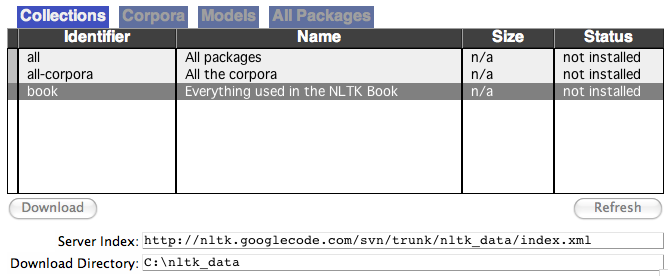
**1.2   Getting Started with NLTK**

Before going further you should install NLTK 3.0, downloadable for free from http://nltk.org/. Follow the instructions there to download the version required for your platform.

Once you've installed NLTK, start up the Python interpreter as before, and install the data required for the book by typing the following two commands at the Python prompt, then selecting the book collection as shown in [1.1](https://www.nltk.org/book/ch01.html" \l "fig-nltk-downloader).

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | **>>> import nltk**  **>>> nltk.download()** | |



***Figure 1.1****: Downloading the NLTK Book Collection: browse the available packages using nltk.download(). The****Collections****tab on the downloader shows how the packages are grouped into sets, and you should select the line labeled****book****to obtain all data required for the examples and exercises in this book. It consists of about 30 compressed files requiring about 100Mb disk space. The full collection of data (i.e.,****all****in the downloader) is nearly ten times this size (at the time of writing) and continues to expand.*

Once the data is downloaded to your machine, you can load some of it using the Python interpreter. The first step is to type a special command at the Python prompt which tells the interpreter to load some texts for us to explore: from nltk.book import \*. This says "from NLTK's book module, load all items." The book module contains all the data you will need as you read this chapter. After printing a welcome message, it loads the text of several books (this will take a few seconds). Here's the command again, together with the output that you will see. Take care to get spelling and punctuation right, and remember that you don't type the >>>.

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

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

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | **>>> text1**  **<Text: Moby Dick by Herman Melville 1851>**  **>>> text2**  **<Text: Sense and Sensibility by Jane Austen 1811>**  **>>>** | |

Now that we can use the Python interpreter, and have some data to work with, we're ready to get started.

**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:

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | **>>> text1.concordance("monstrous")**  **Displaying 11 of 11 matches:**  **ong the former , one was of a most monstrous size . ... This came towards us ,**  **ON OF THE PSALMS . " Touching that monstrous bulk of the whale or ork we have r**  **ll over with a heathenish array of monstrous clubs and spears . Some were thick**  **d as you gazed , and wondered what monstrous cannibal and savage could ever hav**  **that has survived the flood ; most monstrous and most mountainous ! That Himmal**  **they might scout at Moby Dick as a monstrous fable , or still worse and more de**  **th of Radney .'" CHAPTER 55 Of the monstrous Pictures of Whales . I shall ere l**  **ing Scenes . In connexion with the monstrous pictures of whales , I am strongly**  **ere to enter upon those still more monstrous stories of them which are to be fo**  **ght have been rummaged out of this monstrous cabinet there is no telling . But**  **of Whale - Bones ; for Whales of a monstrous size are oftentimes cast up dead u**  **>>>** | |

The first time you use a concordance on a particular text, it takes a few extra seconds to build an index so that subsequent searches are fast.

**Note**

**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!)

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:

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | **>>> text1.similar("monstrous")**  **mean part maddens doleful gamesome subtly uncommon careful untoward**  **exasperate loving passing mouldy christian few true mystifying**  **imperial modifies contemptible**  **>>> text2.similar("monstrous")**  **very heartily so exceedingly remarkably as vast a great amazingly**  **extremely good sweet**  **>>>** | |

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:

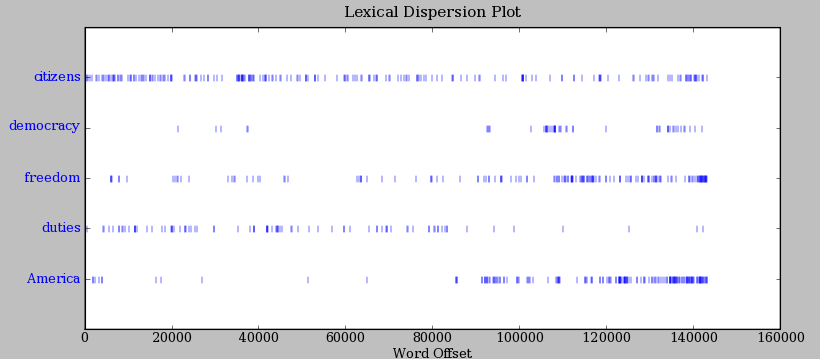
|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | **>>> text2.common\_contexts(["monstrous", "very"])**  **a\_pretty is\_pretty am\_glad be\_glad a\_lucky**  **>>>** | |

**Note**

**Your Turn:** Pick another pair of words and compare their usage in two different texts, using the similar() and common\_contexts() functions.

