CS 120: Intro to Computer Programming II

In-Class Activity - 06 Trees - Activity 4

Activity 1 - Turn in this one

Let's play around with **default arguments** - while I often ban these on homework assignments (because I want you to focus on the problem I've assigned), they are extremely valuable in the Real World!

A default argument in Python is simply a parameter which is declared with an 'equals' sign, which gives it a value if the caller doesn't provide one. If you want both ordinary arguments, and arguments with defaults, in the same function, that's fine, but the default args must come last.

For this activity, you will write a recursive function that prints out the contents of a tree, but you will indent the printouts to make it visually obvious which values are inside which subtrees. To do this, declare the following function:

```
def pretty_print_tree(root, indent="")
```

The `pretty_print()` function can be called in two ways: with two parameters, where you supply an indent, and with only one argument, where the argument defaults to `""` :

```
root = ...
pretty_print_tree(root, "      ")
pretty_print_tree(root)            # indent will be set to ""
```

This function must print out the tree in a pre-order traversal. When it recurses into its children, it **must** pass each of the two children two spaces more than it was given. For example, here's what `tree1` from Activity 1, in the previous day, would look like:

```
74
  4
    0
    17
  80
    77
    96
```

Write the function, and discuss with your group the usefulness of default arguments. How would a user probably call this function?

Solution:

```
def pretty_print_tree(root, indent=""):
    if root is None:
        return root

    print(f"{indent}{root.val}")

    pretty_print_tree(root.left,  indent+"  ")
    pretty_print_tree(root.right, indent+"  ")
```

(activity continues on next page)

Activity 2 - Turn in this one

For the next couple activities, it will be useful to have relatively large trees, and it will be useful if we can build them randomly, rather than by hand.

Adapt the BST `insert()` function we've seen (slide deck 06, slide 30 and following) so that it builds **random** trees. Make the following changes:

- Remove the value parameter; you don't need it.
- When you build a new node, choose its value **randomly**
- When you recurse, choose **randomly** which way it goes.

Then, you can write a simple function which builds a tree by calling this new `random_insert()` function many times, in a loop. Once you have it all written, test it out a few times: build a medium-sized tree, and then pass it to your `pretty_print_tree()` function to see what it looks like.

REMEMBER: In Python, you generate random numbers by importing the `random` library, and then calling:

```
random.randint(min_value, max_value)
```

Activity 3 - Turn in this one

Often, we find ourselves wanting to print out some additional information about each node in the tree. Modify your "pretty print" function so that it will print out, along with each node, the number of nodes in that subtree as well. (You'll need to write a simple function to calculate the size of a tree, but you know how to do that...)

Try it out with a few trees. **Start small,** but then try larger ones.

Solution:

```
def tree_size(root):
    if root is None:
        return 0
    else:
        return 1 + tree_size(root.left) + tree_size(root.left)

def pretty_print_tree(root, indent=""):
    if root is None:
        return root

    print(f"{indent}{root.val}    size={tree_size(root)}")

    pretty_print_tree(root.left,  indent+"  ")
    pretty_print_tree(root.right, indent+"  ")
```