This walkthrough is meant to mirror and replace for Humanities scholars---and hopefully some Social Sciences scholars as well--the Python lessons featured on software-carpentry.org. It is not that numpy or the related techniques shown in the posted lessons have no place in the Humanities or Social Sciences, to the contrary there is much of value to be learned in that lesson, but rather that the methods underpinning that lesson are mostly foreign to these academic disciplines and so stand in the way of learning programming via Python. By featuring qualitative textual data from a Twitter feed rather than quantitative numeric data from a biological study it is hoped that neither the contents nor the methods will be quite so foreign to those within the Humanities and Social Sciences. Perhaps this lesson will be of value to those with more quantitative backgrounds as well.

1. Python / Jupyter Basics

In this first section we'll quickly cover some basic syntax, control structures, objects, and output in Python so that when it comes to looking at the basics of actual analysis we are in a better position to see the big picture.

Boxes and Markdown

Open `Lesson #1.ipynb`

Let's begin by typing a brief description of what we are about to do into a *markdown* box at the top of a new Python 3 workbook in Jupyter notebooks. Enter something like the following in a markdown box.

```
Learning the basics of Python.  When done I will have created my own word
histogram.
```

Note that while markdown is raw text on input an interpreter will convert various character sequences into formatting commands that will change the final output. Change the original line to say:

```
**Learning the basics of Python.**  When done I will have created *my own* word
counter.
```

It will continue to look just as you typed it until you either `hold shift and press enter` or press the `play button` to process the block. Do either now and your text will render as:

Learning the basics of Python. When done I will have created *my own* word counter.

Note that you can double click on the block to edit the text and then process the block again.

We won't do any more here but you are encouraged to play and experiment on your own later.

There in no single standard to appeal to for markdown formatting although there is general agreement on some core behaviours. When working within Jupyter you should refer to the Markdown section of the Jupyter docs. You may also find it useful to refer to the GitHub Markdown Guide since that is another popular place where Markdown is used. If you are looking for a template document this guide was written in markdown and should be available wherever you got the PDF version as a .md file.

Note that there are other types of cells/blocks as well. Make sure that the right cell type is turned on for the content or you won't get the results you are expecting.

Comments

In the cell immediately below the markdown description we just made is where we'll build our histogram tool. Make sure that the type of the cell is code.

Before we write any actual code we should plan out our program. We'll do this with comments. Comments are portions of the code that are not interpreted by the language when the program is run. Typically comments are helpful descriptions of surrounding code for humans to read. Sometimes they are used during testing to turn on and off lines used in debugging.

There are two types of comments in Python, single line and multi line. Single line comments start with a #, the hash or pound symbol found above the numeral 3 on most keyboards. Let's enter the following single line comments to roughly describe what we are going to be doing:

```
# This program will count the words in a string and graph the 10 most common
words

# Take some input

# Process the input word by word, counting each word

# Produce the chart
```

Multi line comments are openend and closed with triple double quotes: """. We don't have a use for them now but we likely will before the workshop is over.

Let's create or move to a new cell below this one where we can learn and practice how to use the pieces of the python language that we will need to create our word histogram.

Functions

Type the following into the new code block.

```
print("hello world")
```

Notice that the text is coloured. This is the Python interpreter running in the background telling you what various components of the statement are.

See what the line does by running the cell.

```
hello world!
```

Congratulations, you have just written and run your first program!

Let's look at what happened here. "print" is what is known as a function, meaning that it is a sort of list of commands or processes that the computer is to perform. Functions are always invoked by writing out the name of the function followed by a set of parentheses. What is inside the parenthesis are the inputs into the function, what it needs or expects to carry out the instructions inside. There can be many inputs, just one (as is the case here), or no inputs (such functions are known as "empty functions"). Here our input is the text string *hello word!*. We know that it is a string of text characters because we have wrapped it in quotations.

> *What happens when the quotations are:*
>
> a. *removed?*
> b. *swapped with single quotes?*
> c. *used inconsistently (e.g. no closing quote, open with a single but closed with a double, etc.)?*
> d. *doubles are used inside singles (e.g. " 'hello wold!' ") or vice versa?*
> e. *the quote type used inside is the same as is used outside (e.g. " "hello wold!" ")*

[Go over errors and show how to escape characters with `\`. If they have done the commandline part of this already then they should be reminded that they have seen this before in reference to spaces in filenames.]

Note that print() is quite happy to take more than one input:

```
print("hello","world","!")
```

or no input

```
print()
```

IMPORTANT: A major change from Python 2 to Python 3 is the shift from print just being a statement:

```
print "hello world!"
```

to print being implemented as a function (for consistency):

```
print("hello world!")
```

It is a small change that may not seem to matter but it really does because it renders the vast majority of Python 2 programs incompatible with Python 3.

print() is a built-in function, it comes pre-installed and activated with every version of Python. There are other functions that are built in but which we must load, others that we will have to install separately and then load, and still others that we'll create ourselves.

For the moment though we'll leave these further details of functions behind and look at another core concept, variables.

Variables

Variables provide us named containers to hold our objects making it easier to pass the contents around within the program without actually having to provide the original object. For a parallel in language consider how convenient it is to use the phrase "my parents" to refer to people who are my parents rather than carrying them around with me everywhere and having to point to them each time I refer to them. Let's not get carried away with the analogy.

In the case our hello world example imagine how annoying and counterproductive it would be if what we were working with was not "hello world!" but the text of Moby Dick!

