1 Administrative Topics

- Stephanie returns the quizzes.
- Any questions about Project 7?

2 More about Classes

Today we continue learning about how to design and implement our own classes. One of the most complicated parts of designing is deciding exactly how pro-active to be with regard to making sure our input is correct.

Let’s revisit Homework 7 for a moment. In that homework, you needed to implement a class that stored information about a Whiteboard. In that class, we had a field named “content”. As I designed the class, I had it in my mind that the value of content would be as string. When you wrote your code, you needed to decide to what extent you would enforce that. What if I added this instruction?:

require that the value of ”content” be a string
There are 3 strategies one can take:

1. passive-aggressive. Just assume the data is the right type and operate on it accordingly. If it is the wrong type, there will just be an exception (dramatic blow-up), and that might happen later. e.g.

```python
class Whiteboard:
    def __init__(self, content):
        self.content = content

    def addContent(self, new_content):
        self.content = self.content + new_content

    def printContent(self):
        print self.content

    def getContent(self):
        return self.content

    def __str__(self):
        return self.content

    def erase(self):
        self.content = ''

stephanie_whiteboard = Whiteboard(12)
stephanie_whiteboard.printContent()
stephanie_whiteboard.addContent( "and some pictures")
print stephanie_whiteboard.getContent()
```

The error shows up when we try to add content:

```bash
mb-023542:lectures srtaylor$ python lecture_21.py
12
Traceback (most recent call last):
  File "lecture_21.py", line 23, in <module>
    stephanie_whiteboard.addContent( "and some pictures")
  File "lecture_21.py", line 6, in addContent
    self.content = self.content + new_content
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```
2. compliant, fix-it-under-the-hood. do your best to massage the data before putting it into your fields (e.g. call the str function on it). This never lets the user know they did anything wrong.

```python
class Whiteboard:
    def __init__(self, content):
        # New code here - call the str function!
        self.content = str(content)

    def addContent(self, new_content):
        self.content = self.content + new_content

    def printContent(self):
        print self.content

    def getContent(self):
        return self.content

    def __str__(self):
        return self.content

    def erase(self):
        self.content = ''

stephanie_whiteboard = Whiteboard(12)
stephanie_whiteboard.printContent()
stephanie_whiteboard.addContent(" and some pictures")
print stephanie_whiteboard.getContent()
```

and there is no error

```
mb−023542:lectures srtaylor$ python lecture_21.py
12
12 and some pictures
```
3. pro-active. Explicitly test the type of the data. Throw an error if it isn’t correct. This introduces a new Python concept - it is possible for the code-writer to make the code crash on purpose. We do so by “raising an exception”:

```python
class Whiteboard:
    def __init__(self, content):
        # New code here – raise an exception!
        if type(content) != str:
            raise Exception("Type Error", "Content must be a string")
        self.content = content

    def addContent(self, new_content):
        self.content = self.content + new_content

    def printContent(self):
        print(self.content)

    def getContent(self):
        return self.content

    def __str__(self):
        return self.content

    def erase(self):
        self.content = ''

stephanie_whiteboard = Whiteboard(12)
stephanie_whiteboard.printContent()
stephanie_whiteboard.addContent(" and some pictures")
print(stephanie_whiteboard.getContent())
```

and the error happens earlier:

```
mb-023542:lectures srtaaylor$ python lecture_21.py
Traceback (most recent call last):
  File "lecture_21.py", line 23, in <module>
    stephanie_whiteboard = Whiteboard(12)
  File "lecture_21.py", line 4, in __init__
    raise Exception("Type Error", "Content must be a string")
Exception: ("Type Error", "Content must be a string")
```

For the most part, we design our code to use the passive-aggressive approach.
3 Graphics Objects

This week, we will transition away from population dynamics to the movements of physical objects (think “gravity” and “friction”). Your projects will involve animations of balls bouncing against floors and walls. To accomplish this, we will be using a Python package written by John Zelle (the author of the recommended text book). Today, my goal is to introduce this to you and to talk about how to design classes that will draw more complicated objects than he supplies in his package.

The package has several classes that we will use:

<table>
<thead>
<tr>
<th>Object Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Window</td>
<td>an object that allows you to put graphics into a window</td>
</tr>
<tr>
<td>Point</td>
<td>an object that allows you to specify the x-y coordinates of a graphics object</td>
</tr>
<tr>
<td>Circle</td>
<td>a circular graphics object – lets you draw a circle</td>
</tr>
<tr>
<td>Rectangle</td>
<td>a rectangular graphics object – lets you draw a rectangle</td>
</tr>
<tr>
<td>Oval</td>
<td>an elliptic graphics object – lets you draw an oval</td>
</tr>
<tr>
<td>Line</td>
<td>a line graphics object</td>
</tr>
<tr>
<td>Polygon</td>
<td>a graphics object that is constructed by providing all the vertices</td>
</tr>
</tbody>
</table>

The Graphwin methods we are interested in are:

- `win.getMouse()` – waits for a mouse click
- `win.close()` – closes the window
- `win.checkMouse()` – True if a mouse has recently clicked on the window

Zelle has organized the methods for the shape functions well. There are some methods (and functions) specific to each shape. Most notably, the functions used to create each object are unique:

- `Circle(centerpt, radius)`. centerpt is a Point specifying the location of the center of the circle. radius is a number.
• Rectangle( pt1, pt2 ). Each pt is a Point specifying opposite corners of
  the rectangle (e.g. upper left and lower right)

• Oval( pt1, pt2 ). Each pt is a Point specifying opposite corners of the
  rectangle that bounds the oval you want to draw

• Line( pt1, pt2 ). Each pt is an endpoint of the line segment.

• Polygon( pt1, pt2, pt3, ...) or Polygon([pt1, pt2, pt3, ...]). Each pt
  is a vertex of a polygon.

There are several methods that are general - -they can be used on any of
the shape objects. Suppose that obj is a Circle, Rectangle, Oval, Line, or
Polygon. Then, the follow methods are defined:

• obj.draw(win) – places obj in the window (win) (which must be a
  GraphWin)

• obj.move(dx, dy) – moves the obj by a given amount

• obj.setOutline(clr) – sets the outline color

• obj.setWidth(width) – sets the width of the outline (in pixels)

• obj.setFill(clr) – sets the fill color

For the color-setting methods, clr must be either a string with a color name
(e.g. ‘green’, ‘blue’, etc.) or the output of the function color_rgb (e.g.
graphics.color_rgb( 255, 150, 20 )).

To draw a simple circle, you use code like this:

