Lab 07: Recursion and Introduction to Classes

Due: Tuesday, February 27th, 11:59pm (SD time)

Starter Files

Download lab07.zip Inside the archive, you will find starter files for the questions in this lab. You can't import anything to solve the problems.

Extra Credit Opportunity

You have the opportunity to receive a **1 point** extra credit on that lab if you submit your last attempt early (refer to this section of each lab for the early submission deadline). Note: Each lab is graded out of 10 points, and you could possibly have more than 10 points in one lab.

Early Submission Date (lab7): Sunday, February 25th, 11:59pm (SD time)

Introduction:

So far most of the problems we solved used the same pattern:

```
if condition:
    Base case (maybe more cases)
else:
    Recursive step (maybe multiple)
```

But sometimes a recursive step can be placed before any actions are taken. The following example takes you through the logic of this approach.

Example:

Write a **recursive** function that takes a list of integers, and returns a new list with each element doubled. In the first recursion lecture, we have introduced a solution:

```python
def double_list(lst):
    if len(lst) == 0:
        return []
    return [lst[0] * 2] + double_list(lst[1:])
```

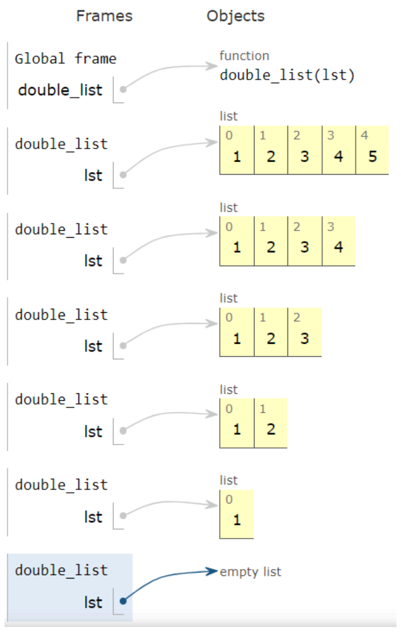
The concern for this solution is time complexity. When Python concatenates two lists using the + operation, it essentially makes a new list and copies all elements one by one. (remember, “+” returns a copy). This makes each recursive call copy the entire length of the current list, which brings the time complexity from O(n) to O(n^2) (see Notes.1).

Is there an approach that doesn’t involve copying? Yes, if we remember that the built-in `append()` mutates a list. In the following solution, we made the recursive call to get “the remaining list doubled” first, and then added the current number to the list.

```python
def double_list(lst):
    if len(lst) == 0:
        return []

    remaining = double_list(lst[:-1])  # get remaining information first
    remaining.append(lst[-1] * 2)  # adds to the growing list in O(1)
    return remaining
```

In the environment diagram, consecutive recursion calls are made without processing any information first, until it gets to the base case, where an empty list is returned. Then we are always appending to the same list object and returning that object. In total, only 1 new list object is created, and all appends happen in O(1). You may use Python Tutor to visualize the code execution.



Important Takeaway: Recursive calls don't have to be in the return statement, but can be anywhere, especially if you need information from the remaining list elements to process the result for this call.

Notes:

1. In this class, we won't make restrictions on how the recursion questions need to be solved or the time complexity of your solution, but some questions might be solved only if you make recursive call(s) first.
2. You will learn about the reason and mathematical derivations in DSC 40B, time complexity questions in this class will not involve tricky unrolling recursions. You will at most be expected to know recursion time complexity regarding the number of calls (i.e. each call will be O(1));
3. The total complexity of the solution above is still O(n^2), instead of the expected O(n). If you're interested, read on to the Optional part;
4. In the real-world setting, you will need to choose from recursive functions and iterative functions. This example shows that in the doubling list situation, looping through the list is much easier than trying to craft a recursive function of the same time complexity, due to some Python details. Some algorithms, however, are much easier when you think recursively, as you will learn in DSC 30 and DSC 40B.

(Optional) Another detail occurs on the line of recursive call, because list slicing is taking linear time. In order to make the total time complexity O(n), you will need to introduce another argument indicating the index of the number processed in the current call to avoid slicing. This is not possible in our questions because in most cases if you're not allowed to use a helper function, the header of the function will only be `double_list(lst)`

```python
def double_list(lst, cur_idx=None):
    if cur_idx is None:
        cur_idx = len(lst) - 1
    elif cur_idx < 0:  # modified base case
        return []

    remaining = double_list(lst, cur_idx - 1)  # get remaining information first
    remaining.append(lst[cur_idx])  # adds to the growing list in O(1)
    return remaining
```

Testing

After finishing each question, run the file with the following command-line options to avoid compile time errors and test the correctness of your implementation:

- No options: `>>> python3 lab07.py`
- Interactive mode: `>>> python3 -i lab07.py`
- Doctest **(Recommended)**: `>>> python3 -m doctest lab07.py`

For Windows users, please use `py` or `python` instead of `python3`.

Required Questions

DO NOT IMPORT ANY PACKAGES.

