May 17-18, 2021 – Hands-On Intro to the Basics of Scientific Programming
Prof. Joshua Weitz, Prof. James C. Gumbart, and the 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. The materials are adapted from a semester long course entitled ”Foundations of Quantitative Biosciences” developed by Prof. Joshua Weitz in Fall 2016/2017/2018/2019/2020 as the cornerstone class for the QBioS Ph.D. at Georgia Tech. The materials have been adapted to focus on epidemics modeling and to account for the greater variety of backgrounds of students in the workshop.

Today we will focus on the basics of coding that can help you build models... whether of gene expression, cellular dynamics, game theory, or some other problem linked to dynamics of living systems. Let's get started!

1 Getting Started

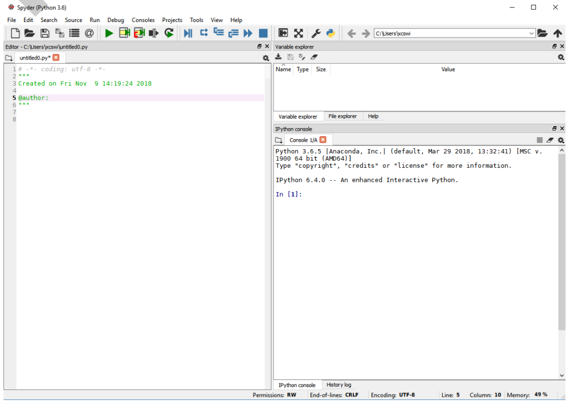
Python is a dynamic programming language with a vast region of applications due to its diverse library of packages. We will be using Python in this laboratory guide to simulate and model the dynamics of living systems from molecules, cells, organisms, to ecosystems. Python is particularly good at:

- 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, but environments such as the iPython notebook also allow for interactive work. So, let's get started! This chapter will help you gain practical experience using Python. When you open Spyder, using the desktop link or application manager, you should see a window that looks like this:


Spyder (Python 3.6)
File Edit Search Source Run Debug Consoles Projects Tools View Help
C:\Users\xcswi
Editor - C:\Users\xcswi\untitled0.py
untitled0.py*
Variable explorer
Name Type Size Value
Variable explorer File explorer Help
IPython console
Console 1/A
IPython console History log

```
# -*- coding: utf-8 -*-
"""
Created on Fri Nov  9 14:19:24 2018

@author:
"""
```

```
Python 3.6.5 |Anaconda, Inc.| (default, Mar 29 2018, 13:32:41) [MSC v.
1900 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 6.4.0 -- An enhanced Interactive Python.

In [1]:
```

Permissions: RW End-of-lines: CRLF Encoding: UTF-8 Line: 5 Column: 10 Memory: 49 %

The key elements are the Command Window, File Editor, and Help Window. The command window is where you enter commands, and you should see a prompt that looks like this:

```
In [1]:
```

You can do basic math at this prompt, for example

```
In [1]: 3+4
Out[1]: 7
```

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 in this tutorial) importing them is simple. Remember, that in every new session you must import the packages you wish to use. For example

```
[obeytabs,tabsize=4]
In [ ]: import numpy as np
```

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 all following code in the session. Therefore it is good practice to have import statements at the beginning of your code. With the Numpy package imported, we may do calculations such as

```
[obeytabs,tabsize=4]
In [ ]: np.exp(1)
Out[ ]: 2.7182818284590451
```

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.exp?
Call signature:  np.exp(*args, **kwargs)
Type:            ufunc
String form:     <ufunc 'exp'>
Docstring:
exp(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, 

Calculate the exponential of all elements in the input array.

Parameters
----------
x : array_like
    Input values.
out : ndarray, None, or tuple of ndarray and None, optional
    A location into which the result is stored. If provided, it must have
    a shape that the inputs broadcast to. If not provided or ‘None‘,
    a freshly-allocated array is returned. A tuple (possible only as a
    keyword argument) must have length equal to the number of outputs.
where : array_like, optional
    Values of True indicate to calculate the ufunc at that position, values
    of False indicate to leave the value in the output alone.
**kwargs
    For other keyword-only arguments, see the
    :ref:‘ufunc docs <ufuncs.kwargs>‘.

