Calculating Orbits With Python

Aims

Laboratory Skills

This experiment is designed to develop your computing skills. Specifically you will learn:

- the basics of the programming language Python;
- how to use Python to automate difficult calculations;
- an understanding of the trade-off between precision and computing time in simulations.

Astronomical knowledge and understanding

In addition, this experiment will help you:

- to solve Newton's laws of motion for an orbiting planet;
- to gain an intuitive grasp of the nature of elliptical orbits.

Introduction

In PHY106 you have looked at the orbits of planets and other bodies in our solar system, whilst in PHY104 you will shortly see how clever manipulation of Newton's Law of Gravity allows astronomers to measure the masses of celestial objects from tiny dwarf planets to entire clusters of Galaxies. In this lab you'll directly examine the solutions to Newton's law of gravity by calculating the orbit of the Earth around the Sun, and investigating the consequences of minor perturbations to the Earth's orbital speed. Although Newton's laws can be solved exactly for a two-body system, here we will take a computational approach. Whilst not strictly necessary, a computational approach is easily generalised to a planetary system containing many planets, for which there is no analytical solution.

The calculations in this lab are not simple. I'm sure you've all experienced "bracket hell" in Excel, where entering a complicated formula for a cell gives you a reference that looks something like this

=ABS(B2*SQRT((0.000005/A2)$^{2+(0.000003/1.3382282)}$2 + (0.00025/2454365.9)^2)).

This formula is opaque, hard to debug if it doesn't work, and not even that complicated! What do you do if you want to perform more complicated calculations? Or to make plots not available in Excel?

In our pre-lab session you're going to learn about Python, a powerful, but simple computing language. Python makes complex calculations easy, and allows you to do things otherwise impossible within Excel. Python will be used throughout your astronomy degree for small tasks - this lab serves as your introduction. Whilst you won't be expert in Python by the end of this lab, hopefully you will have a glimpse of how useful it can be.

This is a short lab, and we will only cover the basics of Python programming. If you want to learn more, there is an excellent course online (http://interactivepython.org/courselib/static/thinkcspy/index.html), and more detailed links at the bottom of this page.

Pre-Lab Exercise

In the pre-lab we're going to learn some basics of the Python language. In the afternoon session we'll get our hands dirty using Python to compute orbits for planets around stars. Throughout this pre-lab exercise, **enter the code shown into your own Python session, so you can see it run for yourself.**

Is python installed on my managed desktop machine?

Look for "Python EPD" under "All Programs". If it's there, then great. If not, there should be a "Software Center" icon on the desktop. Run the software center by double clicking this icon. Search for "python" and install Python-2.7, if it is not already installed.

How to run Python

There are two ways to run Python *shell mode* and *program mode*. In *shell mode* you enter commands into the *Python interpreter* and they are executed straight away.

You can start the shell mode by looking for "Python EPD" in "All Programs" and running "iPython". The example below shows shell mode being used to do simple calculation. Have a go at this yourself.

```
In [1]: print 2+3
        5
```

You can also write programs in Python using the *program mode*. Under "Python EPD" run "IDLE". This starts a Python shell. Click "File->New Window" and a blank window appears in which you can write, and save a python program. You can run a python program by double-clicking on the file in windows explorer, or by pressing F5 in IDLE.

Exercise: Enter the code above and save it into a file called "test.py". Run it and see what happens. Note that you can also run your saved program by double-clicking on it from Windows Explorer.

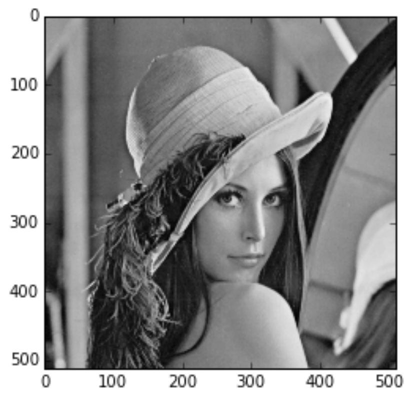
Python's *shell mode* is most useful when you want to play around, and see the results of what you are typing immediately. Writing a *python program* and saving it is more useful if you want to be able to run your code later, or have a saved record of what you did.

