

# Jupyter Notebook for Week 3

## NLP with Python Chapter 1, sections 1-4

- Student Name: Maryam Agalarova
- Date: October 8, 2020
- Assignment Due: September 10, 2020
- Instructor: Lisa Rhody
- Methods of Text Analysis, Fall 2020

## Natural Language Processing with Python

**NOTE from Lisa:** During this course, we'll be working through significant portions of the open access book *Natural Language Processing with Python*, by Steven Bird, Ewan Klein, and Edward Loper, first published by O'Reilly. However, we'll be following along with the online version, which is regularly updated and can be found at <http://nltk.org/book> (<http://nltk.org/book>). You may wish to begin by reading the Preface to the book (<https://www.nltk.org/book/ch00.html> (<https://www.nltk.org/book/ch00.html>)), which explains some of the mechanics for reading and following along with activities. You will read the definitional distinctions between **natural** and **artificial** language.

While these notebooks will offer the code that you will need to work through basic exercises, I welcome, and even encourage, you taking this opportunity to also work with some of the other exercises that the book provides which are not fully included here.

While the book is designed to be used while writing directly into the Python interpreter, I'm asking for you to complete these assignments in Jupyter notebooks, which allows me to combine the copy of the text with the cells where you can type and execute code.

## Chapter 1: Language Processing and Python

It is easy to get our hands on millions of words of text. What can we do with it, assuming we can write some simple programs? In this chapter we'll address the following questions:

- 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?
- What are some of the interesting challenges of natural language processing?

This chapter is divided into sections that skip between two quite different styles. In the "computing with language" sections we will take on some linguistically motivated programming tasks without necessarily explaining how they work. In the "closer look at Python" sections we will systematically review key programming concepts. We'll flag the two styles in the section titles, but later chapters will mix both styles without being so up-front about it. We hope this style of introduction gives you an authentic taste of what will come later, while covering a range of elementary concepts in linguistics and computer science. If you have basic familiarity with both areas, you can skip to 5; we will repeat any important points in later chapters, and if you miss anything you can easily consult the online reference material at <http://nltk.org/> (<http://nltk.org/>). If the material is completely new to you, this chapter will raise more questions than it answers, questions that are addressed in the rest of this book.

### 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.

**NOTE from Lisa:** Because we are working in Jupyter notebooks and not in the **interpreter** (which is defined below), your interface will look a little bit different. That's ok.

## 1.1 Getting Started with Python

One of the friendly things about Python is that it allows you to type directly into the interactive **interpreter** — the program that will be running your Python programs. You can access the Python interpreter using a simple graphical interface called the Interactive DeveLopment Environment (IDLE).

On a Mac you can find this under Applications→MacPython, and on Windows under All Programs→Python. Under Unix you can run Python from the shell by typing `idle` (if this is not installed, try typing `python`). The interpreter will print a blurb about your Python version; simply check that you are running Python 3.2 or later (here it is for 3.4.2):

```
Python 3.4.2 (default, Oct 15 2014, 22:01:37)
[GCC 4.2.1 Compatible Apple LLVM 5.1 (clang-503.0.40)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

### Note

If you are unable to run the Python interpreter, you probably don't have Python installed correctly. Please visit <http://python.org/> for detailed instructions. NLTK 3.0 works for Python 2.6 and 2.7. If you are using one of these older versions, note that the `/` operator rounds fractional results downwards (so `1/3` will give you `0`). In order to get the expected behavior of division you need to type: `from __future__ import division`

The `>>>` prompt indicates that the Python interpreter is now waiting for input. When copying examples from this book, don't type the `>>>` yourself. Now, let's begin by using Python as a calculator:

```
>>> 1 + 5 * 2 - 3
8
>>>
```

Once the interpreter has finished calculating the answer and displaying it, the prompt reappears. This means the Python interpreter is waiting for another instruction.

**NOTE from Lisa:** The Jupyter notebook works differently than the interpreter. Rather than having a `>>>>` you will see:

In [ ]:

That means you can type the equation directly into the cell. When you press either the `>|` button above or you press the `shift + enter` keys, you will "execute" the cell, which means that it will run, and your cursor will go to the next cell.

In the cell below, type the following:

```
1 + 5 * 2 - 3
```

Then press the `shift + enter` (or `return`) key. The expected action should be that the answer appears below the cell next to the word `OUT [ ]`:

```
In [1]: #type your equation here.  
1 + 5 * 2 - 3
```

```
Out[1]: 8
```

The preceding examples demonstrate how you can work interactively with the Python interpreter, experimenting with various expressions in the language to see what they do. Now let's try a nonsensical expression to see how the interpreter handles it: Try typing 1 + (without anything after the plus sign).

```
In [ ]: #type 1+ and press
```

This produced a syntax error. In Python, it doesn't make sense to end an instruction with a plus sign. The Python interpreter indicates the line where the problem occurred (line 1 of , which stands for "standard input").

Now that we can use the Python interpreter, we're ready to start working with language data.

## 1.2 Getting Started with NLTK

**NOTE from Lisa:** This portion of the chapter is about downloading and installing the text corpora that we are using first. You completed this activity during the Week 2 Jupyter notebook assignment, so I am going to skip the portion of the lesson here, because downloading and installing NLTK's corpora is time and resource intensive for your computer. So the next few cells will deviate from the book a bit. If you would like to re-do this part or if the following section does not work for you, you will want to revisit this section of NLP with Python book (<https://www.nltk.org/book/ch01.html> (<https://www.nltk.org/book/ch01.html>)).

We will pick up at the point in which we import the corpora (different from downloading, importing means that we are making those resources available to Python for use).

When we get started, we need to let Python know that we are working with certain resources. These resources can be called "libraries" or "packages." We are also able to make particular parts of a package available. We'll do this by saying that from NLTK, we want to import into existing memory all of the texts that are in the "book" portion of NLTK. We do that with the following:

```
In [2]: import nltk
        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
```

You have now told Python that you want to have NLTK ready and from NLTK's book corpora, you want to install everything (the \* is used as a wild card in many computer programs). So, the correct answer should look like a list of titles. They begin text1:, text2:, etc. In other words, rather than having to type "Moby Dick by Herman Melville 1951" every time you want to use it, NLTK has given it a shortened name: text1. If you give Python this name, it will return to you the title of the text. Try that in the next cell.

```
In [102]: text1
```

```
Out[102]: <Text: Moby Dick by Herman Melville 1851>
```

```
In [103]: text2
```

```
Out[103]: <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 [104]: text1.concordance("monstrous")
```

```
Displaying 11 of 11 matches:
ong the former , one was of a most monstrous size . ... This came to
wards us ,
ON OF THE PSALMS . " Touching that monstrous bulk of the whale or or
k 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 cou
ld 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 a
re to be fo
ght have been rummaged out of this monstrous cabinet there is no tel
ling . But
of Whale - Bones ; for Whales of a monstrous size are oftentimes cas
t up dead u
```

**Your Turn:** Try searching for other words; to save re-typing, you might be able to use up-arrow, Ctrl-up-arrow or Alt-p to access the previous command and modify the word being searched. You can also try searches on some of the other texts we have included. For example, search Sense and Sensibility for the word affection, using

```
text2.concordance("affection")
```

Search the book of Genesis to find out how long some people lived, using

```
text3.concordance("lived")
```

You could look at text4, the Inaugural Address Corpus, to see examples of English going back to 1789, and search for words like nation, terror, god to see how these words have been used differently over time. We've also included text5, the NPS Chat Corpus: search this for unconventional words like im, ur, lol. (Note that this corpus is uncensored!)

```
In [4]: text3.concordance("lived")
```

```
Displaying 25 of 38 matches:
ay when they were created . And Adam lived an hundred and thirty yea
rs , and be
```

ughters : And all the days that Adam lived were nine hundred and thi  
 rty 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  
 twelve years : and he died . And Enos lived ninety years , and begat  
 Cainan : An  
 years , and begat Cainan : And Enos lived after he begat Cainan eig  
 ht hundred  
 ive years : and he died . And Cainan lived seventy years and begat M  
 ahalaleel :  
 rs and begat Mahalaleel : And Cainan lived after he begat Mahalaleel  
 eight hund  
 years : and he died . And Mahalaleel lived sixty and five years , an  
 d begat Jar  
 s , and begat Jared : And Mahalaleel lived after he begat Jared eigh  
 t 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 eigh  
 t hundred y  
 and two yea and he died . And Enoch lived sixty and five years , an  
 d begat Met  
 ; for God took him . And Methuselah lived an hundred eighty and sev  
 en years ,  
 , and begat Lamech . And Methuselah lived after he begat Lamech sev  
 en 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 hun  
 dred and fi  
 xad two years after the flo And Shem lived after he begat Arphaxad f  
 ive hundred  
 at sons and daughters . And Arphaxad lived five and thirty years , a  
 nd 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 , a  
 nd begat Pe  
 y years , and begat Peleg : And Eber lived after he begat Peleg four  
 hundred an

```
In [3]: text2.concordance("affection")
```

Displaying 25 of 79 matches:

, however , and , as a mark of his affection for the three girls , h  
e left them  
t . It was very well known that no affection was ever supposed to ex  
ist between  
deration of politeness or maternal affection on the side of the form  
er , the tw  
d the suspicion -- the hope of his affection for me may warrant , wi  
thout impru  
hich forbade the indulgence of his affection . She knew that his mot  
her neither  
rd she gave one with still greater affection . Though her late conve  
rsation wit  
can never hope to feel or inspire affection again , and if her home  
be uncomfo  
m of the sense , elegance , mutual affection , and domestic comfort  
of the fami  
, and which recommended him to her affection beyond every thing else  
. His soci  
ween the parties might forward the affection of Mr . Willoughby , an  
equally st  
the most pointed assurance of her affection . Elinor could not be s  
urprised at  
he natural consequence of a strong affection in a young and ardent m  
ind . This  
opinion . But by an appeal to her affection for her mother , by rep  
resenting t  
every alteration of a place which affection had established as perf  
ect with hi  
e will always have one claim of my affection , which no other can po  
ssibly shar  
f the evening declared at once his affection and happiness . " Shall  
we see you  
ause he took leave of us with less affection than his usual behaviou  
r has shewn  
ness ." " I want no proof of their affection ," said Elinor ; " but  
of their en  
onths , without telling her of his affection ;-- that they should pa  
rt without  
ould be the natural result of your affection for her . She used to b  
e all unres  
distinguished Elinor by no mark of affection . Marianne saw and list  
ened with i  
th no inclination for expense , no affection for strangers , no prof  
ession , an  
till distinguished her by the same affection which once she had felt  
no doubt o



al of her confidence in Edward ' s affection , to the remembrance of every mark  
 was made ? Had he never owned his affection to yourself ?" " Oh , n  
 o ; but if

```
In [106]: text2.concordance("feminine")
```

```
Displaying 1 of 1 matches:
ng ?-- delicate -- tender -- truly feminine -- was it not ?" " Your
wife !-- T
```

Once you've spent a little while examining these texts, we hope you have a new sense of the richness and diversity of language. In the next chapter you will learn how to access a broader range of text, including text in languages other than English.

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 [5]: text1.similar("monstrous")
```

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

```
In [6]: 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 uses this word quite differently from Melville; for her, *monstrous* has positive connotations, and sometimes functions as an intensifier like the word *very*.

The term `common_contexts` allows us to examine just the contexts that are shared by two or more words, such as *monstrous* and *very*. We have to enclose these words by square brackets as well as parentheses, and separate them with a comma:

