Lab 05: Higher-Order Functions

Total Points: 10 (Correctness) + 1 extra credit (optional)
Due: Tuesday, February 13th, 11:59 pm (SD time)

Starter Files

Download lab05.zip. Inside the archive, you will find starter files for the questions in this lab. You can't import anything to solve the problems.

Extra Credit Opportunity

You have the opportunity to receive a **1 point** extra credit on that lab if you submit your last attempt early (refer to this section of each lab for the early submission deadline). Note: Each lab is graded out of 10 points, and you could possibly have more than 10 points in one lab.

Early Submission Date (lab05): Friday, February 9th, 11:59pm (SD time)

Testing

After finishing each question, run the file with the following command-line options to avoid compile-time errors and test the correctness of your implementation:

- No options: `>>> python3 lab05.py`
- Interactive mode: `>>> python3 -i lab05.py`
- Doctest **(Recommended)**: `>>> python3 -m doctest lab05.py`

For Windows users, please use `py` or `python` instead of `python3`.

Important Requirements and Notes

1. **DO NOT IMPORT ANY PACKAGES.**
2. It may be useful to add doctests when the given doctests do not cover all cases to avoid losing points from the Autograder/any hidden tests, but you are not required to add doctests in the labs.
3. Style/asserts are not required or graded on any labs, but it's recommended that you also follow the style guide for clean code. Method descriptions are provided in this lab.

4. You may assume all inputs are valid.

Introduction

A Higher-Order Function (HOF) is a function that manipulates other functions by taking in functions as arguments, returning a function, or both. We will be exploring many applications of higher-order functions. Here is an example of how higher-order functions may be used in Python.

```
def higher_order(f, x):
    return f(x)**2
```

Important: The pre-defined functions below will be used throughout this write-up as helper methods to the doctests.

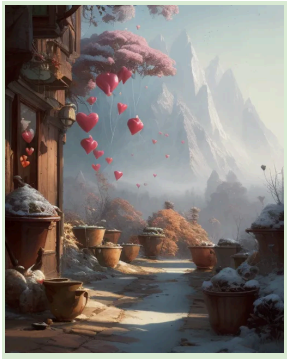
```
def identity(x):
    return x

def squared(x):
    return x**2

def cubed(x):
    return x**3

def root(x):
    return round(x ** 0.5, 2)
```

Required Questions



Citizens of Pythonia not only work hard, but they also enjoy celebrating holidays. The cities are all decorated for St. Valentine's, and many stores give out chocolates and candies to spread joy.

You participate in a friendly competition to determine who can celebrate St. Valentine's most successfully.

Question 1:

In order to do so, you decided to compare how much candy each of you got. However, simply adding up and comparing is boring, one of your friends has the idea to apply a function to the quantities of candy that you all collected.

Write a function `update_candy` that takes in two parameters:

- **candy_dict** (dictionary) - where the keys are the friend names (as strings) and the values are lists of positive integers that represent how much candy each person collected.
- **func** (function) - function passed in as a parameter.

Your function should apply a given function `func` to each integer in the list and return an updated dictionary.

Requirements: One line dictionary comprehension

Reminder: Dictionary comprehension is very similar to list comprehensions. This link might be helpful.

```
{key: value for (key, value) in iterable}
```

Example:

```
>>> example  = {x:x+10 for x in [1,2,3,4,5]}
>>> print(example)
{1: 11, 2: 12, 3: 13, 4: 14, 5: 15}
```

Assumption:

- All inputted functions will take a single parameter as an input and always be correct.
- All inputted dictionaries will have the correct format of string: list, and at least 1 item in the dictionary
- The list can be empty: see 3rd doctest

```
def update_candy(candy_dict, func):
    """
    >>> candy_dict = {"pat": [10, 17, 24], "andy": [24, 13, 21]}
    >>> update_candy(candy_dict, squared)
    {'pat': [100, 289, 576], 'andy': [576, 169, 441]}

    >>> candy_dict = {"ryan": [5, 10, 15], "derrick": [6], "deandre":
[9, 1]}
    >>> update_candy(candy_dict, identity)
    {'ryan': [5, 10, 15], 'derrick': [6], 'deandre': [9, 1]}

    >>> candy_dict = {"pj": [], "amari": [8, 13, 1], "nick": [6, 8,
6]}
    >>> update_candy(candy_dict, lambda x: x + 1)
    {'pj': [], 'amari': [9, 14, 2], 'nick': [7, 9, 7]}
    """
```

Question 2:

After applying the function on everyone's quantities you realized that you won! But your friends disagreed and suggested another idea: sum up all the new values and apply another function!

