CS 42—Object-Oriented Programming in Python

Tuesday, November 13, 2018

Object-oriented programming (OOP) in Python

Terminology

The terms below are the ones that the Python documentation uses most often, when referring to object-oriented concepts. These terms are themselves defined using the language-agnostic definitions from today's other handout. Note that languages other than Python—as well as some Python programmers and resources—might use different terms than the ones below.

object	any value in Python
class	describes the interface and implementation of an object
instance	an object that has been created from a class's description
type (of an object)	the class used to create that object
method	describes a behavior of an instance
data attribute	describes a data member of an instance
attribute	either a method or data attribute of an instance
class attribute	a data attribute defined in the namespace of a class (rather than in the namespace of an instance of that class). All instances of a class share the same set of class attributes.
`self`	the name of an object's reference to itself

Gotchas

If you're used to OOP in other languages these things in Python might seem weird at first.

- All methods (including the constructor) must explicitly declare `self` as the first parameter.[1] This rule makes it easy to distinguish attributes from local variables, in a method body. (If you're used to Java, forgetting to declare `self` as the first parameter of all your methods might be your most common mistake, when you start to write object-oriented Python programs.)
- Even though methods explicitly declare a `self` parameter, method calls don't take an explicit argument for the instance. Instead, Python makes sure that `self` is bound to the instance. (If you're used to Java, this means that—despite the weirdness of method *declarations*—Python method *calls* work just the same as Java ones.)
- In a method body, we **must** use `self` to access attributes of an instance.
- By convention, we use an underscore at the beginning of the name of any attribute that corresponds to an implementation detail (e.g., data attributes or private helper methods), like so: `self._data`. (There are no `public` and `private` declarations in Python.)
- The constructor is called `__init__`. The `__str__` method is like Java's `toString`. Both these methods are examples of "special methods", and they're Python's way of describing operations (such as initialization or converting to a string) that many instances might want to support.

[1] Technically, the first parameter doesn't need to be named `self`, but the convention is to do so. In fact, it's so much of a convention in Python programming that you should treat it as rule of the language.

Classes are (you guessed it!) just namespaces

When Python sees a statement that starts like this:

```
class Stack:
   …class body…
```

it performs the following steps:

1. Create a new namespace (which we'll refer to as *N*).
2. Run the class body as if it were a function body, using *N* as the local (and currently active) namespace.
3. Bind the name `Stack` to *N* in the originally active namespace.

Instances are (yep!) just namespaces

When Python sees a statement like this:

```
s = Stack()
```

it performs the following steps:

1. Create a new namespace (which we'll refer to as *N*).
2. Call `Stack.__init__` as if it were a function, binding `self` to *N* in the body of the local namespace for `__init__`.
3. After the constructor returns, bind the name `s` to *N* in the originally active namespace.

Assigning to an attribute creates a binding in the instance's namespace

When Python sees a statement like this:

```
s.number = 1000
```

Python binds `number` to the value `1000` in `s`'s namespace.

This statement is an example of an **object modification**. Most object modifications have the form

```
name.attribute = expression
```

where *name* and *attribute* are a valid Python variable names, *name* is bound to an instance, and *expression* is a valid Python expression. Note that this rule means that **instances are mutable**.

Attribute resolution: instance, class, superclass(es)

When a running program refers to an instance's attribute, Python will try to look up the value for that attribute's name. Python always runs the same algorithm to resolve an attribute's name:

1. **Instance:** Look for a binding in the namespace of the instance. If a binding exists for the attribute's name, use the corresponding value.
2. **Class:** If resolution fails for the instance's namespace, then look for a binding in the namespace of the instance's class. If a binding exists for the attribute's name, use the corresponding value.
3. **Superclass(es):** If resolution fails for the instance's class, then look for a binding in the namespace(s) of the instance's superclass(es). If a binding exists for the attribute's name, use the corresponding value. (Note: we haven't talked about superclasses yet.)
4. **Error:** If resolution fails for the superclass(es), throw an `AttributeError`.

Next time: inheritance