# Space Lander!

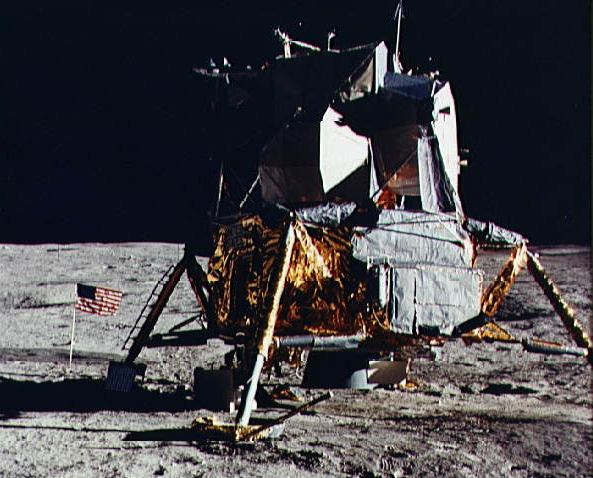
(Mini-Project 1)

This is a team or individual assignment. You may work with up one other person provided you indicate in your source code by making a comment at the top of your submission:

# Mini project 1 team: Ralph W. Emerson and Henry D. Thoreau

## Background

Computer simulation is used by engineers to test out ideas before actually building expensive machines or putting people in dangerous situations. Simulation is a critical part of the space program by NASA, for example. For this program, you will create a simulation of a vehicle landing on a celestial body.



Here is how it works: You are in control of a space lander ship, descending to the surface of the celestial body for a landing. Gravity steadily accelerates your ship faster and faster toward the surface. You, the astronaut piloting the ship, have a single control: a button with the label "thrust" on it. Applying thrust slows your ship down. Your goal is to get your ship to land at a slow enough speed so that it doesn't crash on impact. What's the catch? You have only a limited amount of fuel. If you slow down your ship too much too early, you will run out of fuel and crash into the surface.

Your mission: Program this simulation in Python. There are 4 parts to this assignment (complete them in order). Successful completion of each part will earn the following mini-project grades:

Part 1: 50%

Part 2: 80%

Part 3: 90%

Part 4: 100%

## Getting Started

## Download test\_lander.py and cisc106.py from Canvas and place it in your code folder. Create a file called lander.py in the same code folder and open both in Wing. You do not need to change any code in test\_lander.py.

## For all 4 parts of the project, use the Design Recipe to write your functions. When you believe you have your functions running properly, comment out your assertEqual() statements before running the test\_lander.py script.

## Part 1

In lander.py, you should create global variables to keep the following information:

* **altitude**: how far the ship is from the surface, in meters. Initial value: 1000 meters.
* **velocity**: how fast the ship is moving, in meters/second. A positive velocity means that that ship is moving toward the surface. A negative velocity means that the ship is moving away. Initial value: 40 meters/second.
* **fuel**: how many units of fuel are left in the tank. Initial value: 25 units.
* **strength**: the structural strength of the ship. Initial value: 4 units
* **gravity**: the acceleration from gravity of the celestial body in meters/second2. Initial value: 1.622 (the gravity of the moon)
* **velocity\_change\_per\_thrust**: the change in velocity for each unit of thrust applied. Initial value: 4 meters/second

You then need to add and implement code for each of the following functions (you may have more if you wish):

* **get\_status():** returns a string representation of the current status that looks like:  
  Alt = 4.00 Vel = 0.00 Fuel = 0 Str = 4

Make sure to match this format exactly (2 decimal places, spacing, etc)

* **thrust(number):** applies number units of fuel to give thrust to the ship, (one unit of thrust uses one unit of fuel) and decreases the velocity by number \* velocity\_change\_per\_thrust. Additional rules:
  + number may be either a float or int **-** round down to the nearest unit of fuel
  + if number is greater than the structural strength of the ship then it is reduced to the strength to avoid fracturing the ship's structure
  + if number is greater than the current fuel amount then only the current fuel amount is used to decrease velocity
  + reduce the fuel amount to reflect usage
* **update\_onesecond():** updates the velocity and altitude for one second of simulation.
  + Because acceleration and velocity are both stored with implicit per second unit, your new velocity is simply your old velocity plus gravity.
  + Your new altitude is your old altitude minus your velocity.
  + If altitude goes below zero, set it to zero
* **has\_crashed()**
  + returns True if the altitude is less than or equal to 0 and the velocity is greater than the structural strength of the ship
* **has\_safely\_landed()**
  + returns True if the altitude is less than or equal to 0 and the velocity is less than or equal to the structural strength of the ship

Run the tests in test\_lander.py to make sure you pass the Part 1 tests.

