May 15, 2023 – Hands-On Intro to the Basics of Scientific Programming
Prof. Joshua Weitz and QBioS Cohort

Would you like to be able to build a computational model of a living system but don't know how? Then you are in the right spot.

This tutorial is meant to be interactive...that is *you* should be reading, typing, thinking, and asking questions. There are no source files to share, the point is for you to think about and enter the code and see the result. And then, to vary the code and see what changes. This is a more effective strategy than just *copying* code others have written and running it.

The material for this introductory tutorial are adapted from a semester long course entitled "Foundations of Quantitative Biosciences" developed by Prof. Joshua Weitz in Fall 2016/2017/2018 as the cornerstone class for the QBioS Ph.D. at Georgia Tech.

Today we will focus on the basics of coding that can help you build models... whether of gene expression, protein dynamics, or some other problem linked to dynamics of living systems. A few notes before we get started:

- If you are experienced in coding, then please feel free to skip directly to Tutorial 1.
- Please work on your own computer, save files for future reference, but also work with a partner so you can discuss and help each other as you go.

Let's get started!

1 Getting Started

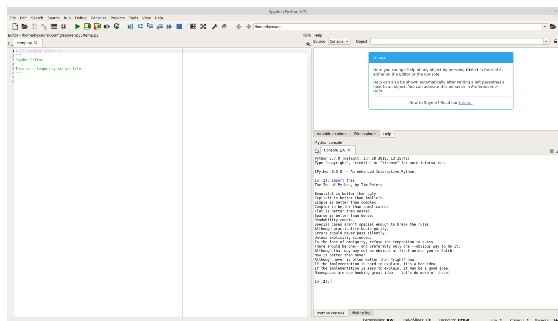
Python is a dynamic programming language with a vast region of applications due its diverse library of packages. We will be using it in this class to simulate and model the dynamics of living systems from scales of molecules to ecosystems. Python is particularly good at:

- Building Multi Purpose Applications
- Web Development
- Rapid Prototyping

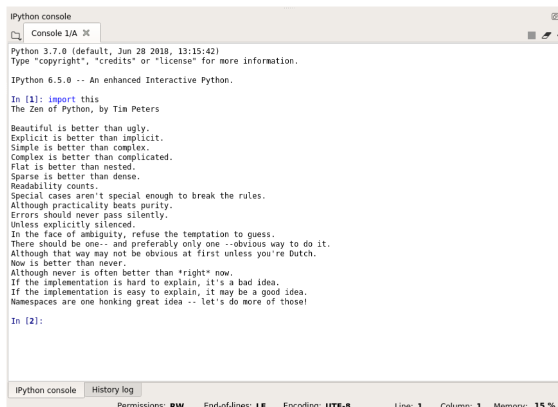
and is okay at:

- Interfacing with other languages
- Speed of for loops (as compared to C)

There are a variety of Integrated Development Environments(IDEs) for Python. For this course we will use Spyder. So, let's get started! In today's class, you will gain practical experiencing using Python. **If you are uncertain how to launch jupyter, try searching for "anaconda" using the search bar**. When you open the anaconda manager select Spyder, using the desktop link or application manager, then you should see a window that looks like this:



It is worth mentioning that you can use any python interface that you would like, but if you are unfamiliar with the language, we recommend using a jupyter notebook, google colab, or the Spyder inferface. We will be doing most of our work in the Console, the region to the bottom right.



The Console is where you enter commands, and you should see a prompt that looks like this:

```
In [ ]:
```

You can do basic math at this prompt (in order to run the cell press Shift+Enter), for example

```
In [ ]: 3+4
Out[ ]: 7
```

and (using the numpy package)

```
In [ ]: import numpy as np
        np.exp(1)
Out[ ]: 2.7182818284590451
```

Python's greatest strength is the diverse library of packages available to it. There are an incredible amount of packages that are useful for everything from 3D design to Genomics. Making use of this diverse set of packages requires that they be imported (loaded), into the current workspace. As long as the package is installed, Anaconda contains all the packages for this workshop, importing them is simple. Remember, that in every new session you must import the packages you wish to use.

