Wyedean School and Sixth Form Centre

Computational challenge 2023

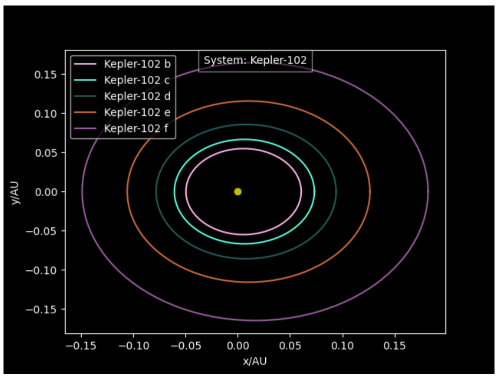
Solar systems.

Written and coded by Owyn Fennell and kingsley brown
Computational challenge 2023

Coding platform and libraries used:

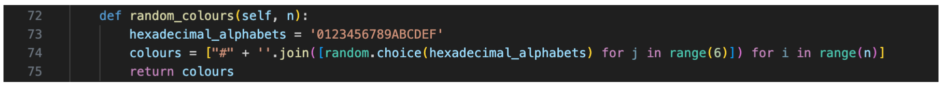
Programming language:
We used python as our programming language of choice. It is a high level general coding platform.
We used python because it's a very intuitive language and can have many modules installed on it. This is useful as it can make tasks easier and help us plot our data.

Libraries used:
Matplotlib:
Matplotlib is a python library that allows you to plot graphs in python. It also allows you to add 3rd party packages to it like which allows you to animate a plot that you have made on matplot. This all makes matplot very versatile and a must have for a project like this on python or any high level python graphing project that one is doing.



NumPy:
NumPy provides a lot of the maths needed for this project. It provides the basis for arrays used in the coding as numpy arrays are densely packed in memory and have a homogenous type so processing them is faster. It helps with the maths surrounding vectors and provides trigonometric functions and pi which is used for generating angles . Numpy also provides the linspace function which is useful for generating a range of values such as times or angles to loop through.

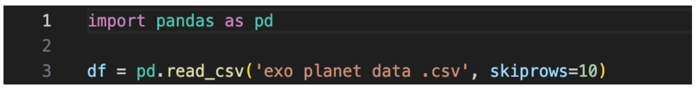
Random:
random is a library for generating random numbers; it's used for getting random colours in the case of our code.



```
def random_colours(self, n):
    hexadecimal_alphabets = '0123456789ABCDEF'
    colours = ["#" + ''.join([random.choice(hexadecimal_alphabets) for j in range(6)]) for i in range(n)]
    return colours
```

Data:
 data is a file that uses pandas to get data from csv files like our large data set.

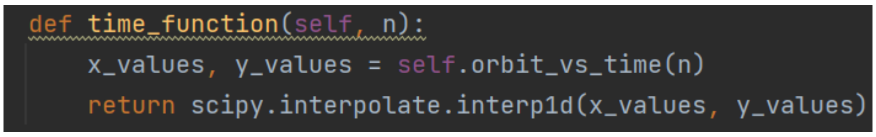
Pandas:
Allows you to go through and do analysis and manipulation on large data sets. It can be used to pull data from the large dataset.



```
import pandas as pd

df = pd.read_csv('exo planet data .csv', skiprows=10)
```

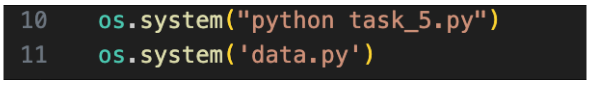
SciPy:

Scipy is a module that allows integration and many other difficult maths problems. Here it is used in task 5 to interpolate the function that gives orbital polar angle as output and time as input so we have a the angle as a function of time.



```python
def time_function(self, n):
    x_values, y_values = self.orbit_vs_time(n)
    return scipy.interpolate.interp1d(x_values, y_values)
```

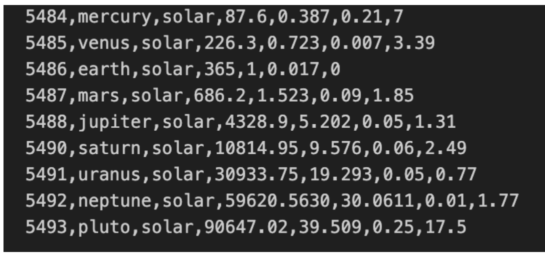
Os:
Os is a module that allows you to search through the operating system of the computer to search for other files and folders of code. It was used to get task 5 and the data set for use in the code.



```python
os.system("python task_5.py")
os.system('data.py')
```

Prerequisite code:
For the tasks to run we had 3 pieces of code that was exported in to the files and used.

Large data set:
The large data set was a csv file with nearly 5500 planets in it. This was obtained from downloading it from the caltech nasa exoplanet archives. We downloaded 6 different pieces of data for each planet(as shown in the pic to the right). Planet name, host name, orbital period in days, orbital semi major axis in au, eccentricity and β in degrees (tilt of the orbit). For this data set to be used we needed to add our solar system at the bottom, and also make sure each planet set we selected only had one star in the system (the data set still contains a lot of planet systems with more than one star so should be cross referenced with the nasa exoplanet data set). so as not to make a false orbit as we did not code for the effects 2 stars gravity would have on the system. To select data from this you could write the planet id which is a number given before the list of data due to issues obtained with the data module one would have to be subtracted from the id on the csv to make it run in the code (and example of this is pluto it has the id 5493 but to be coded for needs the id 5492). Some tasks like task 6 are easier and do not require the id on the name of the system of the planets.

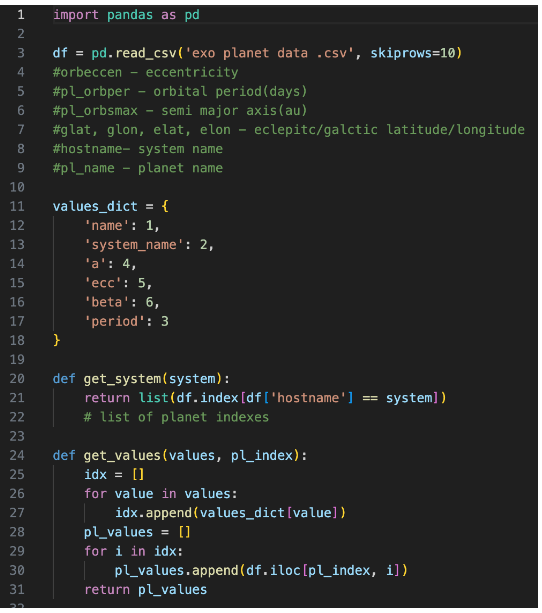


```
5484,mercury,solar,87.6,0.387,0.21,7
5485,venus,solar,226.3,0.723,0.007,3.39
5486,earth,solar,365,1,0.017,0
5487,mars,solar,686.2,1.523,0.09,1.85
5488,jupiter,solar,4328.9,5.202,0.05,1.31
5490,saturn,solar,10814.95,9.576,0.06,2.49
5491,uranus,solar,30933.75,19.293,0.05,0.77
5492,neptune,solar,59620.5630,30.0611,0.01,1.77
5493,pluto,solar,90647.02,39.509,0.25,17.5
```

Exo planets we used:

system name	lower bound	upper bound	id in data set
61 virginis	33	36	61 vir
dmpp-1	122	126	DMPP-1
hd-10180	459	465	HD 10180
7 canis majoris	36	38	7 CMa
24 sextanis	9	11	24 sex
47 ursae majoris	22	25	47 UMa

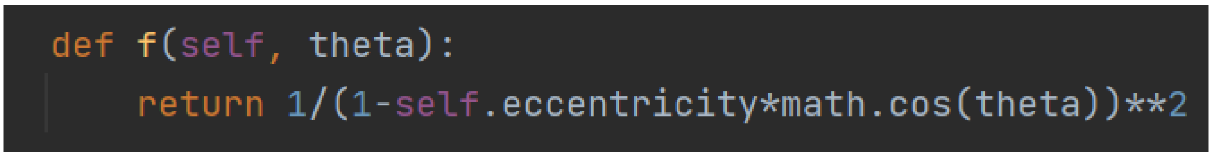
au microscopi	16	48	AU Mic
corot-7	113	116	CoRoT-7
corot-24	98	100	CoRoT-24
epic 249893012	170	173	EPIC 24989301
inner solar	5483	5487	solar
solar	5483	5492	solar
hd-164922	707	711	HD 164922
hd 219134 inner	882	887	HD 219134
hd-219134	882	888	HD 219134
k2-138	1333	1339	K2-138
kepler 102	1947	1951	Kepler-102



```python
import pandas as pd

df = pd.read_csv('exo planet data .csv', skiprows=10)
#orbeccen - eccentricity
#pl_orbper - orbital period(days)
#pl_orbsmax - semi major axis(au)
#glat, glon, elat, elon - eclepitc/galctic latitude/longitude
#hostname- system name
#pl_name - planet name

values_dict = {
    'name': 1,
    'system_name': 2,
    'a': 4,
    'ecc': 5,
    'beta': 6,
    'period': 3
}

def get_system(system):
    return list(df.index[df['hostname'] == system])
    # list of planet indexes

def get_values(values, pl_index):
    idx = []
    for value in values:
        idx.append(values_dict[value])
    pl_values = []
    for i in idx:
        pl_values.append(df.iloc[pl_index, i])
    return pl_values
```

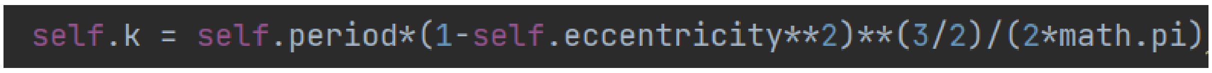
Data.py:
Data.py was the name for the python code that was used to read the csv. This code reads the csv through the use of pandas. a dictionary was used to associate each value used with a column index and a get values function and get system function to make it simpler to use in the rest of the code

Task5:
Task 5 is some code that was used for tasks 3, 4, 6 and 7. Kepler's 2nd law gives the time elapsed during an orbit as a function of the angle of rotation of the orbital body which in the code is expressed by the integral of this function from 0 to the angle of rotation(theta):



