DSCI 511: Data acquistion and pre-processing Chapter 0: System configuration and processing fundementals

0.0 If this is your first experience with Python

There are no pre-requesites for this course but we will be getting off of the ground fast with programming. So, if you do not have programming experience or are unfamiliar with the Python syntax, please consider following the (10 hr) online Python course from codecademy to get in the swing of things before week 1.

- https://www.codecademy.com/learn/learn-python (https://www.codecademy.com/learn/learn-python)

Note: some introduction to Python is provided in this notebook, but this is mostly just a fast/light refresher/introduction to some of the common commands we'll be using.

0.1 System Configuration

We'll be using a few free software technologies throughout this course, mostly doing data science in Python, but with some linux/unix command line tools in the background, at least to set things up. If you do anything before week 1 please be sure to get your system up and running! Before beginning work in this course you are required to set up **Python 3 with Jupyter notebooks**. If you do not yet have this set up in your working environment then please follow with one of the options in section 1.2.

0.1.1 Sign up for a GitHub account

Git on its own is a type of version control software that software developers use to maintain a record of the versions of their code. On the other hand, GitHub (https://www.github.com) is an online socialization of this software that allows teams of programmers to work on code together by maintaining versions in a coordinated repository. One very important side effect of GitHub is:

- users can freely upload, make available, and access data and code

These lecture notes are being presented in a hybrid markdown-code-data format known as a Jupyter Notebook, allowing for a multimedia exposition that GitHub supports well, e.g., one open-access DS textbook (https://jakevdp.github.io/PythonDataScienceHandbook/) is based on this idea. So, in addition to being a repository through which you can publicize and version your code, it's also an online space in which you can richly document materials with code in a rendered format for others to see. Note: in the interest of simplicity I will not be requiring work on GitHub or the use Git(Hub) directly in class. While I support any

efforts along these lines, any Git command usage will have to be independently learned. But if you do not already have an account I recommend you go to https://www.github.com (https://www.github.com) and sign up.

0.1.2 Python and Jupyter

We will be using the Python programming language for all assignments. However, for its value in typesetting and exposition all work should be in using Jupyter noteboks. Also, since there are still multiple active versions out there, its worth stating right away: **please make sure your installation includes Python version 3.**

0.1.2.1 What is Jupyter?

In brief, Jupyter is an html-based development environment with extensive markdown capabilities. A Jupyter Notebook is an interactive development interface with language-specific kernels. These allow developers to explore programming with objects held persistently in memory. Python was the flagship language for Jupyter (nee IPython), and has grown around an extensive suite of modules for analysis and visualization. These together with Jupyter make for an excellent suite of tools for exploratory data analysis and data-driven report generation. These lecture notes are being presented as a Jupyter Notebook.

0.1.3 Installing Python and Jupyter (and a command line environment)

0.1.3.1 Anaconda: an all-in-one distribution with GUI installer

There are a few ways to get set up with Python and Jupyter notebook, and the fastest all-in-one route is probably through the Anaconda distribution by Continuum:

- https://www.continuum.io/downloads (https://www.continuum.io/downloads)

Anaconda is specifically set up for data science and will come bundled with most of what you need to get going, in terms of modules, etcetera. If you've not done much programming before you might find its GUI installers for multiple operating systems convenient. Note: Anaconda comes with Jupyter.

0.1.3.2 Setting up a command line environment

If you're more adventurous you might like to download your versions of python and manange environments yourself using command line utilities. These differ across operating systems and also might require some setup. Even though we won't be using command line utilities much in this course, and even if you're going to use Anaconda, it's still a good idea to get your working enviornment set up with some helpful command line tools, but its ok if this takes a bit more time to gel. Even though we're working in Jupyter notebooks to learn, explore, share, and prototyle for data science, its still generally best to deploy code in scripts and using bash/shell commands from a command line.