```
In [7]: text2.common_contexts(["monstrous", "very"])
```

```
a_pretty am_glad a_lucky is_pretty be_glad
```

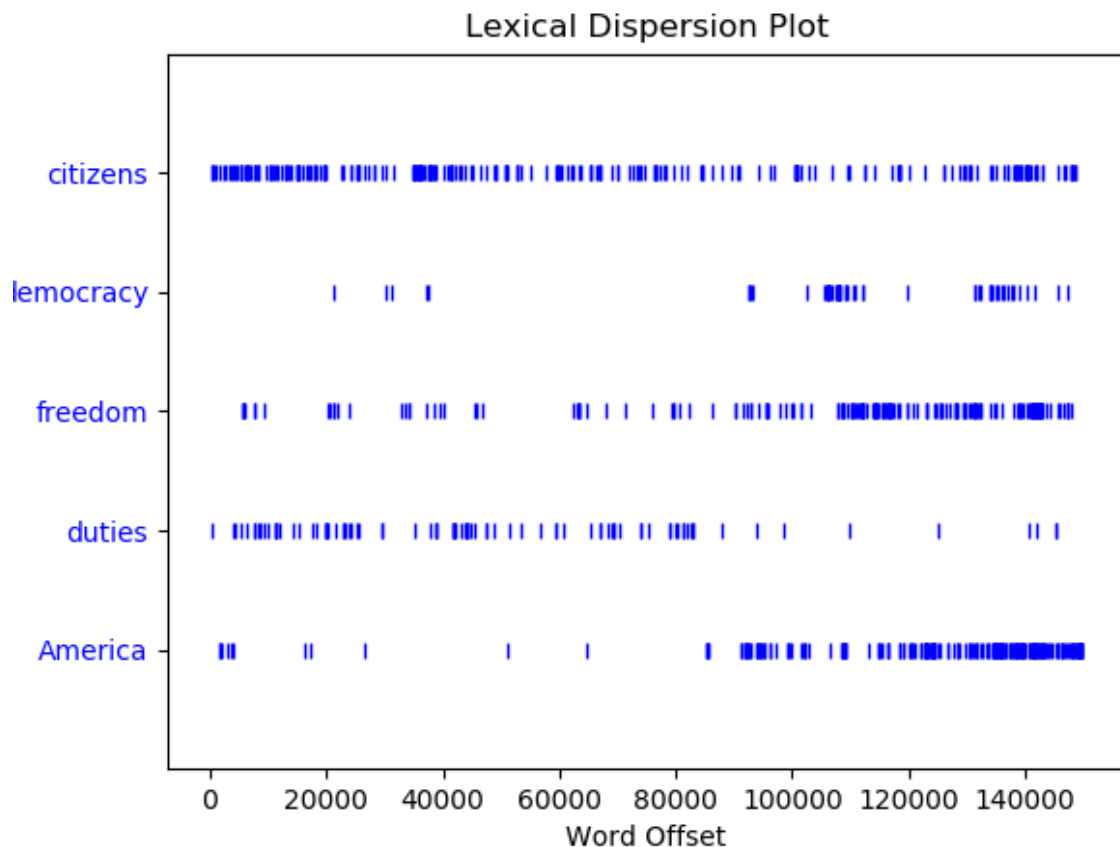
**Your turn:** pick another pair of words and compare their usage in two different texts, using the `similar()` and `common_contexts()` functions.

```
In [11]: text1.common_contexts(["best", "great"])
```

```
the_whales the_and
```

```
In [14]: %matplotlib notebook
```

```
In [15]: text4.dispersion_plot(["citizens", "democracy", "freedom", "duties", "America"])
```



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 (<https://www.nltk.org/book/ch01.html#fig-inaugural>) 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.

**Important Note:** You need to have Python's NumPy and Matplotlib packages installed in order to produce the graphical plots used in this book. If you have already installed Anaconda, these should already be on your computer.

**Note** You can also plot the frequency of word usage through time using <https://books.google.com/ngrams> (<https://books.google.com/ngrams>). Which is to say that *this* is the function that Google's N-grams viewer is making use of.

**Also note** There is an exercise in the <http://nltk.org> (<http://nltk.org>) book that uses a function to generate random text based on a book from the corpus. This feature no longer works in Python 3.0 or above, but is expected to be added back in eventually.

## 1.4 Counting Vocabulary

The most obvious fact about texts that emerges from the preceding examples is that they differ in the vocabulary they use. In this section we will see how to use the computer to count the words in a text in a variety of useful ways. As before, you will jump right in and experiment with the Python interpreter, even though you may not have studied Python systematically yet. Test your understanding by modifying the examples, and trying the exercises at the end of the chapter.

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:

```
In [16]: len(text3)
```

```
Out[16]: 44764
```

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. When we count the number of tokens in a text, say, the phrase *to be or not to be*, we are counting occurrences of these sequences. Thus, in our example phrase there are two occurrences of *to*, two of *be*, and one each of *or* and *not*. But there are only four distinct vocabulary items in this phrase. How many distinct words does the book of Genesis contain? To work this out in Python, we have to pose the question slightly differently. The vocabulary of a text is just the set of tokens that it uses, since in a set, all duplicates are collapsed together. In Python we can obtain the vocabulary items of `text3` with the command: `set(text3)`. When you do this, many screens of words will fly past. Now try the following:

```
In [17]: sorted(set(text3))
```

```
Out[17]: ['!',
          '"',
          '(',
          ')',
          ',',
          ')',
          '.',
          ')',
          ':',
          ';',
          ';)',
          '?',
          '?)',
          'A',
          'Abel',
          'Abelmizraim',
          'Abidah',
          'Abide',
          'Abimael',
          'Abimelech',
          'Abr',
          'Abrah',
          'Abraham',
          'Abram',
          'Accad',
          'Achbor',
          'Adah',
          'Adam',
          'Adbeel',
          'Admah',
          'Adullamite',
          'After',
          'Aholibamah',
          'Ahuzzath',
          'Ajah',
```

'Akan',  
'All',  
'Allonbachuth',  
'Almighty',  
'Almodad',  
'Also',  
'Alvah',  
'Alvan',  
'Am',  
'Amal',  
'Amalek',  
'Amalekites',  
'Ammon',  
'Amorite',  
'Amorites',  
'Amraphel',  
'An',  
'Anah',  
'Anamim',  
'And',  
'Aner',  
'Angel',  
'Appoint',  
'Aram',  
'Aran',  
'Ararat',  
'Arbah',  
'Ard',  
'Are',  
'Areli',  
'Arioch',  
'Arise',  
'Arkite',  
'Arodi',  
'Arphaxad',  
'Art',  
'Arvadite',  
'As',  
'Asenath',  
'Ashbel',  
'Asher',  
'Ashkenaz',  
'Ashteroth',  
'Ask',  
'Asshur',  
'Asshurim',  
'Assyr',  
'Assyria',  
'At',  
'Atad',

'Avith',  
'Baalhanan',  
'Babel',  
'Bashemath',  
'Be',  
'Because',  
'Becher',  
'Bedad',  
'Beeri',  
'Beerlahairoi',  
'Beersheba',  
'Behold',  
'Bela',  
'Belah',  
'Benam',  
'Benjamin',  
'Beno',  
'Beor',  
'Bera',  
'Bered',  
'Beriah',  
'Bethel',  
'Bethlehem',  
'Bethuel',  
'Beware',  
'Bilhah',  
'Bilhan',  
'Binding',  
'Birsha',  
'Bless',  
'Blessed',  
'Both',  
'Bow',  
'Bozrah',  
'Bring',  
'But',  
'Buz',  
'By',  
'Cain',  
'Cainan',  
'Calah',  
'Calneh',  
'Can',  
'Cana',  
'Canaan',  
'Canaanite',  
'Canaanites',  
'Canaanitish',  
'Caphtorim',  
'Carmi',

'Casluhim',  
'Cast',  
'Cause',  
'Chaldees',  
'Chedorlaomer',  
'Cheran',  
'Cherubims',  
'Chesed',  
'Chezib',  
'Come',  
'Cursed',  
'Cush',  
'Damascus',  
'Dan',  
'Day',  
'Deborah',  
'Dedan',  
'Deliver',  
'Diklah',  
'Din',  
'Dinah',  
'Dinhabah',  
'Discern',  
'Dishan',  
'Dishon',  
'Do',  
'Dodanim',  
'Dothan',  
'Drink',  
'Duke',  
'Dumah',  
'Earth',  
'Ebal',  
'Eber',  
'Edar',  
'Eden',  
'Edom',  
'Edomites',  
'Egy',  
'Egypt',  
'Egyptia',  
'Egyptian',  
'Egyptians',  
'Ehi',  
'Elah',  
'Elam',  
'Elbethel',  
'Eldaah',  
'EleloheIsrael',  
'Eliezer',

'Eliphaz',  
'Elishah',  
'Ellasar',  
'Elon',  
'Elparan',  
'Emins',  
'En',  
'Enmishpat',  
'Eno',  
'Enoch',  
'Enos',  
'Ephah',  
'Epher',  
'Ephra',  
'Ephraim',  
'Ephrath',  
'Ephron',  
'Er',  
'Erech',  
'Eri',  
'Es',  
'Esau',  
'Escape',  
'Esek',  
'Eshban',  
'Eshcol',  
'Ethiopia',  
'Euphrat',  
'Euphrates',  
'Eve',  
'Even',  
'Every',  
'Except',  
'Ezbon',  
'Ezer',  
'Fear',  
'Feed',  
'Fifteen',  
'Fill',  
'For',  
'Forasmuch',  
'Forgive',  
'From',  
'Fulfil',  
'G',  
'Gad',  
'Gaham',  
'Galeed',  
'Gatam',  
'Gather',



'Gaza',  
'Gentiles',  
'Gera',  
'Gerar',  
'Gershon',  
'Get',  
'Gether',  
'Gihon',  
'Gilead',  
'Girgashites',  
'Girgasite',  
'Give',  
'Go',  
'God',  
'Gomer',  
'Gomorra',  
'Goshen',  
'Guni',  
'Hadad',  
'Hadar',  
'Hadoram',  
'Hagar',  
'Haggi',  
'Hai',  
'Ham',  
'Hamathite',  
'Hamor',  
'Hamul',  
'Hanoah',  
'Happy',  
'Haran',  
'Hast',  
'Haste',  
'Have',  
'Havilah',  
'Hazarmaveth',  
'Hazeontamar',  
'Hazo',  
'He',  
'Hear',  
'Heaven',  
'Heber',  
'Hebrew',  
'Hebrews',  
'Hebron',  
'Hemam',  
'Hemdan',  
'Here',  
'Hereby',  
'Heth',

'Hezron',  
'Hiddekel',  
'Hinder',  
'Hirah',  
'His',  
'Hitti',  
'Hittite',  
'Hittites',  
'Hivite',  
'Hobah',  
'Hori',  
'Horite',  
'Horites',  
'How',  
'Hul',  
'Huppim',  
'Husham',  
'Hushim',  
'Huz',  
'I',  
'If',  
'In',  
'Irad',  
'Iram',  
'Is',  
'Isa',  
'Isaac',  
'Iscah',  
'Ishbak',  
'Ishmael',  
'Ishmeelites',  
'Ishuah',  
'Isra',  
'Israel',  
'Issachar',  
'Isui',  
'It',  
'Ithran',  
'Jaalam',  
'Jabal',  
'Jabbok',  
'Jac',  
'Jachin',  
'Jacob',  
'Jahleel',  
'Jahzeel',  
'Jamin',  
'Japhe',  
'Japheth',  
'Jared',

'Javan',  
'Jebusite',  
'Jebusites',  
'Jegarsahadutha',  
'Jehovahjireh',  
'Jemuel',  
'Jerah',  
'Jetheth',  
'Jetur',  
'Jeush',  
'Jezer',  
'Jidlaph',  
'Jimnah',  
'Job',  
'Jobab',  
'Jokshan',  
'Joktan',  
'Jordan',  
'Joseph',  
'Jubal',  
'Judah',  
'Judge',  
'Judith',  
'Kadesh',  
'Kadmonites',  
'Karnaim',  
'Kedar',  
'Kedemah',  
'Kemuel',  
'Kenaz',  
'Kenites',  
'Kenizzites',  
'Keturah',  
'Kiriathaim',  
'Kirjatharba',  
'Kittim',  
'Know',  
'Kohath',  
'Kor',  
'Korah',  
'LO',  
'LORD',  
'Laban',  
'Lahairoi',  
'Lamech',  
'Lasha',  
'Lay',  
'Leah',  
'Lehabim',  
'Lest',

'Let',  
'Letushim',  
'Leummim',  
'Levi',  
'Lie',  
'Lift',  
'Lo',  
'Look',  
'Lot',  
'Lotan',  
'Lud',  
'Ludim',  
'Luz',  
'Maachah',  
'Machir',  
'Machpelah',  
'Madai',  
'Magdiel',  
'Magog',  
'Mahalaleel',  
'Mahalath',  
'Mahanaim',  
'Make',  
'Malchiel',  
'Male',  
'Mam',  
'Mamre',  
'Man',  
'Manahath',  
'Manass',  
'Manasseh',  
'Mash',  
'Masrekah',  
'Massa',  
'Matred',  
'Me',  
'Medan',  
'Mehetabel',  
'Mehujael',  
'Melchizedek',  
'Merari',  
'Mesha',  
'Meshech',  
'Mesopotamia',  
'Methusa',  
'Methusael',  
'Methuselah',  
'Mezahab',  
'Mibsam',  
'Mibzar',

'Midian',  
'Midianites',  
'Milcah',  
'Mishma',  
'Mizpah',  
'Mizraim',  
'Mizz',  
'Moab',  
'Moabites',  
'Moreh',  
'Moreover',  
'Moriah',  
'Muppim',  
'My',  
'Naamah',  
'Naaman',  
'Nahath',  
'Nahor',  
'Naphish',  
'Naphtali',  
'Naphtuhim',  
'Nay',  
'Nebajoth',  
'Neither',  
'Night',  
'Nimrod',  
'Nineveh',  
'Noah',  
'Nod',  
'Not',  
'Now',  
'O',  
'Obal',  
'Of',  
'Oh',  
'Ohad',  
'Omar',  
'On',  
'Onam',  
'Onan',  
'Only',  
'Ophir',  
'Our',  
'Out',  
'Padan',  
'Padanaram',  
'Paran',  
'Pass',  
'Pathrusim',  
'Pau',

'Peace',  
'Peleg',  
'Peniel',  
'Penuel',  
'Peradventure',  
'Perizzit',  
'Perizzite',  
'Perizzites',  
'Phallu',  
'Phara',  
'Pharaoh',  
'Pharez',  
'Phichol',  
'Philistim',  
'Philistines',  
'Phut',  
'Phuvah',  
'Pildash',  
'Pinon',  
'Pison',  
'Potiphar',  
'Potipherah',  
'Put',  
'Raamah',  
'Rachel',  
'Rameses',  
'Rebek',  
'Rebekah',  
'Rehoboth',  
'Remain',  
'Rephaims',  
'Resen',  
'Return',  
'Reu',  
'Reub',  
'Reuben',  
'Reuel',  
'Reumah',  
'Riphath',  
'Rosh',  
'Sabtah',  
'Sabtech',  
'Said',  
'Salah',  
'Salem',  
'Samlah',  
'Sarah',  
'Sarai',  
'Saul',  
'Save',

'Say',  
'Se',  
'Seba',  
'See',  
'Seeing',  
'Seir',  
'Sell',  
'Send',  
'Sephah',  
'Serah',  
'Sered',  
'Serug',  
'Set',  
'Seth',  
'Shalem',  
'Shall',  
'Shalt',  
'Shammah',  
'Shaul',  
'Shaveh',  
'She',  
'Sheba',  
'Shebah',  
'Shechem',  
'Shed',  
'Shel',  
'Shelah',  
'Sheleph',  
'Shem',  
'Shemeber',  
'Shepho',  
'Shillem',  
'Shiloh',  
'Shimron',  
'Shinab',  
'Shinar',  
'Shobal',  
'Should',  
'Shuah',  
'Shuni',  
'Shur',  
'Sichem',  
'Siddim',  
'Sidon',  
'Simeon',  
'Sinite',  
'Sitnah',  
'Slay',  
'So',  
'Sod',

'Sodom',  
'Sojourn',  
'Some',  
'Spake',  
'Speak',  
'Spirit',  
'Stand',  
'Succoth',  
'Surely',  
'Swear',  
'Syrian',  
'Take',  
'Tamar',  
'Tarshish',  
'Tebah',  
'Tell',  
'Tema',  
'Teman',  
'Temani',  
'Terah',  
'Thahash',  
'That',  
'The',  
'Then',  
'There',  
'Therefore',  
'These',  
'They',  
'Thirty',  
'This',  
'Thorns',  
'Thou',  
'Thus',  
'Thy',  
'Tidal',  
'Timna',  
'Timnah',  
'Timnath',  
'Tiras',  
'To',  
'Togarmah',  
'Tola',  
'Tubal',  
'Tubalcain',  
'Twelve',  
'Two',  
'Unstable',  
'Until',  
'Unto',  
'Up',



'Upon',  
'Ur',  
'Uz',  
'Uzal',  
'We',  
'What',  
'When',  
'Whence',  
'Where',  
'Whereas',  
'Wherefore',  
'Which',  
'While',  
'Who',  
'Whose',  
'Whoso',  
'Why',  
'Wilt',  
'With',  
'Woman',  
'Ye',  
'Yea',  
'Yet',  
'Zaavan',  
'Zaphnathpaaneah',  
'Zar',  
'Zarah',  
'Zeboiim',  
'Zeboim',  
'Zebul',  
'Zebulun',  
'Zemarite',  
'Zepho',  
'Zerah',  
'Zibeon',  
'Zidon',  
'Zillah',  
'Zilpah',  
'Zimran',  
'Ziphion',  
'Zo',  
'Zoar',  
'Zohar',  
'Zuzims',  
'a',  
'abated',  
'abide',  
'able',  
'abode',  
'abomination',

'about',  
'above',  
'abroad',  
'absent',  
'abundantly',  
'accept',  
'accepted',  
'according',  
'acknowledged',  
'activity',  
'add',  
'adder',  
'afar',  
'afflict',  
'affliction',  
'afraid',  
'after',  
'afterward',  
'afterwards',  
'aga',  
'again',  
'against',  
'age',  
'aileth',  
'air',  
'al',  
'alive',  
'all',  
'almon',  
'alo',  
'alone',  
'aloud',  
'also',  
'altar',  
'altogether',  
'always',  
'am',  
'among',  
'amongst',  
'an',  
'and',  
'angel',  
'angels',  
'anger',  
'angry',  
'anguish',  
'anointedst',  
'anoth',  
'another',  
'answer',

'answered',  
'any',  
'anything',  
'appe',  
'appear',  
'appeared',  
'appease',  
'appoint',  
'appointed',  
'aprons',  
'archer',  
'archers',  
'are',  
'arise',  
'ark',  
'armed',  
'arms',  
'army',  
'arose',  
'arrayed',  
'art',  
'artificer',  
'as',  
'ascending',  
'ash',  
'ashamed',  
'ask',  
'asked',  
'asketh',  
'ass',  
'assembly',  
'asses',  
'assigned',  
'asswaged',  
'at',  
'attained',  
'audience',  
'avenged',  
'aw',  
'awaked',  
'away',  
'awoke',  
'back',  
'backward',  
'bad',  
'bade',  
'badest',  
'badne',  
'bak',  
'bake',

'bakemeats',  
'baker',  
'bakers',  
'balm',  
'bands',  
'bank',  
'bare',  
'barr',  
'barren',  
'basket',  
'baskets',  
'battle',  
'bdellium',  
'be',  
'bear',  
'beari',  
'bearing',  
'beast',  
'beasts',  
'beautiful',  
'became',  
'because',  
'become',  
'bed',  
'been',  
'befall',  
'befell',  
'before',  
'began',  
'begat',  
'beget',  
'begettest',  
'begin',  
'beginning',  
'begotten',  
'beguiled',  
'beheld',  
'behind',  
'behold',  
'being',  
'believed',  
'belly',  
'belong',  
'beneath',  
'bereaved',  
'beside',  
'besides',  
'besought',  
'best',  
'betimes',

'better',  
'between',  
'betwixt',  
'beyond',  
'binding',  
'bird',  
'birds',  
'birthday',  
'birthright',  
'biteth',  
'bitter',  
'blame',  
'blameless',  
'blasted',  
'bless',  
'blessed',  
'blesseth',  
'blessi',  
'blessing',  
'blessings',  
'blindness',  
'blood',  
'blossoms',  
'bodies',  
'boldly',  
'bondman',  
'bondmen',  
'bondwoman',  
'bone',  
'bones',  
'book',  
'booths',  
'border',  
'borders',  
'born',  
'bosom',  
'both',  
'bottle',  
'bou',  
'boug',  
'bough',  
'bought',  
'bound',  
'bow',  
'bowed',  
'bowels',  
'bowing',  
'boys',  
'bracelets',  
'branches',

'brass',  
'bre',  
'breach',  
'bread',  
'breadth',  
'break',  
'breaketh',  
'breaking',  
'breasts',  
'breath',  
'breathed',  
'breed',  
'brethren',  
'brick',  
'brimstone',  
'bring',  
'brink',  
'broken',  
'brook',  
'broth',  
'brother',  
'brought',  
'brown',  
'bruise',  
'budded',  
'build',  
'buildied',  
'built',  
'bulls',  
'bundle',  
'bundles',  
'burdens',  
'buried',  
'burn',  
'burning',  
'burnt',  
'bury',  
'buryingplace',  
'business',  
'but',  
'butler',  
'butlers',  
'butlership',  
'butter',  
'buy',  
'by',  
'cakes',  
'calf',  
'call',  
'called',

'came',  
'camel',  
'camels',  
'camest',  
'can',  
'cannot',  
'canst',  
'captain',  
'captive',  
'captives',  
'carcases',  
'carried',  
'carry',  
'cast',  
'castles',  
'catt',  
'cattle',  
'caught',  
'cause',  
'caused',  
'cave',  
'cease',  
'ceased',  
'certain',  
'certainly',  
'chain',  
'chamber',  
'change',  
'changed',  
'changes',  
'charge',  
'charged',  
'chariot',  
'chariots',  
'chesnut',  
'chi',  
'chief',  
'child',  
'childless',  
'childr',  
'children',  
'chode',  
'choice',  
'chose',  
'circumcis',  
'circumcise',  
'circumcised',  
'citi',  
'cities',  
'city',

```
'clave',  
'clean',  
'clear',  
'cleave',  
'clo',  
'closed',  
'clothed',  
'clothes',  
'cloud',  
'clusters',  
'co',  
'coat',  
'coats',  
'coffin',  
'cold',  
...]
```

```
In [ ]: len(set(text3))
```

By wrapping `sorted()` around the Python expression `set(text3)`, 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. 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. 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.

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 (remember if you're using Python 2, to start with `from __future__ import division`).

