Introduction to Object-Oriented Programming in Python

WEEK 1

Instructor: Ph.D. Oleg Tymchuk

Overview

Classes and Objects

Encapsulation

Inheritance

Polymorphism

Operator Overloading

Objects and Classes

- Class can be defined as a template/blueprint that describes the behavior/state that the object of its type support

- The form of class definition:
  class ClassName[(superclass)]:
  [attributes and methods]

- Example:

```
class MyClass:
    """A simple example class"""
    i = 12345

    def f(self):
        return 'hello world'
```

- Objects are instantiations of the class (objects have states and behaviors)

- Object instantiation syntax:
  object = ClassName()

- Attributes and methods invoke:
  object.attribute
  object.method()

- Example

```
>>> x = MyClass()
>>> print(x.i, x.f(), sep='\n')
12345
hello world
```

Objects and Classes. Example

```
# Class definition.
class Employee:

    # Class attribute.
    company = 'CCC'

    # Constructor.
    def __init__(self, name, surname, age):
        # Instance attributes
        self.name = name
        self.surname = surname
        self.age = age

    # Instance method
    def msg(self):
        return self.company + '; ' +\
               'employee: ' +\
               self.name + ' ' + self.surname
```

```
>>> e1 = Employee('Ivan', 'Wolf', '23')
>>> e2 = Employee('Karl', 'Jonson', '25')
>>> print(e1.msg(), e2.msg())
```

CCC; employee: Ivan Wolf CCC; employee: Karl Jonson

```
>>> Employee.company = 'BBB'
>>> print(e1.msg(), e2.msg())
```

BBB; employee: Ivan Wolf BBB; employee: Karl Jonson

Objects and Classes. Built-In Class Attributes

Attribute	Description
__dict__	Dictionary containing the class's namespace
__doc__	The class's documentation string, or None if unavailable
__name__	The class's name
__module__	The name of the module the class was defined in, or None if unavailable
__bases__	A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list

```
>>> print(e1.__dict__)
{'name': 'Ivan', 'surname': 'Wolf', 'age': '23'}

>>> print(e1.__doc__)
None

>>> print(Employee.__name__)
Employee
```

```
>>> print(e1.__module__)
__main__

>>> print(Employee.__bases__)
(<class 'object'>,)
```

Encapsulation

- Encapsulation is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit
- In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class (data hiding)
- Implementing Public/Protected/Private Interfaces

Name	Notation	Behavior
name	Public	Accessible from anywhere. Python attributes and methods are public by default
_name	Protected	Like a public member, but it shouldn't be directly access from outside
__name	Private	Accessible only in their own class

Encapsulation. Getters/Setters/Deleters

- Access to protected/private attributes is achieved through property attributes
- class property(fget=None, fset=None, fdel=None, doc=None) → return a property attribute
- A property object has getter, setter, and deleter methods usable as decorators

```
class C:
    def __init__(self):
        self.__x = None

    def getx(self):
        return self.__x
    def setx(self, value):
        self.__x = value
    def delx(self):
        del self.__x

    x = property(getx, setx, delx, "'x' property.")
```

```
class C:
    def __init__(self):
        self.__x = None

    @property
    def x(self):
        """'x' property."""
        return self.__x
    @x.setter
    def x(self, value):
        self.__x = value
    @x.deleter
    def x(self):
        del self.__x
```

Inheritance

- Inheritance is a form of software reuse in which you create a class that absorbs an existing class's data and behaviors and enhances them with new capabilities.

- The existing class is called the base class (superclass), and the new class is referred to as the derived class (subclass).

