ICS-33: In-Lab Programming Exam #3

Name (printed: Last, First): _____ Lab # (1-10): _____

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

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

1: Details of tail:

The **tail** generator function (a decorator for iterables) takes one or more arguments: all will be **iterable**. When called, it returns an **iterable** result that produces **some** of the values from the **iterable argument that produces the most values**: it produces values from it only **after all the other iterables have stop producing values**. There will always be exactly one iterable argument that produces more values than any of the others (so only one can be infinite). Using this defintion, executing

```
for i in tail('abc', 'abcdef', [1,2]):
   print(i)
```

prints

```
d
e
f
```

Notice in this example that

1. All arguments are **str** and therefore all are **iterable**.
2. The second argument produces more values than any of the others.
3. The second iterable is the only one that produces a 4th value (**d**) so that is the first value produced by **tail**, followed by all other values in the second iterable: **e** then **f**.

Of course, we cannot generally compute the length of an iterable, and one of the iterables might be infinite: recall only one can be infinite, because one iterable must produce more values than any of the others.

Hints: Store a list of **iterators** still producing values. You may have to do something special for the produced value **d** in the example above; so, you may have to **debug** code that first produces only **e** then **f**, omitting **d**.

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

```
list( tail('abc', 'abcdef', [1,2]) )
```

produces the list

```
['d', 'e', 'f']
```

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

Finally, **You may NOT import any functions from functools** to help you solve this problem. Solve it with with the standard Python tools that you know.

2: Details of min_cuts_odd:

The **min_cuts_odd** function takes one argument that is a **str** containing lower-case letters. It returns a **str** containing those letters and the **minimum** number of **'|'** (vertical stroke) characters, **cutting** the string into substrings, such that every character appearing in a substring appears an **odd** number of times (call this the **odd-count** criteria). For example, calling **min_cuts_odd('bacacababa')** returns the solution **'bac|acaba|ba'**. This solution cuts the argument string into 3 substrings (**'bac'**, **'acaba'**, and **'ba'**) that each satisfy the **odd-count** criteria; there is no solution with fewer than 2 **'|'** characters.

There is alway a trivial solution to the **odd-count** criteria that is typically non-minimal: for example, **'cbbbbcca'** has the trivial (non-minimal) solution **'c|b|b|b|b|c|c|a'**. There are often multiple solutions for the same input: for example, **'cbbbbcca'** has two minimal solutions, **'cb|bbbc|ca'** and **'cbbb|bc|ca'**. Your solution doesn't have to match the one I show, but it should always (a) satisfy the **odd-count** criteria and (b) have the same/minimum number of **'|'** characters.

Note: you can use all of Python's programming features when solving this problem, but to solve it fully, you will need to use recursion.

Think carefully about how to write this recursive function. Given the specification of the function (its argument and result), think carefully about how to write the base case and how to successfully combine the solutions of recursively solved subproblems. "Its elephants all the way down."

Hint 1: The base case is a string satisfying the **odd-count** criteria: it uses 0 (the minimum number of) **'|'** characters. Define the helper function **all_odd(s : str) -> bool**, which returns whether or not every character in **s** occurs an **odd** number of times, which is tested by itself and worth a small amount of credit; note that **s.count('a')** counts the number of times that character **'a'** appears in string **s**: for example, **'abcaba'.count('a')** returns **3**.

Hint 2: Try placing one **'|'** character between each pair of characters and recuring to discover if the solution with the **'|'** character in that position leads to the minimum number of **'|'** characters overall, while still satisfying the **odd-count** criteria.

3: Details of transformkey_dict:

Define a class named **transformkey_dict**, derived from the **dict** class. You use it like this

```
d = transformkey_dict(str.lower)
d['aLPHa'] = 1
d['BetA'] = 2
d['ALPHA'] += 10
print(d['alPHA'], d['BEta'])  # Note: using [] to get an item from d
```

```
print(d('aLpHa'), d('BeTa'))  # Note: using () to call d
```

Briefly, when a **tranformkey_dict** is initialized (calling **__init__**) it is passed a first argument that is a function of one parameter (whose type will be the key-type of the dictionary); it transforms any key into a "simpler" one. Above, the **str.lower** function is passed; it takes any **str** key and returns the equivalent **str** but in all lower-case letters: **str.lower('aLPHa')** returns **'alpha'**.

Whenever a key's value is gotten (calling **__getitem__**) or stored (calling **__setitem__**) the supplied key is first transformed into the simpler one and then that simpler one is actually used as the key in the dictionary. So writing **d['aLPHa'] = 1** has the same meaning as **d['alpha'] = 1**. And writing **d['ALPHA'] += 10** has the same meaning as **d['alpha'] += 10**. The result in the example above is the first **print** statement prints: **11 2**: the current associations with the simplified **'alpha'** and **'beta'** keys in the dictionary.

In addition a **transformkey_dict** also uses another dictionary to associate each simplified key with the actual key that it was **first** stored with. When we write **d('ALPHA')** (calling **__call__**) it simplifies the key to **'alpha'** and finds that the first time that this simplified key was stored its actual key was **'aLPHa'**. So, the second **print** statement prints: **aLPHa BetA**: the first keys associated with the simplified keys **'alpha'** and **'beta'** (from the other dictionary).

Define the **transform_dict** so that it operates as described above, producing the same results as the example above. **By using inheritance, other functions like len or iter should work correctly on it without you having to write any code.** Some calls to these (and other methods) may appear in the testing code for these functions.

I also wrote a special **__str__** method, so I could print all the information in a **transform_dict**, to help me debug the methods needed in this class. I converted both the dictionaries used by objects in this class to strings. There are no requirements that will be tested for a **__str__** ,method defined in this class; write it for your benefit only.

Extra Credit: 1 point

The actual **__init__** for a dictionary allows

1. a first positional argument that is an iterable producing key-value pairs when interated over
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}
```

For one extra credit point, write **transformkey_dict**s so it can be initialized in the same way:

```
d = transformkey_dict(str.lower,[('ALPHA', 1), ('BETA',2)], GAmma=3, delTA = 4)
```

Of course, if we write

```
d = transformkey_dict(str.lower)
```

that should still work as described above.