The power of Python

The following is an example of how powerful Python can be. Don’t worry if you don’t understand what’s happening at all - just follow along by running the code you see below in Python's *shell mode*.

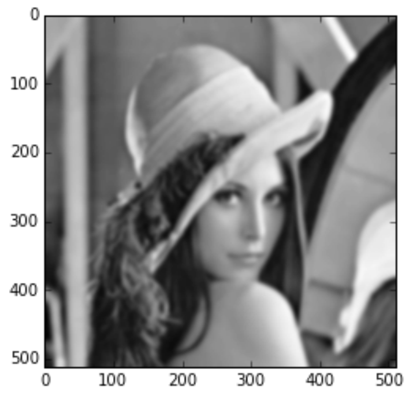
We will use Python to load and display an image

In [3]:
```
#ignore the first line! It makes plots appear in my notes
%matplotlib inline
from scipy import misc
image = misc.lena()
import matplotlib.pyplot as plt
plt.imshow(image,origin='upper',cmap='gray')
plt.show()
```



We can use Python to blur the image...

In [4]:
```
from scipy import ndimage
blurred_image = ndimage.gaussian_filter(image, sigma=3)
plt.imshow(blurred_image,origin='upper',cmap='gray')
plt.show()
```



And we can do lots of other complicated stuff very easily. However, we are getting ahead of ourselves. Let's go back to the basics.

Variables

The heart of any programming language is the *variable*. You can think of a variable as a labelled box in which you can store things. For example

In [5]:
```
x = 5
```

This statement says to the computer "put the value of '5' inside the box labelled 'x'". This is useful because we can use it later in calculations. For example:

```
In [6]: y = 2*x
        print x
        print y
```

```
5
10
```

shows how we can use variables in a programming language. Variables can store many different types of object; including integer numbers, decimal numbers, strings of characters and many more! For example, all of the following are OK.

```
In [7]: x = 5
        myName = 'Stuart'
        pi = 3.141592653589793
```

Python will keep track of what type of object is stored in each variable.

```
In [8]: print type(myName)
        print type(pi)
        print type(x)
```

```
<type 'str'>
<type 'float'>
<type 'int'>
```

The `type` of a variable is the type of the object it currently refers to.

We use variables in a program to "remember" things, like the current score at the football game. But variables are variable. This means they can change over time. You can assign a value to a variable, and later assign a different value to the same variable.

```
In [9]: x = 5
        print x

        x = 10
        print x
```

```
5
10
```

The statement `x=5` is not a mathematical equation. It's an instruction to the computer which means "take the value on the RHS (5), and store it in a variable with the name on the LHS (x)". The RHS is calculated first, then the result is put in the variable named on the left. Therefore, the following statment makes perfect sense in Python!

```
In [10]: x = x + 3
         print x
```

```
13
```

Operators

Operators are how Python does division, multiplication etc. The following are all operators whose meaning is reasonably clear:

```
In [11]: print 20 + 32
         print 5**2
         print 100/4
         print x*2 + 1

         52
         25
         25
         27
```

Note the lack of brackets in the last example! Learning the order the operators work in can make your code easy to read. Powers (like squaring a number) are done first, then the multiplications, then the additions. Therefore the following two statements give the same result, but one is much easier to read!

```
In [12]: x = 100
         y = 4
         print 100 + x * y**2
         print 100 + (x * (y**2))

         1700
         1700
```

A warning about division and integers

Python has one pretty serious design flaw. Look at the code below and the output from it. Does it do what you expect?

```
In [13]: new_value = 3/2
         print new_value

         1
```

The problem is that when python divides integer numbers, the result is also an integer number! To get the behaviour we want, we need to make sure at least one number in the division is a decimal number, by including the decimal point

```
In [14]: new_value = 3./2
         print new_value

         1.5
```

