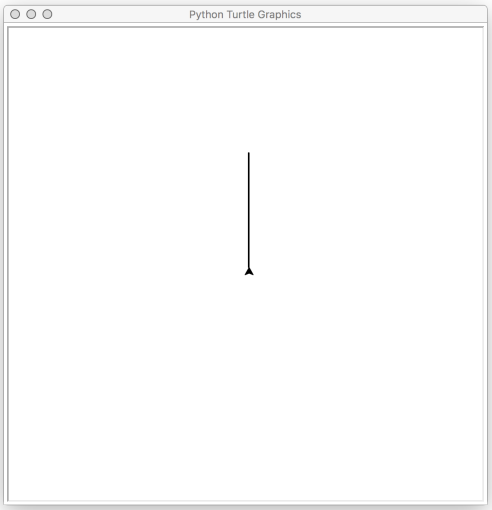
Computer Science I
A Tree of Trees — Lecture Part 2

08/16/2018

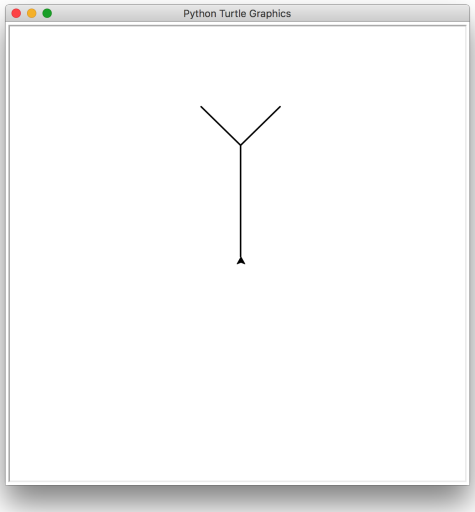
Problem

Write a program that draws variations on a *Y-tree*. Each level is one or more "trees".

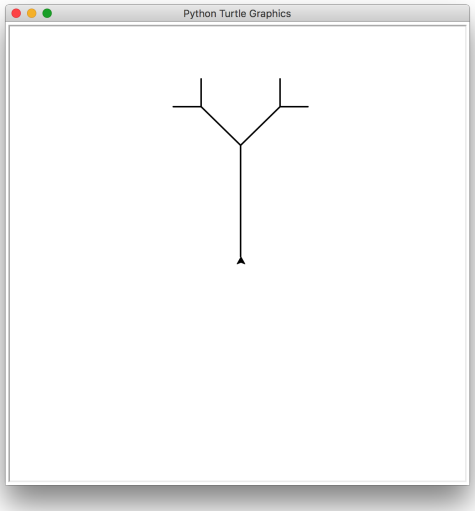
When the drawing completes, the program should print the total sum of the lengths of all the branches of the tree including the trunk.



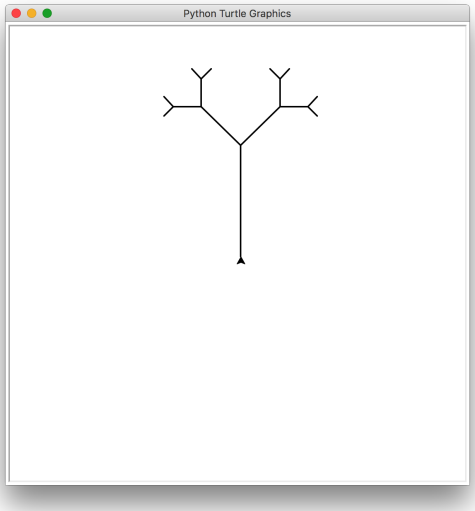
`levels == 1`



`levels == 2`



`levels == 3`



`levels == 4`

Note that a `level` counts the number of branch line segments from the base of the tree to the end of any branch. Also, there are 2 branches for level 2, 4 branches for level 3, and 8 branches for level 4.

The program will draw each of the above trees in sequence, and pause until the user presses the ENTER key. It draws these *Y-tree variations* as follows:

- If `levels == 0`, it draws nothing, and the turtle appears to be the "seed" of all subsequent trees drawn.
- If `levels == 1`, it draws a tree trunk of length `size` (a somewhat degenerate tree).
- If `levels == 2`, it draws a tree, consisting of a trunk of length `size` that splits into two branches, each of length `size/2`; this is a Y-tree. The tree is symmetric and there is a right angle between the two branches.
- If `levels == 3`, it draws a tree with four more branches. The tree is the same as for `levels == 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 the last `level == 4`, the program follows the pattern drawing trees with yet more branches. This 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.