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

Copyright 2023: Joshua Weitz – For teaching purposes only, do not copy, disseminate, or distribution without permission of the author.

In addition to importing with the **import** command, we also use the **as** command to shorten the name. From numpy to np. Once a package is imported, it is made available for every cell downstream. Therefore it is good practice to have import statements in the first cell. Python has several functions that are built in (and you can use it like a calculator). Numpy (a python package), has **many** more functions for advanced calculation, for example, to learn more about the exponential function:

```
In [ ]: np.info(np.exp)
 exp(x[, out])

 Calculate the exponential of all elements in the input array.

 Parameters
 ----------
 x : array_like
 Input values.

 Returns
 -------
 out : ndarray
 Output array, element-wise exponential of ‘x’.

 See Also
 --------
 expm1 : Calculate ‘‘exp(x) - 1’’ for all elements in the array.
 exp2  : Calculate ‘‘2**x’’ for all elements in the array.

 Notes
 -----
 The irrational number ‘‘e’’ is also known as Euler’s number.  It is
 approximately 2.718281, and is the base of the natural logarithm,
 ‘‘ln’’ (this means that, if :math:‘x = \ln y = \log_e y’,
 then :math:‘e^x = y’. For real input, ‘‘exp(x)’’ is always positive.

 For complex arguments, ‘‘x = a + ib’’, we can write
 :math:‘e^x = e^a e^{ib}’.  The first term, :math:‘e^a’, is already
 known (it is the real argument, described above).  The second term,
 :math:‘e^{ib}’, is :math:‘\cos b + i \sin b’, a function with
 magnitude 1 and a periodic phase.

 References
 ...
```

For numpy functions use np.info() , for base python packages use help()

Many functions have names that you expect (how do you think you should calculate cosine or sine of a value, for example)? Try it out! If you don’t know the name of the function you can use

Copyright 2023: Joshua Weitz – For teaching purposes only, do not copy, disseminate, or distribution without permission of the author.

the command np.lookfor or help.

Python is not just a calculator. It is also a programming language that can store values in memory. For example, the command

```
In [ ]: x=3
        x
Out [ ]: 3
```

tells Python that the variable x has the value 3 and now every time you use “x”, Python will substitute the value 3, for example:

```
In [ ]: y=x+1
        print(y)
Out [ ]: 4
```

Note that if you don’t want Python to report back the answer/value you can either call the variable or use a print() function. It is very important to realize the “=” sign does not mean that Python checks to see if the two sides are equal to each other, but rather states that whatever is on the left should be assigned the value of that on the right. If you want to check the truth of a particular statement, for example is x equal to 3, or alternatively, is y equal to 3 then you would type:

```
In [ ]: x==3
Out [ ]: True

In [ ]: y==3
Out [ ]: False
```

The double “==” sign tells Python to logically compare what is on the left with that on the right and return 1 if true and 0 if false.

Python can also handle arrays of values (for example a vector or a matrix). The simplest way is to use the numpy arange function, which defines a sequential vector, for example

```
In [ ]:v = np.arange(1,5)
       print(v)
Out [ ]: [1 2 3 4]
```

you can also modify the increments by adding a step parameter at the end

```
In [ ]: w = np.arange(1,9,2)
        print(w)
Out [ ]: [1 3 5 7]
```

Any entry can be examined using the brackets

```
In [ ]: w[3]
Out [ ]: 7
```

and basic math can be performed automatically on vectors (and matrices), for example

```
In [ ]: np.log(w)
out [ ]: array([ 0.        ,  1.09861229,  1.60943791,  1.94591015])
```

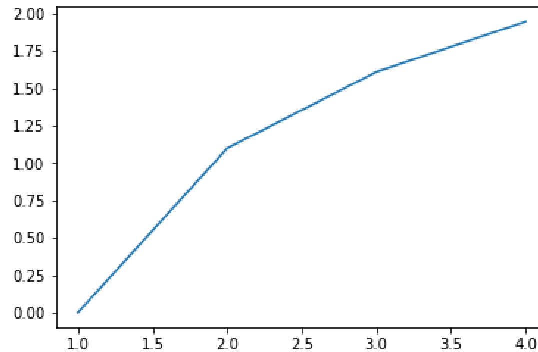
Copyright 2023: Joshua Weitz – For teaching purposes only, do not copy, disseminate, or distribution without permission of the author.

