Lab 5: Trees **lab05.zip (lab05.zip)**

Due at 11:59pm on Friday, 03/01/2019.

Starter Files

Download lab05.zip (lab05.zip). Inside the archive, you will find starter files for the questions in this lab, along with a copy of the Ok (ok) autograder.

Submission

By the end of this lab, you should have submitted the lab with `python3 ok --submit`. You may submit more than once before the deadline; only the final submission will be graded. Check that you have successfully submitted your code on okpy.org (https://okpy.org/).

Topics

Consult this section if you need a refresher on the material for this lab. It's okay to skip directly to the questions and refer back here should you get stuck.

Sequences

Trees

Checkoff

Q1: Partition Options

Recall the `count_partitions` function from lecture:

```
>>> def count_partitions(total, biggest_num):
...     if total == 0:
...         return 1
...     elif total < 0:
...         return 0
...     elif biggest_num == 0:
...         return 0
...     else:
...         with_biggest = count_partitions(total - biggest_num, biggest_num)
...         without_biggest = count_partitions(total, biggest_num - 1)
...         return with_biggest + without_biggest
...
```

If you don't understand what's going on, feel free to flag down an AI or TA to explain. Once you're confident that you understand, consider the following modification. `partition_options` returns all possible partitions of `total` using numbers no biggest than `biggest_num`. `count_partitions` returned the number of possible partitions, but here we are going to actually return all the possible unique sums that sum tot `total`. Some examples:

```
>>> partition_options(2, 1)
[[1, 1]]
>>> partition_options(2, 2)
[[2], [1, 1]]
>>> partition_options(3, 3)
[[3], [2, 1], [1, 1, 1]]
>>> partition_options(4, 3)
[[3, 1], [2, 2], [2, 1, 1], [1, 1, 1, 1]]
>>> partition_options(6, 4)
[[4, 2], [4, 1, 1], [3, 3], [3, 2, 1], [3, 1, 1, 1], [2, 2, 2], [2, 2, 1, 1], [2, 1, 1, 1
```

Now implement `partition_options`, using `count_partitions` as a reference. If you would like to be checked off, put yourself on the checkoff queue in lab.

```
>>> def partitions_options(total, biggest_num):
...     if total == 0:
...         return [[]]
...     elif total < 0:
...         return []
...     elif biggest_num == 0:
...         return []
...     else:
...         with_biggest = ____________
...         without_biggest = ____________
...         ___________ = [[________] _______ for elem in with_biggest] # what extra step
...         return with_biggest + without_biggest
...
```

Required Questions

Q2: WWPD: Lists?

What would Python display? Try to figure it out before you type it into the interpreter!

> Use Ok to test your knowledge with the following "What Would Python Display?" questions:
>
> ```
> python3 ok -q lists -u
> ```

```
>>> [x*x for x in range(5)]
------

>>> [n for n in range(10) if n % 2 == 0]
------

>>> [i+5 for i in [n for n in range(1,4)]]
------
```

```
>>> [i**2 for i in range(10) if i < 3]
------

>>> lst = ['hi' for i in [1, 2, 3]]
>>> print(lst)
------

>>> lst + [i for i in ['1', '2', '3']]
------
```

Q3: If This Not That

Define `if_this_not_that`, which takes a list of integers `i_list` and an integer `this`. For each element in `i_list`, if the element is larger than `this`, then print the element. Otherwise, print `"that"`.

```
def if_this_not_that(i_list, this):
    """Define a function which takes a list of integers `i_list` and an integer
    `this`. For each element in `i_list`, print the element if it is larger
    than `this`; otherwise, print the word "that".

    >>> original_list = [1, 2, 3, 4, 5]
    >>> if_this_not_that(original_list, 3)
    that
    that
    that
    4
    5
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q if_this_not_that
```

Q4: Acorn Finder

The squirrels on campus need your help! There are a lot of trees on campus and the squirrels would like to know which ones contain acorns. Define the function `acorn_finder`, which takes in a tree and returns `True` if the tree contains a node with the value `'acorn'` and `False` otherwise.

```
def acorn_finder(t):
    """Returns True if t contains a node with the value 'acorn' and
    False otherwise.

    >>> scrat = tree('acorn')
    >>> acorn_finder(scrat)
    True
    >>> sproul = tree('roots', [tree('branch1', [tree('leaf'), tree('acorn')]), tree('brai
    >>> acorn_finder(sproul)
    True
    >>> numbers = tree(1, [tree(2), tree(3, [tree(4), tree(5)]), tree(6, [tree(7)])])
    >>> acorn_finder(numbers)
    False
    >>> t = tree(1, [tree('acorn',[tree('not acorn')])])
    >>> acorn_finder(t)
    True
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q acorn_finder
```

Optional Questions

Shakespeare and Dictionaries

We will use dictionaries to approximate the entire works of Shakespeare! We're going to use a bigram language model. Here's the idea: We start with some word -- we'll use "The" as an example. Then we look through all of the texts of Shakespeare and for every instance of "The" we record the word that follows "The" and add it to a list, known as the *successors* of "The". Now suppose we've done this for every word Shakespeare has used, ever.

