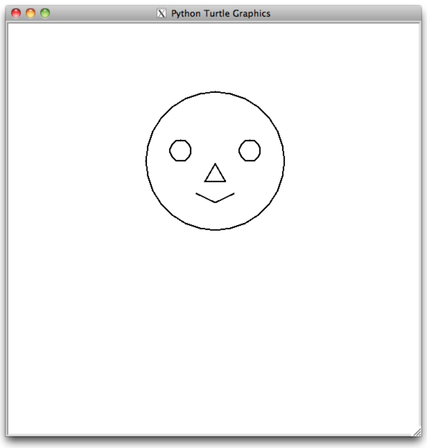
Problem-Based Intro. to Computer Science
Fancy Faces (Lecture Part 2)

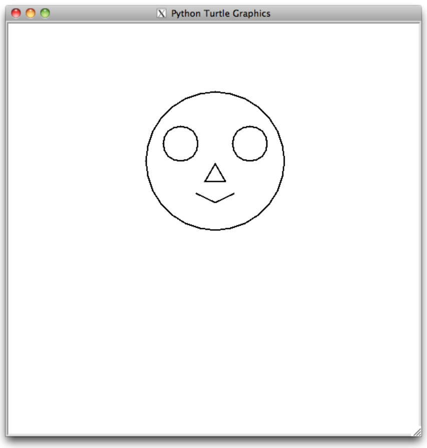
09/08/2012

Problem

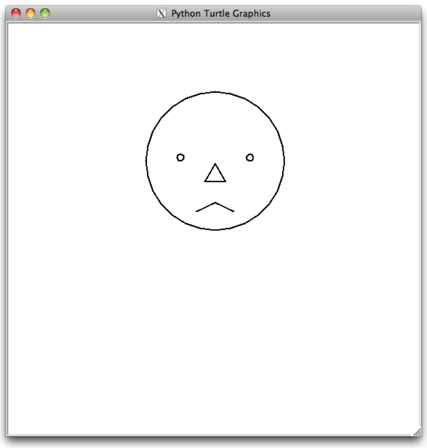
After last week's face-drawing triumph, we have been contracted by the ACME Stick Figure Company to produce faces for their upcoming line of stick figures. While marketing confirms that our "traditional" happy face has broad appeal, select segments of our target demographic might prefer larger or smaller eyes and might prefer a frown. In order to keep production costs down, we need a *general* program that can produce a variety of faces, rather than writing a *specific* program for each face.



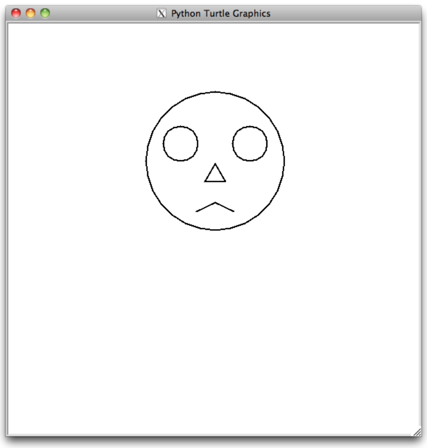
```
mouthShape == "smile"
eyeRadius == 15
```



```
mouthShape == "smile"
eyeRadius == 25
```



```
mouthShape == "frown"
eyeRadius == 5
```



```
mouthShape == "frown"
eyeRadius == 25
```

Problem Analysis and Solution Design

Development

What are some of the similarities and differences amongst the variety of faces?

- similar: shape and size of the border
- similar: location of center of mouth
- different: shape of mouth ("up" for smile; "down" for frown)
- similar: location, shape, and size of nose
- similar: shape of eyes and location of their bottoms
- different: size (radius) of each eye

The many similarities suggest that a small collection of functions can be reused to produce a variety of faces.

The differences suggest *parameterizing* some of these functions by *arguments* in order to draw the specific features.

For example, our solution from last week had a function to draw an eye. In our solution for this week, we will continue to have a function to draw an eye, but that function will take an argument specifying the radius of the eye to draw. We have already used many functions that take arguments (e.g., `turtle.circle`, `turtle.forward`); now we are defining our own functions that take arguments.

