Moonlander

CPE 101: Fundamentals of Computer Science
Winter 2019 - Cal Poly, San Luis Obispo

Purpose

To gain experience writing functions that return values or perform I/O operations, as well as integrate a collection of functions into a complete program.

Description

Download the project files at:
http://users.csc.calpoly.edu/~dkauffma/101/moonlander.zip

- `moonlander.py` Starting file for your program's functions
- `test[1-3].txt` Test cases for the full program

Use the `wget` command to download the archive to the CSL server:

```
wget http://users.csc.calpoly.edu/~dkauffma/101/moonlander.zip
```

Unzip the archive:

```
unzip moonlander.zip
```

Background

You are tasked with writing a program that simulates landing the Lunar Module (LM) from the Apollo space program on the moon. The simulation starts when the retro-rockets cut off and the astronaut takes control of the LM. 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. With the manual thrusters off and the LM in free-fall, 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 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 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 perish immediately.

The initial conditions of 1300 meters altitude and 500 liters of fuel were used in the original simulation 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.

Implementation

Instructor Executable

Before you begin, try the instructor's executable with the command `/home/dkauffma/moonlander.exe` on one of the CSL servers. Start at 1300 meters with 500 liters of fuel and see how you do.

Function Definitions

In the first part of this project, you must write and test the functions that will eventually be used to complete the finished Moonlander program. All function definitions must be written in moonlander.py, which has already been started for you. These functions fall into two categories: those that calculate and those that do I/O (input/output). Test your functions that calculate using assert statements in a file called mltests.py. You will test your functions that do I/O using the Python interpreter.

I/O Functions

See the executable for the exact text to be displayed with these functions.

```
show_welcome() -> NoneType
```

Display the welcome message.

```
get_altitude() -> int
```

Prompt the user for an integer in the interval [1, 9999]. If this value is invalid, display an error message and prompt the user again; otherwise, return this integer. This function must use a while loop.

```
get_fuel() -> int
```

Prompt the user for a positive integer. If this value is non-positive, display an error message and prompt the user again; otherwise, return the integer. This function must use a while loop.

```
get_fuel_rate(fuel: int) -> int
```

Prompt the user for an integer in the interval [0, 9]. If this value is invalid, display an error message and prompt the user again; otherwise, return the lesser of the user-entered value or the amount of fuel remaining on the LM (the argument passed to this function). In other words, the user cannot use more fuel than the amount remaining in the lander. This function must use a while loop.

```
display_state(time: int, altitude: float, velocity: float, fuel: int, fuel_rate: int) -> NoneType
```

Display the state of the LM as indicated by the parameters, using Python's string format function to ensure that all values are properly lined up. If the lander is out of fuel, display the corresponding message instead. At the start and end of the game, this function should also print the corresponding "LM state" message.

```
display_landing_status(velocity: float) -> NoneType
```

Display the status of the LM upon landing, conforming to one of three possible outputs depending of the velocity of the LM at landing, which are:

- Status at landing - The eagle has landed!
  Print if the final velocity is between 0 and -1 meters per second, inclusive.

- Status at landing - Enjoy your oxygen while it lasts!
  Print if the final velocity is between -1 and -10 meters per second, exclusive.

- Status at landing - Ouch - that hurt!
  Print if the final velocity is ≤ -10 meters per second.

Calculate Functions

The subscripts in the formulas below indicate whether the value is of the previous or next time step and will help you determine the order these functions must be called in the second part of the project. Subscripts should not be included as part of variable names.

```
update_acceleration(gravity: float, fuel_rate: int) -> float
```

Return the new acceleration based on the provided inputs and the formula:

$$acceleration_1 = gravity * ((fuel\_rate_1 / 5) - 1)$$

Use 1.62 as the gravitational constant for the moon when calling this function.

```
update_altitude(altitude: float, velocity: float, acceleration: float) -> float
```

Return the new altitude based on the provided inputs and the formula:

$$altitude_1 = altitude_0 + velocity_0 + (acceleration_1 / 2)$$

(As the surface of the moon stubbornly limits an altitude to non-negative values, ensure your code does the same).

```
update_velocity(velocity: float, acceleration: float) -> float
```

Return the new velocity based on the provided inputs and the formula:

$$velocity_1 = velocity_0 + acceleration_1$$

```
update_fuel(fuel: int, fuel_rate: int) -> int
```

Return the new amount of fuel based on the provided inputs and the formula:

$$fuel_1 = fuel_0 - fuel\_rate_1$$

`main` Function

Begin this part with your fully tested moonlander.py file. Now you must write a main function that simulates the advancement of the LM from time period zero to landing, calling the functions written previously along with some additional code to connect them together. 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 rewrite code, so be sure to allocate enough time.

In main, you do not need to write nested loops, nor do you need to write a loop followed by another loop. The problem may be solved in both of those ways, but it should be solved more simply by writing a single loop.

The main function should **not** do any I/O; that is, do not call `input` or `print` inside main.

Testing

Calculate Functions

Test these functions using assert statements in mltests.py. You must provide at least 3 tests per function. Each test should be written with its own assert statement.

I/O Functions

Test these functions in the Python interpreter:

1. `>>> import moonlander`
2. `>>> moonlander.show_welcome()`
3. Repeat Step 2 for other functions you wish to test

You should also test with invalid inputs to functions. For example, try entering a negative number for the initial altitude and ensure the proper error message is displayed. You need not test with inputs other than integers.

`main` Function

Use the `diff` command to compare your program's output against that of the moonlander.exe executable using the provided test cases. Be sure to test the program using other inputs as the instructor's test suite has additional test cases (in other words, some test cases are hidden from you).

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

   ```
   python3 moonlander.py
   ```

   If your version does not behave the same as the solution, modify the appropriate functions in moonlander.py and test some more. Once you are confident that your code behaves just like the instructor's version, go to Step 2.

2. Run the executable with the test1.txt file and save the results in a file called exe.txt (though you can name it anything):

   ```
   /home/dkauffma/moonlander.exe < test1.txt > exe.txt
   ```

3. Run your moonlander.py module with the test1.txt file and save the results in a file called my.txt (or, again, any name you prefer):

   ```
   python3 moonlander.py < test1.txt > my.txt
   ```

4. Compare both files to see if they match:

   ```
   diff exe.txt my.txt -sy
   ```

   The `-s` option will let you know if both files are equal (otherwise it succeeds silently). The `-y` option will display the content of both files side-by-side, with a `|` symbol indicating that that line has a difference.

5. Continue modifying moonlander.py and comparing its output to the instructor's version until the diff command reports that the files are identical.

6. Be sure to perform the above steps with the other provided test cases as well as any other cases that you create.

Submission

On a CSL server with moonlander.py and mltests.py in your current directory:

```
/home/dkauffma/casey.exe 101 moonlander
```