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" \l "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.

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | **>>> text4.dispersion\_plot(["citizens", "democracy", "freedom", "duties", "America"])**  **>>>** | |



***Figure 1.2****: Lexical Dispersion Plot for Words in U.S. Presidential Inaugural Addresses: This can be used to investigate changes in language use over time.*

**Note**

**Important:** You need to have Python's NumPy and Matplotlib packages installed in order to produce the graphical plots used in this book. Please see http://nltk.org/ for installation instructions.

**Note**

You can also plot the frequency of word usage through time using https://books.google.com/ngrams

Now, just for fun, let's try generating some random text in the various styles we have just seen. To do this, we type the name of the text followed by the term generate. (We need to include the parentheses, but there's nothing that goes between them.)

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | **>>> text3.generate()**  **In the beginning of his brother is a hairy man , whose top may reach**  **unto heaven ; and ye shall sow the land of Egypt there was no bread in**  **all that he was taken out of the month , upon the earth . So shall thy**  **wages be ? And they made their father ; and Isaac was old , and kissed**  **him : and Laban with his cattle in the midst of the hands of Esau thy**  **first born , and Phichol the chief butler unto his son Isaac , she**  **>>>** | |

**Note**

The generate() method is not available in NLTK 3.0 but will be reinstated in a subsequent version.

**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:

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | **>>> len(text3)**  **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:

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | **>>> sorted(set(text3))** **[[1]](https://www.nltk.org/book/ch01.html#ref-sorted-set)**  **['!', "'", '(', ')', ',', ',)', '.', '.)', ':', ';', ';)', '?', '?)',**  **'A', 'Abel', 'Abelmizraim', 'Abidah', 'Abide', 'Abimael', 'Abimelech',**  **'Abr', 'Abrah', 'Abraham', 'Abram', 'Accad', 'Achbor', 'Adah', ...]**  **>>> len(set(text3))** **[[2]](https://www.nltk.org/book/ch01.html#ref-len-set)**  **2789**  **>>>** | |

By wrapping sorted() around the Python expression set(text3) [[1]](https://www.nltk.org/book/ch01.html#sorted-set), 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 [[2]](https://www.nltk.org/book/ch01.html#len-set). Although it has 44,764 tokens, this book has only 2,789 distinct words, or "word types." 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 6 times on average (remember if you're using Python 2, to start with from \_\_future\_\_ import division).

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | **>>> len(set(text3)) / len(text3)**  **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:

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | **>>> text3.count("smote")**  **5**  **>>> 100 \* text4.count('a') / len(text4)**  **1.4643016433938312**  **>>>** | |

**Note**

**Your Turn:** 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?

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():

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | **>>> def lexical\_diversity(text):** **[[1]](https://www.nltk.org/book/ch01.html#ref-fun-parameter1)**  **... return len(set(text)) / len(text)** **[[2]](https://www.nltk.org/book/ch01.html#ref-locvar)**  **...**  **>>> def percentage(count, total):** **[[3]](https://www.nltk.org/book/ch01.html#ref-fun-parameter2)**  **... return 100 \* count / total**  **...** | |

**Caution!**

The Python interpreter changes the prompt from >>> to ... after encountering the colon at the end of the first line. The ... prompt indicates that Python expects an **indented code block** to appear next. It is up to you to do the indentation, by typing four spaces or hitting the tab key. To finish the indented block just enter a blank line.

In the definition of lexical\_diversity() [[1]](https://www.nltk.org/book/ch01.html#fun-parameter1), 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 [[2]](https://www.nltk.org/book/ch01.html#locvar). Similarly, percentage() is defined to take two parameters, named count and total [[3]](https://www.nltk.org/book/ch01.html#fun-parameter2).