Let's create and use some variables. We'll start by typing the following in a new block:

```
firstVar = "hello world!"
```

What we have done here is create a variable called "myVar" and assign to it the string "hello world!". Three important points to note:

1. **The variable name is short and meaningful within this context.** It is the first variable that we have created in python. Avoid the temptation to call your variables in other programs things like "var1" or "x" because doing so will make it much harder to debug the program.
2. **The equal sign is the assignment operator and *not* the equality operator.** When invoked it assigns the name on the left to the object on the right.
3. **Running this code block will result in no output.** As with the command line "silence is golden" so we'll only see output when we explicitly ask for it (there are a few exceptions in Jupyter, such as when we have only a variable name on a line and its content gets printed out) or deserve it. Hence the importance of print().

> *How do we view the content of our variable?*

We can assign many different types of data to a variable, even changing the type after the variable is created (this is not true with other "strict typing" languages). Let's assign it a numerical value:

```
firstVar = 5
```

We are going to need a variable that holds a string to run our word counting tool so let's do create that now in our master block that we will use throughout.

```
# Take some input
inputString = "this is some sample text"
```

Control Structures Part 1: Loops

Now that we have some content to use as a draft we need to process it. To do this we'll need to make use of both loops and conditionals. We'll start with a loops.

Loops are used when we want to do the same thing over and over again, typically with changes to at least one of the inputs. This allows us to code more efficiently. In Python there are two types of loops, *while loops* and *for loops*. While loops run until a condition is met. For loops run until an entire list is consumed. The exception to both these descriptions is that both loops can be ended using a `break` statement within the loop.

Let's look at an example of each:

- While

```
fruit = 'banana'
index = 0
while index < len(fruit):
    letter = fruit[index]
    print(letter)
    index = index + 1
```

- For fruit = 'banana' for letter in fruit: print(letter)

[Walk through each of these line by line with the attendees.]

As you can see, for loops are usually the more compact and efficient choice when the entire list is to be processed. Do not forget the while loop though, it is the correct choice when you don't know where to stop processing a list.

> *Note that they only thing special about the variable names used is that they are meaningful for humans. Prove this to yourself by changing each of 'fruit', 'index', and 'banana' to something totally different within each code block. Make sure to do this consistently.*

If we want to count *all* the words is a given body of text then we should use a for loop. We should also test to make sure that we are reading the input properly so let's print the result of each pass through the loop:

```
# Process the input word by word, counting each word
for word in inputString:
    print(word)
```

> *What happens when we run this code? Why?*

The problem is that Python is seeing our input as a single string of characters rather than a list of words (Note though that this is further proof that the variable names only matter to us). We could fix this now but let's finish the control logic of the program first, thinking of this as

character histogram for the moment.

Lists and containers, including strings

Lists

As me move through the loop we are going to need to collect the items we come across---whether they are characters or words or anything else---and count them. To do this we need to use containers and lists. Let's have a quick primer on lists and containers.

- Ordered collection of objects. strings are a special type of list where each element is a character and the list type has its own methods.

```
list1 = ["hello","world","!"]
```

- indexing and slicing

> *If [x] is the indexing command then what will list1[1] return? Why?*
> *What about list1[1:2]?*
> *What will list1[0]="Hello" do?*

- Lists can hold almost any type of object and can be heterogeneous.

```
list2=[5,4.0,"wow"]
```

Strings

Strings are *ordered sequences of characters* and as sequences we can index and slice strings just like lists. We saw this happening with the loop examples and this is what is happening with our little program right now.

```
string1 = "Hello World!"
```

> *Why do we get "e" as the return? Zero based counting.*

```
string1[0:5]
```

> *What values will just give us "world!"? string1[6:12] and string1[6:]*
> *What about [0:0] and [6:3] and negative values?*

Strings are immutable. We modify them only by overwriting them.

Dictionaries

The container type that we will use for our little program though is a dictionary. Dictionaries use key-value pairs for indexing rather than locations in a list.

```
dict1 = {'one':'un','two':'deux','three':'trois'}
```

> *Display the contents of the dictionary*

- note how the order has (likely) changed!
- Accessing elements

  ```
  dict1[two]
  ```

- *dict1[two]* fails, no variable named "two"

  ```
  dict1[‘two’]

  dict1[‘four’]
  ```

- *dict1[‘four’]* fails, no index named "four"
- Checking members

```
‘four’ in dict1

‘trois’ in dict1
```

- Adding/modifying members

```
outFile = open(‘output.txt’, ‘r’)

for line in outFile:

    print linedict1[four] =  ‘quatre’
```

- Checking for inverse members. Can’t do that yet (Dictionaries don’t work that way by default). For that we need functions, loops, and conditionals.

We will use a dictionary that uses the word/letter as the key and returns a count of that letter. For the moment let's just make sure that we can load the dictionary and worry about actually counting later:

```
# Take some input
inputString = "this is some sample text"

# Process the input word by word, counting each word
histdict = {}
for word in inputString:
    histdict[word] = 1

print(histdict)

> Note that both histdict and the print statement are outside the loop.  What
will happen if we move either or both of them inside the loop?  Try it.
```

What we need now is a way to count the occurrences of a word/letter. To do this we need to be able to recognize when we have seen an item before or not and then change the behaviour accordingly. To do this we need our second control structure: conditionals.

