Project 2

Due: Wednesday, July 21st at 11:59 p.m. PDT

Please read the **entire document** before writing any code.

Contents

1 Introduction

1.1 Objectives

For this project, you will:

- practice making decisions in Python,
- practice writing loops in Python,
- experience some of the issues and subtleties involved with solving problems that include repetitive tasks,
- practice writing and calling functions that you have written,
- practice testing your code, and
- appreciate the benefits of modular development.

1.2 Collaboration

You **may not collaborate** in any way on your project. See the syllabus for more details.

1.3 Required Header File

All students are *required* to have the following header comment at the top of their source files. Note that the stuff to the right of the colon in bold is information for an imaginary example student. (Your header comment is not expected to have bold text.)

```
# Project 2
#
# Name: ‹Your name here›
# Instructor: Brian Jones
# Section: ‹Your section here›
```

2 Problem Description

You are part of a team writing a program that simulates landing the Lunar Module (LM) from the Apollo space program on the moon. The simulation picks up when the retrorockets cut off and the pilot/astronaut takes over control of the LM. The user of the simulator specifies the initial amount of fuel and initial altitude. The LM starts at the user-specified altitude with an initial velocity of zero meters per second and has the user-specified amount of fuel on board.

The manual thrusters are off and the LM is in free-fall. This means that lunar gravity is causing the LM to accelerate toward the surface according to the gravitational constant for the moon. The pilot can control the rate of descent

using the thrusters of the LM. The thrusters are controlled by entering integer values from 0 to 9 which represent the rate of fuel flow. A value of 5 maintains the current velocity, 0 means free-fall, 9 means maximum thrust, and all other values are a variation of those possibilities and can be calculated with an equation that is provided below.

To make things interesting (and to reflect reality) the LM has a limited amount of fuel. If you run out of fuel before touching down, then you have lost control of the LM and it free falls to the surface.

The goal is to land the LM on the surface with a velocity between 0 and -1 meters per second, inclusive, using the least amount of fuel possible. A landing velocity between -1 and -10 meters per second (exclusive) means you survive but the LM is damaged and will not be able to leave the surface of the moon. You will be able to enjoy the surface of the moon as long as your oxygen lasts but you will not leave the moon alive. Velocities faster than (less than) or equal to -10 meters per second mean the LM sustains severe damage and you will die immediately.

The initial conditions of 1300 meters altitude and 500 liters of fuel were used in the original, and were chosen so than an optimal landing should use approximately 300 liters of fuel. It's a considerable challenge at first, and you may enjoy trying it.

FYI: This program is modeled after a version that was popular on HP-28S programmable calculators in the 1970s. It is interesting to note that the desktop computers you are working on cost about the same amount (unadjusted for inflation) as the calculators did but are quite a bit more powerful (more memory, storage, computational power, not to mention more sophisticated display, keyboard, mouse, et cetera). This program took approximately 90 seconds to load on the calculator but will be virtually instantaneous on the machines you are running today. Credits to Dick Nungester of HP for the provided equations.

Before you begin, take a look at the sample solution program in the `executables` directory of your GitHub repository and try running it. There are instructor versions for Mac, Windows, and the CSSE department Unix servers named `moonlander_mac`, `moonlander_windows`, and `moonlander_unix` respectively. You may find it convenient to move the relevant executable from the `executables` directory into the root directory for your repository. You will need to put `./` in front of the filename to run it. And, these are not Python files, so you do not put python in front. For example, to use the Mac executable, you would type this at the terminal:

```
./moonlander_mac
```

If this fails and says some variant of "Permission denied", run the following command and try again:

```
chmod +x moonlander_mac
```

Start at 1300 meters with 500 liters of fuel and see how you do.

This project has two parts:

1. In Section 3, your manager asks you to write a group of functions that will eventually be used to complete the finished Moonlander Project. Your task for Part 1 is to write and test the functions you've been asked to implement. Your function definitions will be written in their own file called `lander_funcs.py`. This file has already been started for you. The functions you write will fall into two categories, functions that calculate and functions that do I/O (input/output). Test your functions that calculate using the unit testing tools that we have been using all quarter in a file called `funcs_tests.py`. This file has also already been started for you. You will test your functions that do I/O using a file that has already been completely written for you called `io_tests.py`.
2. In Section 4 (start this only after you have written and thoroughly tested your functions), your boss is so impressed with your work on the functions that you will be asked to complete the Moonlander Project. Believe it or not, completing Part 1 of the project is the majority of the work. Developing your code this way allows you to test your functions before moving on to code the main game loop.

   NOTE: The `io_tests.py` from Part 1 does not do moon landings. All it does is test that your functions exists, work correctly, and they can be called without crashing the program. Moon landings have to wait for Part 2.

