

Python Course: Lecture 9

January 24, 2006

1 Defining New Types

- Python provides some built-in objects—e.g. integers, dictionaries, lists. It would be nice if you could create your own objects.
- Like the built-in objects, you want your custom objects to contain *data* and define *behavior*.
- This approach is called *object-oriented* programming. Python utilizes this programming style extensively.
- Use the `class` keyword to define a class. The most basic class is an empty one to which we can add attributes dynamically.

```
>>> class A(object):
...     pass
...
>>> a = A()
>>> a
<__main__.A instance at 0xb732ce8c>
>>> dir(a) # See what's inside an empty class
['__doc__', '__module__']
>>> a.x = 5 # Dynamically set some attributes
>>> a.y = 7
>>> a
<__main__.A instance at 0xb732ce8c>
>>> dir(a) # Note that x and y now show up in this list
['__doc__', '__module__', 'x', 'y']
>>> a.x # The attributes have the values we specified
5
>>> a.y
7
```

- In this example *A* is a *class* and *a* is an *instance* of the class.
- This is a way to bundle items together with names—you could get the same effect with tuples and ordering conventions.

```
>>> class Stats(object):
...     pass
...
>>> s = Stats()
>>> s.mu = 5
>>> s.sd = 0.125
>>> s
<__main__.Stats object at 0xb732cb6c>
>>> s.mu
5
>>> s.sd
0.125
```

- By default, different instances are not equal even if all their attributes are the same.

```
>>> s1 = Stats()
>>> s2 = Stats()
>>> s1 == s2
False
>>> s1.mu = 5
>>> s2.mu = 5
>>> s1 == s2
False
```

- Objects are mutable.

```
>>> s3 = s1 # s3 is an alias of s1, not a new object
>>> s1.mu
5
>>> s3.mu
5
>>> s1.mu = 7
>>> s3.mu
7
```

2 Constructors

- You don't want to have to add attributes dynamically every time you create a class instance. You want a set of attributes to always be associated with a particular class.

- For example, we want a `Stats` object that knows it contains a mean and a standard deviation.
- Do it like so.

```
>>> class Stats(object):
...     def __init__(self, mu, sd):
...         self.mu = mu
...         self.sd = sd
...
>>> s1 = Stats(5, 0.125)
>>> s1
<__main__.Stats object at 0xb733144c>
>>> s1.mu
5
>>> s1.sd
0.125
```

- There are a couple of things going on here.
- We defined a *member function* (aka *method*) of the `Stats` class.
- It has the special name `__init__` surrounded by double underscores. More on naming conventions for special Python functions later.
- `__init__` takes 3 arguments. Two are the user-specified mean and standard deviation—the other is a special `self` value.
- The function uses `self` as part of its code, e.g. it assigns values to the `mu` attribute of `self` (e.g. `self.mu = mu`).
- The `__init__` function is a *constructor*. It gets called implicitly when we create an instance of `Stats` (e.g. `s1 = Stats(5, 0.2)`).
- The value of `self` is a `Stats` object instance. (e.g. It will be the same as `s1`.)
- The code you write in a class definition defines the behavior for all instances of that class. The `self` parameter is used to keep track of specific different instances.
- If you have done object-oriented programming in Java or C++, Python looks much the same except the `this` pointer is called `self` and always has to be written explicitly.
- That you have to write explicitly it is a design decision: when given a choice, Python prefers the explicit over the implicit.

3 Member Functions

- Constructors are a special kind of member function.
- You can define any kind of member function you want. They're just like other Python functions, except they always take a `self` instance variable as the first parameter.
- `self` is just a variable name, not a keyword. You don't *have* to call this variable `self`, but you should, because this is a well-established Python convention and people will find your code confusing if you don't adhere to it.
- Here is an object that models a mathematical vector in a plane. In addition to the constructor, it defines a function called `length` that uses the Pythagorean theorem $\sqrt{x^2 + y^2}$ to calculate the length of the vector.

```
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
>>> v1 = Vector(3,4)
>>> v1.length()
5.0
```

- `length` is another function like `__init__`, except we call it explicitly by name instead of implicitly as part of instance creation.
- Even though the caller doesn't specify any arguments, the code for `length` gets passed a `self` argument.
- You can think of the Python interpreter passing in whatever's to the immediate left of the function name (in this case, the value of `v1`) as the `self` argument.