Control Structures Part 2: Conditionals

The core conditional control structure in Python is the if. It is pretty straightforward to use, as follows:

```
x = 5

y = 4

if x < y:

    print ("x is less than y")
```

Of course this is not the only case, y can also be less than x. We can add this as follows:

```
x = 5
y = 4

if x < y:
    print ("x is less than y")
else:
    print ("y is less than x")
```

Still, we have not caught all the cases

```
x = 5
y = 4

if x < y:
    print ("x is less than y")
elif x > y:
    print ("y is less than x")
elif x == y:
    print("x is equal to y")
else:
    print("Something weird is happening")
```

While it seems unnecessary to add the last line because all the cases are covered it is good practice to explicitly trap all the cases you expect and then use else to grab any that you don't.

We can also combine conditionals with logical operators

```
x = 5
if 0 < x and x < 10:
    print("x is a positive single digit number")
```

We can now go back and modify our program by adding a conditional that will allow us to initialize an entry for an item to 1 when we see it for the first time or to do something else if we've seen it before:

```
# Take some input
inputString = "this is some sample text"

# Process the input word by word, counting each word
dicthist={}
for word in inputString:
    if word not in dicthist:
        dicthist[word] = 1
    else:
        dicthist[word] = 2

print(dicthist)
```

Of course, we need some way to account for items that occur more than twice. For this we need to talk about operators.

Operators

Objects are the nouns of the language. You do things to them with other things. Strings, integers, variables, lists, etc. are all types of objects. Operators are the things you use to do things to objects. When you do something to an object with an operator we refer to that object as an operand. Things like + - ^ * ** / = == < > <= >= are all operators.

You've already seen punctuation, these are the characters (including spaces) that keep the syntax together. These include spaces, : , () [] {}

Expressions arise when we link objects together with operators and the right punctuation.

Statements are things that the computer will do. Expressions are kinds of statements but think of them as separate things. Expressions have values, statements work with values.

> *Create some variables and assign numbers to them. Try all the standard operators + * / - on them.*
> *What does 1 // 2 produce? Why? Integer division.*

IMPORTANT: In Python 2 / is *integer division by default*. Can overcome this by adding decimal placeholders to at least one component of the formula.

> *What does the **operator do?** * is the exponent. Often ^ elsewhere.*

```
2 ** 3
```

> *What does the % operator do? * % is the modulo operator. It returns the remainder when the first number is divided by the second.*

```
7 % 2
```

- returns 1 since 2 goes into 7 three times with one left over.

> *What operators can we use with lists? Try some out!*

```
list1 + list2

list1 * 3

list1 + string1
```

Note that *list1 + string1* fails. Need to recast string1 as a member of a list:

```
list1 + [string1]
```

With operators we can modify our program to continually add each new instance of an item that it discovers:

```
# Take some input
inputString = "this is some sample text"

# Process the input word by word, counting each word
dicthist={}
for word in inputString:
    if word not in dicthist:
        dicthist[word] = 1
    else:
        dicthist[word] = dicthist[word] + 1

print(dicthist)
```

Alternatively, we can use the increment operator to make this cleaner:

```
# Take some input
inputString = "this is some sample text"

# Process the input word by word, counting each word
dicthist={}
for word in inputString:
    if word not in dicthist:
        dicthist[word] = 1
    else:
        dicthist[word] += 1

print(dicthist)
```

It works! For letters anyway. Let's see how to get it working for words.

Methods

As we have seen, the program as currently written treats our input as a string, which is a sort of list that has an ordered set of characters as its members. What we need to do is turn it into a list. We could write a program that would read through the string adding each character to a new string until finding the end ofa word, adding that word to a list, and then continuing the process until everything is done and *then* processing this new list rather than the initial string. Ugh. That's a lot of work. Fortunately someone else has already done this for us. Even more fortunately they have done it for lots of similar actions and there is an easy way to invoke them, methods.

Methods are special functions that are associated with certain data types. They are invoked by typing the name of the variable holding that data type and then immediately following that with a

dot and then the name of the method.

Let's try some methods out on some lists and some strings.

- list methods. append, expand, and sort. Note that list methods are all void. They modify the original list and return None.

```
list1.sort()

list1.reverse

list1.index("Hello")

list[1].capitalize()

list1[1]=list[1].capitalize()
```

- Adding

  list1.append(list2)

[Once append is on the table it is possible to demonstrate the difference between passing by reference (append) and passing by value (+). Note as well that appending a list to a list will make the appended list a *member of* the first list while using + will make all its members members.

```
list1 = ["hello","world","!"]
list2 = [5,4.0,"wow"]
list3 = list1 + list2
list4 = list1
list4.append(list2)
print(list3)
print(list4)
list1[0]="HELLO"
print(list3)
print(list4)
```

]

Need to know the available methods? Tab/Shift-Tab completion.

- How will we capitalize the "*wow*" in list1?

```
list1[3][2].capitalize()

list1.insert(2,"two")
```

- deleting elements

```
list1.pop(2)

list1.remove("wow")
```

- *list1.remove("wow")*fails. Need to access the list in the list.
- Only removes the first instance
- string methods: len, upper, lower, replace, etc.

```
len(string1)

string1.swapcase()

string1.swapcase()
```

- Why didn't it go back to lowercase the second time? We're not modifying the string.

```
string1.replace("!", "!!!")
```

- *'trois' in dict1* fails because it is a value, not a key

```
dict1Val = dict1.values()

'trois' in dict1Val
```

- Or go straight for the throat with *'trois'* in *dict1.values()*

What we want for our program is the split method:

```
# Take some input
inputString = "this is some sample text"

# Process the input word by word, counting each word
dicthist={}
for word in inputString.split():
    if word not in dicthist:
        dicthist[word] = 1
    else:
        dicthist[word] += 1

print(dicthist)
```

Now that we have our data in a useful format we can look at plotting it as a histogram.