Modules

Modules provide powerful features in Python, without us having to write the code to do it ourselves! Modules can be used to send email, post tweets, or fit complex models to data. The large number of modules available for scientific computing is what will make Python so useful to you over your degree. You've already seen an example of using modules when we manipulated the "Lena" image earlier. Modules are included in your code using the "import" statement

```
In [15]: import math
         math.sqrt(100)

Out[15]: 10.0
```

The first line loads the `math` module. The dot notation in the second line means "the sqrt function that is contained within the math module". This statement uses this function to calculate the square root of 100. The math module contains many such functions, for calculating sines, cosines and converting degrees to radians, for example

In [16]:
```
print math.pi
print math.log10(100)
print math.sin(math.pi/2.0)
print math.radians(90.0)
```

```
3.14159265359
2.0
1.0
1.57079632679
```

You can find a complete list of modules that come with Python as standard here (https://docs.python.org/2/py-modindex.html). A great rule-of-thumb in Python is, if you're writing code to do something complicated, stop. See if there's a module that does it for you. **The best learning tool for Python is google!**

If you are using the iPython interpreter to run Python interactively, you can get help on any module or function by using the "help" function, e.g.

```
help(math.radians)
```

Exercise:

At the URL above, look up the documentation for the math module. Use it to calculate the factorials of 1, 2 and 3

In [17]:
```
### SOLUTION
print help(math.factorial)
print math.factorial(1)
print math.factorial(2)
print math.factorial(3)
```

```
Help on built-in function factorial in module math:

factorial(...)
    factorial(x) -> Integral

    Find x!. Raise a ValueError if x is negative or non-integral.

None
1
2
6
```

Lists and Iteration

A **list** is a container for lots of values. The simplest way to create a list is to enclose the elements in square brackets. Lists can also be stored in variables.

In [18]:
```
list_of_my_friends = ["Mary","Joe","Midge","Neil Armstrong"]
distances_in_au = [1.0,1.5,5.2,9.5]
```

Another useful way of using lists is to create an empty list, and *append* values to it

In [19]:
```
fruits = []
fruits.append('apple')
fruits.append('pear')
fruits.append('kumquat')
print fruits
```

```
['apple', 'pear', 'kumquat']
```

We can access the elements of a list by using the *index*. For example:

```
In [20]: print fruits[0]
         print fruits[1]
         print fruits[2]
         print fruits[-1]
```

```
apple
pear
kumquat
kumquat
```

Note the special index -1 above - this is a handy way of accessing the last element in a list.

Closely connected with the idea of lists is that of **iteration** - doing the same thing lots of times. We may want to repeat the same code for every item in a list, or keep adding things to a list whilst a condition is met. These things are pretty simple in Python. For example:

```
In [21]: for friend in list_of_my_friends:
             print 'Dear ', friend, ' will you come to my birthday?'
```

```
Dear  Mary  will you come to my birthday?
Dear  Joe  will you come to my birthday?
Dear  Midge  will you come to my birthday?
Dear  Neil Armstrong  will you come to my birthday?
```

```
In [22]: while len(fruits) < 6:
             fruits.append('banana')
         print fruits
```

```
['apple', 'pear', 'kumquat', 'banana', 'banana', 'banana']
```

Note that the code inside the loops above are *indented* by four spaces. This is how Python knows what code belongs to the loop, and what belongs to the main code. So in the `while` example, the `fruits.append` line is indented and gets repeated. The `print fruits` line is not indented. It belongs to the main code, and will only happen after the while loop has finished.

