Lab 4: Recursion, Lists, and Data Abstraction **lab04.zip (lab04.zip)**

Due at 11:59pm on Friday, 02/22/2019.

Starter Files

Download lab04.zip (lab04.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.

List Introduction

Data Abstraction

Required Questions

Lists Practice

Q1: List Indexing

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

```
python3 ok -q 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.

```
>>> 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]
______
```

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.

Q2: 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
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q distance
```

Q3: 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'
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q closer_city
```

Q4: 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.

Recursion Practice

Q5: Pascal's Triangle

Here's a part of the Pascal's trangle:

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
```

Every number in Pascal's triangle is defined as the sum of the item above it and the item that is directly to the upper left of it. Use `0` if the entry is empty.
Define the procedure `pascal(row, column)` which takes a row and a column, and finds the value at that position in the triangle. Rows and columns are zero-indexed; that is, the first row is row 0 instead of 1.

```
def pascal(row, column):
    """Returns a number corresponding to the value at that location
    in Pascal's Triangle.
    >>> pascal(0, 0)
    1
    >>> pascal(0, 5)    # Empty entry; outside of Pascal's Triangle
    0
    >>> pascal(3, 2)    # Row 4 (1 3 3 1), 3rd entry
    3
    """

    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q pascal
```

Optional Question

This question can be found in `lab04_extra.py`.

Q6: Hailstone

Recall the `hailstone` function from homework 1 (/hw/hw01/#q5). First, pick a positive integer `n` as the start. If `n` is even, divide it by 2. If `n` is odd, multiply it by 3 and add 1. Repeat this process until `n` is 1. Write a recursive version of hailstone that prints out the values of the sequence and returns the number of steps.

> *Hint:* When taking the recursive leap of faith, consider both the return value and side effect of this function.

```
this_file = __file__

def hailstone(n):
    """Print out the hailstone sequence starting at n, and return the
    number of elements in the sequence.

    >>> a = hailstone(10)
    10
    5
    16
    8
    4
    2
    1
    >>> a
    7
    >>> # Do not use while/for loops!
    >>> from construct_check import check
    >>> check(this_file, 'hailstone',
    ...       ['While', 'For'])
    True
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q hailstone
```

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)