CPE 101
Project 2-Moonlander - Lunar Lander Simulator
Part A

Objectives:

- Practice on making decisions in python.
- To Practice writing loops in python.
- To experience some of the issues and subtleties involved with solving problems that include repetitive tasks.
- To practice writing and calling functions with return value and those perform I/O operations.
- To practice testing your code.
- To appreciate the benefits of modular development.

Required File Header:

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 red is information for an imaginary example student. (Please put YOUR appropriate information. Also, your header comment is not expected to have red text.) All program files require this header.

```
# Project 2 - Moonlander
#
# Name: Your Name
# Instructor: S. Einakian
# Section: Your Section
```

Resources:

- Your instructor, via office hours, email, etc.
- Tutoring
- Your TA

- **Given Codes and Files:**
  - List of functions that you need to implement are in the landerFuncs.py.
  - Unit Test for all implemented functions must be in the funcs_tests.py.
  - A starting tester file for your program I/O functions, ioTests.py (the complete program is given).
  - Two executable file moonlanderInst and ioInst, which you can run to find out how your program should work. (Same as skaterInst in Project 1)

Download these files from PolyLearn to your directory.

Ground Rules:

- You may not discuss this project with anyone other than your instructor, TA, or the tutors.
- Your program must not use any global data (that is, all data must be declared within some function).
- Your program must implement the required functions as specified, including function names and input parameters.
- Your program must use the required functions appropriately. That is, it's not enough to create the functions, you must also use them.
- Your program must mimic the sample program's behavior ***exactly*** and in all cases.

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 a user-specified altitude with an initial velocity of zero meters per second and has a user-specified amount of fuel on board.

The manual thrusters are off and the LM is in free-fall -- meaning 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 -- 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 you have lost control of the LM and it freefalls 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 meters per second 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 die immediately.

The initial conditions of 1300 meters altitude and 500 liters of fuel were used in the original, and were chosen so that 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 -- virtually instantaneous on the machines you are running on today. Credits to Dick Nungester of HP for the provided equations.

Before you begin, take a look at the sample solution program by copying it to your account and running it. First log on to the UNIX servers and then navigate to the directory where you would like to copy the file through PolyLearn. Run the program using following command:

```
./moonlanderInst
```

Start at 1300 meters with 200 liters of fuel, and see how you do.
(if you had permission denied message, same as Project 1, use the following command to make the file executable:

```
chmod +x mmonlanderInst )
```

The project has two parts:

1. In Part A, 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 **landerFuncs.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 **ioTests.py**.

Part A: Functions

- You must implement the following functions in a file called landerFuncs.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.

- Note that input and output occur only in the functions which specifically have that job. The functions that calculate *do not* do I/O, and vice versa.

CAUTION: Pay attention to the data types involved in these equations and be sure you are not losing any necessary precision due to Python's rules for dealing with mixed-type expressions.

Functions that perform I/O:

1. **showWelcome()**

Display the welcome message. See the sample program for the exact text.

2. **getFuel()**

This function has *no parameters* and a *return positive integer.* This function must prompt a user for positive integer value and return it. It must display an error message if 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 error message.

3. **getAltitude()**

This function must prompt a user for a real value between [1, 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).

4. **displayLMState(elapsedTime, altitude, velocity, fuelAmount, fuelRate)**

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 LM has been flown;
2) a double representing the LM's altitude;
3) a double representing the LM's velocity;
4) an int representing the amount of fuel on the LM;
5) an int representing the current rate of fuel usage.

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

5. **getFuelRate(currentFuel)**

I/O - The parameter is 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 valid 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 use more fuel than is left on the lunar lander!* See the sample program for the exact text and formatting of the prompt and any error message(s).

6. **dispalyLMLandingStatus(velocity)**

This function, as its name implies, displays the status of the LM upon landing. There are three possible outputs depending of 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 <= -10 meters per second

See the sample program for the exact text and formatting.

Functions that Calculate:
NOTE: The "**tp**" (time period) subscripts are intended to show how the state at each new second of time (tp_1) depends on the state at the current second (tp0) plus the newly entered rate of fuel consumption (marked as tp_1). These formulas must be re-evaluated for every second of the simulation.
- References to $\mathbf{tp_0}$ indicate the immediately previous time period (1 second ago) and references to $\mathbf{tp_1}$ indicate the current time period.

1. **updateAcceleration(gravity, fuelRate)**

The parameters are:
1) a float representing the gravitational constant for the moon (i.e., 1.62).
2) an int representing the current rate of fuel usage

The function calculates and returns the acceleration using the formula
$\text{acceleration}_{tp1}$ = gravitational constant * ((rate of fuel consumption_{tp1} / 5) - 1)

2. **updateAltitude(altitude, velocity, acceleration)**

The parameters are:

1) a double representing the current altitude;
2) a double representing the current velocity;
3) a double representing the current acceleration.

The function calculates and returns the new altitude based on the provided inputs and the formula

$$altitude_{tp1} = altitude_{tp0} + velocity_{tp0} + (acceleration_{tp1} / 2)$$

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

3. **updateVelocity(velocity, acceleration)**

The parameters are:

1) a double representing the current velocity;
2) a double representing the current acceleration.

The function calculates and returns the new velocity based on the provided inputs and the formula

$$velocity_{tp1} = velocity_{tp0} + acceleration_{tp1}$$

4. **updateFuel(fuelAmount, fuelRate)**

The parameters are:

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

The function calculates the remaining fuel. Note: This is a trivial function as long as your getFuelRate function behaves correctly.

$$fuel_{tp1} = fuel_{tp0} - \text{rate of fuel consumption}_{tp1}$$

Testing (READ CAREFULLY)

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. Perform three tests per function.

Testing your functions that calculate:

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

Testing your functions that do Input/Output:

Testing functions that do I/O is a little trickier. To assist you, I have written a file called ioTests.py that calls each of your I/O functions in order. Do NOT change this file. I also provide you with an instructor executable called ioInst that calls 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 (in1) to test the output from your ioTests.py versus the output from my instructor executable (much the same way you tested your skater.py program). To get the instructor executable and the input file copy them from PolyLearn.

To test your I/O functions using ioTests.py follow these steps:

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

   python3 ioTests.py

If your version does not behave the same as the instructor, modify the appropriate function in landerFuncs.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 ioInst executable with the in1 input file and save the results in a file called inst1 (short for instructor test 1):

   ./ioInst < in1 > inst1
3) Run the ioTests.py module with the in1 test file and save the results in a file called my1:

   python3 ioTests.py < in1 > my1

4) Diff the output of both executables to see if they match:

   diff my1 inst1 -wB

The –wB flag will ignore the whitespace differences. If both files have same text, nothing will display in the terminal.

Be sure to make your own additional input files to tests your I/O functions. I recommend testing with invalid input for initial altitude and initial fuel amount. For example, try entering a negative number for the initial altitude. (You need NOT test with input other than integers.)

Submission:

Submit the following files:

- landerFuncs.py
- funcs_tests.py

to the PolyLearn.

Grading Rubric:

landerFuncs	10 Points
funcs_tests	30
pass ioTests	30
pass extra tests	20
comments	10

Full Credit (100%) : 1/30 @11:55PM
80%: 1/31 @11:55PM