Tradition that I skipped in class this quarter: Before each recursion lecture we (tutors and I) perform a little skit in front of the class. Due to the holiday last Friday, I had to skip it but I'd like the tradition to continue. So this is a link to one of the past skits. It is optional but this class may have references to it in the labs, homeworks and exams (for fun, mostly): Link

After talking to a dragon in a dungeon, he told you about a Land of Recursion in Pythonia, where dragons live. He warned you to be careful there and start by visiting just one dragon first.

Question 1:

You want to visit the oldest and the wisest dragon, so that you can learn from its experience.

Question 1.1:

Before you solve the proposed question, let's simplify it and solve the classic problem of finding a maximum element in a list recursively. Write a **recursive** function that takes **a non-empty list** of integers and returns the maximum value in the list.

Requirements:

- No built-in functions (len() is fine).
- The function must be recursive.
- No loops/list comprehension/map/filter allowed. No helper/inner function allowed.

Hint:

- You might need to do the recursive call first, and do some processing after.

```
def max_recursion(lst):
    """
    >>> max_recursion([1, 2, 3, 4])
    4
    >>> max_recursion([3, 2, 4, 2])
    4
    >>> max_recursion([6, 7, 8, 2, 5])
    8
    """
```

Question 1.2:

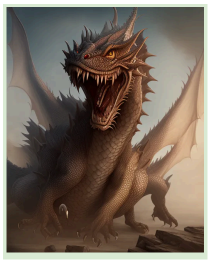
Now, using the ideas (do not use 1.1 as a helper), solve the original problem. Write a **recursive** function that takes a **non-empty** list of tuples, where the first item in a tuple is a dragon name and another one is its age. Your function should return a tuple with the oldest dragon. In case there is a tie, return the first tuple.

Requirements:

- No built-in functions (len() is fine).
- The function must be recursive.
- No loops/list comprehension/map/filter allowed. No helper/inner function allowed.

```
def oldest_dragon(all_dragons):
    """
    >>> oldest_dragon([('Skakmat', 10)])
    ('Skakmat', 10)
    >>> oldest_dragon([('Skakmat', 5), ('Alduin', 17)])
    ('Alduin', 17)
    >>> oldest_dragon([('Skakmat', 17), ('Alduin', 17)])
    ('Skakmat', 17)
    """
```

Question 2:



One piece of advice from the wise dragon was to avoid the place where dragons with the same name live; they are mean and hungry. You want to record the distribution of dragons in the Land of Recursion so that you know which areas to avoid.

Write a **recursive** function that takes a list of dragon names as strings. Return a dictionary recording the frequency of each dragon name (how many times it appears in the list).

Requirements:

- No built-in functions (len() is fine).
- The function must be recursive.
- No loops/list comprehension/map/filter allowed. No helper/inner function allowed.

Hint:

- Try to think of the easiest case first, and build your recursive call upon that;
- You might need to do the recursive call first, and do some processing after.

Note:

1. The order of dictionary keys should not matter, so your output should have output in any order;
2. In doctests, we use dict(sorted(d.items())) to sort the keys, so any correct answer will have the same format when testing. You don't need to worry about this.

```
def dragon_count(plants):
    """
    >>> dragon_count(['Skakmat', 'Alduin', 'Alduin'])
    {'Alduin': 2, 'Skakmat': 1}
    >>> dragon_count([])
    {}
    """
```

Question 3:

The rumor about your visit to the recursion land is spreading and many dragons want to meet you in person, although you only want to invite some of them.

Write a **recursive** function that takes a list of all dragons who want to meet you and another list of dragons that you wish to invite (both may contain duplicates since different dragons may have the same name). Return a list with the unique names of dragons that you wish to invite and who want to see you as well. The order of the names doesn't matter. Strings are case-sensitive.

Note: In doctests, we used `sorted()` to sort the values, so any correct answer will have the same format when testing. You don't need to worry about this.

Requirements:
- No built-in functions (len(), `in`, 'not`, `.append()` are fine).
- The function must be recursive.
- No loops/list comprehension/map/filter allowed. No helper/inner function allowed (i.e. remove(), count(), index(),pop() etc)

```
def send_invitations(all_dragons, my_choices):
    """
    >>> all_dragons = ['Skakmat', 'Alduin', 'Skakmat', 'Dracora',
'Dracar']
    >>> send_invitations(all_dragons, [])
    []

    >>> send_invitations(all_dragons, ['Alduin'])
    ['Alduin']

    >>> all_dragons = ["Dracora", "Dracar", "Dracar", "Alduin",
"Pyrion"]
    >>> sorted(send_invitations(all_dragons, ["Dracar", "Dracar",
"Pyrion"]))
```

```
    ['Dracar', 'Pyrion']

    >>> sorted(send_invitations(all_dragons, ["Pyrion", "Dracar",
"Talonx"]))
    ['Dracar', 'Pyrion']
    """
```

Question 4:

You have just received all the RSVP responses from the dragons you invited so it is time to see who is coming.
Write a **recursive** algorithm that checks if their response contains a substring (such as ‘yes’ in their RSVP) (**case-sensitive**).