```
In [18]: len(set(text3)) / len(text3)
```

```
Out[18]: 0.06230453042623537
```

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:

```
In [19]: text3.count("smote")
```

```
Out[19]: 5
```



```
In [20]: 100 * text4.count('a') / len(text4)
```

```
Out[20]: 1.457973123627309
```

**Exercise:** 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? [Use the first cell to count the instances of "lol" and the second cell to calculate the percentage of the entire text.]

```
In [21]: text5.count("lol")
```

```
Out[21]: 704
```

```
In [22]: 100 * text5.count('lol') / len(text5)
```

```
Out[22]: 1.5640968673628082
```

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. Now you only have to type a short name instead of one or more complete lines of Python code, and you can re-use it as often as you like. The block of code that does a task for us is called a **function**, and we define a short name for our function with the keyword `def`. The next example shows how to define two new functions, `lexical_diversity()` and `percentage()`:

```
In [23]: def lexical_diversity(text):  
         return len(set(text)) / len(text)
```

```
In [24]: def percentage(count, total):  
         return 100 * count / total
```

Notice that once you type the first row of the function, the cursor automatically inserts a tab. That's because the Python interpreter knows to expect an **indented code block**. If you were working in the actual Python interpreter, you would need to remember to insert that tab yourself either with four space bars or the tab key. To finish the indented block, just press enter/return.

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:

```
In [25]: lexical_diversity(text3)
```

```
Out[25]: 0.06230453042623537
```

```
In [26]: lexical_diversity(text5)
```

```
Out[26]: 0.13477005109975562
```

```
In [27]: percentage(4, 5)
```

```
Out[27]: 80.0
```

```
In [28]: percentage(text4.count('a'), len(text4))
```

```
Out[28]: 1.457973123627309
```

To recap, we use or *call* a function such as `lexical_diversity()` by typing its name, followed by an open parenthesis, the name of the text, and then a close parenthesis. These parentheses will show up often; their role is to separate the name of a task — such as `lexical_diversity()` — from the data that the task is to be performed on — such as `text3`. The data value that we place in the parentheses when we call a function is an **argument** to the function.

You have already encountered several functions in this chapter, such as `len()`, `set()`, and `sorted()`. By convention, we will always add an empty pair of parentheses after a function name, as in `len()`, just to make clear that what we are talking about is a function rather than some other kind of Python expression. Functions are an important concept in programming, and we only mention them at the outset to give newcomers a sense of the power and creativity of programming. Don't worry if you find it a bit confusing right now.

Later we'll see how to use functions when tabulating data, as in 1.1

(<https://www.nltk.org/book/ch01.html#tab-brown-types> (<https://www.nltk.org/book/ch01.html#tab-brown-types>)). Each row of the table will involve the same computation but with different data, and we'll do this repetitive work using a function.

## 2 A Closer Look at Python: Texts as Lists of Words

You've seen some important elements of the Python programming language. Let's take a few moments to review them systematically.

### 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*:

```
In [29]: sent1 = ['Call', 'me', 'Ishmael', '.']
```

```
In [30]: sent1
```

```
Out[30]: ['Call', 'me', 'Ishmael', '.']
```

```
In [31]: len(sent1)
```

```
Out[31]: 4
```

```
In [32]: lexical_diversity(sent1)
```

```
Out[32]: 1.0
```

Some more lists have been defined for you, one for the opening sentence of each of our texts, `sent2 ... sent9`. We inspect two of them here; you can see the rest for yourself using the Python interpreter (if you get an error which says that `sent2` is not defined, you need to first type `from nltk.book import *`).

```
In [33]: from nltk.book import *
```

```
In [34]: sent2
```

```
Out[34]: ['The',  
          'family',  
          'of',  
          'Dashwood',  
          'had',  
          'long',  
          'been',  
          'settled',  
          'in',  
          'Sussex',  
          '.']
```

```
In [35]: sent3
```

```
Out[35]: ['In',  
          'the',  
          'beginning',  
          'God',  
          'created',  
          'the',  
          'heaven',  
          'and',  
          'the',  
          'earth',  
          '.']
```

**Your Turn:** 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']`. Repeat some of the other Python operations we saw earlier in 1, e.g., `sorted(ex1)`, `len(set(ex1))`, `ex1.count('the')`.

```
In [36]: myex1 = ['same', 'mystery', 'and']
```

```
In [37]: myex1
```

```
Out[37]: ['same', 'mystery', 'and']
```

```
In [38]: sorted(myex1)
```

```
Out[38]: ['and', 'mystery', 'same']
```

A pleasant surprise is that we can use Python's addition operator on lists. Adding two lists creates a new list with everything from the first list, followed by everything from the second list:

```
In [39]: ['Monty', 'Python'] + ['and', 'the', 'Holy', 'Grail']
```

```
Out[39]: ['Monty', 'Python', 'and', 'the', 'Holy', 'Grail']
```

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.

We don't have to literally type the lists either; we can use short names that refer to pre-defined lists.

```
In [40]: sent4 + sent1
```

```
Out[40]: ['Fellow',  
          '-',  
          'Citizens',  
          'of',  
          'the',  
          'Senate',  
          'and',  
          'of',  
          'the',  
          'House',  
          'of',  
          'Representatives',  
          ':',  
          'Call',  
          'me',  
          'Ishmael',  
          '.']
```

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.

```
In [41]: sent1.append("Some")  
sent1
```

```
Out[41]: ['Call', 'me', 'Ishmael', '.', 'Some']
```

## 2.2 Indexing Lists

As we have seen, a text in Python is a list of words, represented using a combination of brackets and quotes. Just as with an ordinary page of text, we can count up the total number of words in `text1` with `len(text1)`, and count the occurrences in a text of a particular word — say, `'heaven'` — using `text1.count('heaven')`.

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:

```
In [42]: text4[173]
```

```
Out[42]: 'awaken'
```

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

```
In [43]: text4.index('awaken')
```

```
Out[43]: 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**.

```
In [44]: text5[16715:16735]
```

```
Out[44]: ['U86',  
          'thats',  
          'why',  
          'something',  
          'like',  
          'gamefly',  
          'is',  
          'so',  
          'good',  
          'because',  
          'you',  
          'can',  
          'actually',  
          'play',  
          'a',  
          'full',  
          'game',  
          'without',  
          'buying',  
          'it']
```

```
In [45]: text6[1600:1625]
```

```
Out[45]: ['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:

```
In [46]: sent = ['word1', 'word2', 'word3', 'word4', 'word5',  
                 'word6', 'word7', 'word8', 'word9', 'word10']
```

```
In [47]: sent[0]
```

```
Out[47]: 'word1'
```

```
In [48]: sent[9]
```

```
Out[48]: 'word10'
```



Notice that our indexes start from zero: sent element zero, written `sent[0]`, is the first word, `'word1'`, whereas sent element 9 is `'word10'`. The reason is simple: the moment Python accesses the content of a list from the computer's memory, it is already at the first element; we have to tell it how many elements forward to go. Thus, zero steps forward leaves it at the first element.

**Note** This practice of counting from zero is initially confusing, but typical of modern programming languages. You'll quickly get the hang of it if you've mastered the system of counting centuries where 19XY is a year in the 20th century, or if you live in a country where the floors of a building are numbered from 1, and so walking up  $n-1$  flights of stairs takes you to level  $n$ .

Now, if we accidentally use an index that is too large, we get an error:

```
In [49]: sent[10]

-----
IndexError                                Traceback (most recent call last)
<ipython-input-49-baa8b4ccd19d> in <module>
----> 1 sent[10]

IndexError: list index out of range
```

This time it is not a syntax error, because the program fragment is syntactically correct. Instead, it is a **runtime error**, and it produces a `Traceback` message that shows the context of the error, followed by the name of the error, `IndexError`, and a brief explanation.

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:

```
In [50]: sent[5:8]

Out[50]: ['word6', 'word7', 'word8']
```

```
In [51]: sent[5]

Out[51]: 'word6'
```

```
In [52]: sent[6]

Out[52]: 'word7'
```

```
In [53]: sent[7]
```

```
Out[53]: 'word8'
```

By convention, `m:n` means elements `m...n-1`. As the next example shows, we can omit the first number if the slice begins at the start of the list, and we can omit the second number if the slice goes to the end:

```
In [54]: sent[:3]
```

```
Out[54]: ['word1', 'word2', 'word3']
```

```
In [55]: text2[141525:]
```

```
Out[55]: ['among',  
          'the',  
          'merits',  
          'and',  
          'the',  
          'happiness',  
          'of',  
          'Elinor',  
          'and',  
          'Marianne',  
          ',',  
          'let',  
          'it',  
          'not',  
          'be',  
          'ranked',  
          'as',  
          'the',  
          'least',  
          'considerable',  
          ',',  
          'that',  
          'though',  
          'sisters',  
          ',',  
          'and',  
          'living',  
          'almost',  
          'within',  
          'sight',  
          'of',  
          'each',  
          'other',  
          ',',]
```

```
'they',
'could',
'live',
'without',
'disagreement',
'between',
'themselves',
',',
'or',
'producing',
'coolness',
'between',
'their',
'husbands',
'.',
'THE',
'END']
```

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.

```
In [56]: sent[0] = 'First'
sent[9] = 'Last'
len(sent)
```

```
Out[56]: 10
```

```
In [57]: sent[1:9] = ['Second', 'Third']
sent
```

```
Out[57]: ['First', 'Second', 'Third', 'Last']
```

```
In [58]: sent[9]
```

```
-----
-----
IndexError                                Traceback (most recent call
1 last)
<ipython-input-58-9d73df3f8677> in <module>
----> 1 sent[9]

IndexError: list index out of range
```

**Your Turn:** Take a few minutes to define a sentence of your own and modify individual words and groups of words (slices) using the same methods used earlier. Check your understanding by trying the exercises on lists at the end of this chapter. (You can add cells for practice if you would like.)

```
In [59]: text5[160:165]
```

```
Out[59]: ['serious', 'JOIN', 'PART', 'I', "'ll"]
```

```
In [61]: sent[:5]
```

```
Out[61]: ['First', 'Second', 'Third', 'Last']
```

## 2.3 Variables

From the start of 1, you have had access to texts called `text1`, `text2`, and so on. It saved a lot of typing to be able to refer to a 250,000-word book with a short name like this! In general, we can make up names for anything we care to calculate. We did this ourselves in the previous sections, e.g., defining a **variable** `sent1`, as follows:

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

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. The equals sign is slightly misleading, since information is moving from the right side to the left. It might help to think of it as a left-arrow. The name of the variable can be anything you like, e.g., `my_sent`, `sentence`, `xyzzzy`. It must start with a letter, and can include numbers and underscores. Here are some examples of variables and assignments:

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

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

```
In [ ]: wOrDs = sorted(noun_phrase)  
wOrDs
```

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

**Note:** Notice in the previous example that we split the definition of `my_sent` over two lines. Python expressions can be split across multiple lines, so long as this happens within any kind of brackets. Python uses the `"..."` prompt to indicate that more input is expected. It doesn't matter how much indentation is used in these continuation lines, but some indentation usually makes them easier to read.

It is good to choose meaningful variable names to remind you — and to help anyone else who reads your Python code — what your code is meant to do. Python does not try to make sense of the names; it blindly follows your instructions, and does not object if you do something confusing, such as `one = 'two'` or `two = 3`. The only restriction is that a variable name cannot be any of Python's reserved words, such as `def`, `if`, `not`, and `import`. If you use a reserved word, Python will produce a syntax error:

```
In [ ]: not = 'Camelot'
```

We will often use variables to hold intermediate steps of a computation, especially when this makes the code easier to follow. Thus `len(set(text1))` could also be written:

```
In [ ]: vocab = set(text1)
        vocab_size = len(vocab)
        vocab_size
```

### 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 (eg `name = 'Monty'`), index a string (eg. Each letter of the word Monty becomes an index of 5 letters: M, o, n, t, y), and slice a string (eg. select only a few letters from the index of letters that make up the word Monty): [Note from Lisa: I've changed the text here for clarity...]