Problem Analysis and Solution Design

Investigation

How do we solve the problem for `levels == 1`? This seems simple.

```
import turtle as tt  # a short alias for calling turtle functions

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

How do we solve the problem for `levels == 2`? This also seems simple.

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

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

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

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

And we might solve the problem and move back to the initial position and orientation for `levels == 2` this way:

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

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

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

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

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

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

```
def draw_tree4(size):
    tt.forward(size)
    tt.left(45)
    draw_tree3(size/2)
```

```
        tt.right(45)
        tt.right(45)
        draw_tree3(size/2)
        tt.left(45)
        tt.forward(-size)
```

How do we solve the problem for `levels == 0`?

```
    def draw_tree0(size):
        pass
```

Note that `pass` is the Python command to do nothing.

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

```
    def draw_tree1(size):
        tt.forward(size)
        tt.left(45)
        draw_tree0(size/2)
        tt.right(45)
        tt.right(45)
        draw_tree0(size/2)
        tt.left(45)
        tt.forward(-size)
```

Pre-conditions and Post-conditions

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

Also, note that we are careful to move the turtle back 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 at the end as at the beginning of the procedure.

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 `draw_tree1`.) This is called a **post-condition** because the code needs to ensure that a specific state has been established *after* the procedure finishes its execution.

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

Returning the Total Length

It seems we forgot that we need to return the total sum of the lengths of all the branches of the tree including the trunk.

This means we need to write *fruitful functions* that *return values* to the caller. A fruitful function is a procedure that produces, or returns, a meaningful value to the code that called it.

The return statement is the way Python sends a value back to the caller of a function. When the `return` statement executes, it stops executing the current function and sends the value of its argument to the calling function.

For example, here is a fruitful function that returns a very imprecise value for the mathematical value **PI**:

```
def PI():
    return 3.141
```

Here the `print` function prints twice the return value:

```
>>> print( 2 * PI() )
6.282
>>>
```

The `PI` function returned its value, the $*$ multiplied it, and `print` output the result followed by a `newline`.

The example of `return` above showed an argument; if there is no argument, and only the Python keyword `return` appears, then the returned value is a special value **None**, which represents nothing. The procedures we have written so far actually returned `None` when they reached the end of execution.

```
>>> # print is not a fruitful function; it returns None.
>>> print( print( "hello" ) )
hello
None
>>>
```

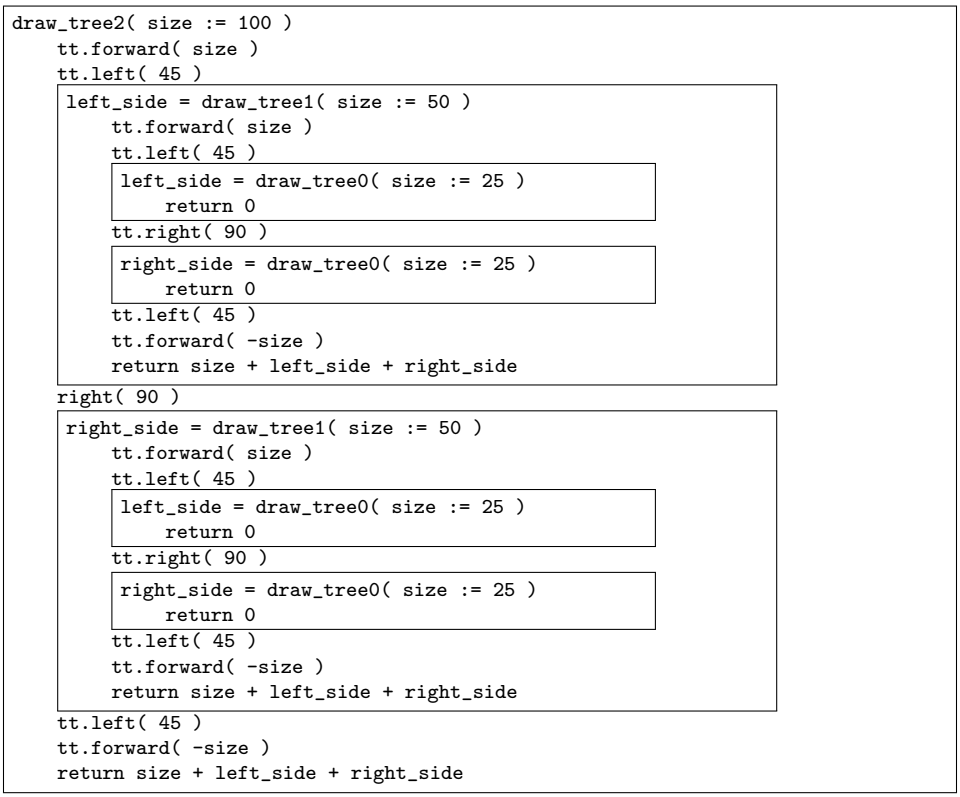
To get the total length, we have to save the lengths of the left side and right side, then add them to the length of the trunk. For example, a function for level 3 to do that would look like this:

