This material assumes that you have programmed before. This first lecture provides a quick introduction to programming in Python for those who either haven't used Python before or need a quick refresher.

Let's start with a hypothetical problem we want to solve. We are interested in understanding the relationship between the weather and the number of mosquitos occuring in a particular year so that we can plan mosquito control measures accordingly. Since we want to apply these mosquito control measures at a number of different sites we need to understand both the relationship at a particular site and whether or not it is consistent across sites. The data we have to address this problem comes from the local government and are stored in tables in comma-separated values (CSV) files. Each file holds the data for a single location, each row holds the information for a single year at that location, and the columns hold the data on both mosquito numbers and the average temperature and rainfall from the beginning of mosquito breeding season. The first few rows of our first file look like:

~~~

year,temperature,rainfall,mosquitos

2001,87,222,198 2002,72,103,105 2003,77,176,166

~~~

Objectives

- Conduct variable assignment, looping, and conditionals in Python
- Use an external Python library
- Read tabular data from a file
- Subset and perform analysis on data
- Display simple graphs

Loading Data

There are many ways to read in data. The most basic reads in each line as a string.

In [1]:
```
ofile = open('mosquito_data_A1.csv', 'r')
print ofile.read()
```

```
year,temperature,rainfall,mosquitos
2001,87,222,198
2002,72,103,105
2003,77,176,166
2004,89,236,210
2005,88,283,242
2006,89,151,147
2007,71,121,117
2008,88,267,232
```

```
2009,85,211,191
2010,75,101,106
```

An easier way to read files, especially in a uniform format is to use a package called numpy which incidentially is also used for array and matrix manipulation. Python has some built in abilities, but much of the flexibility of python comes from the ability to load different packages. To load a package:

```
In [2]: import numpy as np
```

We will now use numpy to read our file. Many text reading options exist including pandas and astropy.io.ascii.

```
In [3]: np.genfromtxt('mosquito_data_A1.csv', unpack = True, skiprows = 1, delimit
```

```
Out[3]: array([[ 2001.,  2002.,  2003.,  2004.,  2005.,  2006.,  2007.,  2008.,
                 2009.,  2010.],
               [   87.,    72.,    77.,    89.,    88.,    89.,    71.,    88.,
                   85.,    75.],
               [  222.,   103.,   176.,   236.,   283.,   151.,   121.,   267.,
                  211.,   101.],
               [  198.,   105.,   166.,   210.,   242.,   147.,   117.,   232.,
                  191.,   106.]])
```

The `genfromtext` function belongs to the `numpy` library. In order to run it we need to tell Python that it is part of `numpy` and we do this using the dot notation, which is used everywhere in Python to refer to parts of larger things.

When we are finished typing and press Shift+Enter, the notebook runs our command and shows us its output. In this case, the output is the data we just loaded.

We gave the genfromtxt function a few different pieces of information.

First we gave it the name of the file to read. Notice this is the only information that does not have a word = X format. That means this is required and has no default value.

Next we said unpack = True, this means we want each column to go to a separate array, not each line.

skiprows = 1: there is text in the first row, so we want to skip it

delimiter = ',': since this is a csv file, rows are separated by commas. The default for numpy.genfromtxt is tab separated.

Our call to `numpy.genfromtxt` read data into memory, but didn't save it anywhere. To do that, we need to assign the array to a variable. In Python we use = to assign a new value to a variable like this:

```
In [4]: d = np.genfromtxt('mosquito_data_A1.csv', skiprows = 1, delimiter = ',', u
```

This statement doesn't produce any output because assignment doesn't display anything. If we want to check that our data has been loaded, we can print the variable's value:

```
In [5]: print d
```

```
[[ 2001.  2002.  2003.  2004.  2005.  2006.  2007.  2008.  2009.  2010.]
 [   87.    72.    77.    89.    88.    89.    71.    88.    85.    75.]
 [  222.   103.   176.   236.   283.   151.   121.   267.   211.   101.]
 [  198.   105.   166.   210.   242.   147.   117.   232.   191.   106.]]
```

`print d` tells Python to display the text. Alternatively we could just include `data` as the last value in a code cell:

A word on variable names. The variable name I've chosen for the data doesn't really communicate any information to anyone about what it's holding, which means that when I come back to my code next month to change something I'm going to have a more difficult time understanding what the code is actually doing. This brings us to one of our first major lessons for the morning, which is that in order to understand what our code is doing so that we can quickly make changes in the future, we need to *write code for people, not computers*, and an important first step is to *use meaningful varible names*.

