Homework 2

Software Design **Allen B. Downey**
Fall 2004

Due: Thursday 16 September.
The reading for this assignment is Chapters 4–6 of *How to think...*.

2.1 Encapsulation and Generalization

1. We have already seen two ways to draw a square. Here is a third:

   ```
   bob = Turtle(world)
   for i in range(4):
       fd(bob, 90)
       lt(bob)
   ```

   Create a file named `polygon.py` and type in this code. Run it and confirm that it draws a square[1].

2. Encapsulate this code in a function called `square` and write a line of code that invokes it. HINT: when you change your program, you should be able to run the new version without restarting `World.py`.

3. Generalize the function so that it takes a turtle and a distance, `dist`, as parameters.

4. Generalize the function so that it takes an additional parameter, `n`, and draws an n-sided regular polygon. Hint: the angles of an n-sided regular polygon are $360/n$ degrees. You might want to change the name of the function to `polygon`.

5. Write a function called `circle` that takes a turtle, `t`, and radius, `r`, as parameters and that draws an approximate circle by invoking `polygon` with an appropriate distance and number of sides. Test your function with a range of values of `r`.

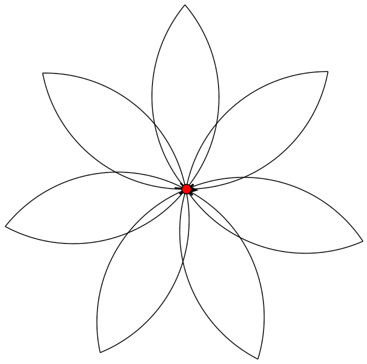
   Hint: figure out the circumference of the circle and make sure that `dist * n = circumference`.

6. Make a more general version of `circle` called `arc` that takes an additional parameter `theta`, which determines what fraction of a circle to draw. `theta` is in units of degrees, so when `theta=360`, `arc` should draw a complete circle.

7. Chances are that when you started to write `arc` you realized that you weren't able to reuse `polygon`, even though `arc` and `polygon` are very similar. Maybe you ended up doing "design by copy and paste." Although this solution is not as pretty as we might like, it is good enough for this homework, so you can go to the next step if you want to. But if you are feeling inspired, you might want to refactor the program at this point so that `arc`, `circle`, and `polygon` all invoke a single (very general) function.

[1] You might find it helpful to put your code in a file named `turtle_code.py`, because that's what `World.py` runs by default. But when you are done, please put your solution in `polygon.py`.

8. Write an appropriately general function called `flower` that can draw flowers like this:



9. As usual, you should look over your code to see if there are opportunities to make it better, and add comments that explain what each function does, and that explain anything that is not obvious. Add a comment at the beginning that has your name in it. Print a copy of the code and, if you make a flower you are proud of, a copy of the output (by hitting `Print Canvas` and then typing `lpr gui.eps`).

2.2 Wanderers

1. Download and run `Wanderer.py`:

   ```
   wget http://wb/sd/code/Wanderer.py
   python Wanderer.py
   ```

   Print a copy of the code (`a2ps -1 Wanderer.py`) and read it over. It uses some features we haven't studied yet, but you should be able to understand enough to modify it.

2. The `distance` function is not implemented correctly. You should change it so that it actually computes the distance from the given turtle, `self`, to the origin.

3. One problem with Wanderers is that they eventually go off the screen, and they usually don't come back. One way you might try to solve this problem is to modify `step` so that if a Wanderer is too far from the origin, it turns around.

   Make this change and run the program again. Does it work? (You might want to draw a circle at the appropriate radius).

4. Think of a way to keep the Wanderers in bounds. Ideally your algorithm should be simple (easy to implement), effective (it should keep the Wanderers in bounds), and gentle (it shouldn't cause the Wanderers to behave too erratically near the boundary).

5. Prepare and print your code as usual.

2.3 More fun

The following exercises are optional. You don't have to turn them in, but if you do something cool, I would be happy to see it.

1. Modify `step` so that Wanderers "orbit;" that is, so they try to stay the same distance from the origin.

2. Modify `step` so that Wanderers interact with each other. For example, you could have them run after each other, or avoid each other, or bounce off each other when they collide. `World` objects have an attribute called `animals` that is a list of the animals that are currently in the world.

3. Create different kinds of Turtles (not just Wanderers) that have different definitions of `step`. You could have one kind that runs after other turtles, and another kind that runs away.

4. Make Turtles behave like a paintbrush.

   (a) Add a Turtle attribute named `thickness` that controls the thickness of the line.

   (b) Modify `World.py` so that when a Turtle moves, the thickness of the line is the turtle's `thickness`.

   (c) Modify `step` so that it reduces the thickness each time the turtle moves.

   (d) Write a function called `dip` that resets the `thickness`.

   (e) Add a dip button to the Turtle Control Panel

   (f) Use Turtles to make calligraphy.