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

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | **>>> lexical\_diversity(text3)**  **0.06230453042623537**  **>>> lexical\_diversity(text5)**  **0.13477005109975562**  **>>> percentage(4, 5)**  **80.0**  **>>> percentage(text4.count('a'), len(text4))**  **1.4643016433938312**  **>>>** | |

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" \l "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.

***Table 1.1****:*

Lexical Diversity of Various Genres in the *Brown Corpus*

| **Genre** | **Tokens** | **Types** | **Lexical diversity** |
| --- | --- | --- | --- |
| skill and hobbies | 82345 | 11935 | 0.145 |
| humor | 21695 | 5017 | 0.231 |
| fiction: science | 14470 | 3233 | 0.223 |
| press: reportage | 100554 | 14394 | 0.143 |
| fiction: romance | 70022 | 8452 | 0.121 |
| religion | 39399 | 6373 | 0.162 |

**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*:

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | **>>> sent1 = ['Call', 'me', 'Ishmael', '.']**  **>>>** | |

After the prompt we've given a name we made up, sent1, followed by the equals sign, and then some quoted words, separated with commas, and surrounded with brackets. This bracketed material is known as a **list** in Python: it is how we store a text. We can inspect it by typing the name [[1]](https://www.nltk.org/book/ch01.html#inspect-var). We can ask for its length [[2]](https://www.nltk.org/book/ch01.html#len-sent). We can even apply our own lexical\_diversity() function to it [[3]](https://www.nltk.org/book/ch01.html#apply-function).

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | **>>> sent1** **[[1]](https://www.nltk.org/book/ch01.html#ref-inspect-var)**  **['Call', 'me', 'Ishmael', '.']**  **>>> len(sent1)** **[[2]](https://www.nltk.org/book/ch01.html#ref-len-sent)**  **4**  **>>> lexical\_diversity(sent1)** **[[3]](https://www.nltk.org/book/ch01.html#ref-apply-function)**  **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 \*).

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | **>>> sent2**  **['The', 'family', 'of', 'Dashwood', 'had', 'long',**  **'been', 'settled', 'in', 'Sussex', '.']**  **>>> sent3**  **['In', 'the', 'beginning', 'God', 'created', 'the',**  **'heaven', 'and', 'the', 'earth', '.']**  **>>>** | |

**Note**

**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](https://www.nltk.org/book/ch01.html#sec-computing-with-language-texts-and-words), e.g., sorted(ex1), len(set(ex1)), ex1.count('the').

A pleasant surprise is that we can use Python's addition operator on lists. Adding two lists [[1]](https://www.nltk.org/book/ch01.html#list-plus-list) creates a new list with everything from the first list, followed by everything from the second list:

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | **>>> ['Monty', 'Python'] + ['and', 'the', 'Holy', 'Grail']** **[[1]](https://www.nltk.org/book/ch01.html#ref-list-plus-list)**  **['Monty', 'Python', 'and', 'the', 'Holy', 'Grail']**  **>>>** | |

**Note**

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.

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | **>>> sent4 + sent1**  **['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.

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | **>>> sent1.append("Some")**  **>>> sent1**  **['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:

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | **>>> text4[173]**  **'awaken'**  **>>>** | |

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

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | **>>> text4.index('awaken')**  **173**  **>>>** | |

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

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | **>>> text5[16715:16735]**  **['U86', 'thats', 'why', 'something', 'like', 'gamefly', 'is', 'so', 'good',**  **'because', 'you', 'can', 'actually', 'play', 'a', 'full', 'game', 'without',**  **'buying', 'it']**  **>>> text6[1600:1625]**  **['We', "'", 're', 'an', 'anarcho', '-', 'syndicalist', 'commune', '.', 'We',**  **'take', 'it', 'in', 'turns', 'to', 'act', 'as', 'a', 'sort', 'of', 'executive',**  **'officer', 'for', 'the', 'week']**  **>>>** | |

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

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | **>>> sent = ['word1', 'word2', 'word3', 'word4', 'word5',**  **... 'word6', 'word7', 'word8', 'word9', 'word10']**  **>>> sent[0]**  **'word1'**  **>>> sent[9]**  **'word10'**  **>>>** | |

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:

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | **>>> sent[10]**  **Traceback (most recent call last):**  **File "<stdin>", line 1, in ?**  **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:

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | **>>> sent[5:8]**  **['word6', 'word7', 'word8']**  **>>> sent[5]**  **'word6'**  **>>> sent[6]**  **'word7'**  **>>> sent[7]**  **'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 [[1]](https://www.nltk.org/book/ch01.html#slice2), and we can omit the second number if the slice goes to the end [[2]](https://www.nltk.org/book/ch01.html#slice3):