In [6]:
```
data = np.genfromtxt('mosquito_data_A1.csv', skiprows = 1, delimiter = ',
```

Let's deconstruct that statement:

- The first argument in the function genfromtxt is the name of the file. This is required. One way you can tell is that it doesn't say x = Y like all of the other arguments
- skiprows = 1: this is to tell genfromtxt how many rows to skip because they contain header information - we are skipping 1 row. This does have the format X = Y. This is called a keyword argument and it is optional. This means that there is a default value if you don't include it
- delimiter = ',': this says that columns are separated by commas. The default behavior is to separate by tabs
- unpack = True: this tells numpy that you want the information organized first by column, then by row.

Let's talk a little more about unpacking

In [7]:
```
a, b, c = [1, 2, 3]
print a
print b
```

```
1
2
```

Let's unpack data

In [8]:
```
year, temperature, rainfall, mosquitos = data
```

In [9]:
```
print year
```

```
[ 2001.  2002.  2003.  2004.  2005.  2006.  2007.  2008.  2009.  2010.]
```

Common Containers

Once we have imported the data we can start doing things with it. First, let's ask what type of thing `data` refers to:

In [10]:
```
print type(temperature)
```

```
<type 'numpy.ndarray'>
```

type() asks the variable temperature what variable type it is (float, interger, list, dictionary, ...) temperature is a numpy array. What can you do to a numpy array?

In [11]:
```
print temperature * 4
```

```
[ 348.  288.  308.  356.  352.  356.  284.  352.  340.  300.]
```

In [12]:
```
print temperature + temperature
```

```
[ 174.  144.  154.  178.  176.  178.  142.  176.  170.  150.]
```

in general operations are done element wise in an array. There is also a matrix object if you need to do linear algebra. Another very common type is a list.

In [13]:
```
my_list = [1, 2, 3, 'a', 'c']
print type(my_list)
```

```
<type 'list'>
```

Let's try the same operations on a list

In [14]:
```
print my_list * 4
```

```
[1, 2, 3, 'a', 'c', 1, 2, 3, 'a', 'c', 1, 2, 3, 'a', 'c', 1, 2, 3, 'a',
'c']
```

SURPRISE! This repeated the list 4 times.

In [15]:
```
print my_list + [4]
```

```
[1, 2, 3, 'a', 'c', 4]
```

Here the + acted as an append.

Differences between list and arrays:

- A list can have mixed types (floats, ints, strings, other lists), a numpy array cannot. This is part of the reason why some matematical operations only work on numpy arrays

Notice I cheated a little. What if we just try to add a number?

In [16]:
```
print my_list + 4
```

```
---------------------------------------------------------------------
```

```
--
TypeError                                 Traceback (most recent call
last)
<ipython-input-16-6008aad3738f> in <module>()
----> 1 print my_list + 4

TypeError: can only concatenate list (not "int") to list
```

We got an error message.

Error Messages

Notice how helpful this error message is:

- It tells you what kind of error you got: in this case a TypeError
- You get a useful error message telling you that you cannot combine a list with an integer
- There is an -----> pointing to the offending line

Accessing Data

so far we have printed the whole array - but what if we only want part of an array? We index

In [ ]:
```
print year
print year[0]
print year[2:5]
```

Important things to notice:

- First, Python indexing starts at zero. In contrast, programming languages like R and MATLAB start counting at 1, because that's what human beings have done for thousands of years. Languages in the C family (including C++, Java, Perl, and Python) count from 0 because that's simpler for computers to do. This means that if we have 5 things in Python they are numbered 0, 1, 2, 3, 4, and the first row in a data frame is always row 0.
- the final array includes the starting index and excludes the ending index. This says start with the 3rd element (index 2) and got to (but don't include) the 6th element (index 5)