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
 ...
```

Our console is an IPython console. Adding "?" after a function will give more information. In other consoles use help(function name). 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 the command np.lookfor() or help(). There is also an abundance of documentation on the internet.

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

```
[obeytabs,tabsize=4]
In [ ]: x=3
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
Out[ ]: 4
```

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. Instead, Python interprets the = sign to assign the value on the right to the variable on the left. If you want to check the truth of a particular statement – 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. Note that if you want Python to report back the answer/value you can either call the variable or use a print() function.

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

```
[obeytabs,tabsize=4]
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

```
[obeytabs,tabsize=4]
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
```

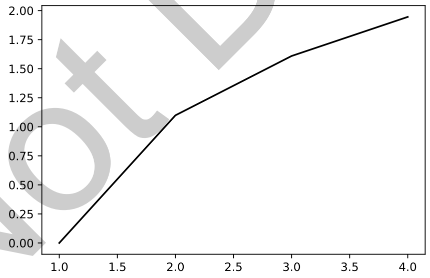
and basic math can be performed automatically on vectors (and matrices), for example

```
In [ ]: np.log(w)
out [ ]: array([0.69314718, 1.79175947, 2.30258509, 2.63905733])
```

Python using the Matplotlib library can also plot graphs, surfaces, and more. To create a simple plot, use the plot command

```
[obeytabs,tabsize=4]
In [ ]: import matplotlib.pyplot as plt
In [ ]: plt.plot(v,np.log(w))
```

which leads to:



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

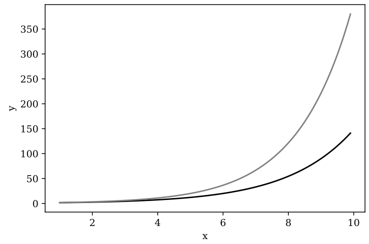
Once you have a lot of commands, it will get exhausting typing them again and again (especially when you make mistakes). Instead, you will want to use a "script". A script is a list of commands in a file that you can execute directly from the command window. To create a script go to the File menu and select New > File. Now type in a few commands, such as:

```
#my_first_file.py
import numpy as np
import matplotlib.pyplot as plt


# Create some vectors
x = np.arange(1,10,0.1)
y1 = np.exp(0.5*x)
y2 = np.exp(0.6*x)
```

```
#plot the vectors
plt.plot(x,y1,'k') #Use a black line
plt.plot(x,y2,c=[0.5,0.5,0.5]) #Use a grey line
plt.xlabel('x') #Label the axis
plt.ylabel('y')
#save the image to a file
plt.savefig('my_first.pdf',bbox_inches='tight')
```

Save this file as my_first_file.py in the same folder you're currently working in. Now click on the green play triangle at the top bar of Spyder and it should execute the commands in the script to yield the figure:



The problem with this script is that changing the arguments in the exponential functions requires editing the script and then re-running the code. It would be more convenient to designate a variable change from the command window and have the code automatically update its output. The problem is that a script cannot return a variable or accept a variable as input. To do so requires a "function". Functions are program files that can be called from the Command window, can accept inputs, and return outputs. To start one, open a new file and type:

```
def logGrowth(N,t):
    """
    function dNdt = logGrowth(N,t)

    logGrowth gives the growth rate of a population of size N at time t
    usage: dNdt = logGrowth(N,t)
    """
    r = 0.5
    K = 100
    dNdt = r*N*(1-N/K)
    return dNdt
```

Now the new function can be accessed just like one of Python's built-in functions, for example, type the following code into a new file named "Lab1s2.py". Note that to use code from another file, import the file

```
[obeytabs,tabsize=4]
import numpy as np
import matplotlib.pyplot as plt
from Lab1_Functions import logGrowth

vN = np.arange(0,110)
plt.plot(vN,logGrowth(vN,0))
plt.xlabel('N')
plt.ylabel('dN/dt')
```

This gives an upside down parabola, denoting that growth rate is positive between 0 and 100 and negative when N is greater than 100. Note that the argument `t` is not used in the `logGrowth` function. Not all inputs have to be used. We will update this function later to both accept and utilize all inputs.

3 Getting Started with Core Techniques

3.1 Create loops

There are many loops that enable the repetition of a fixed set of commands. The two central loops examined here are “for” and “while” loops. Both loops start with a keyword such as `for` or `while` and repeat the indented code. 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): #counting
print(1-i) #the statement you want to repeat
```

A counter variable can be incremented by any real number, e.g.,:

```
In [ ]: for i in np.arange(1,0,-.1): #counting
print(1-i) #the statement you want to repeat
```

For loops can also span an arbitrary set of values:

```
In [ ]: for i in [1,3,5,7]:
print(1-i) # the statement you want to repeat
```

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)

A “while” loop is the most robust of loops; formally speaking it is a universal loop such that all loops can be built using it. That’s a topic for some other class – for our purposes, both while and for loops may be appropriate depending on the circumstance. 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 $2^a \geq n$.

