Python in Data Science

IDLE used: Google Colab, PyCharm

First Program

```
print("Hello, World!")
```

```
Hello, World!
```

Skeleton Program

Here is a simple template for the major elements of any program. It's a good idea to follow these conventions, since they were developed by the Python community to make code easier to read.

```
"""
Title of Your Program

Brief explanation of what it does. Describe any FILES or resources that your program needs, as well as what it produces/RETURNS.
"""

# ___________________ Import statements

import string


# ___________________ Global variables
# Globals are variables that are available to everything in the program
# you can INITIALIZE it if you like: that is, assign a value to it

my_int = 0          # an integer with the value 0
my_string = ""      # an empty string
my_list = []        # an empty list
my_dict = {}        # an empty dictionary
my_boolean = True


# ___________________ Functions
# functions are mini-programs inside programs. You can write them
# inside the program or you can write them as separate programs
# then import them using the import statement above


def my_function(name):
    if type(name) == str:       # make sure the variable holds a string
        name = name.upper()     # takes the value passed in and uc's it
    return name                 # return name to whatever called the function


# _______________________________________ MAIN
# this section is where you do all your business!

name = "steve"          # assign a value to the variable

name2 = my_function(name)   # call the function, send it the name
                            # put the returned value into a new variable

print(name2)            # print whatever we got back
```

Variables, Lists, Dictionaries, Tuples

```
# Variables

x = 3                   # int or integer

y = "Hello, World!"     # str or string

z = True                # boolean, includes True/False, None

my_list = [1, "hello", True]    # list, which can include any kind of variable
my_list2 = [1, ["hello", "world"], 3]

print(my_list2[1][0])

my_dictionary = {"eggs": 2, "bacon": True}  # key:value, key:value

# and one I never use:

my_tuple = (1, 2)       # a list of constants, which cannot be changed
```

We can access a variable:

```
print(x)

print(y)

print(y[0])     # every string is stored in memory as a list, so you can get a letter using an index value
print(my_list)

print(my_list[1])   # get an item in your list by passing the index value to the print statement

print(my_dictionary)

print(my_dictionary["bacon"])   # get an item by passing the key to print
```

```
3
Hello, World!
H
[1, 'hello', True]
hello
{'eggs': 2, 'bacon': True}
True
```

The print statement can take multiple arguments:

```
print("The value of x is ", x)

print("The second value in my list and dictionary is ", my_list[1], " and ", my_dictionary["bacon"])
```

Searches

Most of the DS programs we will write are complicated searches.

The easiest search is this: we ask the user for some word or phrase, then search a text for it.

```
search_term = input("Word to search: ")     # Puts up a prompt for the user. Notice the space at the end!

# now we will test our string y to see if the word is in it.

if search_term in y:
    print("Found it!")
```

Sometimes you want to search for a word without worrying about capital letters.

Python includes **methods**, which are things you can do to a string:

- lower()
- upper()
- capitalize()
- swapcase()

You can also test a string to see if it's a number:

string.isnumeric()

Or to see if it's in lower-case:

string.islower()

python.org has a list of all the methods on strings

```
print(y.lower())    # don't forget the two parentheses to indicate it's a function


print(y.upper())
```

Split String into Words, Then Search Within Word

```
frost = """AFTER APPLEPICKING
My long two-pointed ladder's sticking through a tree
Toward heaven still,
...
"""
```

```
print(frost)

Words = frost.split()
print(words[2])
print(frost[50])

for word in words:
    if "o" in word:
        print(word)
```

User Input and String Word Search

```
"""
My first program. Takes a line of poetry, a user search term,
and returns a string if the term is in the line.
"""

x = input("Enter word you want to find: ")    # asks a user for input.
y = "Let us go then you and I"   # from T. S. Eliot

if x.lower() in y.lower():      # ensure term and line are all in lower-case
    print("Yes!")

print("All done")
```

```
What word do you want to find? LeT
Yes!
All done
```

How to open Files

Opening files is much easier when python is run on your own computer. When we run it on this Google site, it gets much more complicated.

```
with open(myfile.txt","r") as fh:
    text=file.read()
    text=file.readlines()
    file.close()
```

Python comes with **PACKAGES**. These are collections of pre-written code. People can add packages of their own and make them freely available to other python coders, like you. Google wrote a package called **files**.

It contains **functions** that you can use.

To import a a package into your program, the syntax is as follows:

from SITE import PACKAGE

or

from PACKAGE import CLASS

Then, to use the function, the syntax is as follows:

package.function()

```
"""To upload a file from your own computer"""

from google.colab import files
uploaded = files.upload()       # triggers a file window on your computer

# now the variable named uploaded contains the data in your file:

print(uploaded)

# What kind of variable is uploaded? Is it a string, list, or dictionary?

type(uploaded)

# It's a dictionary, that is a list where you name your own keys {key: value}
# the KEY here is the name of the file you uploaded
```

Loops

Sometimes you want to do one thing only. That's called a **statement**.

But sometimes you want to do things over and over. That's a **LOOP**. (It's technically a statement, too.)

There are two major LOOPs.

- For Loop
- While Loop

```
for item in my_list:        # goes through each item in your list one-by-one
    print(item)             # prints that one item, then loops back and gets the next item
```

A **while loop** tests to see if a statement is True. If the statement is True, the loop continues.

```
while x == 3:           # asks if x holds the value 3 (which it does at first)
    print(x)            # prints x if the while statement is True
    x = x - 1           # subtracts 1 from x. Now the while test is False, and print won't fire.
```

You can break out of a loop at any time by using break

You can do nothing inside a loop by using pass or continue

```
for letter in y:
    if letter == "!":       # once python sees the "!" it will break out of the loop and won't print "!"
        break
    if letter != "o":       # != means "not equal to".  ; ! = "bang"
        print(letter)
    if letter == "o":       # == means "equal to" (remember that = assigns a value)
        print("OH")
```

Loops with tests are very useful.

Let's say you want to go through a list of customers who have spent more than $100.00 and send them a coupon.

You have a dictionary for each customer called mycustomers:

```
if my_customers["total_spent"] > 100

  send_email(my_customers["email"])
```

Or you could send an email to a student who is getting less than a C. And so on.