## Part 2

In lander.py, you need to add and implement code for each of the following functions (you may have more if you wish):

* **reset\_lander(a, v, f)****:**
  + Resets the altitude, velocity, and fuel to the parameters a,v,f.
* **human\_controller():**
  + Displays the current status to the user using the get\_status function
  + ask the user "How much thrust this round? “.
  + returns an amount of thrust as a whole number (rounded down) from user input
* **simulate\_landing(player):** 
  + Assume player is a function with no parameters that returns an amount of thrust.
  + while the lander has not crashed or landed safely
    - call player function to get the thrust value
    - apply the thrust returned from the player function
    - update the simulation one second
  + If the lander crashes, return the string "Oh no the lander has crashed! Better skill next time!"
  + If the lander safely lands, return the string "Great success! You should apply for an internship with NASA!"

Run the tests in test\_lander.py to make sure you pass the Part 2 tests.

**IMPORTANT: After completing Part 2 you can now play the simulation as a game! To do this, add the following two lines to the end of your lander.py file and run lander.py in Wing:**

**if \_\_name\_\_ == "\_\_main\_\_":**

**simulate\_landing(human\_controller)**

## Part 3

You need to design a smart controller to press the thrust button so that we can send the ship on an autonomous landing mission. It is up to you to figure out how much thrust to use at each iteration to safely land your lander.

**Optional:** Once you implement the function smart\_controller(), you can run turtle\_lander.py to graphically display your lander in action! (Not Required for Submission)

* **smart\_controller():** 
  + returns the amount of thrust to use on the next thrust
  + remember you can access the current velocity, altitude, fuel but this function should not update these global variables – it is just calculating the next thrust value to use based on the current state of these variables
  + try and save fuel unless you think you will crash! – the function can

return 0 thrust

* + Your smart controller will be tested starting at higher heights with different amounts of fuel. Try and make it as robust as possible.

Run the tests in test\_lander.py to make sure you pass the Part 3 tests.

## Part 4

Some celestial bodies have an atmosphere that effects the structural strength of your ship. To simulate this, we will add 2 more global variables

* **atmosphere\_lower**: The lower altitude of the atmosphere. Initial value: -1 to indicate no atmosphere.
* **atmosphere\_upper**: The higher altitude of the atmosphere. Initial value: -1 to indicate no atmosphere.

The longer your ship stays in the atmosphere the lower its strength becomes. You still must spend fuel to accelerate and you still must decelerate enough to land safely after passing through the atmosphere.

Define and/or update the following functions to take into account the atmosphere.

* **reset\_world(g, s, a\_l, a\_u):** Resets the gravity, strength, atmosphere\_lower and atmosphere\_upper to the parameters g,s,a\_l,a\_u.
* **update\_onesecond():** the same as update\_onesecond from part 1, except now if the ship is within the atmosphere lower and upper limits, the strength is reduced by 1.
* **has\_disintegrated():** returns True if the ship’s strength is 0 or below, False otherwise.
* **simulate\_landing(player):** The same as in part 2, except now you must check for disintegration in the main loop. If the ship has disintegrated, return the string "Oh no the lander has disintegrated in the atmosphere!"
* **thrust(number):** You now have thrusters on both the bottom and top of your ship and can activate the top thrusters by calling thrust with a negative value. This will cause your ship's velocity to increase instead of decreasing.
  + **Be careful:** for each unit of negative thrust, the ship still burns one unit of fuel
* **smart\_controller():** Returns an amount of thrust taking into account the atmosphere. It is up to you to figure out the best way distribute your thrusts.

Modify your smart\_controller to make it through the atmosphere and land safely! It must now pass the tests in part 4.