3 Specification: Functions

You are to implement the following functions in `lander_funcs.py`. I have started this file for you. It contains "stubs" for all the functions. That is, the functions are all in the file, but are empty except for the keyword `pass`. Remove the word `pass` as you implement a function. This way, the entire file can be run and tested incrementally as you start to complete the individual functions.

```
show_welcome()
get_altitude()
get_fuel()
display_lm_state(elapsed_time, altitude, velocity, fuel_amount, fuel_rate)
get_fuel_rate(current_fuel)
display_lm_landing_status(velocity)

update_acceleration(gravity, fuel_rate)
update_altitude(altitude, velocity, acceleration)
update_velocity(velocity, acceleration)
update_fuel(fuel, fuel_rate)
```

- Note that input and output occur only in the functions which specifically have that job—i.e. the first six functions. The functions that calculate *do not* do I/O, and vice versa.
- References to tp_0 indicate the immediately previous time period (1 second ago) and references to tp_1 indicate the current time period.

3.1 Functions that do I/O

3.1.1 `show_welcome()`

This function displays the welcome message. See the sample program for the exact text.

3.1.2 `get_altitude()`

This function has no parameters and returns a `int`. The function will prompt the user for a value between 1 and 9999, inclusive. It must display an error message if the user enters a value outside this range and re-prompt for a valid value. See the sample program for the exact text of the prompt and any error message(s).

3.1.3 `get_fuel()`

This function has no parameters and returns an `int`. The function will prompt the user for a positive integer value and return it. It must display an error message of the user enters a negative or zero value and re-prompt for a valid value. See the sample program for the exact text of the prompt and any error message(s).

3.1.4 `display_lm_state(elapsed_time, altitude, velocity, fuel_amount, fuel_rate)`

This function must display the state of the LM as indicated by the parameters. The parameters are:

1. an `int` representing the elapsed time the LN has been flown;

2. a `float` representing the LM's altitude;
3. a `float` representing the LM's velocity;
4. an `int` representing the amount of fuel on the LM; and
5. an `int` representing the current rate of fuel usage.

See the sample program for the exact text and format of the display.

3.1.5 `get_fuel_rate(current_fuel)`

This function takes an `int` representing the current amount of fuel in the LM. The function prompts the user for an integer value and makes sure it is between 0 and 9, inclusive. It must display an error message if the user enters a value outside this range and re-prompt for a value value. *The function must return the lesser of the user-entered value or the amount of fuel remaining on the LM (value passed in as a parameter). The user cannot user more fuel than is left on the lunar lander!* See the sample program for the exact text of the prompt and error message(s).

3.1.6 `display_lm_landing_status(velocity)`

This function displays the status of the LM upon landing. There are three possible outputs depending on the velocity of the LM at landing, they are:

```
Status at landing - The eagle has landed!
Status at landing - Enjoy your oxygen while it lasts!
Status at landing - Ouch - that hurt!
```

1. Print the first message if the final velocity is between 0 and -1 meters per second, inclusive.
2. Print the second message if the final velocity is between -1 and -10 meters per second, exclusive.
3. Print the third message if the final velocity is $\leqslant -10$ meters per second.

See the sample program to see them in action.

3.2 Functions that Calculate

3.2.1 `update_acceleration(gravity, fuel_rate)`

The parameters are:

- a `float` representing the gravitational constant for the moon (a value of 1.62 will be used when calling this function in Section 4); and
- an `int` representing the current rate of fuel usage.

This function calculates are returns the new acceleration using the following formula.

$$\text{acceleration}_{\text{tp}_1} = \text{gravity} * \left(\frac{\text{fuel\_rate}_{\text{tp}_1}}{5} - 1 \right)$$

3.2.2 `update_altitude(altitude, velocity, acceleration)`

The parameters are:

- a `float` representing the previous altitude;
- a `float` representing the previous velocity; and
- a `float` representing the current acceleration.

This function calculates and returns the new altitude using the following formula.

$$\text{altitude}_{\text{tp}_1} = \text{altitude}_{\text{tp}_0} + \text{velocity}_{\text{tp}_0} + \frac{\text{acceleration}_{\text{tp}_1}}{2}$$

Remember, however, that the surface of the moon stubbornly limits the altitude to non-negative values, and your code must do the same.

3.2.3 `update_velocity(velocity, acceleration)`

The parameters are:

1. a `float` representing the previous velocity; and
2. a `float` representing the current acceleration.

This function calculates and returns the new velocity based on the following formula.

$$\text{velocity}_{\text{tp}_1} = \text{velocity}_{\text{tp}_0} + \text{acceleration}_{\text{tp}_1}$$

3.2.4 `update_fuel(fuel, fuel_rate)`

The parameters are:

1. an `int` representing the previous amount of fuel on the LM; and
2. an `int` representing the current rate of fuel usage.

