



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, recursive functions, single inheritance, and writing dunder methods. You are free to 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 `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 **lists** 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 **lists**.

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 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 always 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 recursing 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 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 **strs**) and the second **lambda** specifies what **values** are legal (here they must be non-negative **ints**).

```
d = check_dict(lambda x : type(x) is str,\n               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 associations 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 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)
```