Lab 5 Solutions **lab05.zip (lab05.zip)**

Solution Files

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.

Required Questions

Lists Practice

Q1: List Indexing

Use Ok to test your knowledge with the following "List Indexing" questions:

```
python3 ok -q list-indexing -u
```

For each of the following lists, what is the list indexing expression that evaluates to `7`? For example, if `x = [7]`, then the answer would be `x[0]`. You can use the interpreter or Python Tutor to experiment with your answers. If the code would cause an error, type `Error`.

```
>>> x = [1, 3, [5, 7], 9]
------

>>> x = [[3, [5, 7], 9]]
------
```

What would Python display? If you get stuck, try it out in the Python interpreter!

```
>>> lst = [3, 2, 7, [84, 83, 82]]
>>> lst[4]
------

>>> lst[3][0]
------
```

Q2: Couple

Implement the function `couple`, which takes in two lists and returns a list that contains lists with i-th elements of two sequences coupled together. You can assume the lengths of two sequences are the same. Try using a list comprehension.

> *Hint*: You may find the built in range (https://www.w3schools.com/python/ref_func_range.asp) function helpful.

```
def couple(lst1, lst2):
    """Return a list that contains lists with i-th elements of two sequences
    coupled together.
    >>> lst1 = [1, 2, 3]
    >>> lst2 = [4, 5, 6]
    >>> couple(lst1, lst2)
    [[1, 4], [2, 5], [3, 6]]
    >>> lst3 = ['c', 6]
    >>> lst4 = ['s', '1']
    >>> couple(lst3, lst4)
    [['c', 's'], [6, '1']]
    """
    assert len(lst1) == len(lst2)
    return [[lst1[i], lst2[i]] for i in range(0, len(lst1))]
```

Use Ok to test your code:

```
python3 ok -q couple
```

City Data Abstraction

Say we have an abstract data type for cities. A city has a name, a latitude coordinate, and a longitude coordinate.

Our ADT has one **constructor**:

- `make_city(name, lat, lon)` : Creates a city object with the given name, latitude, and longitude.

We also have the following **selectors** in order to get the information for each city:

- `get_name(city)` : Returns the city's name
- `get_lat(city)` : Returns the city's latitude
- `get_lon(city)` : Returns the city's longitude

Here is how we would use the constructor and selectors to create cities and extract their information:

```
>>> berkeley = make_city('Berkeley', 122, 37)
>>> get_name(berkeley)
'Berkeley'
>>> get_lat(berkeley)
122
>>> new_york = make_city('New York City', 74, 40)
>>> get_lon(new_york)
40
```

All of the selector and constructor functions can be found in the lab file, if you are curious to see how they are implemented. However, the point of data abstraction is that we do not need to know how an abstract data type is implemented, but rather just how we can interact with and use the data type.

Q3: Distance

We will now implement the function `distance`, which computes the distance between two city objects. Recall that the distance between two coordinate pairs `(x1, y1)` and `(x2, y2)` can be found by calculating the `sqrt` of `(x1 - x2)**2 + (y1 - y2)**2`. We have already imported `sqrt` for your convenience. Use the latitude and longitude of a city as its coordinates; you'll need to use the selectors to access this info!

```
from math import sqrt
def distance(city1, city2):
    """
    >>> city1 = make_city('city1', 0, 1)
    >>> city2 = make_city('city2', 0, 2)
    >>> distance(city1, city2)
    1.0
    >>> city3 = make_city('city3', 6.5, 12)
    >>> city4 = make_city('city4', 2.5, 15)
    >>> distance(city3, city4)
    5.0
    """
    lat_1, lon_1 = get_lat(city1), get_lon(city1)
    lat_2, lon_2 = get_lat(city2), get_lon(city2)
    return sqrt((lat_1 - lat_2)**2 + (lon_1 - lon_2)**2)

# Video walkthrough: https://youtu.be/4vB06Ymrico
```

Use Ok to test your code:

```
python3 ok -q distance
```

Q4: Closer city

Next, implement `closer_city`, a function that takes a latitude, longitude, and two cities, and returns the name of the city that is relatively closer to the provided latitude and longitude.

You may only use the selectors and constructors introduced above and the `distance` function you just defined for this question.

