`This font indicates code that you will type or copy/paste`

💡 indicates a possible future modification of the project

Note that these instructions are for Python 2.x! Python 3.x, the more recent version, may or may not know what to do with this code. Oops. Most compilers, including Canopy and Anaconda, will run either version of Python.

Prerequisites: trig/precalculus, introductory mechanics. Prior coding experience helpful but not required.

<u>How to make a planetary orbit simulator</u>

Before we begin coding, let's revisit some of our fundamental Newtonian mechanics equations (those are all we will need in order to create our sim).

- $d = (p_x^2 + p_y^2)^{½}$ d=distance between radii
    - You'll need to know other basic trigonometric identities/theorems
- $\Delta p = ½ a t^2 + v_o t$ p = position (along any dimension)
- $\Delta v = a t$ a = acceleration t=time m=mass
- $v_f^2 = v_o^2 + 2a\Delta p$ v = velocity F=force
- $F=ma$
- $F_{grav} = -GMm/d^2$ (negative because it's an attractive force)
    - Note: $G = 6.67*10\text{-}11\ N*m^2/kg^2$

Now, let's begin!

First you'll type:

```
import numpy as np
import matplotlib.pyplot as plt
import time
import pylab as pl
```

numpy, matplotlib, time, and pylab are all packages containing useful functions. Python makes it very easy to import function packages: just type "import". The "as" indicates that you're giving the package a nickname, which makes it easier for you to call on it later (i.e. less typing – programmers are lazy).

Then we set the initial conditions.

> 💡 You could use Python's `raw_input('prompt: ')` function to have users input their own data.

For now we will use the values for the earth and the moon, since we know that the moon has a fairly regular orbit.

To create a variable in Python, just type the variable and set it equal to a value. In Python, the variable is always on the left of the equals sign, and the value it is being assigned to is on the right.

```
t = 0
M = 6e24
m = 7.3e22
```

```
vx = 0.1
vy = 1000.
px = 3.8e8
py = 3.
G = 6.6e-11
```

The p in px and py stands for "position".

Our base units for this simulation are meters, seconds, and kilograms.

> Fun fact: the United States once lost a NASA orbiter because some engineers on the project used metric units, while others used US customary units. Please use metric units.

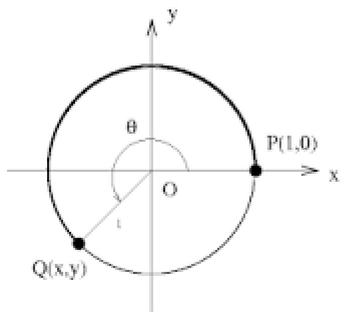
We will also create two lists, one for x and one for y values of the moon over time. For simplicity's sake, the earth is fixed at (0,0). We name the lists pxs and pys (as in px's and py's). For now they have only one entry: the starting position. The brackets inform Python that we're creating a list and not a variable.

```
pxs = [px]
pys = [py]
```

> 💡 You could add a third dimension
>
> 💡 You could add a third body, or *n* bodies. Would one be fixed in the center? How could the graph automatically accommodate their full positional range?

We will later have Python turn the list into graph points: (x_1,y_1), (x_2,y_2), and so on. Python figures out what you want pretty well.

To calculate the x and y components of force, we must first find the angle measure of the earth and moon at any given time, from an arbitrary fixed line at 0 radians.



The angle measure is the arctangent of the y and x position of the moon. Note that the arctangent will be the same at, say, π/4 and 5π/4, since y/x = (-y)/(-x). The traditional solution is to create an if/else statement that evaluates the sign of the x and y values and determines which angle measure is appropriate. However, the numpy package has a function – arctan2 – that does all that for us.

So, we define theta in terms of arctan2. Since px and py change, the best course of action is to create a function "theta" that is called upon any time we need to find the angle measure. We do this by defining theta with expected inputs, calculating the arctangent from those inputs, and returning that angle measure as the output. Note that arctan2 expects the y value first.

```
def theta(_py, _px):
    return np.arctan2(_py, _px)
```

The “np.” informs Python that the arctan2 function is located within the numpy package. The underscore (“_”) prevents Python from getting mixed up between the px above and px in the theta function.

Now, to give the simulation user something to do, we will create a means for them to input the simulation length and simulation precision. We will need new variable: ttot (total time) and deltat. The format for requesting user input is:

```
ttot = float(raw_input('simulation length: '))
deltat = 1 / float(raw_input('iterations per second: '))
iters = ttot / deltat
```

For the raw_input function, Python displays the prompt (in this case “simulation length: ”), waits for the user to type a value, and returns that value as a string. In Python, there are several flavors of variables. A string can contain numbers or letters, and is treated as words. Therefore Python cannot attribute mathematical meaning to strings. A simple fix is the float() function. A float is another type of variable, in which values are treated as decimal values (as opposed to integers or strings). Typing float(x) converts string x into float x, enabling us to perform mathematical operations with it.

The value ttot is an amount of time, in seconds. The deltat, or amount of time between calculations, is the inverse of the number of iterations per second.

> Note: for this simulation, seconds always (excepting the time.sleep function) refers to moon-orbiting-earth seconds, not sitting-at-the-computer seconds. In other words, the user will see results much quicker by running the simulation than by staring at the sky.

The number of iterations is the total time divided by deltat – effectively the total time times the number of iterations per second. We couldn’t do division if the inputs were being treated as strings.

> 💡 What happens if the input for “iterations per second” is 0? What happens if there are non-numerical characters for either raw_input? Craft an if/else statement that prohibits the program proceeding until valid values are inputted.