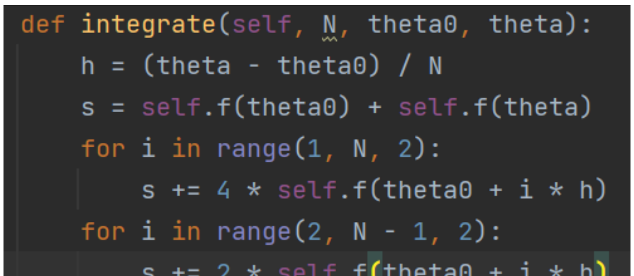
```python
def f(self, theta):
    return 1/(1-self.eccentricity*math.cos(theta))**2
```

Which is then multiplied by this constant:



```python
self.k = self.period*(1-self.eccentricity**2)**(3/2)/(2*math.pi)
```

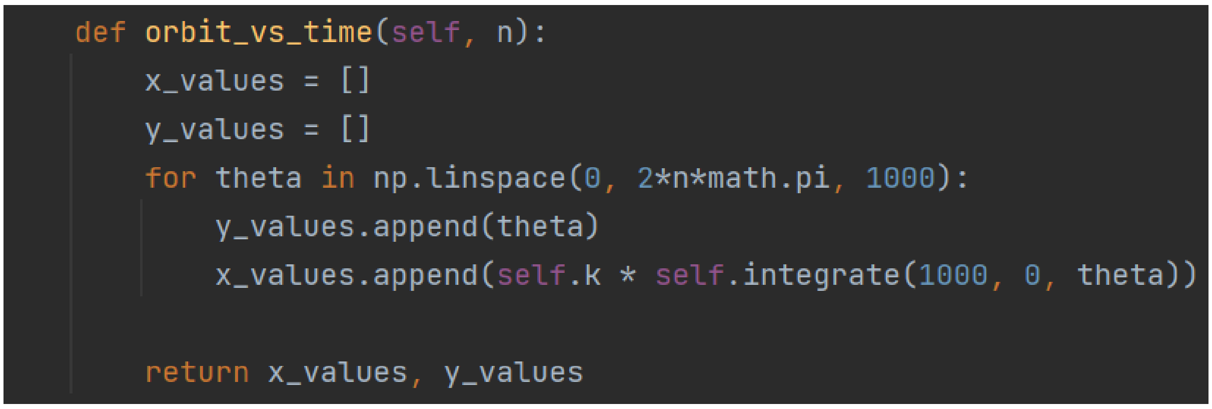
The integrand is evaluated here using Simpson's ⅓ rule, where s is the sum of the function evaluated at N



```python
def integrate(self, N, theta0, theta):
    h = (theta - theta0) / N
    s = self.f(theta0) + self.f(theta)
    for i in range(1, N, 2):
        s += 4 * self.f(theta0 + i * h)
    for i in range(2, N - 1, 2):
        s += 2 * self.f(theta0 + i * h)
```

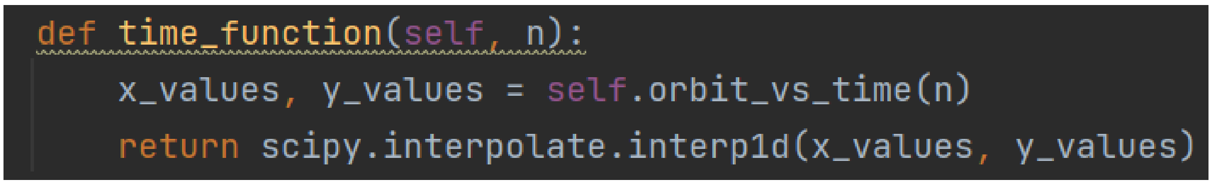
subintervals. The first for loop generates every odd subinterval(excluding the first and final subinterval) to be multiplied by 4 where the input to the integrand at each subinterval increases by the number of the subinterval to be evaluated(i) multiplied by the gap between subintervals(h). The second for loop serves the same purpose but for the even subintervals which are then multiplied by 2. Finally, the sum of all the sub intervals multiplied by h/3 is returned.

This function calculates the angles and times for a range of values up to n orbits using the linspace function so that 1000 angles are calculated between 0 and 2*pi*n which is when the nth orbit is completed.



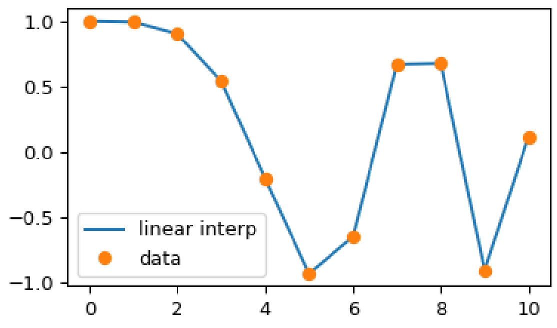
```python
def orbit_vs_time(self, n):
    x_values = []
    y_values = []
    for theta in np.linspace(0, 2*n*math.pi, 1000):
        y_values.append(theta)
        x_values.append(self.k * self.integrate(1000, 0, theta))

    return x_values, y_values
```

Using the angle and time values calculated above the Scipy module is used to interpolate those values and return the angle as a function of time.



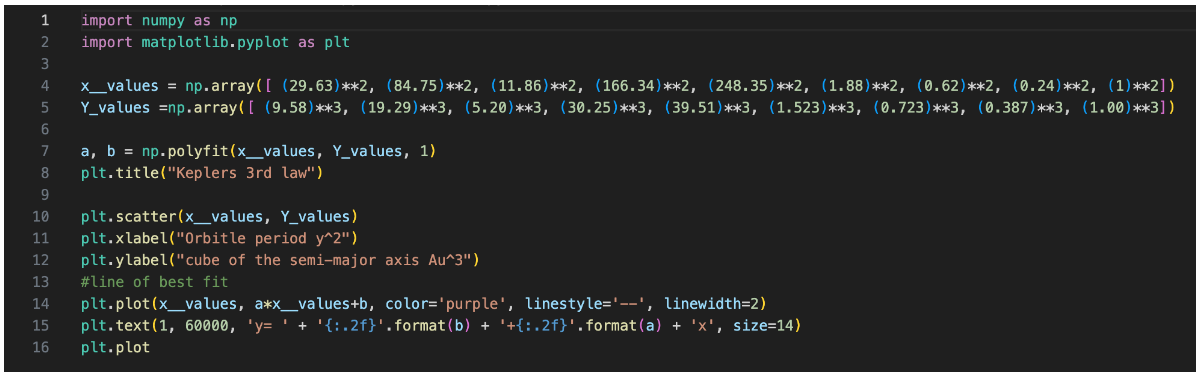
```python
def time_function(self, n):
    x_values, y_values = self.orbit_vs_time(n)
    return scipy.interpolate.interp1d(x_values, y_values)
```

The interp1d function works simply by connecting the data points linearly, as shown by the diagram. Therefore, 1000 data points are calculated to ensure accuracy.



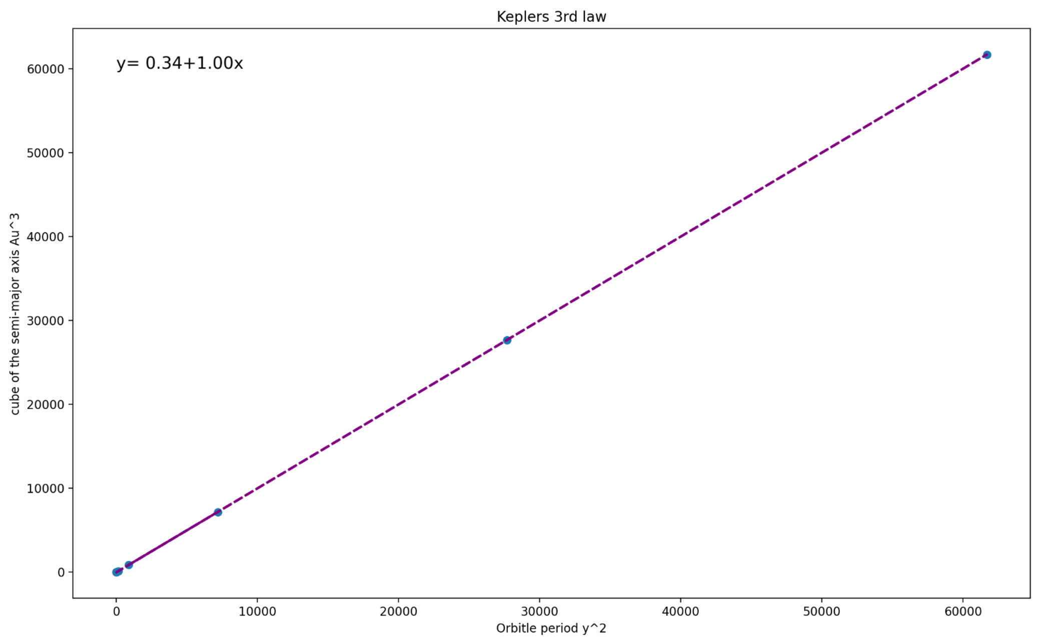
Tasks:

Task 1:



```
1   import numpy as np
2   import matplotlib.pyplot as plt
3
4   x__values = np.array([ (29.63)**2, (84.75)**2, (11.86)**2, (166.34)**2, (248.35)**2, (1.88)**2, (0.62)**2, (0.24)**2, (1)**2])
5   Y_values =np.array([ (9.58)**3, (19.29)**3, (5.20)**3, (30.25)**3, (39.51)**3, (1.523)**3, (0.723)**3, (0.387)**3, (1.00)**3])
6
7   a, b = np.polyfit(x__values, Y_values, 1)
8   plt.title("Keplers 3rd law")
9
10  plt.scatter(x__values, Y_values)
11  plt.xlabel("Orbitle period y^2")
12  plt.ylabel("cube of the semi-major axis Au^3")
13  #line of best fit
14  plt.plot(x__values, a*x__values+b, color='purple', linestyle='--', linewidth=2)
15  plt.text(1, 60000, 'y= ' + '{:.2f}'.format(b) + '+{:.2f}'.format(a) + 'x', size=14)
16  plt.plot
```

