Lecture 03: Modules in Python

We will start out with the **`math` module**. At the end, we will introduce the **`NumPy` module**. This will be the workhorse of most of our numerical calculations.

Modules are one of the most powerful (and initially confusing) aspects of Python. This will show you more about how to use them.

At a basic level Python is a somewhat "small" scripting language. It's small in the sense that it doesn't come with a lot of native commands to do things we normally want.

For example, perhaps the most fundemantal aspect of linear algebra is the ability to *linearly combine* vectors; i.e.,

$$v_1 = [x_1, y_1, z_1], \quad v_2 = [x_2, y_2, z_2] \quad \text{and} \quad c_1 v_1 + c_2 v_2 = [c_1 x_1 + c_2 x_2, c_1 y_1 + c_2 y_2, c_1 z_1 + c_2 z_2]$$

What happens if we try this in Python? Remember, lists aren't vectors.

```
In [1]: v0 = [1,2,3]
        v1 = [4,5,6]
        v0 + v1

Out[1]: [1, 2, 3, 4, 5, 6]
```

We already found that adding list together just gives a longer list. Just like if you "add" more courses to your course list this semester. This is very useful for some of the things Python was built for. But not helpful for numercial linear algebra. What about scalar multiplication?

```
In [2]: 3*v0

Out[2]: [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Well, it doesn't complain. And it behaves consistently with the way list add.

```
In [3]: 3*v0 == v0 + v0 + v0

Out[3]: True
```

But don't try this:

```
In [4]: 3.3*v0
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-4-5349aa74e44c> in <module>()
----> 1 3.3*v0

TypeError: can't multiply sequence by non-int of type 'float'
```

Or this:

```
In [ ]: v0*v1
```

There are simple ways around the problem. A loop will work.

```
In [ ]: v2 = []
        c0, c1 = 3.1, 2.7
        for i in range(len(v1)):
            v2.append(c0*v0[i]+c1*v1[i])
        print(v2)
```

But, I've been telling you that Python is *simple*. This is not simple. What a waste! Everytime you want to do something simple you have to go to all this trouble. Well, you could make a function!

```
In [ ]: def linearly_combine_lists(c0,c1,v0,v1):
            if len(v0) != len(v1): return 'the lists need to be the same length!'
            v2 = []
            for i in range(len(v0)):
                v2.append(c0*v0[i]+c1*v1[i])
            return v2
```

And it works.

```
In [ ]: linearly_combine_lists(c0,c1,v0,v1)
```

But it still seems like a big fuss. And Python loops are slow compared to compiled C code. If you are going to need to fuss everytime you want to add things, you might as well use C, which at least runs fast.

What if there was a better way?

There is! Just the same way *you* can make new functions to do useful things, everyone else can too!

And it turns out there are a huge number of really smart people out there who love making things in Python. So if someone comes up with a really efficient way to add vectors, multply matrices, and solve linear equations, they might put them into a big library (aka module) and let you use it as if it were native Python code.

HOW MODULES WORK

So far, everything we've done has used the simplest operations (e.g., + , - , * , *, * / , % , //) on simple data such as int and float. You can check out the Python "Standard Module" at:

https://docs.python.org/3/library/ (https://docs.python.org/3/library/)

It's pretty good documentation.

Python also has a hanful of modules than come with a standard instalation, but are initally hidden. After that, there are a huge collection of third-party modules that come indivudually.

We know we are going to want functions like sin cos, log, etc. Let's take a look at that. To prove we don't have those functions:

```
In [ ]: print(sin(0))
```

METHOD 0

First, I'll show the "BAD" way to import a module, and why. Then I'll show you the good way to do it. And then I'll show you some more other good ways.

"math" is one of the modules that comes with a standard Python install. Here's the syntax for getting its entire list of functions and data:

```
In [ ]: from math import * # DON'T ACTUALLY DO THIS IN YOUR CODE!
```

Everything in the math module is now in our enviroment! For example,

```
In [ ]: print(pi,e)
        print(sin(pi))
        print(cos(pi))
        print(log(e))
```

But this is the BAD way to do it! For several reasons. Two bigs reasons are:

```
First: You will often want to use several at the same time.
        The different modules can have functions and data called the same thing.
        Dumping a bunch of new functions and data in your enviroment can clobber lo
ts
        of things you want to leave alone. Perhaps you want a variable named "e" th
at is
        not Euler's constat? Too bad, you are getting Euler's constant!

