Object Oriented Python — A Quick Review

by

Avinash Kak
kak@purdue.edu

Last Modified: June 19, 2006

- This review was created specially for the students taking ECE 495F, **Introduction to Computer Security**, to help them with writing Python scripts for solving homework problems.

- This review was excerpted from the forthcoming book “**Scripting with Objects: A Comparative Presentation of Perl and Python**” by Avinash Kak

- Avinash Kak is also the author of “**Programming with Objects: A Comparative Presentation of Object-Oriented Programming with C++ and Java**”, published by John-Wiley & Sons, New York, 2003. This book presents a new approach to the combined learning of two large object-oriented languages, C++ and Java. It is being used as a text in a number of educational programs around the world. This book has also been translated into Chinese. For further information, visit `http://programming-with-objects.com`.

The Main Concepts

The following fundamental notions of object-oriented programming in general apply to object-oriented scripting also:

- **Class**
- **Encapsulation**
- **Inheritance**
- **Polymorphism**

What is a Class

- At a high level of conceptualization, a class can be thought of as a category. We may think of “Cat” as a class.

- A specific cat would then be an **instance** of this class.

- For the purpose of writing code, a class is a data structure with **attributes**.

- An instance constructed from a class will have specific values for the attributes.

- To endow instances with behaviors, a class can be provided with **methods**.

- A **method** is a function that you **invoke on** an instance or a class.

- A method that is invoked on an instance is sometimes called an **instance method**.

- You can also invoke a method directly on a class, in which case it is called a **class method** or a **static method**.

- Attributes that take values on a per-instance basis are sometimes referred to as **instance attributes** or **instance variables**.

- You can also have attributes that take on values on a per-class basis. Such attributes are called **class attributes** or **static attributes** or **class variables**.

Encapsulation

- Hiding, or controlling the access to, the implementation-related attributes and the methods of a class is called **encapsulation**.

- With appropriate data encapsulation, a class will present a well-defined **public interface** for its clients (the users of the class).

- A client would only be able to access those data attributes and invoke those methods that are in the **public interface**.

Inheritance and Polymorphism

- **Inheritance** in object-oriented code allows a subclass to inherit some or all of the attributes and methods of its superclass(es).

- **Polymorphism** basically means that a given category of objects can exhibit multiple identities at the same time, in the sense that a `Cat` object is not only of type `Cat`, but also of types `FourLegged` and `Animal`, *all at the same time.*

- Therefore, if we were to make an array like

  ```
  animals = (kitty, fido, tabby, quacker, spot);
  ```

  of cats, dogs, and a duck — *instances made from different classes in some* `Animal` *hierarchy* — and if we were to invoke a method named `calculateIQ()` on this list of animals in the following fashion

  ```
  for item in animals[:]:
      item.calculateIQ();
  ```

  polymorphism would cause the correct implementation code for `calculateIQ()` to be automatically invoked for each of the animals.

More on the Previous Example on Polymorphism

- In many object-oriented languages, a method such as `calculateIQ()` would need to be declared for the root class `Animal` for the control loop shown above to work properly.

- All of the public methods and attributes defined for the root class would constitute the public interface of the class hierarchy and each class in the hierarchy would be free to provide its own implementation for the methods declared in the root class.

- The sort of loop we showed on the previous page would constitute manipulating the instances made from the classes in the hierarchy through the public interface defined by the root class.

- Polymorphism, in a nutshell, allows us to manipulate objects belonging to the different classes of a hierarchy through a common interface defined for a root class.

The OO Terminology used in Python

- Python literature refers to practically everything as an object since practically all entities in Python are objects in the sense of possessing either retrievable data attributes or invocable methods or both using the dot operator that is commonly used in object-oriented programming for such purposes.

- A user-defined class (for that matter, any class) in Python is an object with certain **predefined attributes**. These are not to be confused with the more traditional **programmer-supplied attributes** such as the **class and instance variables** and the **programmer-supplied methods**.

- By the same token, an **instance** constructed from a class is an object with certain other **predefined attributes** that again are not to be confused with the **programmer-supplied instance and class variables** associated with the instance and the **programmer-supplied methods** that can be invoked on the instance.

- What are commonly referred to as **data attributes** of a class in Perl are frequently called **instance variables** and **class variables** in Python. In Python, the word **attribute** is used to describe any property, variable or method, that can be invoked with the dot operator on either the class or an instances constructed from the class.

- Obviously, the **attributes** available for a class include the programmer-supplied class and instance variables and the programmer-supplied methods. This usage of **attribute** makes it all encompassing, in the sense that it now includes the predefined data attributes and methods, the programmer-supplied class and instance variables, and, of course, the programmer-supplied methods.

- Our usage of **method** remains the same as before; these are functions that can be called on an object using the object-oriented call syntax that for Python is of the form `obj.method()`, where `obj` may either be an instance of a class or the class itself.

- Therefore the predefined functions that can be invoked on either the class itself or on class instances are also referred to as methods.

- The predefined attributes, both variables and methods, employ a special naming convention: *the names begin and end with two underscores.*

- You may think of the **predefined** attributes as the **external properties** of classes and instances and the **programmer-supplied** attributes (in the form of instance and class variables and methods) as the **internal properties**.

- We should also mention that Python makes a distinction between **function objects** and **callables**. While all function objects are callable, not all callables are function objects.

- A **function object** can only be created with a `def` statement. On the other hand, a **callable** is any object that can be called like a function. For example, a class name can be called directly to yield an instance of the class. An instance object can also be called; what such a call returns depends on whether or not the underlying class provides a definition for the `__call__` method.

Defining a Class in Python

- We will present the full definition of a Python class in stages.

- We will start out with a very simple example of a class to make the reader familiar with the `__init__()` method whose role is to initialize the instance returned by a call to the constructor.