Python lists/arrays are 0 indexed, meaning the first element's index is 0 not 1

Python using the Matplotlib library can also plot graphs and surfaces and all sorts of things. To create a simple plot, use the plot command.

```
In [ ]: import matplotlib.pyplot as plt
        plt.plot(v,np.log(w))
```

which leads to:



2 Building "Programs" from "Scripts" and "Functions"

However, once you have a lot of commands, it will get exhausting typing them (especially when you make mistakes). So, Python has functions, which is a list of commands, that execute in order. To create a function create a new cell and define a function using def.

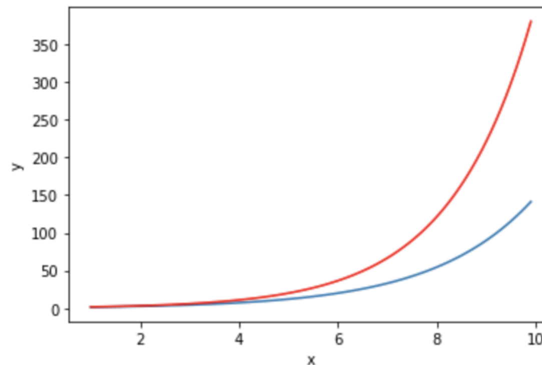
```
# My first function
def func():
    x = np.arange(1,10)
    y1 = np.exp(x*.5)
    y2 = np.exp(x*.6)
    plt.plot(x,y1)
    plt.plot(x,y2,color='red')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.show()
```

Run the cell this is placed in, and then in the next cell put.

```
In [ ]: func()
```

and it should do all the commands in your script and give you your figure.

Copyright 2023: Joshua Weitz – For teaching purposes only, do not copy, disseminate, or distribution without permission of the author.



The problem with this function is that if you wanted to change the exponents in these files, you would have to edit the function and then re-run it, instead of designating the change in the current cell. Moreover, this function does not return a variable or accept a variable as input.

```
def logGrowth(t,N):
    '''
    logGrowth gives the growth rate of a population of size N at time t
    '''
    r = .5
    K = 100
    dNdt = r*N*(1-N/K)
    return(dNdt)
```

Now you can use your function like one of Python's, for example:

```
In [ ]: j = np.arange(110)
        plt.plot(j,logGrowth(0,j))
        plt.xlabel('N')
        plt.ylabel('dN/dT')
        plt.show()
```

gives an upside down parabola, denoting that growth rate is positive between 0 and 100 and negative when N is greater than 100.

3 Getting Started with Core Techniques

3.1 Create loops

The two types of loops we will discuss are "for" and "while" loops. Both loop start with a keyword such as for or while and they end with the word end. The "for" loop allows us to repeat certain commands many times with a "counter" variable. Here is one example:

```
In [ ]: for i in np.arange(1,4):
            print(1-i)
```

You can also increment your counter variable by any real number like

Copyright 2023: Joshua Weitz – For teaching purposes only, do not copy, disseminate, or distribution without permission of the author.

```
In [ ]: for i in np.arange(1,0,-.1):
            print(1-i)
```

Or you can even use a set of arbitrary values:

```
In [ ]: for i in [1,3,5,7]:
        print(1-i)
```

Challenge problem: Exercise on matrices

Define a random matrix A of size 3-by-3. Use a double "for loop" to calculate the square of the entries in A and store the values in another matrix B. (Hint: type np.lookfor('random') if you don't know how to define a random matrix)

The second type of loop is the "while" loop. The "while" loop repeats a sequence of commands as long as some condition is met. For example, given a number n, the following code will return the smallest non-negative integer a such that $2^a \geq n$.

```
In [ ]: def smallexp(n):
            a = 0
            while 2**a < n:
                a = a+1
            return(a)
In [ ]: a = smallexp(4)
        print(a)
Out [ ]: 2
```

