Black Problem 1 (lab): Python Turtles

Copied from:
https://www.cs.hmc.edu/twiki/bin/view/CS5/PrettyPicturesBlack on 3/22/2017

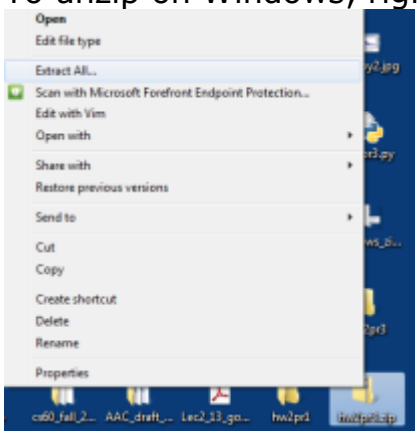
[35 points; individual or pair] filename: `hw2pr1.zip`

For this problem, you'll submit a ***zipped folder*** of files.

It is easiest to start with this folder: **hw2pr1.zip**

Here's how to get started:

- Download the above folder.
- Unzip that folder. ***Please do NOT use external tools for this.*** Rather:
  - To unzip on MacOS, simply double-click the `hw2pr1.zip` file you've downloaded
  - To unzip on Windows, right-click and choose *Extract All…*, e.g.,



- Navigate into the `hw2pr1` folder that results.
  - Be sure you've unzipped the folder; Windows will let you navigate a zipped folder, but will not let you run Python programs from within it
- You'll see an image file named `poly_example.png` (a septagon)
- Right-click on the `hw2pr1.py` file to open it with IDLE.

When it's time to submit, you'll re-zip the `hw2pr1` folder (instructions below) to include

- Your updated `hw2pr1.py` file, with additional functions
- At least three new screenshot images
  - One each of your spiral, your svtree, and your snowflake

- For extra credit, we invite you to include other images and functions of your own design…

Using turtle graphics

Canopy users If you're using Canopy, the turtle graphics should work for you. If you're using another specially-packaged Python version (such as Idle), or if you have difficulties, you may need to use a plain-Python install (i.e., the command line) for turtle to work. ***Alternatively***, you can use one of these two online Python interpreters that support turtle-graphics. Each has example code (linked at right) to get you started, as well:

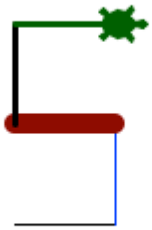
- turtle within interactive Python Here is example code for this site.
- turtle within skulpt's in-browser Python Here is example code for this site.

In a *plain* install of Python 2.7.x, turtle works pretty well just "out of the box."

You can check this, e.g., by typing these commands:

```
>>> ================================ RESTART ======
==========================
>>>
>>> from turtle import *
>>> forward(50)
>>> color("blue")
>>> left(90)
>>> forward(50)
>>> width(10)
>>> color("darkred")
>>> left(90)
>>> forward(50)
>>> right(90)
>>> width(3)
>>> color("black")
>>> forward(50)
>>> shape('turtle')
>>> color("darkgreen")
>>> right(90)
>>> forward(50)
```

If everything works, you should see the following picture drawn in a new window:



Closing the turtle window

To close the turtle window, type ***at the prompt***

```
>>> done()
```

…then you can click on the red window-closing button of the turtle window.

Warning! ***Don't include*** `done()` in your code or your functions!

Turtle graphics not working well?

An alternative is to try one of the two online turtle-graphics options, linked above.

Python's turtle graphics can work differently on different systems. If it's working poorly for you (stalling/freezing), it's because IDLE and turtle are both trying to control Python's graphics and are getting in each other's way.

In such cases we have had success with running turtle functions from the command-line, rather than from IDLE.This page describes how to run Python from the command-line, both for Windows machines and Macs.

Trying some more turtle commands

Here are some additional examples, highlighting a few more commands, such as `up`, `down`, `xcor`, and `ycor`:

```
>>> from turtle import *          # needed to access the turtle
commands
>>> forward(100)     <-- turtle goes forward 100 steps
>>> right(90)        <-- turtle turns right 90 degrees
>>> up()             <-- turtle lifts its pen up off of the paper
>>> forward(100)     <-- turtle goes forward 100 steps
>>> down()           <-- turtle puts its pen down on the paper
>>> color("red")     <-- turtle uses red pen
>>> circle(100)      <-- turtle draws circle of radius 100
>>> color("blue")    <-- turtle changes to blue pen
>>> forward(50)      <-- turtle moves forward 50 steps
>>> xcor()           <-- turtle returns its current x-coordinate
>>> ycor()           <-- turtle returns its current y-coordinate
```

For the complete set of `turtle` commands, go to the official Python turtle page.