Being lazy (aka awesome)

print the first 3 elements

```
In [ ]: print year[:3]
```

print from the 3rd element to the end

```
In [ ]: print year[2:]
```

print the last 3 elements

```
In [ ]: print year[-3:]
```

This is awesome because you don't have to know how long your array is

Methods and Objects

Python is an object oriented language which means that everything is an object. Objects know things about themselves and can sometimes perform simple operations on themselves. What does that mean in practice?

```
In [ ]: x = 2
```

```
In [ ]: dir(x)
```

Here x is just an integer, but it knows how to add with other object (**add**), you can ask it its bit_length, conjugate, etc. So it is not just an integer, it is an object

```
In [ ]: print x.bit_length()
```

You will notice that some of these methods and attributes have __ or _ and some do not. In general the ones with nothing are for common use and the ones with _ and __ you won't need unless you are trying to do something really funky - like redefine what integer addition means.

So what can you do to arrays?

```
In [ ]: print temperature.mean()
```

arrays have lots of useful methods. In addition to the dir() function, in ipython you can also type variable. to get a list of options

```
In [ ]: print data.
```

```
In [ ]: print year.min()
```

You can even operate on a piece of an array

```
In [ ]: print mosquitos[1:3].std()
```

Challenge

Import the data from `mosquito_data_A2.csv`, create new variables that hold the arrays, and print the means and standard deviations for the weather variables (rainfall and temperature).

Solution

```
In [ ]: year, temperature, rainfall, mosquitos = np.genfromtxt('mosquito_data_A1.c
        print temperature.mean()
        print temperature.std()
        print rainfall.mean()
        print rainfall.std()
```

Loops

Once we have some data we often want to be able to loop over it to perform the same operation repeatedly. A `for` loop in Python takes the general form

~~~

for item in list:

    do_something

~~~

So if we want to loop over the temperatures and print out there values in degrees Celcius (instead of Farenheit) we can use:

```
In [ ]: for temp_in_f in temperature:
            temp_in_c = (temp_in_f - 32) * 5 / 9.0
            print temp_in_c
```

That looks good, but why did we use 9.0 instead of 9? The reason is that computers store integers and numbers with decimals as different types: integers and floating point numbers (or floats). Addition, subtraction and multiplication work on both as we'd expect, but division works differently. If we divide one integer by another, we get the quotient without the remainder:

```
In [ ]: print '10/3 is:', 10 / 3
```

If either part of the division is a float, on the other hand, the computer creates a floating-point answer:

```
In [ ]: print '10/3.0 is:', 10 / 3.0
```

The computer does this for historical reasons: integer operations were much faster on early machines, and this behavior is actually useful in a lot of situations. However, it's still confusing, so Python 3 produces a floating-point answer when dividing integers if it needs to. We're still using Python 2.7 in this class, so if we want 5/9 to give us the right answer, we have to write it as 5.0/9, 5/9.0, or some other variation.

Conditionals

The other standard thing we need to know how to do in Python is conditionals, or if/then/else statements. In Python the basic syntax is:

~~~

if condition:

    do_something

~~~

So if we want to loop over the temperatures and print out only those temperatures that are greater than 80 degrees we would use:

```
In [ ]: if temperature[0] > 80:
            print "The temperature is greater than 80"
```

We can also use == for equality, <= for less than or equal to, >= for greater than or equal to, and `!=` for not equal to.

Additional conditions can be handled using `elif` and `else`:

```
In [ ]: if temperature[0] < 87:
            print "The temperature is < 87"
        elif temperature[0] > 87:
            print "The temperature is > 87"
        else:
            print " The temperature is equal to 87"
```

Challenge

Import the data from `mosquito_data_A2.csv` and loop over the temperature values. For each temperature

print out whether it is greater than the mean, less than the mean, or equal to the mean.