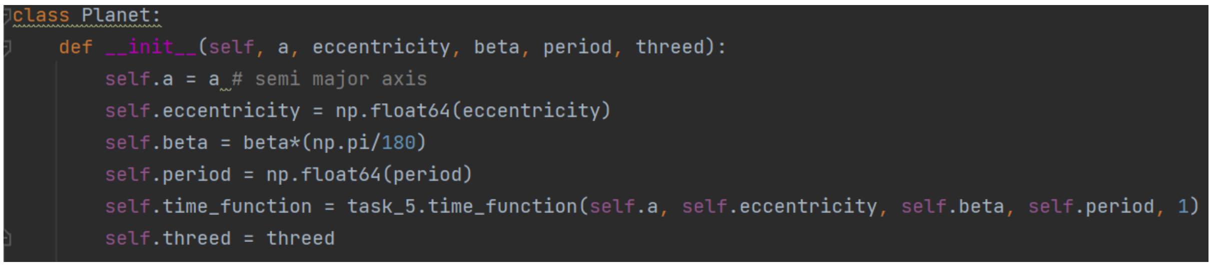
This is task 1 in which we plotted the square of the orbital period against the cube of the semi major axis. The code for this task was purposely made basic to show what we were doing for new coders, therefore we did not pull any data from other finals to make this code more visual.
In this code we imported numPy for the maths and matplotlib to plot the very basic linear line.
In lines 4 and 5 an array was made using NumPy. In this array we included all of the data for our solar system to be plotted later in the x and y axis.
In line 7 we polyfited the x and y axis to get values a and b ; these are the gradient and the y intercept of the line.
In lines 8-12 we gave the graph basic properties like a name and axis titles. The plots were also plotted using plt.scatter we did this so it would plot point not a line allowing us to plot a line of best fit over it.
In line 14 we converted our a and b values into a plot of the line of best fit.
In line 15 we put text for the line of best fit and put the placement it appears on the graph.



Task 2, 3 and 4:

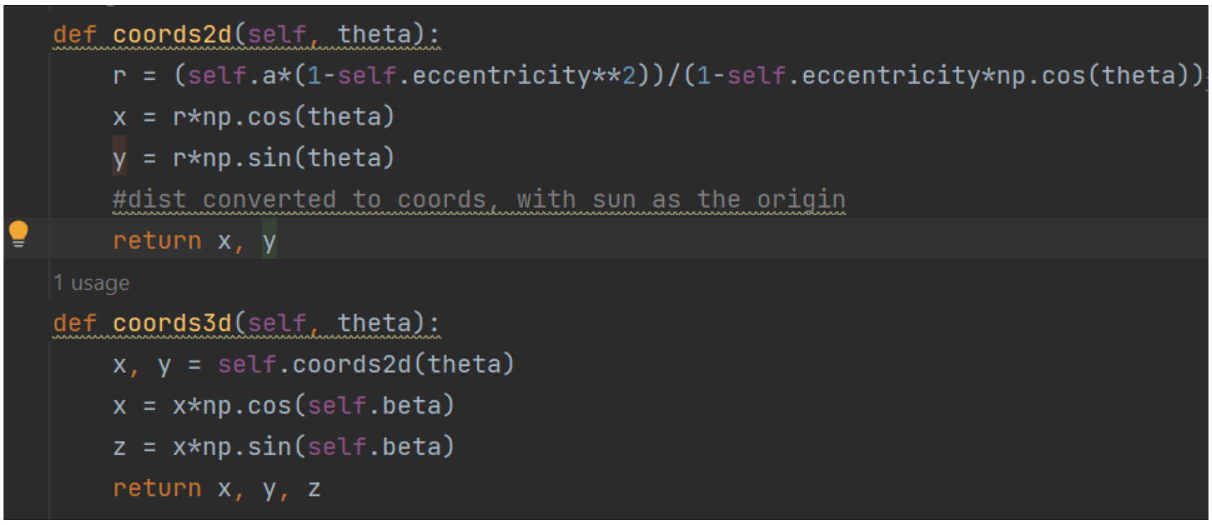
These tasks were done on the same python file as much of the code is reused.

There is a planet class with attributes: semi-major axis, eccentricity, inclination angle beta, orbital period, and whether the planet is in 3D or not. The time function is calculated by the task_5 python file and returned as one of the attributes.



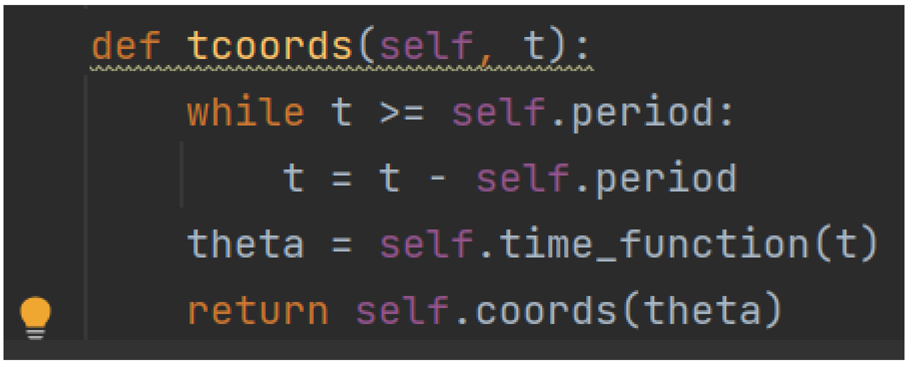
```python
class Planet:
    def __init__(self, a, eccentricity, beta, period, threed):
        self.a = a # semi major axis
        self.eccentricity = np.float64(eccentricity)
        self.beta = beta*(np.pi/180)
        self.period = np.float64(period)
        self.time_function = task_5.time_function(self.a, self.eccentricity, self.beta, self.period, 1)
        self.threed = threed
```

The class is useful as it stores functions needed for all the planets such as getting the coordinates of the planet at a given angle:



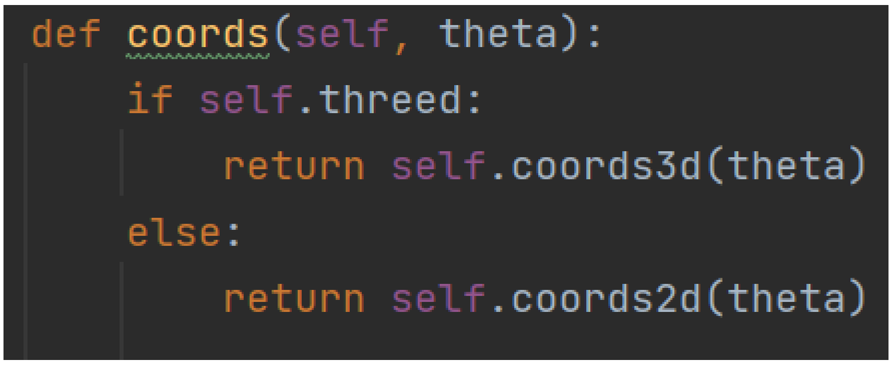
```python
def coords2d(self, theta):
    r = (self.a*(1-self.eccentricity**2))/(1-self.eccentricity*np.cos(theta))
    x = r*np.cos(theta)
    y = r*np.sin(theta)
    #dist converted to coords, with sun as the origin
    return x, y
1 usage
def coords3d(self, theta):
    x, y = self.coords2d(theta)
    x = x*np.cos(self.beta)
    z = x*np.sin(self.beta)
    return x, y, z
```

For the animation tasks, task 5 is used to get the orbital polar angle at a given time:



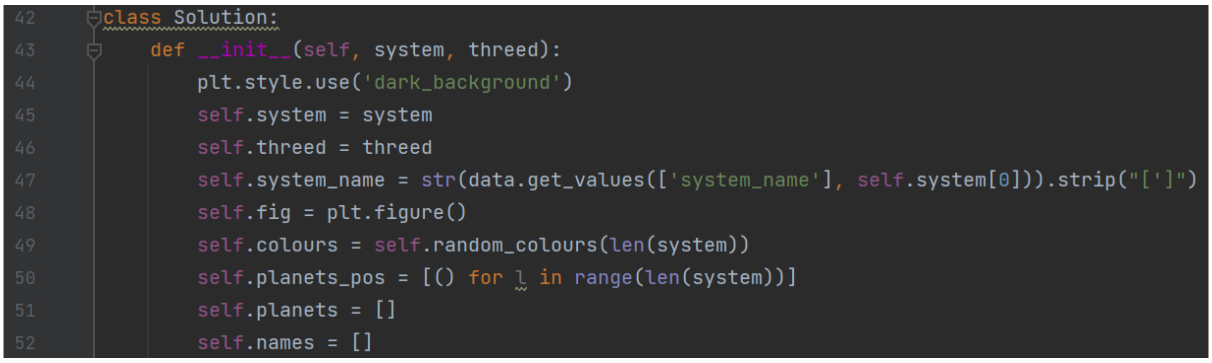
```python
def tcoords(self, t):
    while t >= self.period:
        t = t - self.period
    theta = self.time_function(t)
    return self.coords(theta)
```

To reduce the number of calculations done in task 5, the function is only calculated for one orbit and the while loop means the time given as input starts from 0 once an orbit is completed as the orbital velocities are the same for each orbit. The coords function gives either 2d or 3d coordinates based on the 3d attribute.



