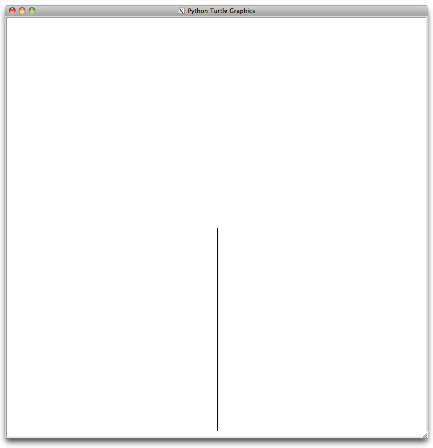
Computer Science I
Recursive Tree Lecture (1 of 2)

09/09/2018

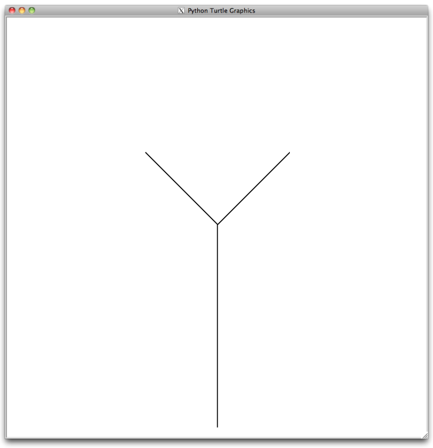
Problem

Write a program that prompts the user for a number of `segments` (a non-negative integer) and a `size` (a positive integer) and then draws a *Y-tree*. Here's how it should work.

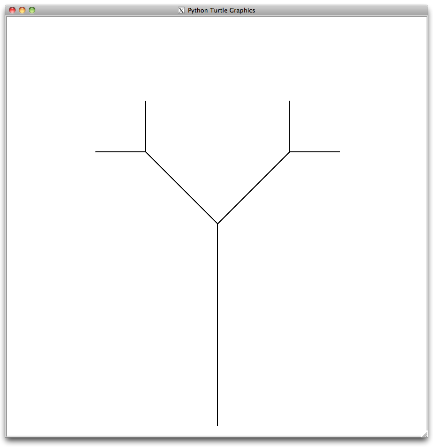
- If `segments == 0`, it draws nothing.
- If `segments == 1`, it draws a trunk (a line) of length `size` (a tree with no branches).
- If `segments == 2`, it draws a little tree, consisting of a trunk of length `size` that splits into two branches (two lines) of length `size/2`. The tree is symmetric and there is a right angle between the two branches.
- If `segments == 3`, it draws a tree with four more branches. The tree is the same as for `segments == 2`, but with additional splits at the ends of the branches of length `size/2`, where each of the new branches is of length `size/4`.
- For each greater value of `segments`, the program draws trees with yet more branches. Each tree is similar to the previous tree, but with additional splits, where each of the new branches is half the size of the branch from which it splits.



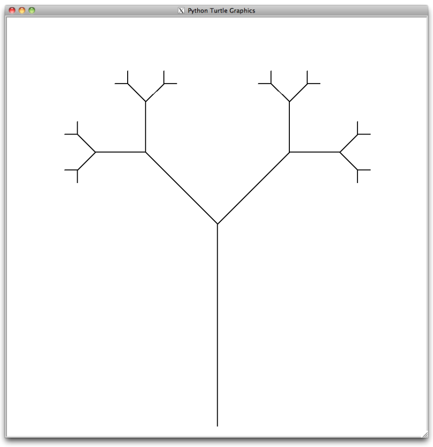
`segments is 1`



`segments is 2`



`segments is 3`



`segments is 5`

Note that `segments` counts the number of line segments you traverse to get from the tip of any branch to the base of the tree.

