SciComp with Py

OOP: Part 1

Vladimir Kulyukin
Department of Computer Science
Utah State University

Outline

- OOP Basics
  - Class vs Object
  - Attributes
  - Polymorphism & Encapsulation
  - Duck Typing
  - Inheritance
- Defining & Constructing Objects

OOP Basics

Class vs. Object

- A **class** is a definition (blueprint, description) of states and behaviors of objects that belong to it
- An **object** is a member of its class that behaves according to the definition (blueprint/description) of the class
- Objects of a class are also called **instances** of that class

Attributes: Data & Methods

- Class attributes in Py correspond to class members in C++/Java
- There are two types of attributes in a Py class: data and methods
- A data attribute does not have to be declared in the class definition unlike in C++/Java; it begins to exist after it is assigned a value
- A member function must be defined within the class scope
- All method attributes in Python correspond to virtual functions in C++

Polymorphism

- Polymorphism is a noun derived from two Greek words ***poly*** (many) and ***morph*** (form)
- Wiktionary (http://en.wiktionary.org) defines polymorphism as the ability to assume multiple forms or shapes
- In OOP, polymorphism refers to the use of the same operation on objects of different types/classes

Riley's Duck Test

- In technical literature, ***polymorphism*** is sometimes used synonymously with ***duck typing***
- Duck typing allegedly takes its name from the duck test attributed to James Whitcomb Riley, an American writer and poet: **“When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.”**

Riley's Duck Test

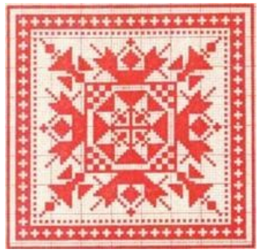
- The basic principle of duck typing can be expressed as follows: ***it is not the type of the object that matters but the operations that the object supports***
- In non-duck-typed languages:
  - the programmer defines the functions **walk** and **quack** as **walk(Duck x)** and **quack(Duck x)**
  - If **x** is not of type **Duck**, a <u>compile time error</u> is signaled
- In duck-typed languages:
  - the programmer defines the functions **walk** and **quack** as **walk(x)** and **quack(x)**
  - If **x** is not of type **Duck**, a <u>run time error</u> is signaled

Critiques of Duck Typing

- **Critique 1**: Duck typing increases the cognitive load on the programmer because the programmer cannot infer types from local code segments and therefore must always be aware of the big picture
- **Critique 2**: Duck typing makes project planning more difficult because in many cases only project managers (not software developers) need to know the big picture
- **Critique 3**: Duck typing increases/complicates software testing

Encapsulation

- Encapsulation is the principle of hiding unnecessary information from the world
- A class defines the data that its objects need
- Users of objects may not want to know most of the data
- Example: A snowflake class defines all the necessary components for the snowflake to be drawn but the end users are only interested in drawing snowflakes



Encapsulation Caveats

- Encapsulation in the sense that it exists in C++/Java via **public**, **protected**, **private** is not possible in Python
- Python is an **open design** language: if a client of a class so chooses, the client can always access data inside an object
- Bottom line: while encapsulation mechanisms exist in Python, they are not bulletproof

Polymorphism vs. Encapsulation

- Both polymorphism and encapsulation are data abstraction principles
- Polymorphism in duck type languages allows the programmer to apply methods to an object without knowing the object's type
- Encapsulation allows the programmer to manipulate objects without knowing internal details of objects

Inheritance

- Inheritance is an OOP principle that supports code reuse and abstraction
- If a class **C** defines a set of attributes (data and methods), the programmer can derive a subclass of **C** without having to re-implement **C**'s attributes
- In OOP, subclasses ***inherit*** attributes of superclasses

Multiple Superclasses

- A class in Python can have multiple superclasses: this is called ***multiple inheritance***
- Caveat: if several superclasses implement the same method, the method resolution is required
- The method resolution order is a graph search algorithm
- Cautionary advice: unless you really need multiple inheritance, you should avoid it

Defining Classes and Constructing Objects

Py Class Blueprint

```
class ClassName:
        <statement-1>
        ...
        <statement-N>
```

Classes & Namespaces

- When a class definition is evaluated, a new namespace/package is created
- All assignments of local variables occur in the new namespace
- Function definitions bind function names in the new namespace

Example

Class variable

```
class C:
    """ Example Class """
    x = 12

    def sayHi(self):
        print 'Hi, I am C!
```

Instance method;
Note self as the
1st argument!

Note that there is no argument
passed to sayHi(); What happened to self?

```
>>> C
<class __main__.C at 0x7f2444917c80>
>>> c = C()
>>> c.x
>>> assert(C.x == c.x == 12)
>>> c.sayHi()
Hi, I am a Py C object!
>>> C.x = 15
>>> assert(C.x == c.x == 15)
```

There is no new() in Py

Calling Methods on Instances

- The statement **c.sayHi()** is converted to C.sayHi(c) so that **self** is bound to **c**
- In general, suppose there is a class **C** with a method **f(self, x1, ..., xn)**
- Suppose we do:

  >>> c = C()

  >>> c.f(v1, ..., vn)
- Then **c.f(v1, ..., vn)** is the same as **C.f(c, v1, ..., vn)**