Functions

You've already seen how to **use** functions from the math module. How can you write your own functions? Functions are used to group together lines of code, so they can be re-used again and again. The definition of a function in python looks like this:

```
def name(argument1, argument2): statements
```

Function names can be anything you like, but a good function name makes it clear what the function *does*. The *arguments* are a list of values that should be provided to the function for it to work. Each argument is separated by commas, and you can have as many as you like. Inside the function is a list of code statements. These define what the function actually *does*. Importantly, the statements in a function have to be embedded by four spaces - this is how Python knows what statements belong to a function, and which to the main code. Let's look at an example:

```
In [23]: def metres2AU(distInMetres):
             """Converts a given distance in metres to the corresponding value in AU"""
             AU = distInMetres/1.5e11
             return AU
```

This function is called "metres2AU". It has one argument - the distance in metres. The statements inside the function calculate the distance in AU and then "return" the value to the user of the function. Below is an example of how this might be used

In [24]:
```
marsSunDistance = 2.279e11
distInAU = metres2AU(marsSunDistance)
print distInAU
```

```
1.51933333333
```

Since this is the most complicated bit of code you'll come across, let's break it down. We put the value 2.279e11 into a variable called `marsSunDistance`. We then call the function `metres2AU` with `marsSunDistance` as an argument. Since this is a variable containing the number 2.279e11, the function gets given the value 2.279e11 to work with.

The function metres2AU executes the code we asked it to. When we wrote the function we called it's argument `distInMetres`. So the number 2.279e11 gets put into a variable called `distInMetres` for the function to use. The first statement in the function divides this number by 1.5e11 and *returns* this value to the user. We save this value in a new variable called distInAU, and print it out.

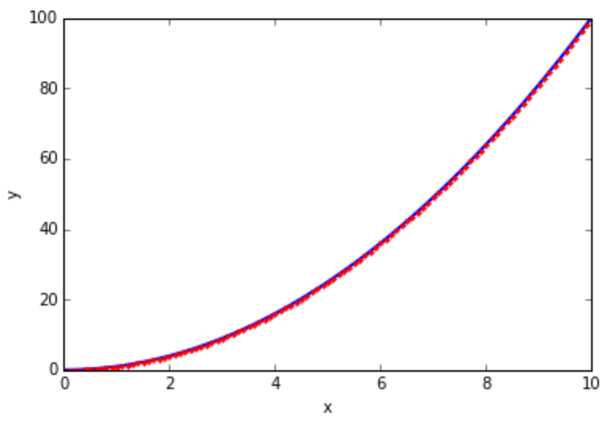
If you can understand this example, you've cracked programming! If you can't understand it, then ask a demonstrator for help now - it will pay off later!

Plotting in Python

Python is very nice for plotting data. It is much more flexible than Excel and provides nice plots that can be saved in almost any file format. Like most things in Python, plotting is handled via a module. In this case, the `matplotlib` module has a `pyplot` sub-module which is used for all types of plots. Full documentation and help for the matplotlib module is found here (http://matplotlib.org/api/pyplot_summary.html), and a quick tutorial can be found here (http://matplotlib.org/users/pyplot_tutorial.html).

The code below gives an example of how to plot x-y data with Matplotlib.

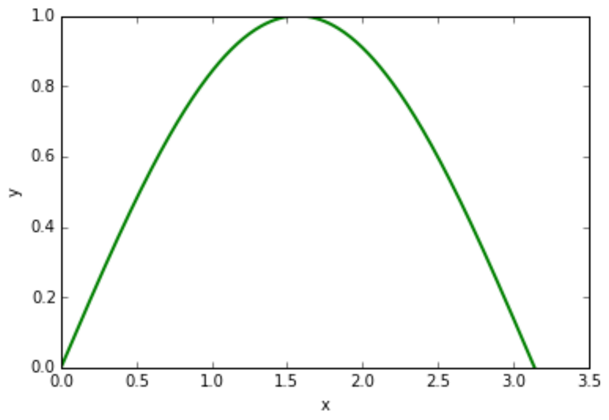
In [25]:
```
# in python, comment lines start with a hash
# these are ignored by the computer,
# but make code more readable to humans!

# make empty lists for our x-data and y-data
x = []
y = []

# start at x = 0.0
xval = 0.0

# iterate while x is less than 10
while xval < 10.0:
    x.append(xval) # store this value of xval in our list of x data
    y.append(xval**2) # store x**2 in our list of y-values

    # now increase the value stored in xval by 0.1
    xval = xval + 0.1

# the remaining code is not indented, and does not get repeated!
# import the matplotlib module, and rename it to save typing
import matplotlib.pyplot as plt

# plot the x-y data
plt.plot(x,y,linewidth=2)

# we can customize many aspects, such as using lines vs symbols
# and colors. For example, the next command plots red dots instead
# of a line
plt.plot(x,y,'r.')

# add labels
plt.xlabel('x')
plt.ylabel('y')

# display plots with plt.show()
plt.show()
```