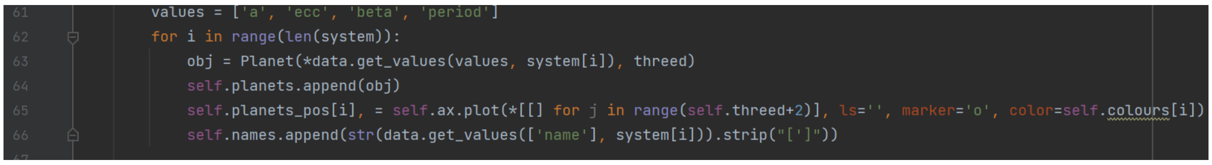
```python
def coords(self, theta):
    if self.threed:
        return self.coords3d(theta)
    else:
        return self.coords2d(theta)
```

The solution is also stored as a class so can be created with any system in the dataset as long as a list of indices for the planets is given as one of the parameters(system).



```python
class Solution:
    def __init__(self, system, threed):
        plt.style.use('dark_background')
        self.system = system
        self.threed = threed
        self.system_name = str(data.get_values(['system_name'], self.system[0])).strip("[']")
        self.fig = plt.figure()
        self.colours = self.random_colours(len(system))
        self.planets_pos = [() for i in range(len(system))]
        self.planets = []
        self.names = []
```

The planets attribute is a list of planet objects so that throughout the code they can be iterated on to make plots. The names attribute is a list of the names of the planets used for creating the legend. These attributes are created here:

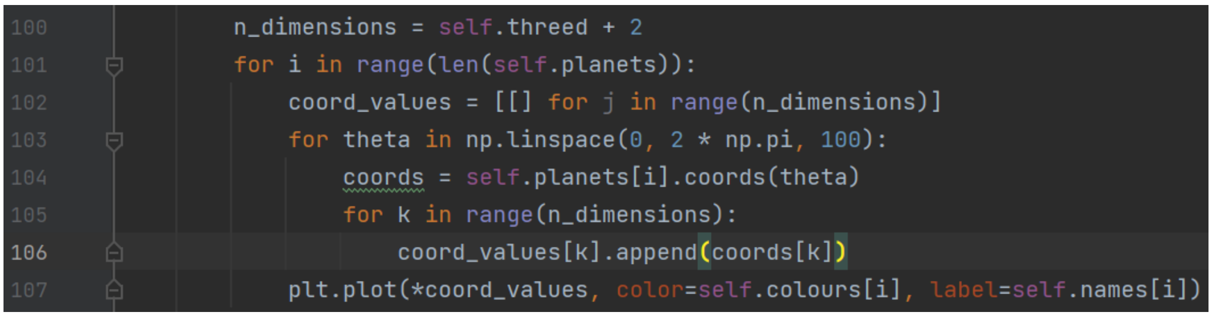


```python
        values = ['a', 'ecc', 'beta', 'period']
        for i in range(len(system)):
            obj = Planet(*data.get_values(values, system[i]), threed)
            self.planets.append(obj)
            self.planets_pos[i], = self.ax.plot(*[[] for j in range(self.threed+2)], ls='', marker='o', color=self.colours[i])
            self.names.append(str(data.get_values(['name'], system[i])).strip("[']"))
```

The planets_pos attribute is also created here and is a list of plots for each planet and is initially empty.

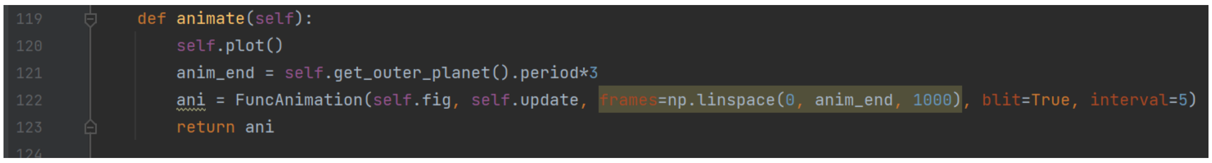
<u>Plotting:</u>

As the program works for 2d and 3d plots the number of dimensions is unknown so is calculated. The coord_values variable is a list that stores lists for the x data and y data and z data if the plot is in 3d. The coordinates for each are then calculated for a range of values of orbital polar angle and put in the coord_values list in lines 105-106. The coordinates are then re-calculated and plotted for each of the planets.



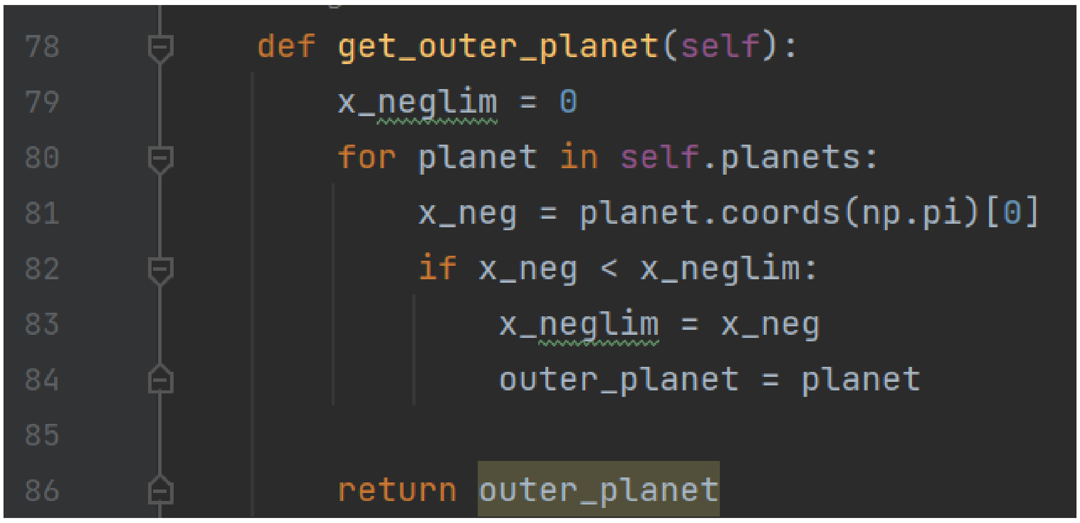
```python
        n_dimensions = self.threed + 2
        for i in range(len(self.planets)):
            coord_values = [[] for j in range(n_dimensions)]
            for theta in np.linspace(0, 2 * np.pi, 100):
                coords = self.planets[i].coords(theta)
                for k in range(n_dimensions):
                    coord_values[k].append(coords[k])
            plt.plot(*coord_values, color=self.colours[i], label=self.names[i])
```

<u>Animating:</u>
The animations were created using Matplotlibs animation library.



```python
119     def animate(self):
120         self.plot()
121         anim_end = self.get_outer_planet().period*3
122         ani = FuncAnimation(self.fig, self.update, frames=np.linspace(0, anim_end, 1000), blit=True, interval=5)
123         return ani
124
```

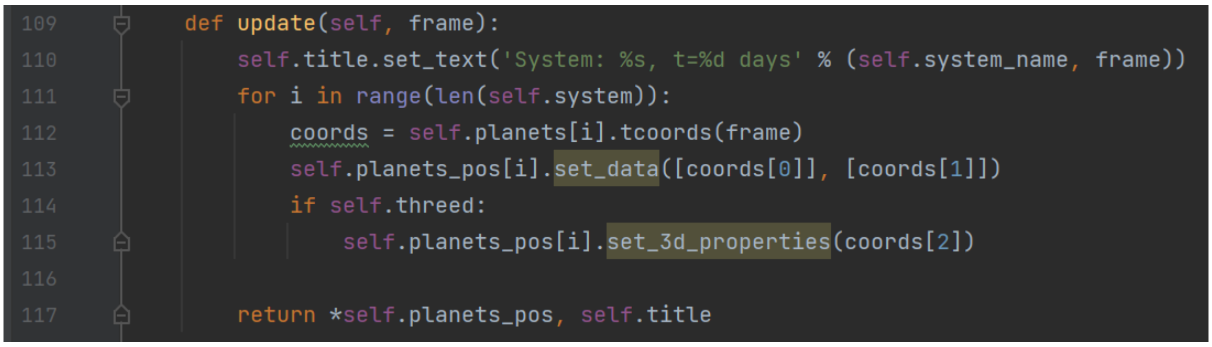
The background plots are created using the plot() function described previously. In line 121, the animation is set to end after three of the outer-planets orbits are completed. It is necessary to calculate the end of the animation this way as otherwise the animations for each system would run at different speeds and end at different points. The outer planet is found using this function:



```python
    def get_outer_planet(self):
        x_neglim = 0
        for planet in self.planets:
            x_neg = planet.coords(np.pi)[0]
            if x_neg < x_neglim:
                x_neglim = x_neg
                outer_planet = planet

        return outer_planet
```

This loops through the list of planets so that in each iteration the outer planet is the planet which so far has the furthest coordinates from the centre at a polar angle of π radians. Therefore, after the loop has gone through all of the planets the planet returned will be the outer planet.

Finally the animation is created using Matplotlibs FuncAnimation which uses blitting meaning that the background of the animation is only drawn once and only a sequence of plots(planets_pos) as well as the title is redrawn each frame which makes rendering the animation faster. This requires using an update function:

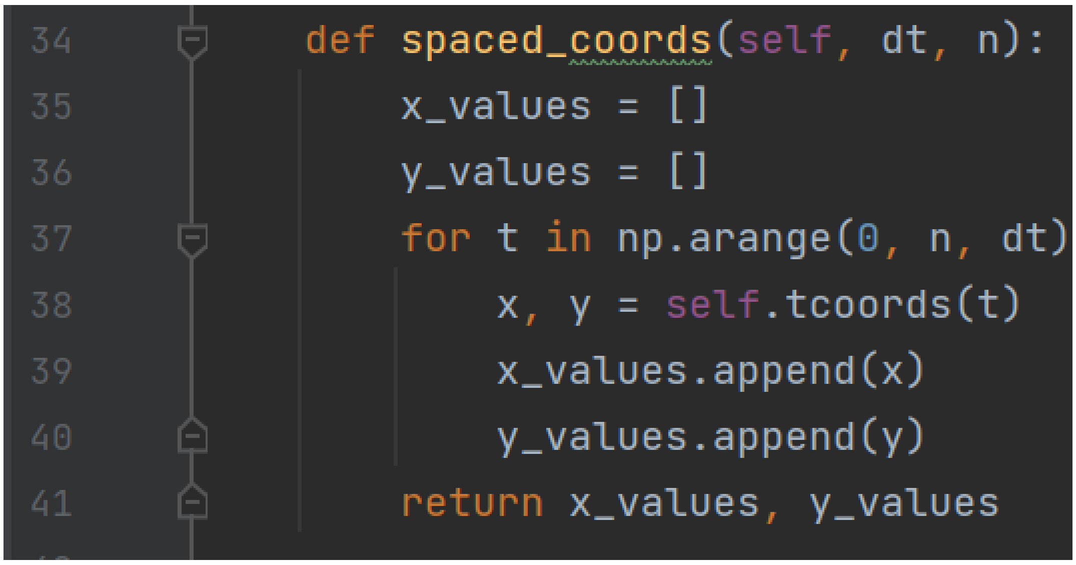


```python
    def update(self, frame):
        self.title.set_text('System: %s, t=%d days' % (self.system_name, frame))
        for i in range(len(self.system)):
            coords = self.planets[i].tcoords(frame)
            self.planets_pos[i].set_data([coords[0]], [coords[1]])
            if self.thread:
                self.planets_pos[i].set_3d_properties(coords[2])

        return *self.planets_pos, self.title
```

The frame variable is generated in line 122 using the frames parameter which creates an array of times to iterate through and pass a new item in the array to the update function each time it is run. The title is updated to show the time elapsed during the orbits and the coordinates for the planets_pos is updated with the tcoords function which uses task 5. The planets_pos and title must then be returned to FuncAnimation to indicate what must be redrawn each frame.

