Lab 4 Solutions **lab04.zip (lab04.zip)**

Solution Files

Required Questions

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.

Important: For all WWPD questions, type `Function` if you believe the answer is `<function...>`, `Error` if it errors, and `Nothing` if nothing is displayed.

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()
------
```

Q2: Divide

Implement `divide`, which takes two lists of positive integers `quotients` and `divisors`. It returns a dictionary whose keys are the elements of `quotients`. For each key `q`, its corresponding value is a list of all the elements of `divisors` that can be evenly divided by `q`.

> Hint: The value for each key needs be a list, so list comprehension might be useful here.

```
def divide(quotients, divisors):
    """Return a dictonary in which each quotient q is a key for the list of
    divisors that it divides evenly.

    >>> divide([3, 4, 5], [8, 9, 10, 11, 12])
    {3: [9, 12], 4: [8, 12], 5: [10]}
    >>> divide(range(1, 5), range(20, 25))
    {1: [20, 21, 22, 23, 24], 2: [20, 22, 24], 3: [21, 24], 4: [20, 24]}
    """
    return {q: [d for d in divisors if d % q == 0] for q in quotients}
```

Use Ok to test your code:

```
python3 ok -q divide
```

Q3: Buying Fruit

Implement the `buy` function that takes three parameters:

1. `fruits_to_buy` : A list of strings representing the fruits you need to buy. *At least one of each fruit must be bought.*
2. `prices` : A dictionary where the keys are fruit names (strings) and the values are positive integers representing the cost of each fruit.
3. `total_amount` : An integer representing the total money available for purchasing the fruits. Take a look at the docstring for more details on the input structure.

The function should print **all possible ways** to buy the required fruits so that the combined cost equals `total_amount` . You can only select fruits mentioned in `fruits_to_buy` list.

> **Note**: You can use the `display` function to format the output. Call `display(fruit, count)` for each fruit and its corresponding quantity to generate a string showing the type and amount of fruit bought.

> **Hint**: How can you ensure that every combination includes at least one of each fruit listed in `fruits_to_buy` ?

```
def buy(fruits_to_buy, prices, total_amount):
    """Print ways to buy some of each fruit so that the sum of prices is amount.

    >>> prices = {'oranges': 4, 'apples': 3, 'bananas': 2, 'kiwis': 9}
    >>> buy(['apples', 'oranges', 'bananas'], prices, 12)  # We can only buy apple, orange
    [2 apples][1 orange][1 banana]
    >>> buy(['apples', 'oranges', 'bananas'], prices, 16)
    [2 apples][1 orange][3 bananas]
    [2 apples][2 oranges][1 banana]
    >>> buy(['apples', 'kiwis'], prices, 36)
    [3 apples][3 kiwis]
    [6 apples][2 kiwis]
    [9 apples][1 kiwi]
    """
    def add(fruits, amount, cart):
        if fruits == [] and amount == 0:
            print(cart)
        elif fruits and amount > 0:
            fruit = fruits[0]
            price = prices[fruit]
            for k in range(1, amount // price + 1):
                # Hint: The display function will help you add fruit to the cart.
                add(fruits[1:], amount - price * k, cart + display(fruit, k))
    add(fruits_to_buy, total_amount, '')

def display(fruit, count):
    """Display a count of a fruit in square brackets.

    >>> display('apples', 3)
    '[3 apples]'
    >>> display('apples', 1)
    '[1 apple]'
    >>> print(display('apples', 3) + display('kiwis', 3))
    [3 apples][3 kiwis]
    """
    assert count >= 1 and fruit[-1] == 's'
    if count == 1:
        fruit = fruit[:-1]  # get rid of the plural s
    return '[' + str(count) + ' ' + fruit + ']'
```

Use Ok to test your code:

```
python3 ok -q buy
```

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.

Cities

Say we have a data abstraction 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, when writing a program about cities, we do not need to know the implementation.

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
    """
    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
```

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 `get_name` `get_lat` `get_lon`, constructors `make_city`, 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
```

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*.

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 Assignment

If you are in a regular section of CS 61A, fill out this lab attendance and feedback form (https://forms.gle/dHxj8gttNWRY6Ptm9). (If you are in the mega section, you don't need to fill out the form.)

Then, submit this assignment by uploading any files you've edited **to the appropriate Gradescope assignment.** Lab 00 (../lab00/#submit-with-gradescope) has detailed instructions.