CSC148 Lab#9, winter 2015

learning goals

In this lab you will explore unit tests, assertions, and tracing function calls.

set-up

Download minimax_test.py and save it under a new directory named **lab09**. You should also download and save bt_functions.py, node_sol.py, and function.py in the same area.

unit tests

In class you've seen some unit tests for **minimax** run on instances of **SubtractSquareState**. We claim that the chance of an implementation passing all those tests randomly — based on sheer luck — is 1 in 625, or less than 1 percent.

That may, or may not, sound very convincing. Add some more tests to **minimax_test.py** to decrease the chance of randomly passing the tests to less than 1 in 10,000. You should run the tests against your own **strategy_minimax.py** and our **subtract_square_state.py**. Don't worry if your **minimax** doesn't pass every test — the point of unit testing is to help find, and eventually squash, bugs.

Now consider **bt_functions.py**. This module has functions to **insert** and **delete** nodes. Devise some test cases for these methods, and implement them in a new unit test file called **bst_test.py**, considering important examples of input. You'll need to consider cases where values are inserted in various orders.

Run your tests against our solutions in **bst.py** — maybe you'll find some bugs!

assertions

Much of the code you've developed this semester has involved branching recursion — there are two or more recursive calls executed. You may well have found that a traditional debugger (such as Wing's debugger) is not too helpful in such cases — many recursive calls pile up on the call stack, and are hard to interpret.

One answer is to use **print(...)** statements, but these tend to create "noisy" output, and they need to be removed when the code is considered finished. A good alternative is assertions, implemented as **assert** statements.

In class we looked at some assert statements. Read over **bt_functions.py** and devise **assert** statements for enforcing type contracts. Think about helpful strings to be produced when the assertion is falsified.

Now read over **node_sol.py**. Consider any important assumptions you believe to be true before and/or after any while loop. Devise **assert** statements to place in the code.

function calls and namespaces

In Danny's week 10 lecture slides there is an example of tracing some twisted function calls. Follow the given method, that is:

- evaluate each function argument from inner-most to outer-most, left to right,
- functions calls with bodies that are not finished being evaluated stay on the call stack

... and use it on function.py. Do this on paper first, the verify the steps you have taken by using the Wing debugger:

1. put a break point in the body of **h(n)**
2. click the little lady bug to start the debugger
3. be sure that you are displaying the stack frames, as well as the stack data
4. click the "step into" tool (tool bar) to step through your code.