Now it’s time to craft the simulation itself. We could nest our instructions within a function, but as Python reads code top-to-bottom, that won’t be necessary (provided our code is well-organized).

We want to recalculate position values many times (specifically, “iters” times). The simplest loop in Python is a for/in loop. For/in statements take the form “for i in [*list*]”, where i is a placeholder and needn’t appear anywhere else. Anything under a for/in statement will be repeated once for every value in the list. Here, the calculations will be repeated “iters” times. The number of repetitions is an integer, so we convert iters to an integer with the int() function. Python will round as necessary.

The range(x) function creates a list with x entries ranging from 0 to x-1. The specific values are irrelevant; the number of entries is taken into account by the “for/in” loop.

The code looks like:

```
for i in list(range(int(iters))):
```

So, anything underneath and indented will be repeated “iters” times, rounded.

Take care that anything being repeated is indented! In lieu of parentheses (or whatever it is that other programs use), Python uses indents to denote relations between lines of code, and it’s very fussy about it!

The first order of business in our loop is to find the earth-moon distance, needed for force calculations.

```
emd = (px**2 + py**2)**0.5
```

In Python, two stars (**) denotes exponentiation. Knowing this, one should recognize the above as the Pythagorean Theorem. Take care with parentheses, too!

Now we calculate the gravitational force. Take care with sign! (You’re probably learning pretty quickly that there’s a lot that can go wrong with coding. Get it correct now and you won’t spend two weeks debugging afterwards!)

```
F = -G*m*M / (emd**2)
t += deltat
```

“+=” tells Python to take the current t value and add deltat. t itself isn’t used for our calculations, but it’s good physics practice (and potentially helpful for debugging any modifications you make).

Now we update the other conditions. First we must calculate the force components. Fortunately this is among the first lessons in introductory mechanics courses and in vector algebra.

```
Fx = F*np.cos(theta(py, px))
Fy = F*np.sin(theta(py, px))
```

For variable names, be consistent in capitalization.
Now we use our Newtonian mechanics equations:

```
ax = Fx / m
ay = Fy / m
vxf = vx + ax * deltat
vyf = vy + ay * deltat
```

See where we’re going with this?

For this simulation, we will average the initial and final speeds in calculating change in position (there are probably neater ways of doing it). Second semester calculus students will recognize this as a trapezoidal sum.

```
px += (vx + vxf) * deltat / 2
py += (vy + vyf) * deltat/2
vx = vxf
vy = vyf
```

Finally, we add our newly calculated position values to our lists with the append function.

```
pxs.append(px)
pys.append(py)
```

If you like, you can introduce a pause (below) in order to build suspense. It will also make the simulation more realistic, since professional astrophysicists run simulations that take much longer than this. However, we divide deltat by a large number so that it doesn't take 27 days to get our graph.

Optional:
```
time.sleep(deltat/10000)
```

We are done with our loop. What remains should not be indented.

Our last portion is the creation of a basic plot with axes. matplotlib.pyplot contains all the graphing tools we will need.

```
fig = plt.figure()
ax = plt.gca()
```

> 💡 There are methods of graphing in which you see the graph forming, rather than merely viewing the final product.

Then we plot the data onto the graph, telling Python that our data are our two position lists.

```
pl.plot(pxs,pys)
```

Now we change the limits of the graph. The moon's orbit around the earth comprises a near-circle with radius $3.8*10^8$ meters, so the following limits should be appropriate.

```
pl.xlim(-1e9, 1e9)
pl.ylim(-1e9, 1e9)
```

> 💡 In many cases, the display is not square, and the graph will be stretched so that the near-circle appears to be an ellipse. The same thing happens with many graphing calculators. Experiment with limits to counteract this effect.

1e9 = $1*10^9$; Python deals with scientific notation very nicely.

Finally, we display the graph for the user!

```
plt.show()
```

Helpful hint: the moon orbits the earth in 2.4 million seconds. That's 2400000.0. For simulations with that many data points, don't do a whole iteration per second! Inputs on the order of (1000000, 0.0001) will give you a nice quick arc.

Now you can mess around with the 💡's!

Teaching tools

What is the expected orbital velocity? Does ours match that? (Hint: $v=\sqrt{\frac{GM}{r}}$, so v = (G*M/emd)**0.5)

Can you calculate potential and kinetic energy, and graph or print those data? Is total energy conserved? (Hint: it should be)

Do F, a, v, and emd have constant magnitude? (They won't unless the orbit is perfectly circular, and most aren't)

Can you calculate eccentricity? Does it match that of the real earth and moon?

Does our system follow Kepler's laws? How might Python calculate that?

If the velocity or mass of the moon changes, how long does it take for us to notice?

How might ternary systems work? Can you generate code for one, and try to get stable orbits?

In real life, perhaps for more massive/speedy bodies, what other factors are there? Relativistic effects? Do the bodies' own revolutions affect their motion?

Author's note

Python is a great language for beginning programmers, especially ones who enjoy math and science. I found it to be fairly intuitive and well-documented online. Also, codecademy.com is a good resource for learning code.

Thanks to codecademy.com for teaching me Python (even if it was the wrong version!), scipy.org and pythoncentral.io for helping me find functions, and Northwestern's astrophysics grad students for helping me to debug. Most of all, my thanks to Doc V for giving us such an open-ended senior project and for inspiring mine with his own efforts to bring science education to students lacking in resources.

This project is for those students and for students like me, who attend schools with programming classes but physically cannot fit them into their schedules.

Elisabeth Finkel
5/23/2017