

Python Course: Lecture 10

January 26, 2006

1 Types, Objects, and Classes

- Every piece of data in Python is an *object*.
- You can create your own kinds of data object by writing Python code that defines *classes*.

```
class Vector(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __len__(self):
        return (self.x**2 + self.y**2)**0.5
```

- Classes are syntactic conveniences that make it easy to associate blocks of code with particular data structures.
- To use a class, you create a particular *instance* of that class by calling its constructor.

```
>>> v1 = Vector(3,4)
>>> len(v1)
5
>>> v2 = Vector(1,2)
>>> len(v2)
2
```

- Each instance is a separate Python object. They do have something in common, though, inasmuch as they are instances of the same class. They are, in a sense, distinct entities, but the same kind of entity.
- All objects have a *type*.

```
>>> type(4)
<type 'int'>
>>> type([1,2,3,'infinity'])
<type 'list'>
>>> type(4.2+7j)
<type 'complex'>
>>> type({})
<type 'dict'>
```

- A list of types is provided by the `types` module.

```
>>> type(4) == types.IntType
True
>>> type(4) == types.StringType
False
```

- The type of an instance is determined by its class.

```
>>> type(Vector(4,5))
<class 'Vector'>
```

- Classes also support the notion of *inheritance* which is a useful way of sharing functionality between different parts of a program.
- The `isinstance` function is useful for determining what kind of object you're dealing with.
- `isinstance` returns `True` if the first argument of the second argument or one of its parent classes. This is an ontological assertion. For example, let `B` be a subclass of `A` and `b` be an instance of `B`.

```
>>> isinstance(b, tcheck.B)
True
>>> isinstance(b, tcheck.A)
True
```

2 The Class Type

- Class definitions can be objects as well as instances.

```
>>> Vector(3,4)
<point.Vector object at 0xb756cbec>
>>> Vector
<class 'point.Vector'>
```

- You can assign class definitions to variables, pass them into functions, etc.
- Use `issubclass` to determine the hierarchical relationships between classes.
- The `__class__` attribute of any instance is its class object.

```
>>> Vector(3,4).__class__
<class 'point.Vector'>
>>> Vector(3,4).__class__ == Vector
True
```

- Here's my favorite way to write stringification.

```
def __repr__(self):
    return "<%s>" % self.__class__.__name__
```

- This allows you to implement class factories.

```
>>> vectory_factory = Vector
>>> vectory_factory(3,4)
<class 'point.Vector'>
```

3 Operator Overloading

- Add one vector to another.

```
class Vector(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def length(self):
        return (self.x**2 + self.y**2)**0.5

    def __add__(self, other):
        return self.__class__(self.x + other.x, self.y+other.y)
```

- Show this first with an `add` function, then with an operator function.
- Other useful operators: `__len__`, `__cmp__`.

4 Custom Properties

- You can customize the way Python objects get and set attribute values.
- For example the following class

```
class C(object):

    def __init__(self):
        self.__x = 0

    def getx(self):
        return self.__x

    def setx(self, x):
        if x < 0: x = 0
        self.__x = x

    x = property(getx, setx)
```

works like so

```
>>> c = C()
>>> c.x # Zero by default
0
>>> c.x = 5 # Positive numbers are okay
>>> c.x
5
>>> c.x = -10 # Negative numbers get set to zero
>>> c.x
0
```

5 Default Iterator

- You can specify a default iterator behavior for any Python object.

- The `__iter__` function returns a generator called when the object is placed in a `for` loop.

```
class SomeList(object):
    def __init__(self, l = []):
        self.l = l

    def __iter__(self):
        for item in self.l:
            yield item

...
>>> q = SomeList([1,3,4])
>>> q
<point.SomeList object at 0xb756ce2c>
>>> [x for x in q]
[1, 3, 4]
```

6 Built-in Class Inheritance

- You can inherit from built-in classes.

```
class MyInt(int):
    pass

class MyList(list):
    pass

class MyDict(dict):
    pass
```

- The first of these would be weird, but the next two are quite useful.
- Often you describe a data structure as being “really just a hash of...”; e.g. the consonant cluster counter.
- Other useful special member functions: `__getitem__`, `__setitem__`

7 Everything is Public

- Unlike other object-oriented languages (Java, C++, Ruby), Python has no notion of a public vs. a private function.
- Any function/attribute of an object is always available for calling/reading/assignment.

- It's good programming form to indicate in the documentation which functions are public and/or private, but the language does not enforce this.
- Attribute names that begin with a single underscore do not show up in response to a `dir()` command. This is as close as Python gets to a notion of a private member, but it's a naming convention as much as anything.

8 Exceptions are Objects

- Exceptions are objects.
- If your program generates exceptions, it should derive them from the `Exception` class. (e.g. from Guido's tutorial)

```
>>> class MyError(Exception):
...     def __init__(self, value):
...         self.value = value
...     def __str__(self):
...         return repr(self.value)
...
>>> try:
...     raise MyError(2*2)
... except MyError, e:
...     print 'My exception occurred, value:', e.value
...
My exception occurred, value: 4
>>> raise MyError, 'oops!'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
__main__.MyError: 'oops!'
```

9 Old Style Classes

- Prior to Python 2.2, the language had a different scheme for defining objects.
- Objects were not subclasses of the built-in `object` class.

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __len__(self):
```

```
return (self.x**2 + self.y**2)**0.5
```

- All old-style object instances are of the same type.

```
>>> v1 = Vector(4,5)
>>> type(v1)
<type 'instance'>
>>> b = OtherClass()
>>> type(b)
<type 'instance'>
```

- This led to an artificial dichotomy between built-in and user-defined objects that needlessly complicated the language.
- For implementation reasons old-style classes do not support the `super` function.
- Here's the (somewhat opaque) error message that you get when you forget this.

```
>>> c = SuperErrorChild()
>>> c.func()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "example.py", line 18, in func
    super(SuperErrorChild, self).func()
TypeError: super() argument 1 must be type, not classobj
```

- All objects derived from built-in Python objects are automatically new style classes.
- Support for old-style classes exists, but you should use the new style.

10 Object-Oriented Programming

- Benefits of object-oriented programming...
- Information hiding
- Self documentation
- Permit reuse (modulo refactoring)
- Model the problem space in the code