Let's go back to "The". Now, we randomly choose a word from this list, say "cat". Then we look up the successors of "cat" and randomly choose a word from that list, and we continue this process. This eventually will terminate in a period (".") and we will have generated a Shakespearean sentence!

The object that we'll be looking things up in is called a "successor table", although really it's just a dictionary. The keys in this dictionary are words, and the values are lists of successors to those words.

Q5: Successor Tables

Here's an incomplete definition of the `build_successors_table` function. The input is a list of words (corresponding to a Shakespearean text), and the output is a successors table. (By default, the first word is a successor to "."). See the example below.

> Note: there are two places where you need to write code, denoted by the two `"*** YOUR CODE HERE ***"`

```
def build_successors_table(tokens):
    """Return a dictionary: keys are words; values are lists of successors.

    >>> text = ['We', 'came', 'to', 'investigate', ',', 'catch', 'bad', 'guys', 'and', 't
    >>> table = build_successors_table(text)
    >>> sorted(table)
    [',', '.', 'We', 'and', 'bad', 'came', 'catch', 'eat', 'guys', 'investigate', 'pie',
    >>> table['to']
    ['investigate', 'eat']
    >>> table['pie']
    ['.']
    >>> table['.']
    ['We']
    """
    table = {}
    prev = '.'
    for word in tokens:
        if prev not in table:
            "*** YOUR CODE HERE ***"
        "*** YOUR CODE HERE ***"
        prev = word
    return table
```

Use Ok to test your code:

```
python3 ok -q build_successors_table
```

Q6: Construct the Sentence

Let's generate some sentences! Suppose we're given a starting word. We can look up this word in our table to find its list of successors, and then randomly select a word from this list to be the next word in the sentence. Then we just repeat until we reach some ending punctuation.

> *Hint*: to randomly select from a list, import the Python random library with `import random` and use the expression `random.choice(my_list)`

This might not be a bad time to play around with adding strings together as well. Let's fill in the `construct_sent` function!

```
def construct_sent(word, table):
    """Prints a random sentence starting with word, sampling from
    table.

    >>> table = {'Wow': ['!'], 'Sentences': ['are'], 'are': ['cool'], 'cool': ['.']}
    >>> construct_sent('Wow', table)
    'Wow!'
    >>> construct_sent('Sentences', table)
    'Sentences are cool.'
    """
    import random
    result = ''
    while word not in ['.', '!', '?']:
        "*** YOUR CODE HERE ***"
    return result.strip() + word
```

Use Ok to test your code:

```
python3 ok -q construct_sent
```

Putting it all together

Great! Now let's try to run our functions with some actual data. The following snippet included in the skeleton code will return a list containing the words in all of the works of Shakespeare.

> *Warning*: Do **NOT** try to print the return result of this function.

```
def shakespeare_tokens(path='shakespeare.txt', url='http://composingprograms.com/shakespea
    """Return the words of Shakespeare's plays as a list."""
    import os
    from urllib.request import urlopen
    if os.path.exists(path):
        return open('shakespeare.txt', encoding='ascii').read().split()
    else:
        shakespeare = urlopen(url)
        return shakespeare.read().decode(encoding='ascii').split()
```

Uncomment the following two lines to run the above function and build the successors table from those tokens.

```
# Uncomment the following two lines
# tokens = shakespeare_tokens()
# table = build_successors_table(tokens)
```

Next, let's define a utility function that constructs sentences from this successors table:

```
>>> def sent():
...     return construct_sent('The', table)
>>> sent()
" The plebeians have done us must be news-cramm'd."

>>> sent()
" The ravish'd thee , with the mercy of beauty!"

>>> sent()
" The bird of Tunis , or two white and plucker down with better ; that's God's sake."
```

Notice that all the sentences start with the word "The". With a few modifications, we can make our sentences start with a random word. The following `random_sent` function (defined in your starter file) will do the trick:

```
def random_sent():
    import random
    return construct_sent(random.choice(table['.']), table)
```

Go ahead and load your file into Python (be sure to use the `-i` flag). You can now call the `random_sent` function to generate random Shakespearean sentences!

```
>>> random_sent()
' Long live by thy name , then , Dost thou more angel , good Master Deep-vow , And tak'st

>>> random_sent()
' Yes , why blame him , as is as I shall find a case , That plays at the public weal or tl
```