Variable

A named space in memory with an **address**. It comes in different **types**. We are interested in

- **int** integer (1, 2, 3, 4, and all real numbers)
- **float** float (anumber with a decimal point, 3.1416)
- **str** string ("a", 'b', "Hello world!")

```
x = 3
print(x)
print(type(x))
print ("The memory address of x is ", hex(id(x)))

y = "Hello, World!"
print(y)
```

```
3
<class 'int'>
The memory address of x is  0x5570f80c0a40
Hello, World!
```

Lists

A sequence of variables. Called an **array** in other languages. Indexed by position.

```
letters = list("hello")    # python automatically ITERATES through the string, taking each letter independently
print(letters)
print(letters[0])           # each letter is INDEXED. 0 is the first index. So print the first letter

# so ... you can print
name = "slowly"
print(name[2])          # print the third letter
print(name[-2:])        # print the second-to-last letter up to the end
```

```
['h', 'e', 'l', 'l', 'o']
h
o
ly
```

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

vowels = ["a", "e", "i", "o", "u", "y"]

# Some things are lists that you don't expect to be lists. Like words and sentences.
# You can also make a list by using the built-in function list()

letters = list("hello")    # python automatically ITERATES through the string, taking each letter independently
print(letters)
# You can see how the word has been turned into a list!

print(letters[0])           # each letter is INDEXED. 0 is the first index. So print the first letter

# so ... you can print
name = "slowly"
print(name[2])          # print the third letter
print(name[-2])        # print the second-to-last letter
print(name[-1])         # print the last letter
```

Dictionaries

Like lists. Where lists are indexed automatically using the numbers 0 1 2 3 etc., dictionaries let you invent your own indexes.

So, a list might be user = ["Sam", 020034, "sam@place.com"]

And user[0] returns "Sam"

With a dictionary, you can say:

user = { "name":"Sam", "id":020034, "email":"sam@place.com"}

And user["name"] returns "Sam"

```
# Let's say we want to record the publishing information of a book

# We can put each part on a different line. NB putting a comma after the
last element doesn't hurt, and sometimes helps

moby = {
    "author":"Melville",
    "title":"Moby Dick",
    "date":1851,
}

print(moby["date"])
```

Codes as a Dictionary

Instead of loading a file with the parts-of-speech codes, we can just put them into a dictionary variable

```
code_dictionary = {"CC": "coordinating conjunction",
"CD": "cardinal digit",
"DT": "determiner",
"EX": "existential there",
"FW": "foreign word",
"IN": "preposition/subordinating conjunction",
"JJ": "adjective (large)",
"JJR": "adjective, comparative (larger)",
"JJS": "adjective, superlative (largest)",
"LS": "list market",
"MD": "modal (could, will)",
"NN": "noun, singular (cat, tree)",
"NNS": "noun plural (desks)",
"NNP": "proper noun, singular (sarah)",
"NNPS": "proper noun, plural (indians or americans)",
"PDT": "predeterminer (all, both, half)",
"POS": "possessive ending (parent\ 's)",
"PRP": "personal pronoun (hers, herself, him,himself)",
"PRP$": "possessive pronoun (her, his, mine, my, our )",
"RB": "adverb (occasionally, swiftly)",
"RBR": "adverb, comparative (greater)",
"RBS": "adverb, superlative (biggest)",
"RP": "particle (about)",
"TO": "infinite marker (to)",
"UH": "interjection (goodbye)",
"VB": "verb (ask)",
"VBG": "verb gerund (judging)",
"VBD": "verb past tense (pleaded)",
"VBN": "verb past participle (reunified)",
"VBP": "verb, present tense not 3rd person singular(wrap)",
"VBZ": "verb, present tense with 3rd person singular (bases)",
"WDT": "wh-determiner (that, what)",
"WP": "wh- pronoun (who)",

"WRB": "wh- adverb (how)"}
```

```
# then make the relevant changes in the function definition and run the
code again.
```

Dictionaries and Lists and Data Sets

We can look for interesting data sets on Kaggle.com

Here's an interesting data set: the National Research Council of Canada's Word-Emotion Association Lexicon: https://saifmohammad.com/WebPages/NRC-Emotion-Lexicon.htm

Pick a word and it tells you which of several basic emotions it is associated with.

STRUCTURE:

'# Word', 'Positive', 'Negative', 'Anger', 'Anticipation', 'Disgust', 'Fear', 'Joy', 'Sadness', 'Surprise', 'Trust\n']

(You must upload the file NRC-emotions.txt)

```
with open('NRC-emotion.txt', 'r') as fh:
    nrc_lines = fh.readlines()
    fh.close()

# now each line is in a list variable called nrc_lines
print(nrc_lines[1])

# we need to go through each line and split it at the commas
# then put the elements into a list

words = []

for line in nrc_lines:
    temporary_elements = line.split(",")
    words.append(temporary_elements)

print(words[3])


# use csv module

import csv

with open('NRC-emotion.csv', 'r') as csv_file:
    csv_reader = csv.reader(csv_file)
    for row in csv_reader:
        print(row)
```

We finally have a **list** that we can use. The first (zero) element is the word.

One problem is that the last element contains a new line. Also notice that the elements are **strings**, not numbers!

The word *aback* is neither negative nor positive, and it has no clear associations with any of the emotions listed.

Now let's look for words that have both surprise and anger:

```
for word in words:
    if "1" in word[9] and "1" in word[3]:
        print(word[0])
```

TESTS

These are the diamonds in your flowchart.

The most important is the **if ... then** statement. You ask **if true**, then do something. Or **if false**, then do something.

Python resolves the test first. If the test resolves to TRUE, then it executes the statement after the colon (:). True includes:

• Mathematical tests (5 > 3)

• comparison of strings ("hi" == "hi)

• existence of a variable (if name)

• checking if a list contains an element (if x in list)

```
# DIAMOND: test
# remember from above: vowels = ["a", "e", "i", "o", "u", "y"]

if "a" in vowels:
  print("True")
else:
    print("False")
```

FUNCTIONS + search text for word!

Functions are like food trucks. Stuff gets cooked in them!

You **pass in** an order, and the cooks **return** your food.

Everything used within the food truck stays in the food truck. No pans or whisks or measuring cups leave the truck! In other words, all variables that you use are **local** to the function--they stay in the function and cannot be used outside it.

