Quiz #1: Review of Iteration and Python Data Types ICS-33 Spring 2014

Name _____

Lab # (1-8) _____

Recall the Academic Integrity statement that you signed. Write all answers clearly on these pages, ensuring your final answers are easily recognizable. The number of points for each problem is clearly marked, for a total of 25 points. I will post my solutions on the web on Wednesday, off the **Solutions** link, after class.

Download the **q1helper** project folder (available for Friday on the weekly schedule) in which to write/test/debug your code. Submit your completed **q1solution.py** module online and write your answers on this quiz (hand in only one two-sided sheet: not multiple pages attached/stapled). Both are due at the start of class on Wednesday.

1. (5 pts) Define the following two functions: each returns a function. (a) The **one_of** function takes a list as an argument; it returns a function that takes a value as an argument and returns **True** if that value is in the list. For example: **vowel = one_of(['a','e','i','o','u'])** then **vowel('i')** returns **True**. (b) The **opposite_of** function takes a predicate (a function taking one argument and returning a **bool**) as an argument; it returns a function that takes a value as an argument and returns the opposite (negation) of the result the function argument return on that value. For example: **composite = opposite_of(is_prime)** then **composite(6)** returns **True** (**is_prime(6)** return **False**). The **predicate** module defines **is_prime**.

```
def one_of(l:[]):
    def check(x):
        return x in l
    return check
    #or, just use the line below
    #return lambda x : return x in l
```

```
def opposite_of(f):
    def negated(x):
        return not f(x)
    return negated
    #or , just use the line below
    #return lambda x : not f(x)
```

2. (5 pts) Use the **sorted** function, which takes any iterable as an argument and returns a list, to define the following two one-line functions. In each case write the **key** function as a **lambda** to specify how to sort the data. (a) The **sort_names** function takes a list of 2-tuples as its argument, like **('Joe','Smith')**; it returns a list of these same 2-tuples sorted by last name and then first name. (b) The **sort_age** function takes a list of 2-tuples as an argument, each like **('Joe',(9,23,1992))** where the second index is a 3-tuple specifying the person's month, day, and year of birth; it returns a list of 2-tuples sorted by age: from **youngest to oldest**.

```
def sort_names(data : [(str,str)]) -> [(str,str)]:
    return sorted(data,key=(lambda x: (x[1],x[0])))


def sort_ages(data : [(str,(int,int,int))]) -> [(str,(int,int,int))]:
    return sorted(data,key=(lambda x: (x[1][2],x[1][0],x[1][1])),reverse=True)
```

3. (5 pts) Use comprehensions to define the following two one-line functions. Both take as arguments dictionaries whose keys are names (**str**) and whose associated values are their number of siblings (**int**). (a) The **big_family** function returns a list of those names with 3 or more siblings. (b) The **only_child** function returns a dictionary with the same keys, but whose associated values are **bool**: **True** iff that person has no siblings.

```
def big_family(d:{str:int}) -> []:
    return [p for p,f in d.items() if f >=3]


def only_child(d:{str:int}) -> {str:bool}:
    return {k:v==0 for k,v in d.items()}
```

4. (5 pts) Define the **follows** function, which takes one argument: a **str** that is one word; it returns a dictionary whose keys are every letter in the word that is followed by another letter, and whose values are a set of the letters that directly follow that letter in the word.. For example **follows1('bookeeper')** returns **{'e':{'e','p','r'}, 'o':{'o','k'}, 'p':{'e'}, 'b':{'o'}, 'k':{'e'}}**; of course, the **dict/set** values may appear in any order when printed. Hint: loop through all the indexes in the string (except the last), updating a local dictionary for each letter in the string, so that its associated set also contains the letter that directly follows it. Write **follows1** using/returning a **dict** and a slightly simpler **follows2** using/returning a **defaultdict**.

```
def follows1(s : str) -> {str:{str}}:
    answer = {}
    for i in range(len(s)-1):
        key = s[i]
        if key not in answer:
            answer[key] = set()
        answer[key].add(s[i+1])
    return answer


def follows2(s : str) -> {str:{str}}:
    answer = defaultdict(set)
    for i in range(len(s)-1):
        answer[s[i]].add(s[i+1])
    return answer
```

5. (5 pts) Define the **reverse** function, which takes one argument: a dictionary in which each key is associated with a set: for a concrete example, the keys might be names (**str**) and the sets might contain skills that each key has (also **str**). In this case, **reverse** returns a dictionary whose keys are skills and whose associated set is the names of the people with those skills. You can use **dict** or **defaultdict** (**defaultdict** results in simpler code).

```
def reverse (d:{str:{str}}) -> {str:{str}}:
    dr = defaultdict(set)
    for k,vs in d.items():
        for v in vs:
            dr[v].add(k)
    return dr
```