```
In [62]: name = 'Monty'  
         name[0]
```

```
Out[62]: 'M'
```

```
In [63]: name[:4]
```

```
Out[63]: 'Mont'
```

We can also perform multiplication and addition with strings:

```
In [64]: name * 2
```

```
Out[64]: 'MontyMonty'
```

```
In [65]: name + '1'
```

```
Out[65]: 'Monty1'
```

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

```
In [66]: ' '.join(['Monty', 'Python'])
```

```
Out[66]: 'Monty Python'
```

```
In [67]: 'Monty Python'.split()
```

```
Out[67]: ['Monty', 'Python']
```

We will come back to the topic of strings in Chapter 3. For the time being, we have two important building blocks -- lists and strings -- and are ready to get back to some language analysis.

### 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 \(https://www.nltk.org/book/ch01.html#sec-computing-with-language-texts-and-words\)](https://www.nltk.org/book/ch01.html#sec-computing-with-language-texts-and-words), 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. As in [1 \(https://www.nltk.org/book/ch01.html#sec-computing-with-language-texts-and-words\)](https://www.nltk.org/book/ch01.html#sec-computing-with-language-texts-and-words), you can try new features of the Python language by copying them into the interpreter, and you'll learn about these features systematically in the following section.

Before continuing further, you might like to check your understanding of the last section by predicting the output of the following code. You can use the interpreter to check whether you got it right. If you're not sure how to do this task, it would be a good idea to review the previous section before continuing further.

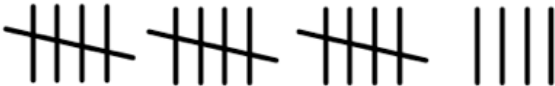
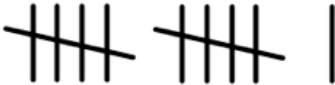


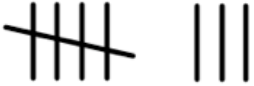
```
In [68]: saying = ['After', 'all', 'is', 'said', 'and', 'done',  
                  'more', 'is', 'said', 'than', 'done']  
tokens = set(saying)  
tokens = sorted(tokens)  
tokens[-2:]
```

```
Out[68]: ['said', 'than']
```

#### 3.1 Frequency Distributions

How can we automatically identify the words of a text that are most informative about the topic and genre of the text? Imagine how you might go about finding the 50 most frequent words of a book. One method would be to keep a tally for each vocabulary item, like that shown in [3.1 \(https://www.nltk.org/book/ch01.html#fig-tally\)](https://www.nltk.org/book/ch01.html#fig-tally). The tally would need thousands of rows, and it would be an exceedingly laborious process — so laborious that we would rather assign the task to a machine.

# Word Tally

|           |  |
|-----------|--|
| the       |  |
| been      |  |
| message   |   |
| persevere |   |
| nation    |  |

The table in [3.1 \(https://www.nltk.org/book/ch01.html#fig-tally\)](https://www.nltk.org/book/ch01.html#fig-tally) is known as a **frequency distribution**, and it tells us the frequency of each vocabulary item in the text. (In general, it could count any kind of observable event.) 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 [69]: fdist1 = FreqDist(text1)
         print(fdist1)
```

```
<FreqDist with 19317 samples and 260819 outcomes>
```

```
In [70]: fdist1.most_common(50)
```

```
Out[70]: [(' ', 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),
('--', 1070),
('him', 1058),
('from', 1052),
('be', 1030),
('on', 1005),
('so', 918),
('whale', 906),
('one', 889),
('you', 841),
('had', 767),
('have', 760),
('there', 715),
('But', 705),
('or', 697),
('were', 680),
('now', 646),
('which', 640),
('?', 637),
('me', 627),
('like', 624)]
```

```
In [71]: fdist1['whale']
```

```
Out[71]: 906
```

When we first invoke `FreqDist`, we pass the name of the text as an argument. We can inspect the total number of words ("outcomes") that have been counted up — 260,819 in the case of *Moby Dick*. The expression `most_common(50)` gives us a list of the 50 most frequently occurring types in the text.

**Your Turn:** Try the preceding frequency distribution example for yourself, for `text2`. Be careful to use the correct parentheses and uppercase letters. If you get an error message `NameError: name 'FreqDist' is not defined`, you need to start your work with `from nltk.book import *`

```
In [72]: fdist1 = FreqDist(text2)
         print(fdist1)
```

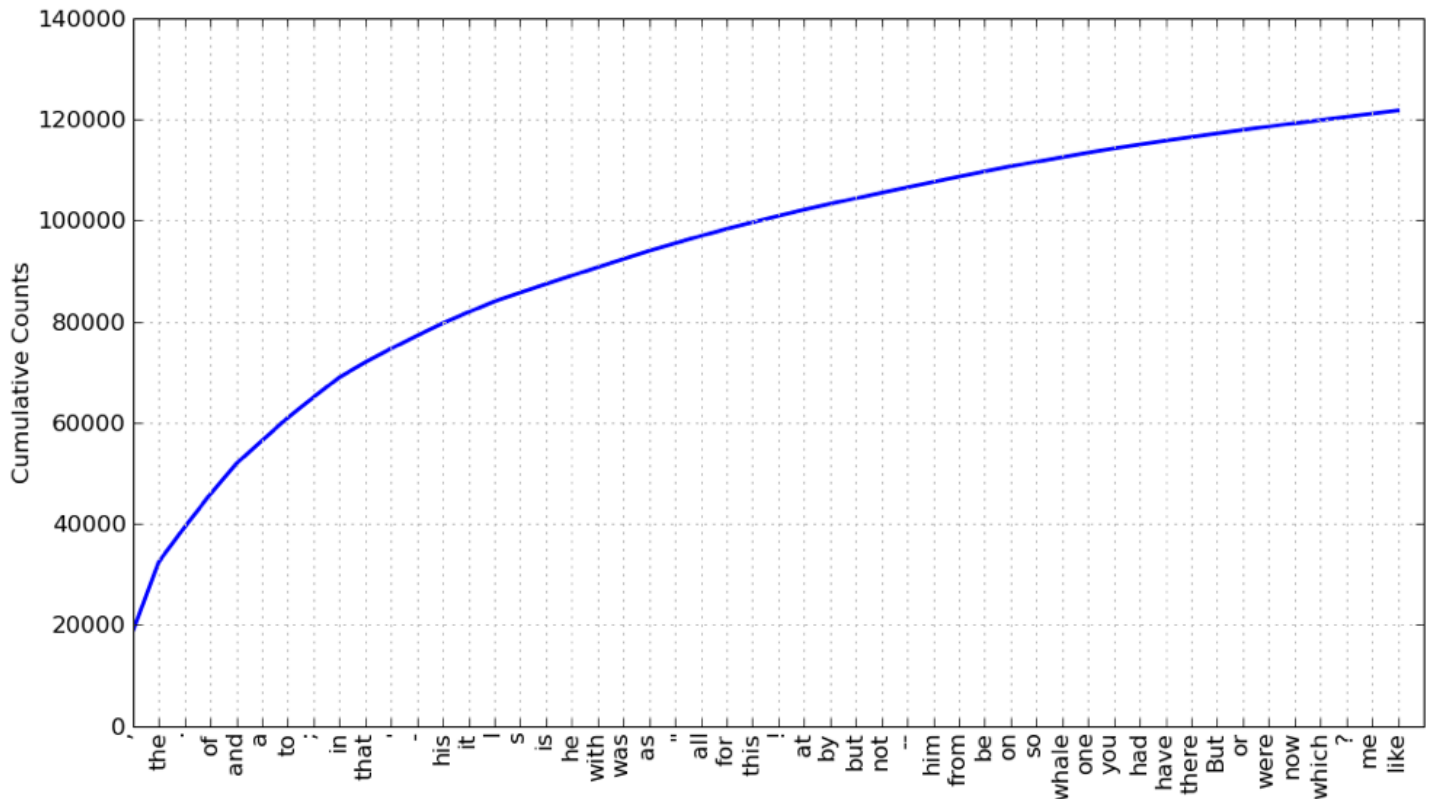
```
<FreqDist with 6833 samples and 141576 outcomes>
```

```
In [73]: fdist1.most_common(10)
```

```
Out[73]: [(' ', 9397),
          ('to', 4063),
          ('.', 3975),
          ('the', 3861),
          ('of', 3565),
          ('and', 3350),
          ('her', 2436),
          ('a', 2043),
          ('I', 2004),
          ('in', 1904)]
```

```
In [ ]: #experiment here.
```

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 below. These 50 words account for nearly half the book!