```
# the shape of a function is:
#   def function_name():
#       action

# you can pass variables to the function:
#   def function_name(local_variable):
#       action
#       return value        and you can return something from the
function!

# ___________________________________ FUNCTION DEFS


def cap(word):
    # capitalizes a word
  print(word.capitalize())
  print("Reached the cap function.")    # test to see you got into the function
```

```
def check_a(word2):
  # checks a word for the letter A
  if "a" in word2:
    print("There's an A!")
  else:
    print("No A")


  if len(word) > 5:
    print("Greater than five.")
  else:
      print("Less than or equal to 5.")

# ___________________________________ MAIN

test = input("Input a word ")
check_a(test)
```

example:

Now let's check a word for vowels

```
# search for vowels

# ___________________________________ FUNCTIONS


def make_lower(user_input):
    print("Inside make_lower function!")
    return user_input.lower()


# ___________________________________ DATA

vowels = ["a", "e", "i", "o", "u", "y"]         # make a list of vowels so the computer knows what they are
word2check = input("What is your word? ")       # get the user's word

# ___________________________________ MAIN

word2check_lower = make_lower(word2check)       # we will make the user's word lower case since searches are case-sensitive
print(word2check_lower)                         # check that we have the user's word

for letter in word2check_lower:             # now we will loop through each letter of the word and see if it's a vowel
    if letter in vowels:
        print(letter, end=" \t")            # print takes an 'end' parameter, which lets you print in a straight line



# ___________________________________ Challenge!
# see if you can search a poem for prepositions. First you have to find a list of them!
poem = """
            """
preps = ["about", "above", "across", "after","against",
         "along", "among", "around", "at", "before", "behind",
         "between", "beyond", "but", "by", "down", "for", "from",
         "in", "into", "like", "near", "of", "off", "on", "onto", "out",
         "over", "past", "plus", "throughout", "to", "towards",
         "under", "until", "up", "upon", "up", "to", "with", "within",
              "without"]

poem_words = poem.split()
# print(poem_words)
num_preps = 0

for word in poem_words:
    if word in preps:
        print(word)
        num_preps += 1

percent_preps = (num_preps/len(poem_words))*100
percent_preps = int(percent_preps)

print("There are ", num_preps, "prepositions in this poem.")
print("That is ", percent_preps, "percent.")
```

Import NLTK

First, let's remind ourselves of the zen of python:

```
import nltk
nltk.download("book")
from nltk.book import *

text2.concordance("excessive")

text2.collocations()

samples = ["whale", "fish", "cetacean", "Moby"]
```

```
fdist = FreqDist(samples)
fdist.items()

fdist.plot()
```

CHAPTER 1 List Comprehensions, p. 23

```
onset = "qu"
qu_words = [w for w in text1 if w.startswith(onset)]
print(sorted(set(qu_words)))
```

Notice that *quadruped* and *quadrupeds* are counted as two different words. So too are *quaff* and *quaffed*.

If we want to know how many vocabulary items a writer uses, then *quaff* and *quaffed* are really only one item.

The word *quaff* is the **lexeme**, which means the word as you would find it in the dictionary. You can add endings to *quaff* like -ed or -ing or -s.

Before you can get to the lexeme and get rid of all the extra -s, -ed, -ing and so on, you have to **tokenize** a sentence. Basically, you split it up into words. *Quaff* is a token, *quaffed* is a token, and a period is a token. NLTK has a tokenizing method.

```
from nltk.tokenize import word_tokenize

sentence = "Quaff and quaffed are the same word!"
word_tokenize(sentence)
```

What is the difference between tokenize and split()?

Let's take a look

```
print(sentence.split())
```

You can see the difference. Split() put the exclamation point next to *word*. So split() breaks things at white-spaces, but not necessarily at punctuation marks.

NLTK designed tokenize to isolate punctuation. There are other tokenizers that NLTK offers, such as PunctSentenceTokenizer.

Attempt to scrape with **BeautifulSoup**

First, import the built-in library **urllib** (url library), and specify the class **request**. Within that, import urlopen.

Then import Beautiful Soup.

Here is the documentation for Beautiful Soup:
https://beautiful-soup-4.readthedocs.io/en/latest/

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
```

Now that we have the tools, let's get the front page of UMass.edu

First, we use urllib's method urlopen to get the page. Then we pass it to Beautiful Soup. Beautiful Soup can parse the page into its component pieces. To do that, send Beautiful Soup 1) the url object html and 2) which parser you'd like to use, html.parser

```
url = "https://www.umass.edu"
html = urlopen(url)
bs = BeautifulSoup(html, 'html.parser')
```

Let's find all the links on the page.

Remember that an html link is coded like this: <a href="">link</a>

So we're looking for all <a> elements **(find_all('a'))**, then looping through those to find all href elements.

```
# print(bs.prettify())
for link in bs.find_all('a'):
  print(link.get('href'))
```

- For our rhyming program, here is the CMU_dictionary in json format. Caution: 6 Megs!

To call it:

```
import json
def pronounce(word):
   word = word.upper()
   value2 = cmu[word]
   return value2
print(pronounce('fish'))
```

```
import nltk
# nltk.download("book")
# from nltk.corpus import gutenberg
gutenberg.fileids()
from nltk.corpus import brown
news_text = brown.words(categories='news')
fdist = nltk.FreqDist([w.lower() for w in news_text])
print("Number of word in total:", len(fdist))
modals = ['can', 'could', 'may', 'might', 'must', 'will', 'would']
for m in modals:
  print(m, ":", fdist[m], "\t", fdist[m]/len(fdist))
```

