CPE 101 ***Note, midterm 1 is on Monday, 3 May (possibly 10 May).
Spring 2021 ***Remind me to release the practice midterm exam one week before !
CRITICAL REMINDER to actually work the practice exam and then come to the review session on the Saturday before the Monday of the exam. It matters a lot. You may cooperate (please do!) on working and understanding the practice exam. There are questions on the actual exam that are very similar, maybe even the same question. AND, the format of the questions is important to work with before the actual exam, you should not waste time trying to figure out the format, how to answer the question. The practice exam will eliminate that problem.

Moonlander - Lunar Lander Simulator

Due Date: Sunday, 18 April, at 11:59 pm. If late, one week max at 70% credit.

Objectives

- More practice 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 that you have written.
- To practice testing your code.
- To appreciate the benefits of modular development: thinking out the design of the code before writing it.

Required File Header

As usual, 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: Cody Coderson
# Section: CPE 101-09 (or 11)
# Date:
```

Resources

- Your text.

- Your instructor.

- Tutoring Center - https://tutoring.csc.calpoly.edu/
- Given Code:
  - A starting functions file for your program functions, `landerFuncs.py`.
  - A starting testing file for your program functions, `funcsTests.py`.
  - A starting tester file for your program I/O functions, `ioTests.py`. (this may change)

These files are on Canvas under the Files section and are subject to update.

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 nearly ***exactly*** and in all cases.

Orientation

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.

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

The project has **two** parts:

1. In Part 1, 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 `funcsTests.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`.

2. In Part 2 (start this after you have written and *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 `ioTests.py` from Part 1 does **not** do moon landings -- all it does is test that your functions exist, work correctly, and they can be called without crashing the program. Moon landings have to wait for Part 2.

Part 1: 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.

- Each function must have at least 3 comments. Short, simple phrases.
  1. # input specs – name parameters and their types
  2. # output specs – give the type of the return value. Otherwise, indicate whether the function prints anything to the screen (or writes something to a file.)
  3. # very simple description of what the function does and anything interesting that should be noted.

- 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.

- References to tp0 indicate the immediately previous time period (1 second ago) and references to tp1 indicate the current time period.

- **NOTE**: The "tp" (time period) subscripts are intended to show how the state at each new second of time (tp1) depends on the state at the previous second (tp0) plus the newly entered rate of fuel consumption (marked as tp1). These formulas must be re-evaluated for every second of the simulation.

Functions that do I/O: (remember to differentiate between output to the screen and a return value back to the calling program.)

`showWelcome()`

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

`getFuel()`

This function has *no parameters* and a *return type of int.* 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.

`getAltitude()`

This function must prompt a user for a real 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).

`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 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;
5) an int representing the current rate of fuel usage.

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

`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).

```
displayLMLandingStatus(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 output for the exact text and formatting.

Functions that Calculate:

```
updateAcceleration(gravity, fuelRate)
```

The parameters are:

1) a float representing the gravitational constant for the moon (use a value of 1.62 when calling this function in Part 2);
2) an int representing the current rate of fuel usage

The function calculates and returns the acceleration using the formula

$\text{acceleration}_{tp1} = \text{gravitational constant} * ((\text{rate of fuel consumption}_{tp1} / 5) - 1)$

```
updateAltitude(altitude, velocity, acceleration)
```

The parameters are:

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

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

$\text{altitude}_{tp1} = \text{altitude}_{tp0} + \text{velocity}_{tp0} + (\text{acceleration}_{tp1} / 2)$

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

`updateVelocity(velocity, acceleration)`

The parameters are:
1) a float representing the current velocity;
2) a float representing the current acceleration.

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

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

`updateFuel(fuel, 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.

$fuel_{tp1} = fuel_{tp0}$ - rate of fuel $consumption_{tp1}$

Part 2: Moonlander simulation

Do NOT begin work on the actual Moonlander simulator yet. Instead, `funcsTests.py` and `ioTests.py` will solely be responsible for testing the functions you have written. You want to assure that they are perfect before you hand them over to your manager (me).

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 before handing them in to your manager.

Testing your 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 give your tests descriptive function names indicating which function is being tested. I provide you with a starting `funcsTests.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!** IF you need more output from program runs to help you figure out what your functions should do, ask me, you may give me the inputs for which you'd like to see the outputs.

Additionally, I provide you with one sample input file (`in1`) to test the output from your `ioTests.py` versus an output that I will provide (screenshot).

Part 2: Moonlander simulation

- Begin this part with your fully tested `landerFuncs.py` file.

- Now you must write a `moonlander.py` that runs the full lunar lander simulation.

- 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 your instructor 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 ends 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.

Testing

Copy the tests from here (although you already should have them from part 1):

```
Some test info may be included in a test folder on Canvas at the end
of this week.
```

Grading

Your program will be graded based on correct implementation of the moonlander functions in `landerFuncs.py`, my function tests passed, the thoroughness of your function tests in `funcsTests.py`, finished moonlander simulator tests passed with your `moonlander.py`, code style, and adherence to the specification.

Handin

`There will be a place under “Assignments” on Canvas to upload your work.`