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 [74]: fdist1.hapaxes()
```

```
Out[74]: ['Sense',
          'Sensibility',
          'Jane',
          'Austen',
          '1811',
          ']',
          'generations',
          'inheritor',
          'bequeath',
          'Gentleman',
```

'relish',  
'devolved',  
'inheriting',  
'moiety',  
'bequest',  
'sale',  
'articulation',  
'economically',  
'tardy',  
'survived',  
'including',  
'legacies',  
'urgency',  
'prudently',  
'caricature',  
'warmed',  
'Three',  
'successively',  
'funeral',  
'dispute',  
'decease',  
'indelicacy',  
'keen',  
'whomsoever',  
'immoveable',  
'acutely',  
'qualified',  
'counsellor',  
'joys',  
'afflicted',  
'strive',  
'humored',  
'imbibed',  
'romance',  
'thirteen',  
'2',  
'installed',  
'degraded',  
'seasons',  
'alloy',  
'impoverishing',  
'headed',  
'stipulate',  
'LET',  
'diminished',  
'REALLY',  
'strikes',  
'Fifteen',  
'healthy',  
'clogged',

'superannuated',  
'amazing',  
'restriction',  
'drains',  
'bind',  
'inconvenient',  
'enlarge',  
'sixpence',  
'discharging',  
'presents',  
'fish',  
'Altogether',  
'cheap',  
'fulfil',  
'removes',  
'stock',  
'belongs',  
'indecorous',  
'neighbourly',  
'3',  
'remembrances',  
'indefatigable',  
'steadier',  
'earthly',  
'7000L',  
'repressed',  
'uninfluenced',  
'asunder',  
'political',  
'parliament',  
'barouches',  
'centered',  
'forcibly',  
'implies',  
'militated',  
'uninteresting',  
'Music',  
'draws',  
'connoisseur',  
'coincide',  
'impenetrable',  
'prose',  
'virtues',  
'ornament',  
'destiny',  
'4',  
'distrusts',  
'innate',  
'cordial',  
'tastes',

'worthy',  
'keeps',  
'literature',  
'imperfection',  
'Esteem',  
'Cold',  
'Ashamed',  
'outstripped',  
'stimulated',  
'prosperous',  
'denote',  
'unpromising',  
'inquietude',  
'forbade',  
'aggrandizement',  
'uncivil',  
'DRAW',  
'IN',  
'endeavor',  
'deliberation',  
'scale',  
'vicinity',  
'5',  
'dispatched',  
'incommode',  
'inhabiting',  
'civilly',  
'replying',  
'northward',  
'disapprobation',  
'conscientiously',  
'impracticable',  
'consisted',  
'packages',  
'depart',  
'west',  
'disposing',  
'undoubtedly',  
'feebly',  
'drift',  
'purse',  
'calculation',  
'ye',  
'branch',  
'6',  
'Valley',  
'fertile',  
'wooded',  
'pasture',  
'demesne',

'wicket',  
'defective',  
'tiled',  
'shutters',  
'honeysuckles',  
'garrets',  
'cheered',  
'recommending',  
'cultivated',  
'woody',  
'terminated',  
'branched',  
'steepest',  
'parlors',  
'remainder',  
'garret',  
'widen',  
'alterations',  
'savings',  
'endeavoring',  
'unpacked',  
'affixed',  
'landlord',  
'sociable',  
'denoting',  
'detract',  
'Conversation',  
'chatty',  
'differed',  
'7',  
'screened',  
'projection',  
'hospitality',  
'dissimilar',  
'talent',  
'unconnected',  
'compass',  
'hunted',  
'resources',  
'Continual',  
'piqued',  
'collecting',  
'noisier',  
'juvenile',  
'ham',  
'chicken',  
'unsatiable',  
'females',  
'manor',  
'moonlight',

'Luckily',  
'fat',  
'witty',  
'hearts',  
'pretended',  
'blush',  
'attacks',  
'adapted',  
'repulsive',  
'boisterous',  
'tore',  
'unlocked',  
'songs',  
'lain',  
'position',  
'celebrated',  
'applauded',  
'sing',  
'forfeited',  
'shameless',  
'amounted',  
'ecstatic',  
'sympathize',  
'estimable',  
'contrasted',  
'insensibility',  
'8',  
'ample',  
'respectably',  
'promotion',  
'zealously',  
'weddings',  
'blushes',  
'decisively',  
'accusation',  
'protect',  
'Infirmity',  
'rheumatism',  
'miracle',  
'losing',  
'commercial',  
'dooming',  
'rheumatic',  
'waistcoats',  
'cramps',  
'rheumatisms',  
'afflict',  
'feeble',  
'despised',  
'Confess',



'flushed',  
'Soon',  
'grate',  
'bedchamber',  
'unaccountably',  
'dejected',  
'9',  
'familiar',  
'occupation',  
'classed',  
'reminding',  
'possessor',  
'stirred',  
'abounded',  
'alternative',  
'valleys',  
'memorable',  
'sunshine',  
'tempting',  
'pencil',  
'threatening',  
'cloud',  
'blue',  
'faces',  
'gales',  
'westerly',  
'resisting',  
'heads',  
'Chagrined',  
'exigence',  
'speed',  
'pointers',  
'apologized',  
'gracefulness',  
'theme',  
'gallantry',  
'lifting',  
'hero',  
'dresses',  
'shooting',  
'jacket',  
'sprained',  
'ankle',  
'indignantly',  
'black',  
'bitch',  
'shades',  
'Court',  
'inherit',  
'adding',

'incommoded',  
'CATCHING',  
'Men',  
'hop',  
'danced',  
'abhor',  
'odious',  
'gross',  
'construction',  
'deemed',  
'ingenuity',  
'smitten',  
'spraining',  
'ankles',  
'10',  
'preserver',  
'precision',  
'styled',  
'cant',  
'outraged',  
'transparency',  
'hardily',  
'passionately',  
'largest',  
'conformity',  
'Encouraged',  
'convert',  
'passages',  
'idolized',  
'acquiesced',  
'decisions',  
'Pope',  
'exhausted',  
'scanty',  
'roads',  
'jest',  
'irksome',  
'persons',  
'sacrificing',  
'undivided',  
'slighting',  
'desperation',  
'seized',  
'rash',  
'delineated',  
'brighter',  
'speculative',  
'incurred',  
'annexed',  
'successful',

'mild',  
'gloominess',  
'undervalue',  
'remembers',  
'patronised',  
'indignity',  
'contemptuously',  
'climate',  
'mosquitoes',  
'nabobs',  
'mohrs',  
'palanquins',  
'coats',  
'Add',  
'brilliancy',  
'imperfections',  
'mass',  
'possessing',  
'disarm',  
'mare',  
'11',  
'occupy',  
'execution',  
'included',  
'witnessing',  
'excellencies',  
'marking',  
'aim',  
'illaudable',  
'illustration',  
'partners',  
'dances',  
'everlasting',  
'talker',  
'history',  
'sisterly',  
'accidentally',  
'contrives',  
'basis',  
'define',  
'prejudices',  
'systems',  
'nought',  
'resumed',  
'criminal',  
'minutiae',  
'pardonable',  
'refinements',  
'connect',  
'disastrous',

'12',  
'arrives',  
'Imagine',  
'gallop',  
'awaken',  
'dream',  
'truths',  
'Seven',  
'Opposition',  
'appeal',  
'representing',  
'faithful',  
'forego',  
'Queen',  
'Mab',  
'overheard',  
'tempers',  
'clearer',  
'Down',  
'whispering',  
'tumbled',  
'stated',  
'unison',  
'attacked',  
'curate',  
'invention',  
'begins',  
'inelegant',  
'mindful',  
'amidst',  
'proprietor',  
'sail',  
'provisions',  
'undertaking',  
'13',  
'dispersing',  
'hardships',  
'None',  
'trick',  
'Newton',  
'conveniently',  
'pry',  
'stare',  
'consultation',  
'leant',  
'Impudence',  
'WHERE',  
'propose',  
'enquired',  
'pleasanter',

'pleasantness',  
'evince',  
'bowling',  
'newly',  
'pleasantest',  
'14',  
'termination',  
'wonderer',  
'comings',  
'goings',  
'befallen',  
'wager',  
'speculation',  
'incompatible',  
'contradictory',  
'ending',  
'happening',  
'Improve',  
'inch',  
'local',  
'unemployed',  
'uselessly',  
'pull',  
'smokes',  
'INconvenience',  
'broader',  
'endear',  
'denoted',  
'inhabited',  
'prescience',  
'Must',  
'degrade',  
'Extend',  
'15',  
'pretext',  
'foresight',  
'noticing',  
'Surprised',  
'partook',  
'powered',  
'Disappointment',  
'exercised',  
'dispatches',  
'exhilaration',  
'Almost',  
'confusedly',  
'torment',  
'indubitable',  
'feeding',  
'Gone',

'quarrelled',  
'suspects',  
'disapproves',  
'transact',  
'dismiss',  
'dares',  
'feels',  
'cavil',  
'blameable',  
'inadvertence',  
'depressed',  
'certainties',  
'unavoidably',  
'Secrecy',  
'practiced',  
'ARE',  
'Concealing',  
'accuse',  
'incautiousness',  
'outweighs',  
'fainter',  
'mighty',  
'concession',  
'Ungracious',  
'stranger',  
'independently',  
'prosperously',  
'representations',  
'swollen',  
'16',  
'risen',  
'headache',  
'potent',  
'oftenest',  
'singing',  
'fetches',  
'mystery',  
'Supposing',  
'inflict',  
'unacknowledged',  
'dearly',  
'revelment',  
'overstrained',  
'witticisms',  
'volume',  
'Shakespeare',  
'Hamlet',  
'Months',  
'Hitherto',  
'stole',

'climbing',  
'disapproved',  
'seclusion',  
'MIND',  
'controlled',  
'stretch',  
'reaching',  
'riding',  
'particularity',  
'quicken',  
'dispersed',  
'continuation',  
'elect',  
'thickly',  
'transporting',  
'showers',  
'inspired',  
'nuisance',  
'swept',  
'SOMETIMES',  
'equals',  
'rises',  
'bottoms',  
'friendliest',  
'directing',  
'extorting',  
'regulate',  
'17',  
'extending',  
'reanimate',  
'parents',  
'fame',  
'famous',  
'Greatness',  
'Strange',  
'Grandeur',  
'external',  
'TWO',  
'describing',  
'accurately',  
'Hunters',  
'hunt',  
'novel',  
'apiece',  
'insufficiency',  
'vanish',  
'booksellers',  
'sellers',  
'Thomson',  
'disputes',

'loose',  
'cash',  
'collection',  
'bulk',  
'heirs',  
'ablest',  
'maxim',  
'presume',  
'steadfast',  
'detected',  
'mistakes',  
'ingenious',  
'oneself',  
'deliberate',  
'subservient',  
'confound',  
'conform',  
'Quite',  
'foolishly',  
'negligent',  
'awkwardness',  
'Shyness',  
'fullest',  
'18',  
'inspiring',  
'reservedness',  
'intimated',  
'upstairs',  
'\*\*\*',  
'minutely',  
'surfaces',  
'uncouth',  
'irregular',  
'rugged',  
'indistinct',  
'medium',  
'hazy',  
'meadows',  
'scattered',  
'answers',  
'unites',  
'utility',  
'rocks',  
'promontories',  
'grey',  
'moss',  
'brush',  
'believes',  
'affects',  
'discrimination',



'possesses',  
'scenery',  
'pretends',  
'tries',  
'defined',  
'blasted',  
'flourishing',  
'tattered',  
'nettles',  
'thistles',  
'heath',  
'blossoms',  
'tower',  
'troop',  
'tidy',  
'villages',  
'banditti',  
'plait',  
'conspicuous',  
'darker',  
'inconsiderately',  
'surpassed',  
'casts',  
'instantaneously',  
'conclusions',  
'theft',  
'contrivance',  
'henceforward',  
'newness',  
'significant',  
'instructions',  
'entertainment',  
'contribute',  
'Impossible',  
'Whitakers',  
'blushing',  
'guessing',  
'hunts',  
'archness',  
'19',  
'unequal',  
'environs',  
'disengaged',  
'detested',  
'Disappointed',  
'qualifications',  
'painfully',  
'extorted',  
'consistency',  
'shortness',

'temporizing',  
'grievance',  
'reformed',  
'renewal',  
'finger',  
'helpless',  
'army',  
'chambers',  
'Temple',  
'circles',  
'gigs',  
'abstruse',  
'navy',  
'solicitations',  
'trades',  
'Columella',  
'fascinating',  
'ere',  
'defy',  
'desponding',  
'judiciously',  
'augment',  
'meritorious',  
'ones',  
'loving',  
'shutting',  
'abundance',  
'forbidden',  
'chained',  
'closing',  
'knocking',  
'turf',  
'casement',  
'Hush',  
'hallooing',  
'plump',  
'prepossessing',  
'briefly',  
'surveying',  
'meantime',  
'nodding',  
'ushered',  
'stared',  
'pitched',  
'ceiling',  
'oftener',  
'20',  
'ours',  
'chaperon',  
'bowing',

'complaining',  
'Dullness',  
'billiard',  
'vile',  
'Gilberts',  
'whip',  
'abused',  
'genuinely',  
'unaffectedly',  
'soured',  
'bias',  
'blunder',  
'establishing',  
'sneer',  
'canvassing',  
'election',  
'fatiguing',  
'Parliament',  
'P',  
'irrational',  
'palm',  
'abuses',  
'languages',  
'decline',  
'changing',  
'gathered',  
';--"',  
'Somehow',  
'Weymouth',  
'praises',  
'objected',  
'21',  
'entertain',  
'unsuitableness',  
'invite',  
'philosophy',  
'ungenteel',  
'unfashionable',  
'doatingly',  
'Benevolent',  
'philanthropic',  
'playthings',  
'boasting',  
'smartness',  
'actual',  
'judicious',  
'extolling',  
'humouring',  
'whims',  
'importunate',

'patterns',  
'foibles',  
'rapacious',  
'beings',  
'credulous',  
'exorbitant',  
'swallow',  
'endurance',  
'viewed',  
'encroachments',  
'mischievous',  
'sashes',  
'untied',  
'bags',  
'knives',  
'composedly',  
'claiming',  
'monkey',  
'pinching',  
'playful',  
'caressing',  
'embraces',  
'scratching',  
'pattern',  
'outdone',  
'consternation',  
'surpass',  
'emergency',  
'assuage',  
'sufferer',  
'kisses',  
'bathed',  
'stuffed',  
'sugar',  
'plums',  
'sobbed',  
'lustily',  
'kicked',  
'brothers',  
'soothings',  
'ineffectual',  
'apricot',  
'marmalade',  
'successfully',  
'bruised',  
'remedy',  
'scratch',  
'medicine',  
'trivial',  
'lies',

'eclat',  
'doat',  
'distractedly',  
'outside',  
'estimate',  
'lief',  
'nasty',  
'Rose',  
'clerk',  
'Simpson',  
'blinded',  
'shrewd',  
'artlessness',  
'niggardly',  
'dealt',  
'consists',  
'Twill',  
'proclaiming',  
'newer',  
'significancy',  
'nods',  
'winks',  
'countless',  
'wittiest',  
'alphabet',  
'inquisitiveness',  
'assertions',  
'petty',  
'unavailing',  
'22',  
'toleration',  
'vulgarity',  
'striving',  
'amusing',  
'mental',  
'rectitude',  
'equality',  
'toward',  
'valueless',  
'curious',  
'manage',  
'amiably',  
'bashful',  
'Astonishment',  
'disbelief',  
'divine',  
'varied',  
'incredulity',  
'swoon',  
'cautiously',

'date',  
'pupil',  
'OCCASION',  
'puts',  
'detecting',  
'likeness',  
'paces',  
'keeping',  
'Besides',  
'wiping',  
'break',  
'Pardon',  
'provide',  
'sometime',  
'carelessly',  
'sheet',  
'subsist',  
'struggled',  
'Writing',  
'separations',  
'confounded',  
'editions',  
'23',  
'inventing',  
'asserted',  
'indisputable',  
'dissatisfaction',  
'evidence',  
'condemning',  
'dupe',  
'illusion',  
'softener',  
'defended',  
'regain',  
'infatuation',  
'frivolous',  
'supposition',  
'alienated',  
'Supported',  
'extinction',  
'mourning',  
'obstacles',  
'perfections',  
'entrusted',  
'aggravation',  
'unshaken',  
'poignant',  
'renewing',  
'venturing',  
'confessedly',

'joking',  
'disclosure',  
'mistrust',  
'joining',  
'club',  
'foresaw',  
'fairer',  
'permission',  
'compliant',  
'frightful',  
'novelty',  
'candlelight',  
'candles',  
'reseated',  
'spoilt',  
'Casino',  
'Ladyship',  
'touched',  
'tuned',  
'toned',  
'rolling',  
'labour',  
'singly',  
'profited',  
'proposals',  
'rivals',  
'forwarding',  
'wrapped',  
'24',  
'undeserving',  
'apologize',  
'ice',  
'Offended',  
'quarrelling',  
'madness',  
'test',  
'lowness',  
'impose',  
'involve',  
'owning',  
...]

## 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 \(https://www.nltk.org/book/ch01.html#ex-set-comprehension-math\)](https://www.nltk.org/book/ch01.html#ex-set-comprehension-math). This means "the set of all  $w$  such that  $w$  is an element of  $V$  (the vocabulary) and  $w$  has property  $P$ ".

$$(1) \quad \text{a. } \{w \mid w \in V \ \& \ P(w)\}$$

$$\text{b. } [w \text{ for } w \text{ in } V \text{ if } p(w)]$$

The corresponding Python expression is given in (1b). (Note that it produces a list, not a set, which means that duplicates are possible.) Observe how similar the two notations are. Let's go one more step and write executable Python code:

```
In [75]: V = set(text9)
         long_words = [w for w in V if len(w) > 15]
         sorted(long_words)
```

```
Out[75]: ['incomprehensible', 'undenominational']
```

For each word  $w$  in the vocabulary  $V$ , we check whether  $\text{len}(w)$  is greater than 15; all other words will be ignored. We will discuss this syntax more carefully later.

**Your Turn:** Try out the previous statements in a cell of your own, and experiment with changing the text and changing the length condition. Does it make a difference to your results if you change the variable names, e.g., using `[word for word in vocab if ...]`?

```
In [76]: V = set(text7)
         long_words = [w for w in V if len(w) > 15]
         sorted(long_words)
```

```
Out[76]: ['238,000-circulation',
         'Arbitrage-related',
         'Bridgestone\\Firestone',
         'Chinese-American',
```



'Corton-Charlemagne',  
'English-speaking',  
'Freeport-McMoRan',  
'Hart-Scott-Rodino',  
'Lafite-Rothschild',  
'Louisiana-Pacific',  
'Macmillan\\McGraw',  
'Macmillan\\McGraw-Hill',  
'Metallgesellschaft',  
'Minneapolis-based',  
'Philadelphia-based',  
'Property\\casualty',  
'Prudential-Bache',  
'Renaissance-style',  
'Sacramento-based',  
'Test-preparation',  
'Trockenbeerenauslesen',  
'Washington-based',  
'abortion-related',  
'achievement-test',  
'acquisition-minded',  
'anti-abortionists',  
'anti-miscarriage',  
'anti-morning-sickness',  
'anti-programmers',  
'asbestos-related',  
'automotive-lighting',  
'automotive-parts',  
'battery-operated',  
'bread-and-butter',  
'building-products',  
'collective-bargaining',  
'computer-assisted',  
'computer-generated',  
'computer-services',  
'computer-system-design',  
'constitutional-law',  
'counterrevolutionary',  
'creditworthiness',  
'current-carrying',  
'diethylstilbestrol',  
'direct-investment',  
'disaster-assistance',  
'disproportionate',  
'dollar-denominated',  
'early-retirement',  
'electric-utility',  
'electrical-safety',  
'environmentalists',  
'equal-opportunity',

'executive-office',  
'financial-services',  
'get-out-the-vote',  
'government-certified',  
'government-funded',  
'government-owned',  
'headcount-control',  
'housing-assistance',  
'identity-management',  
'incentive-backed',  
'industrial-production',  
'industry-supported',  
'information-services',  
'insurance-company',  
'intellectual-property',  
'interest-bearing',  
'investment-grade',  
'investor-relations',  
'labor-management',  
'language-housekeeper',  
'larger-than-normal',  
'less-than-brilliant',  
'life-of-contract',  
'limited-partnership',  
'machine-gun-toting',  
'marketing-communications',  
'most-likely-successor',  
'multibillion-dollar',  
'newspaper-printing',  
'non-biodegradable',  
'non-encapsulating',  
'noncompetitively',  
'over-the-counter',  
'parts-engineering',  
'pianist-comedian',  
'price-depressing',  
'property\\casualty',  
'public-relations',  
'recession-inspired',  
'responsibilities',  
'revenue-desperate',  
'savers\\investors',  
'savings-and-loan',  
'school-improvement',  
'school-sponsored',  
'search-and-seizure',  
'securities-based',  
'self-aggrandizing',  
'self-perpetuating',  
'seven-million-ton',

```
'shareholder-rights',  
'sometimes-exhausting',  
'sometimes-tawdry',  
'state-supervised',  
'stock-manipulation',  
'stock-specialist',  
'substance-abusing',  
'telecommunications',  
'telephone-information',  
'test-preparation',  
'tissue-transplant',  
'truth-in-lending',  
'unconstitutional',  
'weapons-modernization',  
'yet-to-be-formed',  
'yttrium-containing']
```

Let's return to our task of finding words that characterize a text. Notice that the long words in text4 reflect its national focus — *constitutionally*, *transcontinental* — whereas those in text5 reflect its informal content: *booooooooooooooglyyyyyy* and *yuuuuuuuuuuuummmmmmmmmmmmmmmmm*. Have we succeeded in automatically extracting words that typify a text? Well, 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 that are longer than seven characters, that occur more than seven times:

```
In [77]: fdist5 = FreqDist(text5)
         sorted(w for w in set(text5) if len(w) > 7 and fdist5[w] > 7)
```

```
Out[77]: ['#14-19teens',
          '#talkcity_adults',
          '(((((((((',
          '.....',
          'Question',
          'actually',
          'anything',
          'computer',
          'cute.-ass',
          'everyone',
          'football',
          'innocent',
          'listening',
          'remember',
          'seriously',
          'something',
          'together',
          'tomorrow',
          'watching']
```

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 `nltk.bigrams()`:

**(NOTE)** This is a fix that is not reflected in the book. Bigrams are being handled differently in NLTK. The other way to do this is to add one line that reads: `from nltk.util import bigrams`

```
In [78]: list(nltk.bigrams(['more', 'is', 'said', 'than', 'done']))
```

```
Out[78]: [('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. We will see how it works later.

```
In [79]: text4.collocations()
```

```
-----
-----
ValueError                                Traceback (most recent call
last)
<ipython-input-79-56fc6e5515a1> in <module>
----> 1 text4.collocations()

//anaconda3/lib/python3.7/site-packages/nltk/text.py in collocations
(self, num, window_size)
    442
    443         collocation_strings = [
--> 444             w1 + " " + w2 for w1, w2 in self.collocation_list(num, window_size)
    445         ]
    446         print(tokenwrap(collocation_strings, separator="; "))
)

//anaconda3/lib/python3.7/site-packages/nltk/text.py in <listcomp>(.
0)
    442
    443         collocation_strings = [
--> 444             w1 + " " + w2 for w1, w2 in self.collocation_list(num, window_size)
    445         ]
    446         print(tokenwrap(collocation_strings, separator="; "))
)

ValueError: too many values to unpack (expected 2)
```

```
In [80]: text8.collocations()
```

```
-----
-----
ValueError                                Traceback (most recent call
last)
<ipython-input-80-103ab84c5ccd> in <module>
----> 1 text8.collocations()

//anaconda3/lib/python3.7/site-packages/nltk/text.py in collocations
(self, num, window_size)
    442
    443         collocation_strings = [
--> 444             w1 + " " + w2 for w1, w2 in self.collocation_list(num, window_size)
    445         ]
    446         print(tokenwrap(collocation_strings, separator="; "))
)

//anaconda3/lib/python3.7/site-packages/nltk/text.py in <listcomp>(.
0)
    442
    443         collocation_strings = [
--> 444             w1 + " " + w2 for w1, w2 in self.collocation_list(num, window_size)
    445         ]
    446         print(tokenwrap(collocation_strings, separator="; "))
)

ValueError: too many values to unpack (expected 2)
```

Th collocations that emerge are very specific to the genre of the texts. In order to find *red wine* as a collocation, we would need to process a much larger body of text.