Example Web Scraping with BS4

```
import requests
from bs4 import BeautifulSoup
from textblob import TextBlob
import nltk
nltk.download('punkt')

# and to tag parts of speech, so let's get that:
nltk.download('averaged_perceptron_tagger')

url_base = "https://www.poetryfoundation.org/"
poem = "/poems/51652/tonight-56d22f898fcd7"
url = url_base + poem
print(url)
html_doc = requests.get(url)
bs = BeautifulSoup(html_doc.content, 'html.parser')

all_divs = bs.find_all("div", class_ = "c-feature-hd")

for this_div in all_divs:
    poem_title = bs.find_all("h1", class_ = "c-hdgSans c-hdgSans_2 c-mix-hdgSans_inline")
#print(poem_title)
```

```
# find every div with a style element that matches the poem's style
poem_content = bs.findAll("div", {"style" : "text-indent: -1em; padding-left: 1em;"})

# now loop through all the divs that we found and print their contents
cnt = 0
poemTxt = []
for content in poem_content:
    cnt = cnt + 1

print("Number of lines in the poem: ")
print(cnt)

poemTxt = []
for content in poem_content:
    poemTxt = content.text
    res = res + len(poemTxt.split())

print ("The number of words in are : " +  str(res))

poemTxt = []
res = 0
res2 = 0
for content in poem_content:
    poemTxt = content.text
    res = res + (poemTxt.count('tonight'))
    res2 = res2 + (poemTxt.count('God'))

print ("The number of times 'tonight' appears in the poem is " +  str(res) + " times")
print ("The number of times 'God' appears in the poem is " +  str(res2) + " times")
```

Conditional Frequency Distribution

```
from nltk.corpus import inaugural   # examples of available corpora (plural of corpus) on pp. 46-47
cfd = nltk.ConditionalFreqDist(
    (target, fileid[:4])
    for fileid in inaugural.fileids()
    for w in inaugural.words(fileid)
    for target in ['america', 'citizen']
    if w.lower().startswith(target))
cfd.plot()
```

Using HELP

```
help(nltk.corpus.reader)
# load a text file from Project Gutenberg
```

Load your own corpus

```
raw = gutenberg.raw("burgess-busterbrown.txt")
Raw[:30]
words = gutenberg.words("burgess-busterbrown.txt")
words[:20]
```

You define the path to your own corpus, calling it corpus_root
to use on colab, you need to mount your Google Drive first or upload the file to your runtime

```
from google.colab import drive
drive.mount('/content/drive')
from nltk.corpus import PlaintextCorpusReader
corpus_root = "/content/drive/My Drive/Colab Notebooks/texts/"

# !ls "drive/My Drive/Colab Notebooks/"
wordlists = PlaintextCorpusReader(corpus_root, '.txt')
# print(wordlists.words('gatsby.txt'))
gatsby = wordlists.words('gatsby.txt')
print(gatsby)
```

Get a text Directly from Gutenberg

see pages 79 to 81 in NLTK book

```
from urllib.request import urlopen      # CAUTION! not the same as in the book.
url = "https://gutenberg.org/files/2554/2554-0.txt"     # CAUTION! new url, not the same as in the book
raw = urlopen(url).read()
print(raw[:75])      # print the first 75 characters of the raw text

fd_tokens = nltk.tokenize(raw)
print(fd_tokens[:10])
```

CMU Dictionary

```
entries = nltk.corpus.cmudict.entries()
len(entries)
for entry in entries[39943:39951]:
  print(entry)
prondict = nltk.corpus.cmudict.dict()
```

```
prondict['fire']
```

```
[['F', 'AY1', 'ER0'], ['F', 'AY1', 'R']]
```

Swadesh List (translation)

```
from nltk.corpus import swadesh
fr2en = swadesh.entries(['fr', 'en'])

# make a dictionary from the lists
trans = dict(fr2en)
trans['chien']
```

WordNet

```
from nltk.corpus import wordnet as wn
wn.synsets('motorcar')
wn.synset('car.n.01').definition()
wn.synset('car.n.01').lemmas()
```

Hyponyms

```
motorcar = wn.synset('car.n.01')
types_of_motorcar = motorcar.hyponyms()
sorted([lemma.name() for synset in types_of_motorcar for lemma in
synset.lemmas()])
paths = motorcar.hypernym_paths()
len(paths)
[synset.name for synset in paths[0]]
```

Probability

We will use the random package built in to Python.

It has a method called choice that chooses between options in a list.

```
import random
```

Independent events

In the following code, the second flip is completely independent of the first.

If your first flip is heads, it makes no difference to the second flip.

```
heads = 0
tails = 0

two_heads = 0
two_tails = 0
one_each = 0
```

```
random.seed()      # seed initializes the random number generator, and
uses the system time as a starting number.

max_range = 10      # change this value for more flips


def random_flip():
    return random.choice(["H", "T"])

for _ in range(max_range):             # the underline is a built-in
variable. It holds whatever value the loop is at now.

    first = random_flip()
    print(first, end=" ")
    if first == "T":
        tails += 1
    else:
        heads += 1

    second = random_flip()
    print(second, end="\t")
    if second == "H":
        heads += 1
    else:
        tails += 1

    if second == "T" and first == "T":
        two_tails += 1
        print("Two tails")
    if second == "H" and first == "H":
        two_heads += 1
        print("Two heads")
    if second != first:
        one_each += 1
        print("One each")

    # clear values
    first = ""
    second = ""



print("\nTwo heads: {}".format(two_heads/max_range))
print("Two tails: {}".format(two_tails/max_range))
print("One each: {}".format(one_each/max_range))

print("\nTotal heads: {}".format(heads))
print("Total tails: {}".format(tails))

# To increase accuracy, change the value of max_range to 10000.
# And comment out the first 5 print statements!
```

Bayes Algorithm

Bayes is too much engine for such a little job. The probability of flipping tails having just flipped heads is 0.5, and vice versa. We could get a huge list of flipped pairs and guess the next one, but because there's no correlation between one event (a flip) and the next—we're dealing with discrete events—we're not going to get a meaningful linear or polynomial function.

p(E|F) = p(E|F) * p(E) / (p(F|E) * p(E) + p(F|not E) * p(not F))

TextBlob

A "wrapper" for NLTK. Some people like the commands more than the ones in NLTK. Nevertheless, you still have to import nltk.

```
from textblob import TextBlob

# we also need the package punkt from nltk:
import nltk
nltk.download('punkt')

# and to tag parts of speech, so let's get that:
nltk.download('averaged_perceptron_tagger')
gravity = TextBlob('A screaming comes across the sky.')     # tb is an
object, here it's a function that takes a string or file

print(gravity.tokens)    # print all the items in the sentence, including
punctuation

print(gravity.words)     # print only the words in a sentence

# Question: How do we get the number of nouns in a sentence?
# Question: How do we get the number of words in a sentence?
```

Parts of Speech

TextBlob will parse a sentence for you, that is it will get all the parts of speech. But you have to loop through each word.

