1. a. Tuples and lists are very similar, they both store items, and any item in either is accessible through commands like tuplelist[1], for example. The objects in a list can be changed, while the objects within a tuple cannot. An array is very similar to a list, except it can only store the same data type, where lists (and tuples) can store multiple data types.

b. A for loop and while loop are similar, because they both tell the program to perform an action until a criteria is met. The for loop is generally used in situations where a calculation (or some other function of the program) needs to be done for every iteration within an object such as a list or array. A while loop will continue to run until the while condition is met. A for loop is better for avoiding the infinite loop because it is generally used to run through an object and no more, while the while loop is running until a condition is met. If the code is improperly written to where the condition is never met, the loop will run “forever”.

c. The definition of object, in programming, is a collection of data and methods to act on that data. An object is an instance of a class. One example I remember is to envision “car” as a class, and then all the different types of cars, such as Ford Mustang, Lamborghini Diablo, etc. as the objects. Each object shares similarities, they all have engines, wheels, seats, doors, etc., but each car (object) has their own specific options, shapes, etc.

d. This is a tough question, haha. I think something that would make for a good python object is my change jar. It is about a quart in size and has a coin counter in the lid. The attributes that this jar has are things like its size, material, and weight, which would change as the jar is filled. Methods that the jar would have would be addcoin(), subtractcoin(), empty(), showcurrentvalue().

1. Integrating Problem

Code is in the folder, but to answer the questions, it is more accurate to sum from the lower end of the integral, to the higher end. As the numbers are summed and stored, the error is proportional to the numbers added. So starting at the larger number, starts with a larger error (which cannot get smaller), vs starting with a small number (therefore small error) and working upwards.

1. Rocket Problem

The answer I want to give is that yes, we should be able to use MC methods for this problem. Simply because we know how well MC can be used to solve integrals, even complicated integrals, including some that would otherwise take literally millions of years for a computer to calculate. So using the MC method of integration over complicated integrals, like you get in more advanced mechanical motion calculations, should theoretically allow you to calculate a numerical solution. Of course, we have to consider the error. MC methods are not perfect, no matter how good they are. Getting rockets into space needs fairly precise calculations, so MC methods may not be able to give you that precision.

1. For the Monty Hall problem, the player should always choose to switch to the other door when given the opportunity. The reasoning is simple. We can look at it through a couple steps. First off, it is important to know that the host of the show always knows which door ahs the money behind it. If he didn’t, he could never be sure that he would open a door with a goat behind it. Knowing that the host knows, the player picks a door. On this initial random choice, there is a 1/3 chance that the player has picked the money door. But to make the analogy clearer, we will say that there is a 2/3 chance that our player has chosen a goat door. The key difference here, when switching, is that if you have chosen a goat door, the host only has one option to open, the second goat door. Meaning that when you choose incorrectly on your initial pick (2/3 of the time), and you switch, you will always choose the winning door, compared to the 1/3 of the time that your initial pick was correct, making you now choose a non-money door. So when always switching, the odds of picking wrong first, and then switching to the winning door, is 2/3, and the odds of picking correctly first, and switching to the wrong door is only 1/3, vs. when the player stays put, and their odds of choosing the winning door are just 1/3.

Pseudo-code for MH problem:

We will be creating a sim of the Monty Hall problem, and running it a sufficient number of times to gauge the odds of winning when not switching vs always switching.

*Creating some variables and objects*:

Make a couple lists to track wins for each version of the game

Make a couple counting variables, i=1 and j =1

Make a while loop for i<some number large enough to give us a sufficient sample size

Create 3 door Boolean objects =False

Choose a winning door using randint, make doorN = True

Create a list with our doors

Choose a player door with randint

Test if players chosen door ==Doors[N]

If True, append list for first pick winner

Calculate the odds of winning

Make a while loop for j<large number

Follow the same pattern as above

But if player chose correct, change to a losing door, and vice versa, as we know the player cannot pick a second losing door. They will always win->lose or lose->win, and nothing else.

If player has won, append list for switching winner

Calculate the odds of winning when switching.