TACC
TEXAS ADVANCED COMPUTING CENTER
WWW.TACC.UTEXAS.EDU

TEXAS
The University of Texas at Austin

BigData-Hack-2022 Day 2

Let's do this!

PRESENTED BY:

Where do we go?
Where do we go now?

Look at each problem you are going to tackle, and figure out the requirements - what is needed to solve? Figure out the logic on how to solve it, and apply the algorithm.

Think about a Scientific Process

Let's meet Joe.

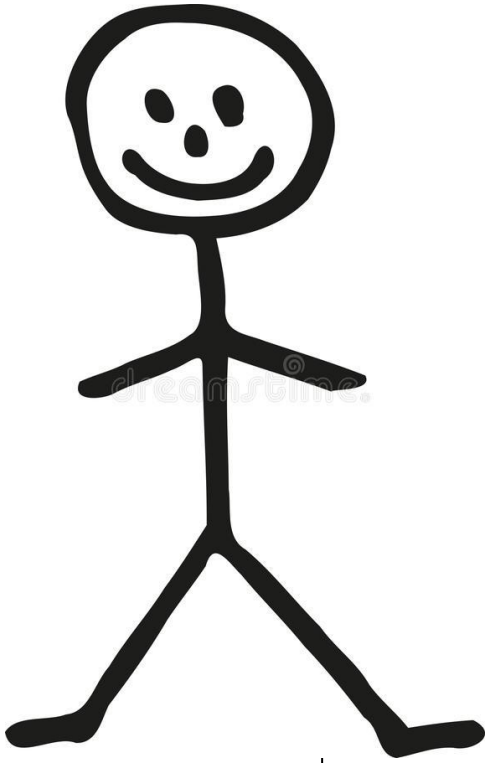
Joe might get sick.

Joe will be sick for 5 days.

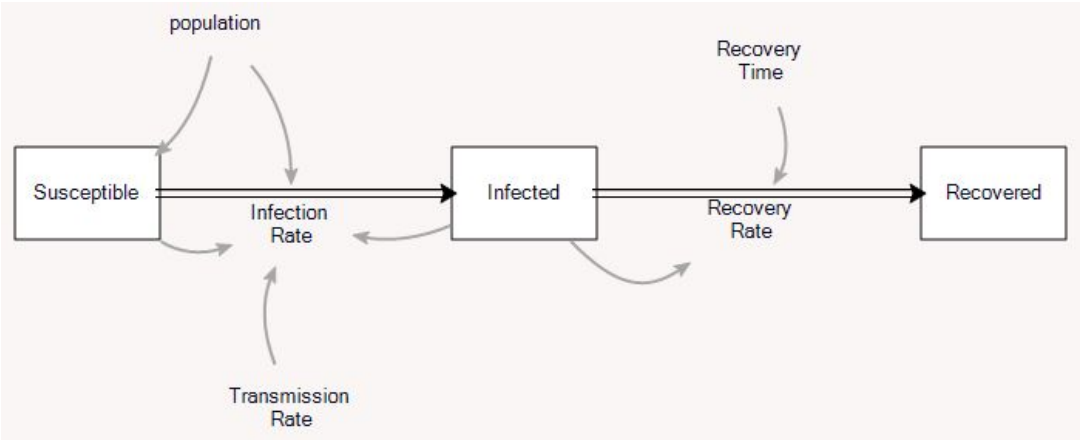
After 5 days, Joe gets better.

Once Joe gets better, Joe can no longer get sick.

How would we "code" Joe?



The SIR Model



Task 1 - Code Joe

Variables to hold data

Mathematical Operations to do math :)

Conditionals to make decisions

Loops to repeat our process

Functions/Subroutines to reuse code

Objects or Classes to define our "things"

Let's meet Joe.