This function calculates and returns the amount of fuel remaining based on the following formula. (Note: This is a trivial function as long as your `get_fuel_rate` function behaves correctly.)

$$\text{fuel}_{\text{tp}_1} = \text{fuel}_{\text{tp}_0} - \text{fuel\_rate}_{\text{tp}_1}$$

3.3 Function Testing

This part will describe in detail how to test your Moonlander functions that you have written. You are responsible for ensuring they are completely correct before handing them in to your manager.

3.3.1 Testing Functions that Calculate

Test these functions using the unit testing techniques we have been using all quarter. You must provide at least two tests per function. Each test should be written in its own testing function. Be sure to give your tests descriptive function names indicating which function is being tested. I provide you with a starting file `funcs_tests.py` file. Add additional tests to this file.

3.3.2 Testing Functions that do I/O

Testing functions that do I/O is a little trickier. To assist you, I have written a file called `io_tests.py` that calls each of your I/O functions in order. Do NOT change this file. I also provide you with instructor executables called `io_mac`, `io_windows`, and `io_unix` that call the same functions in the same order with the same input parameters. Play with the instructor executable until you understand how each of the I/O functions should behave.

Additionally, I provide you with one sample input file (`io_in1.txt`) to test the output from your `io_tests.py` versus the output from my instructor executable. You can get the instructor executable from the `executables` directory of your GitHub repository and the input file from the `test_files` directory of your GitHub repository.

To test your I/O functions using `io_tests.py`, follow these steps (in these steps I use the Mac version of the executable, if you're on Windows or a different unix operating system, use the appropriate executable):

1. Run the instructor executable until you are very familiar with how the code should behave. Then run your code by typing the command below. Be sure your `lander_funcs.py` file is in the same directory.

   ```
   python3 io_tests.py
   ```

   If your version does not behave the same as the instructor, modify the appropriate function in `lander_funcs.py` and test some more. Once you are confident that your code behaves just like the instructor version, move on to step two.

2. Run the instructor executable with the `io_in1.txt` input file and save the results in a file called `inst_out1.txt` (short for instructor test 1):

   ```
   ./io_mac < io_in1.txt > inst_out1.txt
   ```

   If this fails and says some variant of "Permission denied", run the following command and try again:

   ```
   chmod +x io_mac
   ```

3. Run the `io_tests.py` module with the `io_in1.txt` test file and save the results in a file called `my_out1.txt`:

   ```
   python3 io_tests.py < io_in1.txt > my_out1.txt
   ```

4. Diff these outputs to see if they match:

   ```
   diff inst_out1.txt my_out1.txt
   ```

Be sure to make your own additional input files to test your I/O functions. I recommend testing with invalid input for initial altitude and fuel amount. For example, try entering a negative number for the initial altitude. You need NOT—shouldn't—test with input other than integers. Pay close attention to where you put blank lines in your I/O functions, you'll want to have this correct before moving on.

4 Specification: Moonlander Simulation

- Begin this part only after **fully testing** your functions in Section 3. (If your functions don't work, there is no hope that this does).
- Now you must write a `moonlander.py` that runs the full lunar lander simulation. I recommend that you run the instructor version of the fully functioning program until you are familiar with how the program behaves.
- In your `moonlander.py`, you must code a simulator-loop that advances the LM from time period zero to landing, and calls to the functions of Part 1 along with minimal code to "glue" it together. One of the challenges in this project will be understanding the overall structure that has been provided to you. Don't hesitate to ask me questions early in the process to help make sure you understand what you are supposed to be doing. You will probably go down some dead end and need to remove and/or rewrite code. This is part of the process so enjoy the ride!
- You do *not* need to write nested loops in this part. Nor do you need to write a loop followed by another loop. The problem may be solved in both of those ways, but it can be solved more simply by writing a single loop.

4.1 Testing the Moonlander Simulation

Diff your output versus the instructor solution executable. I provide you with three sample test cases (`moon1`, `moon2`, `moon3`). Be sure to test with some of your own. My "final" tester has many more tests.

5 Grading

Your program grade will be based on:

- thoroughness of your `funcs_tests.py`,
- adherence to the specification,
- the number of unit test cases passed,
- the number of `diff` test cases passed, and
- program style.

Your program will be tested with test cases that you have not seen. Be sure to test your program thoroughly!!! To pass a test case, your output must match mine EXACTLY. Use `diff` to make sure your output matches mine in the test cases given and in test cases that you make up. If there are any differences whatsoever, you will not pass the test case.

6 GitHub Submission

Push your finished code back to GitHub. Refer to Lab 1, as needed, to remember how to push your local code.