Note that in the above example we used the conditional statement, $2^a < n$ to decide whether the statement within the while loop should be proceeded. Such conditional statements are also used in "if" statements that are discussed in the next section. The rational operator used here is "$<$", which means "less than." Other rational operators that are available in Python include:

```
>  greater than
>= greater than or equal
<= less than or equal
== equal
!= not equal
```

Simple conditional statements can be combined by logical operators 'and' and 'or' into compound expressing such as the following:

```
(5 > 1) and (5 == 6)
(5 > 1) or (5 == 6)
```

3.2 Make decisions

Now, let's suppose you want your code to make a decision, and the "if" statement is what you need. The general form in Python is as follows:

Copyright 2023: Joshua Weitz – For teaching purposes only, do not copy, disseminate, or distribution without permission of the author.

```
if expression1:
  statements1
elif expression2:
  statements2
else:
  statements3
```

Challenge problem: Recursive factorials

Exercise: Finish the following pseudo-code that gives the factorial of a positive number n, using the recursive formula $n! = n(n-1)$.

```
def factorial_recur(n):
    if #:
        N = 1
    else:
        N = #
    return(N)
```

3.3 Go fast, i.e., "vectorize"

Remember: `for` loops are SLOW. One way to make your Python code run faster is to "vectorize" the algorithm you use in the code. Vectorization can be done by converting for and while loops to equivalent vector or matrix operations. A simple example would be the following timeit is a magic command that captures the runtime of the cell in jupyter. Commands on the same line as timeit are not included in the timer. The double percent sign is for jupyter notebook.

```
In [ ]: %%timeit x = 1 ; y = []
        for i in np.arange(1001):
            y.append(np.log10(x))
            x = x + .01
In [ ]: %%timeit
        x = np.arange(.01,10,.01)
        y = np.log10(x)
```

For more complicated code, vectorization is not always so obvious. Nonetheless, there is a simple procedure that will help: elementwise operators.

3.4 Elementwise operators

In Python, you can do calculations element by element. For example, if you have a variable `X` and want to square it. In Python squaring is denoted by ** instead.

```
In [ ]: x = np.array([5,5,5,5])
        x^2
```

Copyright 2023: Joshua Weitz – For teaching purposes only, do not copy, disseminate, or distribution without permission of the author.

Did it work? Yes! In numpy elementwise multiplication it is the default behavior to interpret the subsequent operator element by element. This is also helpful for a whole matrix. If you would like to do dot multiplication, simply use @ operator

```
In [ ]: x = 5 * np.ones((4,4))
        x @ x
```

Contrast the two answers. In the first, you squared all the 5-s. In the second you multiplied two square matrices by each other.

Challenge problem: Element-wise operations: square root

Take the square root of the numbers between 1 to 20 using an elementwise operator.

3.5 Find values you want to know about

Given an array X, the command

```
In [ ]: x[x!=0]
```

returns the indices of all nonzero elements of X. You can also use a logical expression to define X. For example,

```
In [ ]: x[x>2]
```

returns the indices corresponding to the entries of X that are greater than 2. Notice that X could also be a matrix. In this case, using the np.argwhere() returns a list of the elements that satisfy the condition.

```
In [ ]: np.argwhere(x>2)
```

However, when you want to locate the entries that satisfy more than one logical expressions, the 'elementwise' logical operators (& and |) are used in place of 'short-circuit' logical operators (&& and ||).

Challenge problem: Finding entries in matrices

Exercise: Build a 5-by-5 random matrix A using the command

```
In [ ]: A = np.random.rand(5,5)
```

Find the linear indices of entries whose value is smaller than 1/4 and bigger than 1/6. Check the answer with your neighbors.

3.6 Save and load data

Saving and loading objects (variables,arrays, or other forms of data) can be done in a multitude of ways in Python, however; for this course we will use the native methods for numpy.

Copyright 2023: Joshua Weitz – For teaching purposes only, do not copy, disseminate, or distribution without permission of the author.

```
In [ ]: import numpy as np
        numpy_array = np.array([10,20,30,40])
        np.save('numpy_store.npy',numpy_array)
```

and numpy will save the specified array in the file name. When you want to load all the variables from the file specified by filename, just type

```
In [ ]: numpy_loaded = np.load('numpy_store.npy')
```

Copyright 2023: Joshua Weitz – For teaching purposes only, do not copy, disseminate, or distribution without permission of the author.

