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 #

This in-lab programming exam is worth a total of 50 points (half the points of the first two In-Lab exams). It requires you to write a generator function (decorator for iterables), a recursive function, and a class derived using single inheritance. I will supply a short script for calling the functions and methods and printing their results, so that you can check them visually for correctness (and more easily debug them). The end of the script calls **driver.driver()** to test your code using batch self-checks (similar to the ones that I will use for grading purposes).

You will have approximately 105 minutes to work on the exam, after logging in and setting up your computer (so that is about 35 minutes/problem). You will then write, test, and debug the functions/methods: they will be in a project folder named **exam**. You may write-on/annotate these pages, as we will collect them from you at the end of the exam and not re-use them.

We will test your functions/methods only for correctness; each test you pass will improve your grade; functions/methods that produce no correct results will earn no credit. This means that your functions/methods must define exactly the parameters specified in the descriptions below and must work on all the example arguments; your functions/methods should also work on any other similar/correct arguments. To aid you writing and debugging these methods

1. Write clear, concise, and simple Python code (there are no statement restrictions).
2. Choose good names for local variables.

You do not need to include any comments in your code; but, feel free to add them to aid yourself. We will **not** grade on appropriate names, comments, or Python idioms: only on correctness. You may also call extra **print** methods in your code or use the Eclipse Debugger to help you understand/debug your code.

You may import and use any functions in the standard Python library and the **goody** and **prompt** modules (which will be included in the project file that you will download), unless told otherwise in the problem specification. Documentation for Python's standard library and these modules will be available during the exam. I have written all the standard import statements that I needed; feel free to include other imports, or change the form of the included imports to be simpler to use.

If you are having problems with the operating system (logging on, downloading the correct folder/files, accessing the Python documentation, submitting your solution) or Eclipse (starting it, setting it up for Python, running Python scripts, running the Eclipse debugger, turning on line numbers) please talk to the staff as soon as possible. We are trying to test only your programming ability, so we will help you with these other activities. But, we cannot help you understand, test, or debug your programming errors. I have asked the TAs and Tutors to NOT ANSWER ANY QUESTIONS about the exam itself: read it carefully and look at the test cases for clarification.

You should have a good understanding of the solution to Quizzes #4, #5, and #6. You should be familiar with **for** and **while** loops, the use of **try/except** statements, the **min** and **sorted** functions (and their use of the **key** parameter for determining order), lambdas, comprehensions, ***args** as a parameter specification, using **iter** and **next** on iterables, generator functions, recusive functions, single inheritance, and writing dunder methods. You are free to

1 of 6

use/avoid whatever Python language features you are comfortable/uncomfortable with. Grading depends only on whether your functions follow the requirements and work correctly.

Before submitting your program, ensure that it runs (has no syntax errors) and ensure there are no strange/unneeded **import** statements at the top of your file.

Summary:

This exam is worth 50 points. It requires you to write one generator function, one recursive function, and one class (using single inheritance) according to the specifications below. It will be graded as follows. If you solve no problems correctly, you will score at least 0 points; if you solve one problem completely correctly (no matter which one), you will score at least 36 points (72%, a **C-**); if you solve two problems completely correctly (no matter which ones), you will score at least 43 points (86%, a **B**); if you solve all three problems completely correctly, you will score 50 points (100% an **A**).

The module you will download...

- imports functions that are used for testing the code you will write.
- defines functions with reasonably annotated parameter names (you can change the names) and whose bodies are **pass**; thus, they will return **None** and will satisfy no tests.
- defines a class in which you must define the headers and bodies of its methods.
- includes a script that tests these functions/methods individually on different legal inputs, printing the results (sometimes showing extra information). You are also welcome to add and run your own tests inside this script.

  The script tests are followed by batch self-check tests.

The next three sections explain the details of these two functions and the class. You should briefly read all three and decide in what order to try solving them. If you are not making good progress toward a solution, move on to work on a different problems (which you might find easier) and return if you have time.

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.

2 of 6

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 NWSD:

The **NWSD** function takes two arguments: the first in an **int** (name it **n**) and the second is a **set** of positive **int**s (so, it cannot contain **0**; name it **s**). **NWDS** returns an **int** computing the **N**umber of **W**ays to **S**um to **n** all the values in the **set s** when allowing **D**uplicates. For example, calling **NWSD(5,{1,2,3,4,5})** returns **7**, because there are **7** different ways for the value's in **{1,2,3,4,5}** to sum to **5**. You do not have to compute **how** these values sum to **5**; just compute **how many** sum to **5**.

1. as **5**
2. as **4+1**
3. as **3+2**
4. as **3+1+1**
5. as **2+2+1**
6. as **2+1+1+1**
7. as **1+1+1+1+1**

The second arugment can be any arbirary set; so, another example is calling **NSWD(7,{2, 3, 5})** which returns **2**:

1. as **5+2**,
2. as **3+2+2**

You must write the **NWSD** function recursively. For small arguments, such an **NWSD** can compute its result in under a few seconds. In addition, you may use all of Python's other programming features (including rebinding and mutation).

Think carefully about how to write this recursive function. Given the specification of the function (including its arguments, result, and their types), think carefully about how to write the base case(s) and how to break down the problem into similar/strictly smaller subproblems and successfully combine the solutions of these recursively solved subproblems. "Its elephants all the way down."