<u>Task 6:</u>

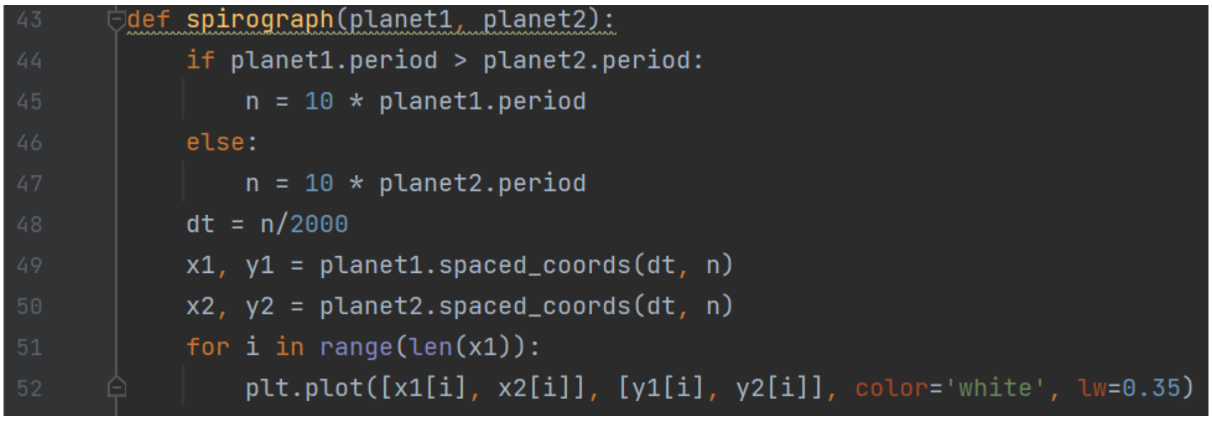
This program uses the same Planet class as tasks 2,3, and 4 with an additional function called spirograph:



```python
    def spaced_coords(self, dt, n):
        x_values = []
        y_values = []
        for t in np.arange(0, n, dt)
            x, y = self.tcoords(t)
            x_values.append(x)
            y_values.append(y)
        return x_values, y_values
```

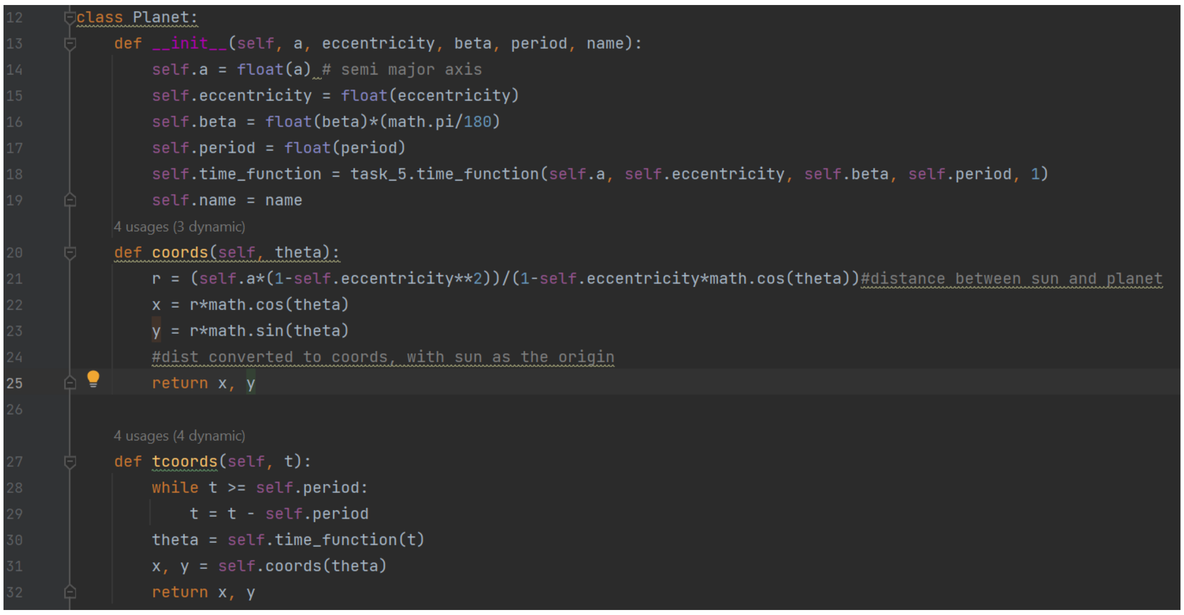
The dt parameter represents the change in time between when each line is plotted and n represents the time when the function stops plotting data. The x_values and y_values represent the list of coordinates of the planet for each time step dt.

The spirograph is then created here and is set to end after 10 of the outer planet's orbits in lines 44-47. The time step is calculated so that 2000 lines are drawn. The data is then calculated using the spaced_coords function in lines 49-50. Finally, in lines 51-52 the data is plotted such that each datapoint for one planet is connected as a line to the datapoint of the other planet at the same time.



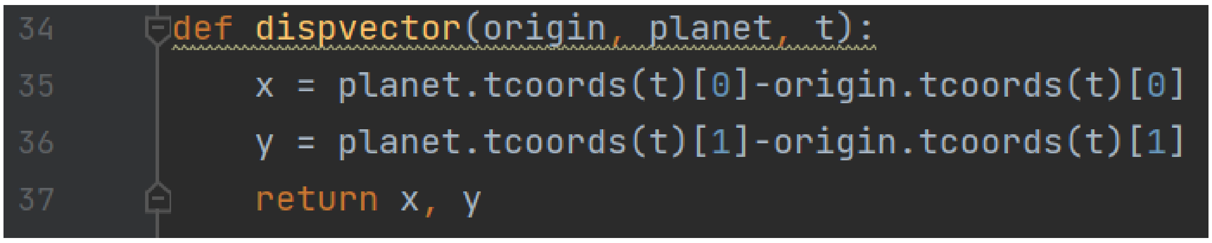
```python
def spirograph(planet1, planet2):
    if planet1.period > planet2.period:
        n = 10 * planet1.period
    else:
        n = 10 * planet2.period
    dt = n/2000
    x1, y1 = planet1.spaced_coords(dt, n)
    x2, y2 = planet2.spaced_coords(dt, n)
    for i in range(len(x1)):
        plt.plot([x1[i], x2[i]], [y1[i], y2[i]], color='white', lw=0.35)
```

Task 7:
This also uses a Planet class similar to the one in tasks 2, 3, and 4.



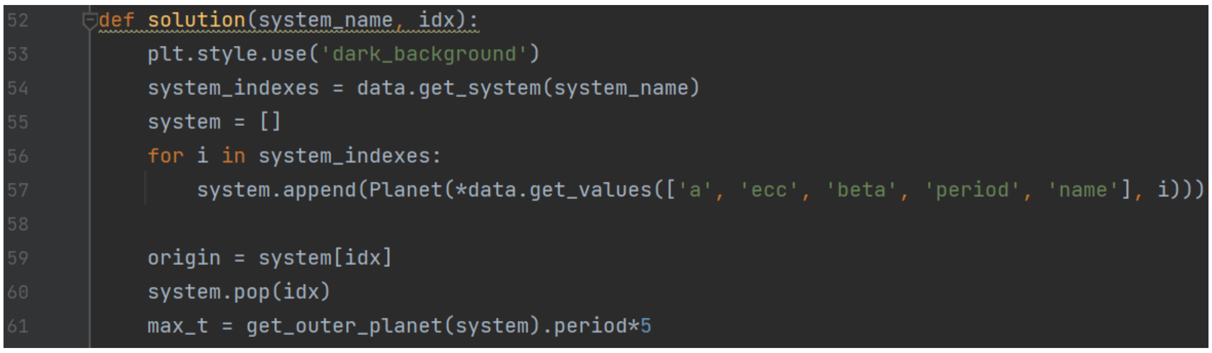
```python
class Planet:
    def __init__(self, a, eccentricity, beta, period, name):
        self.a = float(a) # semi major axis
        self.eccentricity = float(eccentricity)
        self.beta = float(beta)*(math.pi/180)
        self.period = float(period)
        self.time_function = task_5.time_function(self.a, self.eccentricity, self.beta, self.period, 1)
        self.name = name
    4 usages (3 dynamic)
    def coords(self, theta):
        r = (self.a*(1-self.eccentricity**2))/(1-self.eccentricity*math.cos(theta))#distance between sun and planet
        x = r*math.cos(theta)
        y = r*math.sin(theta)
        #dist converted to coords, with sun as the origin
        return x, y

    4 usages (4 dynamic)
    def tcoords(self, t):
        while t >= self.period:
            t = t - self.period
        theta = self.time_function(t)
        x, y = self.coords(theta)
        return x, y
```

A function is used to get the displacement vector from one planet to another at a given time(t).



