Language Processing and Python

CIS/STA 9665

Reminders:
Lab Report 1 Due Sep 8
Project Team Formation Due tonight (Sep 7)

Objective

- What can we achieve by combining simple programming techniques with large quantities of text?
- How can we automatically extract key words and phrases that sum up the style and content of a text?
- What tools and techniques does the Python programming language provide for such work?

Outline

- **1. Computing with Language: Texts and Words**
- 2. A Closer Look at Python: Texts as Lists of Words
- 3. Computing with Language: Simple Statistics
- 4. Back to Python: Making Decisions and Taking Control

1 Computing with Language: Texts and Words

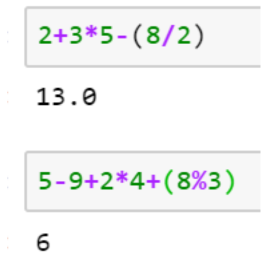
- We're all very familiar with text, since we read and write it every day. Here we will treat text as raw data for the programs we write, programs that manipulate and analyze it in a variety of interesting ways. But before we can do this, we have to get started with the Python interpreter.

Jupyter Shortcut

- A to insert a new cell above the current cell
- B to insert a new cell below.
- D + D (press the key twice) to delete the current cell
- Shift+Enter: run the cell
- Enter: continue to write in the current cell

1.1 Getting Started with Python

- Exercise 1
  - Enter two more expressions of your own. You can use asterisk (*) for multiplication and slash (/) for division, and parentheses for bracketing expressions.



```
2+3*5-(8/2)
13.0
5-9+2*4+(8%3)
6
```

1.2 Getting Started with NLTK

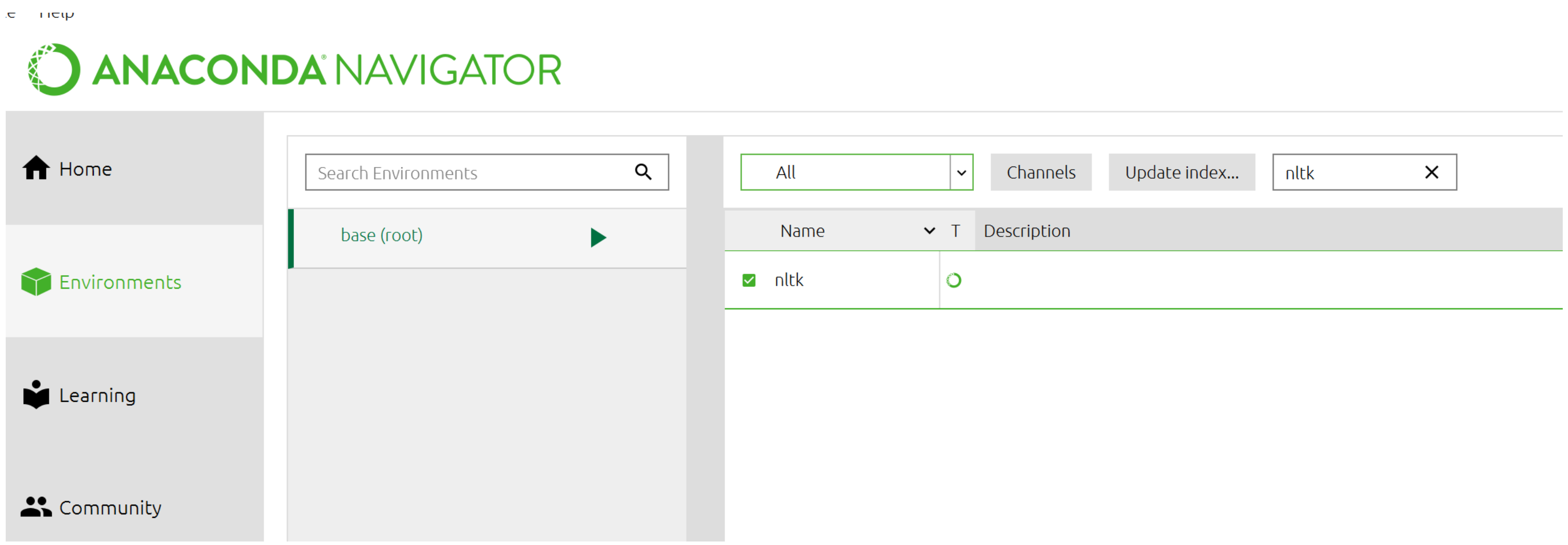
- What is NLTK?
- Install NLTK 3.0

What is NLTK?

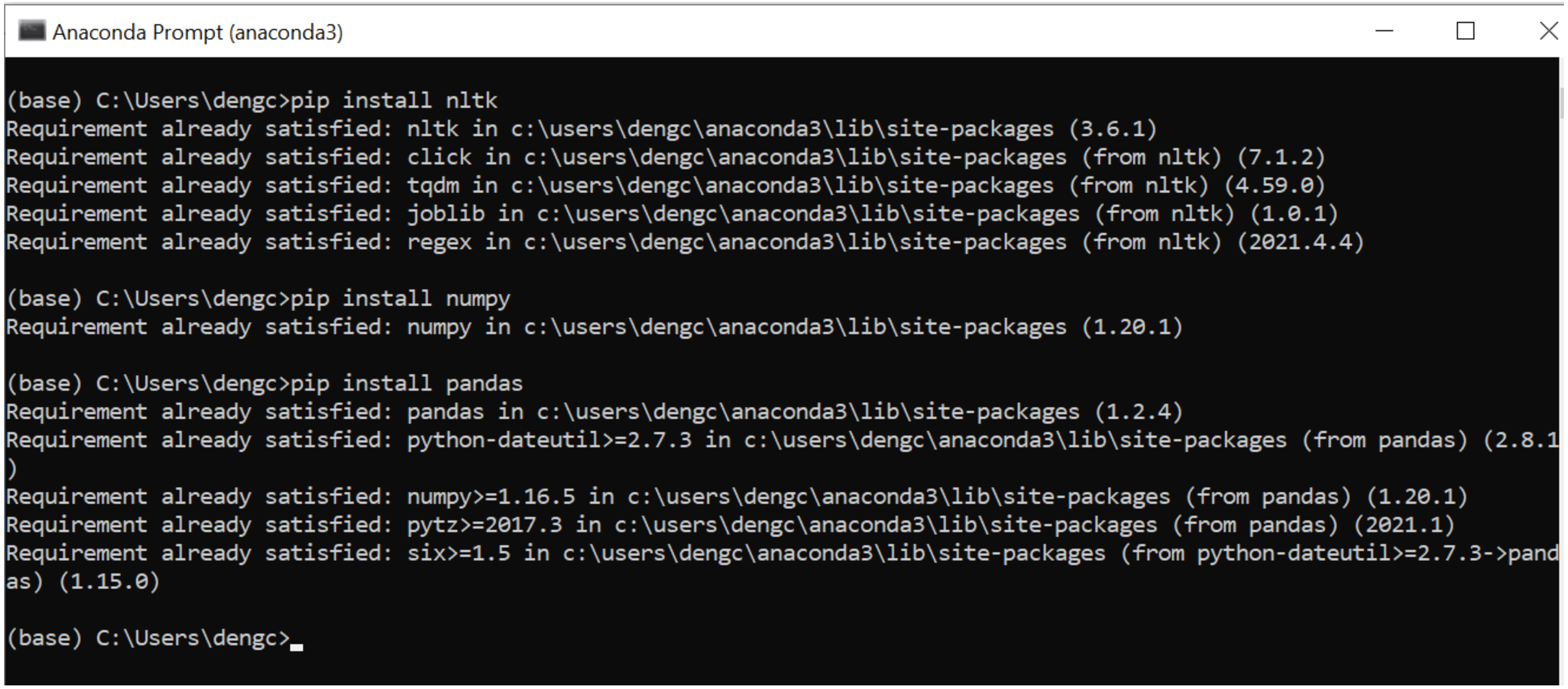
- The Natural Language Toolkit, or more commonly NLTK, is a suite of **libraries and programs** for **symbolic and statistical natural language processing** (NLP) for English written in the Python programming language.
- It was developed by Steven Bird and Edward Loper in the Department of Computer and Information Science at the University of Pennsylvania.

-from wikipedia

Download NLTK-from Anaconda (recommended)



Anaconda Prompt

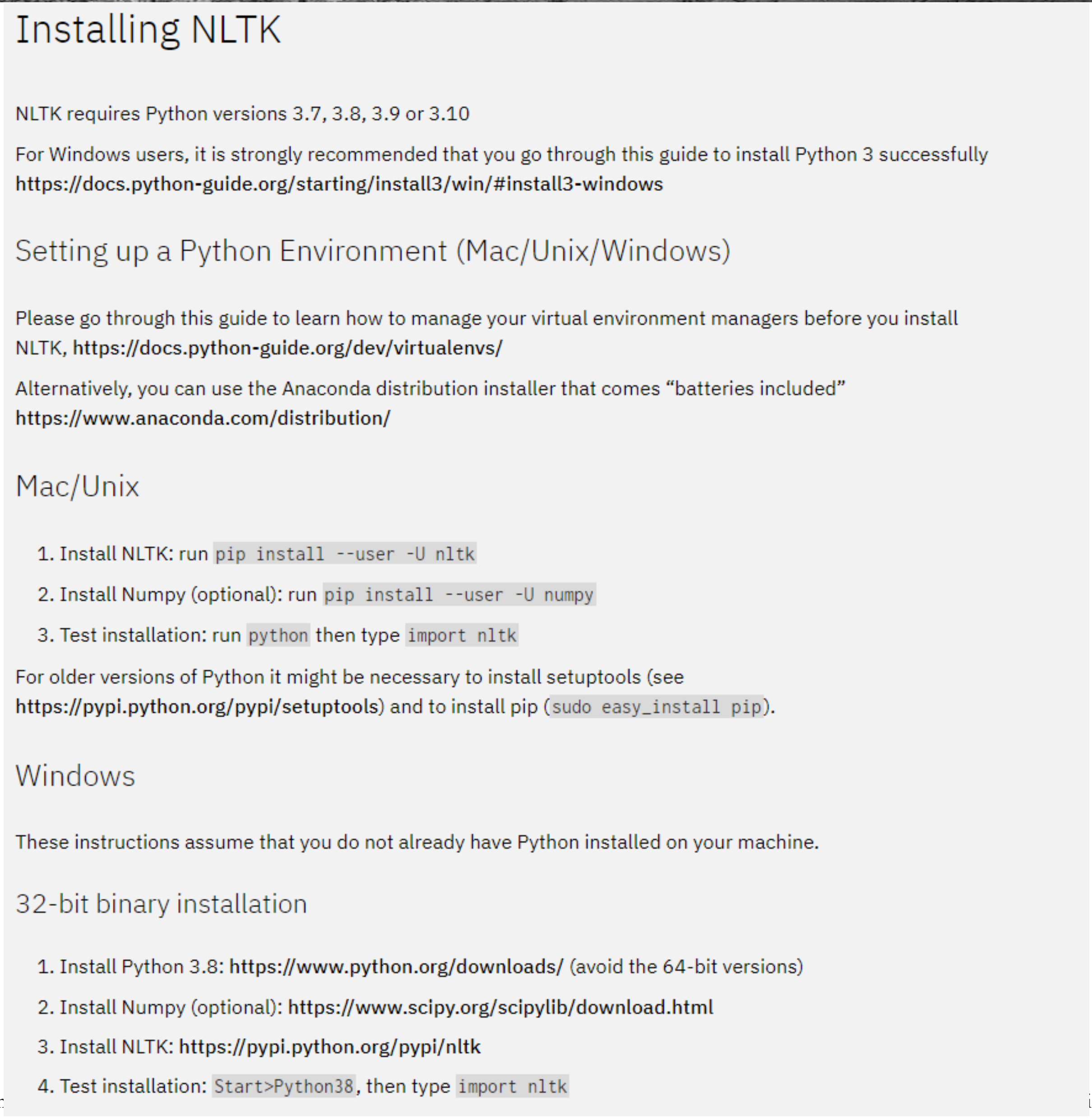


```
(base) C:\Users\dengc>pip install nltk
Requirement already satisfied: nltk in c:\users\dengc\anaconda3\lib\site-packages (3.6.1)
Requirement already satisfied: click in c:\users\dengc\anaconda3\lib\site-packages (from nltk) (7.1.2)
Requirement already satisfied: tqdm in c:\users\dengc\anaconda3\lib\site-packages (from nltk) (4.59.0)
Requirement already satisfied: joblib in c:\users\dengc\anaconda3\lib\site-packages (from nltk) (1.0.1)
Requirement already satisfied: regex in c:\users\dengc\anaconda3\lib\site-packages (from nltk) (2021.4.4)

(base) C:\Users\dengc>pip install numpy
Requirement already satisfied: numpy in c:\users\dengc\anaconda3\lib\site-packages (1.20.1)

(base) C:\Users\dengc>pip install pandas
Requirement already satisfied: pandas in c:\users\dengc\anaconda3\lib\site-packages (1.2.4)
Requirement already satisfied: python-dateutil>=2.7.3 in c:\users\dengc\anaconda3\lib\site-packages (from pandas) (2.8.1
)
Requirement already satisfied: numpy>=1.16.5 in c:\users\dengc\anaconda3\lib\site-packages (from pandas) (1.20.1)
Requirement already satisfied: pytz>=2017.3 in c:\users\dengc\anaconda3\lib\site-packages (from pandas) (2021.1)
Requirement already satisfied: six>=1.5 in c:\users\dengc\anaconda3\lib\site-packages (from python-dateutil>=2.7.3->pand
as) (1.15.0)

(base) C:\Users\dengc>_
```

If you do not install Anaconda, you should install NLTK separately

Please follow the link:

http://www.nltk.org/install.html



Installing NLTK

NLTK requires Python versions 3.7, 3.8, 3.9 or 3.10

For Windows users, it is strongly recommended that you go through this guide to install Python 3 successfully **https://docs.python-guide.org/starting/install3/win/#install3-windows**

Setting up a Python Environment (Mac/Unix/Windows)

Please go through this guide to learn how to manage your virtual environment managers before you install NLTK, **https://docs.python-guide.org/dev/virtualenvs/**

Alternatively, you can use the Anaconda distribution installer that comes "batteries included" **https://www.anaconda.com/distribution/**

Mac/Unix

1. Install NLTK: run `pip install --user -U nltk`
2. Install Numpy (optional): run `pip install --user -U numpy`
3. Test installation: run `python` then type `import nltk`

For older versions of Python it might be necessary to install setuptools (see **https://pypi.python.org/pypi/setuptools**) and to install pip (`sudo easy_install pip`).

Windows

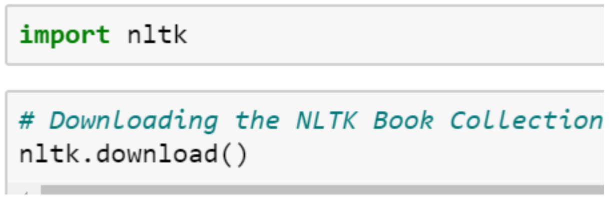
These instructions assume that you do not already have Python installed on your machine.

32-bit binary installation

1. Install Python 3.8: **https://www.python.org/downloads/** (avoid the 64-bit versions)
2. Install Numpy (optional): **https://www.scipy.org/scipylib/download.html**
3. Install NLTK: **https://pypi.python.org/pypi/nltk**
4. Test installation: `Start>Python38`, then type `import nltk`

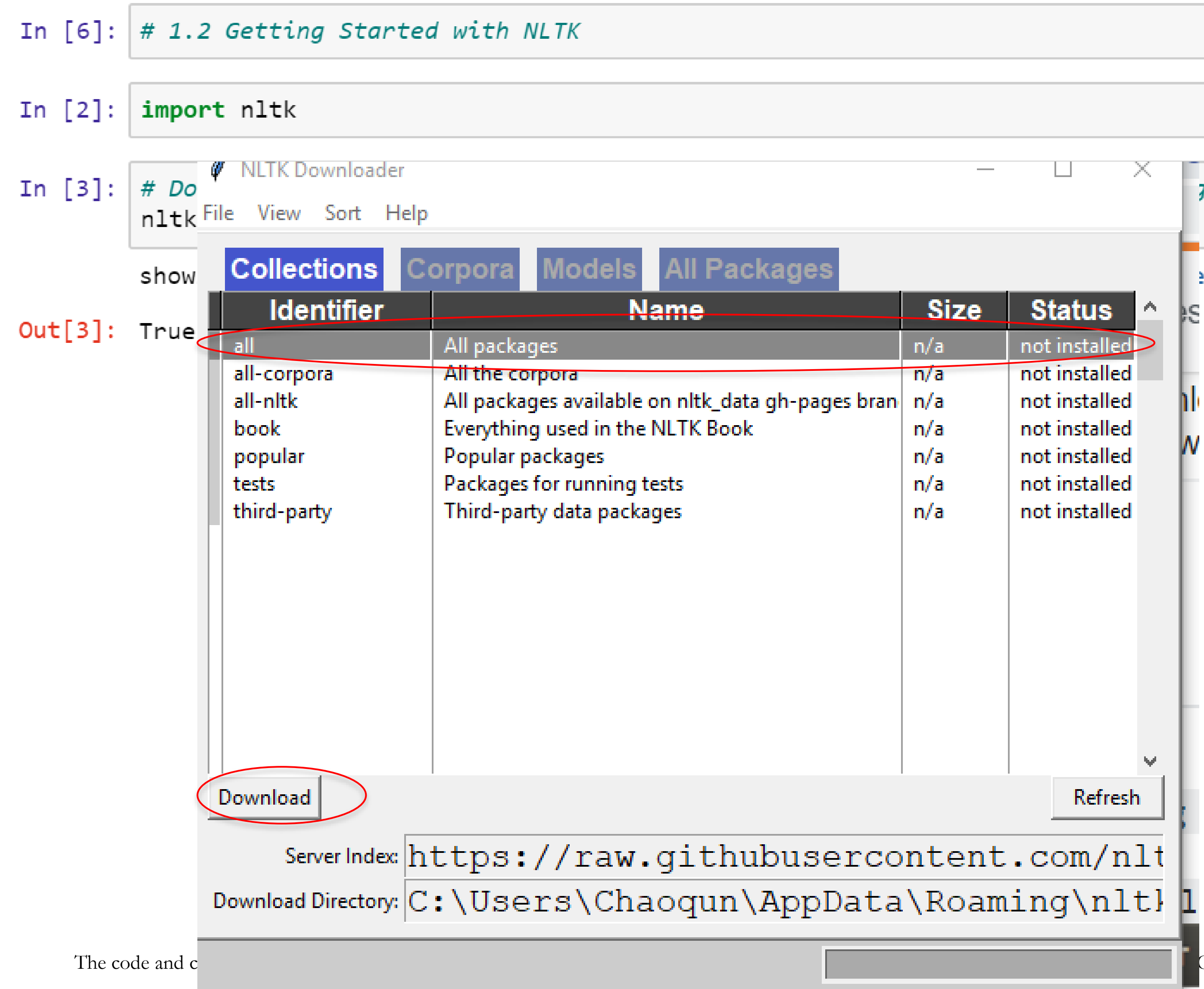
Import NLTK

- Install NLTK (Poll)
- Import NLTK
- Download all the NLTK Book Collection



```
import nltk
```

```
# Downloading the NLTK Book Collection
nltk.download()
```



```
In [6]: # 1.2 Getting Started with NLTK

In [2]: import nltk

In [3]: # Do                                            nltk.download()
        nltk
        show                                            ex.xml
Out[3]: True
```