Charts

The core library used for plotting in Python is matplotlib. It has website that is full of examples, some of which might be exactly what you want: matplotlib.org. Ultimately it is a very powerful package that can quickly become quite complex to use (We could easily spend a day just on it). Extra help is available from the External Resources Page. We'll glance at the basics, drop in a solution, and then move on (You'll be seeing it again).

Let's start by playing with the following example:

```
import matplotlib.pyplot as plt

#special ipython/jupyter command that keeps the output in this window rather
than opening another one.
%matplotlib inline

plt.figure()
plt.plot([1, 2, 3, 4], [10, 20, 25, 30], color='lightblue', linewidth=3)
plt.scatter([0.3, 3.8, 1.2, 2.5], [11, 25, 9, 26], color='darkgreen',
marker='^')
plt.bar([1,2,3,4],[12,3,25,18], width=0.2, align='center')
plt.xlim(0.5, 4.5)
plt.ylim(0,50)
plt.show()
```

> *See what happens as you change the values around, comment lines out, and change*

the options.

Note three important things: 1. The very first line of this block *imports* a new library. The library is matplotlib. Actually, the whole library isn't loaded, just a portion of it called "pyplot". This portion is also renamed "plt". 2. A special ipython/jupyter only command is issued that stops the plot from appearing in its own window. The percentage sign at the beginning indicates that it is this type of command.

> *How could you print out the line, bar, and scatter plot separately but from the same cell?*

It is time for us to add a bar chart to our program. We can copy up a lot of content from the example. We'll add one feature that will allow us to include labels in the plot, making the whole program look like this.

```
# Take some input
inputString = "this is some sample text"

# Process the input word by word, counting each word
dicthist={}
for word in inputString.split():
if word not in dicthist:
    dicthist[word] = 1
else:
    dicthist[word] += 1

print(dicthist)

import matplotlib.pyplot as plt
%matplotlib inline

plt.figure()
plt.bar(range(len(dicthist.values())),dicthist.values(), width=0.2,
align='center')
plt.xticks(range(len(dicthist)), list(dicthist.keys()),rotation='vertical')

plt.show()
```

User Input

At times it can be really useful to take user input. We can do this with the input() function, changing the inputString line to:

```
inputString = input("Please enter some sample text")
```

Tweaking

Figure getting a little small? Consider changing its dimensions by swapping `plt.figure(figsize=(15, 6))` for `plt.figure()`.

Too much information on the chart? Set it to only print the top 10 most common items:

And that should be enough of the basics. You've seen all the primary pieces of python at this point *and* make a working program

2. Basic Analysis

With the basics out of the way we can move to using them to work with a specific data type: Tweets. While perhaps not a truely representative sample of what is important Twitter does offer an important snapshot into a variety of communities and trends. In this portion of the workshop we'll focus on handling some pre-collected tweets while drawing on our histogram skills to see what words are most strongly associated with the chosen topic.

The topic? We'll go with something likely to be of general interest: Trump and Clinton in the US Presidential Primaries.

Planning the Program

Let's start by opening "Lesson #2.ipynb".

In the top cell we'll start by planning out our program with comments:

```
# This program will read twitter data stored in a file line by line and then
produce a graph showing the most common words used in the tweets.

# Load twitter data from a file

# Process that data to extract the tweet text and build a list of words

# Produce the chart
```

As always there will be sub steps but this is enough to get us started for now.

Reading data from a file

Let us play with opening files in a new cell:

```
open("trumpTweets.json", "r")
```

When we run this we'll see output like the following:

```
<_io.TextIOWrapper name='trumpTweets.json' mode='r' encoding='UTF-8'>
```

What we have done is create a connection to a file and what is displayed are the details of this connection. What we have failed to do is capture this connection with a variable so that we can work with the file. This is easily remedied by simply assigning the output of the open file function to a variable

```
inputFile = open("trumpTweets.json", "r")
```

Before we go any further we need to make sure that we close the file too. Python is pretty good about doing this for us but hiccups happen and when they do they can result in data corruption, especially when we are writing ("w") or appending ("a"). So let's close the file at the end of the cell block:

```
inputFile = open("trumpTweets.json", "r")

inputFile.close()
```

We'll just remember this line is here and not display it further.