- First, here is the simplest possible definition for a class in Python:

  ```
  class SimpleClass:
      pass
  ```

  An instance of this class may be constructed by invoking its predefined system-supplied default constructor

  ```
  x = SimpleClass()
  ```

- Here is a class with a user-supplied constructor initializer in the form of `__init__()`. This method is automatically invoked to initialize the state of the instance returned by a call to `Person()`:

  ```
  #!/usr/bin/python
  #------------------    class Person    ----------------
  class Person:
      def __init__( self, a_name, an_age ):
          self.name = a_name
          self.age  = an_age
  #--------------   end of class definition   -----------
  a_person = Person( 'Zaphod', 114 )
  print a_person.name                # Zaphod
  print a_person.age                 # 114
  ```

- Being an object in its own right, every class comes equipped with the following predefined attributes:

```
__name__    :   string name of the class

__doc__     :   documentation string for the class

__bases__   :   tuple of parent classes from which the
                class is derived

__dict__    :   dictionary whose keys are the names of the
                class variables and class methods, and the
                names of the instance methods of the class
                and whose values are the corresponding
                bindings

__module__  :   module in which the class is defined
```

- And since every class instance is also an object in its own right, it also comes equipped with certain predefined attributes. We will be particularly interested in the following two:

```
__class__    :   string name of the class from which the
                instance was made

__dict__    :   dictionary whose keys are the names of the
                instance variables.
```

- It is important to realize that the namespace as represented by the dictionary `__dict__` for a class object is not the same as the namespace as represented by the dictionary `__dict__` for a class instance object.

- As an alternative to invoking `__dict__` on a class name, one can also use the global built-in function **`dirs()`**, as in

  ```
  dirs( MyClass )
  ```

  which returns a tuple of just the attribute names for the class (both directly defined for the class and inherited from a class's superclasses).

Illustrating the Values for System-Supplied Attributes

- To illustrate the values for the predefined attributes for class and instance objects, consider the following extension of the previous script:

```
#!/usr/bin/python

### ClassSysAttributes.py

#------------------    class Person    -----------------
class Person:
    'A very simple class'
    def __init__( self, a_name, an_age ):
        self.name = a_name
        self.age  = an_age

#--------------   end of class definition   ------------

a_person = Person( 'Zaphod', 114 )
print a_person.name             # Zaphod
print a_person.age              # 114
print Person.__name__           # Person
print Person.__doc__            # A very simple class
print Person.__module__         # __main__
print Person.__bases__          # ()
print Person.__dict__
                  # {'__module__': '__main__',
                  #  '__doc__': 'A very simple class',
                  #  '__init__': <function __init__ at 0x804f5a4>}

print a_person.__class__    # __main__.Person
print a_person.__dict__     # {'age': 114, 'name': 'Zaphod'}
```

- Note how Python allows for a documentation string in the second line of the class definition.

- When the thread of execution enters a **class definition** in a Python script, Python creates a new namespace that is local to the class. This is the namespace that is printed out when we query the `__dict__` attribute on the class itself.

- When the thread of execution exits the class definition, the definition of the class is bound to the name of the class in the scope enclosing the class.

- The `__class__` attribute of the **instance object** holds the name of the class from which the instance was created.

- The `__dict__` attribute for the **instance object** holds the instance attributes and their values for the specific instance in question.

Class Definition: The General Syntax

- Here is a more general definition for a Python **class**:

```
class MyClass :
   'optional documentation string'

   class_var1

   class_var2=val2

   def __init__( self, var3 = default3 ) :
       'optional documentation string'
       attribute3 = var3
       rest_of_construction_init_suite

   def some_method( self, some_parameters ) :
       'optional documentation string'
       method_suite

   ......
   ......
   ......
```

- Note the class variables, `class_var1` and `class_var2`. Such variables exist on a per class basis, meaning that they are **static**.

- A class variable can be given a default value in a class definition, as shown for `class_var2`.

- In general, the header of `__init()__` may look like

  ```
  def __init__(self, var1, var2, var3 = default3) :
  ```

  This constructor initializer could be for a class that has three instance variables, with the last default initialized as shown. The first parameter, typically named `self`, is implicitly set to the instance under construction.

- If you do not provide a class with its own `__init()__`, the system will provide the class with a default `__init()`. You override the default definition by providing your own implementation for `__init__()`.

- The syntax for a user-defined method for a class is the same as for stand-alone Python functions, except for the special significance accorded the first parameter, typically named `self`. It is meant to be bound implicitly to a reference to the instance on which the method is invoked.

New Style Versus Classic Classes in Python

- Python 2.2 introduced **new style classes** while retaining the old classes for backward compatibility.

- The old style classes are now referred to as the **classic classes**.

- The basic motivation for the new style classes was to allow subclassing of the built-in classes. It was not previously possible to extend, say, the string class `str` to create a more customized string class. But now you can do that with ease.

- All new style classes are subclassed, either directly or indirectly, from the root class **object**.

- The **object** class defines a set of methods with default implementations that are inherited by all classes derived from **object**.

- A case in point is the `__getattribute__()` method that gets invoked whenever a class method is invoked. Its implementation in the `object` class is a do-nothing implementation. A class that inherits from `object` can provide an override implementation for `__getattribute__()` if something special needs to be done because a method was invoked.

What does a New Style Class get from the Root Class object

- A list of attributes defined for the `object` class can be seen by printing out the list returned by the built-in `dir()` function:

```
    print dir( object )
```

  This call returns

```
    ['__class__', '__delattr__', '__doc__', '__getattribute__',  \
     '__hash__', '__init__', '__new__', '__reduce__',            \
     '__reduce_ex__', '__repr__', '__setattr__', '__str__']
```

- We can also examine the attribute list available for the `object` class by printing out the contents of its `__dict__` attribute by

```
        print object.__dict__
```

  which returns

```
  {'__setattr__': <slot wrapper '__setattr__' of 'object' objects>,
   '__reduce_ex__': <method '__reduce_ex__' of 'object' objects>,
   '__new__': <built-in method __new__ of type object at 0x400fcda0>,
   '__reduce__': <method '__reduce__' of 'object' objects>,
   '__str__': <slot wrapper '__str__' of 'object' objects>,
   '__getattribute__': <slot wrapper '__getattribute__' of 'object' object
   '__class__': <attribute '__class__' of 'object' objects>,
   '__delattr__': <slot wrapper '__delattr__' of 'object' objects>,
   '__repr__': <slot wrapper '__repr__' of 'object' objects>,
   '__hash__': <slot wrapper '__hash__' of 'object' objects>,
   '__doc__': 'The most base type',
   '__init__': <slot wrapper '__init__' of 'object' objects>
  }
```

- Whereas `object.__dict__` returns all the attribute names and their bindings for the object class, the call to `dir( object )` returns a list of just the attribute names.