> **Hint**: How can use your `distance` function to find the distance between the given location and each of the given cities?

```
def closer_city(lat, lon, city1, city2):
    """
    Returns the name of either city1 or city2, whichever is closest to
    coordinate (lat, lon).

    >>> berkeley = make_city('Berkeley', 37.87, 112.26)
    >>> stanford = make_city('Stanford', 34.05, 118.25)
    >>> closer_city(38.33, 121.44, berkeley, stanford)
    'Stanford'
    >>> bucharest = make_city('Bucharest', 44.43, 26.10)
    >>> vienna = make_city('Vienna', 48.20, 16.37)
    >>> closer_city(41.29, 174.78, bucharest, vienna)
    'Bucharest'
    """
    new_city = make_city('arb', lat, lon)
    dist1 = distance(city1, new_city)
    dist2 = distance(city2, new_city)
    if dist1 < dist2:
         return get_name(city1)
    return get_name(city2)

# Video walkthrough: https://youtu.be/Owwdz_Km87g
```

Use Ok to test your code:

```
python3 ok -q closer_city
```

Q5: Don't violate the abstraction barrier!

> Note: this question has no code-writing component (if you implemented `distance` and `closer_city` correctly!)

When writing functions that use an ADT, we should use the constructor(s) and selector(s) whenever possible instead of assuming the ADT's implementation. Relying on a data abstraction's underlying implementation is known as *violating the abstraction barrier*, and we never want to do this!

It's possible that you passed the doctests for `distance` and `closer_city` even if you violated the abstraction barrier. To check whether or not you did so, run the following command:

Use Ok to test your code:

```
python3 ok -q check_abstraction
```

The `make_check_abstraction` function exists only for the doctest, which swaps out the implementations of the `city` abstraction with something else, runs the tests from the previous two parts, then restores the original abstraction.

The nature of the abstraction barrier guarantees that changing the implementation of an ADT shouldn't affect the functionality of any programs that use that ADT, as long as the constructors and selectors were used properly.

If you passed the Ok tests for the previous questions but not this one, the fix is simple! Just replace any code that violates the abstraction barrier, i.e. creating a city with a new list object or indexing into a city, with the appropriate constructor or selector.

Make sure that your functions pass the tests with both the first and the second implementations of the City ADT and that you understand why they should work for both before moving on.

Trees

Q6: Nut 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 nuts. Define the function `nut_finder`, which takes in a tree and returns `True` if the tree contains a node with the value `'nut'` and `False` otherwise.

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

    >>> scrat = tree('nut')
    >>> nut_finder(scrat)
    True
    >>> sproul = tree('roots', [tree('branch1', [tree('leaf'), tree('nut')]), tr
    >>> nut_finder(sproul)
    True
    >>> numbers = tree(1, [tree(2), tree(3, [tree(4), tree(5)]), tree(6, [tree(7
    >>> nut_finder(numbers)
    False
    >>> t = tree(1, [tree('nut',[tree('not nut')])])
    >>> nut_finder(t)
    True
    """
    if label(t) == 'nut':
        return True
    for b in branches(t):
        if nut_finder(b):
            return True
    return False

# Alternative solution
def nut_finder_alt(t):
    if label(t) == 'nut':
        return True
    return True in [nut_finder(b) for b in branches(t)]
```

Use Ok to test your code:

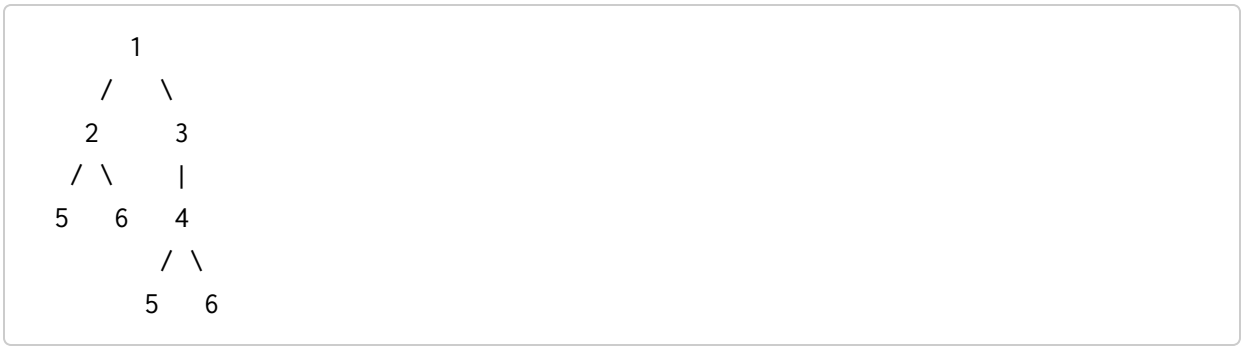
```
python3 ok -q nut_finder
```

Q7: Sprout leaves

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

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

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