```
def draw_tree3( size):
    # ...
    left_side = draw_tree2( size / 2 )
    # ...
    right_side = draw_tree2( size / 2 )
    # ...
    # add current trunk size to left and right and return the sum
    return size + left_side + right_side
```

And now, `draw_tree0` would have to `return 0` instead of `pass`.

Execution Diagram

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



There is something strange in the execution trace: every function has a `size` parameter, and each function's size has a different value.

Scope: Why there can be so many different `size` names

As the functions call each other, there are multiple, different `size`s, and each function keeps its own value bound to its parameter. The **call stack** is a pile of function calls holding the values for each function's parameters, as well as other data associated with the function call.

The stack starts at the bottom with the first function call. It grows when one function calls another, and that one calls another, and so on. The stack shrinks by one **stack frame** when a function finishes executing, and the stack becomes empty when the first function finishes.

Each stack frame holds the value of its `size` parameter for that function. The code for that function at the top will only see the value in that topmost stack frame because that stack frame *hides* the `size` values of the other stack frames below.

If we consider the execution of `draw_tree4( 100 )` as the first function, and we build the stack up until it calls `draw_tree0` the first time inside `draw_tree1`, then we get this picture of the call stack at the point in execution of drawing the leftmost branch of a level 4 tree:

```
total = draw_tree0( size := 6.25 )
    return 0
```

```
total = draw_tree1( size := 12.5 )
    ...
    left_side = draw_tree0( 12.5 / 2 )
    ...
```

```
total = draw_tree2( size := 25 )
    ...
    left_side = draw_tree1( 25 / 2 )
    ...
```

```
total = draw_tree3( size := 50 )
    ...
    left_side = draw_tree2( 50 / 2 )
    ...
```

```
total = draw_tree4( size := 100 )
    ...
    left_side = draw_tree3( 100 / 2 )
    ...
```

Implementation

See `combo_tree.py`.

One thing to notice in the solution module: there is a special if statement at the end, around the code that starts things. This is known as an **if guard**, which we almost always want to put into our program modules to control when and whether the code executes when imported.

```
if __name__ == "__main__":
    run_test_cases()
```

We can either run a module directly as a main program, or we can import that module the way we import the turtle. When we import the module so that another module can use its procedures, we do not want any "main code" to execute. The **if guard** checks the interpreter's `__name__` variable binding, and, if it is the string `__main__`, then the interpreter is running the module as a main program.

Execution Testing (Test Cases, Procedures, etc.)

Each level represents a case to test, and the program pauses after drawing each tree combination. The first level should draw only a trunk of the given `size`, and the second level should draw a trunk of the given `size` and 2 branches of `size/2`. The third and fourth levels should work in an analogous fashion.

Assessment

We assess our work by asking ourselves questions like these:

- What works well?
  Each combination of tree structure looks fine.
- What works poorly?
  The test procedure prints the `total`, but we must hand-calculate the result to verify that the value is correct.
- How could the code be improved?
  We could compute the expected result and compare the returned value using this formula: $expected = levels * size$. Then check whether `total == expected`, and report an error if there is not a match.
- What missing conditions or situations were identified in testing?
  Is it ok that, negative numbers and numbers greater than 4 are non-legal values, are not checked by the code? The *docstring* for `init_world_and_draw_tree` is missing the necessary *pre-conditions* stating that the value of `levels`; it only states that the argument must be a *NonNegativeInteger*.
- What improvements might make the code smaller, faster, more reliable?
  The answers to the previous questions identify several changes that might improve the code.