4 Stringification

- It would be nice for an object to define the way it gets printed out.
- The process of converting an generic data structure into a single string is called “stringification”.
- Python objects have two stringification functions: `__repr__` and `__str__`.

- Like `__init__` they are member functions that get called implicitly in particular contexts.
- `__repr__` is called whenever you print an object in interactive mode.
- `__str__` is called when you print everywhere else, e.g. after a `print` statement or in `"%s"` string interpolation.
- If you don't specify a `__str__`, `__repr__` is used for both. It is the default.
- By convention, `__repr__` is for terse output that's helpful during debugging and `__str__` is used for nicely formatted end-user readable output.

5 Inheritance

- Objects can be instances of certain classes, and classes can be organized into hierarchies.
- Class hierarchies in the programming space can be used to model ontologies in the problem space.
- For example, in linguistics we have the generic concept of words, and we also have the concept of specific types of words like nouns and verbs.
- Nouns and verbs have some things in common by virtue of both being words, but they also have distinguishing features.
- Python object hierarchies provide a way of encoding this kind of relationship.

6 Inheritance and Constructors

- Object creation of a child class provides a good example of how classes share functionality via the `super` keyword.
- Say we have a Python class called `Parent`.

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

- This is the standard class definition we've seen before. Note the `object` in parentheses. That indicates that the `Parent` class is a subclass of the generic Python `object` class.
- All of the top level classes in your hierarchies should be a child of `object`.
- Now say we have a child class of `Parent` called `Child`.

```
class Child(Parent):
    def __init__(self, x, y):
        super(Child, self).__init__(x)
        self.y = y
```

- The `Parent` in parenthesis means that `Child` is a child of `Parent`.
- Here's a debugger trace of the creation an instance of `Child` with values `x=5` and `y=9`.
- First we call the `Child` constructor and step into it. Note that the `x` and `y` variables inside the constructor have the values assigned to them by the caller, and `self` is a `Child` object.

```
>>> pdb.run("c = Child(5, 9)")
> <string>(1)?()
(Pdb) s
--Call--
> ooex.py(11).__init__()
-> def __init__(self, x, y):
(Pdb) self
<ooex.Child object at 0xb725a10c>
(Pdb) !x
5
(Pdb) !y
9
```

- The first line in the `Child` constructor is a `super` call. This tells Python to find the parent class of `Child` and call its `__init__` function with instance variable `self` and argument `x`.

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

class Child(Parent):
    def __init__(self, x, y):
        super(Child, self).__init__(x) <== We're here before the
        self.y = y                               super call
```

```
-----
(Pdb) n
> ooex.py(12).__init__()
-> super(Child, self).__init__(x)
```

- It we step into this function, we'll find ourselves in the constructor for `Parent`. Note that the value of `x` got passed in as 5 from the other constructor and the `self` variable points to the same `Child` object instance.

```
class Parent(object):
    def __init__(self, x): <== We're here after the super call
        self.x = x
```

```
class Child(Parent):
    def __init__(self, x, y):
        super(Child, self).__init__(x)
        self.y = y
```

```
-----
(Pdb) s
--Call--
> ooex.py(6).__init__()
-> def __init__(self, x):
(Pdb) self
<ooex.Child object at 0xb725a10c>
(Pdb) !x
5
```

- The `Parent` constructor sets the `x` value, then returns control to the `Child` constructor, which sets the `y` value.

```
(Pdb) n
> ooex.py(7).__init__()
-> self.x = x
(Pdb)
--Return--
> ooex.py(7).__init__()->None
-> self.x = x
(Pdb)
> ooex.py(13).__init__()
-> self.y = y
(Pdb)
--Return--
> ooex.py(13).__init__()->None
-> self.y = y
(Pdb)
--Return--
```

```
> <string>(1)?()->None
(Pdb)
>>>
```

- When construction is all done, the instance that was getting passed along in the `self` variable is now assigned to the variable `c`. This is a `Child` object and it has both `x` and `y` attributes.

```
>>> c
<ooex.Child object at 0xb725a10c>
>>> c.x
5
>>> c.y
9
```