- Ordinarily, `dir()` called on a class shows not only the attribute names in the class's `__dict__` dictionary but also the attribute names inherited by the argument class from all its parent classes.

How Python Creates an Instance from a New Style Class

Python uses the following two-step procedure for constructing an instance from a **new-style** class:

- **STEP 1:** The call to the constructor creates what may be referred to as a generic instance from the class.

- The generic instance's memory allocation is customized with the code in the method `__new__()` of the class. The method `__new__()` may either be defined directly for the class or the class may inherit it from one of its parent classes.

- The method `__new__()` is implicitly considered by Python to be a static method. Its first parameter is meant to be set equal to the name of the class whose instance is desired and it must return the instance created.

- If a class does not provide its own definition for `__new__()`, a search is conducted for this method in the inheritance tree that converges on the class (more on that later).

- **STEP 2:** Then the method `__init__()` of the class is invoked to initialize/configure the instance returned by `__new__()`. If a class does not provide its own `__init__()`, an inherited version of the same is used.

A Simple Example that Illustrates the use of `__new__()` and `__init__()` for Instance Construction

- In the following script we define a class `X` and provide it with a static method `__new__()` and an instance method `__init__()`.

- We do not need any special declaration for `__new__()` to be recognized as static because this method is special-cased by Python.

- Note the contents of the namespace dictionary `__dict__` created for class `X`, as printed out by `X.__dict__`, at the end of the processing of the class definition by the Python compiler. On the other hand, `dir(X)` also shows the names inherited by `X`.

- Also note that the namespace dictionary `xobj.__dict__` created at runtime for the instance `xobj` is empty — for obvious reasons.

- As stated earlier, when `dir()` is called on a class, it returns a list of all the attributes that can be invoked on the class and on the instances made from the class. The returned list also includes the attributes inherited by a class from its base classes.

- When called on an instance, as in `dir( xobj )`, the returned list is the same as sbove plus any instance variables defined for the class.

```
#!/usr/bin/python

#------------------------  class X ----------------------------
class X( object ):

    def __new__( cls ):
        print "__new__ invoked"
        return object.__new__( cls )

    def __init__( self ):
        print "__init__ invoked"

#----------------------  Test Code  ---------------------------

xobj = X()                        # __new__ invoked
                                  # __init__ invoked
print X.__dict__
        # {'__module__': '__main__',
        #  '__new__': <staticmethod object at 0x4036835c>,
        #  '__dict__': <attribute '__dict__' of 'X' objects>,
        #  '__weakref__': <attribute '__weakref__' of 'X' objects>,
        #  '__doc__': None,
        #  '__init__': <function __init__ at 0x403b2d4c>}

print xobj.__dict__               # {}
print dir(X)
        # ['__class__', '__delattr__', '__dict__', '__doc__',
        #  '__getattribute__', '__hash__', '__init__',
        #  '__module__', '__new__', '__reduce__',
        #  '__reduce_ex__', '__repr__', '__setattr__', '__str__',
        #  '__weakref__']
print dir( xobj )
        # ['__class__', '__delattr__', '__dict__', '__doc__',
        #  '__getattribute__', '__hash__', '__init__',
        #  '__module__', '__new__', '__reduce__',
        #  '__reduce_ex__', '__repr__', '__setattr__', '__str__',
        #  '__weakref__']
```

How Python Constructs an Instance from a Classic Class

- There does not exist a separate `__new__()` method for constructing an instance from a classic class.

- The call to the class itself results in the construction of an instance object that is subsequently (and automatically) initialized by the class's `__init__()` method if the class is provided with such a method.

- The script shown below defines a classic Python class `X`. It is classic because `X` is not subclassed from the root class `object`.

- The class is not provided with a `__new__()` method because it does not need one for instance construction.

- Note the contents of the namespace dictionary `__dict__` that is created by the compiler for class `X` and the namespace created at runtime for the instance `xobj`. Since the class does not define any instance variables or instance methods, the latter namespace is empty at this time.

- Also shown are the outputs produced by the function `dir()` when called on the class and on the instance.

```
#!/usr/bin/python

#-----------------------  class X ----------------------------
class X:
    def __init__( self ):
        print "__init__ invoked"

#-------------------------  Test Code  -----------------------
xobj = X()                      # __init__ invoked
print X.__dict__
             # {'__module__': '__main__',
             #  '__doc__': None,
             #  '__init__': <function __init__ at 0x403b2d84>}
print xobj.__dict__      # {}
print dir( X )      # ['__doc__', '__init__', '__module__']
print dir( xobj )   # ['__doc__', '__init__', '__module__']
```

The Syntax for Defining a Method

- As the reader has already seen, the methods defined for a class must have special syntax that reserves the first parameter for the object on which the method is invoked. This parameter is typically named `self` for instance methods, but could be any legal Python identifier.

- In the script shown next, when we invoke the constructor using the syntax

```
xobj = X( 10 )
```

  the parameter `self` in the call to `__init__()` is set implicitly to the **instance** under construction and the parameter `nn` to the value 10.

- A method may call any other method of a class, but such a call must always use class-qualified syntax, as shown by the definition of `bar()`.

- One would think that a function such as `baz()` could be called by using the syntax `X.baz()`, as we tried to do in the commented out line, since after all the name `baz` belongs to the namespace of the class `X` (as can be verified by examining the `__dict__` attribute of `X`). But that does not work. **So the function `baz()` is useless for all practical purposes.**

```
#!/usr/bin/python

#-----------------------  class X ----------------------------
class X:
    def __init__( self, nn ):
        self.n = nn

    def getn( self ):
        return self.n

    def foo( self, arg1, arg2, arg3 = 1000 ):
        self.n = arg1 + arg2 + arg3

    def bar( self ):
        self.foo( 7, 8 , 9)

    def baz():
        self.n = 314
#----------------  end of class definition  --------------------

xobj = X( 10 )
print xobj.getn()       # 10

xobj.foo( 20, 30 )
print xobj.getn()       # 1050

xobj.bar()
print xobj.getn()       # 24

#X.baz()                # ERROR
```

A Method Can be Defined Outside a Class

- It is not necessary for the body of a method to be enclosed by a class.