NLTK Downloader

File View Sort Help

Collections | Corpora | Models | All Packages

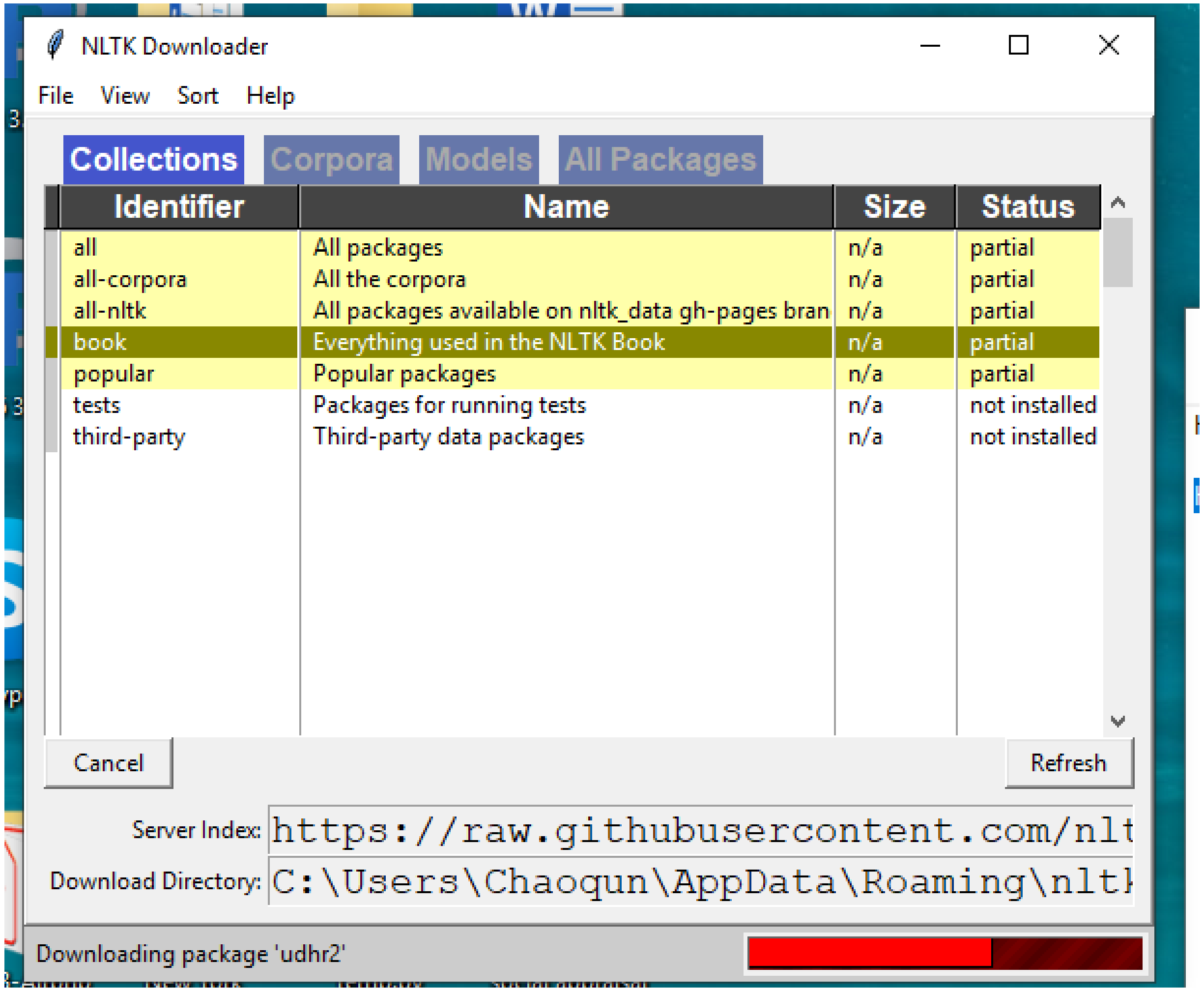
Identifier	Name	Size	Status
all	All packages	n/a	not installed
all-corpora	All the corpora	n/a	not installed
all-nltk	All packages available on nltk_data gh-pages bran	n/a	not installed
book	Everything used in the NLTK Book	n/a	not installed
popular	Popular packages	n/a	not installed
tests	Packages for running tests	n/a	not installed
third-party	Third-party data packages	n/a	not installed

Download   Refresh

Server Index: https://raw.githubusercontent.com/nlt

Download Directory: C:\Users\Chaoqun\AppData\Roaming\nltk

Choose "all packages";
-click "download".
After finishing, we closed this window

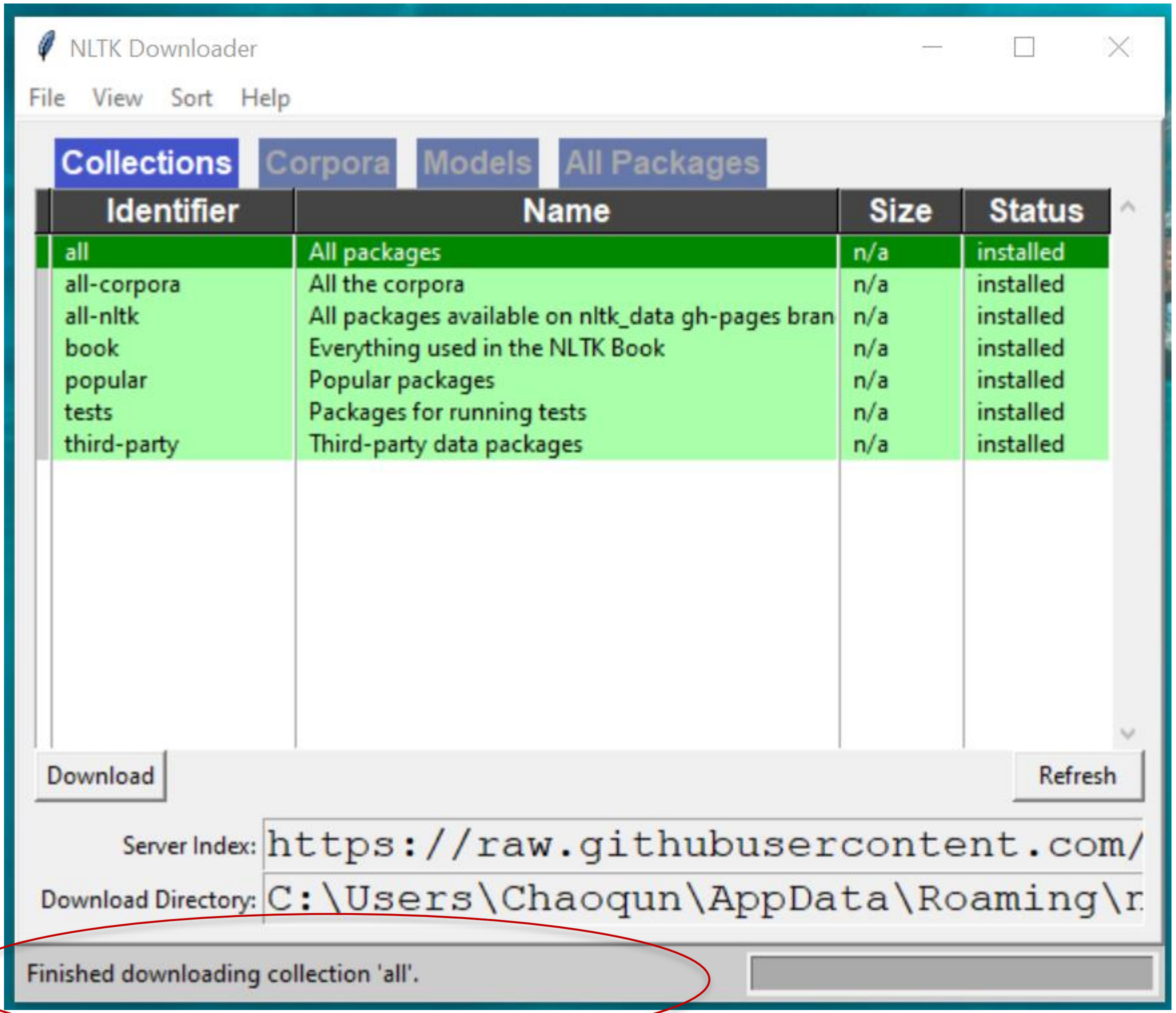


Identifier	Name	Size	Status
all	All packages	n/a	partial
all-corpora	All the corpora	n/a	partial
all-nltk	All packages available on nltk_data gh-pages bran	n/a	partial
book	Everything used in the NLTK Book	n/a	partial
popular	Popular packages	n/a	partial
tests	Packages for running tests	n/a	not installed
third-party	Third-party data packages	n/a	not installed

Server Index: https://raw.githubusercontent.com/nlt

Download Directory: C:\Users\Chaoqun\AppData\Roaming\nltk

Downloading package 'udhr2'



Identifier	Name	Size	Status
all	All packages	n/a	installed
all-corpora	All the corpora	n/a	installed
all-nltk	All packages available on nltk_data gh-pages bran	n/a	installed
book	Everything used in the NLTK Book	n/a	installed
popular	Popular packages	n/a	installed
tests	Packages for running tests	n/a	installed
third-party	Third-party data packages	n/a	installed

Server Index: https://raw.githubusercontent.com/

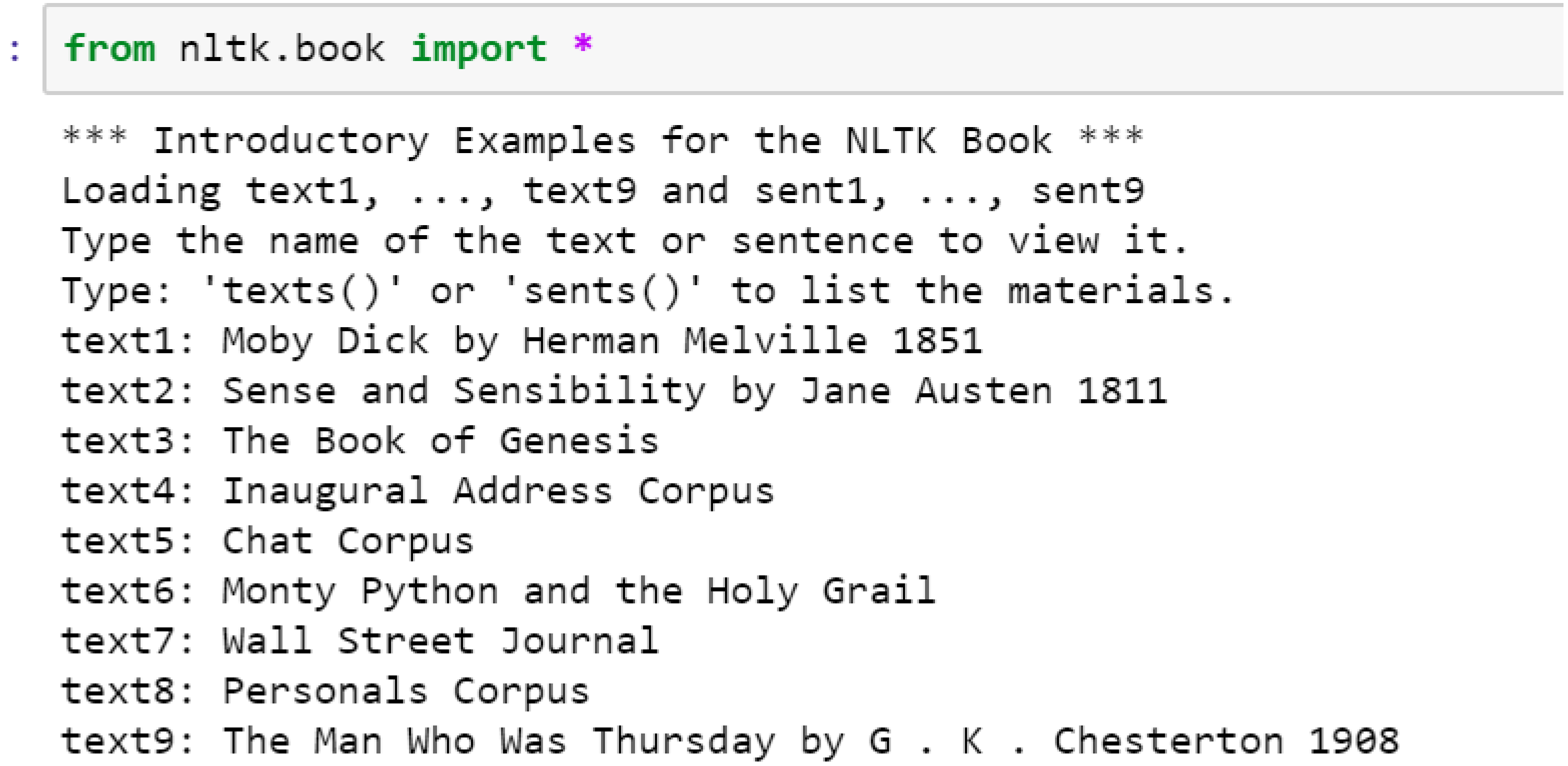
Download Directory: C:\Users\Chaoqun\AppData\Roaming\r

Finished downloading collection 'all'.

Close this Window

It shows "True"

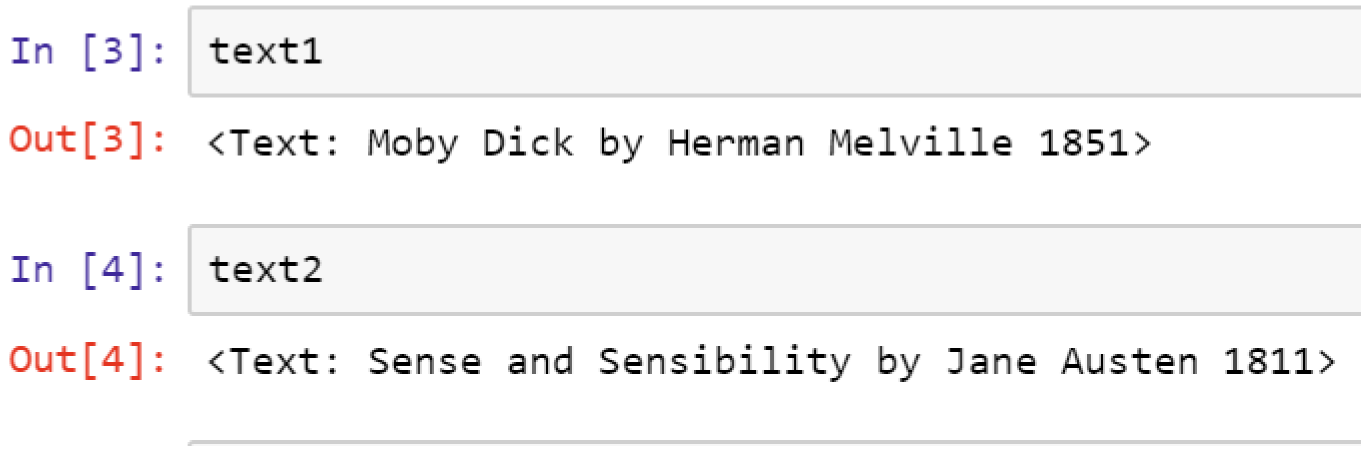
Once the data is downloaded to your machine, you can load some of it using the Python interpreter. The first step is to type a special command at the Python prompt which tells the interpreter to load some texts for us to explore: from nltk.book import *. This says "from NLTK's book module, load all items." The book module contains all the data you will need as you read this chapter.



```
: from nltk.book import *
```

```
*** Introductory Examples for the NLTK Book ***
Loading text1, ..., text9 and sent1, ..., sent9
Type the name of the text or sentence to view it.
Type: 'texts()' or 'sents()' to list the materials.
text1: Moby Dick by Herman Melville 1851
text2: Sense and Sensibility by Jane Austen 1811
text3: The Book of Genesis
text4: Inaugural Address Corpus
text5: Chat Corpus
text6: Monty Python and the Holy Grail
text7: Wall Street Journal
text8: Personals Corpus
text9: The Man Who Was Thursday by G . K . Chesterton 1908
```

