Lab 5: Trees, Data Abstraction

lab05.zip (lab05.zip)

Due by 11:59pm on Wednesday, September 27.

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.

Required Questions

Getting Started Videos

Dictionaries

Consult the drop-down if you need a refresher on dictionaries. It's okay to skip directly to the questions and refer back here should you get stuck.

Dictionaries

Q1: Dictionaries

Use Ok to test your knowledge with the following "What Would Python Display?" questions:

```
python3 ok -q pokemon -u
```

```
>>> pokemon = {'pikachu': 25, 'dragonair': 148, 'mew': 151}
>>> pokemon['pikachu']
------

>>> len(pokemon)
------

>>> 'mewtwo' in pokemon
------

>>> 'pikachu' in pokemon
------

>>> 25 in pokemon
------

>>> 148 in pokemon.values()
------

>>> 151 in pokemon.keys()
------

>>> 'mew' in pokemon.keys()
------
```

Trees

Consult the drop-down if you need a refresher on trees. It's okay to skip directly to the questions and refer back here should you get stuck.

Trees

Tree ADT

Our `tree` abstract data type consists of a root and a list of its `branches`. To create a tree and access its root value and branches, use the following interface of constructor and selectors:

- Constructor
  - `tree(label, branches=[])`: creates a tree object with the given `label` value at its root node and list of `branches`. Notice that the second argument to this constructor, `branches`, is optional — if you want to make a tree with no branches, leave this argument empty.
- Selectors
  - `label(tree)`: returns the value in the root node of `tree`.
  - `branches(tree)`: returns the list of branches of the given `tree` (each of which is also a tree)
- Convenience function
  - `is_leaf(tree)`: returns `True` if `tree`'s list of `branches` is empty, and `False` otherwise.

For example, the tree generated by

```
number_tree = tree(1,
         [tree(2),
          tree(3,
               [tree(4),
                tree(5)]),
          tree(6,
               [tree(7)])])
```

would look like this:

```
   1
 / | \
2  3  6
  / \  \
 4   5  7
```

To extract the number `3` from this tree, which is the label of the root of its second branch, we would do this:

```
label(branches(number_tree)[1])
```

The lab file contains the implementation of the tree ADT, if you would like to view it. However, as with any data abstraction, we should only concern ourselves with what our functions do rather than their specific implementation! You *do not* need to look at the

implementation of the tree ADT for this problem, and your solutions should not depend on the specifics of our implementation beyond what is specified in the interface above.

Q2: Finding Berries!

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 berries. Define the function `berry_finder`, which takes in a tree and returns `True` if the tree contains a node with the value `'berry'` and `False` otherwise.

> **Hint**: To iterate through each of the branches of a particular tree, you can consider using a `for` loop to get each branch.

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

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

Use Ok to test your code:

```
python3 ok -q berry_finder
```

Q3: Replace Loki at Leaf

Define `replace_loki_at_leaf`, which takes a tree `t` and a value `lokis_replacement`. `replace_loki_at_leaf` returns a new tree that's the same as `t` except that every leaf label equal to `"loki"` has been replaced with `lokis_replacement`.

If you want to learn about the Norse mythology referenced in this problem, you can read about it here (https://en.wikipedia.org/wiki/Yggdrasil).

```
def replace_loki_at_leaf(t, lokis_replacement):
    """Returns a new tree where every leaf value equal to "loki" has
    been replaced with lokis_replacement.

    >>> yggdrasil = tree('odin',
    ...                  [tree('balder',
    ...                        [tree('loki'),
    ...                         tree('freya')]),
    ...                   tree('frigg',
    ...                        [tree('loki')]),
    ...                   tree('loki',
    ...                        [tree('sif'),
    ...                         tree('loki')]),
    ...                   tree('loki')])
    >>> laerad = copy_tree(yggdrasil) # copy yggdrasil for testing purposes
    >>> print_tree(replace_loki_at_leaf(yggdrasil, 'freya'))
    odin
      balder
        freya
        freya
      frigg
        freya
      loki
        sif
        freya
      freya
    >>> laerad == yggdrasil # Make sure original tree is unmodified
    True
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q replace_loki_at_leaf
```

Data Abstraction

Consult the drop-down if you need a refresher on data abstraction. It's okay to skip directly to the questions and refer back here should you get stuck.