```
def sprout_leaves(t, values):
    """Sprout new leaves containing the data in values 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
    """
    if is_leaf(t):
        return tree(label(t), [tree(v) for v in values])
    return tree(label(t), [sprout_leaves(s, values) for s in branches(t)])
```

Use Ok to test your code:

```
python3 ok -q sprout_leaves
```

Submit

Make sure to submit this assignment by running:

```
python3 ok --submit
```

Optional Questions

> While "Add Characters" is optional, it is good practice for the Cats project and is thus highly recommended!
>
> Note: The lab wasn't properly released (sorry), so if you downloaded the lab before 12:30PM on Monday, 2/24, please redownload the lab to get a new copy of lab05.ok and drag that file into your current folder to run ok tests for the extra questions.

Q8: Add Characters

Given two words, `w1` and `w2`, we say `w1` is a subsequence of `w2` if all the letters in `w1` appear in `w2` in the same order (but not necessarily all together). That is, you can add letters to any position in `w1` to get `w2`. For example, "sing" is a substring of "ab**s**orb**ing**" and "cat" is a substring of "**c**ontr**a**s**t**".

Implement `add_chars`, which takes in `w1` and `w2`, where `w1` is a substring of `w2`. This means that `w1` is shorter than `w2`. It should return a string containing the characters you need to add to `w1` to get `w2`. **Your solution must use recursion**.

In the example above, you need to add the characters "aborb" to "sing" to get "absorbing", and you need to add "ontrs" to "cat" to get "contrast".

The letters in the string you return should be in the order you have to add them from left to right. If there are multiple characters in the `w2` that could correspond to characters in `w1`, use the leftmost one. For example, `add_words("coy", "cacophony")` should return "acphon", not "caphon" because the first "c" in "coy" corresponds to the first "c" in "**c**ac**o**phon**y**".

```
def add_chars(w1, w2):
    """
    Return a string containing the characters you need to add to w1 to get w2.

    You may assume that w1 is a subsequence of w2.

    >>> add_chars("owl", "howl")
    'h'
    >>> add_chars("want", "wanton")
    'on'
    >>> add_chars("rat", "radiate")
    'diae'
    >>> add_chars("a", "prepare")
    'prepre'
    >>> add_chars("resin", "recursion")
    'curo'
    >>> add_chars("fin", "effusion")
    'efuso'
    >>> add_chars("coy", "cacophony")
    'acphon'
    >>> from construct_check import check
    >>> # ban iteration and sets
    >>> check(LAB_SOURCE_FILE, 'add_chars',
    ...       ['For', 'While', 'Set', 'SetComp']) # Must use recursion
    True
    """
    if not w1:
        return w2
    elif w1[0] == w2[0]:
        return add_chars(w1[1:], w2[1:])
    return w2[0] + add_chars(w1, w2[1:])
```

Use Ok to test your code:

```
python3 ok -q add_chars
```

Q9: 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
    """
    if not t1:
        return t2
    if not t2:
        return t1
    new_label = label(t1) + label(t2)
    t1_children, t2_children = branches(t1), branches(t2)
    length_t1, length_t2 = len(t1_children), len(t2_children)
    if length_t1 < length_t2:
        t1_children += [None for _ in range(length_t1, length_t2)]
    elif len(t1_children) > len(t2_children):
```

```
        t2_children += [None for _ in range(length_t2, length_t1)]
    return tree(new_label, [add_trees(child1, child2) for child1, child2 in zip
```

Use Ok to test your code:

```
python3 ok -q add_trees
```

Fun Question!

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.

Q10: 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',
    >>> table = build_successors_table(text)
    >>> sorted(table)
    [',', '.', 'We', 'and', 'bad', 'came', 'catch', 'eat', 'guys', 'investigate'
    >>> table['to']
    ['investigate', 'eat']
    >>> table['pie']
    ['.']
    >>> table['.']
    ['We']
    """
    table = {}
    prev = '.'
    for word in tokens:
        if prev not in table:
            table[prev] = []
        table[prev] += [word]
        prev = word
    return table
```

Use Ok to test your code:

```
python3 ok -q build_successors_table
```

Q11: 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 ['.', '!', '?']:
        result += word + ' '
        word = random.choice(table[word])
    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.co
    """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 s
```

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 , A

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

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)