1. I need a list of planet coordinate dictionaries so I can compare each of their distances from Earth to the radio range. I need a function to loop through these coords and return True if at least one of them is within range. Then, I’ll loop through this process for however many trials there are. Finally I’ll find the average success rate over however many trials there are and append that figure to a list. Repeat for however many galaxy setups there are (10). I have a function checkPlanetInRange\_TF() to return a boolean that represent whether a single planet is in range, a function checkMultiplePlanetsInRange\_TF() that returns a boolean that’s True if at least one planet is in range, and placePlanetsReturnCoords() that returns random cartesian planet coords for checkMultiplePlanetsInRange\_TF() to work with. I had to be careful to one have only one set of planet coords initialized at a time. Before, I had billions of sets of coords committed to a list, but that broke my laptop. Right now, the setup of my program generates one set of coords, checks if it's within range, and discards it immediately. The booelan outputs of checkMultiplePlanetsInRange\_TF() are put in a list, where their hit rates can be averaged across trials and committed to another list which contains the probability of contact for every amount of lifeforms.
   1. **NOTICE: I moved two general\_functions methods to the main program because that’s how the project’s graded. They’re both documented in the main program. One finds the average success rate given a list of booleans and the other converts cylindrical coordinates to cartesian coordinates.**
2. Since the Milky Way is kind of hockey puck shaped, using cylindrical coordinates will give the least amount of headache to the human creating the bounds for planet placement. R can easily be bounded as 15,000 <= R <= 50,000 instead of the trigonometric nonsense I’d need to do to express it in cartesian coordinates. Now that the center of the galaxy has been deemed uninhabitable, theta and z are also both very easy to express. Theta is a full rotation around the galaxy or 0 <= theta <= 2 \* pi, and z is just -500 <= z <= 500. It should be noted that z=0 for me is a plane through the center of the galaxy. Now that R, theta, and z are properly bounded, planets can be randomly places by picking random values for R, theta, and z within their ranges. For a single planet R=r.uniform(15\_000, 50\_000), theta=r.uniform(0, 2 \* m.pi), and z=r.uniform(-500, 500). I convert these cylindrical coordinates to cartesian in an imported method from final\_project\_general\_functions.py convertCylindricalToCartesian(). This method takes a dictionary of cylindrical coords and uses the conversion formulas to spit out its cartesian form. I treat it like a black box in my program because I know it works.
3. Since square roots ae computationally intensive and I’ll be needing to find distance literally billions of times, it’s better to find the square of the distance between planets and compare that figure to 225 square light-years. Squaring and adding coordinates together is computationally simple and, most importantly, not recursive. So Python should be able to handle that relatively easily. And really, the square rooting is useless for my purposes since if m^2 > n^2, then m > n and vice versa for positive values of m and n.
4. All I need is a list of probabilities of contact. The list will hold N number of values between 0 and 1 where N is the number of increments I test between 100M and 1,000M lifeforms. In the case of my program N is 10 so the length of the list is 10. Then I’ll feed that list into my handy-dandy graphing module (see next question).
5. I use matplotlib.pyplot methods to show my results. It’s very easy to use because the methods are all very intuitive. All .plot() needs is two lists for x and y values. A list for y-values already exists as probabilities\_list and for a list of x-values, I can just use list(range(100, 1\_000, 100)) to represent the number of lifeforms in millions. I can customize things like titles and axis labels with .title() and .xlabel() and .ylabel(). And MATPLOTLib allows you to save a png of your graph after your program has run, which is what I’ll turn in.