ECS 32B: Programming Assignment #3

Instructor: Aaron Kaloti

Summer Session #2 2020

Contents

1 Changelog

You should always refer to the latest version of this document.

- v.1: Initial version.
- v.2: Made it clear that you are modifying `unordered_list.py` for part #7.
- v.3: `remove()` will only be called on an existing item. Added autograder details.

2 General Submission Details

Partnering on this assignment is prohibited. If you have not already, you should read the section on academic misconduct in the syllabus.

This assignment is due the night of Saturday, September 5. Gradescope will say 12:30 AM on Sunday, September 6, due to the "grace period" (as described in the syllabus). *Rely on the grace period for extra time at your own risk.*

Some students tend to email me very close to the deadline. This is a bad idea. There is no guarantee that I will check my email right before the deadline.

3 Grading Breakdown

The assignment is out of 100 points. Here is a *tentative* breakdown of the worth of each part.

- Part #1: 10
- Part #2: 13
- Part #3: 13
- Part #4: 13
- Part #5: 13
- Part #6: 13
- Part #7: 25

4 Gradescope Autograder Details

You will submit two files:

- `prog3.py`
- `unordered_list.py`

If any part of your code infinitely loops...: I was trying to find a fix for this one issue that the unittest-based autograder has where if on at least one test case, your function infinitely loops, then the entire thing times out and you get no credit for any of the cases. Unfortunately, I've to email the Gradescope help team about it, so they won't give me an answer by the time the assignment is due. Consequently, if your last active submission (once the deadline hits) happens to be an infinitely looping one, then I may switch it to one of your previous submissions to Gradescope on which you got points. **You can figure out which function is infinitely looping by commenting out some of the functions and submitting to Gradescope and repeating this process until the infinite looping stops.**

4.1 Released Test Cases vs. Hidden Test Cases

You will not see the results of certain test cases until after the deadline; these are the hidden test cases, and Gradescope will not tell you whether you have gotten them correct or not until after the deadline. One purpose of this is to promote the idea that you should test if your own code works and not depend solely on the autograder to do it. Unfortunately, Gradescope will display a dash as your score until after the deadline, but you can still tell if you have passed all of the visible test cases if you see no test cases that are marked red for your submission.

Below is the number of released and hidden test cases per part.

- Part #1: 1 released, 2 hidden.
- Part #2: 1 released, 2 hidden.
- Part #3: 1 released, 3 hidden.
- Part #4: 1 released, 2 hidden.
- Part #5: 1 released, 3 hidden.
- Part #6: 1 released, 3 hidden.
- Part #7: 2 released, 3 hidden.

The cases for part #6 each call your function multiple times, such that you can't pass these cases by just unconditionally returning one of `False` or `True`.

4.2 Released Test Cases' Inputs

The released test cases only use inputs that were given in the PDF.

4.3 Changing Gradescope Active Submission

Once the deadline occurs, whatever submission is your **active submission** will be the one that dictates your final score. By default, your active submission will be your latest submission. However, you can change which submission is your active submission, which I talk about a small video that you can find on Canvas here. This video is from my ECS 32A course last summer session, so the autograder referenced therein is for a different assignment. Note that due to the hidden test cases, your score will show as a dash for all of the submissions, but you can still identify how many of the released test cases you got correct for each submission.

5 Coding Problems

For parts #1-#6, all of your code must be placed in a file called `prog3.py`. For part #7, your modifications are to `unordered_list.py`. For each of `unordered_list.py` and `prog3.py`, there should be no code placed outside of any of the functions that you define, unless that code is in the body of a conditional statement of the form `if __name__ == "__main__":`.

We will check by hand to make sure that you obeyed constraints where they are specified, such as that the recursive functions you wrote are actually recursive.

5.1 Part #1 - Recursive `get_product()`

Write a recursive function `get_product()` that takes as argument a list of integers and returns the product of the integers in the list.

If the list is empty, treat the product as 1.

Below are some examples of how your function should behave on an Interpreter.

```
>>> get_product([8, 5, 2, -1, 5])
-400
>>> get_product([3, -2, 4, 0])
0
>>> get_product([3, -2, 4])
-24
```

5.2 Part #2 - Recursive `sum_every_other()`

Write a recursive function called `sum_every_other()` that takes as argument a list of integers and returns the sum of *every other* value in the list, i.e. all of the values in the list that are at even indices.

Below are some examples of how your function should behave on an Interpreter.

```
>>> sum_every_other([8, 15, 20, -1, -5, 4])
23
>>> sum_every_other([8, 15, 20, -1, -5, 4, 2])
25
>>> sum_every_other([8, 15, 20, -1, -5, 4, 2, -5])
25
>>> sum_every_other([8, 15, 20, -1, -5, 4, 2, -5, 15])
40
```

5.3 Part #3 - Recursive `sum_first_k()`

Write a recursive function called `sum_first_k()` that takes two arguments, a list of integers and an integer k, and returns the sum of the first k integers in the given list. If k is larger then the length of the given list, then a `ValueError` should be thrown.