```python
# Make a window big enough to hold the scene
w = 500
h = 500
win = graphics.GraphWin(“Demo”, w, h)

# Make a circle centered at 250,250 with radius 50
center = graphics.Point(250, 250)
c = graphics.Circle(center, 50)

c.draw(win)  # i.e. put it in the window
```
Drawing a Tree

Let’s design a class to draw a very simplified tree (a circle on top of a rectangle). We want it to be initialized with similar information as the simple objects (namely, position and size information). In our case, we will make it simpler - the caller can supply numbers instead of Point objects. We also need a draw method that will take a GraphWin object as a parameter and draw the objects.

What do we need for local variable and what needs to be a field (i.e. what needs to belong in the object’s symbol table)? In general, something should be stored in a field if we need to access that data in more than one method call (i.e. you need the data to stay there when you call the same method twice or you need two different methods to access it).

For this class, I propose we use two fields - one for the circle (the leafy part) and one for the rectangle (the trunk).

class Tree:
    def __init__(self, topCenterX, topCenterY, radius):
        self.leafPart = gr.Circle(gr.Point(topCenterX, topCenterY), radius)
        self.leafPart.setFill('green')
        self.trunk = gr.Rectangle(gr.Point(topCenterX−radius/8.0, topCenterY + radius),
                                  gr.Point(topCenterX+radius/8.0, topCenterY + radius*3))
        self.trunk.setFill('brown')

    def draw(self, win):
        self.trunk.draw(win)
        self.leafPart.draw(win)

win = gr.GraphWin("Demo", 500, 500)

c.draw(win)

# Keep the window open for awhile
win.getMouse()  # Pause to view result
win.close()  # Close window when done
t = Tree( 250, 200, 50 )
t.draw( win )
win.getMouse()
win.close()

Let’s also think about the symbol tables. After the code has finished (but before everything is destroyed, we have 3 symbol tables that we are interested in (I am for now ignoring the symbol tables for the graphics module and the graphics objects).

The main symbol table is:

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>graphics</td>
<td>→ &lt;module&gt;</td>
</tr>
<tr>
<td>win</td>
<td>→ &lt;GraphWin&gt;</td>
</tr>
<tr>
<td>Tree</td>
<td>→ &lt;Class Tree&gt;</td>
</tr>
<tr>
<td>t</td>
<td>→ &lt;Tree&gt;</td>
</tr>
</tbody>
</table>

The Tree class’s symbol table has just the methods:

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong></td>
<td>→ &lt;function&gt;</td>
</tr>
<tr>
<td>draw</td>
<td>→ &lt;function&gt;</td>
</tr>
</tbody>
</table>

The Tree object’s symbol table has the methods and the data:

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong></td>
<td>→ &lt;Tree class table&gt;</td>
</tr>
<tr>
<td>draw</td>
<td>→ &lt;Tree class table&gt;</td>
</tr>
<tr>
<td>leafPart</td>
<td>→ &lt;Circle&gt;</td>
</tr>
<tr>
<td>trunk</td>
<td>→ &lt;Rectangle&gt;</td>
</tr>
</tbody>
</table>

### An Alternate Design for the Tree

I do want to mention an alternate (but un-favored) design. Instead of making the Graphics objects in the init method, save the position information in fields, and make the graphics objects in the draw method. In this case, we don’t make fields for the Graphics objects methods. This is totally fine if we don’t need to do anything more interesting with the tree. But if we want to change the color of the tree (e.g. show the 4 seasons) or if we want to move it (because it is an Ent), then we need to save the Graphics Objects
in a field. If we are going to do that, we might as well do it in the init method. Otherwise, we have the rule that draw must be called before any color-changing code is called. (Think about why.)

```python
class Tree:
    def __init__(self, topCenterX, topCenterY, radius):
        self.topCenterX = topCenterX
        self.topCenterY = topCenterY
        self.radius = radius

    def draw(self, win):
        leafPart = gr.Circle(gr.Point(self.topCenterX, self.topCenterY),
                             self.radius)
        leafPart.setFill('green')
        trunk = gr.Rectangle(gr.Point(self.topCenterX - self.radius/8.0,
                                       self.topCenterY + self.radius),
                              gr.Point(self.topCenterX + self.radius/8.0,
                                       self.topCenterY + self.radius*3))
        trunk.setFill('brown')
        trunk.draw(win)
        leafPart.draw(win)

win = gr.GraphWin("Demo", 500, 500)
t = Tree(250, 200, 50)
t.draw(win)
win.getMouse()
win.close()
```