- A function object created outside a class can be assigned to a name inside the class. That name will acquire the binding that will be the function object. Subsequently, that name can be used in a method call as if the method had been defined inside the class.

- In the script shown on the next page, the important thing to note is that the assignment to `foo` gives `X` an attribute that is a function object. As shown, this object can then serve as an instance method.

```
#!/usr/bin/python

def bar( self, arg1, arg2, arg3 = 1000 ):
    self.n = arg1 + arg2 + arg3

#-----------------------  class X ----------------------------

class X:
    foo = bar

    def __init__( self, nn ):
        self.n = nn

    def getn( self ):
        return self.n

#----------------  end of class definition  -------------------

xobj = X( 10 )
print xobj.getn()       # 10

xobj.foo( 20, 30 )
print xobj.getn()       # 1050
```

A Class Can Have Only One Method of a Given Name

- When the Python compiler digests a method definition, it creates a function binding for the name of the method.

- For example, for the following code fragment

  ```
  class X:
      def foo( self , arg1, arg2 ):
          ....implmentation of foo....

      ....rest of class X....
  ```

  the compiler will introduce the name `foo` as a key in the namespace dictionary for class `X`. The value entered for this key will be the function object corresponding to the body of the method definition shown above.

- So if you examine the attribute

  ```
  X.__dict__
  ```

  after the class is compiled, you'll see the following sort of entry in the namespace dictionary of class `X`:

  ```
  'foo': <function foo at 0x805a5e4>
  ```

- Since all the method names are stored as keys in the namespace dictionary and since the dictionary keys must be unique, this implies that there can exist only one function object for a given method name.

- As a result, if after seeing the above code snippet, the compiler saw another definition for a method named `foo` for the same class, then **regardless of the parameter structure of the function** the new function object will replace the old function object for the value entry for the method name.

- In OO languages like C++ and Java, a function is characterized by its **signature** which consists of the name of the function followed by a list of its parameter types. It is common in C++ and Java for a class to possess two or more functions of the same name as long as their signatures are different.

Method Names Can be Usurped by Data Attribute Names

- We just talked about how there can be only one method of a given name in a class — regardless of the number of arguments taken by the method definitions.

- As a more general case of the same property, a class can have only one attribute of a given name.

- What that means is that if a class definition contains a class variable of a given name *after* a method attribute of the same name has been defined, the binding stored for the name in the namespace dictionary will correspond to the definition that came later.

- Of course, this also works in reverse. If a class definition contains a data attribute of a given name and then later the a method attribute of the same name appears, it is the latter that will be retained.

Bound and Unbound Methods

- To understand how you can endow a Python class with static methods, it is important to understand what is meant by bound and unbound methods in Python.

- In general, when a method is invoked on an instance object or on the class itself, Python associates with the method call the following attributes: `im_self`, `im_func`, and `im_class`.

- Initially, when the method call is first initialized, the `im_self` attribute is set to `None`.

- Subsequently, if the implicitly-supplied first argument to the method call is an instance object, the `im_self` attribute is set to a reference to that instance. **In this case, we say that the method object is bound**.

- Since, in general, a method can be called on any object, what if a function is called directly on the class itself? In this case, the `im_self` attribute of the method objects is set to `None` and the method object is said to be **unbound**.

- In both cases, the `im_class` attribute would be set to the name of the class. And, again in both cases, the `im_func` attribute would be set to the function object in question.

- A method that would ordinarily be called as a bound method on an instance object may also be invoked as an unbound method directly on the class, as shown below:

```
#!/usr/bin/python

### BoundAsUnbound.py

class X:
    def foo( self, mm ):
        print "mm = ", mm

xobj = X()
print X.foo              # <unbound method X.foo>
print xobj.foo
  # <bound method X.foo of <__main__.X instance at 0x403b51cc>>

# call foo() as a bound method:
xobj.foo( 10 )           # mm =  10

# call foo() as an unbound method:
X.foo( xobj, 10 )        # mm =  10
```

- As we will see later, calling a class method as an unbound method is particularly useful when a subclass needs to call a particular base class method.

Using __getattr__ as a Catch-All for Non-Existent Methods

- The role played by `AUTOLOAD` in Perl OO is played by the system-supplied `__getattr__()` method for a Python class.

- If a non-existent method is invoked on an instance object, Python farms out that call to the `__getattr__()` method provided a class possesses, either directly or through inheritance, a definition for this method.

- By a non-existent method call we mean a method call whose definition cannot be found either in the class or through a search in the inheritance tree that converges on that class.

- For this to work, the `__getattr__()` method must be defined with two parameters. The system would set the first to the instance object on which the non-existent method is called and set the second to the name of the non-existent method.

- Additionally, `__getattr__()` must return a callable object. An object is considered **callable** if it can be called with a function-call operation, that is with the '`()` operator, with or without arguments.

__getattr__() versus __getattribute__()

- For new style classes, Python makes available the `__getattribute__()` method that is called whenever a method is invoked.

- The `__getattribute__()` method is defined for the root class `object` with a do-nothing implementation. It can however be overridden in your own class to provide any set-up operations before the code in a method is actually executed.

- There is a difference between the behavior of `__getattr__()` and that of `__getattribute__()`: Whereas the former is called only if a non-existent method is invoked on an instance object, the latter is called whenever a method is invoked.

Subjecting a Callable to a Function Call Operation

- There exists an interesting difference between Perl and Python: the difference between just accessing a callable defined for a class and subjecting the callable to a function call operation with the '`()`' operator.

- This distinction also applies to any callable object, whether or not it is defined for a class.

- For a class `X` with a method named `foo`, what this means is that calling just `X.foo` will return a result different from what is returned by `X.foo()`. The former will return the function object itself and the latter will cause execution of the function object associated with the method name.