Exercise:

Write your own Python code to plot y=`sin(x)` for x values in the range 0 to pi. Save this code as a Python program. You will want to keep a copy for you lab book.

In [26]:
```
###SOLUTION

# import math module (look for comments in code as example of good solutions)
import math

# arrays to hold data
x = []
y = []

# start at 0.0
xval = 0.0

# iterate until pi
while xval < math.pi:
    # store xval for plotting
    x.append(xval)
    # store y=sin(x) for plotting
    y.append(math.sin(xval))
    # increment xval
    xval += 0.01

# the remaining code is not indented, and does not get repeated!
# import the matplotlib module, and rename it to save typing
import matplotlib.pyplot as plt

# plot the x-y data as solid line
plt.plot(x,y,linewidth=2,color='g')

# add labels
plt.xlabel('x')
plt.ylabel('y')

# display plots with plt.show()
plt.show()
```



Summary

You now know just enough Python to be dangerous! In this afternoon's lesson we'll use Python to calculate the Earth's orbit around the Sun.

Lab Session

The goal of this lab session is to calculate the orbit of the Earth around the Sun, and then to see what happens to the orbit of the Earth if the Earth receives minor perturbations to its velocity.

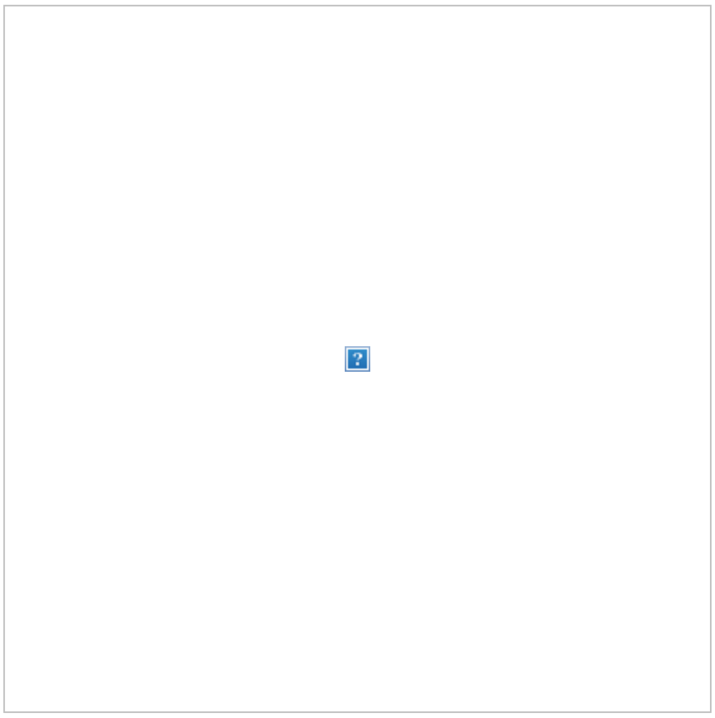
Theory

In reality, two bodies orbit around a common centre of mass. For the Earth-Sun system, the centre of mass is well inside the Sun itself. We will start by making the assumption that the Sun is fixed in space. The gravitational acceleration on the Earth is then

$$a = \frac{GM_\odot}{r^2} = \frac{GM_\odot}{x^2 + y^2}, \tag{1}$$

where r is the distance between the Earth and Sun and (x, y) are the Earth's cartesian co-ordinates, in a system where the sun is at $(0, 0)$. You'll need to calculate the component of that acceleration in the x- and y-directions.

Exercise: Using the diagram below, write equations for the amount of acceleration in the x- and y-directions.