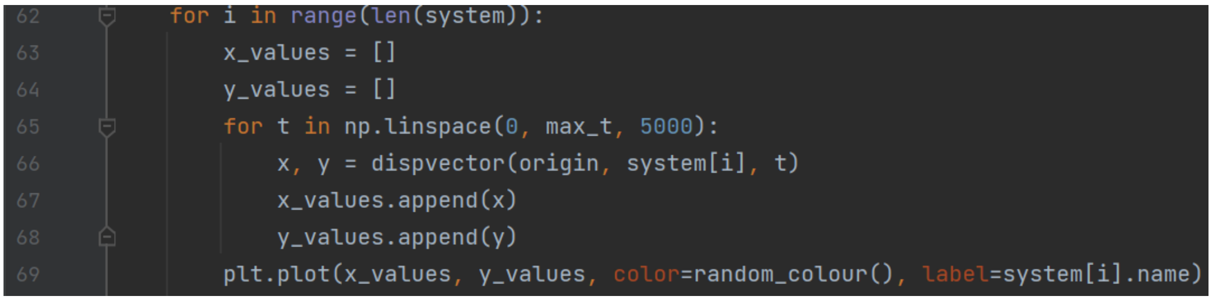
```python
def dispvector(origin, planet, t):
    x = planet.tcoords(t)[0]-origin.tcoords(t)[0]
    y = planet.tcoords(t)[1]-origin.tcoords(t)[1]
    return x, y
```

The solution takes the name of the system to be plotted as a parameter and the index of the planet chosen as the origin within the list of planets created in lines 56-57. Once the origin is assigned that planet is then removed from the list in line 59. The program is set to stop plotting after 5 of the outer planets orbits in line 61.



```python
def solution(system_name, idx):
    plt.style.use('dark_background')
    system_indexes = data.get_system(system_name)
    system = []
    for i in system_indexes:
        system.append(Planet(*data.get_values(['a', 'ecc', 'beta', 'period', 'name'], i)))

    origin = system[idx]
    system.pop(idx)
    max_t = get_outer_planet(system).period*5
```

A for loop then gets the indexes of the planets within the system list and gets the displacement vectors from the planet at each of these indexes over a range of time and plots these values.



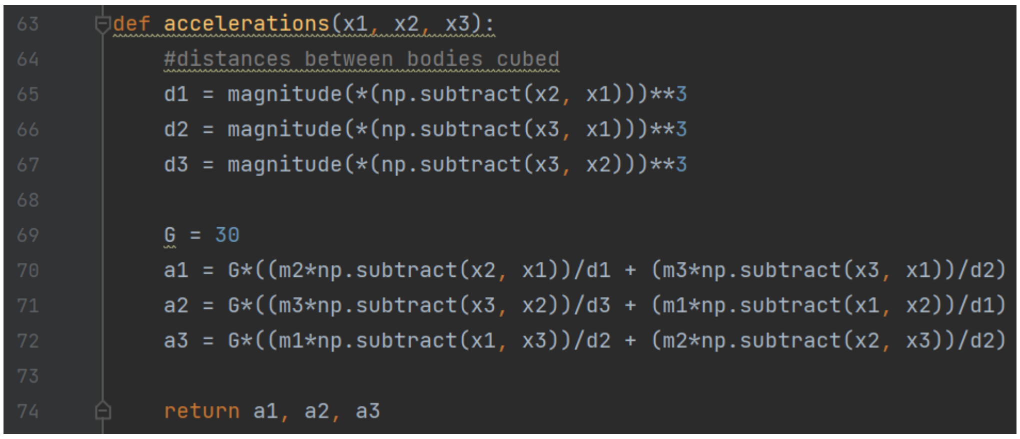
```python
    for i in range(len(system)):
        x_values = []
        y_values = []
        for t in np.linspace(0, max_t, 5000):
            x, y = dispvector(origin, system[i], t)
            x_values.append(x)
            y_values.append(y)
        plt.plot(x_values, y_values, color=random_colour(), label=system[i].name)
```

<u>Extension- three body problem:</u>
The three body problem is the problem of finding the position of three bodies orbiting each other in three-dimensional space as a function of time. The orbiting bodies obey Newton's laws of gravitation:

$$\begin{aligned}
\ddot{\mathbf{r}}_1 &= -Gm_2\frac{\mathbf{r}_1-\mathbf{r}_2}{|\mathbf{r}_1-\mathbf{r}_2|^3} - Gm_3\frac{\mathbf{r}_1-\mathbf{r}_3}{|\mathbf{r}_1-\mathbf{r}_3|^3},\\
\ddot{\mathbf{r}}_2 &= -Gm_3\frac{\mathbf{r}_2-\mathbf{r}_3}{|\mathbf{r}_2-\mathbf{r}_3|^3} - Gm_1\frac{\mathbf{r}_2-\mathbf{r}_1}{|\mathbf{r}_2-\mathbf{r}_1|^3},\\
\ddot{\mathbf{r}}_3 &= -Gm_1\frac{\mathbf{r}_3-\mathbf{r}_1}{|\mathbf{r}_3-\mathbf{r}_1|^3} - Gm_2\frac{\mathbf{r}_3-\mathbf{r}_2}{|\mathbf{r}_3-\mathbf{r}_2|^3}.
\end{aligned}$$

Here r1, r2, and r3 are the three-dimensional position vectors of three bodies and the second derivatives of the positions represent velocity. The constants m1, m2 and m3 are the masses of the bodies and G is the gravitational constant. A closed-form solution to this problem does not exist meaning that it is impossible to form a function that gives the positions of these bodies at a given time. It is also chaotic meaning that slight changes to the starting positions and velocities can cause very different orbits.

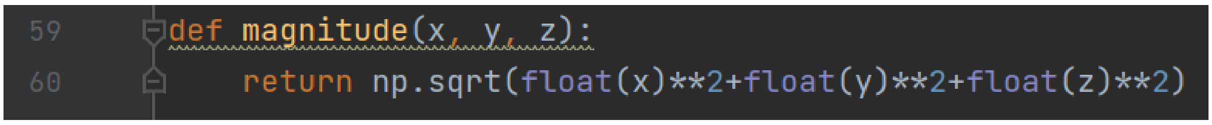
In this project it is solved using numerical approximations. The equations of motion are represented in the code as this function which has position vectors x1, x2 and x3:



```python
def accelerations(x1, x2, x3):
    #distances between bodies cubed
    d1 = magnitude(*(np.subtract(x2, x1)))**3
    d2 = magnitude(*(np.subtract(x3, x1)))**3
    d3 = magnitude(*(np.subtract(x3, x2)))**3

    G = 30
    a1 = G*((m2*np.subtract(x2, x1))/d1 + (m3*np.subtract(x3, x1))/d2)
    a2 = G*((m3*np.subtract(x3, x2))/d3 + (m1*np.subtract(x1, x2))/d1)
    a3 = G*((m1*np.subtract(x1, x3))/d2 + (m2*np.subtract(x2, x3))/d2)

    return a1, a2, a3
```

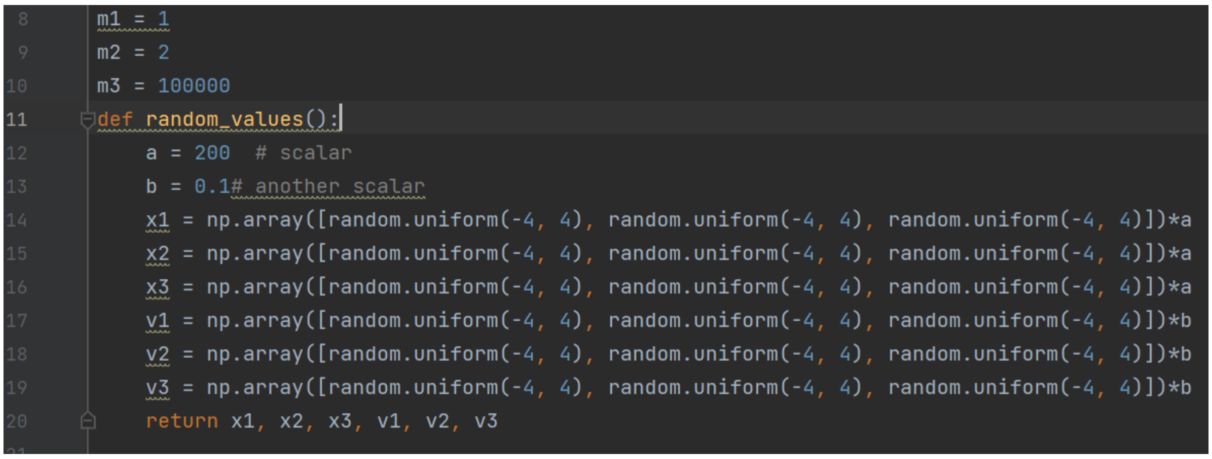
The numpy library is used to subtract these vectors and the vectors are represented as numpy arrays as opposed to python's lists for faster processing speed.

This function is used to find the distance between two bodies which is used in finding the accelerations.



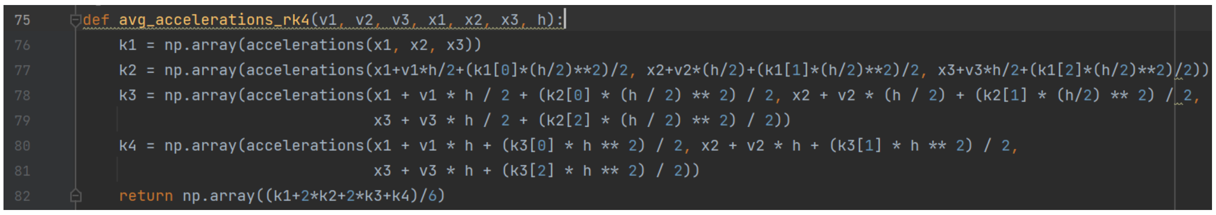
```python
def magnitude(x, y, z):
    return np.sqrt(float(x)**2+float(y)**2+float(z)**2)
```

The masses of the three bodies and the starting positions and velocities are either determined randomly or chosen.



```python
m1 = 1
m2 = 2
m3 = 100000
def random_values():
    a = 200   # scalar
    b = 0.1# another scalar
    x1 = np.array([random.uniform(-4, 4), random.uniform(-4, 4), random.uniform(-4, 4)])*a
    x2 = np.array([random.uniform(-4, 4), random.uniform(-4, 4), random.uniform(-4, 4)])*a
    x3 = np.array([random.uniform(-4, 4), random.uniform(-4, 4), random.uniform(-4, 4)])*a
    v1 = np.array([random.uniform(-4, 4), random.uniform(-4, 4), random.uniform(-4, 4)])*b
    v2 = np.array([random.uniform(-4, 4), random.uniform(-4, 4), random.uniform(-4, 4)])*b
    v3 = np.array([random.uniform(-4, 4), random.uniform(-4, 4), random.uniform(-4, 4)])*b
    return x1, x2, x3, v1, v2, v3
```