Second: You might only want a small number of things from a module, not everything.
        This can have memeory advantanges, and just remove clutter.
```

You *will* have Euler's constant!

In [ ]:
```
e = 'I like pie.'
print(e)
from math import *
print(e)
```

Now that we have a ton of functions in our enviroment in ways we don't understand, the best thing to do is **QUIT THE KERNAL**. This will clear everything out of memory and start again. It's the circle arrow button between the square "stop kernal" button and the Code vs Markdown dropdown menu.

Assuming you **QUIT THE KERNAL**, let's keep going.

In [ ]:
```
e
```

METHOD 1

A much better way to import the contents of math is to give it a unique name in our main enviroment. Then there is a way to refer to the *enviroment* 'math', without it just living in ours.

The way to do this is the following:

```
import math as a_name_you_choose
```

The standard best practice is to make "a_name_you_choose" a two-letter word that describes the module. In some cases, these two-letter words are almost universal.

Now let's do it right:

In [ ]:
```
import math as mat
```

If you are not interested in renaming the module to something shorter, you could also just do

```
import math
```

This would be the same as

```
import math as math
```

Now `mat` is just a *new enviroment* that plugs into the main enviroment.

```
In [ ]: mat
```

Recall that the output looked similar when we were working with functions last week.

When we did form math import *, we could use sin, cos, log, right away. Now we can't without first asking `mat` for them.

```
In [ ]: sin(pi)
```

When asking for "a_function" from mat, we need to use "mat.a_function". For example:

```
In [ ]: print(mat.pi)
        print(mat.e)
        print(mat.sin(mat.pi))
        print(mat.cos(mat.pi))
        print(mat.log(mat.e))
```

```
In [ ]: def Sin(x): return mat.sin(x)
```

`mat` is a name in our `main` enviroment. `mat` is also its own enviroment, with names of things that it knows. We ask for these things with the `.` operators. It works the following way:

```
name_of_thing_in_main_enviroment . specific_thing_we_want_in_subenviroment
```

I put spaces around the "`.`"; it works this way but is not customary.

METHOD 2

Calling everything ma.sin, ma.cos, ma.pi, etc might seem like overkill sometimes. Occasionally, you might just want to use a single function, or two, from a module and not have to worry about being super careful about the names.

In that case, you could just import what you need.

```
In [ ]: from math import sin, cos, pi
```

You could also give it your own name if you want

```
In [ ]: from math import sin as math_sine
```

Now you have sin, cos, and ma.sin, and ma.cos available in your enviroment. These are duplicates of eachother, but the interpreter doesn't know that and will keep them seperate just as if they were different.

```
In [ ]: print(sin(mat.pi))
        print(mat.sin(pi))
        print(math_sine(pi))
```

But rather than importing the same thing twice, you will often want to import another module that has a lot of the same things in it. For example:

```
"NumPy": is going to be one of our go-to modules. This is the module that
has a lot of the useful numerical linear algebra that we will use later.

"SciPy": is a lot like numpy. We will also use this one a lot.
```

These two modules have a lot of features in common, but sometimes use different algorithms with work better or worse depending on the application. Numpy and Scipy are third-party modules that *do not* come with a standard Python install.

```
In [ ]: import numpy as np
        import scipy as sp
```

We will always reserve `np` and `sp` as short names for `numpy` and `scipy`.

Now we have a lot of different sine functions. But a lot of things that aren't the same.