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | **>>> sent[:3]** **[[1]](https://www.nltk.org/book/ch01.html#ref-slice2)**  **['word1', 'word2', 'word3']**  **>>> text2[141525:]** **[[2]](https://www.nltk.org/book/ch01.html#ref-slice3)**  **['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 [[1]](https://www.nltk.org/book/ch01.html#list-assignment). We can also replace an entire slice with new material [[2]](https://www.nltk.org/book/ch01.html#slice-assignment). A consequence of this last change is that the list only has four elements, and accessing a later value generates an error [[3]](https://www.nltk.org/book/ch01.html#list-error).

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | **>>> sent[0] = 'First'** **[[1]](https://www.nltk.org/book/ch01.html#ref-list-assignment)**  **>>> sent[9] = 'Last'**  **>>> len(sent)**  **10**  **>>> sent[1:9] = ['Second', 'Third']** **[[2]](https://www.nltk.org/book/ch01.html#ref-slice-assignment)**  **>>> sent**  **['First', 'Second', 'Third', 'Last']**  **>>> sent[9]** **[[3]](https://www.nltk.org/book/ch01.html#ref-list-error)**  **Traceback (most recent call last):**  **File "<stdin>", line 1, in ?**  **IndexError: list index out of range**  **>>>** | |

**Note**

**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.

**2.3   Variables**

From the start of [1](https://www.nltk.org/book/ch01.html#sec-computing-with-language-texts-and-words), 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:

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | **>>> 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, xyzzy. It must start with a letter, and can include numbers and underscores. Here are some examples of variables and assignments:

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | **>>> my\_sent = ['Bravely', 'bold', 'Sir', 'Robin', ',', 'rode',**  **... 'forth', 'from', 'Camelot', '.']**  **>>> noun\_phrase = my\_sent[1:4]**  **>>> noun\_phrase**  **['bold', 'Sir', 'Robin']**  **>>> wOrDs = sorted(noun\_phrase)**  **>>> wOrDs**  **['Robin', 'Sir', 'bold']**  **>>>** | |

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

**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:

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | **>>> not = 'Camelot'**  **File "<stdin>", line 1**  **not = 'Camelot'**  **^**  **SyntaxError: invalid syntax**  **>>>** | |

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:

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | **>>> vocab = set(text1)**  **>>> vocab\_size = len(vocab)**  **>>> vocab\_size**  **19317**  **>>>** | |

**Caution!**

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

**2.4   Strings**

Some of the methods we used to access the elements of a list also work with individual words, or **strings**. For example, we can assign a string to a variable [[1]](https://www.nltk.org/book/ch01.html#assign-string), index a string [[2]](https://www.nltk.org/book/ch01.html#index-string), and slice a string [[3]](https://www.nltk.org/book/ch01.html#slice-string):

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | **>>> name = 'Monty'** **[[1]](https://www.nltk.org/book/ch01.html#ref-assign-string)**  **>>> name[0]** **[[2]](https://www.nltk.org/book/ch01.html#ref-index-string)**  **'M'**  **>>> name[:4]** **[[3]](https://www.nltk.org/book/ch01.html#ref-slice-string)**  **'Mont'**  **>>>** | |

We can also perform multiplication and addition with strings:

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | **>>> name \* 2**  **'MontyMonty'**  **>>> name + '!'**  **'Monty!'**  **>>>** | |

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

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | **>>> ' '.join(['Monty', 'Python'])**  **'Monty Python'**  **>>> 'Monty Python'.split()**  **['Monty', 'Python']**  **>>>** | |

We will come back to the topic of strings in [3](https://www.nltk.org/book/ch03.html#chap-words). For the time being, we have two important building blocks — lists and strings — and are ready to get back to some language analysis.