Write a function `candy_sum` that takes in three parameters:

- **candy_dict** (dictionary) - where the keys are the friend names (as strings) and the values are lists of positive integers that represent how much candy each person collected.
- **func1, func2** (functions) - functions passed in as parameters.

Your function should:

- apply `func1` to each integer in the list,
- calculate the sum of the new list values,
- and then apply `func2` to the sum.

Your function should then return the updated dictionary.

Requirements: **One line dictionary comprehension**

Assumption:

- All input functions will take a single parameter as an input and always be correct.
- All input dictionaries will have the correct format of string: list, and at least 1 item in the dictionary.
- The lists will always contain at least one item.

```
def candy_sum(dict, func1, func2):
    """
    >>> candy_dict = {"dan": [10, 17, 24], "jay": [100, 21, 5]}
    >>> candy_sum(candy_dict, squared, root)
    {'dan': 31.06, 'jay': 102.3}

    >>> candy_dict = {"sam": [16, 25], "terry": [100, 49]}
    >>> candy_sum(candy_dict, root, identity)
    {'sam': 9.0, 'terry': 17.0}

    >>> candy_dict = {"jalen": [4, 7, 1], "jason": [2, 8, 5, 9, 10]}
    >>> candy_sum(candy_dict, identity, squared)
    {'jalen': 144, 'jason': 1156}
    """
```

Question 3:

Your friends were comparing candies and realized that some candies appeared a lot but some are rare. So all of you agreed that certain types of candies should be treated differently than others. It is time to come up with a different way to apply these functions!

Write a function `candy_counts` that takes in *six* parameters:

- **candy_list** (list) - List of dictionaries, where each dictionary represents a different store. Each store dictionary has keys which are candy types (lowercase strings) and the values are integers that represent the quantity of each candy.
- **rare_candy** (lowercase string) - string of a type of rare candy.
- **common_candy** (lowercase string) - string of a type of candy.

func1, func2, func3 (functions) - functions passed in as parameters.

Your function should return a new dictionary where:

- `func1` is applied to the quantities for `rare_candy`,
- `func2` is applied to the quantities for `common_candy`,
- `func3` is applied to all the quantities that don't represent `rare_candy` or `common_candy`.

Input	`candy_list = [{'kitkat': 4, 'hershey': 3, 'm&m': 2}, {'hershey': 1, 'm&m': 4}, {'kitkat': 2, 'butterfinger': 4}, {'m&m': 3, 'hershey': 1}]`
	
	`rare_candy = 'm&m'`
	`common_candy = 'hershey'`
	`func1 = squared`
	`func2 = root`
	`func3 = cubed`

Output	`[{'kitkat': 64, 'hershey': 1.73, 'm&m': 4}, {'hershey': 1.0, 'm&m': 16}, {'kitkat': 8, 'butterfinger': 64}, {'m&m': 9, 'hershey': 1.0}]`
Explanation	• `kitkat` is neither a `rare_candy` or `common_candy` so `func3` is applied to 4 ($4^3 = 64$). • `hershey` is a `common_candy` so `func2` is applied to 3 ($3^{0.5} = 1.73$). • `m&m` is a `rare_candy` so `func1` is applied to 2 ($2^2 = 4$). • This process is repeated so that every key is checked and the appropriate function is applied to its value.

Notes:

- You can assume that `rare_candy` and `common_candy` aren't the same.
- There is no requirement for the solution, but try to solve it with one line dictionary comprehension.
- All functions will take a single parameter as an input.
- The root function will automatically round correctly for you.

