CS 1110 Spring 2020, Assignment 4: Tre1110 redux

Updates:

The assignment itself, with updates marked in orange, begins on the next page. On this "page 0", we also document the time, location, and nature of the updates, in reverse chronological order,

- Sat Apr 25, 5pm: Typo in a4.py specification of `indentlines()`: Remove stray 'd' in '...abdc...' in the 3^{rd} example. The zip file posted to the course website has been updated with this change:

  `'\tabc\n\tdef\n\tghi' --> '\t\tabdc\n\t\tdef\n\t\tghi'`

  should be

  `'\tabc\n\tdef\n\tghi' --> '\t\tabc\n\t\tdef\n\t\tghi'`

- Wed Apr 22, 9:30am: a4.py: The example given in the specification for `todo_list_to_string()` had copy-paste errors. It has now been fixed in the zip file posted to the course website.

CS 1110 Spring 2020, Assignment 4: Tre1110 redux*

1 Motivation: Get all the things done!

To-do lists, which we saw in A3, need not be just flat lists, but can be hierarchical or nested. One of your instructors has a to-do list that includes items related to research and items related to the course:

- Her research items are broken up into a grant-proposal document she needs to write and, for each of her PhD students,
  - a separate list of what she needs to get back to each student with.
- Her course responsibilities include creating some labs and creating A4 and A5.
  - A4 consists of:
    - writing the skeleton, solution, and testing code for the programming problems
    - writing up the assignment handout
    - (etc.)

How much time does she need to get all this done? Can she get an organized printout of all the things she needs to do? Help her, Obi-Wan, by completing this assignment!

More seriously, the idea behind this assignment is that nested lists are a natural and practical data structure where recursion can really shine.

Contents

*Authors: Lillian Lee.

2 Rules

2.1 New: resubmission after grading feedback is (conditionally) allowed!

You are welcome to resubmit *one* of A4 or A5 after receiving grading feedback; further details to be arranged.

We are arranging for resubmission because we are concerned that there may currently be huge variation in learning situations of students across the course, and we want to reduce the stress of having the deadlines for two heavily-weighted assignments coming at you. We hope that resubmission will allow you to focus on improvement and mastery of the material over a more gradual period of time.

However, many of the course staff are themselves carrying heavy burdens, and so we aren't in a position to be able to offer regrading of resubmissions for both assignments; hence the limit to one of A4 or A5.

2.2 (Same as before) You need to re-group (so to speak) on CMS, regardless of prior groupings

You may work alone or with just one other person, who can be someone you've grouped with before in CS1110, or a different person.

If you are partnering, regardless of whether you were grouped in a previous assignment, the two of you must still form a new group *specifically for this assignment* on CMS before submitting.[1]

If your partnership dissolves, see the course Academic Integrity description about "group divorce" for what to do.

2.3 (Same as before) (Dis-)allowed collaborations and documentation requirements

Our policies are laid out in full on the course Academic Integrity page, but we re-state here **the main rules**: where "you" means you and, if there is one, your one CMS-registered group partner,

1. Never look at, access or possess any portion of another group's work in any form.
2. Never show or share any portion of your work in any form to anyone except a member of the course staff.
3. Never request solutions from outside sources; for example, on online services like StackOverflow.
4. DO specifically acknowledge by name all help you received, whether or not it was "legal" according to (1)-(3).

3 Due dates

(a) If you are partnering: well before submission, follow the "How to form a group" instructions on the course CMS Usage Guide. Both parties need to act on CMS in order for the grouping to take effect.

(b) By 2pm on Thu Apr 30, submit whatever you have done at that point CMS for Assignment A4, following steps 1-3 in the "Updating, verifying, and documenting assignment submission" section of the course Assignments page. It is OK if you haven't finished working on it yet.[2]

(c) By **11:59pm**[3] **on Thu Apr 30**, make your final submission of file `a4.py` to Assignment A4, again following the aforementioned steps 1-3.