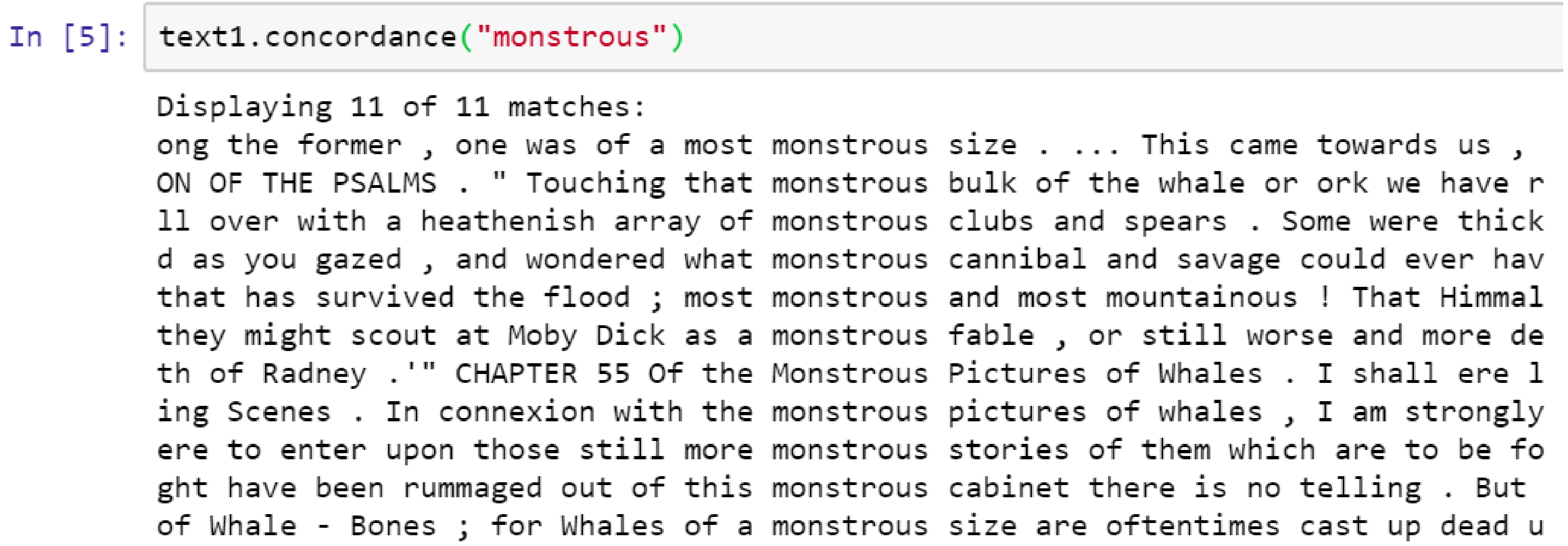
Any time we want to find out about these texts, we just have to enter their names at the Python prompt:



```
In [3]: text1
Out[3]: <Text: Moby Dick by Herman Melville 1851>

In [4]: text2
Out[4]: <Text: Sense and Sensibility by Jane Austen 1811>
```

1.3 Searching Text

There are many ways to examine the context of a text apart from simply reading it. A concordance view shows us every occurrence of a given word, together with some context. Here we look up the word monstrous in Moby Dick by entering text1 followed by a period, then the term concordance, and then placing "monstrous" in parentheses:



```
In [5]: text1.concordance("monstrous")

Displaying 11 of 11 matches:
ong the former , one was of a most monstrous size . ... This came towards us ,
ON OF THE PSALMS . " Touching that monstrous bulk of the whale or ork we have r
ll over with a heathenish array of monstrous clubs and spears . Some were thick
d as you gazed , and wondered what monstrous cannibal and savage could ever hav
that has survived the flood ; most monstrous and most mountainous ! That Himmal
they might scout at Moby Dick as a monstrous fable , or still worse and more de
th of Radney .'" CHAPTER 55 Of the Monstrous Pictures of Whales . I shall ere l
ing Scenes . In connexion with the monstrous pictures of whales , I am strongly
ere to enter upon those still more monstrous stories of them which are to be fo
ght have been rummaged out of this monstrous cabinet there is no telling . But
of Whale - Bones ; for Whales of a monstrous size are oftentimes cast up dead u
```

Exercise 2 -Your Turn

- 1) Search Sense and Sensibility for the word affection
  - using text2.concordance("affection")
- 2) Search the book of Genesis to find out how long some people lived
  - using text3.concordance("lived")

```
In [7]: text3.concordance("lived")
```

```
Displaying 25 of 38 matches:
ay when they were created . And Adam lived an hundred and thirty years , and be
ughters : And all the days that Adam lived were nine hundred and thirty yea and
nd thirty yea and he died . And Seth lived an hundred and five years , and bega
ve years , and begat Enos : And Seth lived after he begat Enos eight hundred an
welve years : and he died . And Enos lived ninety years , and begat Cainan : An
 years , and begat Cainan : And Enos lived after he begat Cainan eight hundred 
ive years : and he died . And Cainan lived seventy years and begat Mahalaleel :
rs and begat Mahalaleel : And Cainan lived after he begat Mahalaleel eight hund
years : and he died . And Mahalaleel lived sixty and five years , and begat Jar
s , and begat Jared : And Mahalaleel lived after he begat Jared eight hundred a
and five yea and he died . And Jared lived an hundred sixty and two years , and
o years , and he begat Eno And Jared lived after he begat Enoch eight hundred y
 and two yea and he died . And Enoch lived sixty and five years , and begat Met
 ; for God took him . And Methuselah lived an hundred eighty and seven years , 
 , and begat Lamech . And Methuselah lived after he begat Lamech seven hundred 
nd nine yea and he died . And Lamech lived an hundred eighty and two years , an
ch the LORD hath cursed . And Lamech lived after he begat Noah five hundred nin
naan shall be his servant . And Noah lived after the flood three hundred and fi
xad two years after the flo And Shem lived after he begat Arphaxad five hundred
at sons and daughters . And Arphaxad lived five and thirty years , and begat Sa
ars , and begat Salah : And Arphaxad lived after he begat Salah four hundred an
begat sons and daughters . And Salah lived thirty years , and begat Eber : And
y years , and begat Eber : And Salah lived after he begat Eber four hundred and
 begat sons and daughters . And Eber lived four and thirty years , and begat Pe
```

A concordance permits us to see words in context.
For example, we saw that *monstrous* occurred in contexts such as *the* ___ *pictures* and *a* ___ *size* .
What other words appear in a similar range of contexts?
We can find out by appending the term similar to the name of the text in question, then inserting the relevant word in parentheses:

```
In [9]: text1.similar("monstrous")

        true contemptible christian abundant few part mean careful puzzled
        mystifying passing curious loving wise doleful gamesome singular
        delightfully perilous fearless
```

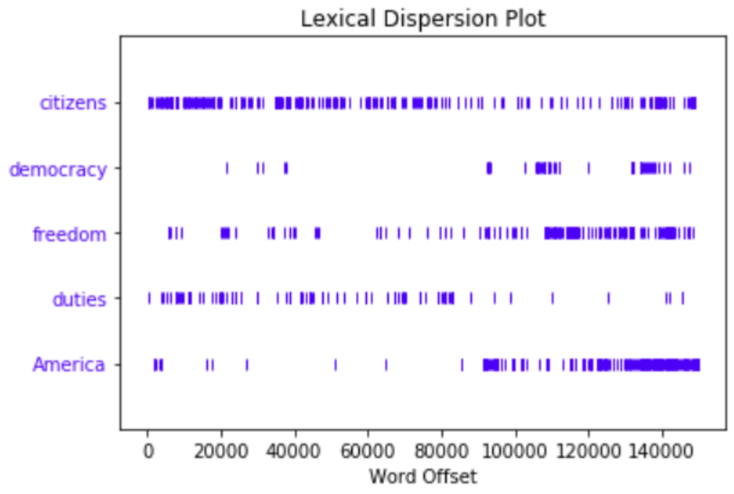
```
In [10]: text2.similar("monstrous")

        very so exceedingly heartily a as good great extremely remarkably
        sweet vast amazingly
```

Observe that we get different results for different texts. Austen (text 2' author) uses this word quite differently from Melville (text1's author); for her, monstrous has positive connotations, and sometimes functions as an intensifier like the word very.

Dispersion Plot

- It is one thing to automatically detect that a particular word occurs in a text, and to display some words that appear in the same context. However, we can also determine the location of a word in the text: how many words from the beginning it appears.
- This positional information can be displayed using a dispersion plot.
- Each stripe represents an instance of a word, and each row represents the entire text.
- In 1.2 we see some striking patterns of word usage over the last 220 years (in an artificial text constructed by joining the texts of the Inaugural Address Corpus end-to-end).
- You can produce this plot as shown below. You might like to try more words (e.g., liberty, constitution), and different texts. Can you predict the dispersion of a word before you view it? As before, take care to get the quotes, commas, brackets and parentheses exactly right.

```
In [22]: from matplotlib import pyplot as plt
         import numpy as np
         import math

In [23]: text4.dispersion_plot(["citizens", "democracy", "freedom", "duties", "America"])
```



Lexical Dispersion Plot for Words in U.S. Presidential Inaugural Addresses: This can be used to investigate changes in language use over time.

The code and content in this slide is cited from Natural Language Processing with Python, by Steven Bird, Ewan Klein, and Edward Loper. O'Reilly Media, 978-0-596-51649-9

1.4 Counting Vocabulary

- Let's begin by finding out the length of a text from start to finish, in terms of the words and punctuation symbols that appear. We use the term len to get the length of something, which we'll apply here to the book of Genesis:

```
len(text3)
```

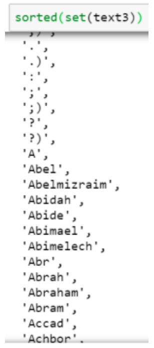
```
44764
```

set ( )

- So Genesis has 44,764 words and punctuation symbols, or "tokens."
- A token is the technical name for a sequence of characters — such as hairy, his, or :) — that we want to treat as a group.
- How many distinct words does the book of Genesis contain?
- In Python we can obtain the vocabulary items of text3 with the command: set(text3).

sorted (set( )) and len(set( ))

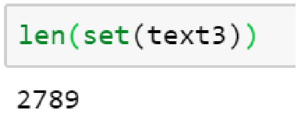
- By wrapping sorted() around the Python expression set(text3) [1], we obtain a sorted list of vocabulary items, beginning with various punctuation symbols and continuing with words starting with A. All capitalized words precede lowercase words.



```
sorted(set(text3))
```

```
 '.',
 '.)',
 ':',
 ';',
 ';)',
 '?',
 '?)',
 'A',
 'Abel',
 'Abelmizraim',
 'Abidah',
 'Abide',
 'Abimael',
 'Abimelech',
 'Abr',
 'Abrah',
 'Abraham',
 'Abram',
 'Accad',
 'Achbor',
```

- We discover the size of the vocabulary indirectly, by asking for the number of items in the set, and again we can use len to obtain this number



```
len(set(text3))
```

```
2789
```

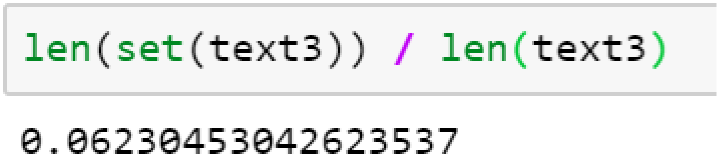
The code and content in this slide is cited from Natural Language Processing with Python, by Steven Bird, Ewan Klein, and Edward Loper. O'Reilly Media, 978-0-596-51649-9

Word Type

- Although it has 44,764 tokens, this book has only 2,789 distinct words, or "word types."
- A **word type** is the form or spelling of the word independently of its specific occurrences in a text — that is, the word considered as a unique item of vocabulary.
- Our count of **2,789 items** will include **punctuation symbols**, so we will generally call these unique items **types** instead of word types.

Lexical Richness

- Now, let's calculate a measure of the lexical richness of the text.
- The next example shows us that the number of distinct words is just 6% of the total number of words, or equivalently that each word is used 16 times on average

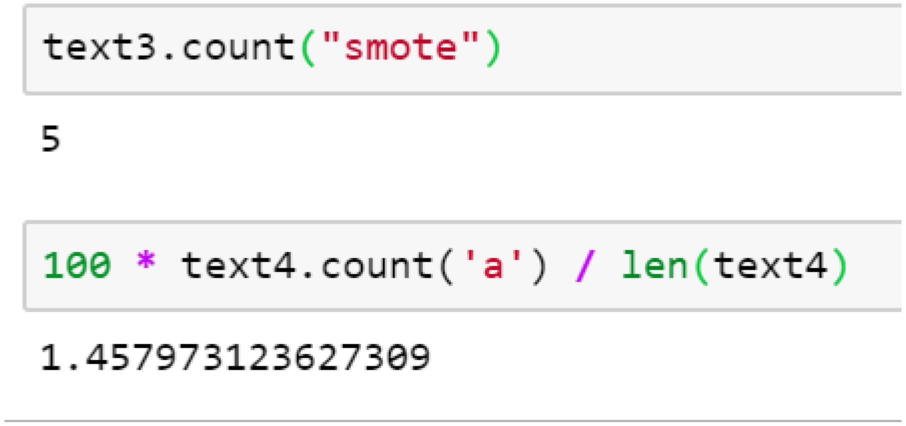


```
len(set(text3)) / len(text3)
```

```
0.06230453042623537
```

count( )

- Next, let's focus on particular words. We can count how often a word occurs in a text, and compute what percentage of the text is taken up by a specific word



```
text3.count("smote")
```

```
5
```

```
100 * text4.count('a') / len(text4)
```

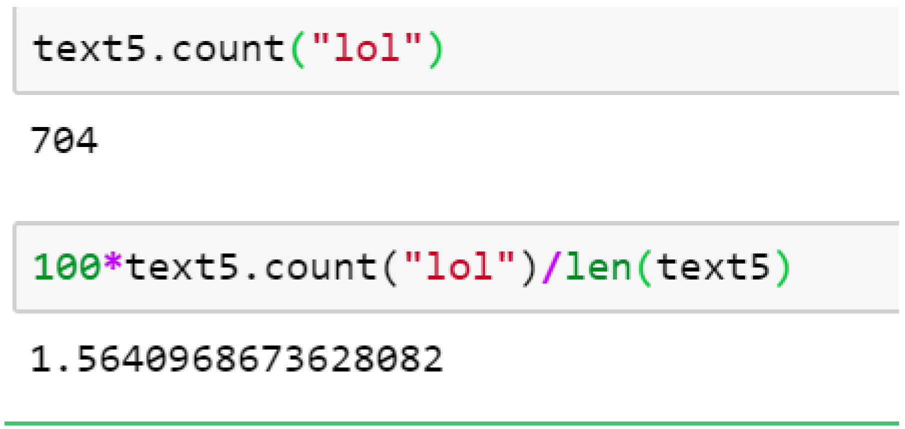
```
1.457973123627309
```

Exercise 3

- How many times does the word “lol” appear in text5? How much is this as a percentage of the total number of words in this text?

Exercise 3

- How many times does the word "lol" appear in text5? How much is this as a percentage of the total number of words in this text?



```
text5.count("lol")
```

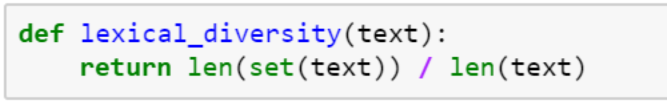
```
704
```

```
100*text5.count("lol")/len(text5)
```

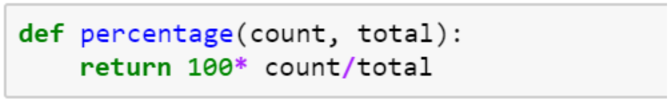
```
1.5640968673628082
```

Function

- You may want to repeat such calculations on several texts, but it is tedious to keep retyping the formula.
- Instead, you can come up with your own name for a task, like "lexical_diversity" or "percentage", and associate it with a block of code.
- The next example shows how to define two new functions, lexical_diversity( ) and percentage( ):



```python
def lexical_diversity(text):
    return len(set(text)) / len(text)
```

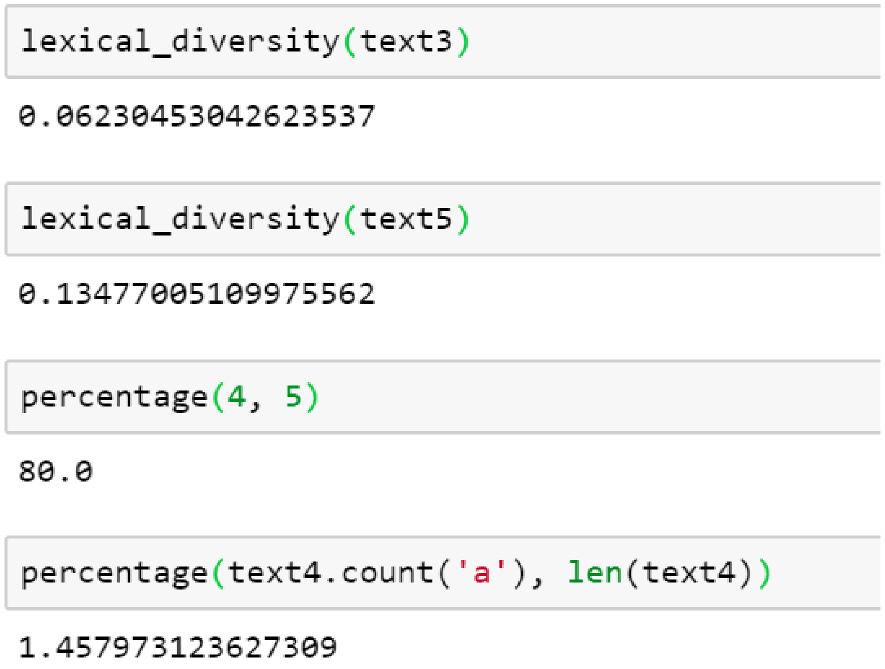


```python
def percentage(count, total):
    return 100* count/total
```