To update the velocities and positions after a time step h, the Runge-Kutta method is used.



```python
def avg_accelerations_rk4(v1, v2, v3, x1, x2, x3, h):
    k1 = np.array(accelerations(x1, x2, x3))
    k2 = np.array(accelerations(x1+v1*h/2+(k1[0]*(h/2)**2)/2, x2+v2*(h/2)+(k1[1]*(h/2)**2)/2, x3+v3*h/2+(k1[2]*(h/2)**2)/2))
    k3 = np.array(accelerations(x1 + v1 * h / 2 + (k2[0] * (h / 2) ** 2) / 2, x2 + v2 * (h / 2) + (k2[1] * (h/2) ** 2) / 2,
                                x3 + v3 * h / 2 + (k2[2] * (h / 2) ** 2) / 2))
    k4 = np.array(accelerations(x1 + v1 * h + (k3[0] * h ** 2) / 2, x2 + v2 * h + (k3[1] * h ** 2) / 2,
                                x3 + v3 * h + (k3[2] * h ** 2) / 2))
    return np.array((k1+2*k2+2*k3+k4)/6)
```

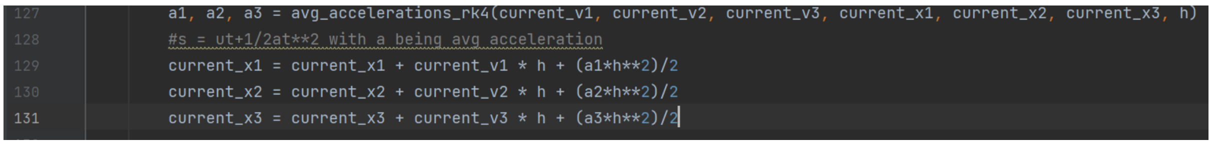
This uses the acceleration formulae to find 4 different accelerations- one is the initial acceleration, two are after half of the time step has elapsed, and one is at the end of the time step. Each set of accelerations is calculated based on the accelerations calculated in the previous set, aside from the first set which only uses the initial positions. This is done using suvat equations:

$x_{n+1} = x_n + v_n t + \frac{1}{2} at^2$

$v_{n+1} = v_n + at$

The function then returns the sum of all these accelerations, however the middle two accelerations are multiplied by two as these more accurately represent the overall acceleration for the time step. This sum is then divided by 6 to provide a weighted average acceleration for that time step.

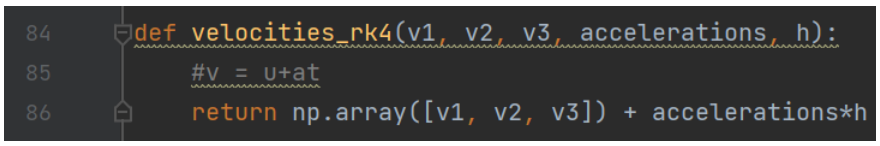
This can be used to update the positions of the three bodies as such:



```python
        a1, a2, a3 = avg_accelerations_rk4(current_v1, current_v2, current_v3, current_x1, current_x2, current_x3, h)
        #s = ut+1/2at**2 with a being avg acceleration
        current_x1 = current_x1 + current_v1 * h + (a1*h**2)/2
        current_x2 = current_x2 + current_v2 * h + (a2*h**2)/2
        current_x3 = current_x3 + current_v3 * h + (a3*h**2)/2
```

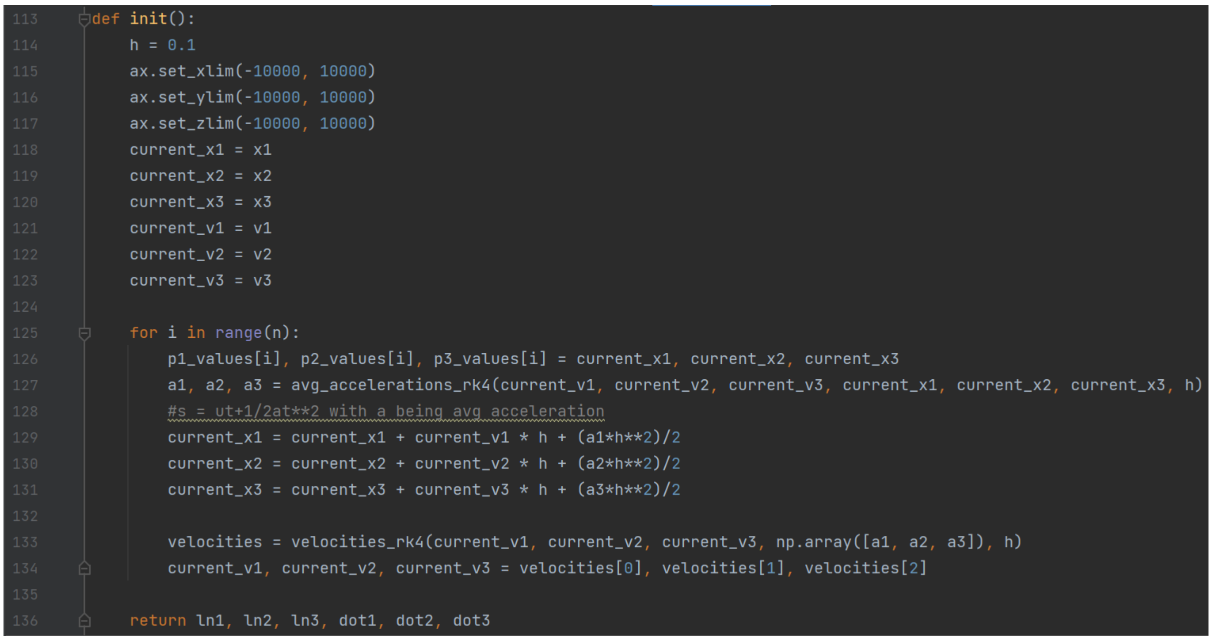
In our program, a fixed time step is used although in practise to improve computational speed an adaptive step size can be used so that for time periods where the error generated by the Runge-Kutta integrator is greater a smaller step size is used and where the error is smaller a larger time step can be used.

After the positions are updated the velocities are then updated. The velocities are updated afterward as the initial velocities are used in updating the positions, not final velocities.



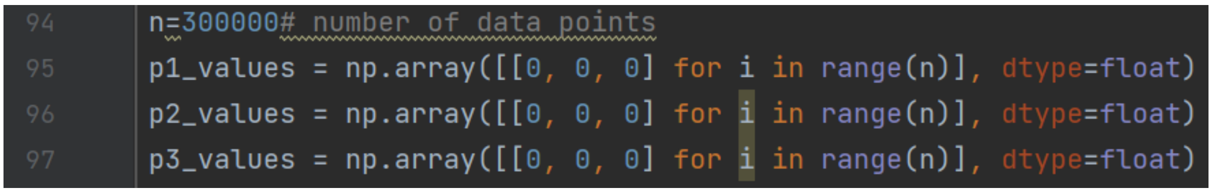
```python
def velocities_rk4(v1, v2, v3, accelerations, h):
    #v = u+at
    return np.array([v1, v2, v3]) + accelerations*h
```

Animating this simulation is done with matplotlibs FuncAnimation and a similar method to task 3 and 4, however this time an initialization function is used:



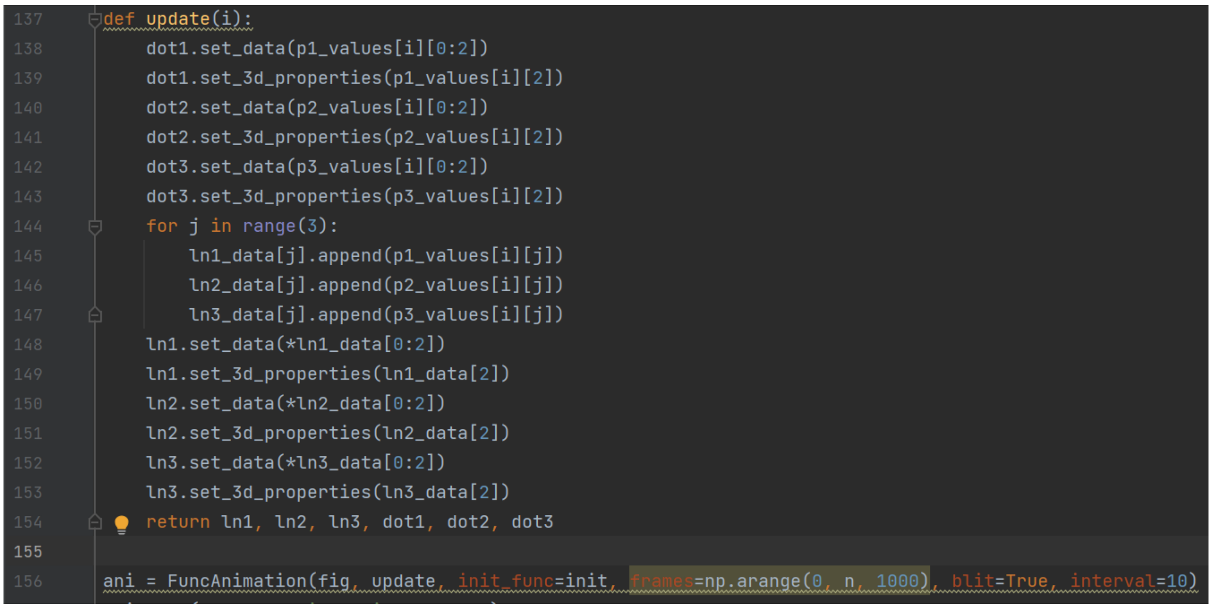
```python
def init():
    h = 0.1
    ax.set_xlim(-10000, 10000)
    ax.set_ylim(-10000, 10000)
    ax.set_zlim(-10000, 10000)
    current_x1 = x1
    current_x2 = x2
    current_x3 = x3
    current_v1 = v1
    current_v2 = v2
    current_v3 = v3

    for i in range(n):
        p1_values[i], p2_values[i], p3_values[i] = current_x1, current_x2, current_x3
        a1, a2, a3 = avg_accelerations_rk4(current_v1, current_v2, current_v3, current_x1, current_x2, current_x3, h)
        #s = ut+1/2at**2 with a being avg acceleration
        current_x1 = current_x1 + current_v1 * h + (a1*h**2)/2
        current_x2 = current_x2 + current_v2 * h + (a2*h**2)/2
        current_x3 = current_x3 + current_v3 * h + (a3*h**2)/2

        velocities = velocities_rk4(current_v1, current_v2, current_v3, np.array([a1, a2, a3]), h)
        current_v1, current_v2, current_v3 = velocities[0], velocities[1], velocities[2]

    return ln1, ln2, ln3, dot1, dot2, dot3
```

