COMP 651 Data Analytics Techniques Assignment 2

10 points possible

1 (7 pts.). Write a Python script that inputs a list of integers and passes it to the following functions, which you implement. Do not have any functions input or output values. All values used by a function should be input before the function is called and passed to it as arguments. The value computed by a function should be returned, not output by it. Use the test code provided below, which satisfies these constraints and outputs the results of all function applications so we can test your implementations. The test code asks the user at the beginning to input a list of integers, which is used as the running list for testing the functions. When we say below that a function is passed the running list of integers, it could actually be passed any list of integers; it just turns out that the running list is what is passed in our test code.

- A function **`min_max()`** that is passed any number of integer arguments and returns a tuple with the smallest and largest among them. Use the "arbitrary argument lists" technique. If only one argument is provided, the two elements in the tuple will be the same. If no argument is provided, return an empty tuple. (To use the list that you input at the beginning, you have to unpack it (use a '*') when you provide it as an argument. Note that, if you do this, you'll be unpacking the list (with '*') when you pass it in and packing the arguments back up (using again a '*') in the argument list in the function definition.)
- A function **`alternates()`** that returns a 2-tuple of lists. The first list has every other element of the list **`ls`** passed in starting at index **`start`** (provided as a second parameter), and the second list has the other elements of **`ls`**, starring at **`start+1`**. The default value for **`start`** is 0. Use slicing. To use slicing to get every other element of a list, see the *Step Argument* slides available from where you downloaded this assignment.
- A function **`list_diff()`** that is passed the running list of integers and a second list of integers. (Prompt for and read the second list of integers before the function call and pass it (and the original list) to the function.) The function should return a list of all the integers in the running list that are not also in the second list. For example, if the two lists are **`[5, 1, 4, 2]`** (as the running list) and **`[6, 1, 4, 3]`** (as the second list), the function should return **`[5, 2]`**. Use **`filter`** and convert the **`filter`** application to a list.
- Write two versions of essentially the same function, call them **`sum_pairs()`** and **`sum_pairs1()`**. It returns a list containing, for each adjacent pair (n_0, n_1) in the list passed in, the value of $n_0 + 2\times n_1$. For the first version (**`sum_pairs()`**), define within the scope of **`sum_pairs()`** a function **`add_double_2nd()`** that, passed a 2-tuple **`t`** of integers, returns **`t[0] + 2*t[1]`**. **`sum_pairs()`** itself returns the result of mapping this function over the list of adjacent pairs in the list passed in. The second version, **`sum_pairs1()`**, does the same but uses a lambda expression instead of a local function. To get the zipped list of adjacent pairs of the list passed in (which you can map or iterate over), use the **`pairwise()`** function defined below (and available on the assignment page), where the parameter is any iterable (such as a sequence). It uses function **`tee()`** from the **`itertools`** library. (Later in the semester, we shall cover iterators, Python functionals that, e.g., provide elements one at a time (see **`next()`** in the listing below) in a **`map`** application or a **`for`** loop.)

```
import itertools
def pairwise(iterable):
    a, b = itertools.tee(iterable)
    next(b, None)
    return zip(a, b)
```

- A function **`cum_sum_pairs()`** that is passed the running list of integers and returns a list of pairs (2-element tuples), where the first element of a pair is the original number and the second is the sum of all previous numbers in the list (0 for the first list element). Consider using list comprehension. Use list comprehension
- A function **`pairs_smaller()`** that is passed the running list of integers and returns a list of pairs (2-element tuples), where the first element in the pair, call it **`x`**, is the original number, and the second is a list of all those numbers in the running list that are less than **`x`**. Use a nested list comprehension.

Recall that you can input an entire list (written with "`[...]`") in one line from the terminal using **`eval(input(⟨`*`prompt`*`⟩))`**.

The following is some test code (available on the assignment page) that you may include in your program file. The console interaction is listed below the code.

```
lst = eval(input("Input a list: "))

print('The smallest and largest elements: ', min_max(*lst))
print('The smallest and largest of no arguments: ', min_max())

start = int(input('The start index, 0 for the default: '))
print('The alternating sequences are ', alternates(lst, start=start)
                                 if start else alternates(lst))
start = int(input('The start index, 0 for the default: '))
print('The alternating sequences are ', alternates(lst, start=start)
                                 if start else alternates(lst))

lst1 = eval(input("Input another list: "))
print('The difference of the two lists: ', list_diff(lst, lst1))

print("The result of adding 1st of a pair to 2 X 2nd of a pair: ",
      sum_pairs(lst))
print("The same result, using a lambda: ", sum_pairs1(lst))

print('The first list of pairs as described: ', cum_sum_pairs(lst))

print('The second list of pairs as described: ', pairs_smaller(lst))
```

```
>>> runfile('C:/User/c651/HW2/Prob1.py', wdir='C:/User/c651/HW2')
Input a list: [7, 2, 12, 9, 15, 4, 11, 5]
The smallest and largest elements:  (2, 15)
The smallest and largest of no arguments:  ()
The start index, 0 for the default: 0
The alternating sequences are  ([7, 12, 15, 11], [2, 9, 4, 5])
The start index, 0 for the default: 2
The alternating sequences are  ([12, 15, 11], [9, 4, 5])
Input another list: [11, 3, 15, 2, 8]
The difference of the two lists:  [7, 12, 9, 4, 5]
The result of adding 1st of a pair to 2 X 2nd of a pair:  [11, 26, 30, 39,
23, 26, 21]
The same result, using a lambda:  [11, 26, 30, 39, 23, 26, 21]
The first list of pairs as described:  [(7, 0), (2, 7), (12, 9), (9, 21),
(15, 30), (4, 45), (11, 49), (5, 60)]
```