To find out the parts of speech, you have to look up the codes: https://www.geeksforgeeks.org/python-part-of-speech-tagging-using-textblob/

JJ is adjective
NN is noun
VB is verb

```
def adjectives(text):
    count = 0
    for (word, tag) in text.tags:
        if tag == 'JJ':
            count = count + 1
    return count
```

```
print('Number of adjectives: ', adjectives(gravity))
```

Our function is inefficient: it only counts adjectives. Try to write one that counts a variety of parts of speech.

```
# first, let's load the codes in json format
# upload "TextBlobCodes.json "
# or go to the bottom of this page and use code_dictionary

# import json

# with open('TextBlobCodes.json') as json_file:
#     codes = json.load(json_file)

    # print(codes['JJ'])

def speech_part(sentence):
    # results = []        # we can use this to hold our results
    for (word, tag) in sentence.tags:
        print("{}: {}".format(word, codes[tag]))

speech_part(gravity)
```

Sentiment Analysis with TextBlob

TextBlob also has a built-in sentiment analysis. And you can build your own.

```
print(gravity.sentiment)

bad_thought = TextBlob('Fish are nasty and smelly.')
print(bad_thought.sentiment)
```

More sentiment analysis tools are VADER and FLAIR. See https://neptune.ai/blog/sentiment-analysis-python-textblob-vs-vader-vs-flair

Check Sentiment of text

In this program, we will load the data set of emotion words from the National Research Council of Canada.

Then we will load a second text and see what emotions each word provokes.

Upload the file NRC-emotion.csv

```
# ___________________________ import statements
import csv

# ___________________________ data
with open('NRC-emotion.csv', 'r') as csv_file:
    csv_reader = csv.reader(csv_file)
    # put the data in csv_reader (which is now just a memory address) into
a list
    nrc_rows = []
    for row in csv_reader:
        nrc_rows.append(row)

    csv_file.close()


# check to make sure it worked, print first 5 rows:
print(nrc_rows[1:6])
```

Note the range notation in our print statement, [1:6].

That means [START:END]

Now we have all the data we need. Or so we think!

What if a poem uses the word *abandons* (with an s)? Or *abandoning*? We'll fix that later. Meanwhile, remember that the NRC data is organized like so:

'Word', 'Positive', 'Negative', 'Anger', 'Anticipation', 'Disgust', 'Fear', 'Joy', 'Sadness', 'Surprise', 'Trust']

```
# we'll worry about that later! For now, let's get a text

with open('apple2.txt', 'rt') as fh:
    poem_lines = fh.readlines()
    fh.close()

# check that it worked:
print(poem_lines)
```

We don't want to bother with the often-used short words of English like *to* and *and* and so forth. These are called **stopwords**. SO let's ignore them

```
# _________________________________ GLOBAL variables
stopwords = [
             "a", "about", "above",...
]

# and the last thing we need are counters for the emotions:
positive = 0
negative = 0
anger = 0
anticipation = 0
disgust = 0
fear = 0
joy = 0
sadness = 0
surprise = 0
trust = 0

# and for the total words of the poem
total_words = 0
```

```
# _________________________________ MAIN

# now let's go through each word of the poem and see what its emotions are

for line in poem_lines:
    line_words = line.split()    # now we have one line split into words

    # let's update the count of the poem's words
    total_words += len(line_words) # add the number of words in this line
to the total

    # now we have to go through each word
    for this_word in line_words:
        # and check every row in the NRC list
        if this_word in stopwords:
            continue
        for row in nrc_rows:
            if this_word == row[0]:
                print(row[0:10])   # print all the data for each word
                # now let's update the emotion counters
                if '1' in row[1]:
                    positive += 1
                if '1' in row[2]:
                    negative += 1
                if '1' in row[3]:
                    anger += 1
                if '1' in row[4]:
                    anticipation += 1
                if '1' in row[5]:
                    disgust += 1
                if '1' in row[6]:
                    fear += 1
                if '1' in row[7]:
                    joy += 1
                if '1' in row[8]:
                    sadness += 1
                if '1' in row[9]:
                    surprise += 1
                if '1' in row[10]:
                    trust += 1

# and let's print what we've got so far
print("\n______________________\n")
print("Total words:", total_words)
print("Positive: ", positive, "\tPrecentage: ", positive/total_words)
print("Negative: ", negative, "\tPrecentage: ", negative/total_words)
print("\nAssociated with: \nAnger: ", anger)
print("Anticipation: ", anticipation)
print("Disgust: ", disgust)
print("Fear: ", fear)
print("Joy: ", joy)
print("Sadness: ", sadness)
print("Surprise: ", surprise)
print("Trust: ", trust)
```

Sentiment Analysis with Vader Sentiment Tool

Let's use the Vader Sentiment Tool

From https://www.geeksforgeeks.org/python-sentiment-analysis-using-vader/

```
!pip install vaderSentiment

# import SentimentIntensityAnalyzer class

# from vaderSentiment.vaderSentiment module.

from vaderSentiment.vaderSentiment import SentimentIntensityAnalyzer



# function to print sentiments

# of the sentence.

def sentiment_scores(sentence):



  # Create a SentimentIntensityAnalyzer object.

  sid_obj = SentimentIntensityAnalyzer()



  # polarity_scores method of SentimentIntensityAnalyzer

  # object gives a sentiment dictionary.

  # which contains pos, neg, neu, and compound scores.

  sentiment_dict = sid_obj.polarity_scores(sentence)



  print("Overall sentiment dictionary is : ", sentiment_dict)

  print("sentence was rated as ", sentiment_dict['neg']*100, "% Negative")

  print("sentence was rated as ", sentiment_dict['neu']*100, "% Neutral")

  print("sentence was rated as ", sentiment_dict['pos']*100, "% Positive")



  print("Sentence Overall Rated As", end = " ")



  # decide sentiment as positive, negative and neutral

  if sentiment_dict['compound'] >= 0.05 :

    print("Positive")



  elif sentiment_dict['compound'] <= - 0.05 :
    print("Negative")



  else :

    print("Neutral")



# Driver code

if __name__ == "__main__" :



  print("\n1st statement :")

  sentence = BK_text[1020:1100]   # This is the sentence from our downloaded book



  # function calling

  sentiment_scores(sentence)



  print("\n2nd Statement :")      # This is from the Vader package to ensure it's working. Next one too.

  sentence = "study is going on as usual"

  sentiment_scores(sentence)



  print("\n3rd Statement :")

  sentence = "I am very sad today."

  sentiment_scores(sentence)
```

NLTK: Download from Gutenberg and Tokenize

We will download a book from Project Gutenberg.