Solution

In [ ]:
```
year, temperature, rainfall, mosquitos = np.genfromtxt('mosquito_data_A1.c
for temp in temperature:
    if temp > temperature.mean():
        print 'temp is greater than mean'
    elif temp < temperature.mean():
        print 'temp is less than mean'
    else:
        print 'temp is equal to mean'
```

Key Points

- Import a library into a program using `import libraryname`.
- Use the `numpy` library to work with data tables in Python.
- Use `variable = value` to assign a value to a variable.
- Use `print something` to display the value of `something`.
- Use `array[start_row:stop_row]` to select rows from a data frame.
- Indices start at 0, not 1.
- Use `array.mean()` and `array.min()` to calculate simple statistics.
- Use `for x in list:` to loop over values
- Use `if condition:` to make conditional decisions

Modularization and Documentation

Now that we've covered some of the basic syntax and libraries in Python we can start to tackle our data analysis problem. We are interested in understanding the relationship between the weather and the number of mosquitos so that we can plan mosquito control measures. Since we want to apply these mosquito control measures at a number of different sites we need to understand how the relationship varies across sites. Remember that we have a series of CSV files with each file containing the data for a single location.

Objectives

- Write code for people, not computers
- Break a program into chunks
- Write and use functions in Python
- Write useful documentation

Starting small

When approaching computational tasks like this one it is typically best to start small, check each piece of code as you go, and make incremental changes. This helps avoid marathon debugging sessions because it's much easier to debug one small piece of the code at a time than to write 100 lines of code and then try to figure out all of the different bugs in it.

Let's start by reading in the data from a single file and conducting a simple regression analysis on it.

In [ ]:
```
year, temperature, rainfall, mosquitos = np.genfromtxt('mosquito_data_A1.c
```

What does our data look like?

Plotting

In [ ]:
```
from matplotlib import pyplot
```

In this case pyplot is a submodule of matplotlib. I know that everything I want to do is in pyplot, so instead of typing matplotlib.pyplot for everything, I can import this way and just type pyplot

In [ ]:
```
pyplot.plot(temperature, mosquitos)
```

That looks pretty ugly, let's make points instead of lines

In [ ]:
```
pyplot.cla()
pyplot.plot(temperature, mosquitos, 'o')
```

Ohhh, that looks like a straight line, let's see if we can fit it using the numpy polyfit function

In [ ]:
```
fit_coeff = np.polyfit(temperature, mosquitos, 1)
```

Here I am giving polyfit the independent variable (rainfall), the dependent variable (mosquitos), and the order of the polynomial to fit (here 1 since its a line). This returns the coefficients of the fit starting with the highest order first

In [ ]:
```
print fit_coeff
```

Now we can plot the fit

In [ ]:
```
fit_to_data = fit_coeff[0] * temperature + fit_coeff[1]
```

```
pyplot.plot(temperature, fit_to_data)
```

Interactive plotting:

- Save
- Zoom
- Pan
- reset
- awesomeness

Modularization

So now I've done a few things interactively:

I. Read in a CSV file of year, temperature, rainfall, and number of mosquitos
II. Convert temperature to celsius
III. Plot data

This could all go into a single function but that has a few disadvantages 1. It makes it harder to test because you have to test everything together rather than a specific task 2. It makes it harder to debug - if something breaks you have to figure out where in you single function that breaking happened rather than being pointed to specific function 3. It makes it harder to read your code. Imagine reading a book with no paragraphs... breaking your code into smaller pieces makes your code more readable. 4. It is hard to reuse your functions. It is unlikely that you will have to write code for this exact file format, but it is probably very likely that you will have to convert to celsius again. Smaller building blocks are much easier to reuse than a single large, specific program

Think of breaking your program into steps, just like we did above

Functions are the paragraphs of programming and the function name is the topic sentence, just like variables these should be very descriptive. Functions take the general form:

def function_name:

```
    do stuff

    return result
```

We'll start with some pseudo code

In [ ]:
```
def read_csv_file():
    '''
    This code will read in a CSV file of year, temperature, rainfall, and
    '''
    pass
```

In [ ]:
```
def convert_fahrenheit_to_celsius():
    '''
    This code will convert an array of tempertures from fahrenheit to cels
```

```
    '''
    pass
```

In [ ]:
```
def plot_data():
    '''
    This code will plot the arrays in x and y with symbol (default "o")
    '''
    pass
```

This is pseudo code. It doesn't do anything, but I now have a skeleton of my program and I just have to fill in the information.