4 Task

4.1 Overview

A major goal of this assignment is practice with recursion, since recursion is a great way to get things done. This goal implies that, for each of the functions we have you implement, *we reserve the right to assign no credit for code that isn't fundamentally based on recursion, or does not use recursion in the way we ask for, when we request a recursive implementation*, even if the code fulfills the specification.

The files needed for the assignment can be found in this zip file. It includes `a3_classes.py` from A3, since we'll be using the Task class again.

[1] This links your submission "portals".

[2] The 2pm checkpoints provide you a chance to alert us if any problems arise. Since you've been warned to submit early, do not expect that we will accept work that doesn't make it onto CMS on time, for whatever reason. There are no so-called "slipdays" and there is no "you get to submit late at the price of a late penalty" policy. Of course, if some special circumstances arise, contact the instructor(s) immediately.

[3] **PLUS** a 9-hour grace period (anyone can take advantage of it, but the intent is to account for students in "distant" time zones relative to Ithaca or for connection issues).

4.2 Test Cases

We've provided test cases in `a4_test.py`. You should make sure you understand the overall testing scenario we've set up, where there's One Todo List to rule them all, `my_todo_list`, made up of Tasks and (lists of ...) lists of sub-Tasks. The code[4] is shown below left. On the right is the kind of output your code should be able to produce: a nicely formatted version of `my_todo_list` (comments have been added, for expository purposes).

```
import a3_classes as a3

# Create some to-do lists

# Todo items I owe to each of my PhD students
owe_s1 = [a3.Task('camera-ready revisions', 8),
          a3.Task('revise slides', 1)]
owe_s2 = [a3.Task('review proof', 3)]
owe_s3 = []
owe_students = [owe_s1, owe_s2, owe_s3]

research_todos = [a3.Task("NSF proposal response", 2),
                  owe_students,
                  a3.Task("Friday talk feedback", 1)]

solncode = [a3.Task("printing", 1),
            a3.Task("counting", 1),
            a3.Task("nesting depth", 1)]

asst4 = [solncode,
      a3.Task("writeup", 6),
      a3.Task("pkg and post", 1),
      a3.Task("grading code and guide", 8)]

course = [asst4,
          a3.Task("lab 9", 10),
          a3.Task("A5", 20),
          a3.Task("lab 10", 11)]

my_todo_list = [research_todos, course]
```

```
BEGIN # start of my_todo_list
  BEGIN # start of research_todos
    NSF pr: 2
    BEGIN # start of owe_students
      BEGIN  # start of owe_s1
        camera: 8
        revise: 1
      END
      BEGIN # start of owe_s2
        review: 3
      END
      BEGIN # start of owe_s3 (yes, it's empty)
      END
    END
    Friday: 1
  END # end of research_todos
  BEGIN # start of course
    BEGIN # start of asst4
      BEGIN start of solncode
        printi: 1
        counti: 1
        nestin: 1
      END
      writeu: 6
      pkg an: 1
      gradin: 8
    END
    lab 9 : 10
    A5    : 20
    lab 10: 11
    END
END
```

Note that `course`, `asst4`, ..., `owe_s1` are all also valid todo lists that could be processed in their own right.

You are free to create more test cases, and encouraged to do so if you have the time, but we are not asking you to submit your testcases.

Nonetheless, *we reserve the right to grade your code in part on its output on different test cases than what we gave you in a4_test.py.*

4.3 The Functions To Write

Here are the specifications for the functions whose bodies you are to write in `a4.py`. Follow the directions given as comments in `a4.py`.

[4] with some reformatting and some shortening of strings for space reasons.

