Testing of code and functionality

To test my game engine, I wrote input/output test cases and compared them with game output. I conducted unit testing through creating input cases for each individual branch of game engine, testing edge cases for each function.

In addition, I used assertions after processing movements of the spaceship, asteroids and bullets to ensure the movements were correctly updated, and to check that asteroids and bullets were correctly removed from collision and scores were updated accordingly. Examples of this included using unittest to perform assertEqual(), assertTrue() and assertFalse() statements to test my code.

To debug my game engine, I used the print statement in a divide and conquer method of debugging, splitting the lines of code into smaller sections to efficiently identify the source of the issue. This allowed me to quickly isolate the line and fix the issue, and then perform regression testing to ensure functionality of other methods are not impacted by the change.

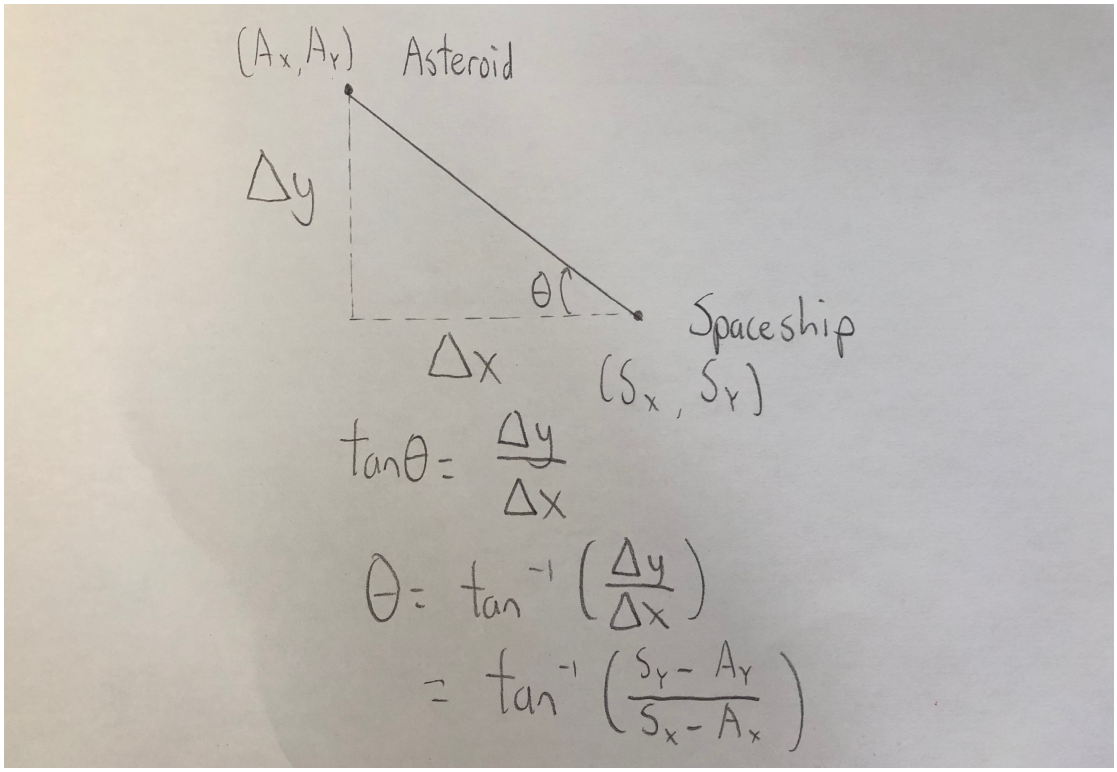
Writing test cases

I tested different ways of raising errors through creating a set of files to be inputted. Firstly, I created an empty file which would yield the game state incomplete message. I also created files which did not have some of the variables such as “score” and “fuel”, and lines which did not match with the expected number of lines. For example, if the asteroids_count was 3, I would only have 2 lines which contains asteroids. Both cases would invoke the game state incomplete message.

I then tested unexpected keys by adding in an extra variable at the end of an object such as spaceship, which would exceed the number of variables show in the self.__repr__() method. To test for value errors, I replaced floats and integers representing an attribute of an object, such as x-coordinate or angle, with strings, list and nulls, invoking the invalid data type message. Finally, I invoked the key and value pair error message by adding an additional argument at the end of a line.