Destruction of Instance Objects

- Just like Perl, Python also comes with an automatic garbage collector. The basic principle on which the Python garbage collector works is the same as in Perl.

- Each object created is kept track of through reference counting. Each time an object is assigned to a variable, its reference count goes up by one, signifying the fact that there is one more variable holding a reference to the object.

- And each time a variable whose referent object either goes out of scope or is changed, the reference count associated with the object is decreased by one.

- When the reference count associated with an object is zero, it becomes a candidate for garbage collection.

- In much the same way as `DESTROY` in Perl, Python provides us with a function `__del__()` whose override definition can be used for cleaning up any acquired system resources, such as filehandles, sockets, etc., before an object is actually destroyed and the memory reclaimed.

Encapsulation Issues for Classes

- Encapsulation is one of the cornerstones of OO. How does it work in Python?

- The same as in Perl. All of the attributes defined for a class are available to all.

- So the language depends on programmer cooperation if software requirements, such as those imposed by code maintenance and code extension considerations, dictate that the class and instance variables of a class be accessed only through the methods provided for that purpose.

- As with Perl, a Python class and a Python instance object are so open that they can be modified *after* they are brought into existence at run time.

Defining Static Attributes for a Class

- A class definition usually includes two different kinds of attributes: those that exist on a per-instance basis and those that exist on a per-class basis.

- The Python classes we have defined so far had the data and method attributes of only the first kind; that is they all existed on a per-instance basis.

- But sometimes it is useful to endow a class with attributes that exist on a per-class basis. Such attributes are also known to be **static**.

- In Python, a variable becomes static if it is declared outside of any method in a class definition. We refer to such variables as **class variables** (as opposed to instance variables).

- For a method to become static, it needs the `staticmethod()` wrapper.

- Shown on the next slide is a class with a class variable `next_serial_number`:

```
#!/usr/bin/python

### Static1.py

#------------------  class Robot  ----------------------
class Robot:
    next_serial_number = 1

    def __init__( self, an_owner ):
        self.owner = an_owner
        self.idNum = self.get_next_idNum()

    def get_next_idNum( self ):
        new_idNum = Robot.next_serial_number
        Robot.next_serial_number += 1
        return  new_idNum

    def get_owner( self ):
        return self.owner
    def get_idNum( self ):
        return self.idNum
#------------  end of class definition  ----------------

robot1 = Robot( "Zaphod" )
print robot1.get_owner(), robot1.get_idNum()    # Zaphod 1

robot2 = Robot( "Trillian" )
print robot2.get_owner(), robot2.get_idNum()    # Trillian 2
```

- A static method is created by supplying a function object to `staticmethod()` as its argument. The function object returned by `staticmethod()` will behave as a static method for the class.

- For example, to make `foo()` a static method of a class, we'd do the following

  ```
  def foo():
      print "foo called"

  foo = staticmethod( foo )
  ```

  Subsequently, when `foo` is called directly on the class using the function call operator '`()`', it is the callable object bound to `foo` in the last statement above that gets executed.

- The same idea works for static methods with arguments and for static methods that need to call other static methods in the same class.

An Instance Variable Hides a Class Variable of the Same Name

- When **self.attribute** notation is used to access a data attribute of a class, an instance variable of a given name will hide a class variable of the same name.

- When a class variable gets hidden in this manner, it can still be accessed with the **class.attribute** notation.

Private Attributes in a Class

- Python provides a mechanism for endowing a class with private data and method attributes.

- This is done though name mangling in such a way that a private name becomes "inaccessible" outside the class.

- Any data-member name or a member-function name that has at least two leading underscores and at most one trailing underscore is private to that class.

- However, it is important to bear in mind that the "privateness" achieved in this manner still depends a great deal on programmer cooperation. Since the result of name mangling is predictable in advance, the names can still be reached outside the class through their mangled versions.

- A name that has at least two leading underscores and at most one trailing underscore is renamed by the compiler by attaching to the name an underscore followed by the class name.

- For example, an attribute name such as

  ```
  __m
  ```

  in a class callded `X` will be mangled into

  ```
  _X__m
  ```

Defining a Class with Slots

- New style classes allow the `__slots__` attribute of a class to be used to name a list of instance variables. Subsequently, no additional instance variables can be assigned to instances of such a class.

- In the following example, we try for `__init__()` to declare an instance variable, `c`. This is in addition to the instance variables already defined through the `__slots__` attribute. However, this becomes the source of error at runtime when the virtual machine executes the statement in the test code.

```
#!/usr/bin/python

### ClassWithSlots2.py

class X( object ):
    __slots__ = ['a', 'b']

    def __init__( self, aa, bb, cc ):
        self.a = aa
        self.b = bb
        self.c = cc            # Will cause error in test code

#------------------------ Test Code ---------------------------
xobj = X( 10, 20, 30 )
           # AttributeError: 'X' object has no attribute 'c'
```

- Accessing an instance variable that is declared as a slot but that has not been initialized will evoke the `AttributeError` from Python. Doing the same for a classic class returns the answer `None`.

Descriptor Classes in Python

- A **descriptor class** is a new-style class with override definitions for at least one of the special system-supplied methods: `__get__()`, `__set__()`, and `__delete__()`.

- When an instance of such a class is used as a static attribute in another class, accesses to those values are processed by the `__get__()` and the `__set__()` methods of the descriptor class.

- The class whose static attributes are instances of the descriptor class is known as the **owner class**.

- Shown in the script below is a simple descriptor class, a new-style class obviously since it is subclassed from `object`, that stores a single data value.

- The print statements in the override definitions of `__get__()` and `__set__()` are merely to see these methods getting invoked automatically.

- The calls to the `__get__()` and `__set__()` methods of a descriptor class are orchestrated by the `__getattribute__()` method of the owner class. If the owner class overrides `__getattribute__()`, the methods `__get__()` and `__set__()` may not be called when accessing instances made from the descriptor class. The owner class must also be a new-style class so that it can inherit `__getattribute__()` from `object`.