Exercise: Using the equations you've derived, write a python function that accepts the x and y position of the Earth as arguments, and returns the x and y acceleration of the Earth. Call your function 'accel', and save your python function in a file named 'orbit.py'

The accelerations in the x- and y-directions are the projections of a above. Therefore $a_x = -a\sin(90 - \theta) = -a\cos\theta$ and $a_y = -a\sin\theta$.

Since e.g. $\cos\theta = x/\sqrt{x^2 + y^2}$ this can be simplified to

$$a_x = \frac{-GM_\odot x}{(x^2 + y^2)^{1.5}}$$

and

$$a_y = \frac{-GM_\odot y}{(x^2 + y^2)^{1.5}}$$

In [28]:
```
### SOLUTION
# There are two ways to do this, using the x,y formula above, or using the
# sine and cosine functions in the math module. If the latter, we use the
# atan2 function to get the angle for a given x and y
def accel1(x,y):
    # i am expecting x and y in m, so work in SI units
    theta = math.atan2(y,x)
    bigG = 6.67e-11 # in SI units
    mSun = 2.0e30 # in SI units
    a = bigG * mSun / (x**2 + y**2)
    ax = -a*math.cos(theta)
    ay = -a*math.sin(theta)
    return (ax,ay)

def accel2(x,y):
    bigG = 6.67e-11 # in SI units
    mSun = 2.0e30 # in SI units
    a = bigG * mSun / (x**2 + y**2)
    ax = -a*x/math.sqrt(x**2 + y**2)
    ay = -a*y/math.sqrt(x**2 + y**2)
    return ax, ay # notice how brackets not essential

# test both at 1AU, 0
print accel1(1.5e11,0)
print accel2(1.5e11,0)
```

```
(-0.005928888888888889, -0.0)
(-0.005928888888888889, -0.0)
```

Hint: if you want to return several values from a function in Python, separate them with commas, as in the example below. You can use the same trick to store the results of the function in two variables...

In [24]:
```
def twentyfourclock(time):
    return time, time+12

am, pm = twentyfourclock(7)
print am, pm
```

```
7 19
```

Method

We now calculate the orbit numerically using the following method. Suppose that at some time, t_0, we know the (x, y) position of the Earth (x_0, y_0) and the velocity of the Earth $(V_{x,0}, V_{y,0})$. We want to know the position and velocity of the Earth after a small time, δt. If we assume that the Earth moves with **constant acceleration** over this small time, we can write

$$\begin{aligned} x_1 &= x_0 + V_{x,0}\delta t + \frac{1}{2}a_{x,0}\delta t^2 \\ V_{x,1} &= V_{x,0} + a_{x,0}\delta t \end{aligned}$$

where x_1 and $V_{x,1}$ are the position and velocity of the Earth after the small timestep, and $a_{x,0}$ is the x-acceleration at t_0. Similar equations hold in the y-direction.

What should we adopt for our position and velocity at t_0? We will start with a circular orbit for the Earth. The size of the Earth's orbit is 1 AU = 1.5×10^{11} m. Say we take our starting point as when the Earth is on the x-axis, so its position is (1 AU, 0). We can also work out the speed of the Earth's orbit (along the y-direction at this position) by balancing the centrifugal force with the attraction of gravity.

