if we should perform and operation or a function?

The Plus Function

```
1  def plus(a, b, c=None, d=None):
2      if c and d:
3          return a+b+c+d
4      elif c:
5          return a+b+c
6      else:
7          return a+b
```

- Consider the function defined above. There are four arguments, two of which are optional.
- If you want arguments of a function to be optional, you let the default values be the last ones as we have done.
- With this function, we may add two, three or four numbers without using the addition operation symbol and obtain the results we expect.

The Plus Function

- The above function can be tested with the codes on this page.
  - It is easy to see that the function can be called with a variable number of arguments.
  - The correct expected additions are performed, and we do not need to be constrained to seeing addition as a binary operation, it can be a multi-argument function as we have shown.

```
a = 4
b = 3
c = -3
d = -4
f = plus(a,b)
h = plus(a,b,c,d)
pass
```

```
a = 4.2
b = 3.5
c = -3.5
d = -4.2
f = plus(a,b)
h = plus(a,b,c,d)
pass
```

No Operation Symbols

- Even though we wrapped the plus operation with a functional symbolism in the above, there is still a point at which that symbol came up.
  - Can we completely do away with the plus operator and still perform addition? If we can do so, we have proved for a fact that the semantics of the plus function and that of the addition operation remain the same despite the difference in syntax.
- Syntactically, addition operation is, at most, binary. The plus function can take as many arguments as we want.
  - Here is the fundamental difference between the operator and the function: Syntax of presentation.

No Operation Symbols

```
1  import operator
2
3  def plus(a, b, c=None, d=None):
4      if c and d:
5          return operator.add(operator.add(a,b),operator.add(c,d))
6      elif c:
7          return operator.add(operator.add(a,b),c)
8      else:
9          return operator.add(a,b)
```

- Python provides us with its own binary addition function with no usage of the addition operation as shown.

Homework

- Create and test a functional equivalent of the multiplication operation that is capable of up to five arguments.

Functions & Methods

- Consider the function here:
  - The function count(num) is a method because its scope of definition is in the class defining the list object to which listIn is simply an instance object.
  - Such functions are usually called with as an augmented form of the object name. The full calling syntax is shown above it.
- The abbreviated form is a type of syntactic sugar

```
def listDict(listIn):
    cDict = {}
    for num in set(listIn):
        cDict[num] = listIn.count(num)
    return cDict
```

```
cDict[num] = list.count(listIn, num)
cDict[num] = listIn.count(num)
```

Functions & Methods

- Functions that are called directly with objects as arguments, not being a member of the creating class are ordinary functions.
  - We have seen many examples of this already range(), len(), etc. are not methods.

Types & Classes

- We have seen so far that every built-in object in Python is intrinsically linked to a type.
- The scalar types of int, float, etc. have been seen. We also have collections. In each case, the type determines the kinds of operations and functions that are permissible.
- These are created by defining methods within the namespace of the class to which the object is an instance of.
- The help system of Python can give you information on the methods attached to a particular class or object

Overloading Operators

- If a and b are integers representing 2 and 5 respectively’
- Let c and d be strings representing “Ade” and “Farouk” respectively
- What meaning will be given to a+b?
- What about c+d?
- Overloading is a complicated way of simply saying that meanings depend on the context of usage.

Simple Class

- We will create a student class with a few simple things that may be important for that class:
  - Name
  - Matriculation identity
  - Grades in a few courses (six in this example)
- These pieces of data constitute the internal state of the class. It will vary from student object to student object.
- Clearly, the student class is the blueprint from which each student is made. Technically, each student is an instance of this class.
  - Making a specific student is to “instantiate” the student class.

Background

- We have done this before:
  - We have created lists, dicts, tuples etc., by instantiating from these classes. We did so by calls to the constructor.
  - In statically typed languages such as Java, C# or C++, we would need to create the constructors when we are creating the class.
  - Python is different. The constructor is created for you. Instead, you will need to create the initializer.
  - This procedure can be confusing to those who have some experience with those languages. Be assured, what Python does is much simpler.