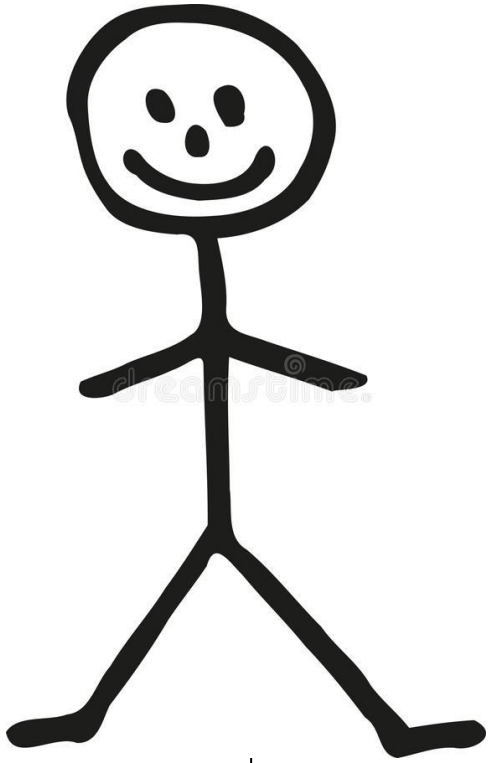
Joe might get sick.

Joe will be sick for 5 days.

After 5 days, Joe gets better.

Once Joe gets better, Joe can no longer get sick.

Let's "code" Joe.



What are Jupyter Notebooks?

A web-based, interactive computing tool for capturing the whole computation process: developing, documenting, and executing code, as well as communicating the results.

How do Jupyter Notebooks Work?

An open notebook has exactly one interactive session connected to a kernel which will execute code sent by the user and communicate back results. This kernel remains active if the web browser window is closed, and reopening the same notebook from the dashboard will reconnect the web application to the same kernel.

What's this mean?

Notebooks are an interface to kernel, the kernel executes your code and outputs back to you through the notebook. The kernel is essentially our programming language we wish to interface with.

Jupyter Notebooks, Structure

- Code Cells

  Code cells allow you to enter and run code
  Run a code cell using Shift-Enter

- Markdown Cells

  Text can be added to Jupyter Notebooks using Markdown cells. Markdown is a popular markup language that is a superset of HTML.

Jupyter Notebooks, Structure

- Markdown Cells

You can add headings:

```
# Heading 1
# Heading 2
## Heading 2.1
## Heading 2.2
```

You can add lists

```
1. First ordered list item
2. Another item
⋅⋅* Unordered sub-list.
1. Actual numbers don't matter, just that it's a number
⋅⋅1. Ordered sub-list
4. And another item.
```

Jupyter Notebooks, Structure

- Markdown Cells

```
pure HTML
  <dl>
    <dt>Definition list</dt>
    <dd>Is something people use sometimes.</dd>

    <dt>Markdown in HTML</dt>
    <dd>Does *not* work **very** well. Use HTML <em>tags</em>.</dd>
  </dl>

And even, Latex!

    $e^{i\pi} + 1 = 0$
```

Jupyter Notebooks, Workflow

Typically, you will work on a computational problem in pieces, organizing related ideas into cells and moving forward once previous parts work correctly. This is much more convenient for interactive exploration than breaking up a computation into scripts that must be executed together, as was previously necessary, especially if parts of them take a long time to run.

Jupyter Notebooks, Workflow

Let a traditional paper lab notebook be your guide:

Each notebook keeps a historical (and dated) record of the analysis as it's being explored.

The notebook is not meant to be anything other than a place for experimentation and development.

Notebooks can be split when they get too long.

Notebooks can be split by topic, if it makes sense.

Jupyter Notebooks, Shortcuts

- **`Shift-Enter`: run cell**
  - Execute the current cell, show output (if any), and jump to the next cell below. If `Shift-Enter` is invoked on the last cell, a new code cell will also be created. Note that in the notebook, typing `Enter` on its own *never* forces execution, but rather just inserts a new line in the current cell. `Shift-Enter` is equivalent to clicking the `Cell | Run` menu item.

Jupyter Notebooks, Shortcuts

- `Ctrl-Enter`: **run cell in-place**
  - Execute the current cell as if it were in “terminal mode”, where any output is shown, but the cursor *remains* in the current cell. The cell’s entire contents are selected after execution, so you can just start typing and only the new input will be in the cell. This is convenient for doing quick experiments in place, or for querying things like filesystem content, without needing to create additional cells that you may not want to be saved in the notebook.