Function

- In the definition of lexical_diversity( ) , we specify a parameter named text .
- This parameter is a "placeholder" for the actual text whose lexical diversity we want to compute, and reoccurs in the block of code that will run when the function is used.
- Similarly, percentage( ) is defined to take two parameters, named count and total

Once Python knows that lexical_diversity() and percentage() are the names for specific blocks of code, we can go ahead and use these functions:



```
lexical_diversity(text3)
```

```
0.06230453042623537
```

```
lexical_diversity(text5)
```

```
0.13477005109975562
```

```
percentage(4, 5)
```

```
80.0
```

```
percentage(text4.count('a'), len(text4))
```

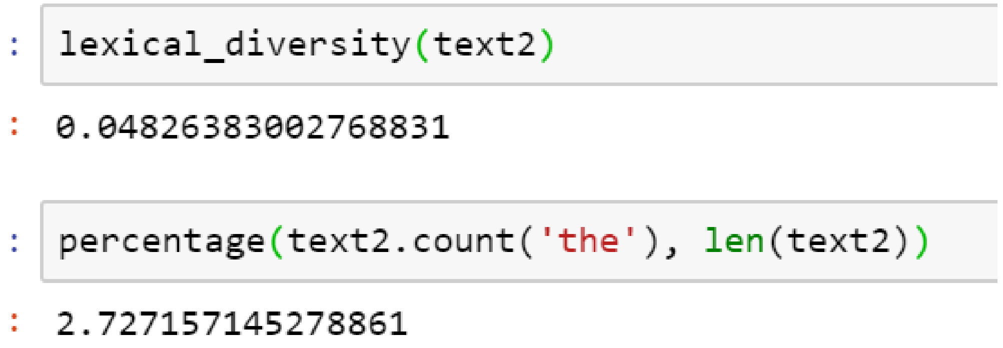
```
1.457973123627309
```

Exercise 4

- Use the function lexical_diversity () to calculate the Lexical Richness of “Sense and Sensibility” and compute what percentage of this text is taken up by a specific word “the”

Exercise 4

- Use the function lexical_diversity () to calculate the Lexical Richness of “Sense and Sensibility”



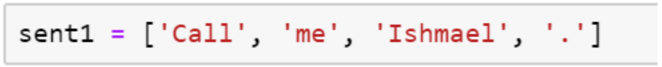
```
: lexical_diversity(text2)

: 0.04826383002768831

: percentage(text2.count('the'), len(text2))

: 2.727157145278861
```

Outline

- 1. Computing with Language: Texts and Words
- **2. A Closer Look at Python: Texts as Lists of Words**
- 3. Computing with Language: Simple Statistics
- 4. Back to Python: Making Decisions and Taking Control

2.1 Lists

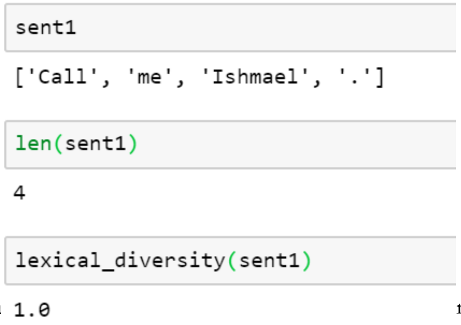
- What is a text? At one level, it is a sequence of symbols on a page such as this one.
- At another level, it is a sequence of chapters, made up of a sequence of sections, where each section is a sequence of paragraphs, and so on.
- However, for our purposes, we will think of a text as nothing more than a sequence of words and punctuation.
- Here's how we represent text in Python, in this case the opening sentence of Moby Dick:



```
sent1 = ['Call', 'me', 'Ishmael', '.']
```

2.1 Lists

- After the prompt we've given a name we made up, sent1, followed by the equals sign, and then some quoted words, separated with commas, and surrounded with brackets.
- This bracketed material is known as a **list** in Python: it is how we store a text.
- We can inspect it by typing the name. We can ask for its length
- We can even apply our own lexical_diversity() function to it



```
sent1
```

```
['Call', 'me', 'Ishmael', '.']
```

```
len(sent1)
```

```
4
```

```
lexical_diversity(sent1)
```

```
1.0
```

Exercise 5

- Make up a few sentences of your own, by typing a name, equals sign, and a list of words, like this: ex1 = [‘Monty’, ‘Python’, ‘and’, ‘the’, ‘Holy’, ‘Grail’, ‘the’]. Repeat some of the other Python operations we saw earlier in 1, e.g., sorted(ex1), len(set(ex1)), ex1.count(‘the‘), lexical_diversity(ex1).

Exercise 5

```python
ex1=['Monty', 'Python', 'and', 'the', 'Holy', 'Grail', 'the']
```

```python
sorted(ex1)
```

```
['Grail', 'Holy', 'Monty', 'Python', 'and', 'the', 'the']
```

```python
len(set(ex1))
```

```
6
```

```python
ex1.count('the')
```

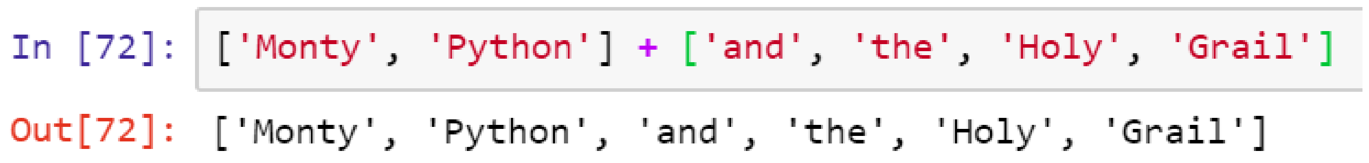
```
2
```

```python
lexical_diversity(ex1)
```

```
0.8571428571428571
```

concatenation

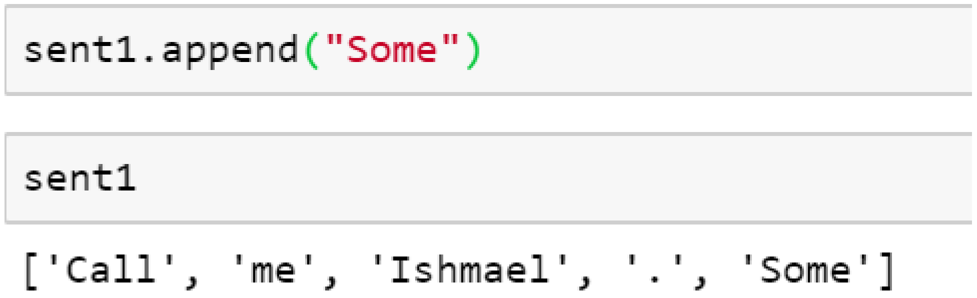
- A pleasant surprise is that we can use Python's addition operator on lists. Adding two lists [1] creates a new list with everything from the first list, followed by everything from the second list
- This special use of the addition operation is called **concatenation**; it combines the lists together into a single list. We can concatenate sentences to build up a text.



```
In [72]: ['Monty', 'Python'] + ['and', 'the', 'Holy', 'Grail']

Out[72]: ['Monty', 'Python', 'and', 'the', 'Holy', 'Grail']
```

appending

- What if we want to add a single item to a list? This is known as appending. When we append() to a list, the list itself is updated as a result of the operation.



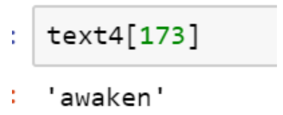
```
sent1.append("Some")
```

```
sent1
```

```
['Call', 'me', 'Ishmael', '.', 'Some']
```

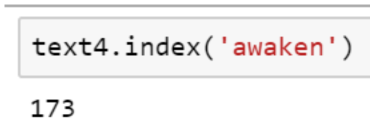
2.2 Indexing Lists

- With some patience, we can pick out the 1st, 173rd, or even 14,278th word in a printed text. Analogously, we can identify the elements of a Python list by their order of occurrence in the list. The number that represents this position is the item's index. We instruct Python to show us the item that occurs at an index such as 173 in a text by writing the name of the text followed by the index inside square brackets:



```
text4[173]
'awaken'
```

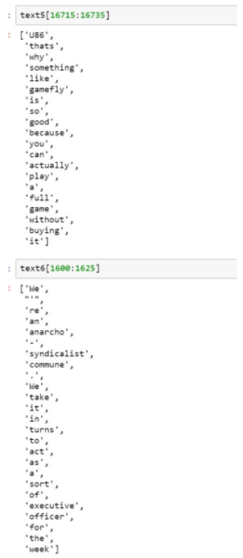
We can do the converse; given a word, find the index of when it first occurs:



```
text4.index('awaken')
```

```
173
```

Indexes are a common way to access the words of a text, or, more generally, the elements of any list. Python permits us to access sublists as well, extracting manageable pieces of language from large texts, a technique known as slicing.



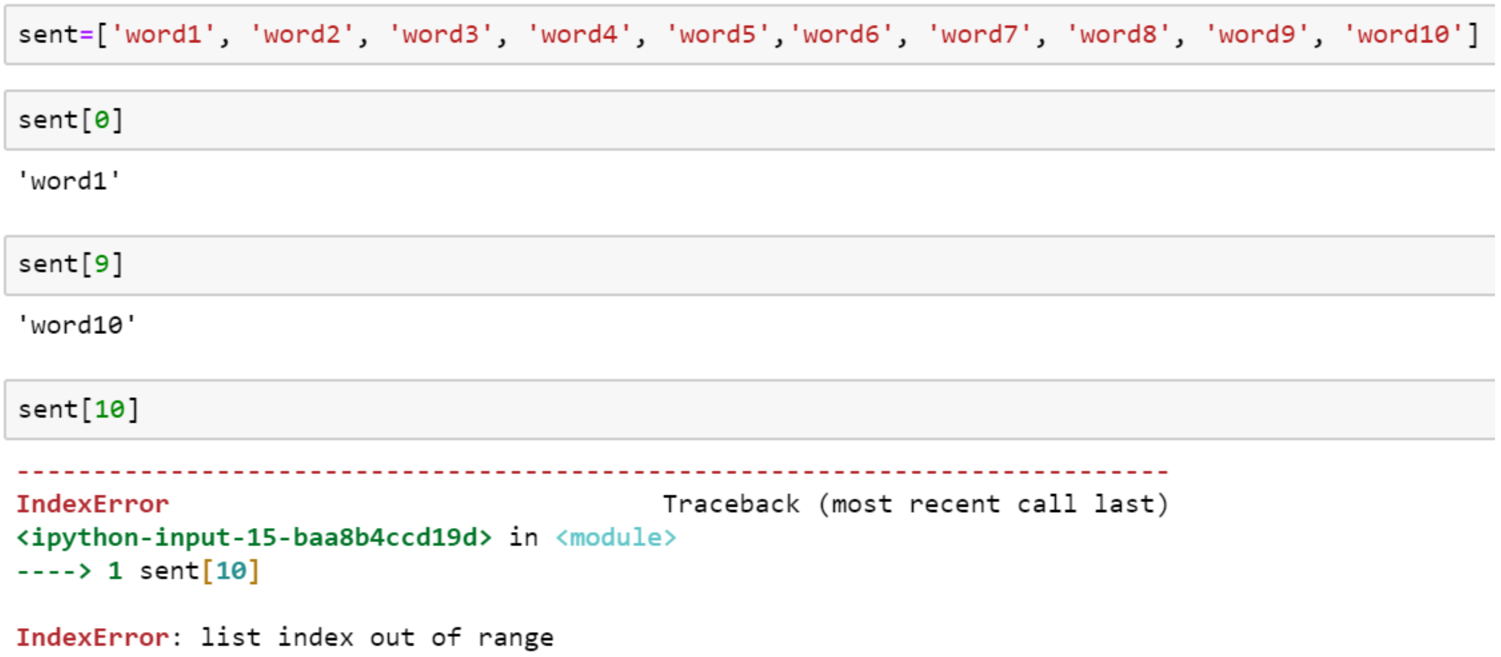
```
text5[16715:16735]
```

```
['U86',
 'thats',
 'why',
 'something',
 'like',
 'gamefly',
 'is',
 'so',
 'good',
 'because',
 'you',
 'can',
 'actually',
 'play',
 'a',
 'full',
 'game',
 'without',
 'buying',
 'it']
```

```
text6[1600:1625]
```

```
['We',
 "'",
 're',
 'an',
 'anarcho',
 '-',
 'syndicalist',
 'commune',
 '.',
 'We',
 'take',
 'it',
 'in',
 'turns',
 'to',
 'act',
 'as',
 'a',
 'sort',
 'of',
 'executive',
 'officer',
 'for',
 'the',
 'week']
```

Indexes have some subtleties, and we'll explore these with the help of an artificial sentence:



```
sent=['word1', 'word2', 'word3', 'word4', 'word5','word6', 'word7', 'word8', 'word9', 'word10']
```

```
sent[0]
```

```
'word1'
```

```
sent[9]
```

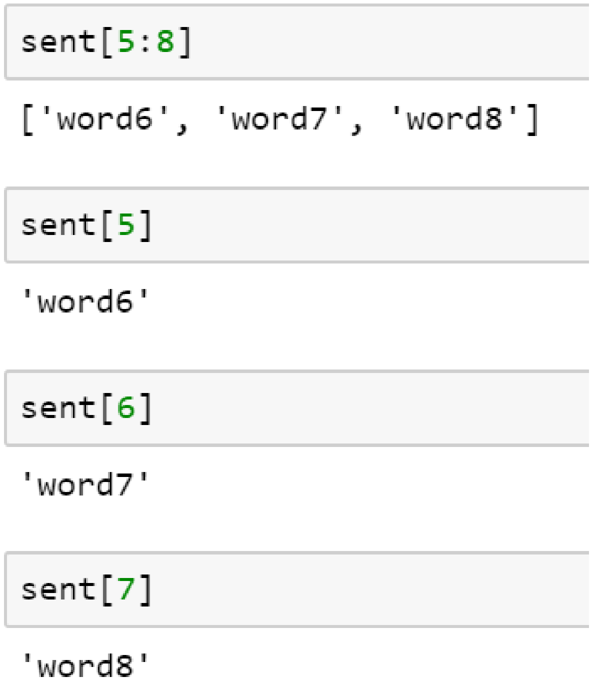
```
'word10'
```

```
sent[10]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-15-baa8b4ccd19d> in <module>
----> 1 sent[10]

IndexError: list index out of range
```

Let's take a closer look at slicing, using our artificial sentence again. Here we verify that the slice 5:8 includes sent elements at indexes 5, 6, and 7:

By convention, m:n means elements m…n-1.



```
sent[5:8]
```

```
['word6', 'word7', 'word8']
```

```
sent[5]
```

```
'word6'
```

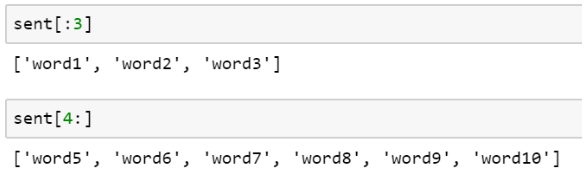
```
sent[6]
```

```
'word7'
```

```
sent[7]
```

```
'word8'
```

As the next example shows, we can omit the first number if the slice begins at the start of the list [1], and we can omit the second number if the slice goes to the end [2]:



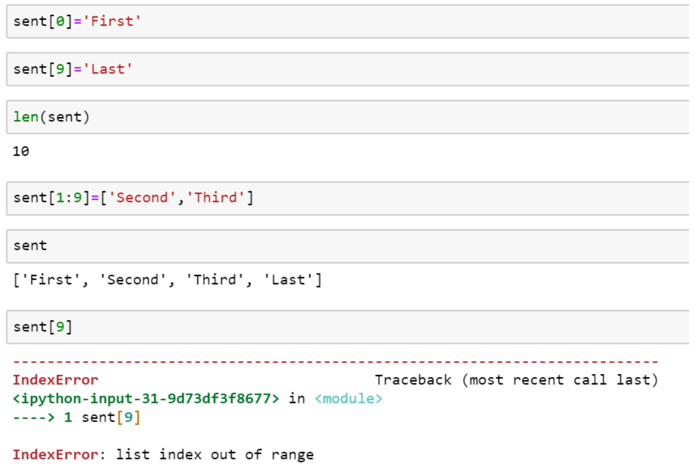
```
sent[:3]
```

```
['word1', 'word2', 'word3']
```

```
sent[4:]
```

```
['word5', 'word6', 'word7', 'word8', 'word9', 'word10']
```

We can modify an element of a list by assigning to one of its index values. In the next example, we put sent[0] on the left of the equals sign. We can also replace an entire slice with new material. A consequence of this last change is that the list only has four elements, and accessing a later value generates an error.



```
sent[0]='First'
```

```
sent[9]='Last'
```

```
len(sent)
```

```
10
```

```
sent[1:9]=['Second','Third']
```

```
sent
```