If we run this line we get nothing returned. If we print inputFile we get the same file info as before. What we need to do is unpack the file line by line (we don't always have to unpack the file line by line but it is good practice for the time being). There is a lot of data in 90 tweets so let's just read one line for now using the readline() method for python file objects:

```
inputFile = open("trumpTweets.json", "r")
print(inputFile.readline())

{"is_quote_status": true, "in_reply_to_status_id_str": null, "coordinates":
null, "possibly_sensitive": false, "in_reply_to_screen_name": null, "lang":
"en", "favorite_count": 0, "favorited": false, "metadata": {"iso_language_code":
"en", "result_type": "recent"}, "entities": {"hashtags": [{"text": "Trump",
"indices": [20, 26]}], "symbols": [], "urls": [{"expanded_url":
"https://twitter.com/realdonaldtrump/status/725524257883697152", "display_url":
"twitter.com/realdonaldtrum\u2026", "indices": [139, 140], "url":
"https://t.co/Nayyhhonr9"}], "user_mentions": [{"id": 3314758074, "indices": [3,
18], "name": "JP Moore (REAL)", "id_str": "3314758074", "screen_name":
"Campaign_Trump"}]}, "source": "<a href=\"http://twitter.com/download/android\"
rel=\"nofollow\">Twitter for Android</a>", "created_at": "Thu Apr 28 04:23:46
+0000 2016", "retweeted": false, "in_reply_to_user_id_str": null, "text": "RT
@Campaign_Trump: #Trump billionaire is staying at Holiday Express. It's shows
you he is just a nice plain no BS down to earth guy.  http\u2026",
"retweet_count": 11, ETC.
```

Things are looking good, but what a blob of text! What we are seeing is JSON formatted data. If it looks a lot like a dictionary that's because that's pretty much what JSON is as far as we are concerned. That means that even though it looks nasty if you know the format and the keys you can quickly grab any content you want. The body of a tweet is associated with the key "text" so let's grab that for the first tweet:

```
inputFile=open("trumpTweets.json", "r")
line = inputFile.readline()
line["text"]
```

But this throws an error:

```
TypeError: string indices must be integers
```

The problem is that while the line has the potential to be a dictionary python is currently reading it just as a string. We need to import the json library and tell python to load the line as json:

```
import json
inputFile=open("trumpTweets.json", "r")
line = json.loads(inputFile.readline())
line["text"]
```

Now we get the text body of the first tweet.

Before we go further let's use the fact that we now have actual json/dictionary data to clean up the presentation of the entire line:

```
print(json.dumps(line, indent = 4))
```

While loads ingests data in the json format dumps pushes it out. Note that the 's' on the end isn't a pluralization, it signifies that we are working with *strings*. If we wanted to work with raw file objects then we would remove it (and need to change lots of things about how we interact with the data so we'll leave the s on).

We could go on with this method of loading data but it isn't really safe. If our program crashes between the open and the close there is no automatic clean-up and close tool to try and prevent data corruption. A better method is `with`:

```
with open("trumpTweets.json", "r") as inputFile:
    line = json.loads(inputFile.readline())
    line["text"]
```

Similar setup, safer practice.

With this new setup let us print the text of all the tweets

```
with open("trumpTweets.json", "r") as inputFile:
    for line in inputFile:
        tweet = json.loads(line)
        print(tweet["text"])
```

> *Finish off loading the data by copying what we have up to our top cell and modifying it to load each line into a list called tweets.*

```
tweets = []
with open("trumpTweets.json", "r") as inputFile:
    for line in inputFile:
        tweets.append(json.loads(line))
```

Tokenize the tweets

Now that we can load our data the next step is to split the body text of the tweets in words so that we can count them.

Let's see what happens when we bring in the tokenization tool from the NLTK (Natural Language Toolkit, a powerful NLP library for Python that comes with Anaconda):

```
from nltk.tokenize import word_tokenize

fake_tweet = "RT @symulation: just an example I'd like to share! :D http://
example.com #NLP #NLTK"
print(word_tokenize(fake_tweet))
```

Which churns out the terrible response:

```
['RT', '@', 'symulation', ':', 'just', 'an', 'example', 'I', "'d", 'like', 'to',
'share', '!', ':', 'D', 'http', ':', '//example.com', '#', 'NLP', '#', 'NLTK']
```

The problem is that Tweets have some pretty non-standard syntax so this isn't as easy as simply splitting a string at the spaces so we'll need to pull a tool to help us out. That tool is a pre-written tweet preprocessor. We'll look it over quickly but mostly accept it and set aside the details for now. Run the provided text pre-processor in your notebook:

```
#Instantiate a text pre-processor
import re

emoticons_str = r"""
    (?:
        [:=;] # Eyes
        [oO\-]? # Nose (optional)
        [D\)\]\(\]/\\OpP] # Mouth
    )"""

regex_str = [
    emoticons_str,
    r'<[^>]+>', # HTML tags
    r'(?:@[\w_]+)', # @-mentions
    r"(?:\#+[\w_]+[\w\'_\-]*[\w_]+)", # hash-tags
    r'http[s]?://(?:[a-z]|[0-9]|[$-_@.&+]|[!*\(\),]|(?:%[0-9a-f][0-9a-f]))+', #
URLs

    r'(?:(?:\d+,?)+(?:\.?\d+)?)', # numbers
    r"(?:[a-z][a-z'\-_]+[a-z])", # words with - and '
    r'(?:[\w_]+)', # other words
    r'(?:\S)' # anything else
]

tokens_re = re.compile(r'('+'|'.join(regex_str)+')', re.VERBOSE | re.IGNORECASE)
emoticon_re = re.compile(r'^'+emoticons_str+'$', re.VERBOSE | re.IGNORECASE)

def tokenize(s):
    return tokens_re.findall(s)

def preprocess(s, lowercase=False):
    tokens = tokenize(s)
    if lowercase:
        tokens = [token if emoticon_re.search(token) else token.lower() for
token in tokens]
    return tokens
```

Once we have run the block with the pre-processor we now have access to the preprocess function in all the cells of the notebook. We modify the previous tokenization as follows:

```
from nltk.tokenize import word_tokenize

fake_tweet = "RT @symulation: just an example I'd like to share! :D http://
example.com #NLP #NLTK"
print(preprocess(fake_tweet))
```

Nice. Now hashtags and at-mentions stay together. Same with emoticons.