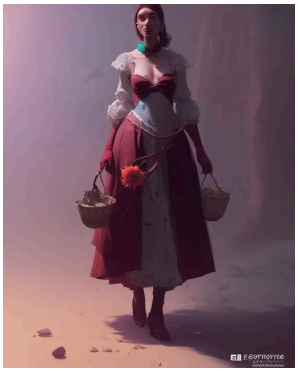
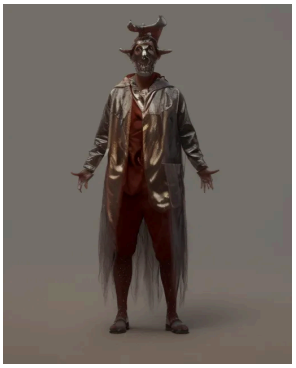
```
def candy_counts(candy_list, rare_candy, common_candy, func1, func2,
func3):
    """
    >>> candy_list = [{'kitkat': 4, 'hershey': 3, 'm&m': 2}, \
{'hershey': 1, 'm&m': 4}, {'kitkat': 2, 'butterfinger': 4}, \
{'m&m': 3, 'hershey': 1}]

    >>> rare = 'm&m'
    >>> common = 'hershey'
    >>> candy_counts(candy_list, rare, common, squared, root, cubed)
    [{'kitkat': 64, 'hershey': 1.73, 'm&m': 4}, {'hershey': 1.0,
'm&m': 16}, \
{'kitkat': 8, 'butterfinger': 64}, {'m&m': 9, 'hershey': 1.0}]

    >>> rare = 'butterfinger'
    >>> common = 'kitkat'
    >>> candy_counts(candy_list, rare, common, identity, squared,
cubed)
    [{'kitkat': 16, 'hershey': 27, 'm&m': 8}, {'hershey': 1, 'm&m':
64}, \
{'kitkat': 4, 'butterfinger': 4}, {'m&m': 27, 'hershey': 1}]

    >>> rare = 'hershey'
    >>> common = 'm&m'
    >>> candy_counts(candy_list, rare, common, squared, root,
identity)
    [{'kitkat': 4, 'hershey': 9, 'm&m': 1.41}, {'hershey': 1, 'm&m':
2.0}, \
{'kitkat': 2, 'butterfinger': 4}, {'m&m': 1.73, 'hershey': 1}]
    """
```

Question 4:

It seems that none of your formula ideas are working, so it's time to seek assistance from someone else to determine a winner. You approached the Pythonia's president, Guido van Rossum and he said: "In order to determine a winner, I invite you to my St. Valentine's party. I will take your costume into account, as well as your ability to express yourself."

Write a function `costume_rating` that takes in two parameters:

- **costume_color** (a string) - the color of your costume.
- **bling_value** (a positive integer) - represents how shiny your costume is.

First, you need to convert the `costume_color` to a `value` based on the table below

costume_color	**numerical value**
blue	7
red	3
green	9
yellow	0
purple	5
orange	2

This function should return an *inner function* that takes one parameter:

- **phrase** (a string) - the phrase said by a trick-or-treater.

and returns the score (as an int, by flooring the result) based on the following formula:

The final formula for the number of candies to return is the following

$$floor(\frac{(costume\ color\ value \times bling\ value)^{(length\ of\ phrase)/10}}{10})$$

```
def costume_rating(costume_color, bling_value):
    """
    >>> costume_rater = costume_rating('blue', 5)
    >>> costume_rater('19 Character Phrase')
    85
    >>> costume_rater('e')
    0
    >>> costume_rater('seven c')
    1

    >>> costume_rater = costume_rating('orange', 8)
    >>> costume_rater('hello i want candy')
    14

    >>> costume_rater = costume_rating('yellow', 5)
    >>> costume_rater('please give me candy i really need candy')
    0
    """
```

Question 5:

The competition above eliminated a good amount of people but the absolute leader is not determined yet. Guido van Rossum proclaimed: “You have a lot of good candy, you made an excellent costume and you are a talented candy collector, it is time to test your coding skills! You are in Pythonia after all”.

