1 Administrative Topics

- I return quiz 6
- Any questions about project 6?

2 Object-oriented design

Our lectures this week are part of a transition from top-down, task-oriented, design to object-oriented design. We are shifting from thinking about the verbs (what do we need to accomplish?) to thinking about the nouns (what are the things we need to think about in order to accomplish our goals?). In object-oriented design, we make objects (based on the nouns in our problem) and give them methods (based on the verbs we have in our problem).

3 Classes

Developing our own objects requires new syntax. To create objects, we need to create classes. Classes are object “types”, or “templates”. They define what methods and what sort of data an object has. An object is often called an “instance” of a class.
What do we know about objects? We know that they have methods specific to the object type. We also know that they somehow store data. Today, we are going to talk about how to define a class for a Student object. The Student object will store the name and year of a student and will have methods called getName and getYear. In other words, we want to be able to use the Student object with code like this:

```python
s = Student( 'Smarty Pants', 2012 )
print s.getName()
print s.getYear()
```

Let’s think about what this code does, in English. It creates a Student object with the given name and year. It then executes methods on that object. Those methods must have access to the name and year data.

So, to define the class Student, Python must supply for us mechanisms to:

1. Create an object of the given class type
2. Pass initial data to that object
3. Store the data in the object
4. Define methods that have some way of accessing the data stored in the object
5. In the future, we will also need some way of manipulating the data

Python provides syntax to accomplish all of these goals. It is here:

```python
class Student:
    def __init__(self, name, year):
        self.name = name
        self.year = year

    def getName(self):
        return self.name

    def getYear(self):
        return self.year
```
There are several things I would like you to observe about this syntax, then we will step through code and discuss the memory model in greater detail. The latter will provide the most complete explanation.

Observations:

- We define a class using the keyword `class`.
- The methods are simply functions defined “inside” the class definition.
- There is a special method named `_init_`, which is a function that Python will call when a new object is being created. This supplies the mechanism for passing in and storing initial data.
- The first parameter of each method is named “self”, and it references the Student object for which the method is called.

### 3.1 Creating an Object

Whenever an object is created (e.g. using syntax like this: `s = Student( 'Smarty Pants', 2012 )`), Python follows a three step process:

1. Create an object of the correct type (using the class definition as a template)
2. Run the `_init_` method for the new object, passing it any arguments it may need
3. Return a reference to the new object
4 Stepping Through Code

Suppose we have a main module containing the class definition and three lines of code that use it:

```python
class Student:
    def __init__(self, name, year):
        self.name = name
        self.year = year

    def getName(self):
        return self.name

    def getYear(self):
        return self.year

s = Student('SP', 2012)
print s.getName()
print s.getYear()
```
Lines 1–10 in the main module contain the class definition. Executing them causes the class definition to be referenced by the main symbol table:
We begin executing line 12 (and draw it directly underneath the main symbol table). Python creates a new object (step 1 for creating a new object), using the class definition as a template. This basically means that the class’s symbol table is copied (in a funky way, with arrows) into a symbol table for the object.
Python moves on to step 2 for creating a new object, and executes the \_\_init\_\_ method for the new Student. Here, we see something interesting that Python does automagically – it passes to the method a reference to the object itself as the first parameter (self), and then passes any additional parameters (name and year).
The first line of `_init_` is an assignment statement that uses dot notation. Recall that we use dots to travel from symbol table to symbol table. The result of executing this code is that we add a row named `name` to the table referenced by `self`. So, we see a new row added to the object’s symbol table. This is how we store data in the object.
The second line of `__init__` causes a new row to be added to the object. This one contains the data for the year.
Python completes step 2 of new-object creation, so the \_init\_ symbol table is erased. Then Python goes on to step 3. It returns a reference to the new object. Finally the assignment statement from line 12 is complete, and an entry for $s$ is added to the main symbol table. Notice that the name and year data are now contained in the symbol table for the object and nowhere else.
Python goes on to execute line 13, which means the method getName must be called for the student object. We can look at this call two ways.

1. Python follows the arrows from the entry in s named getName until it arrives at the function, and calls that function, passing it s as the first parameter.

2. Python translates s.getName() to Student.getName(s).

The latter is simpler, but the former is more accurate. Either way, a symbol table is created, and the self entry is added.
The only line of code in getName is `return self.name`. This is another straightforward use of dot notation. We look up `self` in the `getName` symbol table, and follow the arrow to the table it points to. Then, we loop up the name entry in that table. It is ‘SP’.
The `getName` function returns its value to the main module, and its function table disappears. Now, Python can print 'SP' to the command line (output not shown).
The process is repeated to execute line 14. This time, the `getYear` method is called:
The year entry in the object’s symbol table is accessed:
The year is returned to the main module and it prints it to the command line (output not shown)