```
def smallexp(n):
    a = 0
    while 2**a < n:
        a = a + 1 # statement to execute if the condition is met
    return a
```

such that this command can be invoked from the Python interface: 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 repeated. Such conditional statements are also used in “if” statements. The relational operator used here is ”<”, which means “less than”. Other relational operators that are available in Python include:

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

Simple conditional statements can be combined by logical operators

```
(&, |, !)
```

into compound expressions such as the following:

```
(y > 1) & (x == 6)
```

3.2 Make decisions

Now, let's suppose you want your code to make a decision. In many circumstances, a series of `if` statements will suffice. The general form in Python is as follows:

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

Challenge Problem: Recursive Factorial

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

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

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

Remember: `for` loops are SLOW in Python. One way to make your Python 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 (`datetime.now()` is used as a stop watch):

```
import numpy as np
from datetime import datetime

startTime = datetime.now()

x=1
y=[]
for i in np.arange(1000):
    y.append(np.log10(x))
    x = x + 0.01

print("for loop:", datetime.now()-startTime)

startTime = datetime.now()

x = np.arange(1,11,0.01)
y = np.log10(x)

print("vectorize:", datetime.now()-startTime)
```

You should find that the second command set is faster, even if it returns exactly the same output. However, for more complicated code, vectorization is not always so obvious. Some of the most commonly used functions for vectorizing can be found in the Help browser of Python.

3.4 Find values you want to know about

Given an array X, the command

```
In [ ]: np.argwhere(X!=0)
```

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

```
In [ ]: np.argwhere(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, the function returns the linear indices as if the matrix was stored as a single column of elements (try it!). The ‘elementwise’ logical operators (& and |) can be used to locate the entries that satisfy more than one logical expressions. These commands will be important in solving the next challenge problem.

Challenge Problem: Finding Elements in Matrices

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

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

Find the indices of entries whose values are smaller than 1/4 and bigger than 1/6. Check the answer with your (virtual) neighbors or study partners.

3.5 Save and load data

Saving and loading objects (variables,arrays, or other forms of data) in Python is dependent on the Pickle package (it is typically better to save data in .csv files when possible and Python has good ways to write data to .csv files, but to save and load variables from the workspace, the Pickle package is the way to go).

```
[obeytabs,tabsize=4]
In [ ]: import pickle
object = 10
f = open(’store.pckl’, ’wb’)
pickle.dump(object, f)
f.close()
```

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

```
[obeytabs,tabsize=4]
In [ ]: f = open(’store.pckl’, ’rb’)
object = pickle.load(f)
f.close()
```

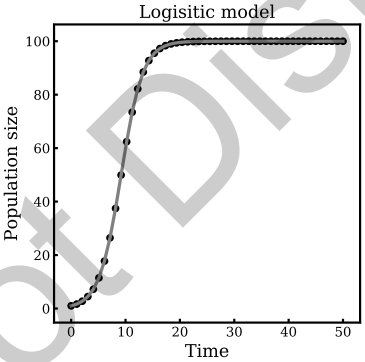
Please keep in mind that these file formats are binary and proprietary, i.e., not human-readable. If you want to save a particular file format use the help to save variables in standard, comma-separated value file format. There are also alternative ways to load and print data.

