Scientific Computing in Python 7 – System Calls and complete models.

Models describe a system but they are also used to test the effect of things on the system. You tried experimenting with models to see if you could get a stable system but how about testing scientific hypotheses. Let's look at our 2 equation panda and cabbage model.

```
def pandafun(y,t):


   if y[1] <1:
      function = np.array([y[0]*-1, (0*y[1])])
   else:
      function = np.array([((0.01*y[0])*(1 - y[0]/(0.5*y[1]))-(0.001*y[0])),
      ((1.6*y[1]) * (1 - (y[1]/1000)) - (2*y[0]))])
   return(function)



y0 = np.array([2,500])
t = np.linspace(0,1000)
answer = odeint(pandafun,y0, t)


cabs = answer[:,1]
pands = answer[:,0]
plt.plot(t,cabs)
plt.plot(t,pands);
plt.show()
```

Under these parameters cabbages require a growth rate of 1.9 to reach a stable state. In most real systems we can't just make things reproduce faster. But if we care about our cabbages and our pandas we want to do things that would actually be possible. How about we artificially enforce a maximum population size of pandas? This in the real world would be done with culling but we will simply stop all growth above the carrying capacity to test the maximum sustainable population size.

```
for i in range(50,300):
    def pandafun(y,t):
        if y[0] > i:
            if y[1] <1:
                function = np.array([y[0]*-1, (0*y[1])])
            elif y[0] <1:
                function = np.array([y[0]*0, ((1.6 * y[1]) * (1 - (y[1]/1000)))])
            else:
                function = np.array([y[0]*0, ((1.6 * y[1]) * (1 - (y[1]/1000)) - (2*y[0]))])
        else:
            if y[1] <1:
                function = np.array([y[0]*-1, (0*y[1])])
            elif y[0] <1:
                function = np.array([y[0]*0, ((1.6 * y[1]) * (1 - (y[1]/1000)))])
            else:
                function = np.array([((0.01*y[0])*(1 - y[0]/(0.5*y[1]))-(0.001*y[0])),
                ((1.6 * y[1]) * (1 - (y[1]/1000)) - (2*y[0]))])
        return(function)



    y0 = np.array([10,500])
    t = np.linspace(0,1000)
    answer = odeint(pandafun,y0, t)


    cabs = answer[:,1]
    pands = answer[:,0]
    plt.plot(t,cabs)
    plt.plot(t,pands);
    plt.show()
```

This will spit out hundreds of plots of different population sizes, there are some odd ones due to the fact that python's ODE solving ability essentially sucks and odeint is an incredibly stiff solver that doesn't know what to do 10% of the time. But in general what we see is a pattern of stable populations until somewhere in the 200s where the whole system collapses.

That's not very precise:

Problem 1: create a list which saves the max population and the final number of both species.

If you did this correctly, you should see that the maximum panda population is 200 individuals. If you keep it at 200 you would have a sustainable population of both.

Well now we know our maximum population size we want a nice plot of this don't we? Python doesn't produce the nicest of plots, but R does. Thankfully there are ways we can make python call R and an R script we have made. We can actually make it call nearly any program. If you want to start building your own automation tools which perform tasks for you this is the start.

This is done with the subprocess module.

```
Import subprocess

From subprocess import subprocess.call as call
```

Subprocess call allows you to call subprocesses such as other programs etc.

Before we try and call an R script though we need to make sure our data is accessible to R. The most efficient way to do that is to make sure it is saved in a csv rather than an array.

We can use the module csv

```
Import csv

file = open('modeloutput.csv', 'w')

writer = csv.writer(File, delimiter = ';')
```

If we put these lines in the start of our model script we have a csv file we can save anything to. And we can create a forloop to read the output of the model. Run a model with a maximum pop size of 200 saving the model output.

```
import csv

file = open('modeloutput.csv', 'w')

writer = csv.writer(file, delimiter = ';', lineterminator = '\n')

for i in range(len(t)):

    time = t[i]

    cab = answer[i,1]
```

```
    pan = answer[i,0]
    writer.writerow([time,cab, pan])
file.close()
```

There we have a csv with our model output, it should be saved in whatever is your current directory. Now we need to make a quick R script to plot this.

```
library(ggplot2)


setwd()
data <- read.csv('modeloutput.csv', header = F)


plot1 <- ggplot(data, aes(x = data$V1, y = data$V2))+
 theme_bw()+
 geom_line(color = 'Red', size = 1)+
 geom_line(aes(x = data$V1, y = data$V3), color = 'Blue', size = 1)+
 xlab('Weeks')+
 ylab('Population Size')


ggsave("ModelOutputPlot.pdf", plot = plot1,)
```

This is a quick R script that will produce a plot of our data as it is and save it. Make sure you change the working directory to be the one you want. Make sure you save the script in the same directory too. I called it modelplot.R.

Now we want our python script to call this R script.

This is fairly simple with :

```
subprocess.call(["Rscript","modelplot.R"])
```

If everything went well you can now create a python script that runs a model and then calls an R script that plots the model.

Problem 2: Make a test in your for loop that allows you to find out at what value of population size the model stops being stable just within the code without looking at the output and then straight away run and plot that model.

Challenge Problem

Go back to our sea level equations model and make it a total model that tests the hypotheses of how much we have to reduce agriculture. Then develop nice plots for it.