CS10 Su23 Discussion 7: Python Data Structures SOLUTIONS

I. Data Structures

Computer programming involves a lot of data processing, utilization and manipulation, where data information displays a representation of facts, concepts, or instructions in a formalized manner that's suitable for communication, interpretation and processing. With the significance of data processing in computer programming, it is important that this data is stored efficiently and is accessible.

Data structures play a big role in accomplishing this. **Data structures** are ways of organizing, processing, retrieving, and storing data. Data structures make accessing and modifying data much easier and more efficient, with multiple variations of data structures out there existing that are designed to arrange data to suit a specific purpose.

Data structures can be classified into two categories: implicitly built-in data structures, and user-defined data structures. Python allows programmers to create their own **user-defined data structures**, where functionality of the data structure is under full control of the user. User-defined data structures are out of scope for this course specifically, but a huge part of your programming journey will involve engaging with a data structures and algorithms courses that will teach you how to create your own data structures. *(The UC Berkeley version of this course is CS61B/L, the second course of the notorious CS61 series!)*

The **built-in data structures** that are supported in Python include lists, dictionaries, tuples, and sets. The implementation of lists in Python are relatively similar to how they're portrayed on *Snap!*. However, dictionaries, tuples, and sets are all new. **Dictionaries** ({ }) are like lists, but hold collections of key-value pairs. Whereas items in lists have an associated index, items in dictionaries have an associated key that you use to look up an item. **Tuples** have a similar structure to lists, but rather thank square brackets ([ ]), tuples are created using parentheses. Tuples are immutable, so once you put an item in a tuple, you cannot change it unless the item you stored in the tuple is mutable (i.e. a tuple of lists). **Sets** also use curly braces ({ }) like dictionaries, but sets represent a collection of unordered elements that are unique, meaning no item in a set repeats itself (i.e {1, 2, 3} would be an example of a set).

Each of these data structure have different functionalities, so depending on what you want your program to accomplish, think about which of these data structures would you benefit from using the most. In this course, we will putting most of our attention around practicing with lists and dictionaries, but do keep tuples and sets in mind in case you run across a situation where you find their usage beneficial.

II. List Comprehension

When playing around with lists, we witnessed that there were two ways in which we could iterate through lists and make changes to it:

1. *Looping through all items in the list using a for i or for each loop*
2. *Abstracting the looping process by using a higher-order function like map, keep, or combine in Snap!*

We can mimic the functionalities of the HOFs blocks on *Snap!* by using list comprehension. **List comprehension** is a convenient way to make a new list based off of values from an old list, which is essentially the functionalities of the map and keep blocks on *Snap!*.

The syntax of list comprehension is rather compressed, but it can be summed down to this structure:

new_list = [<*expression*> for <*item*> in <*old_list*> if <*condition*>]

Oftentimes, you will find the variable holding *item* appear within the *expression*, which indicates that the expression is dependent on the items in the list.

Q1: More & More Translation

Translate the following blocks of code from *Snap!* into lines of Python code using list comprehension:

a)


```
[word[0] for word in my_list if len(word) > 5]
```

b)


```
[sum([num for num in range(11) if num % 2 == 0])]
```

c) Write a list comprehension that finds the index of an item in a list. You may assume that the item appears only once in the list.

```
def find_index(item, lst):
    return [i for i in range(len(lst)) if item == lst[i]][0]
```

III. Dictionaries

Like mentioned previously, **dictionaries** hold *key-valued pairs*, with its structure being as follows:

```
fruits_dict = {'A' : 'apple', 'B' : 'banana', 'C' : 'cherries'}
```

The **key** is held before the colon and the **value** of the item is placed after the colon. To access a value, you would use its associated key rather than its index in the dictionary. *All keys in a dictionary must be unique, but multiple keys can have the same value.*

Although recent updates to Python (3.6+) have made dictionaries *insertion-ordered*, do note that when speaking of dictionaries conceptually, *you should not assume order.*

Fill out the table below to get more familiar with the syntax for dictionaries in Python.

```
class_dict = {'Math':'1A', 'English':'R1A'}
```

Add the key 'CS' with the value '10'	`class_dict['CS'] = '10'`
Access the value of 'Math'	`class_dict['Math']`
Change the value of 'Math' to '1B'	`class_dict['Math'] = '1B'`
Check if 'UGBA' is a key in class_dict	`'UGBA' in class_dict`
Check if '10' is a value in class_dict	`'10' in class_dict.values()`
Get a list of the keys in class_dict	`list(class_dict)`