Question

```
class C3:
    """Example Class C3"""

    def sayHi(self):
        print('Hi, I am a Py C3 object!')

    def add2(self, x, y):
        return x+y

    def mult3(self, x, y, z):
        return x*y*z
```

Consider class C3 on the left. Assume that c3 is a C3 object. Let's convert the following method calls to class function calls:

```
>>> c3.sayHi()
>>> c3.add2(10, 20)
>>> c3.mult3(10, 20, 3)
```

Answer

```
>>> c3 = C3()
>>> c3.sayHi()
Hi, I am a Py C3 object!
>>> C3.sayHi(c3)
Hi, I am a Py C3 object!
>>> c3.add2(10, 12) == C3.add2(c3, 10, 12)
True
>>> c3.mult3(10, 12, 3) == C3.mult3(c3, 10, 12, 3)
True
```

Another Example

```
class C:
    def f(self, x1, x2, x3):
        return [x1, x2, x3]
>>> x = C()
>>> x.f(1, 2, 3)
[1, 2, 3]
>>> C.f(x, 1, 2, 3)      ## equivalent to x.f(1, 2, 3)
[1, 2, 3]
```

Defining Constructor

```
class C:

    def __init__(self, <other arguments>):
        ## some initialization code
        pass
```

Note that a Py class may have at most one __init__()

Example: Constructors & Instance variables

```python
class C:
    """ Example Class """
    x = 12

    def __init__(self):
        self.data = (1, 2)

    def sayHi(self):
        print 'Hi, I am C!'
```

Constructor

Instance variable

```
>>> C2
<class __main__.C2 at 0x7f89b5a5dae0>
>>> c2 = C2()
>>> c2.x
12
>>> c2.data
(1, 2)
>>> C2.data
Traceback (most recent call last):
  File "<pyshell#45>", line 1, in <module>
    C2.data
AttributeError: class C2 has no attribute 'data'
```

Data is an instance variable, not a class variable

Class vs. Object Attributes

- Class attributes of class **C** are the same for all objects of class C
- Object/instance attributes are unique for each object/instance

Accessing Instance Attributes

There are two types of attributes: class attributes & object/instance attributes

```
>>> A.x              ## class attribute reference
>>> A.g              ## class attribute reference
>>> A.__doc__        ## class attribute reference
>>> a = A()          ## a is an instance of
                     ## class A (instantiation)
>>> a.x              ## x is an attribute of object a
```

Class & Instance Attributes

A.py

```
class A:
    ## This is A.x class variable
    x = 12
    def g(self): return 'Hello from A instance'
    ## Change the value of A.x class variable
    def setClassX(self, xval):
      A.x = xval
    ## Change the value of self.x instance
    ## variable
    def setInstanceX(self, xval):
       self.x = xval
```

Tests

```
>>> a01 = A()
>>> a02 = A()
>>> a03 = A()
>>> a01.x
>>> 12
>>> a03.setClassX(50)
>>> a01.x
>>> 50
>>> a02.x
>>> 50
>>> a03.x
>>> 50
```

Create 3 objects of class A

a01.x accesses A.x, because a01.x has not been defined

A.x is set to 50

A.x is set to 50 for all objects of class A

Class & Instance Attributes

A.py

```
class A:
    ## This is A.x class variable
    x = 12
    def g(self): return 'Hello from A instance'
    ## Change the value of A.x class variable
    def setClassX(self, xval):
        A.x = xval
    ## Change the value of self.x instance
    ## variable
    def setInstanceX(self, xval):
        self.x = xval
```

Tests (after reloading A.py)

```
>>> a01 = A()
>>> a02 = A()
>>> a03 = A()
>>> a01.x
>>> 12
>>> a01.setInstanceX(20)
>>> a01.x
>>> 20
>>> a02.x
>>> 12
>>> a03.x
>>> 12
```

Create 3 objects of class A

a01.x accesses A.x, because a01.x has not been defined

set x of a01 to 20

a02.x and a03.x still refer to A.x

Customizing __init__

Comma-separate arguments

Comma-separated key-value pairs

```
class C:
    def __init__(self, *args, **kwargs):
        ## some initialization code
        pass
```

Example

Py

```
class D:
    """Example Class D"""

    def __init__(self, *args, **kwargs):
        print 'args: ', args, ' kwargs: ', kwargs
```

Output

```
>>> d = D()
args:  ()  kwargs:  {}
>>> d = D(1, 2, 3)
args:  (1, 2, 3)  kwargs:  {}
>>> d = D(first=1, second=2, third=3)
args:  ()  kwargs:  {'second': 2, 'third': 3, 'first': 1}
```

Problem

Write the class Toop that sets its data attribute to 1-, 2-, or 3-tuples constructed either from *args or **kwargs.

Solution

Let's do a code walkthrough of Toop.py

Constructor Method __**init**__**()**

- If you want to customize object construction, you need to define __**init**__**()**
- __**init**__**()** is one of the so-called magic methods
- Py does not support method overloading; magic methods such as __**init**__**()** allow programmers to customize object construction

Extended Example

The files Date.py and DateTests.py contain examples of defining and using a simple Date class

Reading & References

- https://docs.python.org/2/tutorial/classes.html
- https://docs.python.org/3.4/tutorial/classes.html
- You can read the above two links to learn more about classes in Py2 and Py3