1 Administrative Topics

- The project isn’t due until tonight, but there were still several turned in last night. Let’s look at those.

2 Parameters to __init__

When we design the class, we decide who should have control over the initial values:

- The user/caller/main code must determine it. Then we include it as a parameter, and force the user to supply a value. We use that value to initialize the field.

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

- The user has the option of supplying the value. We include it as a parameter, but provide a default value, so the user can include or exclude a name argument.

  ```python
  def __init__(self, name='George '):
      self.name = name
  ```
• We mandate the initial name is some specific value (e.g. 'Milo'). We exclude it as a parameter, so the user has no ability to set the value.

```python
def __init__( self):
    self.name = 'Milo'
```

We use the first option for a Student’s name. We could use the second option for the year if we are creating a bunch of MatriculatedStudents at once, and most are going to graduate the same year. We use the third option for a Student’s grades (the list must start empty).

### 3 Lab 9

#### 3.1 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.

<table>
<thead>
<tr>
<th>George</th>
<th>black</th>
</tr>
</thead>
<tbody>
<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:

```python
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']

## 3.2 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)
    terp.color(self.color)
    terp.drawString(self.string, distance=self.distance*scale,
                     angle=self.angle)
```

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)):
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 sooner when it arrives at an overriding method (as it does with the _init__ method).

### 3.3 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−}', 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.