Q2 : Merge_Dicts

Write a function `merge_dicts` that takes in two dictionaries as inputs and returns a new dictionary that contains all entries from both input dictionaries. You can assume that both dictionaries have Strings as keys and numbers as values. For any keys present in both dictionaries, the corresponding value in the output dictionary should be the sum of the values in the inputs.

```
""" Below is an example of how merge_dicts works:
>>> d1 = {'Dan': 10, 'Oski': 15}
>>> d2 = {'Alonzo': 5, 'Oski': 20, 'Dan': -10}
>>> merge_dicts(d1, d2)
{'Dan': 0, 'Alonzo': 5, 'Oski': 35} """

def merge_dicts(dict1, dict2):
    new_dict = {}
    for key in dict1:
        new_dict[key] = dict1[key]
    for key in dict2:
        if key in new_dict:
            new_dict[key] += dict2[key]
        else:
            new_dict[key] = d2[key]
    return new_dict
```

Q3 : LC in Dicts

a) Write a function `keys_with_value()` that takes in a `dictionary` and a value `v`, and returns a list of keys from `dictionary` with value equal to `v`.

```
""" Below is an example of how keys_with_value works:
>>> fruits = {'apple': 10, 'strawberry': 10, 'banana': 5}
>>> keys_with_value(fruits, 10)
['apple', 'strawberry'] """

def keys_With_value(dictionary, value):
    return [k for k in dictionary if dictionary[k] == value]
```

b) For this question, we're going to use tuples! To create a tuple, insert values in between parentheses, separated by commas. For instance, `('a', 'b')` is a tuple with its first item being `'a'` and its second item being `'b'`.

Write a function `conditional_map()` that takes in a dictionary `dict`, a function `func`, and a predicate condition `cond`, and returns a list of key-valued pairs from `dict` stored in tuples that contains keys that satisfy `cond`, with `func` applied to each value.

```
""" Below is an example of how conditional_map works:
>>> fruits = {'apple': 1, 'cherries': 3, 'strawberries': 5}
>>> def is_there_an_i(word):
...     return ("i" in word)
...
>>> def double(value):
...     return value + value
...
>>> conditional_map(fruits, double, is_there_an_i)
[('cherries', 6), ('strawberries', 10)] """

def conditional_map(dict, func, cond):
    return [(key, func(dict[key])) for key in dict if pred(key)]
```

Q4 : What Would Python Do?

Assume we have defined `food_dict` in the Python interpreter as follows:

```
food_dict = {"fruit": "apple", "veggie": "carrot", "beverage":
                    "water", "grain": "rice"}
```

Write down what will be displayed after each of the following lines execute. If the result is an error message, simply write "Error" as your answer. Each subproblem is independent and does not depend on other subproblems.

```
>>> len(food_dict)
```

4

```
>>>list(food_dict)
```

['fruit', 'veggie', 'beverage', 'grain'] **NOTE: we cannot rely on order

```
>>> food_dict[0]
```

Error **0 is not a key

```
>>> ('fruit' in food_dict) and ('apple' in food_dict)
```

False **'apple' is not a key in food_dict

```
>>> ("fruit" in food_dict.keys()) and ("apple" in food_dict.values())
```

True

```
>>> for food in food_dict:
. . .  food += "s"
. . .
>>> food_dict
```

{'fruit': 'apple', 'veggie': 'carrot', 'beverage': 'water', 'grain': 'rice'}

```
>>> def recursion_is_fun(dict1, dict2):
. . .  if dict2 == {}:
. . .          return dict1
. . .  dict2.pop(list(dict2)[0])
. . .  return recursion_is_fun(dict1, dict2)
. . .
>>> copy = food_dict
>>> recursion_is_fun(food_dict, copy)
```

{}

```
>>> more_food = {"protein": "chicken"}
>>> food_dict["more food"] = more_food
>>> food_dict
```

{"more food": {"protein": "chicken"}}

Q5 : Extra Practice

For this extra problem, write a function `unique_vals` that takes a dictionary `d` and returns `True` if every value in `d` only has one corresponding key.

```
""" Below is an example of how unique_vals works:
>>> unique = {'a': 4, 'b': 5, 'c': 3}
>>> one_to_one(unique)
    True

>>> not_uniqe = {'a': 2, 'b': 4, 'c': 2}
>>> one_to_one(not_unique)
    False """

def unique_vals(d):
      seen_values = []
      for value in d.values():
           if value in seen_values:
                return False
      seen_values.append(value)
      return True
```