- The script next defines an owner class, `UserClass`, that is again a new style class. The owner class has been provided with three **static** attributes, `d1`, `d2`, and `d3`. The first two of these are set to instances of the `DescriptorSimple` class.

- We then construct an instance of `UserClass`. When we try to retrieve the value of the `d1` attribute of the class, the following message is displayed on the screen:

```
    Retrieving with owner instance:  \
                            <__main__.UserClass object at 0x403b5b2c>
    and owner type:  <class '__main__.UserClass'>

    100
```

```
#!/usr/bin/python

### DescriptorClass.py

#-------------------  class DescriptorSimple  ------------------

class DescriptorSimple( object ):
    def __init__( self, initval=None ):
        self.val = initval

    def __get__( self, owner_inst, owner_type ):
        print "Retrieving with owner instance: ", owner_inst,   \
                           " and owner type: ", owner_type
        return self.val

    def __set__( self, owner_inst, val ):
        print 'Setting attribute for owner instance: ', owner_inst
        self.val = val

#--------------------  class UserClass  ------------------------

class UserClass( object ):
    d1 = DescriptorSimple( 100 )
    d2 = DescriptorSimple( 200 )
    d3 = 300

#-------------------------  Test Code  ------------------------

u = UserClass()
print u.d1                  # 100
print u.d2                  # 200
print u.d3                  # 300

print UserClass.d1          # 100

u.d1 = 400                  # does the expected thing
u.d2 = 500                  # also does the expected thing

print u.d1                  # 400
print u.d2                  # 500
print UserClass.__dict__
     # {'__module__': '__main__', '__doc__': None,
     #  '__dict__': <attribute '__dict__' of 'UserClass' objects>,
     #  '__weakref__': <attribute '__weakref__' of 'UserClass' objects>,
     #  'd2': <__main__.DescriptorSimple object at 0x403b5b0c>,
     #  'd3': 300,
     #  'd1': <__main__.DescriptorSimple object at 0x403b59ec>}

UserClass.d1 = 600          # Does NOT do what you might think!!!

print UserClass.__dict__
     # {'__module__': '__main__', '__doc__': None,
     #  '__dict__': <attribute '__dict__' of 'UserClass' objects>,
     #  '__weakref__': <attribute '__weakref__' of 'UserClass' objects>,
     #  'd2': <__main__.DescriptorSimple object at 0x403b5b0c>,
     #  'd3': 300,
     #  'd1': 600}

print u.__dict__            # {}
```

Extending a Class in Python

- An inheritance chain in Python is constructed by including the name of the superclass in the header of a subclass, as in

```
    class SomeSubClass( SomeSuperClass ) :
```

- A **derived-class method** overrides a base-class method of the same name.

- Sometimes it is useful for a derived-class method to get a part of its work done by a base-class method of the same name. When a derived-class method makes a call to a base-class method of the same name, we say that the derived-class is extending the base-class method. The syntax for doing this is straightforward in a **single-inheritance chain**; that is when a derived class has a single parent superclass. We will address later the issue of how the same can be done for the case of multiple inheritance; that is when a subclass is derived from multiple parents.

- Extending base-class methods when multiple inheritance is involved becomes a non-trivial issue especially in the presence of what is known as **diamond inheritance**, that is when multiple parents of a subclass may themselves be subclassed from **a common ancestor class**.

- Method extension for the case of single-inheritance is illustrated by the `Employee`–`Manager` class hierarchy in the next script in which the derived-class method `promote()` calls the base-class method `promote()`. Along the same lines, the derived-class method `myprint()` calls the base-class method `myprint()`.

- Extending methods in multiple inheritance hierarchies requires calling `super()`. To illustrate, suppose we wish for a method `foo()` in a derived class `Z` to call on `foo()` in `Z`'s superclasses to do part of the work, `Z`'s `foo()` would need to invoke `super()` with the following specialized syntax:

```
class Z( A, B, C, D ):
    def foo( self ):
        .... do something ....
        super( Z, self ).foo()
```

```
#!/usr/bin/python

### DerivedCallingBase.py

#--------------------  class Employee  -----------------------
class Employee:
    def __init__( self, a_name, a_position ):
        self.name = a_name
        self.position = a_position

    def get_position( self ):
        return self.position

    def set_position( self, new_position ):
        self.position = new_position

    promotion_table = {
        'shop_floor' : 'staff',
        'staff' : 'management',
        'astt_manager' : 'manager',
        'manager' : 'executive'
    }

    def promote( self ):
        self.position = Employee.promotion_table[ self.position ]

    def myprint( self ):
        print "Name:", self.name, " Position:", self.position,


#-------------------  class Manager  -----------------------
class Manager( Employee ):
    def __init__( self, a_name, a_position, a_department ):
        Employee.__init__( self, a_name, a_position )
        self.department = a_department

    def get_department( self ):
        return self.department

    def promote( self ):
        if ( self.position == 'executive' ):
            print "A Manager cannot be promoted beyond Executive"
            return
        Employee.promote( self )

    def myprint( self ):
        Employee.myprint( self )
        print "Dept:", self.department

#-------------------  Test Code -----------------------------

emp = Employee( "Orpheus", "staff" )
emp.promote()
position = emp.get_position()
print "position:",  position
man = Manager( "Trillion", "manager", "sales" )
man.myprint()   # Name: Trillion  Position: manager  Dept: sales
man.promote()
man.myprint()   # Name: Trillion  Position: executive  Dept: sales
man.promote()   # A manager cannot be promoted beyond Executive
```

The New Style Classes Revisited

- As you already know, Python now supports two different kinds of classes, **classic classes** and **new-style classes**.

- A new style class is subclassed from the system-supplied `object` class or from a child class of the `object` class.

- All of the built-in classes are new style classes. That is, the built-in types such as **lists**, **tuples**, **dictionaries**, etc., are now new-style classes. This allows them to be subclassed for defining more specialized utility classes.

- An instance of a new-style is created by the static method `__new__()`. If a user-defined class is not provided with an implementation for this method, its inherited definition from one of the superclasses will be used. Since `object` is at the root of the inheritance tree of all new-style classes, in the absence of any other class making available a `__new__()`, the system-supplied definition for `object`'s `__new__()` will be used.