The `spiral` function (10 points)

Write a function

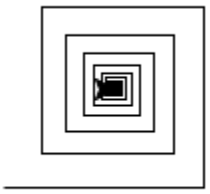
```
spiral(initialLength, angle, multiplier)
```

that uses the `turtle` drawing functions to create a spiral that has a first segment of length `initialLength` and whose neighboring segments form angles of `angle` degrees. The `multiplier` will be a float that will indicate how each segment changes in size from the previous one. For example, with a `multiplier` of `0.5` each side in the spiral should be *half* the length of the previous side.

Base cases!

The spiral should stop drawing when it has reached a side length of less than 1 pixel or greater than 1000 pixels.

Here is a screenshot from the call `spiral(100, 90, 0.9)`



Remember that you may need to type

```
>>> done()
```

at the prompt (not in your code), before closing the window.

Time to take a screenshot!

Once your `spiral` function is working, choose a color, line width, turtle shape, etc., and then render your chosen spiral. (Make sure it's different than the one above!)

Then, take a screenshot (whole-screen or just the turtle window):

- **Windows**: the *snipping tool* in *Accessories* works well. Here is a link describing Windows screenshots
- **Mac**: command-shift-4 provides a "screenshot cursor" that will grab a rectangle and save it to the desktop. Here is a link describing MacOSX screenshots

Rename your screenshot as `spiral.png` (or `spiral.jpg` or any format your machine supports).

Place your screenshot into the `hw2pr1` folder (so it will be there when you submit).

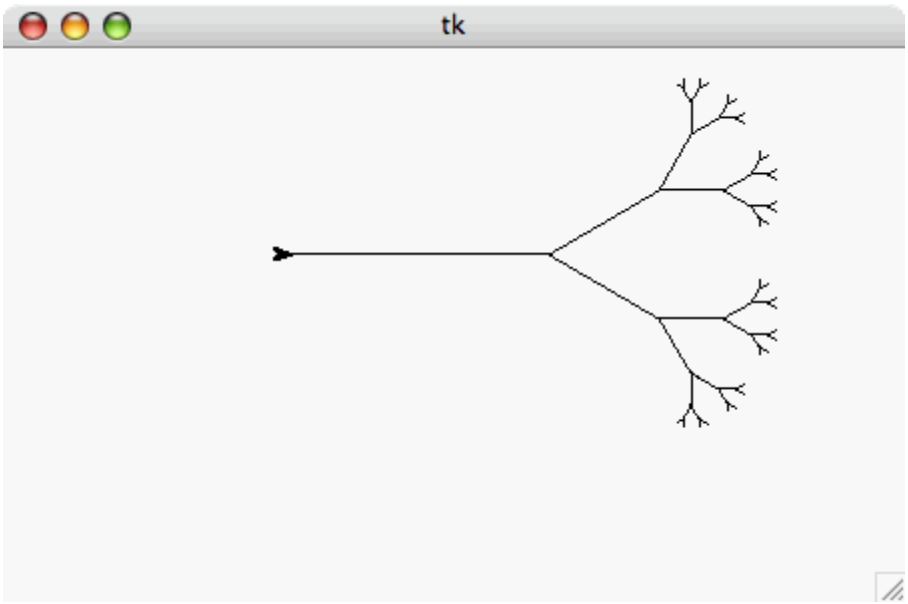
The `svtree` function (15 points)

Side-view tree

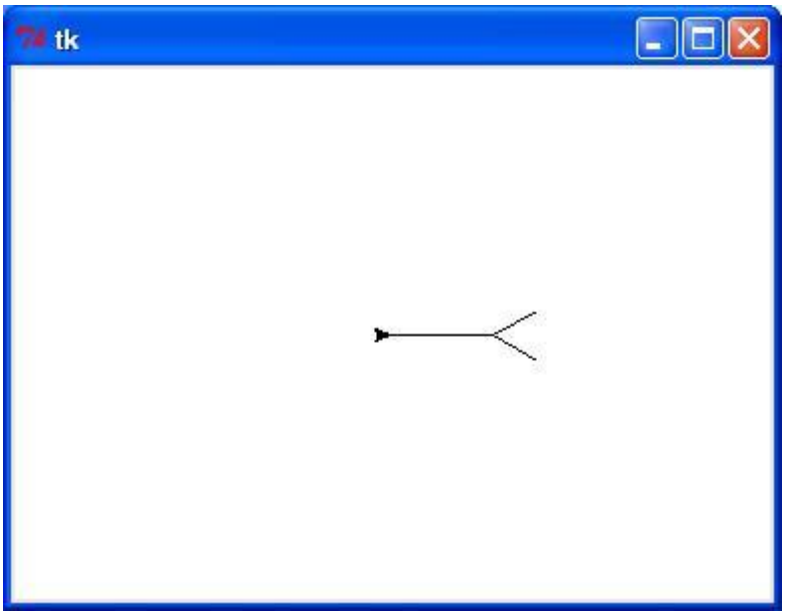
The idea here is to create a function that draws the *side view* of a tree:

```
svtree(trunklength, levels)
```

Here is an example of the output from my function when `svtree(128, 6)` is run:



and another example of the output when `svtree(50, 2)` is run:



Note that these are **really** side view! Calling `left(90)` before the call to `svtree` will yield a more traditional pose.

