1 A new way to make strings

Here is another syntax for creating strings that makes the code a bit easier to read. It uses the % to perform string formatting. Let me start by explaining an example

\[ s = '(%f)F(90)+(%f)F(90)+' % (width, height) \]

This creates a string that draws two sides of a rectangle. We can think of this as being divided into three parts: a format string, %, and the tuple containing the values. The format string tells us what the final string should look like, but uses place-holders for some of the parts. In particular, each '%f' string is a place-holder indicating that a number will appear there. Notice that there are two place-holders in the format string and two values in the final tuple (with and height). The str(width) replaces the first place-holder and str(height) replaces the second place-holder. So, if width is 40.0 and height is 45.0, then the value of \( s \) will be '(40.000000)F(90)+(45.000000)F(90)+'.

You may use this notation within parentheses if you want to concatenate its result with other strings. For example, the code to create a filled rectangle string is this:

\[ bw = '{' + 2*( '%f)F(90)+(%f)F(90)+' % (width, height))+'}' \]
2 Drawing trees using recursion

Let’s draw a binary tree (using code developed for Project 11). We will require that the binary tree be drawn in exactly one orientation – with the top cube at the top, and the left cubes to the left, and the right cubes to the right. This will make it much more straight-forward to compute the locations of the subtrees.

Let’s use recursion. To do this, we will add a drawBinaryTree method to the ThreeDSquare class. It will draw the current cube and then make two recursive calls to draw the left and right subtrees. The base case, which is more of a stopping case, will simply return from the function before anything is drawn. We will use a parameter levels to indicate how many levels the tree should have. The base case, is that levels is zero.

To compute the location of the subtrees, we need to use the width and height of the shape. To use that information, we must store it in the object. So we update the __init__ function to add those fields to the object.

Here is the code for the class:

```python
class ThreeDBox( sh.Shape ):
    def __init__( self, width, height, depth, color=(0,0,0) ):
        back = '(%f)F+(%f)F+(%f)F+(%f)F+' % (width, height, width, height)
        l1 = 'ˆ(%f)F&' % (depth)
        sh.Shape.__init__( self, distance=1, angle=90, color=color,
                          istring = back + l1 + front )
        self.box_width = width
        self.box_height = height
        self.box_depth = depth

    def drawBinaryTree( self, xpos, ypos, zpos, levels ):
        if levels == 0:
            return

        # draw the head/root node
        self.draw( xpos=xpos, ypos=ypos, zpos=zpos )

        # draw the left subtree
        self.drawBinaryTree( xpos=xpos-self.box_width*0.5,
```

2
ypos = ypos − self.box_height, zpos=zpos,
levels=levels−1)

# draw the left subtree
self.drawBinaryTree( xpos = xpos+self.box_width*0.5,
ypos = ypos − self.box_height, zpos=zpos,
levels=levels−1)
2.1 Scaling

To make cubes at lower levels smaller than cubes at higher levels, we can add a scaling factor. First, we use the scale to determine the size of the head node, then we shrink it to 80% for the next level.

```python
def drawBinaryTree(self, xpos, ypos, zpos, levels, scale=1.0):
    if levels == 0:
        return

    # draw the head/root node
    self.draw(xpos=xpos, ypos=ypos, zpos=zpos, scale=scale)

    # draw the left subtree
    self.drawBinaryTree(xpos=xpos-self.box_width*0.5,
                        ypos=ypos-self.box_height, zpos=zpos,
                        levels=levels-1, scale=scale*0.8)

    # draw the left subtree
    self.drawBinaryTree(xpos=xpos+self.box_width*0.5,
                        ypos=ypos-self.box_height, zpos=zpos,
                        levels=levels-1, scale=scale*0.8)
```

Note that in order to truly scale it, we should scale the offsets. But we didn’t have time to do that computation in class.
2.2 Quad Tree

We didn’t cover this in class, we can give more depth to the tree by adding branches in the front and behind the current tree. Let’s call that a quad tree. Here is the code for it:

```python
# draw a quad tree with the head node at (xpos, ypos, zpos)
# with levels levels
def drawQuadTree(self, xpos, ypos, zpos, levels, scale=1.0):
    if levels == 0:
        return

    # draw the head node
    self.draw(xpos=xpos, ypos=ypos, zpos=zpos, scale=scale)

    # draw left subtree
    self.drawQuadTree(xpos=xpos - self.cube_width / 2.0 * scale,
                      ypos=ypos - self.cube_height / 2.0 * scale,
                      zpos=zpos, levels=levels - 1, scale=1 * scale)

    # draw the right subtree
    self.drawQuadTree(xpos=xpos + self.cube_width / 2.0 * scale,
                      ypos=ypos - self.cube_height / 2.0 * scale,
                      zpos=zpos, levels=levels - 1, scale=1 * scale)

    # draw front subtree
    self.drawQuadTree(xpos=xpos,
                      ypos=ypos - self.cube_height / 2.0 * scale,
                      zpos=zpos + self.cube_depth / 2.0 * scale,
                      levels=levels - 1, scale=1 * scale)

    # draw the back subtree
    self.drawQuadTree(xpos=xpos,
                      ypos=ypos - self.cube_height / 2.0 * scale,
                      zpos=zpos - self.cube_depth / 2.0 * scale,
                      levels=levels - 1, scale=1 * scale)
```