- An instance returned by `__new__()` is automatically initialized by the class's `__init__()` method. If a class does not directly provide a definition for this method, an inherited definition is used. If a definition is not available from one of user-defined superclasses, `object`'s `__init__()` used.

- For new style classes, an instance `xobj`'s class can be ascertained by calling `xobj.__class__`. For classic classes, the same is accomplished by calling `type(xobj)`. By the way, `type(xobj)` also works for new style classes.

- To ascertain whether an instance `xobj` is of a certain specific type, we can use the function `isinstance()`, as in `isinstance(xobj, class_name)`.

- When a list of instance variables is assigned to the `__slots__` attribute of a new-style class, an instance of that class cannot be assigned any additional instance variables.

- The new style class come equipped with the `__getattribute__()` method that is inherited from `object`. A class is free to provide an override definition for this method. This method is accessed whenever any class method is accessed.

- Using multiple inheritance, a subclass derived from a mixture of classic and new-style classes is treated as a new style class.

- You cannot multiply inherit from the different built-in types. For example, you cannot construct a class that inherits from both the built-in `dict` and the built-in `list`.

- When the constructor of a built-in type is called without any arguments, it results in an instance with an appropriate default state. For example, `str()` returns an empty string, and `int()` returns the integer 0.

- The built-in types `staticmethod`, `super`, `classmethod`, and `property` have special roles to Python OO. The calls to their constructors return function objects. For example, when `staticmethod()` is called with a class method as its argument, the function object returned by the call can then behave like a static method of the class.

Extending the Built-In Types

- As mentioned already, all of the built-in classes in Python are now new-style classes. That allows for them to be extended for creating more specialized classes.

- Let's say you want a variation on the built-in `dict` class that would allow us to construct a dictionary from two arguments, one a list of keys and the other a list of the corresponding values. That can be done easily by extending `dict` into, say, `MyDict` and providing an appropriate override for the `__init__()` method.

- In general, extending a built-in type may involve overriding just the `__new__()`, or just the `__init__()`, or both. If the extension requires customization of memory allocation for the instances of the new class, you'd need to override `__new__()`.

- The next script shows us subclassing the built-in `int` class. We call the new class `size_limited_int`.

- We want the `size_limited_int` constructor to raise an exception if an attempt is made to construct an integer instant whose integer value is outside the range permitted by the class.

- We also want to provide the class with an override definition for the '+' operator that would allow us to add two integers of type `size_limited_int`, with the returned sum as also being of type `size_limited_int`.

- For the override for the '+' operator through the implementation for `__add__()`, note the call to `super()` in the script below. This causes the parent class's `__add__()` to be invoked for the addition operation. With the syntax used, the call to `super()` is guaranteed to return an object of the subclass type if anything is returned at all.

- Since we have a single-chain inheritance in the script below, we could also have used the simpler syntax shown in the commented-out line shown just after the call to `super()` as opposed to calling `super()`.

- The last statement in the definiton of `__add__()` ensures that the sum of the two integer values would still obey the size constraint represented by the `size_limited_int` class.

```
#!/usr/bin/env python

### SubclassingIntClass.py

#-------------  class size_limited_int  ----------------

class size_limited_int( int ):

    maxSize = 100

    def __new__( cls, intValue, size = 100 ):
        cls.maxSize = size
        if intValue < -cls.maxSize or intValue > cls.maxSize:
            raise Exception( "out of range")
        return int.__new__( cls, intValue )

    def __add__( self, arg ):
        res = super( size_limited_int, self ).__add__( arg )
#       res =  int.__add__( self, arg )
        return size_limited_int( res, self.maxSize )

#----------------------  Test Code  ----------------------

n1 = size_limited_int( 5 )
print n1                                    # 5
print isinstance( n1, size_limited_int )    # True
print isinstance( n1, int )                 # True
print isinstance( n1, object )              # True

try:
    n2 = size_limited_int( 1000 )
except Exception, error:
    print error
                                            # out of range
```

```
n3 = size_limited_int( 10 )
n4 = n1 + n3
print n4                                   # 15
print isinstance( n4, size_limited_int )    # true
print isinstance( n4, int )                # True
print isinstance( n4, object )             # True
```

Abstract Classes and Methods in Python

- Abstract classes play a very important role in OO.

- An abstract class can represent the root interface of a class hierarchy. The concrete classes in the hierarchy can then be the different implementations of the interface, each implementation designed for a different but related purpose.

- As an example, we can have a `Shape` hierarchy. A `Shape` represents an abstract notion. The root class `Shape` may only declare abstract methods such as `area()` and `circumference()`. It would not be able to provide implementations for these methods since, as a pure abstraction, a `Shape` has no geometrical detail. Concrete classes such as `Circle` and `Rectangle` derived from `Shape` would be expected to provide implementations for the abstract methods declared for `Shape`.

- A Python class can be made abstract by having its constructor throw an exception if it should get invoked by a client of the class. This would prevent a client from constructing an instance of what is supposed to be an abstract class. The same can be done for a method if it is supposed to be abstract.

- The `NotImplementedError` exception is used in Python to designate abstract classes and methods.

- A Python class is made abstract by having its constructor raise the `NotImplementedError` exception.

- A method is made abstract by having its body do the same. When a method is made abstract in this manner, there is the expectation that the full implementation of the method will be supplied in a derived class.

Multiple Inheritance in Python OO

- Like Perl, Python allows a class to be derived from multiple base classes. The header of such a derived class would look like

```
class derived_class( base1, base2, base3, .... ):

    ....body of the derived class....
```

- Now the following question arises: suppose we invoke a method on a derived-class instance and the method is not defined directly in the derived class, in what order will the base classes be searched for a definition for the method?

- The order in which the class and its bases are searched for an applicable definition is referred to as the **Method Resolution Order** (MRO).

- Different algorithms are used for determining the MRO for classic classes and for new-style classes.

- In what follows, we will first discuss how MRO works for classic classes in Python — it is the same as in Perl. Next, we will present MRO for the new-style classes.