Hints

The key to happiness with recursive drawing is this: ***the pen must be back at the start (root) of the tree at the end of the function call!*** Furthermore, the turtle must be facing in the same direction as before. That way, each portion of the recursion "takes care of itself" relative to the other parts of the image.

One thing **not** to worry about is the number of branches (anything greater than 1 is OK), the exact angle of branching, the amount of reduction of the `trunklength` in sub-branches, etc. Design your own tree by making aesthetic choices for each of these.

Take a screenshot

Add some artistic embellishments of your own (something beyond the above images, to be sure…).

Then, take a screenshot and rename it `tree.png` or `tree.jpg` or any format. Move that image file into the `hw2pr1` folder so that it will be there when you submit!

The `snowflake` function (10 points)

Note that this is worth only 10 points. This not because it's easier than the previous one, but because it's considerably more subtle. If you get it working, great! If not, this scoring is meant to keep it in perspective.

The Koch Snowflake is three identical sides—it's the sides themselves that are defined recursively. Because of this, we provide the overall `snowflake` function:

```
def snowflake(sidelength, levels):
    """ fractal snowflake function
          sidelength: pixels in the largest-scale triangle side
          levels: the number of recursive levels in each side
    """
    flakeside(sidelength, levels)
    left(120)
    flakeside(sidelength, levels)
    left(120)
    flakeside(sidelength, levels)
    left(120)
```

You're invited to copy this function into your `hw2pr1.py` file.

Then, *your task* is to implement the `flakeside(sidelength, levels)` function that will complete the definition of the Koch Snowflake. More information on that fractal curve ishere, among other places on the web.

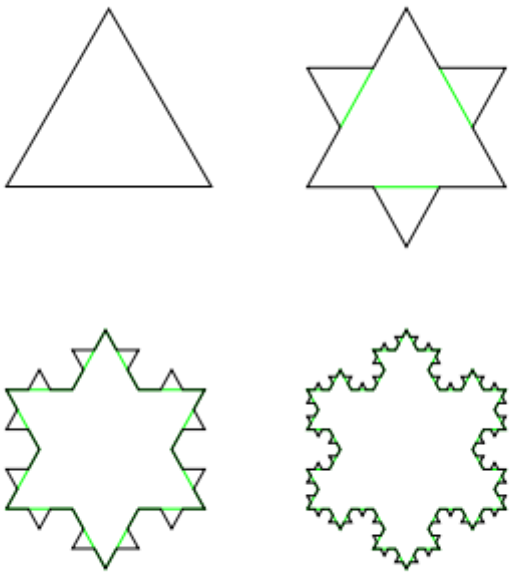
A base-case Koch snowflake side is simply a straight line of length `sidelength`.

Each recursive level replaces the *middle third* of the snowflake's side with a "bump," i.e., two sides that would be part of a one-third-scale equilateral triangle.

As noted, **all** of the recursion occurs in `flakeside`. Remember that

- if `levels` is zero, then `flakeside` should produce a single segment (base case!)
- otherwise, `flakeside` needs to call itself ***four*** times!
- Remember that `flakeside` is only creating one of the three sides of the snowflake!

Here are images of four steps of the overall Koch curve's progression:



Take a screenshot!

Add some artistic choices of your own and then take a screenshot of your favorite snowflake. Rename it `snowflake.png` or `snowflake.jpg` or any format.

Move that image file into the `hw2pr1` folder so that it will be there when you submit!

Zipping and Submitting

Once you're finished, you'll ***submit a zip archive of your whole hw2pr1 folder***.

The folder should include:

- Your `hw2pr1.py` file with all of your functions (and the ones provided above)
- If you used an online turtle-graphics utility, place that code into `hw2pr1.py` and note which site you used
- All of your screenshots should be there, too

Then, zip the folder. ***Again, don't use any external zip utilities.*** There are built-in methods for both Windows and MacOS:

- (Windows) In older Windows you can right-click your `hw2pr1` folder and choose *Send to…*, then *Compressed (zipped) folder*. Here is a link to instructions for Windows 8.
- (MacOS) You can right-click your `hw2pr1` folder and choose *Compress*. Here is a link to the official MacOS instructions.

Be sure to submit your **`hw2pr1.zip`** zipped version of your `hw2pr1` folder to the appropriate spot in the submissions system.