Problem Analysis and Solution Design

Development

How do we solve the problem for `segments == 0`? At the moment, this almost seems too ridiculous to mention.

```
def draw_tree0(size):
    pass # nothing to do!
```

How do we solve the problem for `segments == 1`? This is still simple.

```
def draw_tree1(size):
    forward(size)
```

How do we solve the problem for `segments == 2`? If we know basic turtle commands, it is still not hard.

```
def draw_tree2(size):
    forward(size)
    left(45)
    forward(size/2)
    forward(-size/2)
    right(90)
    forward(size/2)
```

These solutions work, but we've previously seen that it is good for figure-drawing functions to return to their initial position and orientation after drawing.

Obviously, for the case of `segments == 0` this is trivial to achieve — continue to do nothing! For now we are skipping that case; we will add it back at the end.

Using that idea we solve the problem and return to the initial position and orientation for `segments == 1` like this:

```
def draw_tree1(size):
    forward(size)
    forward(-size)
```

And we might solve the problem and return to the initial position and orientation for `segments == 2` this way:

```
def draw_tree2(size):
    forward(size)
    left(45)
    forward(size/2)
    forward(-size/2)
    right(90)
    forward(size/2)
    forward(-size/2)
    left(45)
    forward(-size)
```

We should notice that there is some repetition in the solution for `segments == 2` and that the repeated code is similar to the solution for `segments == 1`. We can replace the repetitive code by combining it with the solution for `segments == 1`.

```
def draw_tree2(size):
    forward(size)
    left(45)
    draw_tree1(size/2)  # reuse the other function
    right(90)
    draw_tree1(size/2)  # again
    left(45)
    forward(-size)
```

How do we solve the problem and return to the initial position and orientation for `segments == 3` (and reuse our solution for `segments == 2`)?

```
def draw_tree3(size):
    forward(size)
    left(45)
    draw_tree2(size/2)
    right(90)
    draw_tree2(size/2)
    left(45)
    forward(-size)
```

How do we solve the problem and return to the initial position and orientation for `segments == 4` (and reuse our solution for `segments == 3`)?

```
def draw_tree4(size):
    forward(size)
    left(45)
    draw_tree3(size/2)
    right(90)
    draw_tree3(size/2)
    left(45)
    forward(-size)
```

Evolution into a Recursive Function Solution

Recall that it is possible for a function to call another function, as can be seen by the fact that our functions call functions in the `turtle` module. Having a function *call itself* is no exception! You should know that when that happens, you should imagine multiple "copies" of the function being "active" at the same time, each one having its own separate values for its parameters and other local variables.

Looking at the similarities between `draw_tree2`, `draw_tree3`, `draw_tree4`, we can use a `segments` parameter to distinguish the execution of different copies of the `draw_tree` function when the program runs.

```
def draw_tree(segments, size):
    if segments == 1:
        forward(size)
        forward(-size)
    else:
        forward(size)
        left(45)
        draw_tree(segments-1, size/2)
        right(90)
        draw_tree(segments-1, size/2)
        left(45)
        forward(-size)
```

Let's bring back the case of `segments == 0`. We can incorporate it into our `draw_tree` function as follows.

```
def draw_tree(segments, size):
    if segments == 0:
        pass
    elif segments == 1:
        forward(size)
        forward(-size)
    else:
        forward(size)
        left(45)
        draw_tree(segments-1, size/2)
        right(90)
        draw_tree(segments-1, size/2)
        left(45)
        forward(-size)
```

Finally, we notice that the case `segments == 1` need not be treated specially — drawing a tree with `segments == 1` corresponds to drawing the trunk, then "drawing" two trees with `segments == 0`.

```
def draw_tree(segments, size):
    if segments == 0:
        pass
    else:
        forward(size)
        left(45)
        draw_tree(segments-1, size/2)
        right(90)
        draw_tree(segments-1, size/2)
        left(45)
        forward(-size)
```

The approach used in this solution is called **recursion**. Recursive solutions to a problem depend on solutions to smaller instances of the same problem. In our case, the steps for drawing the branches for each of the segments is identical. Because of this similarity, we are able to generalize the code to develop our final `draw_tree` function.