Method Resolution Order for Classic Classes

- Consider the following example of multiple inheritance for classic classes:

```
        X         Y
        ^         ^
        |         |
         \       /
           \   /
             Z
```

  where a class `Z` is derived from the base classes `X` and `Y`. The header of class `Z` will therefore look like

```
    class Z( X, Y):
        ....body of class Z....
```

- Let's now construct an instance of `Z` and invoke a method on it

```
    zobj = Z( ...arguments... )
    zobj.foo()
```

  If `Z` does not provide `foo()` directly, Python would need to search through the base classes in some order for a definition of `foo()`.

- MRO for classic classes consists of a left-to-right fashion in a depth-first manner.

- In our example, that means that the compiler will first search in class `X` and then in class `Y`. To illustrate what is meant by "depth-first", assume that `X` and `Y` are themselves derived classes. Now, after searching through `X`, `X`'s bases will be searched for a definition for `foo()`, again in a left-to-right order.

- If you want to have a greater control over the selection of a method, that is if you want to specify that a method from a particular ancestor of a derived class be used, you have to use the syntax

```
base_class_name.method_name( derived_class_instance )
```

- Since Python allows for multiple inheritance, it is possible to create what are known as diamond inheritance loops in a class hierarchy. This happens when two or more classes that serve as bases for a given derived class are themselves derived from a common base. For example, the following class hierarchy exhibits an inheritance loop:

```
     R
    ^ ^
    | |
   /   \
 /       \
|         |
X         Y
^         ^
|         |
 \       /
   \   /
     Z
```

- Since inheritance loops such the one shown above create all sorts of problems when it comes to writing good (good in the sense of being easily maintainable and easily extendible) and bug-free code in C++ that also allows for multiple inheritance, one is naturally curious whether the same difficulties crop up in Python.

- Fortunately, the sort of difficulties that diamond hierarchies create in C++ are not seen in Python.

- The inheritance mechanism in Python's object model consists of linking the namespace dictionary of a derived class object with the namespaces of all the base class sub-objects. So when a search for an attribute in the namespace dictionary of a derived class object does not bear fruit, the search is shifted to the namespace dictionary of a base-class sub-object relevant to the derived-class object. As mentioned already, for classic classes this search takes place in a depth-first, left-to-right manner through all the bases classes listed for a derived class.

Desirable Properties for MRO

- The previous subsection presented the left-to-right depth-first MRO that is used to search for a method (actually the name of any attribute) in a hierarchy of classic classes. We will refer to this lookup algorithm as the **LRDF algorithm** for obvious reasons.

- Straightforward though it is, LRDF has some serious shortcomings for complex class hierarchies, especially hierarchies with inheritance loops, as in

```
      A
     ^ ^
     | |
    /   \
  /       \
B           C
^           ^
|           |
 \         /
   \     /
      D
```

  This sort of an inheritance loop is also referred to as **diamond inheritance** for obvious reasons.

- Let's say that both `A` and `C` provide separate definitions for a method named `foo()`. When we call `foo()` on a `C` instance, obviously it will be `C`'s `foo()` that will get invoked. Now let's also say that `B` and `D` do not provide definitions for a method named `foo()`. If we call `foo()` on a `D` instance, the LRDF rule will invoke `A`'s `foo()` and not `C`'s `foo()` even though `C` is closer to `D` than `A` with regard to inheritance.

- This behavior of LRDF-based MRO is counterintuitive in the sense that after you get accustomed to `C` exhibiting its `foo()` behavior (even though it is based on `foo()` as inherited from `A`), you'd expect `D` — since it is derived from `C` — to exhibit the same behavior. If suddenly `B`'s insertion in the superclass list of `D` — knowing fully well that `B` does not possess `foo()` — causes `D` to acquire its `foo()` behavior from a different class, in this case `A`, you'd be perplexed indeed.

- This behavior of LRDF-based MRO can also be a source of errors. If only `A` and `C` are provided with `foo()`, a programmer would expect that when `foo()` is called on a `D` instance, it will be `C`'s `foo()` that will get invoked, as opposed to `A`'s.

- This limitation of the LRDF-based MRO can be formalized by saying that it lacks **monotonicity**.

- For a more precise definition of monotonicity in name lookup in inheritance graphs, we need to introduce the notion of **superclass linearization**.

- In general, a superclass linearization (*of the inheritance graph*) for a class `C` is the list of all the classes, starting with `C`, that should be searched sequentially from left to right for a name. The superclass linearization for a class `C` will be denoted `L[C]`.

- Here are the superclass linearizations produced by the LRDF-based MRO for the different target classes in the ABCD class hierarchy shown earlier:

```
L[A] = A
L[B] = B A
L[C] = C A
L[D] = D B A C A
```

  When we examine `L[C]`, `C`'s methods get priority over `A`'s methods. However, when we examine `L[D]`, it is exactly the opposite — `A`'s methods get priority over `C`'s methods.

- For an MRO algorithm to be monotonic, if `P` precedes `Q` in the linearization of `R`, then `P` must precede `Q` in the linearization of every subclass that is derived from `R`.

- A good MRO algorithm for superclass linearization must be monotonic.

- A good MRO algorithm must also preserve **local precedence ordering**. What that means is that the order in which the immediate base classes of a given class `P` make their appearance in the linearization for `P` or any of its subclasses must not violate the base class order of `P`. In other words, if `Q` comes before `R` in the immediate bases of `P`, then `Q` must come before `R` in the linearization for `P` and of the direct and indirect subclasses that are derived from `P`.

- So whereas monotonicity refers to the priority to be accorded to the names in a class vis-a-vis the names in a superclass, local precedence order deals with the priority to be accorded to the names in one base class vis-a-vis the names in another base class at the same level of inheritance.

- For some class hierarchies it can be shown trivially that they would not admit good superclass linearizations regardless of what MRO rules are used for the purpose. This happens particularly if two classes in a hierarchy inherit from two separate bases but in opposite order.

- Python uses the C3 algorithm to derive superclass linearizations in multiple inheritance hierarchies consisting of new-style classes.

- The MRO for each class is stored in the `__mro__` attribute of the class.