> *Move this tokenization into our program at the top, modifying it so that rather than saving each tweet to the list of tweets it prints out the tokenization of each tweet.*

```
with open("trumpTweets.json", "r") as inputFile:
    for line in inputFile:
        tweet=(json.loads(line))
        print(preprocess(['text']))
```

Or as a single line:

```
with open("trumpTweets.json", "r") as inputFile:
    for line in inputFile:
        print(preprocess(json.loads(line)['text']))
```

Adding a counter

We could use our histogram at this point but there is a library for counters that gives us some rather nice features, like automatically counting members of a list.

```
from collections import Counter

with open("trumpTweets.json", "r") as inputFile:
    count_all = Counter()
    for line in inputFile:
        tweet=json.loads(line)
        terms = [term for term in preprocess(tweet['text'])]
        count_all.update(terms)
    # Print the first 5 most frequent words
    print(count_all.most_common(5))
```

Note the fancy move where we assign content to terms. Here we are taking what would otherwise be a multi-line for loop and compressing it into a single line. The "term" at immediately after the opening square bracket is the value that is actually passed to terms. Rather, it is the list of all such terms that is passed. Very slick.

Despite our fancy move the tool still prints out a lot of junk though.

```
[('.', 73), ('#Trump', 70), (':', 70), ('RT', 63), ('…', 45)]
```

We need a stopword list. The NLTK has one that is easy to import. We also need to strip punctuation so we'll need the string library, modifying the file as follows:

```
from collections import Counter
from nltk.corpus import stopwords
import string

punctuation = list(string.punctuation)
stop = stopwords.words('english') + punctuation + ['rt', 'via', 'RT', '…']


with open("trumpTweets.json", "r") as inputFile:
    count_all = Counter()
    for line in inputFile:
        tweet=json.loads(line)
        terms = [term for term in preprocess(tweet['text']) if term not in stop]
        count_all.update(terms)
    # Print the first 5 most frequent words
    print(count_all.most_common(5))
```

This is pretty elegant. There is still some junk output though.

> *Change what we have written so far to remove the terms you don't want, like "amp" (short for "ampersand").*

Note that this kind of tuning is inevitable and happens on a task by task basis.

Plotting

Once we have the data as we want it is time to look at plotting a chart of top words (Note that with a few tweaks we could be charting users with the most tweets, tweets with the most likes, etc.). We'll use a snippet in the notebook to give us a boost.

```
# Print the current version of the chart
%matplotlib inline

count_all_dict = dict(count_all.most_common(5))
import matplotlib.pyplot as plt
plt.figure(figsize=(15, 6))
plt.bar(range(len(count_all_dict)), count_all_dict.values(), align='center')
plt.xticks(range(len(count_all_dict)),
list(count_all_dict.keys()),rotation='vertical')

plt.show()
```

> *what happens when we modify or even remove the plt.figure line?*

Try it.

> *how does range(len(count_all_dict)) work?*

3. Advanced Analysis

Here we are going to do things a littel differently and just provide the entire solution up front. At this point in a 2-day course with only a day an Python it is unlikely that a full walkthrough will be possible. They'll also likely be both tired and bored of that approach. So, here we'll look at a well commented script and look to add print statements at various places to see what is happening.

4. Drinking from the Stream

1. Create a twitter account.
2. Register a project/app at https://apps.twitter.com/
3. Connect to stream in segments
4. Watch stream in real-time
5. Save data
6. Use this data with previously created tools.

This consol looks useful for learning/experimenting with the API:
https://dev.twitter.com/rest/tools/console

Will need to write this up. Won't be in time for the first round of this course.

Old Content

[Before we begin participants will need to download a file like `20160308163459-DH2016.json` from the course GitHub repository]

This is a really excellent resource to use when preparing this lesson. It is the first of a series.

This is the official NLTK twitter doc.

Planning our Program

For us this means understanding the problem and being very clear what the steps are towards creating a solution. A step is really any part of the program for which we should carry out independent testing to make sure that it works. This will make the construction of our programs *iterative*.

Most programs can be usefully divided into three major parts:

1. Reading data.
2. Manipultating data.
3. Writing/Displaying data.

So, what we need to do now is add comments that will act as placeholders/sign-posts/instructions as we build the program. So, we add the following to the top of the file:

```
#Task Description: load a corpus of tweets and print a barplot of the most
frequently used terms

#Open the file

#Load the data from the file into a variable

#Slice up the body text of the tweets

#Count the terms

#Produce the chart
```

Here is a nice chart tutorial

Opening & Reading Files

Reading data from files is essential for most computing tasks. There are a number of ways to open a file within Python and we'll look at three here, quickly covering the most basic so that you understand the fundamentals and then jumping to a more advanced approach that will be helpful.

1. Open a file, look at it's type, then read it, line by line, then close it.
2. Open a file using a specific library to do so (CSV), then close it.
3. Use `with`. At this stage actually open the JSON file

Let us open the file with the sample tweet set:

```
open('20160308163459-DH2016.json')
```

What we have just done is run a *function* in Python, passing it a string to use as input. The `open` function is used to open connections to files so that we can read and write to them. The string (of characters) that we passed the open function is the filename. Since it is just a filename and not a full directory or path the open function will look in the directory that the current file/notebook is being run from.

