1 Administrative Topics

- Go over great projects
- The wiki died last night, but is back up. Please upload your pictures by tonight.
- I will have office hours from 1 to 1:30 on Friday (I am going to a conference that afternoon).

2 Lab 9

2.1 Dictionaries

I need to tell you about a very neat data structure that Python supports, that is similar to a list, but doesn’t require the indexing to use numbers. It is called a dictionary and it maps “keys” to values.

Suppose we want to build a dictionary mapping names to favorite color.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>George</td>
<td>black</td>
</tr>
<tr>
<td>Stephanie</td>
<td>purple</td>
</tr>
<tr>
<td>Scott</td>
<td>green</td>
</tr>
<tr>
<td>Milo</td>
<td>blue</td>
</tr>
</tbody>
</table>

Here is the code to do it:
d = {}  # make empty dictionary
d["George"] = 'black'
d["Stephanie"] = 'purple'
d["Scott"] = 'green'
d["Milo"] = 'blue'

We can explicitly test to see if certain values are in the dictionary. And we can access values.

```python
# Is there a color for Linda?
print 'Linda' in d

# Is there a color for George?
print 'George' in d

# What is George's favorite color?
print d['George']
```

## 2.2 Inheritance with Shapes

Let’s look at the `drawString` method in Interpreter. It takes a distance, an angle, and a string. It does not mandate that the string is generated by an L-system. That means we can use this class to draw non-L-system shapes (modularity at work!). For example.

```python
terp = interpreter.Interpreter()
terp.drawString(dstring='F−F−F−F', distance=100, angle=90)
```

And we draw a square.

Let’s design a `Shape` class that can draw the shape specified by a string field.

```python
def __init__(self, distance = 100, angle = 90, color = (0, 0, 0), istring = ''):
    self.distance = distance
    self.angle = angle
    self.color = color
    self.string = istring

def draw(self, xpos, ypos, scale=1.0, orientation=0):
    terp = interpreter.Interpreter()
terp.place(xpos, ypos, orientation)
```
We have a draw method that uses methods in the Interpreter to place, orient, and draw the given string. We should have accessor and mutator methods for the data fields, but we won’t take the time to do that now.

Now we can make specific shapes by *deriving* a subclass the the Shape class (it is a super class). For example a Square:

```python
class Square(Shape):
    def __init__(self, distance=100, color=(0, 0, 0)):
        Shape.__init__(self, distance=distance, angle=90, color=color, istring='−F−F−F−F')
```

We use the parentheses in the class definition to indicate that the Square is derived from a Shape. Another way of putting it is “A Square is a Shape”. The Square class *inherits* all of the methods from the Shape class. That means its symbol table has an entry for each method in Shape, and that its value is an arrow directing you to the corresponding entry in Shape’s symbol table. With one exception... Square has an `__init__` method that *overrides* the `__init__` method of the Shape class. That means the entry for `__init__` in Square’s symbol table contains an arrow pointing to this new method.

When a new Square is instantiated using the code `s = Square()`, a new object of type Square is created, and then Square’s `__init__` method is called on that new object. The body of Square. `__init__` then, in turn, calls Shape. `__init__` because we still want to run that code. Notice that the `Class.method(obj)` notation is used. This is the only way to force the overridden method to run.

When a the line `s.draw(xpos=0,ypos=0)` is executed, python goes to s’s symbol table and looks up draw. It finds a pointer to the draw entry in the Square class symbol table, which in turn contains a pointer to the draw entry in the Shape class. Finally, the draw entry in the Shape class contains a method, and that method is executed. In other words, a chain of arrows leads us from the instance object to the method. It may lead us all the way up the inheritance hierarchy, or it may stop at the first class it encounters (as it does with the `__init__` method).
## 2.3 Filling Shapes

There are a three of ways we can add fill to the shapes. Assuming we have updated the Interpreter code to turn fill on when it encounters \{ and to turn fill off when it encounters \}.

**Version 1.** Make a FilledSquare class derived from the Square class:

```python
class FilledSquare(Square):
    def __init__(self, distance=100, color=(1,0,0)):
        Square.__init__(self, distance, color)
        self.string = \"{\" + self.string + \}"  
```

In this case, we let the parent class determine the string and we just add the fill characters. Note: this works well for basic shapes, but don’t work for composite shapes (e.g. houses, which require the components to be filled individually).

**Version 2.** Make a FilledSquare class derived from the Shape class:

```python
class FilledSquare(Shape):
    def __init__(self, distance=100, color=(1,0,0)):
        Shape.__init__(self, distance=distance, color=color,  
                        istring='\{F−F−F−F−\}', angle=90 )
```

In this case, we explicitly include the braces in the string.

**Version 3.** Add a parameter to the Square’s `__init__` method:

```python
class Square(Shape):
    def __init__(self, distance=100, color=(1,0,0), filled=False):
        bstr = 4*'F−'
        if filled:
            bstr = '{' + bstr + '}'
        Shape.__init__(self, distance=distance, color=color,  
                        istring=bstr, angle=90 )
```

This method works for both basic and composite shapes, but requires an extra parameter for filled versions of shapes.

All three methods are good.