Then we will tokenize it.

```
# the book says to ensure that all scripts begin with these lines:
from __future__ import division
import nltk, re, pprint
# if urllib can't be accessed; use urllib.requests
from urllib.request import urlopen
# NB Crime & Punishment is no longer available

url = "https://www.gutenberg.org/files/28054/28054-0.txt"
raw = urlopen(url).read()
raw[:75]
```

Note that raw begins with "\xef\xbb\xbf" which indicates that the text is in **BYTES**, not **UTF**. We'll need to change that when we use NLTK.

#Tokenization

```
# use nltk downloader to get punkt, which we can use later
nltk.download('punkt')
# turn raw from bytes into a UTF string
raw = raw.decode('utf-8')

tokens = nltk.word_tokenize(raw)
len(tokens)
tokens[:10]
BK_text = nltk.Text(tokens)
BK_text[1020:1100]
# nltk.collocations() uses STOPWORDS
nltk.download('stopwords')
BK_text.collocations()
```

Clean

Clean your text. Here are some basics. Please feel free to copy the code and add your own functions.

```
from nltk.tokenize import RegexpTokenizer
def punctuation(item0):
    # eliminate punctuation for analysis
    tokenizer = RegexpTokenizer(r'\w+')
    return tokenizer.tokenize(item0)


def caps(item2):
    # eliminate capitalization for analysis
    sendback = []
    for item in item2:
        sendback.append(item.lower())
    return sendback


def lines(item3):
    # clean blank lines etc
    sendhome = []
    for line in item3:
        templine = line.strip()
        if len(templine) < 1:
            pass
        else:
            sendhome.append(templine)
    return sendhome
```

Run these by opening a file, reading into memory, and sending it to the functions.

```
# with open('myfile.txt', 'r') as fh:
#     my_text = fh.read()
#     fh.close()

test = "Hello, World. Howaya?"

testwo = punctuation(test)
testhree = caps(testwo)
print(testwo, testhree)

final_text = ' '.join(testhree)
print(final_text)
```

Artificial Intelligence Basics

Markov Chain

In a Markov Chain, you take each element at a time. Let's call it X.

Then you search your data for all elements that follow X. Let's call it Y.

Finally you place the most common Y after X. Repeat.

Go to kaggle.com to find JSON datasets.

```
# Load data.

with open("frost.txt", "r") as fh:
    fh_raw = fh.read()
    fh.close()

# split the raw file into words
frost_words = fh_raw.split()
```

```
# check to make sure that it worked
print(frost_words[6812].lower())
```

Now that we have a list of words, let's pick a word at random.

Python has a built-in module called random.

https://www.w3schools.com/python/module_random.asp

```
import random

# start the random module by seeding it
random.seed()

# We need our first word

# assign a random number between 0 and the last word in the range
location_float = random.randrange(0, len(frost_words) - 1)

# That number is a float (has a decimal point), so let's turn it into an
integer:
loc = int(location_float)

our_seed = frost_words[loc].lower()

print(our_seed)
```

Pick the next word

We're going to want to do this over and over again, so let's write a function.

It needs to:

1. Find all the words that follow our quarry
2. Rank them by popularity
3. Choose the most popular (or 2nd most, or 3rd most, etc.)

As you'll see, *this function does not work*! Look at the documentation for index() and try to figure out why.

```
def frost_nextword(term):
    next_word = ""

    for word in frost_words:
        if word == term:
            # we can just grab the next word, except there might not be
one!
            try:
                where = frost_words.index(word) #in corpus of frost words,
tell me where this word is (7th, 7000th word?) use index fcn for this.
index returns position at first occurrence of the specified value.
                next_word = frost_words[where + 1]
                print("Found {} {}".format(word, next_word))
            except IndexError:
                print("No next word!")

frost_nextword(our_seed)
```

Making pairs

First, let's see how it works.

```
frost_dict = {}

for word_number, word in enumerate(frost_words):     # produces tuple (49,
"it")
    word = word.lower()
    frost_dict.update({word_number: word})           # produces dictionary
{49: "it"}

# print(frost_dict)

output = ""     # output a csv, "key, value1, value2, value3..."
word_friends = []

print(our_seed, end="")

for key, value in frost_dict.items():
    if value == our_seed:
        output = output + ", " + str(frost_dict[key + 1])

        # let's make a list of all words that follow our seed
        word_friends.append(frost_dict[key + 1])

print(output)

print("\nWord friends: {}".format(word_friends))
```

That seemed to work alright.

Let's get those word "friends" into some order. (We can rank them by popularity and get rid of duplicates.)

Since we want to do this over and over, let's put the code above into a function.

```
def get_friends(this_word):
    result = []
    # print("Looking for: {}".format(this_word))

    for key, value in frost_dict.items():
        if value == this_word:
            # print(frost_dict[key + 1])
            result.append(frost_dict[key + 1])

    # now let's alphebetize the list
    sorted_result = sorted(result)
    return sorted_result

print(get_friends(our_seed))
```

Markov Poem

That works ok.

Now, we could add weights that depend on how many times a word shows up in the results list. But if a word shows up three times, and we access the list randomly, then that word is three times more likely to be returned. So the weight is built into the list!

