ICS-33: In-Lab Programming Exam #3

Name (printed: Last, First): _____ Lab # (1-12): _____

Name (signed: First Last): _____ Seat # (1-46): _____

PRINT AND SIGN YOUR NAME ABOVE NOW; FILL IN THE ROOM # AND YOUR MACHINE #

1: Details of prev_n:

The **prev_n** generator function takes three arguments: one **iterable**, one integer (**n**), and one predicate function (**pred**). It returns an **iterable** result that produces **list**s of up to **n** values: the **n** values up to and including the value for which calling **pred** returns **True**. If **pred** is **True** when fewer than **n-1** values precede it, then all the values preceding it appear in the **list** produced.

Executing

```
for i in prev_n( 'a.bcde.f.g.hijk', 3, lambda x : x == '.'):
   print(i)
```

prints

```
['a', '.']
['d', 'e', '.']
['.', 'f', '.']
['.', 'g', '.']
```

Notice in this example that the predicate is **True** for the **'.'** character, and that each produced **list** contains the **3** characters up to and including the **'.'** character; but, the first **list** is smaller because only 1 character precedes the **'.'** character. There are 4 **'.'** characters in **iterable str**, so the generator produces 4 **list**s.

Calling the **list** constructor function on any **iterable** produces a **list** with all the values in that **iterable**. So, calling

```
list( prev_n('a.bcde.f.g.hijk', 3, lambda x : x == '.') )
```

produces the list

```
[['a', '.'], ['d', 'e', '.'], ['.', 'f', '.'], ['.', 'g', '.']]
```

Do not assume anything about the **iterable** argument, other than that it is **iterable**; the testing code uses the **hide** function to "disguise" a simple **iterable** (like a **str**); don't even assume that the **iterable** is finite: so, don't try iterating all the way through it to put its values in a **list** or any other data structure.

Finally, **You may not import any functions from functools** to help you solve this problem.

2: Details of deepest:

The **deepest** function takes one argument that is either an **int** value or a **list** (with possible nested **lists** inside it,

1 of 3

which ultimately store **int** values). It returns an **int** that indicates how deeply nested its **list** argument is: the deepest nesting of some **int** in the **list**. Every **list** (including all the nested **lists**) will store at least one **int** value (so there are **no empty lists**).

Note: you can use iteration when solving parts of this this problem, but to solve it fully, you will also need to use recursion.

Calling **deepest([3, [5], 8])** returns **2** because **5** is nested **2** deep in this **list** (**5** is in a **list** in a **list**); whereas **3** and **8** are only nested **1** deep in their **list**.

Calling **deepest([[1,2,[9,[7],8],[1,3,2]], [1,1]])** returns **4** because the **int** that is the deepest (here **7**) is nested inside **4 lists**.

Think carefully about how to write this recursive function. Given the specification of the function, think carefully about the base case and how to combine successfully solved recursive calls. Review the specification for the parameter. "Its elephants all the way down."

3: Details of check_dict:

Define a class named **check_dict**, derived from the **dict** class: it will check every **key/value** association for legality as it is stored into the dictionary. Use it as follows: the first **lambda** specifies what **keys** are legal (here they must be **str**s) and the second **lambda** specifies what **values** are legal (here they must be non-negative **int**s).

```
d = check_dict(lambda x : type(x) is str,\
               lambda x : type(x) is int and x >= 0)
```

Define **check_dict** by inheritance, so that it operates as described below, producing the same results as the examples below. By using inheritance, other methods like **__len__** or **__contains__** should work correctly without you having to write any code for them. Some calls to these (and other methods) may appear in the testing code for these functions.

Specifically,

1. When a **check_dict** is constructed, it is passed two arguments that are predicates: each is function of one argument that returns a **bool** value. The first predicate checks whether a **key** is legal; the second checks whether a **value** is legal.

   Store only legal **key**/**value** pairs directly in a **check_dict**. **Illegal key**/**value** pairs will not be stored directly in the **check_dict**, but will instead be stored in an auxiliary dictionary (described below).

2. Whenever a **value** is associated with **key**, both are checked for legality by their respective predicates. If both are legal, the association is stored in the **check_dict**; if either or both are illegal, the **value** is appended at the end of a **list** containing the history of assocations for that **key** in the auxiliary dictionary. After executing

   ```
   d['a'] = 1       # legal key and value
   d[2] = 2         # illegal key
   d['z'] = -1      # illegal value
   d[2] = 2.5       # illegal key and value
   ```

   d stores the contents **{'a': 1}** and the auxiliary dictionary stores the contents **{2: [2, 2.5], 'z': [-1]}**. Accessing **d['a']** evaluates to **1**, but accessing any other **key** would raise a **KeyError** exception.

3. Allow **calling check_dict** objects with a single argument representing a **key**: if the **key** is stored in the **check_dict**, return its associated value; otherwise, if the **key** is stored in the auxiliary dictionary, return the **last value** in its associated **list** there. For the example above, calling **d('a')** returns **1**; **d(2)** returns **2.5** (**2**'s

2 of 3

last/most recent association); and **d('z')** returns **-1** (**'z'**'s last/most recent association).

4. Finally, write a generator function named **iter_errors** which produces all the **key**/**list** of value **2-tuples** stored only in the auxiliary dictionary, in (a) decreasing order by how often **key**'s association was stored in the **list** and (b) for associations changed an equal number of times, produce **2-tuples** in increasing alphabetical order based on the string representation of the **key**. If the auxiliary dictionary stores the contents

```
{2: [2, 2.5], 'z': [-1], 'x': ['one']}
```

executing

```
for k,v in d.iter_errors():
    print('error :',repr(k),'->',repr(v))
```

would print

```
error : 2 -> [2, 2.5]
error : 'x' -> ['one']
error : 'z' -> [-1]
```

You can write a special **__str__** method, so you can print all the relevant information in a **check_dict**, to help you debug the methods needed in this class. For example, you might return a string containing both the contents of the **check_dict** and auxiliary dictionary. This method is totally optional: I will test no requirements related to it.

Extra Credit: 1 point

The actual **__init__** for a dictionary allows

1. a positional argument that is either a **dictionary** or an **iterable** that when iterated over produces key-value pairs.
2. any number of named-value arguments.

and uses these to populate the dictionary. For example, executing the code below for a standard **dict**

```
d = dict( [('a',1),('b',2)], c=3, d=4 )
print(d)
```

prints

```
{'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

The same is true after executing

```
d = dict( {'a':1,'b':2}, c=3, d=4 )
```

For one extra credit point, write **check_dict** so it can be initialized in the same way. After executing

```
d2 = check_dict(lambda x : type(x) is str,\
                lambda x : type(x) is int and x >= 0,\
                [('a',1),(2,2)], b=2, c=-3)
```

d stores the contents **{'a': 1, 'b': 2}** and the auxiliary dictionary stores the contents **{2: [2], 'c': [-3]}**. The same is true after executing

```
d2 = check_dict(lambda x : type(x) is str,\
                lambda x : type(x) is int and x >= 0,\
                {'a':1, 2:2}, b=2, c=-3)
```


3 of 3