Jupyter Notebooks, Shortcuts

- **`Alt-Enter`: run cell, insert below**
  - Executes the current cell, shows the output, and inserts a *new* cell between the current cell and the cell below (if one exists). (shortcut for the sequence `Shift-Enter`,`Ctrl-m a`. (`Ctrl-m a` adds a new cell above the current one.))
- **`Esc` and `Enter`: Command mode and edit mode**
  - In command mode, you can easily navigate around the notebook using keyboard shortcuts. In edit mode, you can edit text in cells.

Introduction to Python

Hello World!
Data types
Variables
Arithmetic operations
Relational operations
Input/Output
Control Flow

Do not forget:

Indentation matters!

Python

```
print(“Hello World!”)
```

Let's type that line of code into a Code Cell, and hit Shift-Enter:

```
Hello World!
```

Python

```
print(5)
print(1+1)
```

Let's add the above into another Code Cell, and hit Shift-Enter

```
5
2
```

Python - Variables

You will need to store data into variables
You can use those variables later on
You can perform operations with those variables
Variables are declared with a **name**, followed by '=' and a **value**

An integer, string,…
When declaring a variable, **capitalization** is important:
'A' <> 'a'

Python - Variables

in a code cell:

```
five = 5
one = 1
twodot = 2.0
print (five)
print (one + one)
message = “This is a string”
print (message)
```

Notice: We're not "typing" our variables, we're just setting them and allowing Python to type them for us.

Python – Data Types

in a code cell:

```
integer_variable = 100
floating_point_variable = 100.0
string_variable = “Name”
```

Notice: We're not "typing" our variables, we're just setting them and allowing Python to type them for us.

Python – Data Types

Variables have a type

You can check the type of a variable by using the type() function:

```
print (type(integer_variable))
```

It is also possible to change the type of some basic types:

`str(int/float)`: converts an integer/float to a string
`int(str)`: converts a string to an integer
`float(str)`: converts a string to a float

Be careful: you can only convert data that actually makes sense to be transformed

Python – Arithmetic Operations

+	Addition	1 + 1 = 2
-	Subtraction	5 – 3 = 2
/	Division	4 / 2 = 2
%	Modulo	5 % 2 = 1
*	Multiplication	5 * 2 = 10
//	Floor division	5 // 2 = 2
**	To the power of	2 ** 3 = 8

Python – Arithmetic Operations

Some experiments:

```
print (5/2)
print (5.0/2)
print ("hello" + "world")
print ("some" + 1)
print ("number" * 5)
print (3+5*2)
```

Python – Arithmetic Operations

Some more experiments:

```
number1 = 5.0/2
number2 = 5/2
```

what **`type()`** are they?

```
type(number1)
type(number2)
```

now, convert number2 to an integer:

```
int(number2)
```

Python – Making the output prettier

Let put the following into a new Code Cell:

```
print ("The number that you wrote was : ", numIn)
print ("The number that you wrote was : %d" % numIn)

print ("the string you entered was: ", stringIn)
print ("the string you entered was: %s" % stringIn)
```

Want to make it prettier?

\n for a new line

\t to insert a tab

```
print (" your string: %s\n your number: %d" %(stringIn, numIn))
```

for floating points, use %f

Python – Control Flow

So far we have been writing instruction after instruction where every instruction is executed

What happens if we want to have instructions that are only executed if a given condition is true?