4 Numerically Integrating Differential Equations

In this class we can use Python to numerically solve differential equations, like the logistic growth equation, even when such solutions are not available analytically. The most-used Python program which does the integration is called `integrate.odeint` from the scipy package. Here is a script that integrates the logGrowth function. We will return to this multiple times in the course.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import integrate
from Lab1_Functions import logGrowth
#Numerical solution of the logisitic equation
t0 = 0  # Initial time
tf = 50 # Final time
N0 = 1  # Initial population size
T = np.linspace(t0,tf)  # time steps to report
vNint = integrate.odeint(logGrowth,N0,T)
# Actual solution
r = 0.5
K = 100
vNact = (N0*np.exp(r*T))/(1+N0*(np.exp(r*T)-1)/K) # Actual solution
# Plot results
plt.plot(T,vNint) # Plot numerically integrated solution
plt.scatter(T,vNact,color='red') # Plot actual solution
plt.xlabel('Time')
plt.ylabel('Population size')
plt.title('Logisitic model')
```

Run this script which yields the plot



Here are some important points to keep in mind:

- Python solves ordinary differential equations of the form

$$\frac{d\vec{y}}{dt} = \vec{f}(\vec{y})$$

  where $\vec{f}(\vec{y})$ is a vector of functions. The default solver used in these computational laboratories is **odeint**.

$$\text{YOUT} = \text{scipy.integrate.odeint(ODEFUN,Y0,TSPAN)}$$

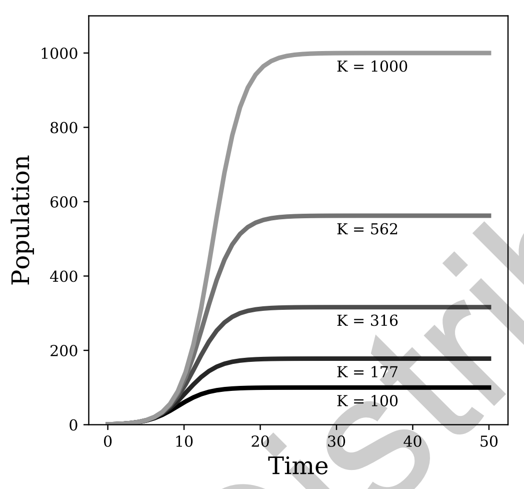
  where

  1. YOUT → value of variable
  2. ODEFUN → name of function containing the dynamics
  3. TSPAN → time limits for integration
  4. Y0 → initial conditions

and further documentation on `odeint` may be found online. The key idea is that Python has a built-in function that numerically integrates a differential equation that must be specified – by you. It integrates the system of equations over a range of time given initial conditions. Let's give this a try, albeit by going farther: accepting additional parameters as input to the dynamical system.

Challenge Problem: Variables and Differential Equations

Read the `odeint` documentation and modify your logistic growth model to accept the parameters r and K. Setting $r = 0.5$, vary the value of K over 1 order of magnitude and compare the results using a plot command. If your code works, the resulting figure should look something like this:



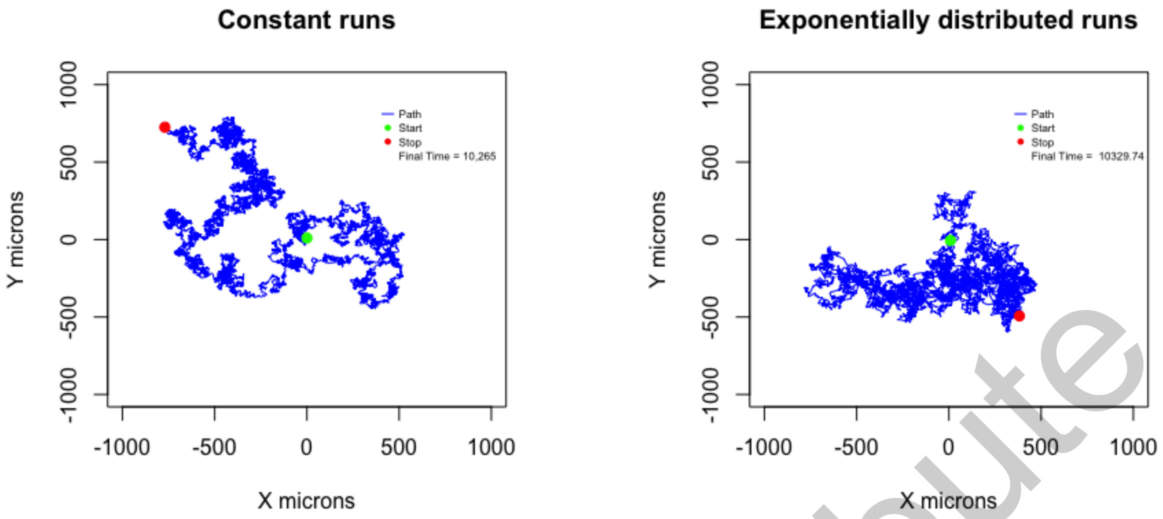
5 Advanced - Individual-Based Simulations

As a final challenge for those more experienced in Python, try and develop a simulation of "diffusion" in which a bacteria swims at a speed of 10 μm/s with each "run" lasting 1 second. In this case the direction of the next run is random. How long will it take, on average, for the bacteria to travel 1mm away from its source? As you may recall, the average time for a diffusing particle or organism to travel a squared distance x^2 on average should be:

$$T = \frac{x^2}{D} \tag{1}$$

It is true that any particular bacteria may travel much farther. What do we mean by *average* here? Indeed, the average distance traveled by bacteria is 0. That is the bacteria is just as likely to go to in any particular direction. But if one squares the distance from the origin, you will find that the average squared distance increases linearly with time.

If you develop code, can you visualize the results? If you change the length in equation, how does T change? Can you confirm the scaling and estimate D? What happens if the durations of each run is exponentially distributed, so that runs last on average 1 second, but can vary in duration? Did the answer with respect to travel time to reach 1mm change in a quantitative or qualitative way? We will get much further into these ideas later in the class. Here are examples of two runs, one with constant duration and one with exponentially distributed durations:



6 Take-home messages

- Building computational models of living systems requires a willingness to code, tinker, try, and try again.
- Expanding your computational toolset will take time.
- Mistakes are part of the learning process and indeed, will help reinforce common pitfalls.
- With time, your objective should be able to code for the most part without having to open a help forum or to try and search for the answer in your favorite search engine.
- The basic skills presented here are not comprehensive, instead they are a gateway to further exploration.
- The modules that follow build on a few of the methods in this laboratory, with a focus on direct integration of computational modeling as an integral part of the quantitative biosciences approach to science.
- You can do it! Really.