CS 120: Intro to Computer Programming II

In-Class Activity - 07 Debugging - Day 2

Some of these examples were taken, or adapted, from sources on the web. The sources are available upon request, but I'm not providing them to you right now, because I want you to solve these yourself.

Leave space for all of your group members to find the bugs! Some people in the group will be able to glance through, and find some of the bugs (although there were a couple that weren't obvious to me until I ran the code). But remember: **our goal is for everybody to learn,** not to just "finish quick." So if you see the bugs, don't point them out - instead, let the whole group work together, and stumble across the bugs together.

Maybe we could even rotate bug-fixing? When we find a new bug in the code, ask somebody who hasn't talked yet to help the group debug this one.

(Follow this policy for **all** of the debugging exercises we do, over the next few days.)

Activity 1 - Turn in this one

Why does the following code sometimes print things out of order?

```
data = []

while True:
    # yeah, we ought to add a try/catch.  That would be good, but it's not
    # the bug I'm looking for here...
    val = input("Enter a number (blank line to end)")
    if val == "":
        break
    data.append(val)

for v in sorted(data):
    print(v)
```

Solution: We are reading the values as strings, and sorting them. But we were hoping to sort them in terms of integers. But unfortunately, string-sorting sorts **lexicographically** - that is, it looks at each character in turn, and will sort two strings based on the first character that is different.

Thus, even though 2 is less than 11, the string `"2"` is **greater** than the string `"11"`.

(activity continues on next page)

Activity 2 - Turn in this one

Why does this code run forever, for most inputs? Fix it so that it doesn't.

```
# calculates the base-10 logarithm of a number, rounding up.
def log10(val):
    count = 0
    while val != 1:
        val = val // 10
        count += 1
    return count
```

Solution: It loops if val goes to 0 - which is normal if we start with 2. But there are other bugs, too - 101 should return 3, but because we're using integer division, 101 goes to 10, and thus will noly return 2. To fix this, we'll do the following:

```
def log10(val):
    assert val != 0
    count = 0
    while val > 1:
        val   /= 10
        count += 1
    return count
```

Activity 3 - Turn in this one

Now that our function seems to work for a few inputs, it's time to automatically test it. I've written the following automatic testing function, which will generate a lot of random numbers, and confirm the return value of `log10()` on each one. Happily, it even automatically generates its own expected results!

```
import random
def test_log10():
    count = 0
    failures = set()
    while True:
        for i in range(1000):
            v = random.randint(1, 1000*1000*1000)
            actual   = log10(v)
            expected = len(str(v))
            if actual != expected:
                print(f"BUG: log10({v}):   actual={actual} expected={expected}")
                failures.add(v)
        count += 1000
        print(f"{count} tests completed.  Failures found so far: {failures}")
```

Try running the tester for a while. Most likely, it won't report any failures. But unfortunately, the **test code has a bug, too!**

Run the test program for a while. (Maybe one of the people in your group can run it for 10 or 20 minutes. But don't waste time waiting for it - do other, more useful things, while you're waiting for a failure to show

up.) Does this report any errors? If so, debug with your group - did the test code find a bug in **your** code, or did you find a bug in the tester?

If your group never finds a bug in the test code, that's alright. Report to your TA how many tests the test code completed.

Solution: The tester has a bug with powers of 10. `log10(100)` should return 2, but the tester thinks that it should be 3. So we can add a special case, which checks for powers of ten:

```
def test_log10():
    for i in range(100):
        v = random.randint(1, 1000*1000)
        actual   = log10(v)
        expected = len(str(v))

        # special case for *exact* powers of 10
        if 10**actual == v:
            expected -= 1

        if actual != expected:
            print(f"BUG: log10({v}):   actual={actual} expected={expected}")
```

(activity continues on next page)

Activity 4 - Optional

OPTIONAL. Complete this if you have time, and turn it in. If you don't have time, you may report to your TA that you ran out of time.

This function attempts to determine whether a binary tree is a BST or not. (In case you're wondering, an empty tree, and a leaf, are both BSTs - just in the most simple, meaningless sense!)

First of all, this function sometimes crashes. Fix that first. (Turn in your version which fixes the crash.)

Unfortunately, that's not enough! This function has a rather fundamental flaw: it will sometimes return `True`, even on trees that are not BSTs. Why is this? If you're not sure, try building some random trees - either by hand, or maybe with your `random_tree()` function from a week or so ago - and see if you can find a tree which is not a BST, but which this function will this **is.** (If you can find one, turn it in.)

```
def is_BST(root):
    if root is None:
        return True
    if root.left is None and root.right is None:
        return True

    if root.left.val > root.val or root.right.val < root.val:
        return False

    return is_BST(root.left) and is_BST(root.right)
```

Solution: First, we fix the crash:

```
def is_BST(root):
    if root is None:
        return True
    if root.left is None and root.right is None:
        return True

    if root .left is not None and root.left .val > root.val:
        return False
    if root.right is not None and root.right.val > root.val:
        return False

    return is_BST(root.left) and is_BST(root.right)
```

What's the lingering problem? We are only comparing values to their **direct** parents! Thus, the code above will return `True` for the following tree:

```
  100
 /
50
 \
  150
```

(activity continues on next page)

Challenge Activity - Do not turn in this one

This is a follow-up to the previous activity. **Don't read this until you've completed the previous activity.** We've come up with another version of `is_BST()`, and this one actually gives the correct answer in all situations! But we're not excited about its performance.

This time, debug a function **to improve its performance** - not for correctness. Can you explain why this function is kind of slow? Sketch out **conceptually** how you might fix it. (If you have time, go ahead and try to write it - but I'm betting that few groups will have time.)

```
def is_BST(root):
    if root is None:
        return True

    if root.left  is not None and nothing_greater_than(root.left , root.val) == False:
        return False
    if root.right is not None and nothing_less_than   (root.right, root.val) == False:
        return False

    return is_BST(root.left) &&

def nothing_greater_than(root, val):
    if root is None:
        return True
    if root.val > val:
        return False
    return nothing_greater_than(root.left , val) and \
           nothing_greater_than(root.right, val)

def nothing_less_than(root, val):
    if root is None:
        return True
    if root.val < val:
        return False
    return nothing_less_than(root.left , val) and \
           nothing_less_than(root.right, val)
```

Solution: Just like our `size()` problem from a couple weeks back, this function recurses into the same object **several times.**

No, I'm not going to give you the solution...you'll have to work on that for yourself!