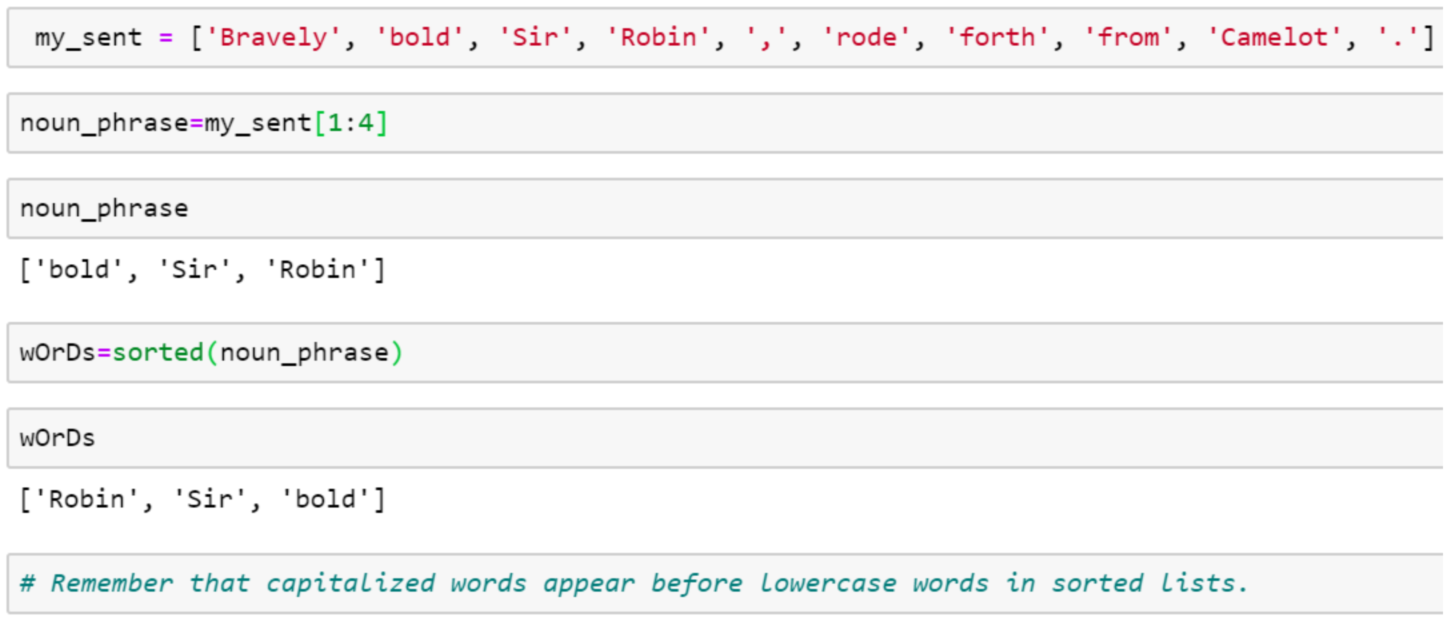
```
['First', 'Second', 'Third', 'Last']
```

```
sent[9]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-31-9d73df3f8677> in <module>
----> 1 sent[9]

IndexError: list index out of range
```

2.3 Variables

Such lines have the form: variable = expression. Python will evaluate the expression, and save its result to the variable. This process is called **assignment**. It does not generate any output; you have to type the variable on a line of its own to inspect its contents.



```
my_sent = ['Bravely', 'bold', 'Sir', 'Robin', ',', 'rode', 'forth', 'from', 'Camelot', '.']
```

```
noun_phrase=my_sent[1:4]
```

```
noun_phrase
```

```
['bold', 'Sir', 'Robin']
```

```
wOrDs=sorted(noun_phrase)
```

```
wOrDs
```

```
['Robin', 'Sir', 'bold']
```

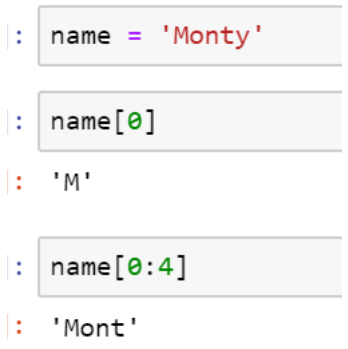
```
# Remember that capitalized words appear before lowercase words in sorted lists.
```

Caution!

- Take care with your choice of names (or identifiers) for Python variables.
- First, you should start the name with a letter, optionally followed by digits (0 to 9) or letters. Thus, abc23 is fine, but 23abc will cause a syntax error.
- Names are case-sensitive, which means that myVar and myvar are distinct variables. Variable names cannot contain whitespace, but you can separate words using an underscore, e.g., my_var.
- Be careful not to insert a hyphen instead of an underscore: my-var is wrong, since Python interprets the "-" as a minus sign.

2.4 Strings

- Some of the methods we used to access the elements of a list also work with individual words, or strings. For example, we can assign a string to a variable [1], index a string [2], and slice a string [3]:



```
: name = 'Monty'

: name[0]
: 'M'

: name[0:4]
: 'Mont'
```

We can also perform multiplication and addition with strings:

```
name*2
```

```
'MontyMonty'
```

```
name+'!'
```

```
'Monty!'
```

We can join the words of a list to make a single string, or split a string into a list, as follows:

```
' '.join(['Monty', 'Python'])
```

```
'Monty Python'
```

```
'Monty Python'.split()
```

```
['Monty', 'Python']
```

Outline

- 1. Computing with Language: Texts and Words
- 2. A Closer Look at Python: Texts as Lists of Words
- **3. Computing with Language: Simple Statistics**
- 4. Back to Python: Making Decisions and Taking Control

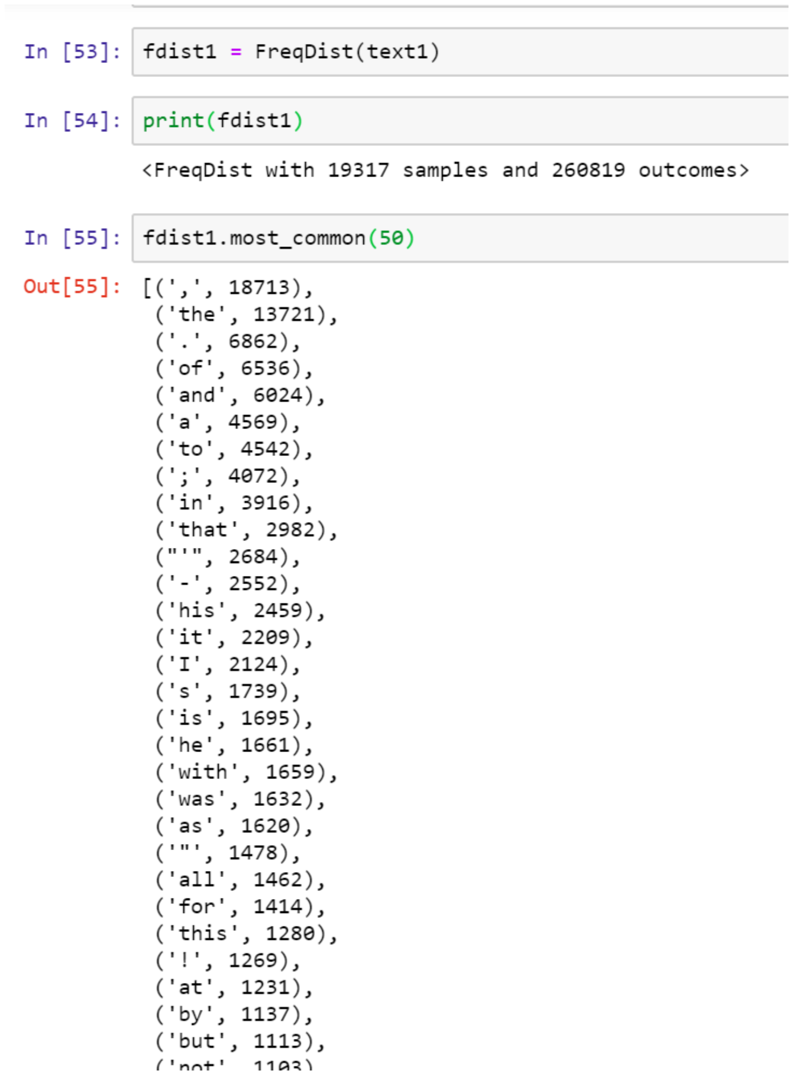
3. Computing with Language: Simple Statistics

- Let's return to our exploration of the ways we can bring our computational resources to bear on large quantities of text. We began this discussion in 1, and saw how to search for words in context, how to compile the vocabulary of a text, how to generate random text in the same style, and so on.

- In this section we pick up the question of what makes a text distinct, and use automatic methods to find characteristic words and expressions of a text.

3.1 Frequency Distributions

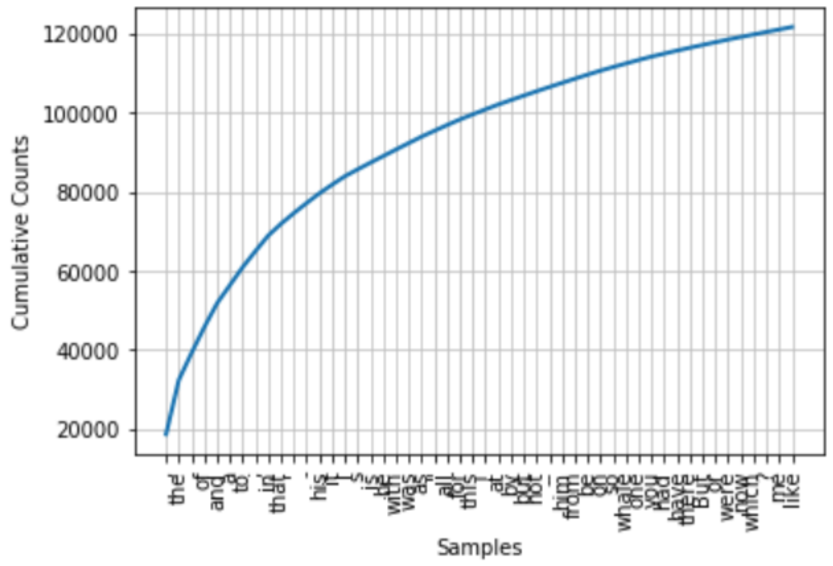
- It is a "distribution" because it tells us how the total number of word tokens in the text are distributed across the vocabulary items.
- Since we often need frequency distributions in language processing, NLTK provides built-in support for them. Let's use a FreqDist to find the 50 most frequent words of Moby Dick:



```
In [53]: fdist1 = FreqDist(text1)

In [54]: print(fdist1)

         <FreqDist with 19317 samples and 260819 outcomes>

In [55]: fdist1.most_common(50)

Out[55]: [(',', 18713),
          ('the', 13721),
          ('.', 6862),
          ('of', 6536),
          ('and', 6024),
          ('a', 4569),
          ('to', 4542),
          (';', 4072),
          ('in', 3916),
          ('that', 2982),
          ("'", 2684),
          ('-', 2552),
          ('his', 2459),
          ('it', 2209),
          ('I', 2124),
          ('s', 1739),
          ('is', 1695),
          ('he', 1661),
          ('with', 1659),
          ('was', 1632),
          ('as', 1620),
          ('"', 1478),
          ('all', 1462),
          ('for', 1414),
          ('this', 1280),
          ('!', 1269),
          ('at', 1231),
          ('by', 1137),
          ('but', 1113),
          ('not', 1103)
```

Do any words produced in the last example help us grasp the topic or genre of this text? Only one word, whale, is slightly informative! It occurs over 900 times. The rest of the words tell us nothing about the text; they're just English "plumbing." What proportion of the text is taken up with such words? We can generate a cumulative frequency plot for these words, using fdist1.plot(50, cumulative=True), to produce the graph in 3.2. These 50 words account for nearly half the book!

```
In [60]: fdist1.plot(50, cumulative=True)
```



```
fdist1['whale']

906
```

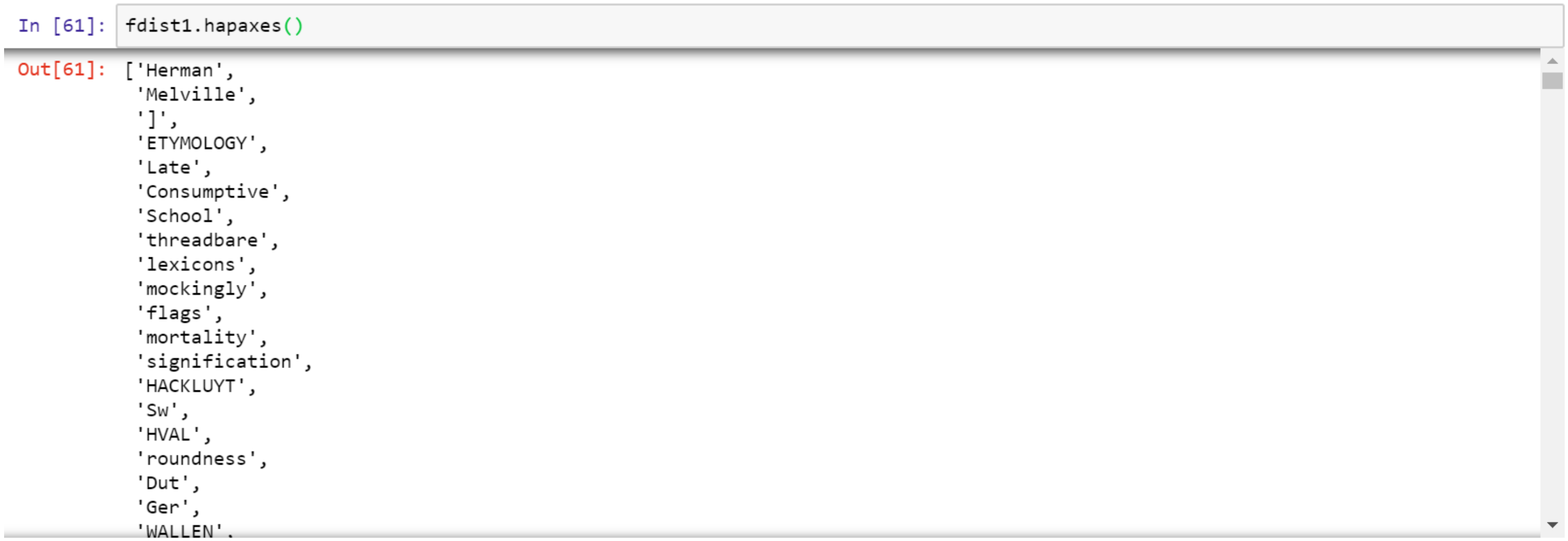
```
: len(set(text1))

: 19317
```

```
: len(text1)

: 260819
```

***Figure 3.2**: Cumulative Frequency Plot for 50 Most Frequently Words in Moby Dick: these account for nearly half of the tokens.*

The code and content in this slide is cited from Natural Language Processing with Python, by Steven Bird, Ewan Klein, and Edward Loper. O'Reilly Media, 978-0-596-51649-9

If the frequent words don't help us, how about the words that occur once only, the so-called **hapaxes**? View them by typing fdist1.hapaxes(). This list contains lexicographer, cetological, contraband, expostulations, and about 9,000 others. It seems that there are too many rare words, and without seeing the context we probably can't guess what half of the hapaxes mean in any case! Since neither frequent nor infrequent words help, we need to try something else.



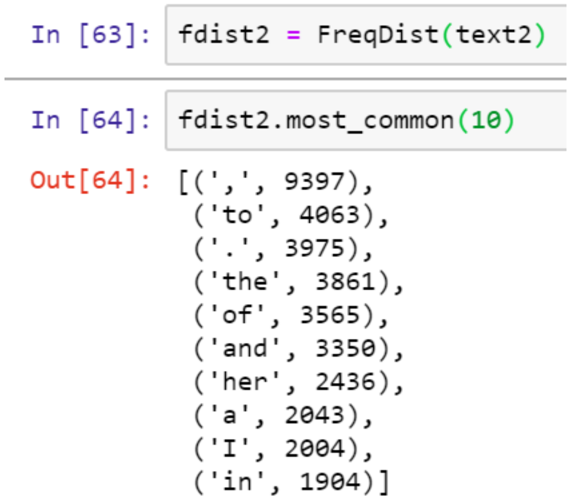
```
In [61]: fdist1.hapaxes()

Out[61]: ['Herman',
          'Melville',
          ']',
          'ETYMOLOGY',
          'Late',
          'Consumptive',
          'School',
          'threadbare',
          'lexicons',
          'mockingly',
          'flags',
          'mortality',
          'signification',
          'HACKLUYT',
          'Sw',
          'HVAL',
          'roundness',
          'Dut',
          'Ger',
          'WALLEN',
```

Exercise 6

- Try the preceding frequency distribution example for yourself, for text2. Output the 10 most frequent tokens.