Inputs:
- `string`: input string.
- `sub`: substring you are checking for.
- i: index from which you will start searching the string for the substring.
- j: index of the letter in the substring you are looking for.

Output: `True` if substring is in the string, `False` otherwise.

Requirement:
- No built-in functions (len() is fine). In particular, You can **not** use .index() or .find(), must look for characters **recursively**.
- No loops/list comprehension/map/filter allowed. No helper/inner function allowed.

Example	**Output**	**Explanation**
>>> rsvp('yes, will be there', 'yes') True	**Step 1:** i=0, j=0 'yes, will be there' ‘yes’ **Step 2:** i=1, j=1 'yes, will be there' ‘yes’ **Step 3:** i=2, j=2	1. We are using default parameters so we start with i=0 and j=0, this means that we will start searching for ‘y’ first. Since ‘y’ in the string matches ‘y’ in the substring, we move on to the next character in the string and the next character in the substring. 2. ‘e’ in the string matches with ‘e’ in the substring, so we move on to the next character in the string and the substring. 3. ‘s’ in the string matches with ‘s’ in the substring, so we move on to the next character in the string and the substring. 4. Since there are no more characters in the substring

	'yes, will be there' 'yes' **Step 4:** i=3, j=3 'yes, will be there' 'yes'	left to look at, we can conclude that we found all characters in the substring occurring in a row, so we can return `True`
>>> rsvp('yep, will be there', 'yes') False	**Step 1:** i=0, j=0 'yep, will be there' 'yes' **Step 2:** i=1, j=1 'yep, will be there' 'yes' **Step 3:** i=2, j=2 'yep, will be there' 'yes' **Step 4:** i=3, j=0 'yep, will be there' 'yes' 'yep, will be there' 'yes' **Step 5:** *repeat prior steps starting from the next character in the string and the first character in the substring	1. We are using default parameters so we start with i=0 and j=0, this means that we will start searching for 'y' first. Since 'y' in the string matches 'y' in the substring, we move on to the next character in the string and the next character in the substring. 2. 'e' in the string matches with 'e' in the substring, so we move on to the next character in the string and the substring 3. 'p' in the string DOES NOT match with 's' in the substring, so we move on to the next character in the string and start looking for the the substring from the beginning (reset j to 0, but not i) 4. Now we start looking at the next character in the string to see if it matches the first character in our substring.
>>> rsvp('word', 'or') True		● "i" is the index of the string that you are looking at ● "j" is the index of the substring that you are looking at. 1. start with i=0, j=0 and compare them : 'w' and 'o' a. since they are not equal, we are still searching for the match to index 0 of substring but we move on

		to the next character in the string, so we update i=1 and stays j=0 b. now we compare 'o' from 'word' and 'o' from 'or' since they are a match then we update i=2 and j=1 and then we look for 'r'...

```
def rsvp(string, sub, i=0, j=0):
    """
    >>> rsvp('yes, will be there', 'yes')
    True
    >>> rsvp('no, sorry', 'yes')
    False
    >>> rsvp('yep, will be there', 'yes')
    False
    """
```

Question 5: Fill in the blanks

You've seen a lot of dragons at your party. You've noticed that they are similar in one way but different in another. It's time to organize your findings in code!

Implement a class called `Dragon` that creates a dragon object. You may add your own instance attributes if needed.
You need to fill in the ***** with the lines of code. The details about the class are given below:

Required Class attributes:	
`greeting`	(str) the greeting that all dragons share
Required Instance attributes:	
`name`	(str) name of the dragon
`vocabulary`	(list of strings) words that dragon knows

energy	(int) initialized to 1000
Required Instance Methods:	
shout(self, amount)	This method returns the first amount of words (inclusive) separated by a dash, but every shout uses 500 energy points. If there is not enough energy, return False. Make sure to update the energy attribute when shout() is successful and do **NOT** update it when it fails. If the amount is larger than the number of all vocabulary, use all of them.
add_new_word(self, word)	This method adds a new word to the vocabulary list as the **first** element.

```
class Dragon:
    """
    >>> dragon1 = Dragon('Skakmat', ['Hun', 'Vah', 'Koor'])
    >>> dragon1.greeting
    'I arrived from the land of recursion!'
    >>> dragon1.shout(2)
    'Hun-Vah'
    >>> dragon1.energy
    500
    >>> dragon1.add_new_word('Lok')
    >>> dragon1.vocabulary
    ['Lok', 'Hun', 'Vah', 'Koor']
    >>> dragon1.shout(3)
    'Lok-Hun-Vah'
    >>> dragon1.shout(1)
    False

    >>> dragon2 = Dragon('Pyrion', [])
    >>> Dragon.greeting
    'I arrived from the land of recursion!'
    >>> dragon2.shout(1)
    ''
    """
    greeting = '*****'

    def __init__(self, name, *****):
        self.***** = *****
        self.***** = *****
        self.energy = *****

    def shout(*****, amount):
        # TODO: Implement this method
        return

    def add_new_word(*****, *****):
        self.vocabulary = *****
```

Submission

By the end of this lab, you should have submitted the lab via Gradescope. You may submit more than once before the deadline; only the final submission will be graded.