1 Administrative Topics

- Show some cool projects.

2 Default parameters and keyword arguments

When you are defining a function, you identify parameters, and it is possible to provide a default value for that parameter. When you call that function, you can choose to provide a value for that parameter if you want to.

```python
def sayHi(name='You '):
    print("Hey, ", name)

sayHi()
sayHi("Stephanie")
sayHi("You")
```

will print out

Hey, You
Hey, Stephanie
Hey, You

When you call a function, the value you provide as input to the function is called an argument. In the above case Stephanie is an argument. There are two ways to provide arguments. One is using positional notation. This
is what we have been using so far. The first argument is the value for the first parameter. The second argument is the value for the second parameter. And so on and so forth. But there is another way to do it, using keyword notation. In this case, you use the name of the parameter and the value of the argument together in something that looks like an assignment statement.

\[
\text{sayHi( name = "George" )}
\]

For functions with many parameters, you can provide the first set of arguments by position and the last set by keyword, but once you start using keywords, you can’t stop.

In the next example, we examine code that shows the power of both default parameter values and keyword arguments. Consider the following function which has a lot of parameters, some of which have default values:

```python
def circle(x, y, scale, fill=True, fcolor = 'black', pcolor = 'black', psize = 1):
    # lots of code here.
```

Then the following calls are legal:

- `circle( 100, 0, 1.0, psize = 3 )`
- `circle( x=100, y=0, scale=1.0, pcolor='red' )`
- `circle( 100, 0, 1.0 )`
- `circle( scale = 1.0, x = 1, y = 2 )`

And the following calls are ILLEGAL:

1. `circle( 100, 0, 1, pcolor='red', 1 )`
2. `circle( 100, 0 )`
3. `circle( 100, 0, fill=False )`

Line 1 is illegal because it uses a positional argument (1) after a keyword argument.

Line 2 is illegal because it fails to provide a value for `scale`. Since `scale` does not have a default value, the caller must supply an argument for it. Line 3 is illegal for the same reason.

3 How do objects fit into our memory model?

Let’s look at a simple image manipulation example to see how we represent objects in memory. We will step through code similar to that on Mon-
day. We use the display function from Monday, but write a new function 
blackDotPixmap to put change the color of four neighboring pixels to black. 
(Note: This isn’t a good function after which to model yours - it is an easy 
function to step through!).

So, the code begin by importing the graphics module, then defines the display 
function, then adds the new function, and finally has the main code:

```python
import graphics

# display Pixmap pm in a window with the given title
def displayPixmap(pm, title):
    w = pm.getWidth()
    h = pm.getHeight()
    win = graphics.GraphWin(title, w, h)
    img = graphics.Image(graphics.Point(w/2.0, h/2.0), pm)
    img.draw(win)
    win.getMouse()  # pause for a click in the window
    win.close()

# Place a 2x2 black dot on the image at (250,50)
def blackDotPixmap(pm):
    pm.setPixel(250, 50, (0,0,0))
    pm.setPixel(250, 51, (0,0,0))
    pm.setPixel(251, 50, (0,0,0))
    pm.setPixel(251, 51, (0,0,0))

# main
if __name__ == '__main__':
    pm = graphics.Pixmap('georgeMiloDrumTrain.ppm')
    blackDotPixmap(pm)
    displayPixmap(pm, "The boys are cute!")
```
Let’s step through the code. By the time we get to line 20, there are three entries in the symbol table:

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>graphics</td>
<td>&lt;module&gt;</td>
</tr>
<tr>
<td>displayPixmap</td>
<td>&lt;fcn&gt;</td>
</tr>
<tr>
<td>blackDotPixmap</td>
<td>&lt;fcn&gt;</td>
</tr>
</tbody>
</table>
Line 21 statement is an assignment statement. We evaluate the right hand side (by calling the Pixmap function in the graphics module) and a Pixmap is returned.
We then update the symbol table by adding an entry whose name is pm and whose value is an arrow (in the table) to a Pixmap object (outside the table):
Next, we call `blackDotPixmap` (line 22).

A new symbol table is created. There is an entry automatically added for the parameter. The name is `pm` and the value is an arrow to the same Pixmap object:
Then, line 15 is executed and pixel (250,50) is colored black:
Now line 16 is executed and pixel (250, 51) is colored black.
Now line 17 is executed and pixel (251, 50) is colored black.
Now line 18 is executed and pixel (251, 51) is colored black.
Since both the main code and blackDotPixmap are pointing to the same object, there is no need for blackDotPixmap to return anything. It is simply manipulating the same pixels main has access to. This is helpful because it cuts down on the amount of memory required and on the time needed to copy data from one memory location to another. So, blackDotPixmap returns None and its symbol table goes away. Control returns to the main code and it is now pointing to the updated Pixmap:

It is this updated Pixmap that is passed in to displayPixmap.

Done with this part of the demo.
What if we don’t want to alter the original pixmap? What if we want to keep it around so that we can apply a different manipulation function? One option is to create two Pixmap objects from the same file. Another is to use the clone method. `pm.clone()` will return a new copy of the Pixmap contained in `pm`.

Suppose we add two lines to the main code from above

```
newpm = pm.clone()
displayPixmap(pm, "Really, I mean it!")
```

The right hand side of the assignment statement evaluates to a pointer to a new Pixmap
And the symbol table is updated to point to both the original Pixmap and a clone:

![Symbol Table Diagram]

The final line displays the Pixmap. It has no effect on the main symbol table.