### 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 [81]: [len(w) for w in text1]
```

```
Out[81]: [1,
          4,
          4,
```

2,  
6,  
8,  
4,  
1,  
9,  
1,  
1,  
8,  
2,  
1,  
4,  
11,  
5,  
2,  
1,  
7,  
6,  
1,  
3,  
4,  
5,  
2,  
10,  
2,  
4,  
1,  
5,  
1,  
4,  
1,  
3,  
5,  
1,  
1,  
3,  
3,  
3,  
1,  
2,  
3,  
4,  
7,  
3,  
3,  
8,  
3,  
8,  
1,  
4,

1,  
5,  
12,  
1,  
9,  
11,  
4,  
3,  
3,  
3,  
5,  
2,  
3,  
3,  
5,  
7,  
2,  
3,  
5,  
1,  
2,  
5,  
2,  
4,  
3,  
3,  
8,  
1,  
2,  
7,  
6,  
8,  
3,  
2,  
3,  
9,  
1,  
1,  
5,  
3,  
4,  
2,  
4,  
2,  
6,  
6,  
1,  
3,  
2,  
5,



4,  
2,  
4,  
4,  
1,  
5,  
1,  
4,  
2,  
2,  
2,  
6,  
2,  
3,  
6,  
7,  
3,  
1,  
7,  
9,  
1,  
3,  
6,  
1,  
1,  
5,  
6,  
5,  
6,  
3,  
13,  
2,  
3,  
4,  
1,  
3,  
7,  
4,  
5,  
2,  
3,  
4,  
2,  
2,  
8,  
1,  
5,  
1,  
3,  
2,

1,  
3,  
3,  
1,  
4,  
1,  
4,  
6,  
2,  
5,  
4,  
9,  
2,  
7,  
1,  
3,  
2,  
3,  
1,  
5,  
2,  
6,  
2,  
7,  
2,  
2,  
7,  
1,  
1,  
10,  
1,  
5,  
1,  
3,  
2,  
2,  
4,  
11,  
4,  
3,  
3,  
1,  
3,  
3,  
1,  
6,  
1,  
1,  
1,  
1,

1,  
4,  
1,  
3,  
1,  
2,  
4,  
1,  
2,  
6,  
2,  
2,  
10,  
1,  
1,  
10,  
5,  
1,  
5,  
1,  
5,  
1,  
5,  
1,  
5,  
1,  
5,  
1,  
5,  
1,  
5,  
1,  
6,  
1,  
3,  
1,  
5,  
1,  
4,  
1,  
7,  
1,  
5,  
1,  
9,  
1,  
5,  
1,  
7,  
1,

7,  
1,  
6,  
1,  
7,  
1,  
7,  
1,  
5,  
1,  
4,  
1,  
4,  
1,  
5,  
1,  
5,  
1,  
4,  
1,  
4,  
1,  
11,  
1,  
8,  
1,  
8,  
2,  
1,  
3,  
1,  
3,  
1,  
9,  
2,  
2,  
4,  
2,  
4,  
4,  
4,  
4,  
4,  
11,  
8,  
3,  
4,  
1,  
4,  
2,  
1,

4,  
5,  
2,  
1,  
3,  
1,  
3,  
7,  
2,  
4,  
4,  
7,  
3,  
4,  
8,  
3,  
6,  
1,  
6,  
2,  
3,  
5,  
1,  
7,  
2,  
8,  
6,  
9,  
2,  
6,  
2,  
5,  
7,  
4,  
2,  
3,  
4,  
10,  
1,  
6,  
2,  
7,  
1,  
9,  
3,  
4,  
3,  
1,  
2,  
5,

4,  
2,  
5,  
1,  
4,  
3,  
8,  
1,  
8,  
5,  
10,  
1,  
7,  
9,  
1,  
2,  
5,  
8,  
1,  
3,  
9,  
6,  
8,  
1,  
3,  
4,  
2,  
1,  
2,  
8,  
3,  
7,  
7,  
9,  
1,  
2,  
4,  
2,  
3,  
5,  
4,  
9,  
1,  
5,  
8,  
3,  
6,  
8,  
2,  
12,

1,  
2,  
9,  
1,  
8,  
4,  
1,  
1,  
3,  
4,  
2,  
4,  
3,  
4,  
13,  
4,  
1,  
7,  
1,  
7,  
1,  
3,  
4,  
2,  
9,  
1,  
2,  
4,  
7,  
3,  
11,  
1,  
9,  
3,  
3,  
1,  
2,  
4,  
4,  
4,  
1,  
4,  
5,  
2,  
1,  
3,  
1,  
3,  
1,  
5,

11,  
1,  
2,  
1,  
4,  
9,  
2,  
4,  
8,  
1,  
6,  
5,  
5,  
2,  
4,  
2,  
4,  
5,  
4,  
4,  
4,  
1,  
3,  
3,  
4,  
4,  
4,  
6,  
5,  
2,  
3,  
4,  
1,  
6,  
1,  
3,  
4,  
4,  
3,  
9,  
5,  
2,  
3,  
1,  
3,  
4,  
4,  
1,  
8,  
1,



3,  
1,  
3,  
4,  
9,  
4,  
5,  
1,  
3,  
3,  
2,  
4,  
7,  
1,  
4,  
4,  
4,  
3,  
5,  
7,  
1,  
3,  
2,  
3,  
10,  
10,  
7,  
2,  
4,  
2,  
2,  
1,  
3,  
1,  
4,  
1,  
3,  
2,  
3,  
4,  
3,  
4,  
5,  
2,  
4,  
2,  
6,  
3,  
5,  
1,

2,  
2,  
4,  
3,  
4,  
5,  
2,  
3,  
4,  
2,  
9,  
1,  
5,  
4,  
1,  
5,  
5,  
3,  
7,  
5,  
3,  
3,  
9,  
3,  
2,  
1,  
3,  
4,  
4,  
4,  
5,  
3,  
3,  
5,  
2,  
3,  
5,  
1,  
4,  
4,  
4,  
6,  
1,  
3,  
4,  
7,  
3,  
4,  
4,  
6,

3,  
8,  
3,  
3,  
5,  
1,  
7,  
7,  
1,  
3,  
6,  
8,  
2,  
4,  
1,  
8,  
7,  
1,  
7,  
1,  
3,  
7,  
1,  
7,  
4,  
6,  
1,  
4,  
2,  
6,  
3,  
10,  
6,  
8,  
2,  
5,  
1,  
2,  
5,  
6,  
14,  
7,  
1,  
8,  
1,  
1,  
3,  
3,  
7,  
5,

6,  
2,  
2,  
7,  
1,  
1,  
9,  
6,  
1,  
4,  
2,  
5,  
5,  
3,  
1,  
3,  
5,  
5,  
3,  
4,  
2,  
2,  
5,  
2,  
2,  
3,  
1,  
1,  
3,  
3,  
4,  
3,  
8,  
1,  
5,  
4,  
2,  
7,  
2,  
5,  
2,  
2,  
5,  
1,  
1,  
5,  
2,  
3,  
5,  
1,

5,  
2,  
4,  
9,  
4,  
4,  
4,  
4,  
2,  
4,  
7,  
2,  
2,  
6,  
1,  
1,  
2,  
4,  
3,  
1,  
3,  
4,  
4,  
3,  
4,  
1,  
3,  
5,  
1,  
3,  
6,  
5,  
1,  
5,  
6,  
9,  
3,  
8,  
7,  
1,  
4,  
9,  
4,  
7,  
7,  
1,  
3,  
2,  
5,  
4,

3,  
6,  
4,  
2,  
2,  
3,  
3,  
2,  
2,  
6,  
1,  
3,  
4,  
5,  
6,  
7,  
6,  
6,  
3,  
5,  
2,  
4,  
7,  
1,  
1,  
5,  
1,  
2,  
2,  
5,  
1,  
4,  
1,  
2,  
5,  
1,  
4,  
2,  
4,  
3,  
13,  
4,  
4,  
5,  
7,  
2,  
3,  
1,  
3,  
9,

2,  
3,  
10,  
4,  
2,  
3,  
6,  
2,  
2,  
7,  
1,  
1,  
8,  
1,  
1,  
6,  
1,  
1,  
3,  
6,  
3,  
8,  
3,  
4,  
3,  
3,  
7,  
6,  
4,  
3,  
1,  
5,  
5,  
3,  
6,  
3,  
11,  
6,  
7,  
1,  
4,  
2,  
2,  
4,  
2,  
6,  
2,  
4,  
5,  
2,

6,  
2,  
4,  
2,  
2,  
7,  
1,  
1,  
5,  
1,  
1,  
8,  
3,  
2,  
9,  
3,  
4,  
2,  
3,  
3,  
1,  
4,  
5,  
7,  
1,  
5,  
4,  
6,  
3,  
5,  
8,  
2,  
3,  
3,  
1,  
8,  
1,  
5,  
3,  
6,  
1,  
3,  
3,  
2,  
1,  
4,  
9,  
4,  
1,  
3,



4,  
4,  
7,  
2,  
1,  
4,  
1,  
7,  
1,  
7,  
3,  
5,  
2,  
3,  
5,  
1,  
3,  
7,  
3,  
3,  
6,  
3,  
4,  
1,  
4,  
2,  
2,  
5,  
1,  
1,  
6,  
1,  
1,  
3,  
4,  
7,  
2,  
1,  
2,  
7,  
4,  
7,  
4,  
4,  
1,  
4,  
2,  
8,  
5,  
1,

6,  
1,  
5,  
3,  
5,  
2,  
4,  
5,  
5,  
3,  
5,  
5,  
1,  
2,  
5,  
2,  
7,  
4,  
2,  
3,  
4,  
1,  
3,  
3,  
4,  
6,  
4,  
7,  
2,  
3,  
3,  
7,  
1,  
2,  
5,  
4,  
4,  
5,  
1,  
5,  
1,  
4,  
5,  
5,  
4,  
1,  
2,  
...]

```
In [82]: fdist = FreqDist(len(w) for w in text1)
         print(fdist)

<FreqDist with 19 samples and 260819 outcomes>
```

```
In [83]: fdist
```

```
Out[83]: FreqDist({3: 50223, 1: 47933, 4: 42345, 2: 38513, 5: 26597, 6: 17111
, 7: 14399, 8: 9966, 9: 6428, 10: 3528, ...})
```

We start by deriving a list of the lengths of words in text1, and the FreqDist then counts the number of times each of these occurs. The result is a distribution containing a quarter of a million items, each of which is a number corresponding to a word token in the text. But there are at most only 20 distinct items being counted, the numbers 1 through 20, because there are only 20 different word lengths. I.e., there are words consisting of just one character, two characters, ..., twenty characters, but none with twenty one or more characters. One might wonder how frequent the different lengths of word are (e.g., how many words of length four appear in the text, are there more words of length five than length four, etc). We can do this as follows:

```
In [84]: fdist.most_common()
```

```
Out[84]: [(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 [85]: fdist.max()
```

```
Out[85]: 3
```

```
In [86]: fdist[3]
```

```
Out[86]: 50223
```

```
In [87]: fdist.freq(3)
```

```
Out[87]: 0.19255882431878046
```

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. Although we will not pursue it here, further analysis of word length might help us understand differences between authors, genres, or languages.

Table 3.1 Functions Defined for NLTK's Frequency Distributions

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

Our discussion of frequency distributions has introduced some important Python concepts, and we will look at them systematically in 4.