4.3.1 sum_hours

```
def sum_hours(tlist):
    """Returns the number of hours it would take to finish all the tasks contained
    in the (possibly empty, possibly nested) Task list tlist.
    If tlist is empty, returns 0.

    Precondition: tlist is a list, possibly empty and possibly nested, of Tasks,
    and the same is true of any of its sublists, or subsublists, or ...

    Examples:
        owe_s1 = [a3_classes.Task('camera-ready revisions', 8),
                  a3_classes.Task('revise slides', 1)]
        owe_s2 = [a3_classes.Task('review proof', 3)]
        owe_s3 = []
        owe_students = [owe_s1, owe_s2, owe_s3]
        research_todos = [a3_classes.Task("NSF proposal response", 2),
                  owe_students,
                  a3_classes.Task("Friday practice-talk feedback", 1)]

        Then,
            owe_students --> 12
            research_todos --> 15
    """
```

4.3.2 todo_list_to_string

```
def todo_list_to_string(tlist):
    """Returns a string version of a (possibly nested, possibly empty) list of
    todo items tlist,
    each task on a separate line printed via task_to_string,
    each list, including the input `tlist` itself, delimited with a starting string
    'BEGIN\n' and ending string 'END'
    and each (sub)list's contents indented by one tab ('\t') per embedding level.
    When printed, this string shows the indent levels visually.

    Precondition: tlist is a list, possibly empty and possibly nested, of Tasks,
    and the same is true of any of its sublists, or subsublists, or ...


    Example:
    Suppose we set up `owe_students` as follows:
    owe_s1 = [a3_classes.Task('camera-ready revisions', 8),
                    a3_classes.Task('revise slides', 1)]
    owe_s2 = [a3_classes.Task('review proof', 3)]
    owe_s3 = []
    owe_students = [owe_s1, owe_s2, owe_s3]

    Then, the following very long string (broken up by "+" for readability) is
    the result of todo_list_to_string(owe_students):
```

```
        'BEGIN\n'
        +'\tBEGIN\n'
        +'\t\tcamera: 8\n'
        +'\t\trevise: 1\n'
        +'\tEND\n'
        +'\tBEGIN\n'
        +'\t\treview: 3\n'
        +'\tEND\n'
        +'\tBEGIN\n'
        +'\tEND\n'
        +'END'
        which, if printed via print(...), looks like this (yes, there's an empty list):
        BEGIN
            BEGIN
                camera: 8
                revise: 1
            END
            BEGIN
                review: 3
            END
            BEGIN
            END
        END
```

4.3.3 `indentlines`

The following function, which we recommend you implement in a *non*-recursive manner, is a helper for the function above. The line numbered '8' below has been altered.

```
1  def indentlines(line_str):
2      """Returns a new string where each line in line_str has been indented by a tab
3      character ('\t').
4
5      Examples:
6          'abc'   -->  '\tabc'
7          'abc\ndef\nghi'  -->  '\tabc\n\tdef\n\tghi'
8          '\tabc\n\tdef\n\tghi' -->    '\t\tabc\n\t\tdef\n\t\tghi'
9
10     This function has the following nice effect, useful for other functions in A4.
11     >>> from a4 import indentlines
12     >>> s = 'abc\ndef\nghi'
13     >>> print(s)
14     abc
15     def
16     ghi
17     >>> print(indentlines(s))
18         abc
19         def
20         ghi
21     >>> print(indentlines(indentlines(s)))
22             abc
23             def
24             ghi
25
26     Precondition: line_str is a non-empty string with no trailing whitespace
27     (no tabs, newlines, or spaces at the end).
28     Lines within line_str are delimited by the newline character '\n'.
29     None of the lines in line_str are empty or contain only whitespace
30     """
```

4.4 Testing Output: read it carefully, even when there's no "Quitting with Error" message at the bottom!

We know that getting the indentation and spacing right in `todo_list_to_string` may prove challenging. Therefore, our testing code not only checks if your output is right, but also, in the case of errors, checks whether your output would be right if we ignored whitespace.

So, below is an excerpt of the output when a test case fails, with line numbers added, where the following should be noted:

Importantly, Lines 27-28 indicate that the function is *not quite correct!*, but *the only problem is due to whitespacing.*

- Thus, if you follow our recommendation in the implementation instructions to first implement `todo_list_to_string` *without* worrying about getting the tabbing or newlines right and without employing helper `indentlines`, you can check for lines like 27-28 to see that you're on the right track, and can *then* proceed to deal with the whitespace.