Specific area of code which could be made more modular

In my player class, the lines 230-266 was a major component of my bot's functionality because it determined the correct direction which my bot would turn towards in order to move to the nearest asteroid. The code would find the angle of inclination between the spaceship and the asteroid using the arctan() function, and my first step was to determine whether the angle was negative. This is illustrated below:



The magnitude of the angle did not represent its quadrant, so I needed to implement a method to calculate the correct angle representation.

To make my code modular, I would create a new function which will help determine which of the quadrants the angle would lie in and return the angle in the quadrant. The spaceship's coordinates and the asteroid's coordinates could be input parameters. The different branches could be determined according to where the spaceship's x position was greater (further right) than the asteroid's x position, and vice versa, and likewise for the y-positions This would simplify the player.action() method.

How to solve scenario where the bot had a continuous line of fire in the direction it was moving instead of firing single bullets

The first step would be to create a function to produce a linear equation of the laser's line. This can be done using the Cartesian equation: $y - y_0 = m (x - x_0)$ where y_0 is the y-coordinate of the spaceship, x_0 is the x-coordinate of the spaceship, and m is the gradient of the line.

The gradient of the line can be calculated by the equation: m = tan(angle), where angle is the angle of the spaceship. The angle of the spaceship can be converted into radians using the function math.radians(angle) from the python math library, and the gradient is calculated using m = math.tan(math.radians(angle)).

This method can receive the spaceship's x-coordinate, y-coordinate and angle as input, calculate the gradient, and then also calculate the intercept of the line using y = mx + b and substituting in the three parameters. The gradient and intercept can be returned and the method will be called every time the spaceship's move_forward(), turn_left() or turn_right() method is called.

Finally, to determine if asteroids collide with the laser, another method can be created which takes in the asteroid's x-coordinate and y-coordinates as input and substitutes the equation to determine if the point is on the line. The method returns True if the asteroid's point is on the line and False otherwise.

This method can be iterated through a list containing all of the current asteroids, and the linear equation representing the line of the laser is updated with each frame.

Explaining the AI in the bot code

Firstly, I set the bot to check its immediate vicinity to determine if nearby asteroids can be reached through turning and firing and did not require movement. The vicinity is determined by either checking that the asteroid's Euclidean distance to the spaceship was less than or equal to 120 units and the angle of inclination (see Part 1 Q5) was less than or equal to 90 degrees, or the asteroid's Euclidean distance to the spaceship was less than or equal to 100 units and the angle was between 90 and 180 degrees.

If an asteroid is nearby, the bot will turn either left or right, depending on which side required less number of turns, and fire a bullet when the inclination angle was approximately 0.

If there are no asteroids nearby, the bot will scan the map for the nearest asteroid. It will turn in the direction of the asteroid, and move directly towards it, while constantly checking if the direction of movement was no longer aligned with the asteroid's position each frame. When the distance is under 120 units for large asteroids and 90 units for small asteroids, the spaceship will fire a bullet.

The bot uses data collected from the spaceship object to determine the position of the spaceship, which affects the calculation of the nearest asteroid's position to make the decision on which direction to turn, when the direction of the spaceship is the same direction as the closest asteroid, when the spaceship has moved close enough to the asteroid to fire the bullet, and when to fire another bullet depending on whether the asteroid was removed or the previous bullet had expired.

The list containing the asteroids is processed to find the asteroid which has the shortest distance to the spaceship, which is affected by whether the asteroid is a large or small asteroid because this also governs how far the spaceship must move before firing a bullet.

The list of bullets returns the position of each bullet and provides data on when the bullet has struck an asteroid or when the current bullet has expired, the fuel provides decision on whether the spaceship can fire a bullet, and the score provides details on the gameplay performance.

I would create a new list object to store all of the existing asteroids in the map in each frame. After appending all of the current asteroids, I would also append the first asteroid from the list of upcoming asteroids. I would then iterate through the list with the strategy explained in Q1 to locate the nearest asteroid.

If the upcoming asteroid is the closest asteroid to the spaceship, then I would locate the second closest asteroid and manoeuvre towards the asteroid. Once that asteroid is destroyed, the upcoming asteroid would appear on the map and the spaceship would immediately move towards it in the shortest path possible. This process would be repeated each frame so new upcoming asteroids would be added to the list created.