Running this function will produce output that looks like the following:

```
<_io.TextIOWrapper name='20160308163459-DH2016.json' mode='r' encoding='UTF-8'>
```

What we have done is create a connection to the specified file and what is being reported here

are the details of that connection. We can see that it is using the TextIOWrapper method of the Python io (for "input/output") library, the name of the file, that the mode of the connection is 'r' (for "read only"), and the format the contents of the file are expected to be read in.

We have not actually read the data in the file, only opened a connection to the file. To read the file we need to create a container to hold the data and copy content from the file into the container line by line.

Mention three types of open files (r,w,a), point out that 'r' is the default but emphasize the importance of getting in the habit of being explicit.

NLTK

I now have an iPython/Jupyter notebook that has the rough work in it to load Twitter data from a file and plot the most frequent terms. It could use some tweaking/clean-up but it will serve as a place holder for the moment.

Mapping locations may be a problem since it seems that most DH related tweets don't have geotagging info in them. At least not so far...

[Within the NLTK is a large corpora of corpora. This should be shown and the differences from raw text sources made plain but it should not be the primary focus, that must remain on the twitter feed.]

Sentiment Analysis http://www.laurentluce.com/posts/twitter-sentiment-analysis-using-python-and-nltk/

TextBlob

[Saw this at the UCalgary course where it was used to quickly do some sentiment analysis. This would be really nice to do here combined with a graph/chart.

See http://textminingonline.com/getting-started-with-textblob (but of course there are other options as well.)]

https://pythonprogramming.net/twitter-sentiment-analysis-nltk-tutorial/

Neo4j (or other Network Graphing Tool)

I think there is a neo4j library for this (actually, it seems that there isn't). Be nice to track the connections within the twitter sphere.

http://mark-kay.net/2014/08/15/network-graph-of-twitter-followers/

Perhaps use graph-tool instead? Or NetworkX. NetworkX seems to be the better option for ease of install and nice documentation (Actually, Graph tool also has very nice documentation).

Graph-Tool is easily installed via ports but not easy for Windows users to install.

```
$ conda install networkx
```

But how to do this on Windows???

Seems that the command prompt is used.

Mallet

(Is there such a library for this? Could we show how to call it from the command line within Python? If the latter is where we go then this might warrant a separate lesson...)

pymallet seems to be an option but there is no documentation for it. =(

Advanced

This will be a call-out section that will cover how to set-up the system to:

- pull directly from the Twitter stream
- Use the `vincent` library for javascript/D3 compatibility