## Variables

From the start of [1](https://www.nltk.org/book/ch01.html#sec-computing-with-language-texts-and-words), 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:

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | **>>> 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, xyzzy. It must start with a letter, and can include numbers and underscores. Here are some examples of variables and assignments:

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | **>>> my\_sent = ['Bravely', 'bold', 'Sir', 'Robin', ',', 'rode',**  **... 'forth', 'from', 'Camelot', '.']**  **>>> noun\_phrase = my\_sent[1:4]**  **>>> noun\_phrase**  **['bold', 'Sir', 'Robin']**  **>>> wOrDs = sorted(noun\_phrase)**  **>>> wOrDs**  **['Robin', 'Sir', 'bold']**  **>>>** | |

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

**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:

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | **>>> not = 'Camelot'**  **File "<stdin>", line 1**  **not = 'Camelot'**  **^**  **SyntaxError: invalid syntax**  **>>>** | |

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:

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | **>>> vocab = set(text1)**  **>>> vocab\_size = len(vocab)**  **>>> vocab\_size**  **19317**  **>>>** | |

**Caution!**

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

## 2.4   Strings

Some of the methods we used to access the elements of a list also work with individual words, or **strings**. For example, we can assign a string to a variable [[1]](https://www.nltk.org/book/ch01.html#assign-string), index a string [[2]](https://www.nltk.org/book/ch01.html#index-string), and slice a string [[3]](https://www.nltk.org/book/ch01.html#slice-string):

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | **>>> name = 'Monty' [[1]](https://www.nltk.org/book/ch01.html#ref-assign-string)**  **>>> name[0] [[2]](https://www.nltk.org/book/ch01.html#ref-index-string)**  **'M'**  **>>> name[:4] [[3]](https://www.nltk.org/book/ch01.html#ref-slice-string)**  **'Mont'**  **>>>** | |

We can also perform multiplication and addition with strings:

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | **>>> name \* 2**  **'MontyMonty'**  **>>> name + '!'**  **'Monty!'**  **>>>** | |

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

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | **>>> ' '.join(['Monty', 'Python'])**  **'Monty Python'**  **>>> 'Monty Python'.split()**  **['Monty', 'Python']**  **>>>** | |

We will come back to the topic of strings in [3](https://www.nltk.org/book/ch03.html#chap-words). 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), 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.

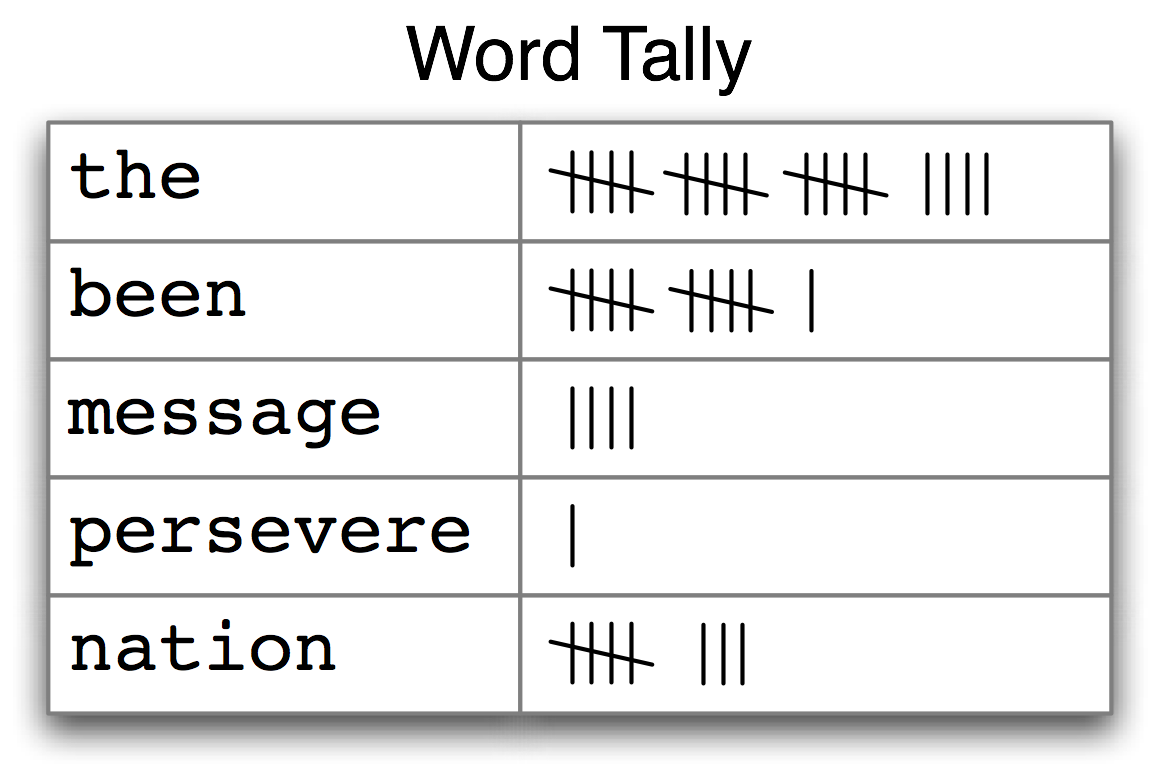
In this section we pick up the question of what makes a text distinct, and use automatic methods to find characteristic words and expressions of a text. As in [1](https://www.nltk.org/book/ch01.html#sec-computing-with-language-texts-and-words), you can try new features of the Python language by copying 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.