Exercise 6

- Try the preceding frequency distribution example for yourself, for text2. Output the 10 most frequent tokens.



```
In [63]: fdist2 = FreqDist(text2)

In [64]: fdist2.most_common(10)

Out[64]: [(',', 9397),
          ('to', 4063),
          ('.', 3975),
          ('the', 3861),
          ('of', 3565),
          ('and', 3350),
          ('her', 2436),
          ('a', 2043),
          ('I', 2004),
          ('in', 1904)]
```

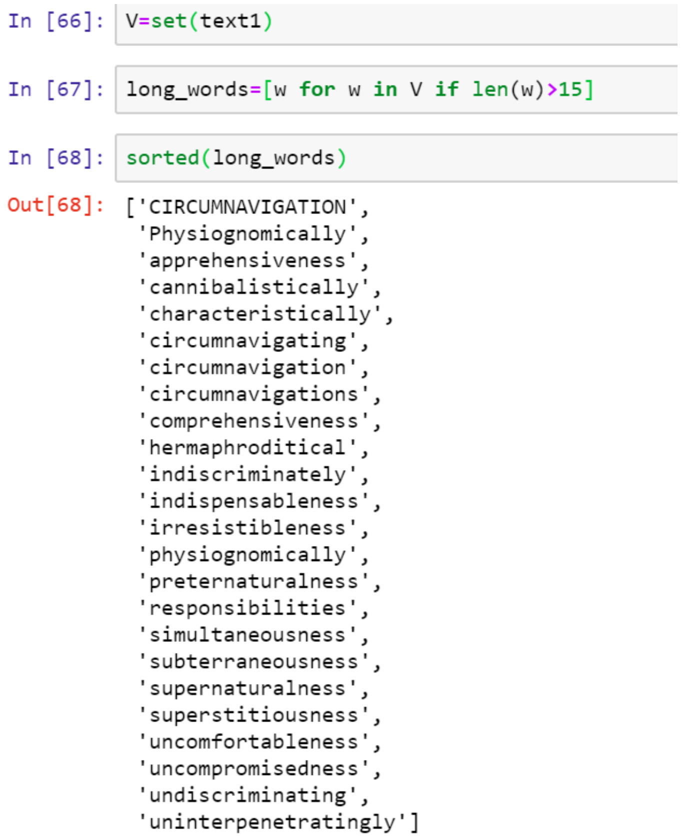
3.2 Fine-grained Selection of Words

- Next, let's look at the long words of a text; perhaps these will be more characteristic and informative. For this we adapt some notation from set theory. We would like to find the words from the vocabulary of the text that are more than 15 characters long. Let's call this property P, so that P(w) is true if and only if w is more than 15 characters long. Now we can express the words of interest using mathematical set notation as shown in (1a). This means "the set of all w such that w is an element of V (the vocabulary) and w has property P".

a. $\{w \mid w \in V \,\&\, P(w)\}$

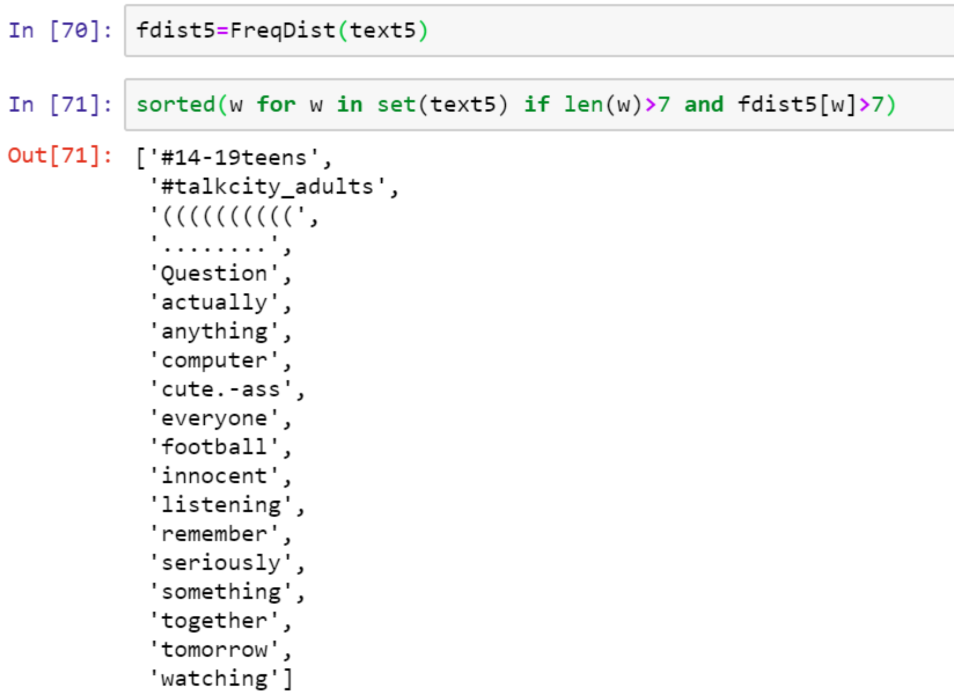
b.
```
[w for w in V if p(w)]
```

For each word w in the vocabulary V, we check whether len(w) is greater than 15; all other words will be ignored



```
In [66]: V=set(text1)

In [67]: long_words=[w for w in V if len(w)>15]

In [68]: sorted(long_words)

Out[68]: ['CIRCUMNAVIGATION',
          'Physiognomically',
          'apprehensiveness',
          'cannibalistically',
          'characteristically',
          'circumnavigating',
          'circumnavigation',
          'circumnavigations',
          'comprehensiveness',
          'hermaphroditical',
          'indiscriminately',
          'indispensableness',
          'irresistibleness',
          'physiognomically',
          'preternaturalness',
          'responsibilities',
          'simultaneousness',
          'subterraneousness',
          'supernaturalness',
          'superstitiousness',
          'uncomfortableness',
          'uncompromisedness',
          'undiscriminating',
          'uninterpenetratingly']
```

These very long words are often hapaxes (i.e., unique) and perhaps it would be better to find frequently occurring long words. This seems promising since it eliminates frequent short words (e.g., the) and infrequent long words (e.g. antiphilosophists). Here are all words from the chat corpus (text5) that are longer than seven characters, that occur more than seven times:



```
In [70]: fdist5=FreqDist(text5)

In [71]: sorted(w for w in set(text5) if len(w)>7 and fdist5[w]>7)

Out[71]: ['#14-19teens',
          '#talkcity_adults',
          '((((((((((',
          '........',
          'Question',
          'actually',
          'anything',
          'computer',
          'cute.-ass',
          'everyone',
          'football',
          'innocent',
          'listening',
          'remember',
          'seriously',
          'something',
          'together',
          'tomorrow',
          'watching']
```

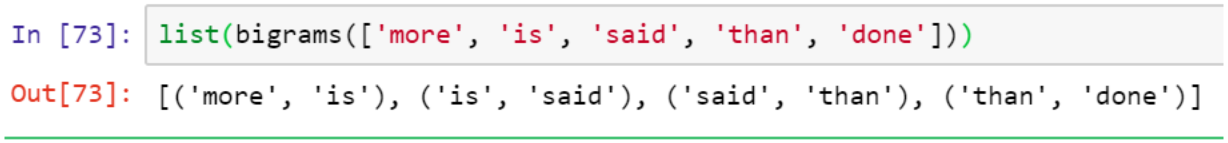
Summary

- Notice how we have used two conditions: len(w) > 7 ensures that the words are longer than seven letters, and fdist5[w] > 7 ensures that these words occur more than seven times. At last we have managed to automatically identify the frequently-occurring content-bearing words of the text.
- It is a modest but important milestone: a tiny piece of code, processing tens of thousands of words, produces some informative output.

3.3. Collocations and Bigrams

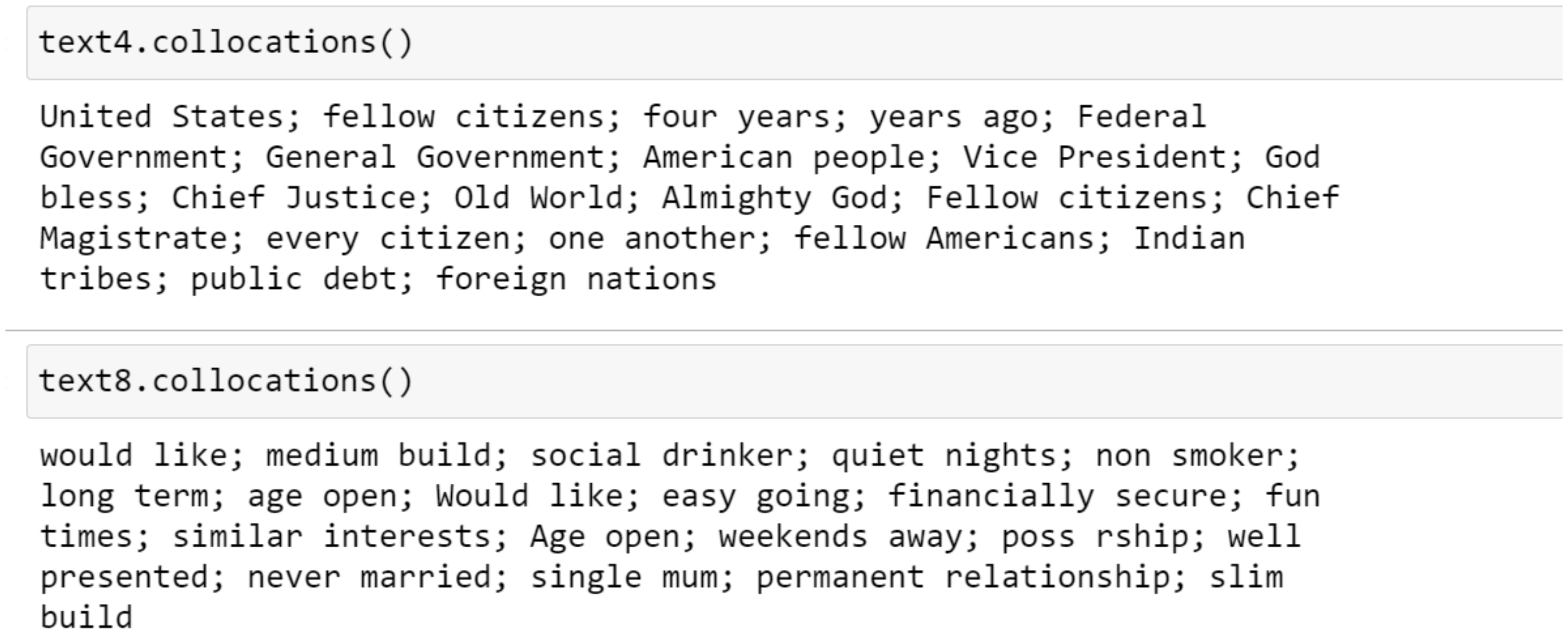
- A **collocation** is a sequence of words that occur together unusually often. Thus red wine is a collocation, whereas the wine is not. A characteristic of collocations is that they are resistant to substitution with words that have similar senses; for example, maroon wine sounds definitely odd.

- To get a handle on collocations, we start off by extracting from a text a list of word pairs, also known as **bigrams**. This is easily accomplished with the function bigrams( ):



```
In [73]: list(bigrams(['more', 'is', 'said', 'than', 'done']))

Out[73]: [('more', 'is'), ('is', 'said'), ('said', 'than'), ('than', 'done')]
```

- Here we see that the pair of words than-done is a bigram, and we write it in Python as ('than', 'done').
- Now, collocations are essentially just **frequent bigrams**, except that we want to pay more attention to the cases that involve rare words. In particular, we want to find bigrams that **occur more often than we would expect** based on the frequency of the individual words. The collocations() function does this for us.



```
text4.collocations()
```

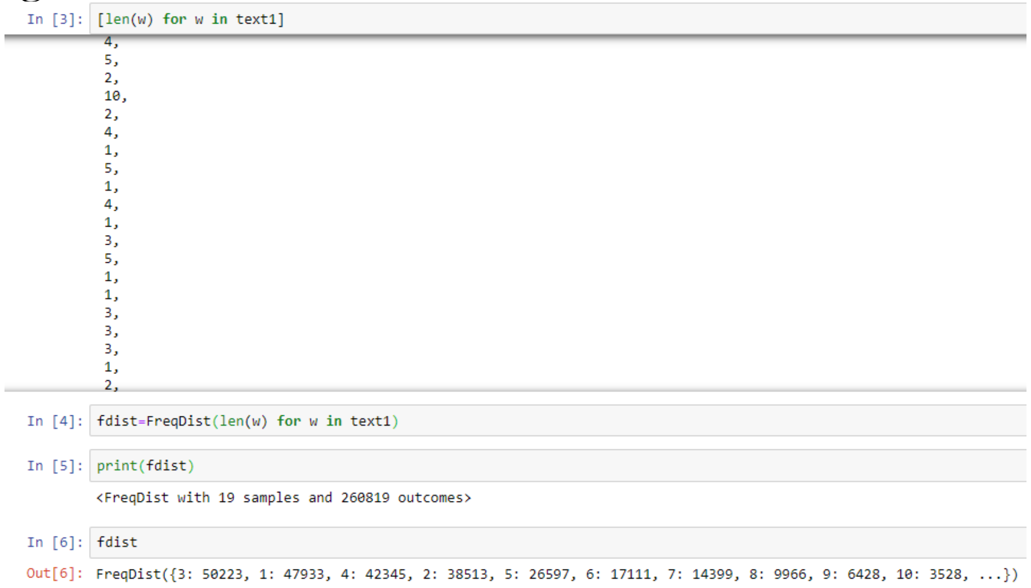
```
United States; fellow citizens; four years; years ago; Federal
Government; General Government; American people; Vice President; God
bless; Chief Justice; Old World; Almighty God; Fellow citizens; Chief
Magistrate; every citizen; one another; fellow Americans; Indian
tribes; public debt; foreign nations
```

```
text8.collocations()
```

```
would like; medium build; social drinker; quiet nights; non smoker;
long term; age open; Would like; easy going; financially secure; fun
times; similar interests; Age open; weekends away; poss rship; well
presented; never married; single mum; permanent relationship; slim
build
```

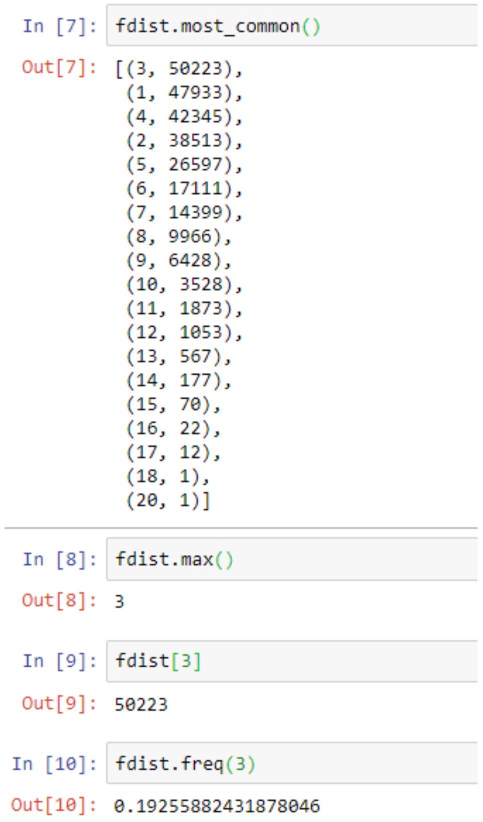
3.4 Counting Other Things

- Counting words is useful, but we can count other things too. For example, we can look at the distribution of word lengths in a text, by creating a FreqDist out of a long list of numbers, where each number is the length of the corresponding word in the text:



```
In [3]: [len(w) for w in text1]
         4,
         5,
         2,
         10,
         2,
         4,
         1,
         5,
         1,
         4,
         1,
         3,
         5,
         1,
         1,
         3,
         3,
         3,
         1,
         2,

In [4]: fdist=FreqDist(len(w) for w in text1)

In [5]: print(fdist)
        <FreqDist with 19 samples and 260819 outcomes>

In [6]: fdist
Out[6]: FreqDist({3: 50223, 1: 47933, 4: 42345, 2: 38513, 5: 26597, 6: 17111, 7: 14399, 8: 9966, 9: 6428, 10: 3528, ...})
```

From this we see that the most frequent word length is 3, and that words of length 3 account for roughly 50,000 (or 20%) of the words making up the book.



```
In [7]: fdist.most_common()
Out[7]: [(3, 50223),
         (1, 47933),
         (4, 42345),
         (2, 38513),
         (5, 26597),
         (6, 17111),
         (7, 14399),
         (8, 9966),
         (9, 6428),
         (10, 3528),
         (11, 1873),
         (12, 1053),
         (13, 567),
         (14, 177),
         (15, 70),
         (16, 22),
         (17, 12),
         (18, 1),
         (20, 1)]

In [8]: fdist.max()
Out[8]: 3

In [9]: fdist[3]
Out[9]: 50223

In [10]: fdist.freq(3)
Out[10]: 0.19255882431878046
```

Functions Defined for NLTK's Frequency Distributions

Example	Description
`fdist = FreqDist(samples)`	create a frequency distribution containing the given samples
`fdist[sample] += 1`	increment the count for this sample
`fdist['monstrous']`	count of the number of times a given sample occurred
`fdist.freq('monstrous')`	frequency of a given sample
`fdist.N()`	total number of samples
`fdist.most_common(n)`	the `n` most common samples and their frequencies
`for sample in fdist:`	iterate over the samples
`fdist.max()`	sample with the greatest count
`fdist.tabulate()`	tabulate the frequency distribution
`fdist.plot()`	graphical plot of the frequency distribution
`fdist.plot(cumulative=True)`	cumulative plot of the frequency distribution
`fdist1 |= fdist2`	update `fdist1` with counts from `fdist2`
`fdist1 < fdist2`	test if samples in `fdist1` occur less frequently than in `fdist2`

```
fdist = FreqDist(text1)
```

```
fdist
```

```
FreqDist({',': 18713, 'the': 13721, '.': 6862, 'of': 6536, 'and': 6024, 'a': 4569, 'to': 4542, ';': 4
072, 'in': 3916, 'that': 2982, ...})
```

```
fdist['monstrous']
```

```
10
```

```
fdist.freq('monstrous')
```

```
3.834076505162584e-05
```

```
fdist.N()
```

```
260819
```

```
fdist.most_common(10)
```

```
[(',', 18713),
 ('the', 13721),
 ('.', 6862),
 ('of', 6536),
 ('and', 6024),
 ('a', 4569),
 ('to', 4542),
 (';', 4072),
 ('in', 3916),
 ('that', 2982)]
```

T

```
fdist.max()
```

Outline

- 1. Computing with Language: Texts and Words
- 2. A Closer Look at Python: Texts as Lists of Words
- 3. Computing with Language: Simple Statistics
- **4. Back to Python: Making Decisions and Taking Control**