Data Abstraction

City ADT

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

Our data abstraction 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 — as users of the city ADT — we do not need to know how an the ADT is implemented. You *do not* need to look at the implementation of the city ADT for this problem, and your solutions should not depend on the specifics of our implementation beyond what is specified in the interface above.

Q4: 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(city_a, city_b):
    """
    >>> city_a = make_city('city_a', 0, 1)
    >>> city_b = make_city('city_b', 0, 2)
    >>> distance(city_a, city_b)
    1.0
    >>> city_c = make_city('city_c', 6.5, 12)
    >>> city_d = make_city('city_d', 2.5, 15)
    >>> distance(city_c, city_d)
    5.0
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q distance
```

Q5: 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 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 you use your `distance` function to find the distance between the given location and each of the given cities?

```
def closer_city(lat, lon, city_a, city_b):
    """
    Returns the name of either city_a or city_b, whichever is closest to
    coordinate (lat, lon). If the two cities are the same distance away
    from the coordinate, consider city_b to be the closer city.

    >>> 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'
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q closer_city
```

Q6: Don't violate the abstraction barrier!

> Note: this question has no code-writing component (if you implemented the previous two questions correctly).

When writing functions that use a data abstraction, we should use the constructor(s) and selector(s) whenever possible instead of assuming the data abstraction'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 the previous questions 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_city_abstraction
```

The `check_city_abstraction` function exists only for the doctest, which swaps out the implementations of the original 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 data abstraction shouldn't affect the functionality of any programs that use that data abstraction, 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 with the appropriate constructor or selector.

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

Check Your Score Locally

You can locally check your score on each question of this assignment by running

```
python3 ok --score
```

This does NOT submit the assignment! When you are satisfied with your score, submit the assignment to Gradescope to receive credit for it.

Submit

Make sure to submit this assignment by uploading any files you've edited **to the appropriate Gradescope assignment.** For a refresher on how to do this, refer to Lab 00 (https://cs61a.org/lab/lab00/#submit-with-gradescope).

Optional Questions

> These questions are optional, but you must complete them in order to be checked off before the end of the lab period. They are also useful practice!

Trees

Q7: Recursion on Trees

Define a function `dejavu`, which takes in a tree of numbers `t` and a number `n`. It returns `True` if there is a path from the root to a leaf such that the sum of the numbers along that path is `n` and `False` otherwise.

```
def dejavu(t, n):
    """
    >>> my_tree = tree(2, [tree(3, [tree(5), tree(7)]), tree(4)])
    >>> dejavu(my_tree, 12) # 2 -> 3 -> 7
    True
    >>> dejavu(my_tree, 5) # Sums of partial paths like 2 -> 3 don't count
    False
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q dejavu
```

Q8: Hailstone Tree

We can represent the hailstone sequence as a tree in the figure below, showing the route different numbers take to reach 1. Remember that a hailstone sequence starts with a number `n`, continuing to `n/2` if `n` is even or `3n+1` if `n` is odd, ending with 1. Write a function

`hailstone_tree(n, h)` which generates a tree of height `h`, containing hailstone numbers that will reach `n`.

> **Hint:** A node of a hailstone tree will always have at least one, and at most two branches (which are also hailstone trees). Under what conditions do you add the second branch?

```
def hailstone_tree(n, h):
    """Generates a tree of hailstone numbers that will reach N, with height H.
    >>> print_tree(hailstone_tree(1, 0))
    1
    >>> print_tree(hailstone_tree(1, 4))
    1
      2
        4
          8
            16
    >>> print_tree(hailstone_tree(8, 3))
    8
      16
        32
          64
        5
          10
    """
    if ___________________________________:
        return ___________________________________
    branches = ___________________________________
    if ___________ and ___________ and ___________:
        branches += ___________________________________
    return tree(n, branches)
```

Use Ok to test your code:

```
python3 ok -q hailstone_tree
```