Now let's make a Markov poem.

```
# our_seed is the first word, let's get the next word
# and let's do that for 50 words

poem_list = []

# start with the first word, our_seed
poem_list.append(our_seed)

for j in range(0, 50):
    possible_words = get_friends(poem_list[j])  # go through each word and get its friends
    nextword = random.choice(possible_words)    # choose one of the friends at random

    poem_list.append(nextword)      # append that friend to the poem, and repeat

print(poem_list)

# make some space
print("\n\n")

# now let's make it look like a poem with some line-breaks:

for x in range(0, len(poem_list) - 1):
    # make line breaks at random points of 3 words, 5 words, 7 words

    linebreak = random.choice([3, 5, 7])
    if x % linebreak == 0:
        print(poem_list[x], end="\n")
    else:
        print(poem_list[x], end=" ")
```

How to Improve it?

If a line makes sense, it's only by accident. So how do we ensure that a sentence generated by a Markov chain is grammatical?

```
# any ideas?
```

Here's a function from the wilds of the internet that makes pairs for a Markov text generator:

```
def make_pairs(frost_words):
    for i in range(len(frost_words) - 1):
        yield( frost_words[i], frost_words[i+1])

pairs = make_pairs(frost_words)

pairs_dict = {}

for word_1, word_2 in pairs:
    # the original internet code doesn't work (!), but here's what it looks like
    print("{} {}".format(word_1, word_2))
```

Search for related words with Wordnet

"""

Searches for every NOUN related to input, using the limited lists in Wordnet
TODO: Choice to see all should return all items in all synsets
" " "

```
from nltk.corpus import wordnet as wn

types = []

search_word = input('Looking for what? ')

seek = wn.synsets(search_word)
print(seek)

decision = input("See all synsets [A] or just one [input synset as NAME.n.0x]? ")
decision.rstrip()
if decision == 'A':
    for synset in wn.synsets(search_word):
        print(synset.lemma_names())
        types.append(synset.hyponyms())
    print(types)
else:
    world = wn.synset(decision)
    bits = world.hyponyms()  # gets every kind of whatever
    types = sorted(set([lemma.name() for synset in bits for lemma in synset.lemmas()]))
    for thing in types:
        if "_" in thing:
            thing = thing.replace("_", " ")
        print(thing)
```

And the one to get hyponyms:

"""

Searches for every NOUN related to tree, using the limited lists in Wordnet
"""

```
from nltk.corpus import wordnet

# insert: flora, flower, tree
natureworld = wordnet.synset('tree.n.01')   # value comes from wordnet.lemmas('tree')
types_nature = natureworld.hyponyms()    # gets every kind of tree

# now get every lemma in the list types_nature and sort them
types = sorted(set([lemma.name() for synset in types_nature for lemma in synset.lemmas()]))

# clean up the output, which has some underlines in it
for tree in types:
    if "_" in tree:
        tree = tree.replace("_", " ")
    print(tree)
```

Projects

Assignment 3: Import NLTK, Download book, Tokenize

```
import nltk
nltk.download("book")
from nltk.book import *
text6.concordance("me") #Monty Python!

from nltk.tokenize import word_tokenize
sentence = "What are you going to do , bleed on me ?"
```

```
tokens = nltk.word_tokenize(sentence)
print("\n")
print(tokens)
```

Assignment 5: Analyse the language for keywords, sentence length, and/or parts of speech.

```
# Separate 2-page analysis in pdf format.
```

```
# Poem------------------------------------------------------------------------
poem = """
I met a traveller from an antique land,
Who said—"Two vast and trunkless legs of stone
Stand in the desert. . . . Near them, on the sand,
Half sunk a shattered visage lies, whose frown,
And wrinkled lip, and sneer of cold command,
Tell that its sculptor well those passions read
Which yet survive, stamped on these lifeless things,
The hand that mocked them, and the heart that fed;
And on the pedestal, these words appear:
My name is Ozymandias, King of Kings;
Look on my Works, ye Mighty, and despair!
Nothing beside remains. Round the decay
Of that colossal Wreck, boundless and bare
The lone and level sands stretch far away."
"""

# Import statements-----------------------------------------------------------------------
from textblob import TextBlob
import nltk
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')


# textblob objects--------------------------------------------------------------
ozymandias1 = TextBlob(
   """
   I met a traveller from an antique land,
   Who said—"Two vast and trunkless legs of stone
   Stand in the desert. . . .
   """)
ozymandias2 = TextBlob(
   """
   Near them, on the sand,
   Half sunk a shattered visage lies, whose frown,
   And wrinkled lip, and sneer of cold command,
   Tell that its sculptor well those passions read
   Which yet survive, stamped on these lifeless things,
   The hand that mocked them, and the heart that fed;
   And on the pedestal, these words appear:
   My name is Ozymandias, King of Kings;
   Look on my Works, ye Mighty, and despair!
   """)

ozymandias3 = TextBlob("Nothing beside remains.")

ozymandias4 = TextBlob(
   """
   Round the decay
   Of that colossal Wreck, boundless and bare
   The lone and level sands stretch far away."
   """)

ozymandias5 = TextBlob( """
I met a traveller from an antique land
Who said Two vast and trunkless legs of stone
Stand in the desert Near them on the sand
Half sunk a shattered visage lies whose frown
And wrinkled lip and sneer of cold command
Tell that its sculptor well those passions read
Which yet survive stamped on these lifeless things
The hand that mocked them and the heart that fed
And on the pedestal these words appear
My name is Ozymandias King of Kings
Look on my Works ye Mighty and despair
Nothing beside remains Round the decay
Of that colossal Wreck boundless and bare
The lone and level sands stretch far away
""")

# Finds sentence lengths----------------------------------------------------------------------

# print(ozymandias.tokens)  # print all the items in the sentence, including punctuation
# print(ozymandias.words)  # print only the words in a sentence
```

```python
# print(ozymandias.sentences)

num_words = len(ozymandias1.words)
print("First sentence has", num_words, "words.")
num_words = len(ozymandias2.words)
print("Second sentence has", num_words, "words.")
num_words = len(ozymandias3.words)
print("Third sentence has", num_words, "words.")
num_words = len(ozymandias4.words)
print("Fourth sentence has", num_words, "words.")

# parts of speech---------------------

# Parts of Speech TextBlob will parse a sentence for you, that is it will get all the parts of speech
# But you have to loop through each word. To find out the parts of speech, you have to look up the
# codes: https: // www.geeksforgeeks.org / python - part - of - speech - tagging - using - textblob /
# JJ is adjective
# NN is noun
# VB is verb

codes = {"CC": "coordinating conjunction",
                  "CD": "cardinal digit",
                  "DT": "determiner",
                  "EX": "existential there",
                  "FW": "foreign word",
                  "IN": "preposition/subordinating conjunction",
                  "JJ": "adjective (large)",
                  "JJR": "adjective, comparative (larger)",
                  "JJS": "adjective, superlative (largest)",
                  "LS": "list market",
                  "MD": "modal (could, will)",
                  "NN": "noun, singular (cat, tree)",
                  "NNS": "noun plural (desks)",
                  "NNP": "proper noun, singular (sarah)",
                  "NNPS": "proper noun, plural (indians or americans)",
                  "PDT": "predeterminer (all, both, half)",
                  "POS": "possessive ending (parent\ 's)",
                  "PRP": "personal pronoun (hers, herself, him,himself)",
                  "PRP$": "possessive pronoun (her, his, mine, my, our )",
                  "RB": "adverb (occasionally, swiftly)",
                  "RBR": "adverb, comparative (greater)",
                  "RBS": "adverb, superlative (biggest)",
                  "RP": "particle (about)",
                  "TO": "infinite marker (to)",
                  "UH": "interjection (goodbye)",
                  "VB": "verb (ask)",
                  "VBG": "verb gerund (judging)",
                  "VBD": "verb past tense (pleaded)",
                  "VBN": "verb past participle (reunified)",
                  "VBP": "verb, present tense not 3rd person singular(wrap)",
                  "VBZ": "verb, present tense with 3rd person singular (bases)",
                  "WDT": "wh-determiner (that, what)",
                  "WP": "wh- pronoun (who)",
                  "WP$": "possessive wh-pronoun (whose)",
                  "WRB": "wh- adverb (how)"}

def adjectives(text):
   count = 0
   for (word, tag) in text.tags:
       if tag == 'JJ' or tag == 'JJS' or tag == 'JJR':
           count = count + 1
   return count

print('Number of adjectives: ', adjectives(ozymandias5))

def nouns(text):
   count = 0
   for (word, tag) in text.tags:
       if tag == 'NN' or tag == 'NNS' or tag == 'NNP' or tag == 'NNPS':
           count = count + 1
   return count

print('Number of nouns: ', nouns(ozymandias5))

def verbs(text):
   count = 0
   for (word, tag) in text.tags:
       if tag == 'VB' or tag == 'VBG' or tag == 'VBD' or tag == 'VBN' or tag == 'VBP' or tag == 'VBZ':
           count = count + 1
   return count

print('Number of verbs: ', verbs(ozymandias5))

print(ozymandias5.sentiment)

def speech_part(sentence):
   results = []        # we can use this to hold our results
   for (word, tag) in sentence.tags:
       print("{}: {}".format(word, codes[tag]))

speech_part(ozymandias5)
```