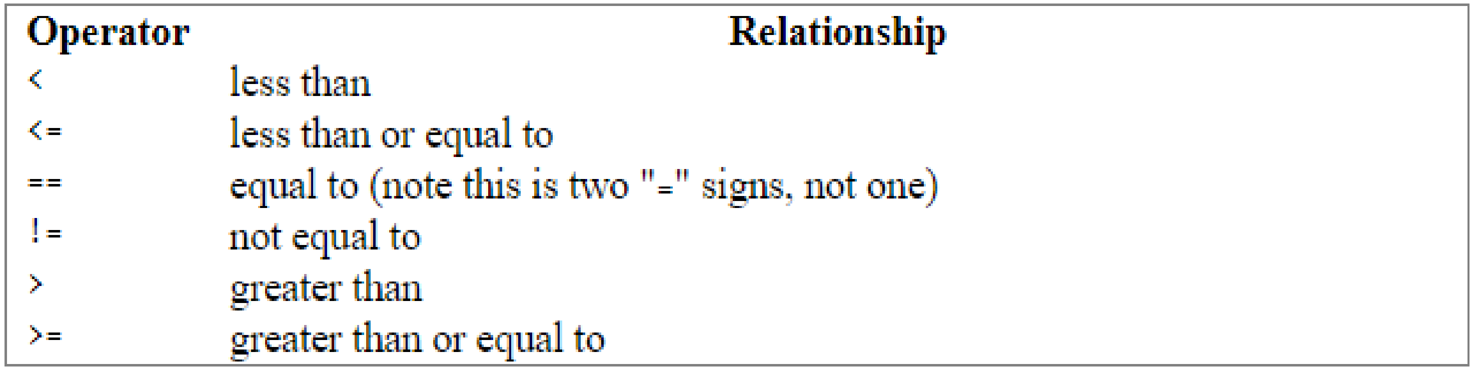
4 Back to Python: Making Decisions and Taking Control

- So far, our little programs have had some interesting qualities: the ability to work with language, and the potential to save human effort through automation. A key feature of programming is the ability of machines to make decisions on our behalf, executing instructions when certain conditions are met, or repeatedly looping through text data until some condition is satisfied. This feature is known as control, and is the focus of this section

4.1 Conditionals

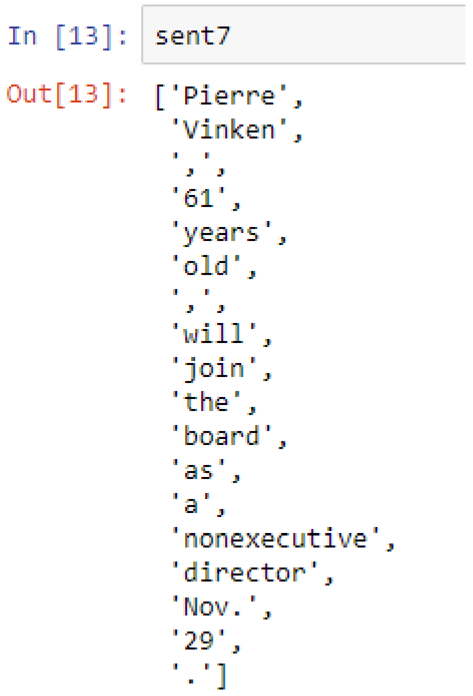
- Python supports a wide range of operators, such as < and >=, for testing the relationship between values

Numerical Comparison Operators

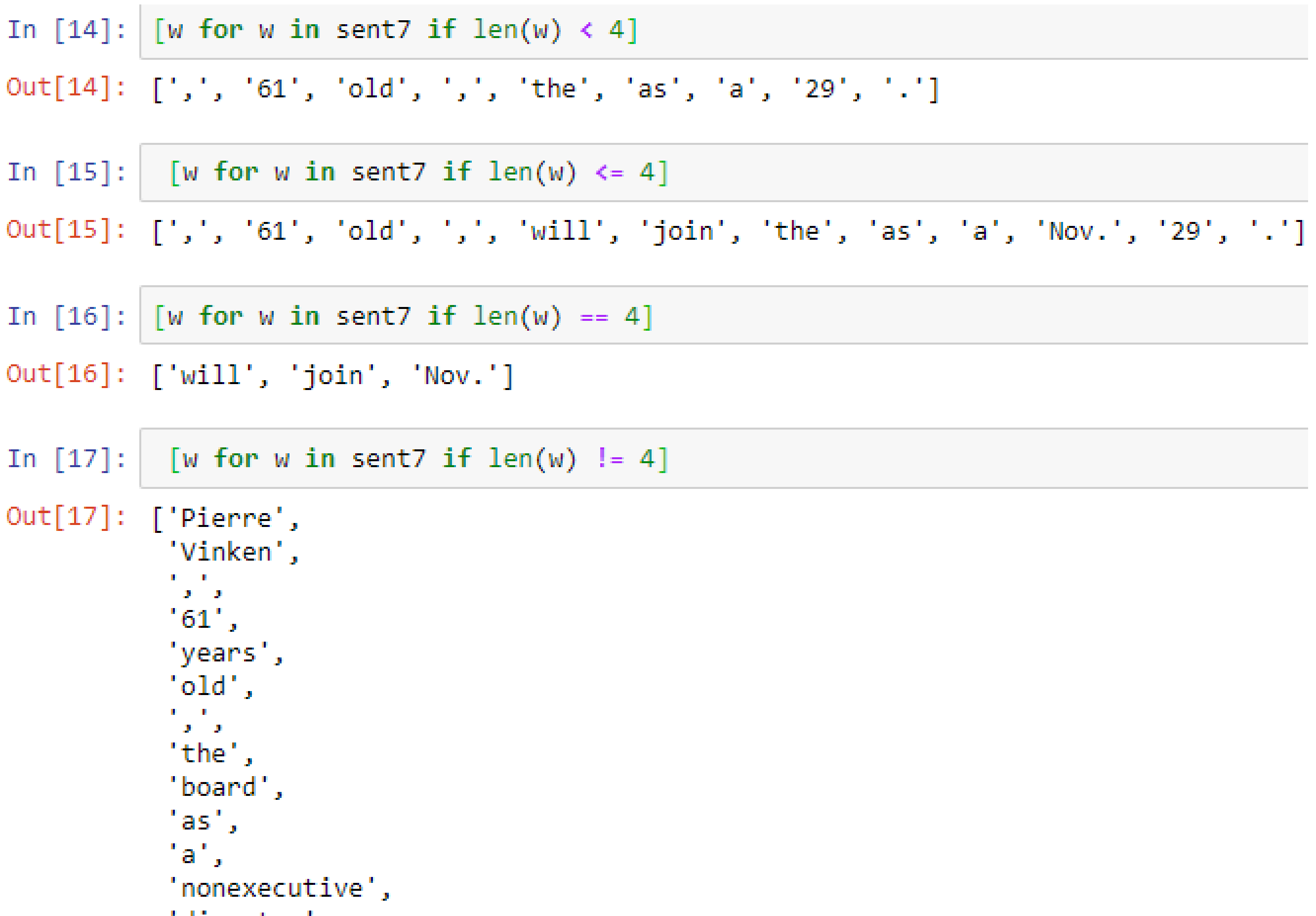


Operator	Relationship
<	less than
<=	less than or equal to
==	equal to (note this is two "=" signs, not one)
!=	not equal to
>	greater than
>=	greater than or equal to

We can use these to select different words from a sentence of news text. Here are some examples — only the operator is changed from one line to the next. They all use sent7, the first sentence from text7 (Wall Street Journal).



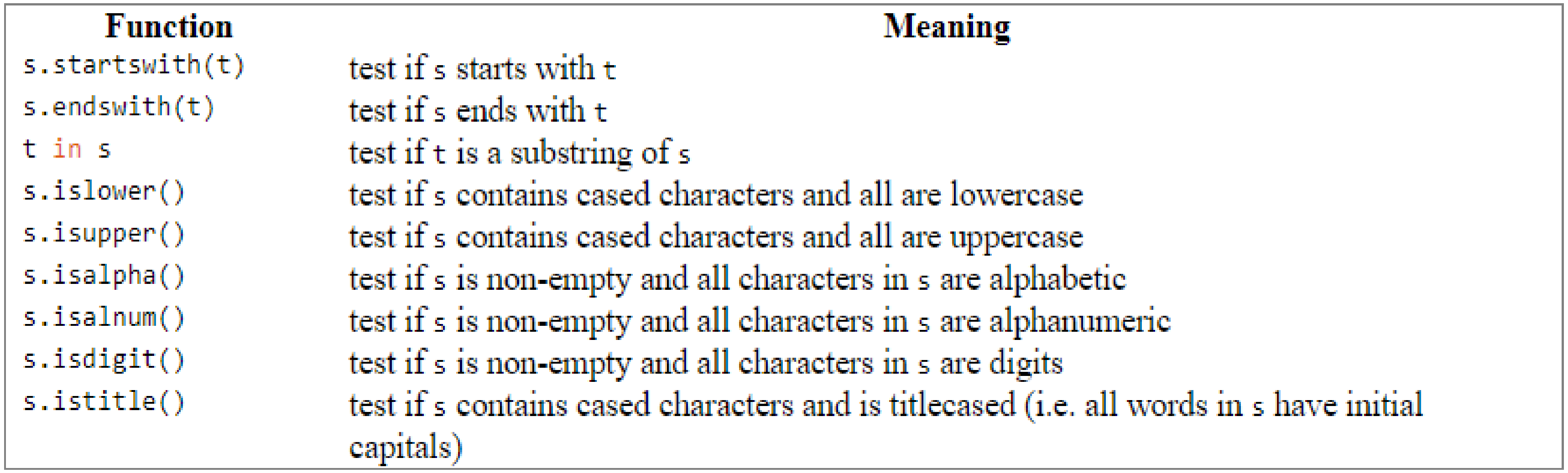
```
In [13]: sent7

Out[13]: ['Pierre',
          'Vinken',
          ',',
          '61',
          'years',
          'old',
          ',',
          'will',
          'join',
          'the',
          'board',
          'as',
          'a',
          'nonexecutive',
          'director',
          'Nov.',
          '29',
          '.']
```



```
In [14]: [w for w in sent7 if len(w) < 4]

Out[14]: [',', '61', 'old', ',', 'the', 'as', 'a', '29', '.']

In [15]:  [w for w in sent7 if len(w) <= 4]

Out[15]: [',', '61', 'old', ',', 'will', 'join', 'the', 'as', 'a', 'Nov.', '29', '.']

In [16]: [w for w in sent7 if len(w) == 4]

Out[16]: ['will', 'join', 'Nov.']

In [17]:  [w for w in sent7 if len(w) != 4]

Out[17]: ['Pierre',
          'Vinken',
          ',',
          '61',
          'years',
          'old',
          ',',
          'the',
          'board',
          'as',
          'a',
          'nonexecutive',
```

Some Word Comparison Operators

- There is a common pattern to all of these examples: [*w* for *w* in text if *condition*], where condition is a Python "test" that yields either true or false.



Function	Meaning
`s.startswith(t)`	test if `s` starts with `t`
`s.endswith(t)`	test if `s` ends with `t`
`t in s`	test if `t` is a substring of `s`
`s.islower()`	test if `s` contains cased characters and all are lowercase
`s.isupper()`	test if `s` contains cased characters and all are uppercase
`s.isalpha()`	test if `s` is non-empty and all characters in `s` are alphabetic
`s.isalnum()`	test if `s` is non-empty and all characters in `s` are alphanumeric
`s.isdigit()`	test if `s` is non-empty and all characters in `s` are digits
`s.istitle()`	test if `s` contains cased characters and is titlecased (i.e. all words in `s` have initial capitals)

```
sorted(w for w in set(sent7) if w.startswith('non'))
```

```
['nonexecutive']
```

```
sorted(item for item in set(sent7) if item.islower())
```

```
['a',
 'as',
 'board',
 'director',
 'join',
 'nonexecutive',
 'old',
 'the',
 'will',
 'years']
```

```
sorted(item for item in set(sent7) if item.isupper())
```

```
[]
```

```
sorted(item for item in set(sent7) if item.isalpha())
```

```
['Pierre',
 'Vinken',
 'a',
 'as',
 'board',
 'director',
 'join',
 'nonexecutive',
 'old',
 'the',
 'will',
 'years']
```

```
sorted(item for item in set(sent7) if item.isalnum())
```

```
['29',
 '61',
 'Pierre',
 'Vinken',
 'a',
 'as',
 'board',
 'director',
 'join',
 'nonexecutive',
 'old',
 'the',
 'will',
 'years']
```

```
sorted(w for w in set(text1) if w.endswith('ableness'))
```

```
['comfortableness',
 'honourableness',
 'immutableness',
 'indispensableness',
 'indomitableness',
 'intolerableness',
 'palpableness',
 'reasonableness',
 'uncomfortableness']
```

```
sorted(term for term in set(text4) if 'gnt' in term)
```

```
['Sovereignty', 'sovereignties', 'sovereignty']
```

```
sorted(item for item in set(sent7) if item.istitle())
```

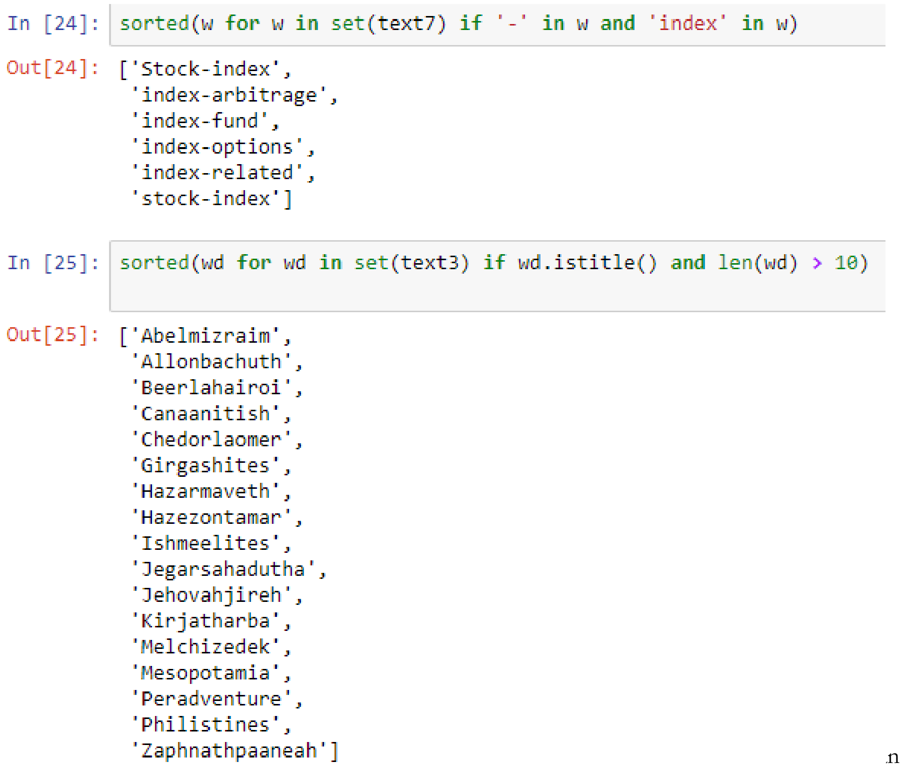
```
['Nov.', 'Pierre', 'Vinken']
```

```
sorted(item for item in set(sent7) if item.isdigit())
```

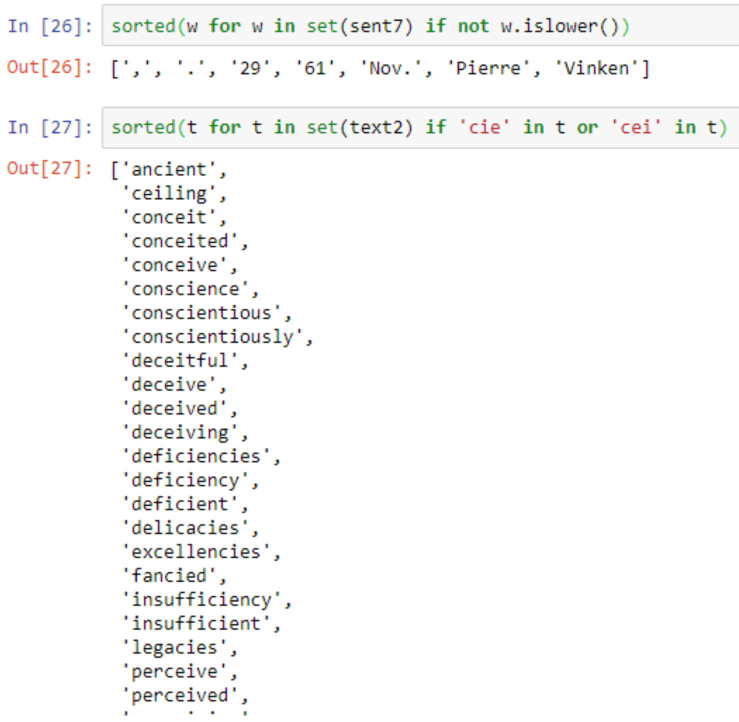
```
['29', '61']
```

We can also create more complex conditions. If c is a condition, then not c is also a condition. If we have two conditions c1 and c2, then we can combine them to form a new condition using conjunction and disjunction: c1 and c2, c1 or c2.

Exercise 7: Run the following examples and try to explain what is going on in each one.



