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

Q1: Factors List

Write `factors_list`, which takes a number `n` and returns a list of its factors in ascending order.

```
def factors_list(n):
    """Return a list containing all the numbers that divide `n` evenly, except
    for the number itself. Make sure the list is in ascending order.

    >>> factors_list(6)
    [1, 2, 3]
    >>> factors_list(8)
    [1, 2, 4]
    >>> factors_list(28)
    [1, 2, 4, 7, 14]
    """
    all_factors = []
    x = 1
    while x < n:
        if n % x == 0:
            all_factors += [x]
        x += 1

    return all_factors
```

Use Ok to test your code:

```
python3 ok -q factors_list
```

Q2: Flatten

Write a function `flatten` that takes a list and "flattens" it. The list could be a deep list, meaning that there could be a multiple layers of nesting within the list.

For example, one use case of `flatten` could be the following:

```
>>> lst = [1, [[2], 3], 4, [5, 6]]
>>> flatten(lst)
[1, 2, 3, 4, 5, 6]
```

Make sure your solution does not mutate the input list.

> **Hint**: you can check if something is a list by using the built-in `type` function. For example:
>
> ```
> >>> type(3) == list
> False
> >>> type([1, 2, 3]) == list
> True
> ```

```
def flatten(s):
    """Returns a flattened version of list s.

    >>> flatten([1, 2, 3])     # normal list
    [1, 2, 3]
    >>> x = [1, [2, 3], 4]     # deep list
    >>> flatten(x)
    [1, 2, 3, 4]
    >>> x # Ensure x is not mutated
    [1, [2, 3], 4]
    >>> x = [[1, [1, 1]], 1, [1, 1]] # deep list
    >>> flatten(x)
    [1, 1, 1, 1, 1, 1]
    >>> x
    [[1, [1, 1]], 1, [1, 1]]
    """
    if not s:
        return []
    elif type(s[0]) == list:
        return flatten(s[0]) + flatten(s[1:])
    else:
        return [s[0]] + flatten(s[1:])
```

Use Ok to test your code:

```
python3 ok -q flatten
```

Data Abstraction

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 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(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
    """
    lat_1, lon_1 = get_lat(city_a), get_lon(city_a)
    lat_2, lon_2 = get_lat(city_b), get_lon(city_b)
    return sqrt((lat_1 - lat_2)**2 + (lon_1 - lon_2)**2)
```

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 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'
    """
    new_city = make_city('arb', lat, lon)
    dist1 = distance(city_a, new_city)
    dist2 = distance(city_b, new_city)
    if dist1 < dist2:
         return get_name(city_a)
    return get_name(city_b)
```

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 the previous two questions correctly).

When writing functions that use an 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.

Trees

Q6: 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
    """
    if label(t) == 'berry':
        return True
    for b in branches(t):
        if berry_finder(b):
            return True
    return False

# Alternative solution
def berry_finder_alt(t):
    if label(t) == 'berry':
        return True
    return True in [berry_finder(b) for b in branches(t)]
```

Use Ok to test your code:

```
python3 ok -q berry_finder
```

Q7: Sprout leaves

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

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:

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

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

Use Ok to test your code:

```
python3 ok -q sprout_leaves
```

Q8: 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 an 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_abstraction
```

The `check_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.

Submit

Make sure to submit this assignment by running:

```
python3 ok --submit
```