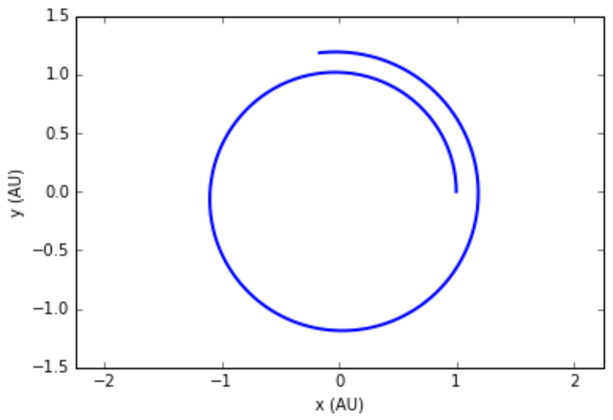
Exercise: The python code below uses your 'accel' function to calculate the orbit of the Earth over one year. **It is broken.** There are a few bugs. Some will cause the code not to run, some are more subtle. Copy the code as it is into your 'orbit.py' file and see if you can fix it.

```
import math

### INSERT YOUR VERSION OF THE ACCEL FUNCTION HERE ###

yearInSecs = 24*60*60*365.256363004
AU         = 1.49598e11
bigG       = 6.673e-11
Msun       = 1.98892e30

# CIRCULAR ORBIT
# Sun fixed at (0,0)
xStart = AU
yStart = 0.0

# arrays for storing position. Put first values in
xPos   = [xStart]
yPos   = [yStart]

# starting velocity
vx = math.sqrt(bigG*Msun/AU)
vy = 0.0

# set timestep to 1/200th of a year
dt = yearInsecs/200

# loop over just longer than a year
# each time we gor round the loop we
# will calculate new positions
t = 0.0
while t < 1.5*yearInSecs:
    t = t+dt

    # get easy access to previous step's X,Y positions
    # by storing their values in variables x0, y0
    x0 = xPos[-1]
    y0 = yPos[-1]

    # calculate the acceleration at the current position
    ax0,ay0 = accel(x0,y0)

    # calculate new positions using eqns of motion
    x1 = x0 + vx*dt + 0.5*ax0*dt**2
    y1 = y0 + vy*dt + 0.5*ax0*dt**2

    # calculate new velocities
    # note how we store them back into vx, vy
    # next time round the loop we will automatically
    # be using updated velocities
    vx = vx + ax0*dt
    vy = vy + ax0*dt

    # save new x and y values in list for plotting later
    xPos.append(x1)
    yPos.append(y1)
```

In [32]:
```python
### SOLUTION
import matplotlib.pyplot as plt
import math

yearInSecs = 24*60*60*365.256363004
AU         = 1.49598e11
bigG       = 6.673e-11
Msun       = 1.98892e30

### INSERT YOUR VERSION OF THE ACCEL FUNCTION HERE ###
def accel(x,y):
    # i am expecting x and y in m, so work in SI units
    theta = math.atan2(y,x)
    # notice how I can use the 'global' variables in here
    # use as an opportunity to discuss variable 'scope'
    # read a little about this if you aren't comfortable
    a = bigG * Msun / (x**2 + y**2)
    ax = -a*math.cos(theta)
    ay = -a*math.sin(theta)
    return (ax,ay)

# CIRCULAR ORBIT
# Sun fixed at (0,0)
xStart = AU
yStart = 0.0

# arrays for storing position. Put first values in
xPos   = [xStart]
yPos   = [yStart]

# starting velocity
### NOTE BUGFIX HERE. START THE PLANET MOVING ALONG Y-AXIS
vy = math.sqrt(bigG*Msun/AU)
vx = 0.0

# set timestep to 1/200th of a year

# BUGFIX - note typos in yearinsecs!
# use this as an opportunity to discuss error
# messages, since failure to fix this gives
# an error message they should understand
dt = yearInSecs/200

# loop over just longer than a year
# each time we gor round the loop we
# will calculate new positions
t = 0.0
while t < 1.5*yearInSecs:
    t = t+dt

    # get easy access to previous step's X,Y positions
    # by storing their values in variables x0, y0
    x0 = xPos[-1]
    y0 = yPos[-1]

    # calculate the acceleration at the current position
    ax0,ay0 = accel(x0,y0)

    # calculate new positions using eqns of motion
    x1 = x0 + vx*dt + 0.5*ax0*dt**2

    #### BUGFIX: example uses ax0 by mistake
    y1 = y0 + vy*dt + 0.5*ay0*dt**2

    # calculate new velocities
    # note how we store them back into vx, vy
    # next time round the loop we will automatically
    # be using updated velocities
    vx = vx + ax0*dt

    #### BUGFIX: example uses ax0 by mistake
    vy = vy + ay0*dt

    # save new x and y values in list for plotting later
    xPos.append(x1)
    yPos.append(y1)

# convert to AU (not expecting this from students)
```

```
xPos = [x/AU for x in xPos]
yPos = [y/AU for y in yPos]

# plot as line
plt.plot(xPos,yPos,lw=2)
# i expect labelled axes!
plt.xlabel('x (AU)')
plt.ylabel('y (AU)')
# set aspect ratio to be equal
plt.axis('equal')
```

```
Out[32]: (-1.5, 1.5, -1.5, 1.5)
```