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | **>>> saying = ['After', 'all', 'is', 'said', 'and', 'done',**  **... 'more', 'is', 'said', 'than', 'done']**  **>>> tokens = set(saying)**  **>>> tokens = sorted(tokens)**  **>>> tokens[-2:]**  **what output do you expect here?**  **>>>** | |

## 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). 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.



***Figure 3.1****: Counting Words Appearing in a Text (a frequency distribution)*

The table in [3.1](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:

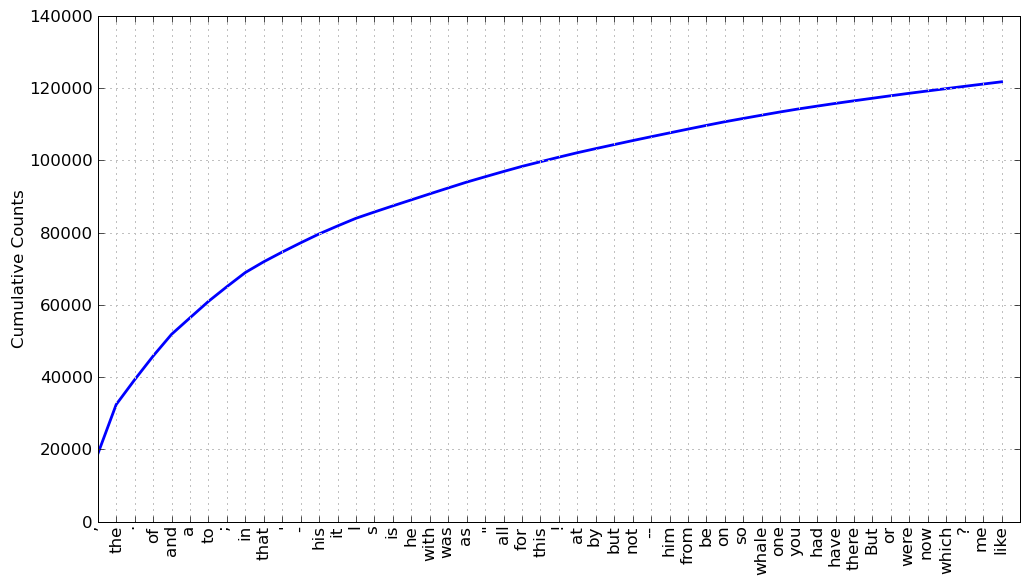
|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | **>>> fdist1 = FreqDist(text1)** **[[1]](https://www.nltk.org/book/ch01.html#ref-freq-dist-call)**  **>>> print(fdist1)** **[[2]](https://www.nltk.org/book/ch01.html#ref-freq-dist-inspect)**  **<FreqDist with 19317 samples and 260819 outcomes>**  **>>> fdist1.most\_common(50)** **[[3]](https://www.nltk.org/book/ch01.html#ref-freq-dist-most-common)**  **[(',', 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)]**  **>>> fdist1['whale']**  **906**  **>>>** | |

When we first invoke FreqDist, we pass the name of the text as an argument [[1]](https://www.nltk.org/book/ch01.html#freq-dist-call). We can inspect the total number of words ("outcomes") that have been counted up [[2]](https://www.nltk.org/book/ch01.html#freq-dist-inspect) — 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 [[3]](https://www.nltk.org/book/ch01.html#freq-dist-most-common).

**Note**

**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 \*

Do any words produced in the last example help us grasp the topic or genre of this text? Only one word, *whale*, is slightly informative! It occurs over 900 times. The rest of the words tell us nothing about the text; they're just English "plumbing." What proportion of the text is taken up with such words? We can generate a cumulative frequency plot for these words, using fdist1.plot(50, cumulative=True), to produce the graph in [3.2](https://www.nltk.org/book/ch01.html#fig-fdist-moby). These 50 words account for nearly half the book!



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