3 of 6

Hint 0: It is useful to think concretely about some specific set **s**. I suggest something small but not too small (not a base case): think about its smaller subproblems and their solutions (which you should be able to compute by hand).

Hint 1: Given **n** and **s**, for what cases can you trivially know the solution without recurring? Recognize and return the correct value for those cases: some might be more general/complicated than you think; you'll undestand more as you investigate solving the problem recursively.

Hint 2: For any value in **s**, count the number of solutions (a) using that value and (b) not using the value. **Do not mutate any parameters**: if you need a mutation of a parameter, make a local copy of its information and then mutate the copy.

Do not import/use any other functions from any other modules.

Memoize cannot decorate NWSD: We cannot use **Memoize** to speed up **NWSD** because it's **set** parameter is mutable (not hashable) and therefore cannot be used as a key in **Memoize**'s **dict**/cache of arguments. If we change its second argument to be a **frozenset** and we could use **Memoize**: the changed function would be very similar, but a bit harder to write, so we will stick with using **set** parameter.

3: Details of history_dict:

Define a class named **history_dict**, derived from the **dict** class: it will store the current associations of its keys, a count of how many times an association for each key has been set, and a few of its most recent associations (not all of them). Use it as follows.

```
d = history_dict(2)
```

The argument specifies the maximum number of previous associations to store. In this case, **d** will store the current values associated with each key, along with up to **2** of the previous values stored for each key.

IMPORTANT: Define **history_dict** by inheritance, so that it operates as described below, producing the same results as in the examples below. By using inheritance, other methods like **__len__** or **__contains__** or iterations by **items** should be inherited and 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 this derived class.

Specifically,

1. When a **history_dict** is constructed, it is passed one argument (call the parameter **n**) that specifies how many previous associations to store/remember for each key. If this argument is not an **int** bigger than **0**, raise an **AssertionError** exception (you may omit any message). Besides using inheritence and storing **n**, store two **specially-named** attributes (dictionaries: **dict** or **defaultdict**) such that for each key in the **history_dict**:
    - **_count** stores a count of how often a new assocation is made for that key.
    - **_history** stores a list of the up to **n** previous associations for that key.

    IMPORTANT: The **see_count** and **see_history** methods are already written in the class; they must return the **_count** and **_history** attributes, so that a **bsc** file can check whether they store the correct information.

4 of 6

2. When a **key** is associated with a value in **__setitem__**, update the inherited **history_dict** and also update its two attribute dictionaries (described above) appropriately. For example, see the result below after executing each statement.

```
d = history_dict(2) # history attribute stores up to 2 old associations
d['a'] = 'a1'       # _count/_history attributes: {'a': 1}/{}
d['a'] = 'a2'       # _count/_history attributes: {'a': 2}/{'a': ['a1']}
d['a'] = 'a3'       # _count/_history attributes: {'a': 3}/{'a': ['a1', 'a2']}
d['a'] = 'a4'       # _count/_history attributes: {'a': 4}/{'a': ['a2', 'a3']}
```

   Note that **d**'s history dictionary stores only a maximum of **2** association before the current one. So at the end, the key **'a'** is associated with **'a4'**; this key has been associated with values **4** times (see **_count**) and the **2** values that it was associated with most recently were **'a2'** and **'a3'** (see **_history**).

3. It is possible to **get** the association of a key in two ways in **__getitem__**. For the above example:
   - Normal: **d['a']** evaluates to **'a4'**, the current value associated with **'a'**.
   - Historical: **d['a',-1]** evaluates to **'a3'** (the value 1 before the current association); **d['a',-2]** evaluates to **'a2'** (the value 2 before the current association).

   Note that writing **d['a',-1]** binds the parameter after **self** in **__getitem__** to the **tuple ('a',-1)**. So, you should...
   - ... first check if the parameter is a key in the **history_dict** and if so, return its value.
   - If not, assume the parameter is a **2-tuple** storing a key followed by a negative integer and use those to return the approprate earlier association.

   If there are any problems trying to get a **normal** or **historical** association (e.g., specifying a key that doesn't exist, or **d['a',-3]** in a **history_dict** that keeps only **2** previous associations), raise a **KeyError** exception (you may omit any message).

4. Finally, write a generator function named **iter_frequency** which produces **2-tuples** for every **key**/**value** association in the **history_dict**, in decreasing order by how often each **key** was associated with a **value**; if two **keys** were associated the same number of times, produce **2-tuples** in increasing alphabetical order based on the string representation of the **key**.

   If we change key **'a'** 2 times, key **'b'** 4 times, and key **'c'** 2 times, the keys in the **2-tuples** would appear in the order **'b'**, then **'a'**, then **'c'**.

You can write a special **__str__** method, so that you can easily print all the relevant information in a **history_dict**, to help you debug the methods in this class. For example, you might return a string containing the contents of the **history_dict** and its two auxiliary dictionaries. This method is totally optional: I will test no requirements related to it; if you don't supply one, it will inherit the one in the **dict** base class.

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}
```


5 of 6

The same is true after executing

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

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

```
  d2 = history_dict(2, [('a',1), ('c',3)], b=2, c=4)
or
  d2 = history_dict(2, {'a': 1, 'c': 3},   b=2, c=4)
```

d2 stores the contents **{'a': 1, 'b': 2, 'c': 4}** and the auxiliary dictionaries store the contents **{'a': 1, 'b': 1, 'c': 2}** (for **_count**) and **{'c': [3]}** (for **_history**).

6 of 6