I used triple quotes, aka doc string to describe the code. There is a help function that can be called on any function which gives you information on that function. By default the help function returns the doc string.

In [ ]:
```
help(plot_data)
```

In [ ]:
```
help(np.polyfit)
```

my main code will look like this:

In [ ]:
```
read_csv_file()
convert_fahrenheit_to_celsius()
plot_data()
```

Now let's fill in the details, starting with the first function

In [ ]:
```
def read_csv_file(filename):
    '''
    This code will read in a CSV file of year, temperature, rainfall, and
    '''
    year, temperature, rainfall, mosquitos = np.genfromtxt(filename, skipr
    return year, temperature, rainfall, mosquitos

def convert_fahrenheit_to_celsius():
    '''
    This code will convert an array of tempertures from fahrenheit to cels
    '''
    pass

def plot_data():
    '''
    This code will plot the arrays in x and y with symbol (default "o")
    '''
    pass
```

Let's test our first function

In [ ]:
```
read_csv_file('mosquito_data_A1.csv')
```

That looks good, but it is just returning arays that are not assigned to variables outside the function so we can't use them. Let's assign them

In [ ]:
```
year, temperature, rainfall, mosquitos = read_csv_file('mosquito_data_A1.c
```

In [ ]:
```
print year
```

Excellent, now let's convert to fahenheit

In [ ]:
```
def read_csv_file(filename):
    '''
    This code will read in a CSV file of year, temperature, rainfall, and
    '''
    year, temperature, rainfall, mosquitos = np.genfromtxt(filename, skipr
    return year, temperature, rainfall, mosquitos

def convert_fahrenheit_to_celsius(temp_in_f):
    '''
    This code will convert an array of tempertures from fahrenheit to cels
    '''
    temp_in_c = (temp_in_f - 32) * 5 / 9.0
    return temp_in_c

def plot_data():
    '''
    This code will plot the arrays in x and y with symbol (default "o")
    '''
    pass
```

In [ ]:
```
year, temperature, rainfall, mosquitos = read_csv_file('mosquito_data_A1.c
temp_in_c = convert_fahrenheit_to_celsius(temperature)
```

In [ ]:
```
print temperature, temp_in_c
```

Challenge

Write the plotting function

Solution

In [ ]:
```
def plot_data(x, y, symbol = 'o'):
```

```
    '''
    This code will plot the arrays in x and y with symbol (default "o")
    '''
```

Notice this function does not have to return anything

One final command: to save a figure from the command line (i.e. not using the GUI)

In [ ]:
```
pyplot.savefig('temp_vs_mosquitos.pdf')
pyplot.close()
```

savefig will save to the current directory unless you give it a path. It is smart enough to figure out your file format from the extension of the file you save. You can also save png files, jpg files, ps files, etc.

Let's add this to our plot_data function

In [ ]:
```
def plot_data(x, y, symbol = 'o'):
    '''
    This code will plot the arrays in x and y with symbol (default "o")
    '''
    pyplot.plot(x, y, symbol)
    pyplot.savefig('temp_vs_mosquitos.pdf')
    pyplot.close()
```

Finally, let's run our whole program

In [ ]:
```
year, temperature, rainfall, mosquitos = read_csv_file('mosquito_data_A1.c
temp_in_c = convert_fahrenheit_to_celsius(temperature)
plot_data()
```

Moving out of the notebook

- In the shell cd to the intermediate/python directory
- open nano
- Copy and paste your functions into nano. At the bottom, not indented, put the calls to the functions
- Save your file as plot_temperature.py
- start an ipython session
- import plot_temperature.py
- from the shell type python plot_temperature.py

Show if **name** == "**main**":

Repeat:

```
import

 run from shell
```

In [ ]: