Lab 10

Learning Objectives:

- **Part 1** gives you some code that returns errors so that you can practice fixing it.
- **Part 2** guides you through handling exceptions for bad user input.
- **Part 3** will help you put all the different parts of a list comprehension together, one by one.
- **Part 4** will have you work with small examples showcasing concepts from functional programming.

Part 1: Testing Numerical Code, Finding Errors, and Fixing Bugs

In the starter repository, you should find a script called `quadratic.py`. In this is a `find_roots` function that returns the 2 roots of a quadratic equation of the form $ax^2 + bx + c = 0$. It's designed to be run from the command line, with `a`, `b`, and `c` given as arguments. For example,

```
python quadratic.py 1 2 -15
```

should print both roots of $1x^2 + 2x - 15 = 0$. You can also use `import quadratic` from the IPython interpreter to access specific functions. For example:

```
root1 , root2 = quadratic.find_roots(1, 2, -15)
```

will perform the same computation as above.

Now test this function on some points recommended here, as well as your own tests:

- $(a, b, c) = (1, 4, 4)$ has roots $-2, -2$.
- $(a, b, c) = (1, 4, -4)$ has roots $0.828, -4.828$
- $(a, b, c) = (2, 0, -4)$ has roots $1.41, \ -1.41$.

Make sure the test result matches the roots provided above! Optional: use assertions.

See what errors you are getting and fix them (or output a descriptive error message to the user on how to proceed). **Most of the time you'll get error messages, but sometimes you may just be exposing bugs within the program that generate incorrect results.**

Part 2: Catching Exceptions

Using `try`/`except` error handling, modify `quadratic.py` so that it can handle bad inputs (something that is not reasonable for the expected parameters `a`, `b`, `c`). For the following examples, see if you can make the program gracefully tell the user what went wrong.

- If you run the program with non-numerical input: `python quadratic.py physics 91si is the best`.
- If you run the program with a quadratic function that has imaginary roots, such as $(a, b, c) = (1, 0, 4)$.
- If you run the program with a zero leading coefficient, $a = 0$.

There are probably ways of doing this part that do not require using `try`/`except` error handling. **The point**

of this part is to practice that particular syntax, though, so see if you can figure out a method that relies on catching exceptions.

Part 3: Building a List Comprehension from the Ground Up

Comprehensions have a lot of pieces of syntax. This part will help you use them all together, implementing one at a time. When you're done, save your solutions in a file named `list_comp.ipynb`.

Write a list comprehension that:

- **Returns a list of the first 10 letters of the alphabet.** Hint: here's an alphabet: (import the string module first) `string.ascii_lowercase`.
- Returns a list of the first 10 letters of the alphabet **except for the sixth letter.**
- Returns a list of the first 10 letters of the alphabet, except the sixth one, **each repeated 1, 2, and 3 times:** `[‘a’, ‘aa’, ‘aaa’, ‘b’, ..., ‘c’, ...]`
- Returns the list like the above, but in a grid: `[[‘a’, ‘aa’, ‘aaa’], [‘b’, ...], [‘c’, ...], ...]]`
- Returns a list like the above, but if the number matches the index of the character *mod 3* (e.g. `‘c’ and 3`, instead print a single capitalized version of that character: `[[‘A’, ‘aa’, ‘aaa’], [‘B’, ‘bb’, ‘bbb’], [‘C’, ‘cc’, ‘ccc’], [‘D’,...]]`

Part 4: Function-fu

This part of the lab is designed to be done interactively, using an IPython Jupyter Notebook and the NumPy library functions. You are now familiar with nearly all the major features in Python, and now is your chance to see how powerful the language can be. Your task is to write functions for the following tasks, in as compact and elegant a form as possible. When you're done, your solutions should be all written in a IPython notebook named `functions.ipynb`.

- Squares: Create a list of the first 10 square numbers. First use a list comprehension, then use a `lambda` function and `map`.
- Products: Use a `lambda` function with `reduce` to multiply up the numbers from 1 to 5 (use `from functools import reduce`).
- Filenames: Say that you have a string with a list of file names `“test1.py test2.py test3.py ...”` and you want to output a list of just the filenames without the .py extension. Do this with a list comprehension.
- Filenames bonus: See if you can only return the filenames of .py files and ignore files with other extensions.