Final Project: Is a picture worth a thousand words?

```
#
BOOK-----------------------------------------------------------------------------
-

# import statements
from textblob import TextBlob
import nltk
nltk.download('punkt')
import re

# text of book from gutenberg
full_book = """
It is a truth universally acknowledged,
...
"""

# use textblob to count total words and sentences
book = TextBlob(full_book)
# create list of words
list_words = book.words
# create list of sentences
list_sentences = book.sentences
# get number of words or sentences and print
print ("total num words: ", len(list_words))
print ("total num sentences: ", len(list_sentences))


# Use regex to isolate dialogue
# set punctuation of interest to "
pattern = r'\"'
# split all the text every place a " is found, resulting in a list containing both quotes and everything in-between
split_by_punct = re.split(pattern, full_book)
# remove non-dialogue by removing very other item in the list (since it's split by ", the list is always [outside of quotes], [inside of quotes],
[outside of quotes]
remove_nondialogue = split_by_punct[1::2]

# count words in list item containing a dialogue sentence, and sum all the sentence lengths together to get overall number of dialogue words
count_words_in_sentence = [len(x.split()) for x in remove_nondialogue]
print("num words dialogue:", sum(count_words_in_sentence))

# count number of each type of sentence
count = 0
question_count = 0
exclamation_count = 0
statement_count = 0
# loop through every list item containing a dialogue sentence and look for punctuation
for sentence in remove_nondialogue:
   if "." in sentence:
       statement_count = statement_count + 1
   if "?" in sentence:
       question_count = question_count + 1
   if "!" in sentence:
       exclamation_count = exclamation_count + 1
   count = statement_count + question_count + exclamation_count
print ("num sentences:", count)
print ("num statements:", statement_count)
print ("num questions:", question_count)
print ("num exclamations:", exclamation_count)


#captions—-----------------------------------------------------------
# full captions copied from srt file
captions = """
1
00:00:39,123 --> 00:00:42,832
(BIRDS CHIRPING)
```

```
2
00:01:36,768 --> 00:01:37,518
(CATTLE LOWING)
...
"""

# string --> list of words
words = captions.split()

# Gets rid of time stamps by looping through every word
def remove_numbers(text):
   i = 0
   # if it contains"00:" or "01:", delete all items that correspond to timestamp heading
   while (i < len(text)):
       if "00:" in text[i]:
           del text[i]
           del text[i]
           del text[i]
           del text[i-1]
       i=i+1
   i = 0
   while (i < len(text)):
       if "01:" in text[i]:
           del text[i]
           del text[i]
           del text[i]
           del text[i-1]
       i=i+1
   return text

words = remove_numbers(words)



#Count words (including speakers and enviro clues)
count = 0
for word in words:
   count = count + 1
print ("num words (including speakers and enviro clues):", count)



# count statements, questions, exclamations, overall sentence count
count = 0
question_count = 0
exclamation_count = 0
statement_count = 0
for word in words:
   if "." in word:
       statement_count = statement_count + 1
   if "?" in word:
       question_count = question_count + 1
   if "!" in word:
       exclamation_count = exclamation_count + 1
   # if "." in word or "?" in word or "!" in word:
   count = statement_count + question_count + exclamation_count
print ("num sentences:", count)
print ("num statements:", statement_count)
print ("num questions:", question_count)
print ("num exclamations:", exclamation_count)
```