Exercise: Once you've got the code working, you'll have lists full of data, but no way of knowing if it's right! Add more code to your program to plot the x position against the y position using matplotlib.

Hint: you can use

```
plt.axis('equal')
```

to set the x and y axes to be equal in scale.

Question: You should have a circular orbit. Do you? If your orbit is not circular can you think why this might be? *Answer this question before proceeding. If you cannot answer the question, ask a demonstrator.*

Exercise: Change the timestep in your code to acheive a circular orbit.

SOLUTION

They won't have a circular orbit. This is because the equations of motion assume constant acceleration - and acceleration is not constant. It's direction is always changing. The solution is to make the timestep smaller, so that the assumption of constant acceleration over a timestep is more accurate.

A timestep of 1/2000 of a year is about right. You can talk about the trade-off between accuracy and computing time.

Elliptical Orbits

Now we shall investigate what happens when the Earth receives a small change to it's initial speed.

Exercise: Increase the initial speed of the Earth in the y-direction by 20%. Calculate the new orbit. You may have to increase the duration of the simulation. Plot the orbit on the same graph as your circular orbit from before.

Exercise: Decrease the initial speed of the Earth in the y-direction by 20%. Calculate the new orbit. Decrease the timestep if necessary to get an accurate orbit . Plot the orbit on the same graph as your circular orbit from before.

Question: Was your original timestep sufficient to get an accurate orbit, or did you need to decrease your timestep? Why might this orbit need a smaller timestep than the others?

SOLUTION

For a faster initial speed they will get an elliptical orbit with apastron equal to the Earth's orbital radius. It will take longer to orbit, so they have to increase simulation length.

For a slower initial orbit, they will get an elliptical orbit whose periastron is equal to the Earth's orbital radius. The acceleration vector in this case changes rapidly, so they will need to decrease the timestep substantially.

The plot should have the lines labelled. You might want to suggest they look at the legend function in matplotlib for this, and the label option.

e.g.

```
plt.plot(x,y,label='Circular Orbit')
plt.legend()
```

Discussion

You have seen that a body moving at a particular speed will make a circular orbit around the Sun, but that changes to the speed of that object send it on elliptical orbits.

Question: How might the work you've done here shed light on the origins of comets: icy bodies whose elliptical orbits bring them close to the Sun.

SOLUTION

Obviously, comets are objects with elliptical orbits, and come very close to the Sun. They originate from the Oort cloud, where they have circular orbits, but encounters with other Oort cloud objects slow them down, and bring them into elliptical orbits with periastrons near the Sun.

Further Reading

We've only scratched the surface of Python here. Python will be used throughout the rest of your course, and will be extremely useful for project work. If you want a really solid foundation in Python and some useful modules, I suggest looking at some of the following links.

- http://www.learnpython.org/
- http://interactivepython.org/courselib/static/thinkcspy/index.html
- http://www.codecademy.com/tracks/python
- matplotlib tutorial (http://www.loria.fr/~rougier/teaching/matplotlib/)
- numpy tutorial (http://wiki.scipy.org/Tentative_NumPy_Tutorial)
- http://www.astropy.org/ (A python module for astronomers)

The cell below loads the style of this notebook. Please ignore it.

```
In [1]: from IPython.core.display import HTML
        css_file = '../styles.css'
        HTML(open(css_file, "r").read())

Out[1]:
```

In [ ]:
```

```