Lines 4-6 are the output you are used to seeing from an `assert_equals` failure.

Line 10 shows the exact string that should have been the output (surrounded by quotation marks).

Lines 12-15 gives a *human-friendly* version of the expected string.

Line 17 shows the exact string that the code returned, surrounded by quotation marks.

Lines 19-24 gives a *human-friendly* version of the function's output. You can easily see that, in comparison to lines 12-15, there's extra blank lines at line 21 and 23.

```
Running test_print_todo_list .................
Testing input empty list
Testing input owes_s1
assert_equals: expected 'BEGIN\n\tcamera: 8\n\trevise: 1\nEND' but instead got 'BEGIN\n\tcamera: 8\n\n\trevise: 1\n\nEN
Line 145 of scratch/a4_test.py: testcase.assert_equals(expected, output)
Quitting with Error

************** Error on owes_s1 **************
What SHOULD have been the output:
'BEGIN\n\tcamera: 8\n\trevise: 1\nEND'
Human-readable version:
BEGIN
  camera: 8
  revise: 1
END
But the code's output is this:
'BEGIN\n\tcamera: 8\n\n\trevise: 1\n\nEND'
Human-readable version:
BEGIN
  camera: 8

  revise: 1

END
<output stopped here>

What if we ignore whitespace in the output?
Then your implementation is CORRECT ignoring spacing ... getting close!
```

In contrast, here's an example excerpt from the testing code you get if your code is producing incomplete output. In particular, note line 18.

```
************** Error on owes_s1 **************
What SHOULD have been the output:
'BEGIN\n\tcamera: 8\n\trevise: 1\nEND'
Human-readable version:
BEGIN
  camera: 8
  revise: 1
END
But the code's output is this:
'BEGIN\n\tcamera: 8\nEND'
Human-readable version:
BEGIN
  camera: 8
END
<output stopped here>

What if we ignore whitespace in the output?
Then your implementation is UNFORTUNATELY NOT CORRECT even ignoring spacing.
```

5 Pre-Submission Checklist and What to Submit

Files to submit: just `a4.py`.

Before submitting, ensure your code obeys the following.[5]

1. Lines are short enough (˜80 characters) that horizontal scrolling is not necessary.
2. Functions are separated from each other by at least two blank lines.

[5]These requirements up speed up the process of reading/grading hundreds of files.

3. You have removed any debugging `print` statements.
4. You have removed all `pass` statements.
5. You have removed "instruction" comments, such as "`# IMPLEMENT THIS FUNCTION`".
6. If you added any helper functions, these have good docstring specifications.

Make sure the following are all true before you submit.

8. You've changed the header comments in all files to list the entire set of people and sources that contributed to the code.
9. You (and your partner) have included your NetIDs in the header of all files.
10. The date in the header comments has been changed to when the files were last edited.
11. You have set your CMS notifications settings to receive email regarding grade changes, and regarding group invitations.
12. (reminder) If working with a partner, you have grouped on CMS. (One has invited on CMS, and one has accepted on CMS.)

6 Advice on debugging recursive implementations (text also posted to Piazza)

Something that makes debugging hard: one's code always looks right.

And recursion is an especially tricky topic, in that beginning students are trying to adjust their mental models as to how recursive functions actually work.

So how can you view your code through new eyes? Here are some suggestions (stemming from hard-won experience), when you find that you've been making lots of little changes and nothing seems to be helping.

1. Save backup copies of your code before making lots of changes, so you can restore things to the last "mostly working" version.
2. As always: add print statements to try to get information as to what the values of your variables are. Lots of issues come down to variables (including parameters) not having the values you expect.
3. Try using Python Tutor to really "see" what your code is doing on particular test cases.
4. Walk away from the computer, and try to write a solution from scratch.

It really will be worth it: once you get the hang of recursion, it leads to such elegant, concise code. The usual progression is: "I just don't see how to do this" ... to ... "How could I ever have done this any other way?!"