To start getting your command line environment set up, here's some operating system specifc routes to follow

On a Mac you'll want to download the homebrew package manager:

- https://brew.sh/ (https://brew.sh/)

Once you have the `brew` command you'll be able to install different versions of command line utilities (including Python and Jupyter), just like on a linux machine—the packages are just available from a different package manager/command (`brew` , instead of `apt`). Oftentimes, Mac will have a less nice version of a command line utility, so there some good blog posts out there about replacing the pre-packaged software on a mac:

https://www.topbug.net/blog/2013/04/14/install-and-use-gnu-command-line-tools-in-mac-os-x/ (https://www.topbug.net/blog/2013/04/14/install-and-use-gnu-command-line-tools-in-mac-os-x/)

On Windows 10 you'll have to first set up the linux bash shell:

- https://www.howtogeek.com/249966/how-to-install-and-use-the-linux-bash-shell-on-windows-10/ (https://www.howtogeek.com/249966/how-to-install-and-use-the-linux-bash-shell-on-windows-10/)

After setting this up you should be able to run the `apt` command (a package manager) to get the utilities you need (like Python) just as for linux, below.

On Linux you'll be able to use `apt-get` from a terminal right away to install software. Here's an old Python 2 tutorial for setting up Jupyter that accurately reflects the process:

- https://www.digitalocean.com/community/tutorials/how-to-set-up-a-jupyter-notebook-to-run-ipython-on-ubuntu-16-04 (https://www.digitalocean.com/community/tutorials/how-to-set-up-a-jupyter-notebook-to-run-ipython-on-ubuntu-16-04)

A reminder: if you follow any of the above instructions please be sure to modify downloaded/installed versions for Python 3 (not 2) as needed.

Once you have you command line environment set up and can use a package manager try downloading a really nice json reader:

- https://stedolan.github.io/jq/ (https://stedolan.github.io/jq/)

Try downloading/installing from either of `brew` or `apt` ; it's really helpful when want to peek at some data!

Also, once you have a command line environment set up its nice to customize. Here's a good blog post about one person's bash profile:

- https://natelandau.com/my-mac-osx-bash_profile/ (https://natelandau.com/my-mac-osx-bash_profile/)

Even though this post is about someone's Mac, the general concepts of profiles, `PATH` variables, alias', and hidden files should all translate, regardless of operating systems and shells.

0.2 Programming Fundementals

0.2.1 How do Jupyter Notebooks work for Python development?

Since these lectures notes are a Jupyter notebook with a Python kernel, this notebook is itself our Python development environment. Interacting with Jupyter for basic programming is straight forward. Commands are typed in code cells and executed by typing:

```
<SHIFT> + <RETURN>
```

In [1]:

```
print("Hello world?")
```

```
Hello world?
```

Objects defined in one cell will persist in and be modified by subsequently executed cells:

In [2]:

```
mystring = "this is some text"
```

In [3]:

```
print(mystring)
```

```
this is some text
```

In [4]:

```
mystring = "and " + mystring + " that I have added to the string"
```

In [5]:

```
print(mystring)
```

```
and this is some text that I have added to the string
```

0.2.2 Python Programming

Python is an object-oriented programming language with a light syntax that makes it easy to read. It's got lots of pre-developed libraries for quantitative analysis and visualization as well as lots of low-level function for system integration and data management.

Your assignments in this course will require Python programming. However, varying degrees of experience are expected making independent learning of Python programming an expectation for the course. In addition to your instructor, Python programming resources for this course include:

- Course textbooks: The Data Science Handbook, Data Science From Scratch, and the Python Data Science Handbook all provide python programing resources.
- The docs: Python's official documentationn (https://docs.python.org) is quite readable
- Your classmates: All assignments will be group-based work. Collaborate and learn from each other.
- Stackoverflow: A search in the help forums often produces best/quickest answer.

Of our textbooks, Data Science From Scratch probably delivers the most on Python's fundementals, which we'll now follow.

0.2.2.1 Python's general formatting and code structure

Python uses whitespace for formatting. Here's an example that loops over some integers, does some addition, and prints them out. Notice that indentation defines a command's depth in the loop:

In [6]:

```
for i in [1, 2, 3, 4, 5]:
    print(i) # first line in "for i" block
    for j in [1, 2, 3, 4, 5]:
        print(j) # first line in "for j" block
        print(i + j) # last line in "for j" block
    print(i) # last line in "for i" block
print("done looping")
```

```
1
1
2
2
3
3
4
4
5
5
6
1
```

```
2
1
3
2
4
3
5
4
6
5
7
2
3
1
4
2
5
3
6
4
7
5
8
3
4
1
5
2
6
3
7
4
8
5
9
4
5
1
6
2
7
3
8
4
9
5
10
5
done looping
```

Whitespace is ignored inside of parentheses and brackets, and backslashes continue lines:

In [7]:

```
long_winded_computation = (1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 + 11 + 12 +
                           13 + 14 + 15 + 16 + 17 + 18 + 19 + 20)
print(long_winded_computation)
```

```
210
```

"blocking out" statement allows you to see object structure

In [8]:

```
list_of_lists = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

print(list_of_lists)

easier_to_read_list_of_lists = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

print(easier_to_read_list_of_lists)
```

```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Sometimes there are no whitespace-ignoring parentheses and Python needs to be told that the statement is not over:

In [9]:

```
two_plus_two_is = 2 +\
2
print(two_plus_two_is)
```

```
4
```

0.2.2.2 Modules

Part of the Python appeal at this point is the wealth of well-constructed code libraries, or, modules. These allow programmers to stand on each others' shoulders and save time on code re-creation.

Many Python features don't exist by default. Python can only do basic arithmatic and has limited string (text) manipulation to start. Modules that help with these issues are `numpy` and `re`. Most of what we'll be working on in this course and DSCI 521 will be done using the following modules: `numpy`, `re`, `sys`, `os`, `scipy`, `pandas`, `PIL`, `matplotlib`, `ipython`, `datetime`, `bs4`, and `requests`. There are probably some others, but we'll get to these on the fly.

Some modules won't come with your Python distribution. So, if you try and run code and a module, e.g., `numpy` doesn't exist, what can you do? Use a package manager to download modules from your computer's command line. If you needed to download numpy using either of the Python Package Index (`pip`) or Anaconda (`conda`), you would execute one of the following commands:

```
>>> pip install numpy
>>> conda install numpy
```

Note: if you have set up Python from multiple sources and/or are downloading modules from multiple sources you might have to configure your `PYTHONPATH` command-line variable to make sure that your system looks for/loads modules from the correct places and in the correct order. This works just like the `PATH` variables discussed in the blog post on command line system configuration (in section 1, above):

- https://natelandau.com/my-mac-osx-bash_profile/ (https://natelandau.com/my-mac-osx-bash_profile/)

Once again, even though this post is about someone's Mac, the general concepts of profiles, `PATH` variables, alias', and hidden files should all translate, regardless of operating systems and shells.

0.2.2.3 Example 1: `numpy`

This module turns Python into an excellent calculator. It means your can calculate averages and many, many more complex numerical tasks with abstracted commands and less control structure.

In [10]:

```
## it often helps to shorten the names of your modules
import numpy as np

first_five_indices = range(5) # range is a built-function that makes integers

## computing an average the old-fashioned way
average = 0
for number in first_five_indices:
    average = average + number
average = average / float(len(first_five_indices))

print(average)

## using numpy's mean function
print(np.mean(first_five_indices))
```

```
2.0
2.0
```

0.2.2.4 Example 2: `regex`

This module turns Python into an excellent string (text) manipulator. This means that `regex` adds capability for things like pattern searching and replacing.

Note: regex is a Python 3 specific module offering improvements over the `re` legacy module, levt over from Python 2. For most task `re` does fine, but `regex` can do a few nice extra things, like character class modification. For simplicity, we'll `import regex as re` whenever we're using it.

In [11]:

```
import re

a_silly_string = "one fish two fish red fish blue fish"
print(a_silly_string)

## removing blue fish from the picture
only_one_fish_left = re.sub(" (two|red) fish","",a_silly_string)
print(only_one_fish_left)
```

```
one fish two fish red fish blue fish
one fish blue fish
```

0.2.3 Functions

0.2.3.1 Explicit functions

A function is a rule for taking zero or more inputs and returning a corresponding output. A common way to define functions is `def` :

In [12]:

```
def double(x):
    """this is where you put an optional docstring
    that explains what the function does.
    for example, this function multiplies its input by 2"""
    return x * 2
```

In [13]:

```
print("this is what happens when you double a number:")
print(double(2))
print("")
print("this is what happens when you double a string:")
print(double("ha"))
print("")
print("this is what happens when you double a list:")
print(double([4,"haha"]))
print("")
```

```
this is what happens when you double a number:
4

this is what happens when you double a string:
haha

this is what happens when you double a list:
[4, 'haha', 4, 'haha']
```

Functions can have preset values:

In [14]:

```
def double(x="trouble!"):
    """this is where you put an optional docstring
    that explains what the function does.
    for example, this function multiplies its input by 2"""
    return x * 2
```

In [15]:

```
print("this is what happens when you double a number:")
print(double(2))
print("")
print("this is what happens when you nothing:")
print(double())
print("")
```

```
this is what happens when you double a number:
4

this is what happens when you nothing:
trouble!trouble!
```

0.2.3.2 Anonymous (lambda) functions

Unlike `def` ined functions, anonmyous functions require extreme brevity of statement. Thus, their syntax orients them towards convenience at defining small, non-complex functions. Lambda functions are commonly used for relatively-simple data transformations or when rule needs to be passed to an explicit function, for example, telling what rule the built in `filter(f, data)` function should use to cut down on some data.

In [16]:

```
## the syntax is `lambda input: output`

lambda_double = lambda x: x * 2 # note x is a dummy variable

print("this is what happens when you double a number:")
print(lambda_double(2))
print("")
print("this is what happens when you double a string:")
print(lambda_double("ha"))
print("")
print("this is what happens when you double a list:")
print(lambda_double([4,"haha"]))
print("")
```

```
this is what happens when you double a number:
4

this is what happens when you double a string:
haha

this is what happens when you double a list:
[4, 'haha', 4, 'haha']
```

0.2.4 Control Flow

0.2.4.1 Conditioning

As with most programming languages, actions can be performed conditionally using `if`. This includes the basic `if` along with its partial counterparts: `elif` and `else`.

In [17]:

```
if 1 > 2:
    message = "if only 1 were greater than two..."
elif 1 > 3:
    message = "elif stands for 'else if'"
else:
    message = "when all else fails use else (if you want to)"
print(message)
```

```
when all else fails use else (if you want to)
```

0.2.4.2 Conditional assignment

A neat feature in Python utilizes control flow to afford succinct nimble object assignment. Here, the value of an assigned object depends on the state of a different object.

In [18]:

```
## The syntax is `object = <VALUE> if <CONDITION> else <VALUE>`

number_of_fishes = 2 if re.search("red", a_silly_string) and re.search("red", a_
silly_string) else 1
print(number_of_fishes)

number_of_fishes_left = 2 if re.search("red", only_one_fish_left) and re.search(
"red", only_one_fish_left) else 1
print(number_of_fishes_left)
```

```
2
1
```

0.2.4.3 Truthiness

In Python, you can use just about any object for its boolean (true/false) value. Mostly, default, empty, and completely unmodified objects will result in `False` boolean interpretations. Otherwise, objects will be interpreted as `True`. The list of falsies includes:

- `False` (Boolean False)
- `None` (A nonexistent value)
- `[]` (an empty list)
- `{}` (an empty dict)
- `""` (an empty string)
- `set()` (an empty set)
- `0` (the integer zero)
- `0.0` (the float zero)

For example, we can use object truthiness to control a while loop:

In [19]:

```
## makes a list of things
things = ["this", "that", "the other", "that", "this", "that"]
while things: ## check the boolean value of `things`; stops when `things` is emp
ty
    ## 'pops' the last element off of the list
    thing = things.pop()
    print("Got another of "+thing+" one")
```

```
Got another of that one
Got another of this one
Got another of that one
Got another of the other one
Got another of that one
Got another of this one
```

0.2.4.4 Infullness

As a type of truthyness operator, Python's `in` checks if a replicate of an object exists inside of another.

Note: `in` checks inclusion in collections (keys, for dictionaries), and is slow when applied to lists.

In [20]:

```
## our example set
first_four_set = {1,2,3,4}

## 1 is in this set
one_is_in = 1 in first_four_set
print(one_is_in)

## 5 is not in this set
five_not_in = 5 in first_four_set
print(five_not_in)
```

```
True
False
```

0.2.4.5 Exceptions

When something goes wrong, Python raises an exception. Unhandled, these will cause your program to crash. You can handle them using the `try` / `except` syntax.

Note: You cannot do a `try` without an `except` , but the `except` can be universal, i.e., you can leave out the `ZeroDivisionError` specification. However, bugs relating to code in `try` / `except` contexts can be very difficult to diagnose, especially if explicit exceptions are not used. Here's a specific divides by zero example:

In [21]:

```
## this prevents the program from crashing when we divide by zero
try:
    print(0 / 0)
except ZeroDivisionError:
    print("cannot divide by zero")
```

```
cannot divide by zero
```

In [ ]:

```

```