Your task is to create a function `encoder_generator` that generates an encoder for some text data. This function should take in three initial parameters and return an inner encoder function that implements encoding based on these parameters.

Function `encoder_generator` takes three parameters:

- **key** (a dictionary of character pairs): This parameter represents a key, which is used for a character substitution during encoding.
    - When substituting, do not replace the same character twice - see doctest 2 with 'black' for an example

- **start_idx** (a nonnegative integer): This parameter represents the position to start encoding at. All characters before this index should be ignored.
- **reverse** (a boolean): If set to `True`, reverse the encoded text. Perform the encoding normally otherwise.

This function returns an inner function which takes a single argument of text input to encode. **This input should be converted to lowercase** and then encoded according to the provided parameters. The encoding process should involve substituting characters using the key and starting at the specified position. If the reverse parameter is True, the encoded text should be reversed.

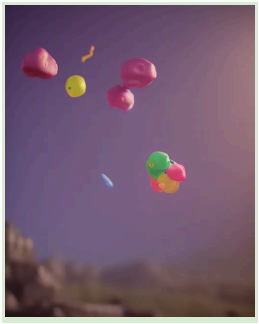
Input	`key = {'e': 'x', 'b': 'y'}`
	`encoder = encoder_generator(key, 1, False)`
	`encoder('hello')`

Output	`'xllo'`
Explanation	Starting from index 1, the string to encode is ‘ello’. From the key dictionary, we know to replace ‘e’ with ‘x’. Since the reverse parameter is false, we return ‘xllo’.

```
def encoder_generator(key, start_idx, reverse):
    """
    >>> key = {'e': 'x',\
               'b': 'y'}
    >>> encoder = encoder_generator(key, 1, False)
    >>> encoder('Hello')
    'xllo'
    >>> encoder('arbys')
    'ryys'

    >>> key = {'a': 'b',\
               'b': 'c'}
    >>> encoder = encoder_generator(key, 0, True)
    >>> encoder('black')
    'kcblc'
    """
```

Question 6:



Your competition is finally over and the winner is determined! It is the person who sits at A13 during the DSC20 lecture. Out of excitement you start throwing candy at the winner and the goal is to throw your candy as close as possible to your friend. In order to do that, you need to calculate the final landing position of the candy, given some values.

Write a function `candy_toss` that takes in three parameters:

- **gravity** (positive number): It represents the value of gravity on your planet.
- **time** (positive number): It represents the time in the air.
- **initial_pos** (tuple with any two numbers)**:** It represents the initial position of the candy in `x, y` format.

This function returns two inner functions such that:

- The first inner function takes one parameter:
  - **x_velocity** (any number)**:** the speed of the candy in the x direction

  The function should return the final X position of the candy by:
  - multiplying `x_velocity` by the `time`.
  - then adding the x `initial_pos` to get the final X position.
  - Gravity is not needed to solve this part.

- The second function takes one parameter:
  - **y_velocity** (any number)**:** the speed of the candy in the y direction

  The function should return the final Y position of the candy using the following formula:

  $$y\_final = y\_initial + (y\_velocity * time) - (½ * gravity * time\text{^}2)$$

Please round your final answer to 2 decimal points.

```
def candy_toss(gravity, time, initial_pos):
    """
    >>> x_pos_calc, y_pos_calc = candy_toss(9.8, 5, (0,0))
    >>> x_pos_calc(10)
```

```
    50
    >>> y_pos_calc(50)
    127.5
    >>> x_pos_calc, y_pos_calc = candy_toss(1, 7.5, (15,15))
    >>> x_pos_calc(10)
    90.0
    >>> y_pos_calc(0.1)
    -12.38
    """
```

.

Submission

Please submit the homework via Gradescope. You may submit more than once before the deadline, and only the final submission will be graded. Please refer to the Gradescope FAQ if you have any issues with your submission.