python™

11/27/07

>>> Overview

* objects
  * class
  * `self`
  * in-object methods
  * nice printing
  * privacy
  * property
  * static vs. dynamic
  * inheritance

>>> clientside

Objects are used nearly identically to those in Java. Suppose we want a better way of storing recipes. Basically, its all the same except that to create a new object you don't need to say `new`.

recipe_client.py

```
1  r = Recipe()
2  r.add("milk",2)    # cups
3  r.add("cheese",1) # cups
4  r.add("pasta",1)   # lbs dry
5
6  if r.needs("milk") and r.how_much("milk")>1:
7      print "We need more milk!"
8
9
10
11
```

››› class and self

The syntax for classes in Python is similar to all other conversions. However, there are some differences. The primary difference is that instance variables must be accessed using the `self` keyword. This is similar to Java's `this`. Treat it as if `self` was an external object. Another difference is that the constructor is called __init__.

recipe.py

```
1  class Recipe:
2      def __init__(self):
3          self.recipe = {}
4
5      def needs(self,food):
6          return food.lower() in self.recipe
7
8      def add(self,food,amount):
9          self.recipe[food] = amount
10
11
```

>>> methods

Unlike in Java, methods also need to be called on self. Just remember to treat `self` as the object as if you were a client.

recipe.py

```
1  class Recipe:
2      def __init__(self):
3          self.recipe = {}
4
5      def needs(self,food):
6          return food.lower() in self.recipe
7
8      def add(self,food,amount):
9          if not self.needs(food):
10             self.recipe[food] = amount
11
```

>>> nice printing

Like Java's toString, Python has a special method to change an object to a string. It is __str__.

recipe.py

```
1  class Recipe:
2      def __init__(self):
3          self.recipe = {}
4
5      def needs(self,food):
6          return food.lower() in self.recipe
7
8      def add(self,food,amount):
9          if not self.needs(food):
10             self.recipe[food] = amount
11
12     def __str__(self):
13         result = "Recipe:\n"
14         for food in self.recipe.keys():
15             result += food + " - " + str(self.recipe[food]) +"\n"
16         return result
```

>>> privacy?

Python has a weak sense of privacy unlike Java. Guido Van Rossum, the creator of Python, describes it on his blog as an "open kimono" language. There are two ways to make both methods and variables semi-private. Prefixing the name with a single underscore is a polite "Don't touch." and a double underscore predictably mangles the name of the method or variable to make it harder to use.

recipe.py

```
1  class Recipe:
2      def __init__(self):
3          self._recipe = {}
4
5      def needs(self,food):
6          return food.lower() in self._recipe
7
8      def add(self,food,amount):
9          if not self.needs(food):
10             self._recipe[food] = amount
11
```

>>> property

To compensate for Python's weak sense of privacy, there is a way of defining a property of an object to limit its use.

recipe.py

```
1  class Recipe:
2      def __init__(self):
3          self._servings = 4
4
5      def get_servings():
6          pass
7
8      def set_servings(self,s):
9          pass
10
11     servings = property(get_servings,set_servings)
```

>>> static

Java's static keyword specifies that the method or variable is used within a class and not an object (an instance of a class). This can be mimiced by not having self as the first argument in the method.

incomplete.py

```
1  class Incomplete:
2      def static(args):
3          ...
4
5      def not_static(self,args):
6          ...
7
8  Incomplete.static() # a static call
9  i = Incomplete()
10 i.not_static()      # a non-static call
11
```

>>> inheritance

Like Java, Python classes can inherit from other classes. In fact, unlike Java, Python can inherit from multiple classes. Here is a simple example of single inheritance. In the example Python is a subclass of Animal and uses Animal's speak method but its own walk method.

animals.py

```
1  class Animal:
2      def walk(self):
3          # some code
4
5      def speak(self):
6          pass
7
8  class Python (Animal):
9      def walk(self):
10         # some other code
11
```