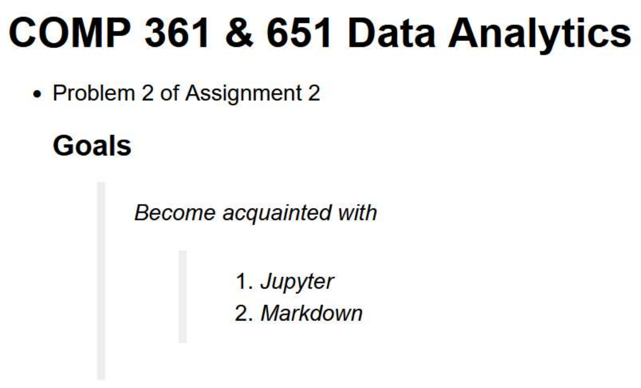
```
The second list of pairs as described:  [(7, [2, 4, 5]), (2, []), (12, [7,
2, 9, 4, 11, 5]), (9, [7, 2, 4, 5]), (15, [7, 2, 12, 9, 4, 11, 5]), (4,
[2]), (11, [7, 2, 9, 4, 5]), (5, [2, 4])]
```

Include docstrings (not the one-line kind) with all you function definitions.
Recall that, when you run pydoc, it executes the code, so don't have top-level code in your file—just function and class definitions AND test code. Put test code inside a conditional

```
if __name__ == "__main__":
```

This condition is true if the file is executed as the current file (not, e.g., as a imported module)—as is the case when it is in the editor pane in Spyder or from the command line. The test code made available already is within the context of such a conditional.

2 (3 pts.) Produce a Python 3 Jupyter notebook whose contents we describe. The first cell (a Markdown cell) renders as in the following screenshot.



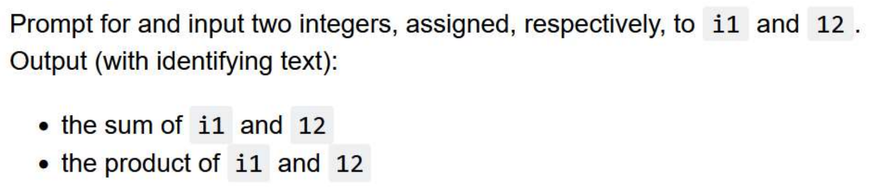
COMP 361 & 651 Data Analytics

- Problem 2 of Assignment 2

Goals

> *Become acquainted with*
>
> > 1. *Jupyter*
> > 2. *Markdown*

This cell is followed by another Markdown cell that renders as follows. (Make all code monospace, as shown. Use backquotes.)



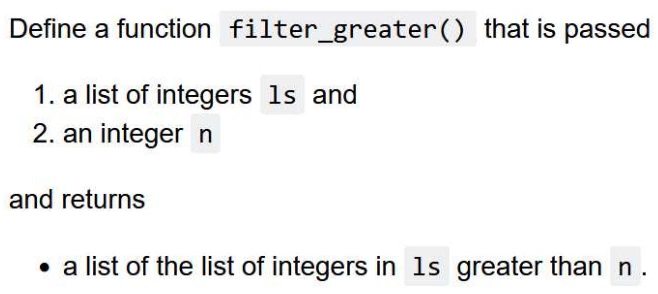
Prompt for and input two integers, assigned, respectively, to `i1` and `12`.
Output (with identifying text):

- the sum of `i1` and `12`
- the product of `i1` and `12`

The next cell is a code cell. The following is an example of the interaction when running it (where 2 is entered for **`i1`** and 3 for **`i2`**). (It produces a box for each **`input()`** command.)

```
Enter an integer: 2
Enter another integer: 3
The sum is  4
The product is  6
```

Next is a Markdown cell, rendered as follows.



Define a function `filter_greater()` that is passed

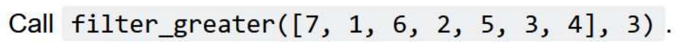
1. a list of integers `ls` and
2. an integer `n`

and returns

- a list of the list of integers in `ls` greater than `n`.

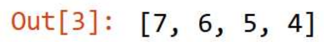
Then comes a code cell with the definition of function **`filter_greater()`**. You must execute this cell to apply **`filter_greater()`** later, but executing it results in no output.

The next cell is a Markdown cell rendering as follows.



Call `filter_greater([7, 1, 6, 2, 5, 3, 4], 3)`.

Finally comes a code cell making the call just mentioned. The following is the output. (Note that the output here is numbered as it is the value of the last (and, it happens, only) statement in the cell. This is unlike what happens when we print output—as above.)



```
Out[3]: [7, 6, 5, 4]
```