- The syntax for a derived class definition looks like this

```
class SubclassName(SuperclassName):
        <statement-1>
          .
          .
          .
        <statement-N>
```

Inheritance. Example

```
class Person:

    def speak(self):
        return 'I can speak'

class Man(Person):

    def wear(self):
        return 'I wear shirt'


class Woman(Person):

    def wear(self):
        return 'I wear skirt'

    def speak(self):
        return 'I can speak a lot'
```

```
>>> man = Man()
>>> print(man.speak(), man.wear(), sep='\n')

I can speak
I wear shirt

>>> woman = Woman()
>>> print(woman.speak(), woman.wear(), sep='\n')

I can speak a lot
I wear skirt
```

Multiple Inheritance

- Python supports a form of multiple inheritance as well. A class definition with multiple base classes looks like this:

```
class SubclassName(Superclass1, Superclass2, Superclass3):
        <statement-1>
          .
          .
          .
        <statement-N>
```

- Method Resolution Order: depth-first, left-to-right.

Thus, if an attribute is not found in Subclass, it is searched in Superclass1, then recursively in the classes of Superclass1, and only if it is not found there, it is searched in Superclass2, and so on

Multiple Inheritance. Example

```
class X:

    def get_name(self):
        return 'X'

class Y:

    def get_name(self):
        return 'Y'

class A(X, Y):

    def who_am_i(self):
        return 'I am a A'

class B(Y, X):

    def who_am_i(self):
        return 'I am a B'
```

```
>>> child_1 = A()
>>> print(child_1.who_am_i())
I am a A
>>> print(child_1.get_name())
X
>>> child_2 = B()
>>> print(child_2.who_am_i())
I am a B
>>> print(child_2.get_name())
Y
```

Inheritance. Superclass methods

- Super is used for calling superclass methods

```
class Polygon:
    def __init__(self, no_of_sides):
        self.n = no_of_sides
        self.sides = [0 for i in range(no_of_sides)]

    def input_sides(self):
        self.sides =
           [float(input('Enter side ' + str(i + 1) + ' : '))
                 for i in range(self.n)]

class Triangle(Polygon):
    def __init__(self):
        super(Triangle, self).__init__(3)

    def find_area(self):
        a, b, c = self.sides
        # calculate the semi-perimeter
        s = (a + b + c) / 2
        area = (s * (s - a) * (s - b) * (s - c)) ** 0.5
        return area
```

```
>>> t = Triangle()
>>> t.input_sides()
>>> print('The area of the triangle is %0.2f' %
          t.find_area())

Enter side 1 : 8
Enter side 2 : 6
Enter side 3 : 7
The area of the triangle is 20.33
```

Polymorphism

- Polymorphism is the capability of a method to do different things based on the object that it is acting upon.

In other words, polymorphism allows you define one interface and have multiple implementations.

```
class Animal:

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

    def talk(self):
        raise NotImplementedError()

class Cat(Animal):

    def talk(self):
        return 'Meow!'

class Dog(Animal):

    def talk(self):
        return 'Woof! Woof!'
```

```
>>> animals = [Cat('Missy'),
               Cat('Mr. Mistoffelees'),
               Dog('Lassie')]

>>> for animal in animals:
            print(animal.name + ': ' + animal.talk())

Missy: Meow!
Mr. Mistoffelees: Meow!
Lassie: Woof! Woof!
```

Operator Overloading

- A class can implement certain operations that are invoked by special syntax (such as arithmetic operations or subscripting and slicing) by defining methods with special names. This is Python's approach to operator overloading, allowing classes to define their own behavior with respect to language operators

- Examples of some special methods

OPERATOR	FUNCTION	METHOD DESCRIPTION
+	__add__(self, other)	Addition
-	__sub__(self, other)	Subtraction
==	__eq__(self, other)	Equal to
!=	__ne__(self, other)	Not equal to
[index]	__getitem__(self, index)	Index operator
in	__contains__(self, value)	Check membership
len	__len__(self)	The number of elements
str	__str__(self)	The string representation

Effective Software Design

Two simple general principles:

- KIS (Keep It Simple)

  No Overengineering, no Spaghetti code.
- DRY (Don’t Repeat Yourself)

  Code duplication equals bug reuse.

Iterative Development (Agile Development):

- one cannot anticipate every detail of a complex problem
- start simple (with something that works), then improve it
- identify emerging patterns and continuously adapt the structure of your code (refactoring, for which you want unittests)