  $ conda install vincent

- sometimes `conda` doesn't work as a good install option. `pip` might be a substitute. For example:

```
Johns-MacBook-Pro:~ simpson$ conda install textblob
Using Anaconda Cloud api site https://api.anaconda.org
Fetching package metadata: ....
Solving package specifications: .
Error:  Package missing in current osx-64 channels:
  - textblob

You can search for this package on anaconda.org with

    anaconda search -t conda textblob
Johns-MacBook-Pro:~ simpson$ anaconda search -t conda textblob
Using Anaconda Cloud api site https://api.anaconda.org
Run 'anaconda show <USER/PACKAGE>' to get more details:
Packages:
     Name                      |  Version | Package Types   | Platforms
     ------------------------- |   ------ | --------------- | ---------------
     MickC/textblob            |    0.9.0 | conda           | linux-64
                                          : Simple, Pythonic text processing.
Sentiment analysis, POS tagging, noun phrase parsing, and more.
     chdoig/textblob           |    0.9.0 | conda           | linux-64, linux-
```

```
32, osx-64
                                        : Simple, Pythonic text processing.
Sentiment analysis, POS tagging, noun phrase parsing, and more.
     derickl/textblob          |    0.9.0 | conda           | osx-64
                                        : Simple, Pythonic text processing.
Sentiment analysis, POS tagging, noun phrase parsing, and more.
     hargup/textblob           |          | conda           | linux-64
                                        : Simple, Pythonic text processing.
Sentiment analysis, POS tagging, noun phrase parsing, and more.
     hisanor/textblob          |   0.11.1 | conda           | linux-64
                                        : Simple, Pythonic text processing.
Sentiment analysis, part-of-speech tagging, noun phrase parsing, and more.
     jacksongs/textblob        |   0.11.0 | conda           | osx-64
                                        : Simple, Pythonic text processing.
Sentiment analysis, part-of-speech tagging, noun phrase parsing, and more.
     memex/textblob            |   0.10.0 | conda           | linux-64, osx-64
     msarahan/textblob         |   0.10.0 | conda           | osx-64
     pdbaines/textblob         |    0.8.4 | conda           | linux-64
                                        : Simple, Pythonic text processing.
Sentiment analysis, POS tagging, noun phrase parsing, and more.
     sloria/textblob           |   0.11.0 | conda           | osx-64
                                        : Simple, Pythonic text processing.
Sentiment analysis, POS tagging, noun phrase parsing, and more.
     sursma/textblob           |    0.8.4 | conda           | win-64
                                        : Simple, Pythonic text processing.
Sentiment analysis, POS tagging, noun phrase parsing, and more.
     timka/textblob            |    0.9.0 | conda           | win-64
                                        : Simple, Pythonic text processing.
Sentiment analysis, POS tagging, noun phrase parsing, and more.
     trifacta/textblob         |    0.9.0 | conda           | linux-64
                                        : Simple, Pythonic text processing.
Sentiment analysis, POS tagging, noun phrase parsing, and more.
     zhenghao/textblob         |    0.9.1 | conda           | osx-64
                                        : Simple, Pythonic text processing.
Sentiment analysis, POS tagging, noun phrase parsing, and more.
Found 14 packages
Johns-MacBook-Pro:~ simpson$ anaconda show hisanor/textblob
Using Anaconda Cloud api site https://api.anaconda.org
Name:    textblob
Summary: Simple, Pythonic text processing. Sentiment analysis, part-of-speech
tagging, noun phrase parsing, and more.
Access:  public
Package Types:  conda
Versions:
   + 0.11.1

To install this package with conda run:
```

```
     conda install --channel https://conda.anaconda.org/jacksongs textblob

Johns-MacBook-Pro:~ simpson$ conda install --channel
https://conda.anaconda.org/jacksongs textblob
Using Anaconda Cloud api site https://api.anaconda.org
Fetching package metadata: ......
Solving package specifications: ....

The following specifications were found to be in conflict:
  - conda -> conda-env
  - conda -> pycosat
  - conda -> python 2.7*
  - conda -> pyyaml
  - conda -> requests
  - conda-env (target=conda-env-2.4.5-py35_0.tar.bz2) -> python
2.7*|3.3*|3.4*|3.5*
  - mkl-service (target=mkl-service-1.1.2-py35_0.tar.bz2) -> python
2.6*|2.7*|3.3*|3.4*|3.5*
  - nltk (target=nltk-3.1-py35_0.tar.bz2) -> numpy *|1.5*|1.6*|1.7*|1.8*|1.9*
  - nltk (target=nltk-3.1-py35_0.tar.bz2) -> python 2.6*|2.7*|3.3*|3.4*|3.5*
  - nltk (target=nltk-3.1-py35_0.tar.bz2) -> pyyaml
  - nltk (target=nltk-3.1-py35_0.tar.bz2) -> six
  - nose (target=nose-1.3.7-py35_0.tar.bz2) -> python 2.6*|2.7*|3.3*|3.4*|3.5*
  - numexpr (target=numexpr-2.4.6-np110py35_1.tar.bz2) -> numpy
1.10*|1.11*|1.6*|1.7*|1.8*|1.9*
  - numexpr (target=numexpr-2.4.6-np110py35_1.tar.bz2) -> python
2.6*|2.7*|3.3*|3.4*|3.5*
  - numpy (target=numpy-1.10.4-py35_0.tar.bz2) -> python
2.6*|2.7*|3.3*|3.4*|3.5*
  - pip (target=pip-8.1.0-py35_0.tar.bz2) -> python 2.7*
  - pip (target=pip-8.1.0-py35_0.tar.bz2) -> setuptools
  - pycosat (target=pycosat-0.6.1-py35_0.tar.bz2) -> python
2.6*|2.7*|3.3*|3.4*|3.5*
  - python 3.5*
  - pyyaml (target=pyyaml-3.11-py35_1.tar.bz2) -> python
2.6*|2.7*|3.3*|3.4*|3.5*
  - requests (target=requests-2.9.1-py35_0.tar.bz2) -> python
2.6*|2.7*|3.3*|3.4*|3.5*
  - scikit-learn (target=scikit-learn-0.17-np110py35_2.tar.bz2) -> numpy
1.10*|1.11*|1.5*|1.6*|1.7*|1.8*|1.9*
  - scikit-learn (target=scikit-learn-0.17-np110py35_2.tar.bz2) -> python
2.6*|2.7*|3.3*|3.4*|3.5*
  - scikit-learn (target=scikit-learn-0.17-np110py35_2.tar.bz2) -> scipy
  - scipy (target=scipy-0.17.0-np110py35_0.tar.bz2) -> numpy
1.10*|1.11*|1.5*|1.6*|1.7*|1.8*|1.9*
  - scipy (target=scipy-0.17.0-np110py35_0.tar.bz2) -> python
2.6*|2.7*|3.3*|3.4*|3.5*
```

```
  - setuptools (target=setuptools-20.2.2-py35_0.tar.bz2) -> python
2.6*|2.7*|3.3*|3.4*|3.5*
  - six (target=six-1.10.0-py35_0.tar.bz2) -> python 2.6*|2.7*|3.3*|3.4*|3.5*
  - textblob -> nltk
  - textblob -> python 2.7*
  - wheel (target=wheel-0.29.0-py35_0.tar.bz2) -> python 2.7*|3.3*|3.4*|3.5*
Use "conda info <package>" to see the dependencies for each package.

Johns-MacBook-Pro:~ simpson$ pip install textblob
Collecting textblob
  Downloading textblob-0.11.1-py2.py3-none-any.whl (634kB)
    100% |████████████████████████████████| 634kB 636kB/s
Requirement already satisfied (use --upgrade to upgrade): nltk>=3.1 in
./anaconda/lib/python3.5/site-packages (from textblob)
Installing collected packages: textblob
Successfully installed textblob-0.11.1
```

*args and **kwargs in Python functions explained

- Function with a while loop with a conditional

```
def find(word, letter)

    index = 0

    while index < len(fruit)

        if word[index] == letter:

            return index

        return -1

    print find(fruit, ‘a’)
```

- In

```
if ‘a’ in fruit:

    print fruit
```

Writing Data

Not sure that there will be data to output here. If there is then just use `with`. When we