Final Program Solution

Our solution to the `draw_tree` function can be written as follows, *with lots of documentation.*

```
# function definitions

def draw_tree( segments, size ):
    """
    draw_tree recursively draws the tree.

    segments -- NonNegInteger;
                number of line segments from the base of the tree to
                the end of any branch should be integral and non-negative.
    size -- PosNumber;
            length of tree "trunk" to draw should be (strictly) positive.

    pre-conditions: segments >= 0, size > 0.
                    turtle is at base of tree,
                    turtle is facing along trunk of tree,
                    turtle is pen-down.
    post-conditions: a segments-level tree was drawn on the canvas,
                     turtle is at base of tree,
                     turtle is facing along trunk of tree,
                     turtle is pen-down.
    """
    if segments == 0:
        # base case: draw nothing
        pass
    elif segments > 0:
        # recursive case: draw trunk and two sub-trees
        turtle.forward( size )
        turtle.left( 45 )
        draw_tree( segments - 1, size / 2 )
        turtle.right( 90 )
        draw_tree( segments - 1, size / 2 )
        turtle.left( 45 )
        turtle.forward( -size )
```

Note that the `draw_tree` expects the turtle's position and orientation to be at the base of the tree and facing along the trunk. This is called a **pre-condition** because a specific state is assumed to have been established *before* the function begins its execution.

Also, note that we are careful to return the turtle to its initial position and orientation at the base of the tree and facing along the trunk after drawing the tree. The turtle must be at exactly the same position and in exactly the same orientation as at the beginning of the drawing. Suppose otherwise: after drawing the left smaller tree, the turtle would not be at the crotch of the tree (the base of the smaller tree) and we would draw the right tree starting somewhere other than the crotch. (See what happens if you remove the "`move by length -size units`" from the pseudocode of `draw_tree`.) This is called a **post-condition** because a specific state is guaranteed to have been established *after* the function finishes its execution.

We should always document pre- and post-conditions for functions. This allows other programmers to know what needs to be done before calling the function and what can be expected after calling the function. Beware: if you call a function without satisfying its pre-conditions, all bets are off!

Execution Diagram

We can visualize the execution of the `draw_tree` function with an *execution diagram*.

```
draw_tree( segments := 2, size := 100 )
    # segments <> 0
    forward( size )
    left( 45 )
    draw_tree( segments := 1, size := 50 )
        # segments <> 0
        forward( size )
        left( 45 )
        draw_tree( segments := 0, size := 25 )
            # segments == 0
            pass
        right( 90 )
        draw_tree( segments := 0, size := 25 )
            # segments == 0
            pass
        left( 45 )
        forward( -size )
    right( 90 )
    draw_tree( segments := 1, size := 50 )
        # segments <> 0
        forward( size )
        left( 45 )
        draw_tree( segments := 0, size := 25 )
            # segments == 0
            pass
        right( 90 )
        draw_tree( segments := 0, size := 25 )
            # segments == 0
            pass
        left( 45 )
        forward( -size )
    left( 45 )
    forward( -size )
```

Testing (Test Cases, Procedures, etc.)

We need to test the base case(s) and the recursive case(s).

For the base case that draws nothing, we should test `segments == 0` and a variety of values for `size` (e.g., 10, 100, 1, 0, possibly also -10).

For the simple recursive case that draws a trunk and recurses to draw nothing, we should test `segments == 1` and a variety of values for `size` (e.g., 10, 100, 1, 0, possibly also -10).

For the complex recursive cases that draw a trunk and recurse to draw something, we need to test `segments > 1` (e.g., 2, 3, 5) and a variety of values for `size`.

NOTE: The variety of values for `size` only matter if we do not adjust the world coordinates to the `size`.

Sample Implementation

See `recursive_tree.py` for an example solution.