The “generic” Student

- The initialization function MUST be named __init__
- Its first argument is “self”
- Other variables that create the internal state for the instance – that makes that instance unique are listed after that.
- In the three statements that define the initializer, we have assigned these to the internal values of the object which are denoted by the leading underscore.

```
1  class Student:
2      def __init__(self, name, matric, grades):
3          self._matric = matric
4          self._name = name
5          self._grades = grades
6      def name(self):
7          return self._name
8      def matric(self):
9          return self._matric
10     def grades(self):
11         return self._grades
12     def avg(self):
13         sum = 0
14         for num in self._grades:
15             sum+=num
16         return sum/len(self._grades)
```

The Student

- You can create a student now by passing a name, a matric and a list of grades that will be unique to the particular student.
- The initializer uses these values to create the internal state of the specific student object so created.
- Next three functions (methods) allow access from outside the class to the identifiers in this internal state.

```
1   class Student:
2       def __init__(self, name, matric, grades):
3           self._matric = matric
4           self._name = name
5           self._grades = grades
6       def name(self):
7           return self._name
8       def matric(self):
9           return self._matric
10      def grades(self):
11          return self._grades
12      def avg(self):
13          sum = 0
14          for num in self._grades:
15              sum+=num
16          return sum/len(self._grades)
```

The Student

- A fourth method, after the initializer, is here finding the student's average in the courses listed.

```
1   class Student:
2       def __init__(self, name, matric, grades):
3           self._matric = matric
4           self._name = name
5           self._grades = grades
6       def name(self):
7           return self._name
8       def matric(self):
9           return self._matric
10      def grades(self):
11          return self._grades
12      def avg(self):
13          sum = 0
14          for num in self._grades:
15              sum+=num
16          return sum/len(self._grades)
```

Instantiating the Student Class

- In each instance here, we have two strings: one for the name, the other for the matriculation identity; and one list of marks for each student.
- We created four instances in a list of students.
  - We could have created them one by one if we chose to. Simply calling the constructor activates the initializer and the internal states of these four instances are defined.

```
18  g1 = [50,56,90,92,93,45]
19  g2 = [20,30,46,90,80,76]
20  g3 = [40,87,67,34,35,13]
21  g4 = [32,45,65,45,87,29]
22  n1 = "Asubiaro"
23  n2 = "Maroko"
24  n3 = "Owambe"
25  n4 = "Miofe"
26  m1 = "SSG123"
27  m2 = "MEC105"
28  m3 = "ELC005"
29  m4 = "IOT001"
30  stdLst = [Student(n1,m1,g1),
31      Student(n2,m2,g2),
32      Student(n3,m3,g3),
33      Student(n4,m4,g4)]
```

Working with the Student Object

- As we have previously remarked, each student object has its own internal state.
  - It can now be used, as we have seen as any other Python object.
  - We can create lists, tuples, dictionaries and sets of students.
  - We can also define things to do with students.
- For example, can we add two students?
  - What meaning will be attributed to such a function? Could we, for example concatenate the manes and add the grades?
  - This could be a prelude to getting the average of the two students if it is sensible.

The Student Object

```
34   namesLst = {}
35 ⌄ for std in stdLst:
36       namesLst[std.avg()] = std.name()
37 ⌄ for keys in sorted(namesLst.keys()):
38       print(keys, namesLst[keys])
```

- In the above code, we created an empty dictionary and used each student's average score as the key to the student's name.
- Note that there are several possibilities here.
  - The key here is a call to the avg() function. The way it is called here is a shorthand. The full call, rarely used is: Student.avg(std) while std.avg(), the shorthand form is called a syntactic sugar!
  - It is important to bear this in mind because when a method is called for student, we do not need to remind it of the marks scored that require averaging! These are already known for each properly instantiated student object!
  - Remember, the initializer would have been called for each at instatiation!

Homework

- Create and test your own student class
  - At least 15 courses with different weights
  - Find each student GPA
  - Rank students by their GPA
  - Mark students automatically if they are not in good standing
  - Print a list with GPA
- Due Monday Feb 7.
- 10 marks