Python – if/else/elif

Let's look at some example of booleans.

type the following into a code cell

```
a = 2
b = 5

print (a>b)
print (a<b)
print (a == b)
print (a != b)
print (b>a or a==b)
print (b<a and a==b)
```

Python – if/else/elif

The if/else construction allows you to define conditions in your program

(Don’t forget your indentation!!)

```
if conditionA:
    statementA
elif conditionB:
    statementB
else:
    statementD
this line will always be executed (after the if/else)
```

Python – if/else/elif

The if/else construction allows you to define conditions in your program

(Indentation is IMPORTANT!)

```
if conditionA:
    statementA
elif conditionB:
    statementB
else:
    statementD
this line will always be executed (after the if/else)
```

conditions are a datatype known as booleans, they can only be true or false

Python – if/else/elif

A simple example

```
simple_input = input(“Please enter a number: “)
if (int(simple_input)>10):
    print ("You entered a number greater than 10")
else:
    print ("you entered a number less than 10")
```

Python – if/else/elif

You can also nest if statements together:

```
if (condition1):
    statement1
    if (condition2):
        statement2
    else:
        if (condition3):
            statement3 # when is this statement executed?
else:  # which ‘if’ does this ‘else’ belong to?
    statement4  # when is this statement executed?
```

Python – For Loops

When we need to iterate, execute the same set of instructions over and over again… we need to loop! and introducing range()

(Indentation is IMPORTANT!)

```
for x in range(0, 3):
    print ("Let's go %d" % x)
```

Python – For Loops, nested loops

When we need to iterate, execute the same set of instructions over and over again… we need to loop! and introducing range()

```
for x in range(0, 3):
    for y in range(0,5):
       print ("Let's go %d %d" % (x,y))
```

Python – While Loops

Sometimes we need to loop while a condition is true...

(remember to indent!)

```
i = 0               # Initialization
while (i < 10):     # Condition
    print (i)       # do_something
    i = i + 1       # Why do we need this?
```

Task 1 - Code Joe

Variables to hold data

Mathematical Operations to do math :)

Conditionals to make decisions

Loops to repeat our process

Functions/Subroutines to reuse code

Objects or Classes to define our "things"

Let's meet Joe.

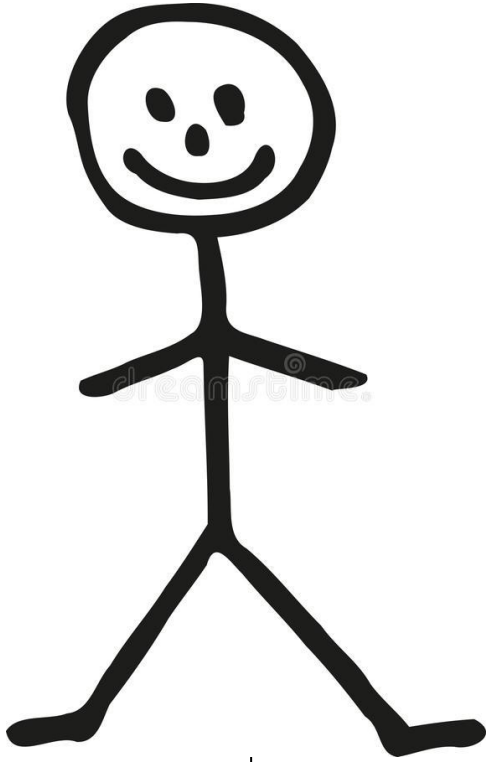
Joe might get sick.

Joe will be sick for 5 days.

After 5 days, Joe gets better.

Once Joe gets better, Joe can no longer get sick.

Let's "code" Joe.

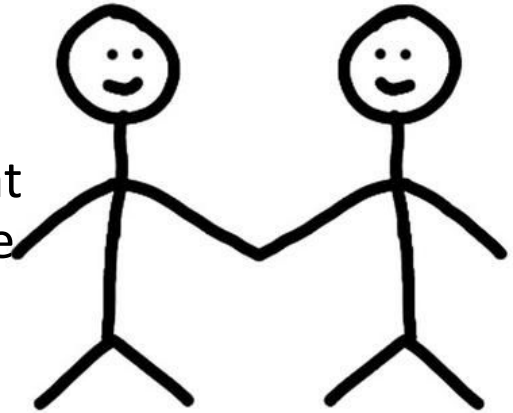


Task 2 Code Joe and Jane

We met Joe.

Joe has a friend, Jane

If Joe gets sick, Jane might get sick.



Modify your code, so when Joe gets sick that triggers Jane to roll a random number to see if Jane gets sick.

Loop through your code until both Joe and Jane get sick and they each get better.