Starting empty arrays:



```python
n=300000# number of data points
p1_values = np.array([[0, 0, 0] for i in range(n)], dtype=float)
p2_values = np.array([[0, 0, 0] for i in range(n)], dtype=float)
p3_values = np.array([[0, 0, 0] for i in range(n)], dtype=float)
```

The init function sets the data in these arrays to be the positions of the three bodies p1, p2, p3. The update function then iterates over these arrays to update the positions. There are 6 different plots to redraw each frame- three for current positions of the bodies and three for the line they plot as they move.



```python
def update(i):
    dot1.set_data(p1_values[i][0:2])
    dot1.set_3d_properties(p1_values[i][2])
    dot2.set_data(p2_values[i][0:2])
    dot2.set_3d_properties(p2_values[i][2])
    dot3.set_data(p3_values[i][0:2])
    dot3.set_3d_properties(p3_values[i][2])
    for j in range(3):
        ln1_data[j].append(p1_values[i][j])
        ln2_data[j].append(p2_values[i][j])
        ln3_data[j].append(p3_values[i][j])
    ln1.set_data(*ln1_data[0:2])
    ln1.set_3d_properties(ln1_data[2])
    ln2.set_data(*ln2_data[0:2])
    ln2.set_3d_properties(ln2_data[2])
    ln3.set_data(*ln3_data[0:2])
    ln3.set_3d_properties(ln3_data[2])
    return ln1, ln2, ln3, dot1, dot2, dot3

ani = FuncAnimation(fig, update, init_func=init, frames=np.arange(0, n, 1000), blit=True, interval=10)
```

How to use the tasks:

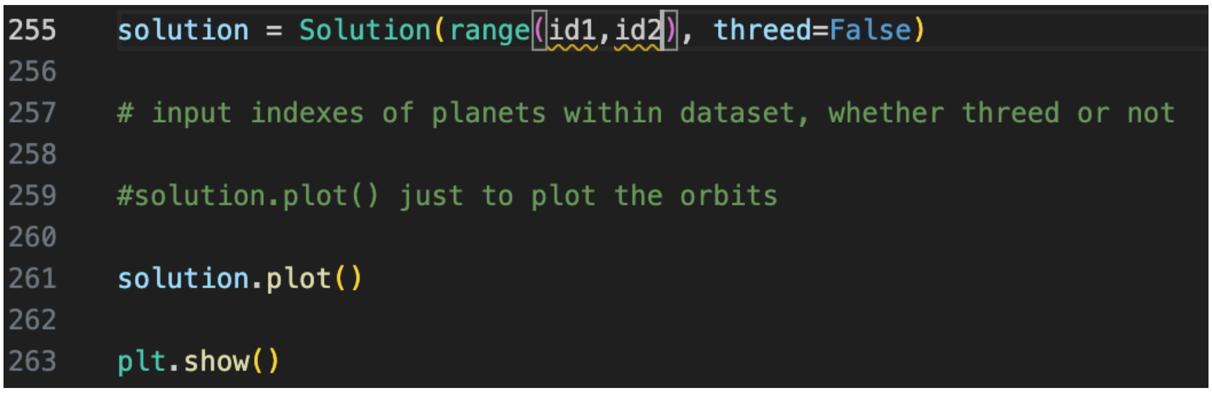
Task 1:

Since task 1 is basic nothing needs to be changed you only need to run the code with matplot installed for it to run.

Task 2:

To plot 2d orbits for task 2 planet ids need to be added in line 255 and thread needs to be false to make it 3d .

line 261 needs to be solution.plot.

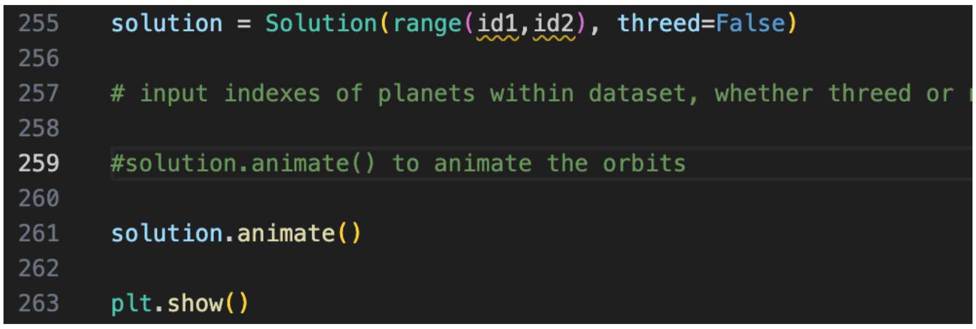


```
255    solution = Solution(range(id1,id2), threed=False)
256
257    # input indexes of planets within dataset, whether threed or not
258
259    #solution.plot() just to plot the orbits
260
261    solution.plot()
262
263    plt.show()
```

Task 3:

To animate the 2d orbits for task 3 planet ids need to be added in line 255 and thread needs to be false

Line 261 needs to be solution.animate

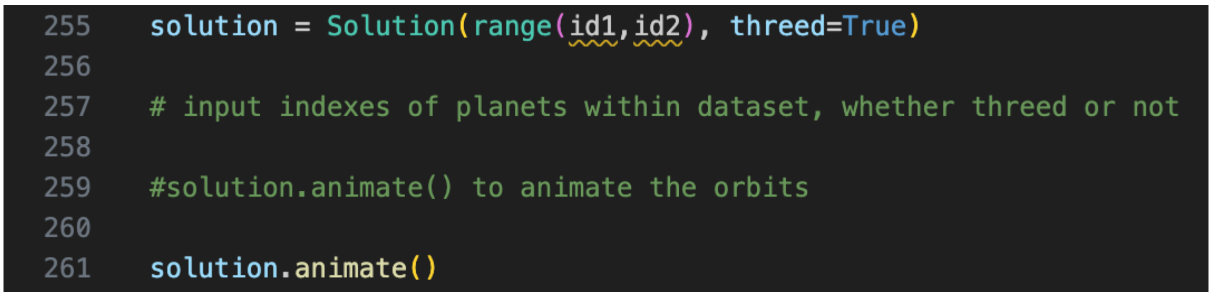


```
255    solution = Solution(range(id1,id2), threed=False)
256
257    # input indexes of planets within dataset, whether threed or
258
259    #solution.animate() to animate the orbits
260
261    solution.animate()
262
263    plt.show()
```

Task 4:

To animate the 3d orbits for task 4 planet ids need to be added in line 255 and thread needs to be true

Line 261 needs to be solution.aimate



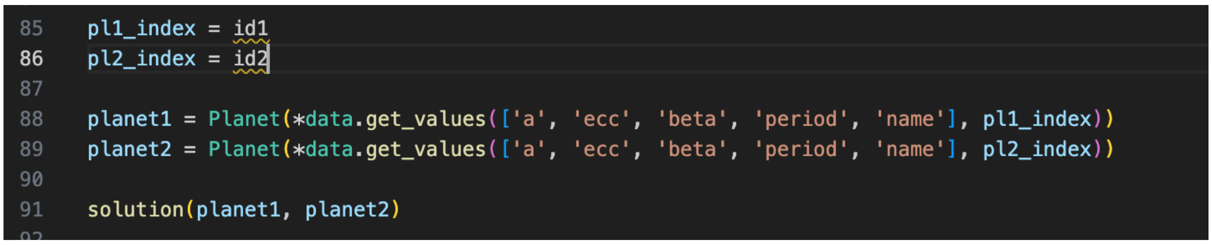
```
255    solution = Solution(range(id1,id2), threed=True)
256
257    # input indexes of planets within dataset, whether threed or not
258
259    #solution.animate() to animate the orbits
260
261    solution.animate()
```

Task 5:

Like task 1 task 5 just produces a plot so nothing needs to be inputed.

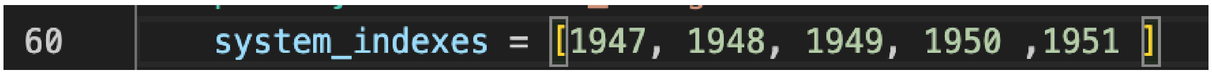
Task 6:

To use task 6 2 planet ids need to be imputed these go in lines 85 and 86 and will produce the end result



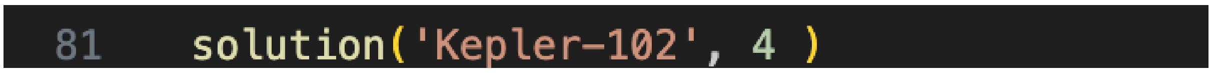
```
85    pl1_index = id1
86    pl2_index = id2
87
88    planet1 = Planet(*data.get_values(['a', 'ecc', 'beta', 'period', 'name'], pl1_index))
89    planet2 = Planet(*data.get_values(['a', 'ecc', 'beta', 'period', 'name'], pl2_index))
90
91    solution(planet1, planet2)
```

Taks 7:
To run task 7 inputs need to be made in 2 lines first is in line 60 where the ids for the system you want get added



```
60        system_indexes = [1947, 1948, 1949, 1950 ,1951 ]
```

The next input is in line 81 where you need to put the name of the system your doing in and you also have to the planet number to the side fo this 0 is the first planet in the system and if you go up it will change the origin of the plot.



```
81    solution('Kepler-102', 4 )
```

Task 8:
Task 8 needs no inputs and just makes the results.