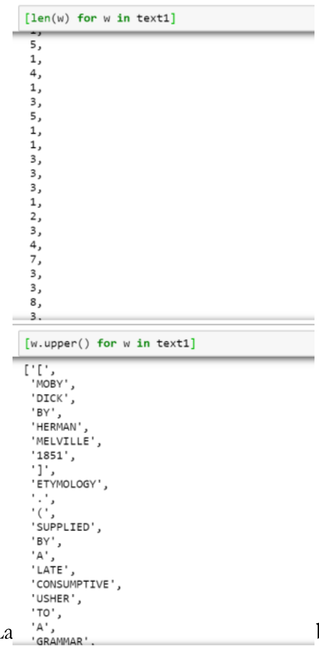
```
In [24]: sorted(w for w in set(text7) if '-' in w and 'index' in w)

Out[24]: ['Stock-index',
          'index-arbitrage',
          'index-fund',
          'index-options',
          'index-related',
          'stock-index']

In [25]: sorted(wd for wd in set(text3) if wd.istitle() and len(wd) > 10)

Out[25]: ['Abelmizraim',
          'Allonbachuth',
          'Beerlahairoi',
          'Canaanitish',
          'Chedorlaomer',
          'Girgashites',
          'Hazarmaveth',
          'Hazezontamar',
          'Ishmeelites',
          'Jegarsahadutha',
          'Jehovahjireh',
          'Kirjatharba',
          'Melchizedek',
          'Mesopotamia',
          'Peradventure',
          'Philistines',
          'Zaphnathpaaneah']
```



```
In [26]: sorted(w for w in set(sent7) if not w.islower())

Out[26]: [',', '.', '29', '61', 'Nov.', 'Pierre', 'Vinken']

In [27]: sorted(t for t in set(text2) if 'cie' in t or 'cei' in t)

Out[27]: ['ancient',
          'ceiling',
          'conceit',
          'conceited',
          'conceive',
          'conscience',
          'conscientious',
          'conscientiously',
          'deceitful',
          'deceive',
          'deceived',
          'deceiving',
          'deficiencies',
          'deficiency',
          'deficient',
          'delicacies',
          'excellencies',
          'fancied',
          'insufficiency',
          'insufficient',
          'legacies',
          'perceive',
          'perceived',
```

4.2 Operating on Every Element

- In 3, we saw some examples of counting items other than words. Let's take a closer look at the notation we used:



```
[len(w) for w in text1]
```

```
 5,
 1,
 4,
 1,
 3,
 5,
 1,
 1,
 3,
 3,
 3,
 1,
 2,
 3,
 4,
 7,
 3,
 3,
 8,
 3,
```

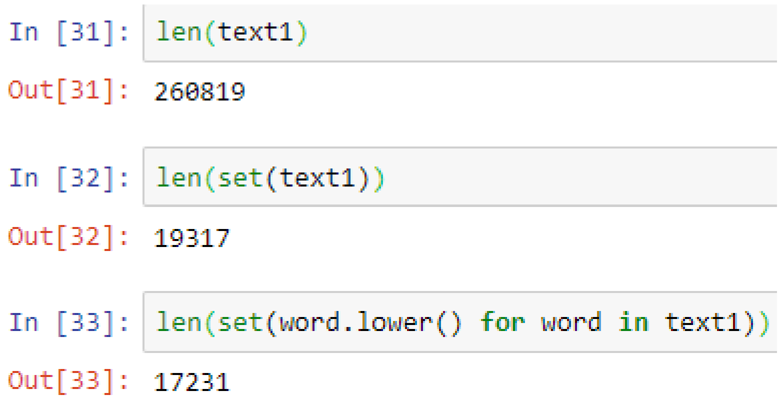
```
[w.upper() for w in text1]
```

```
['[',
 'MOBY',
 'DICK',
 'BY',
 'HERMAN',
 'MELVILLE',
 '1851',
 ']',
 'ETYMOLOGY',
 '.',
 '(',
 'SUPPLIED',
 'BY',
 'A',
 'LATE',
 'CONSUMPTIVE',
 'USHER',
 'TO',
 'A',
 'GRAMMAR',
```

4.2 Operating on Every Element

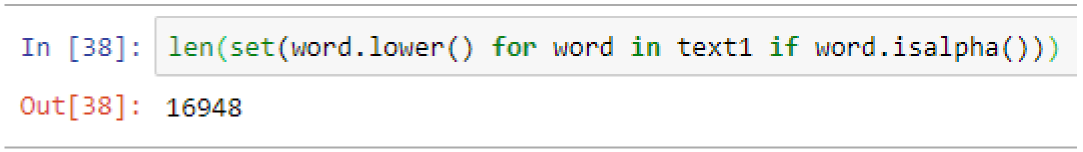
- These expressions have the form [f(w) for ...] or [w.f() for ...], where f is a function that operates on a word to compute its length, or to convert it to uppercase. For now, you don't need to understand the difference between the notations f(w) and w.f()
- Python performs the same operation on every element of a list

Now that we are not double-counting words like *This* and *this*, which differ only in capitalization, we've wiped 2,000 off the vocabulary count!



```
In [31]: len(text1)
Out[31]: 260819

In [32]: len(set(text1))
Out[32]: 19317

In [33]: len(set(word.lower() for word in text1))
Out[33]: 17231
```

We can go a step further and eliminate numbers and punctuation from the vocabulary count by filtering out any non-alphabetic items:



```
In [38]: len(set(word.lower() for word in text1 if word.isalpha()))
Out[38]: 16948
```

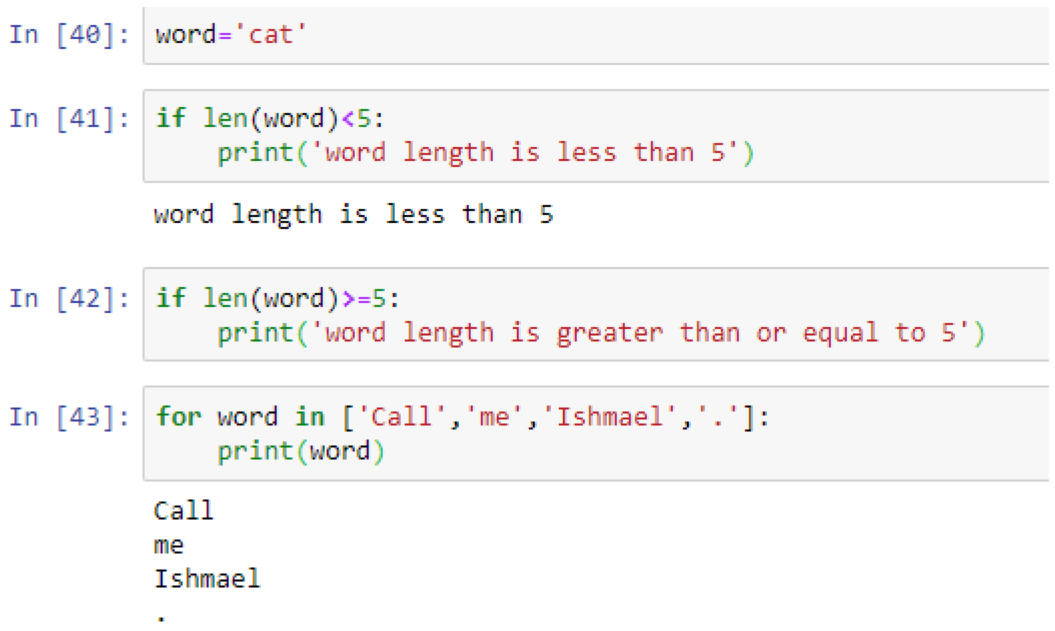
This example is slightly complicated: it lowercases all the purely alphabetic items.

4.3 Nested Code Blocks

- Most programming languages permit us to execute a block of code when a conditional expression, or if statement, is satisfied. We already saw examples of conditional tests in code like [w for w in sent7 if len(w) < 4].
- In the following program, we have created a variable called word containing the string value 'cat'. The if statement checks whether the test len(word) < 5 is true.
- It is, so the body of the if statement is invoked and the print statement is executed, displaying a message to the user.
- Remember to indent the print statement by typing four spaces.

If we change the conditional test to len(word) >= 5, to check that the length of word is greater than or equal to 5, then the test will no longer be true. This time, the body of the if statement will not be executed, and no message is shown to the user:

An if statement is known as a control structure because it controls whether the code in the indented block will be run. Another control structure is the for loop. Try the following, and remember to include the colon and the four spaces:



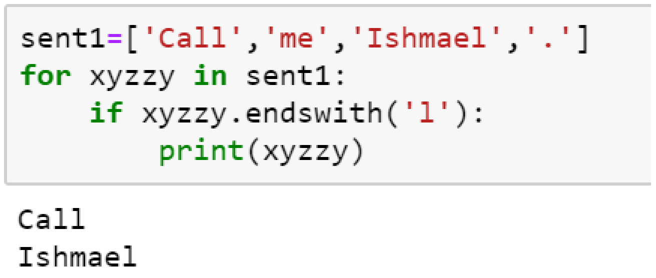
```
In [40]: word='cat'

In [41]: if len(word)<5:
             print('word length is less than 5')

         word length is less than 5

In [42]: if len(word)>=5:
             print('word length is greater than or equal to 5')

In [43]: for word in ['Call','me','Ishmael','.']:
             print(word)

         Call
         me
         Ishmael
         .
```

Loop

- This is called a loop because Python executes the code in circular fashion. It starts by performing the assignment word = 'Call', effectively using the word variable to name the first item of the list.
- Then, it displays the value of word to the user.
- Next, it goes back to the for statement, and performs the assignment word = 'me', before displaying this new value to the user, and so on.
- It continues in this fashion until every item of the list has been processed.

4.4 Looping with Conditions

- Now we can combine the if and for statements. We will loop over every item of the list, and print the item only if it ends with the letter l. We'll pick another name for the variable to demonstrate that Python doesn't try to make sense of variable names.

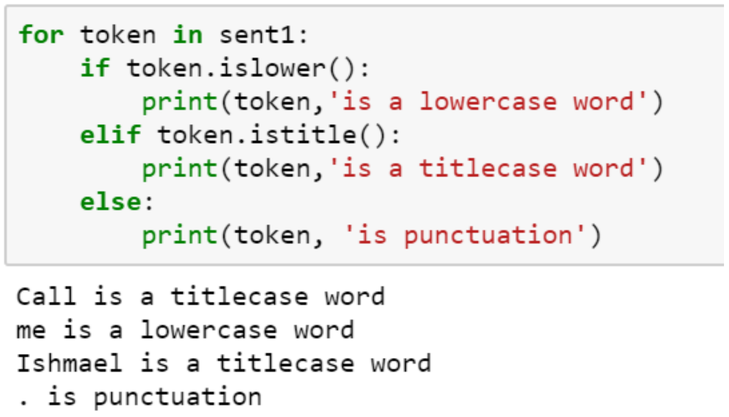


```
sent1=['Call','me','Ishmael','.']
for xyzzy in sent1:
    if xyzzy.endswith('l'):
        print(xyzzy)
```

```
Call
Ishmael
```

You will notice that if and for statements have a colon at the end of the line, before the indentation begins. In fact, all Python control structures end with a colon. The colon indicates that the current statement relates to the indented block that follows.

We can also specify an action to be taken if the condition of the if statement is not met. Here we see the elif (else if) statement, and the else statement. Notice that these also have colons before the indented code.

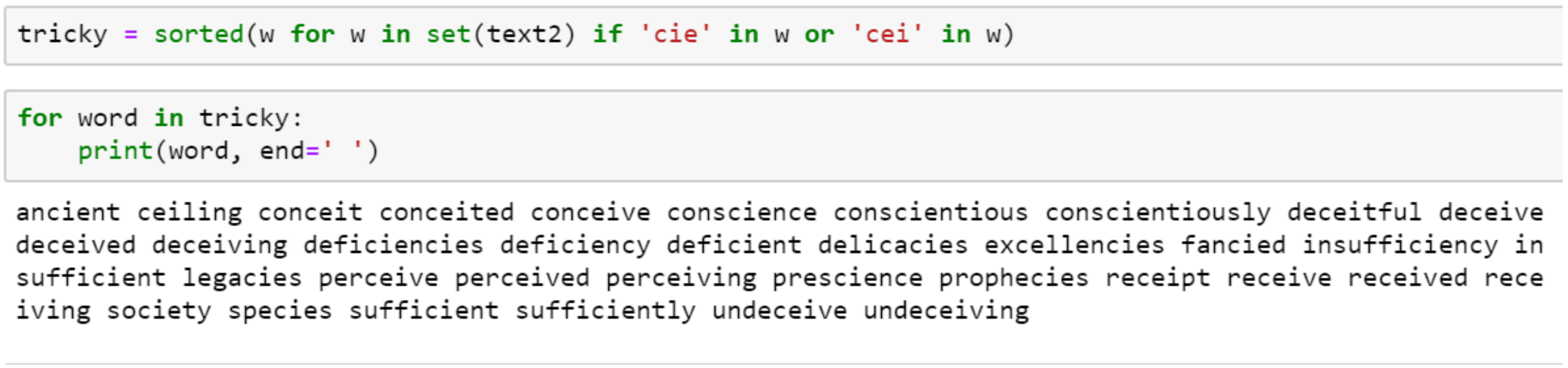


```
for token in sent1:
    if token.islower():
        print(token,'is a lowercase word')
    elif token.istitle():
        print(token,'is a titlecase word')
    else:
        print(token, 'is punctuation')
```

```
Call is a titlecase word
me is a lowercase word
Ishmael is a titlecase word
. is punctuation
```

As you can see, even with this small amount of Python knowledge, you can start to build multiline Python programs. It's important to develop such programs in pieces, testing that each piece does what you expect before combining them into a program.

Finally, let's combine the idioms we've been exploring. First, we create a list of cie and cei words, then we loop over each item and print it. Notice the extra information given in the print statement: end=' '. This tells Python to print a space (not the default newline) after each word.

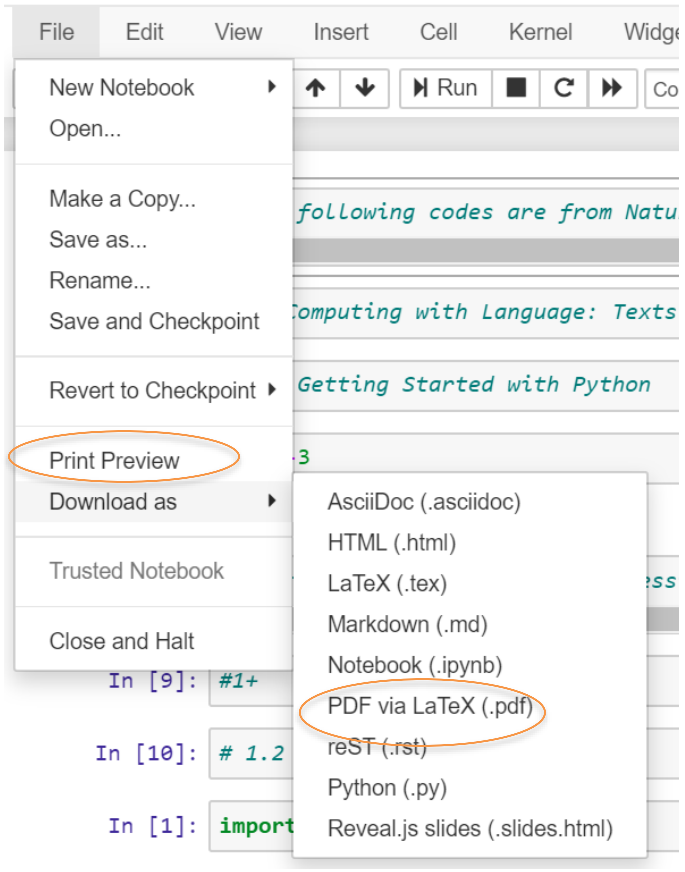


```
tricky = sorted(w for w in set(text2) if 'cie' in w or 'cei' in w)
```

```
for word in tricky:
    print(word, end=' ')
```

```
ancient ceiling conceit conceited conceive conscience conscientious conscientiously deceitful deceive
deceived deceiving deficiencies deficiency deficient delicacies excellencies fancied insufficiency in
sufficient legacies perceive perceived perceiving prescience prophecies receipt receive received rece
iving society species sufficient sufficiently undeceive undeceiving
```

Save your lab report and assignments



```
following codes are from Natu
omputing with Language: Texts
Getting Started with Python
In [9]: #1+
In [10]: # 1.2
In [1]: import
```

Option 1 (recommended):
Print preview→ print as "pdf"

Option 2:
Download as PDF and upload .pdf file to BB
If you have installed Latex

Useful Code Summary

- **Section 1:**

- len(text3): 44764 : the number of words and punctuation symbols- "tokens"-"a sequence of characters"
- len(set(text3): 2789: distinct words-"word types"-"a unique item of vocabulary-include punctuation symbols-"types"
- len(set(text3))/len(text3):2789/44764=6% lexical diversity
- text3.count("smote"): count how often a word occurs in a text
- 100*text4.count("a")/len(text4): compute what percentage of the text is taken up by a specific word (e.g. "a")

Section 2

- len(sent1)
- len(set(sent1))
- sent1.append("word1")
- text4[173]: 'awaken'
- text4.index('awaken'):173
- text5[161:167]: access sublist
- by convention, m:n means elements m... n-1
- indexing and slicing

Section 3

- abc23 is fine
- 23abc will cause a syntax error
- variable names cannot contain whitespace, can not use "-" but use _

Section 4

- slicing and indexing
- ‘ ’.join(['Monty','Python']): join the words of a list to make a single string
- 'Monty Python'.split( ): split a string into a list

Section 4

- fdist=FreqDist(text1)
- fdist
- fdist.most_common(50)
- fdist['whale']
- fdist.plt(50,cumulative=True)
- fdist.hapaxes( ): words that occur once only
- both frequent and infrequent words are useless, we need to use fine-grained selection of words
- list comprehension
- fdist5=FreqDist(text5)
- sorted（w for w in set(text5) if len(w)>7 and fdist5[w]>7)： find out the frequently-occurring content-bearing words of the text

- collocations are essentially just frequent bigrams
- print(": ".join(text4.collocation_list())

- fdist=FreqDist(len(w) for w in text1)
- fdist.most_common()
- [w for w in sent7 if len(w)<4]

- # some word comparison operators
- s.startswith(t)
- s.endswith(t)
- t in s
- s.islower()
- s.isupper()
- s.isalpha(): get rid of digits and punctuations
- s.isalnum()
- s.isdigit()
- s.istitle()

- [word.upper( ) for word in text1]
- **(word.lower( ) for word in text1]: get rid of words only differ in capitalization**
- **[word.lower( ) for word in text1 if word.isapha()]: get rid of numbers and punctuation and words that only differ in capitalization**

Thank You

Questions or Comments?

Assignments

- Lab Report 1 Due Sep 8
- Project Team Formation Due Sep 7