The differences also suggest *choosing* among code blocks based on *conditions* in order to draw the specific features.

For example, our solution from last week had a function to draw a (smiling) mouth. In our solution for this week, we will continue to have a function to draw a smiling mouth, but that function will take an argument specifying the mouth shape (`"smile"` or `"frown"`) and that function will choose to execute one block of code when the mouth shape is `"smile"`, another block of code when the mouth shape is `"frown"`, and yet another block of code when the mouth shape is anything else. This is clearly a job for the if statements already discussed.

Finally, we will want our program to prompt the user for the mouth shape and eye radius. All input from the user starts of as a *string*; while this is acceptable for the mouth shape, we will need to convert to an *integer* for the eye radius.

Algorithm Outlines

We need to partition our approach into several algorithms. Here is an outline of how this program could be structured:

- Main Function
  - prompt the user to input the mouth shape
  - prompt the user to input the eye radius
  - initialize the drawing canvas
  - draw the face using the given shape and radius
  - tell the user to hit *Enter* when done viewing the face
  - close the drawing canvas
- Function to draw the whole face with parameters *mouthShape* and *eyeRadius*
  - draw the border
  - draw the mouth using the given shape
  - draw the nose
  - draw the eyes using the given radius
- Function to draw the border
  - Draw a circle with radius 100
- Function to draw the mouth with a parameter to specify *mouthShape*
  - If *mouthShape* is ”smile” draw a smile
  - Otherwise if *mouthShape* is ”frown” draw a frown
  - Otherwise draw a ”grimace”
- Function to draw the nose
  - move to the center of the face
  - draw an equilateral triangle
  - move back to the bottom of face (the starting point)
- Function to draw the eyes with a parameter to specify *eyeRadius*
  - move to the bottom of one eye
  - call drawEye function with the eye radius
  - move to the bottom of the other eye
  - call drawEye function with the eye radius
  - move to the bottom of the face
- Function to draw an eye with a parameter to specify *eyeRadius*

Next we want to check this design by *executing on paper*. For this we can draw pictures to trace through what the computer would do to execute a program that follows this plan.

Execution Diagram

We can visualize the execution of the `drawFancyFace` function with an *execution diagram*. This picture shows that, when one function calls another function, the *calling function* remains active until the *callee* finishes executing.

```
drawFancyFace( mouthShape := "smile", eyeRadius := 15 )
    drawBorder()
        do nothing # already at bottom of face
        draw a circle of radius 100
        do nothing # already at bottom of face
    drawMouth( mouthShape := "smile" )
        move to center of mouth
        # mouthShape == "smile"
        draw a line up-left of length 30
        move to center of mouth
        draw a line up-right of length 30
        move to center of mouth
        move to bottom of face
    drawNose()
        move to center of face
        draw a triangle of side-length 30
        move to bottom of face
    drawEyes( eyeRadius := 15 )
        move to bottom of left eye
        drawEye(eyeRadius := 15)
            draw a circle of radius eyeRadius
        move to bottom of right eye
        drawEye(eyeRadius := 15)
            draw a circle of radius eyeRadius
        move to bottom of face
```

Implementation

See the accompanying file `fancy_face.py` for a solution that also includes *test functions*.

Testing (Test Cases, Procedures, etc.)

We should be sure that our program is able to produce the variety of faces from the first page, and we should also test our program with some new values.

- `mouthShape == "smile"`, `eyeRadius == 15`
- `mouthShape == "smile"`, `eyeRadius == 25`
- `mouthShape == "frown"`, `eyeRadius == 5`
- `mouthShape == "frown"`, `eyeRadius == 25`
- `mouthShape == "smile"`, `eyeRadius == 35`
- `mouthShape == "frown"`, `eyeRadius == 15`
- `mouthShape == "zzz"`, `eyeRadius == 20`

Although we won't require programs to gracefully handle bad input at this stage, it is illustrative to see how the program behaves in response to bad input:

- `mouthShape = "smile"`, `eyeRadius = "ten"`
- `mouthShape = "smile"`, `eyeRadius = 10.0`