4 Solutions to Challenge Problems

This section includes solutions... try to figure this out on your own before you look! Really! You can do it!

Solution to Challenge Problem: Exercise on Matrices.

```
In []:  #Define a random matrix A of size 3-by-3
         A = np.random.rand((3,3))
         #Initialize an empty matrix B
         B = np.zeros((3,3))
         #Use a double "for loop"
         for i in np.arange(3):
             for j in np.arange(3):
                 #calculate the square of the entries in A
                 sq = A[i][j]**2
                 #and store the values in another matrix B
                 B[i][j]=sq
```

Solution to Challenge Problem: Recursive Factorials. The solution to this recursive problem involves thinking about what happens in a sequence. For example, if 3! = 321 that is equivalent to 3! = 32! = 3 2 1! = 3 2 1. In essence, by breaking the big problem into little problems of the same kind it is possible to get to the point where the answer is known and then reconstruct the solution – in the formula above it would be moving from left to right and back again. Here is one such solution. Note that one would have to add error correction to make sure there are not issues if the user calls the functions with a real number or a non-positive integer.

```
In []: def factorial_recur(n):
        """
        function N = factorial_recur(n)
        Returns the factorial of n
        """
        if n==1:
           N=1
        else:
           N = n*factorial_recur(n-1)
        return N
```

Solution to Challenge Problem: Element-wise operations: square root.

```
In []: import numpy as np
        a = np.arange(1,21,1)
        print(a)
        b = np.sqrt(a)
        print(b)
```

Copyright 2023: Joshua Weitz – For teaching purposes only, do not copy, disseminate, or distribution without permission of the author.

Solution to Challenge Problem: Finding Elements in Matrices. Given that `A=np.random.rand(5,5)` generates a 5×5 matrix, finding indices can be done in a few ways. The most direct way is to use the find command, as follows:

```
In []:  ind = np.argwhere((A>(1/6)) & (A<(1/4)))
```

Note that this answer can be empty if there are no elements that satisfy the condition.

Copyright 2023: Joshua Weitz – For teaching purposes only, do not copy, disseminate, or distribution without permission of the author.

Solution to Challenge Problem: Variables and Differential Equations. Although some variables may be fixed in a given system of equations, there are many circumstances where it is necessary to probe the response of a system to variation in such conditions. In this case, it is possible to pass variables to the odeint command, which will, in turn, pass these variables on the function specified to actually compute the rate of change in the system of equations. It's best seen by example. First, here is a modified code that generates a structure of parameters called `pars` and sets up a loop to change the value of K:

```
In []: def logGrowth_withpars(N,t,pars):
              """
              function dNdt=logGrowth(N,t,pars)
              logGrowth gives the growth rate of a population of size N at time t
              pars is a dictionary that contains parameters
              """
              r = pars['r']
              K = pars['K']
              dNdt = r*N*(1-N/K)
              return dNdt
```

Next, here is a script that calls this new function multiple times, each with a new value of K and then plots the dynamics on the same axes.

```
In []: import numpy as np
        import matplotlib.pyplot as plt
        from scipy import integrate
        from Lab1_Functions import logGrowth_withpars

        #Parameters
        t0=0 #Initial Time
        tf=50 #Final time
        tVec = np.linspace(t0,tf)
        N0 = 1 #Initial population size
        pars={}
        pars['r']=0.5;
        Krange = np.logspace(2,3,num=5)

        for i in range(len(Krange)):
             pars['K']=Krange[i] #Set the value of K
             vNint = integrate.odeint(logGrowth_withpars,
                         N0,
                         tVec,
                         args=(pars,))
             plt.plot(tVec,vNint,color=np.array([0.75,0.75,0.75])*i/len(Krange),linewidth=3)
             plt.text(30,pars['K']-50,'K = {K}'.format(K=int(pars['K'])),fontsize=10)
plt.ylim([0,1100])
```

Copyright 2023: Joshua Weitz – For teaching purposes only, do not copy, disseminate, or distribution without permission of the author.