Lab 9

Learning Objectives:

II. **How do I plan a project and execute my plan?** *How do I use IPython towards an efficient workflow?* This lab will give you the opportunity to dip your toes into the tab-completion, traceback highlighting, and magic functions of this essential scientific computing tool.

III. **What broke and how do I fix it?** *How do I interpret Python exceptions?* This lab will give you the opportunity to work with buggy code, read the exceptions raised, and figure out how to fix them. *How do I interact with Python exceptions?* This lab will give you the opportunity to wrangle with Python's error objects so that your code responds intelligently to unexpected behavior.

IV. **How do I communicate science and Python with others?** *How do I document classes and functions?* This lab will give you the opportunity to practice writing docstrings.

F. **Write and Test, Write and Test....** *Write code in small bits that you can easily test.* If you code an enormous project and then test it all at once, how do you know where your error is? This lab gives you the opportunity to practice testing your code at regular intervals. Object-oriented programming makes it particularly easy to organize code in ways that can be tested.

Part 0 helps you practice writing docstrings in preparation for Milestone 3.
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 0: Docstrings

Copy your `particle.py` code from last lab into your repository for this lab. Then document the `Particle` class and its `get_force` function. Verify that you can access your documentation by importing it in IPython and using the `?` after the function to see the documentation.

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.