More Trees Practice

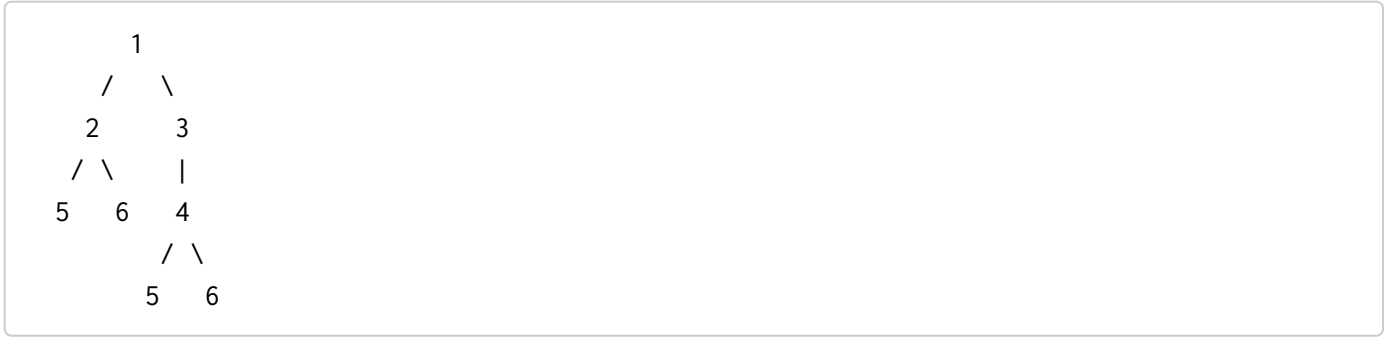
Q7: Sprout leaves

Define a function `sprout_leaves` that takes in a tree, `t`, and a list of values, `vals`. It produces a new tree that is identical to `t`, but where each old leaf node has new branches, one for each value in `vals`.

For example, say we have the tree `t = tree(1, [tree(2), tree(3, [tree(4)])])`:

```
  1
 / \
2   3
    |
    4
```

If we call `sprout_leaves(t, [5, 6])`, the result is the following tree:




```
def sprout_leaves(t, vals):
    """Sprout new leaves containing the data in vals at each leaf in
    the original tree t and return the resulting tree.

    >>> t1 = tree(1, [tree(2), tree(3)])
    >>> print_tree(t1)
    1
      2
      3
    >>> new1 = sprout_leaves(t1, [4, 5])
    >>> print_tree(new1)
    1
      2
        4
        5
      3
        4
        5

    >>> t2 = tree(1, [tree(2, [tree(3)])])
    >>> print_tree(t2)
    1
      2
        3
    >>> new2 = sprout_leaves(t2, [6, 1, 2])
    >>> print_tree(new2)
    1
      2
        3
          6
          1
          2
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q sprout_leaves
```

Q8: Add trees

Define the function `add_trees`, which takes in two trees and returns a new tree where each corresponding node from the first tree is added with the node from the second tree. If a node at any particular position is present in one tree but not the other, it should be present in the new tree as well.

> *Hint*: You may want to use the built-in zip function to iterate over multiple sequences at once.
>
> *Note*: If you feel that this one's a lot harder than the previous tree problems, that's totally fine! This is a pretty difficult problem, but you can do it! Talk about it with other students, and come back to it if you need to.

```
def add_trees(t1, t2):
    """
    >>> numbers = tree(1,
    ...                [tree(2,
    ...                      [tree(3),
    ...                       tree(4)]),
    ...                 tree(5,
    ...                      [tree(6,
    ...                            [tree(7)]),
    ...                       tree(8)])])
    >>> print_tree(add_trees(numbers, numbers))
    2
      4
        6
        8
      10
        12
          14
        16
    >>> print_tree(add_trees(tree(2), tree(3, [tree(4), tree(5)])))
    5
      4
      5
    >>> print_tree(add_trees(tree(2, [tree(3)]), tree(2, [tree(3), tree(4)])))
    4
      6
      4
    >>> print_tree(add_trees(tree(2, [tree(3, [tree(4), tree(5)])]), \
    tree(2, [tree(3, [tree(4)]), tree(5)])))
    4
      6
        8
        5
      5
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q add_trees
```

CS 61A (/)

Weekly Schedule (/weekly.html)

Office Hours (/office-hours.html)

Staff (/staff.html)

Resources (/resources.html)

Studying Guide (/articles/studying.html)

Debugging Guide (/articles/debugging.html)

Composition Guide (/articles/composition.html)

Policies (/articles/about.html)

Assignments (/articles/about.html#assignments)

Exams (/articles/about.html#exams)

Grading (/articles/about.html#grading)