```
In [ ]: print(mat.sin(1))
        print(sp.sin(1))
        print(np.sin(1))
```

Numpy basics

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

Remember what got this whole thing started? Linearly combining lists. Numpy can do this. It has a data type called and ***Array***.

In [ ]:
```
c0, c1 = 3.1, 2.7
v0 = np.array([1,2,3])
v1 = np.array([4,5,6])

print(v0)

v2 = c0*v0 + c1*v1
v2
```

Numpy arrays have a lot of the properties of lists. But you can add them as vectors. We can also multiply arrays, element-by-element.

In [ ]:
```
v3 = v0*v1
v3
```

We won't test it now, but I can assure you that `Numpy`'s array add is much much faster than the for loop over lists that I created at the beginning. Using `Numpy` and `Scipy` is how you can make a lot of Python as fast as C code. That is what `Numpy` and `Scipy` usually do. The hand off your calculation to optimised C code that's running under Python's interface.

There are 3 common ways to create arrays in numpy. We just saw the first way; i.e., pass it a list with numbers.

We can also make a list of integers in the following way:

In [ ]:
```
x = np.arange(10)
print(x)
```

Or, with a small twist:

In [ ]:
```
x = np.arange(3.1,12.5)
print(x)
```

The numbers can start on a non-integer. And they can increase by any amount you want. But you might not make it to the end if things don't fit right.

In [ ]:
```
x = np.arange(3,14,0.21)
print(x)
```

Or, conting by 2's:

In [ ]:
```
x = np.arange(3,15,2)
print(x)
```

All numpy arrays come with a *attribute* call "shape". It tells you the size, and shape.

In [ ]:
```
x.shape
```

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

It's a little piece of metadata attached to the arry that tell Python how to interpret the list of numbers. You can also change the metadata. It won't change the underlying data, but will chage the way Python sees it.

Below are all different sized matrices.

In [ ]:
```
x.shape = (3,2)
print(x)
print(x[0])
print(x[1])
print(x[2])

print(x[:,0])
```

Now that x is shaped like a matrix, you can do matrix multiplication with vectors.

In [ ]:
```
a = np.array([11,9])
print(x.dot(a))
```

It will work for any arrays of the right shape

In [ ]:
```
b = np.array([12,2,-2])
print(b.dot(x))
```

In [ ]:
```
x.dot(b)
```

In [ ]:
```
x.shape = (2,3)
print(x)
```

In [ ]:
```
print(x.dot(b))
print(a.dot(x))
```

In [ ]:
```
x.shape = (1,6)
print(x)
```

In [ ]:
```
x.shape = (6,1)
print(x)
```

Notice the difference between (1,6), (6,1) and (6,). The first two are 1×6 and 6×1 matrices respectively. The other is a 6-dimensional vector.

Back to where we started.

In [ ]:
```
x.shape = (6,)
print(x)
```

"`linspace`" is another alternatiece to arrange. For example,

In [ ]:
```
x = np.linspace(3.2,5.3,num=36)
print(x)
print(x.shape)
print(len(x))
```

`linspace` gives you very good control over the array. You get (in this example) an array with exactly num=36 elements; the first element is exactly 3.2, the last element is exactly 5.3, and the elements in between are evenly spaced.

After you've made the array, it's the same data type as if you've made it any other way.

We can turn this into a square matrix.

In [ ]:
```
x.shape = (6,6)
print(x)
print(len(x))
```

For matrices, there is a short-hand multiplaication operator

In [ ]:
```
x @ x == x.dot(x)
```

Not the same a element-by-element multiplication

In [ ]:
```
x*x == x.dot(x)
```

In [ ]:
```
print(x*x)
print(x @ x )
```

In [ ]:
```
x.shape = (36,)
```

Bare-minimum Matplotlib

Matplotlib is a giant set of functions for plotting. It can be very confusing at times. But plotting and data visusilation is a difficult thing to get just right for many reasons. Right now, I want to justlet you know that it exist and can make simple plots.

In [ ]:
```
import matplotlib.pyplot as plt

# We need to say some magic words to get the plots to show up.
%matplotlib inline
```

In [ ]:
```
y = np.sin(4*x)
plt.plot(x,y)
plt.plot(x,y**2)
plt.plot(x,y**3,'r')
```

In [ ]:
```
plt.plot(x,x-np.cos(8*x))
```

We'll come back to `matplotlib` and `pyplot` in a few lectures. For now, this is enough to get by.