Below are some examples of how your function should behave on an Interpreter.

```
>>> sum_first_k([7, 20, -5, 14, 3, -10, 0, 5], 2)
27
>>> sum_first_k([7, 20, -5, 14, 3, -10, 0, 5], 4)
36
>>> sum_first_k([7, 20, -5, 14, 3, -10, 0, 5], 6)
29
>>> sum_first_k([7, 20, -5, 14, 3, -10, 0, 5], 10)
...
ValueError: k exceeds len(vals).
>>> sum_first_k([], 0)
0
```

5.4 Part #4 - Recursive `echo()`

Write a recursive function called `echo()` that takes as argument a string and returns a copy of that string with each character duplicated right after it.

Below are some examples of how your function should behave on an Interpreter. Note that unless you `print()` the return value when calling your function, the Python interpreter will display the returned strings with single quotation marks around them. Your function should not print anything.

```
>>> echo("abc")
'aabbcc'
>>> print(echo("abc"))
aabbcc
>>> echo("xy")
'xxyy'
>>> echo("hi there")
'hhii  tthheerree'
>>> echo("")
''
```

5.5 Part #5 - Recursive `reverse()`

Write a recursive function called `reverse()` that takes as argument a Python list and returns a copy of that list with its values in reversed order. You cannot modify the original list, and you cannot use any built-in function/method that would make this problem trivial.

Below are some examples of how your function should behave on an Interpreter.

```
>>> reverse([5,8,3])
[3, 8, 5]
>>> reverse([14,-7,2,80,0,5])
[5, 0, 80, 2, -7, 14]
>>> reverse([])
[]
>>> vals = [5,8,3]
>>> reverse(vals)
[3, 8, 5]
>>> vals
[5, 8, 3]
```

5.6 Part #6 - Recursive `foo`

Write a recursive function called `foo` that takes as argument two Python lists of integers. The function should raise a `ValueError` of the lists are not of the same length. The function should return `True` if for each index i (in the range of indices supported by either list), the sum of the value at index i in the first list and the value at index i in the second list equals 10. Otherwise, the function should return `False`.

Below are some examples of how your function should behave on an Interpreter.

```
>>> foo([5,2,-5],[5,8,15])
True
>>> foo([5,2,-5],[5,8,16])
False
>>> foo([5,-2,-5],[5,8,15])
False
>>> foo([5,2,-5],[5,8])
...
ValueError: The given lists have different lengths.
>>> foo([],[])
True
```

5.7 Part #7 - Recursive `UnorderedList size()` and `remove()`

Just so it's clear, for this part, you are modifying `unordered_list.py`.

`unordered_list.py` on Canvas: On Canvas, you can find the linked list implementation of `class UnorderedList` that we went over during lecture in a file called `unordered_list.py`. If you submit that file without changes to the autograder, you will probably get most of the test cases correct for this part, and when we review your code and see that you did not make the changes asked for below, we will remove all of those points that you got from the autograder.

About the `if` statement at the bottom of `unordered_list.py`: If you look at the bottom of `unordered_list.py`, you will see many lines indented into the body of this if statement: `if __name__ == "__main__":`. This `if` statement causes two things: a) if `unordered_list.py` is run (e.g. if you do Run Module in this file on Python IDLE or the equivalent of run in other IDEs), then the lines in said `if` statement will be executed, and b) if you `import` anything (i.e. `class UnorderedList`, as the autograder will do) from `unordered_list.py`, then the lines in the `if` statement will be *ignored*. This allows you to place convenient test code at the bottom of your file without having to remove it when submitting to Gradescope.

`size()` **and** `remove()`: For this part, you will use recursion to do what these methods normally do. ***To be clear***, `size()` and `remove()` will ***not*** be recursive methods[1]; you will write helper recursive methods. In my implementation, I have `size(self)` call a new recursive method `_size(self, current)`, and this method recursively calls itself as is necessary. (I'm explicitly specifying all of the parameters for clarity about how the methods' definitions are.) Similarly, I have `remove(self, item)` call a new recursive method `_remove(self, previous, current, item)` that recursively calls itself as is necessary. Your `_size()` and `_remove()` don't have to have the same parameters (or even the same names) as mine; all that matters is that those methods are recursive and that you do not modify the parameters of `size()` or `remove()`.

You are not allowed to modify the linked list implementation to have a `num_items`, i.e. your `size()` method must still traverse the entire linked list. You should not need to modify anything in `unordered_list.py` other than `size()` and `remove()` and the helper recursive methods[2] that you add.

You may assume that `remove()` will only be called on a target that is in the list.

Below is the output that should appear if you run the `unordered_list.py` file that is on Canvas.

```
54 26 93 17 77 31
6
100 54 26 93 17 77 31
7
100 26 93 17 77 31
6
100 26 17 77 31
5
100 26 17 77
4
```

[1]You can have them be (and ignore the recommendation about the helper recursive methods) if you want, but it would probably require you to add a class member, which I don't think is good form; specifically, I don't think it's good to add some "temporary" class member whose purpose is to effectively serve as a local variable for one method in the class.

[2]The helper recursive method for `size()` doesn't need to be a method, although the one for `remove()` does, since it might need to modify the `head` member of the `UnorderedList` instance. In any event, I recommend that you make the helper recursive function for `size()` be a method anyways, so as to avoid confusion.