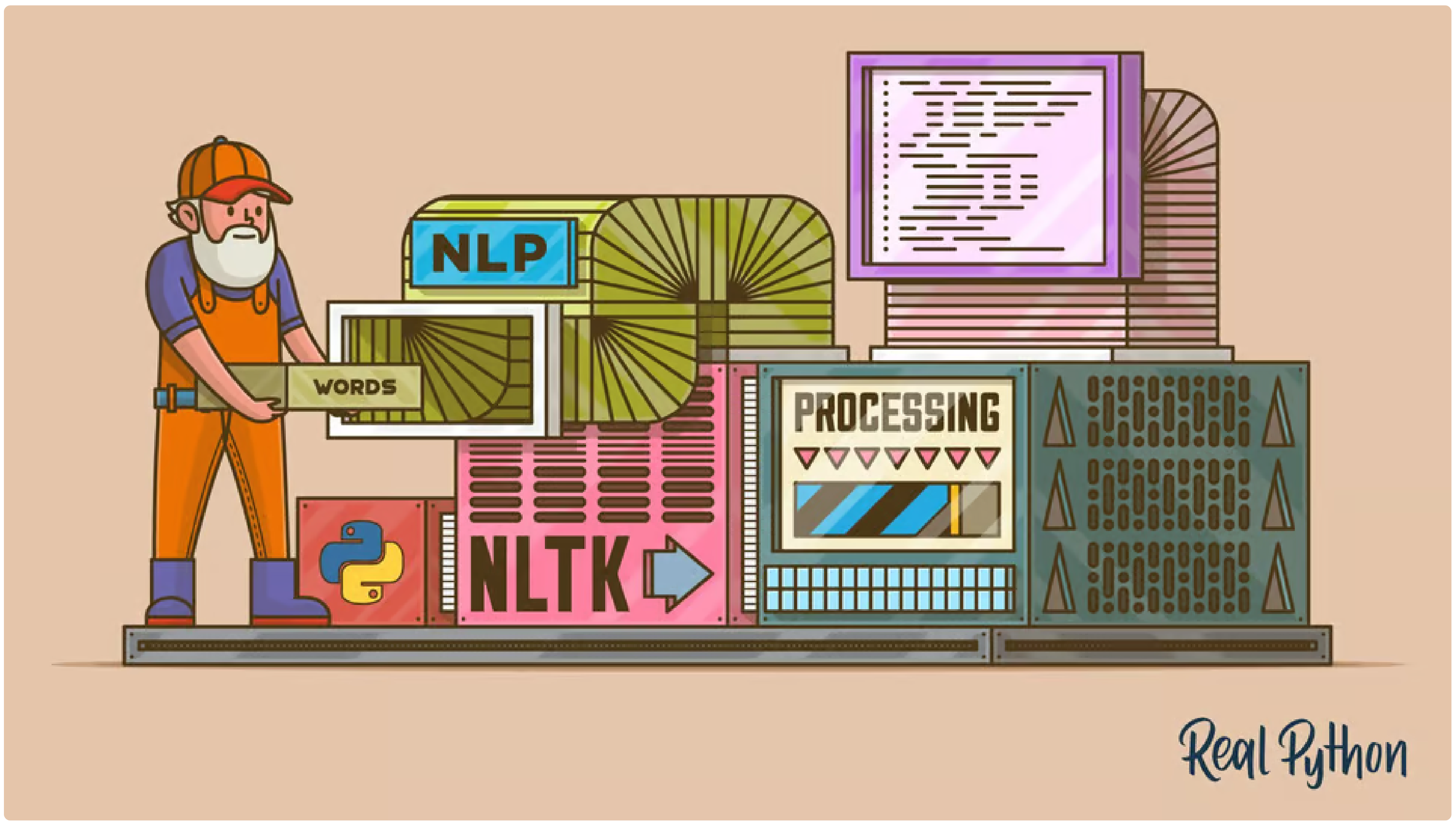


Natural Language Processing With Python's NLTK Package

by Joanna Jablonski 5 Comments

basics data-science

Mark as Completed

Tweet Share Email

Table of Contents

Natural language processing (NLP) is a field that focuses on making natural human language usable by computer programs. **NLTK**, or Natural Language Toolkit, is a Python package that you can use for NLP.

A lot of the data that you could be analyzing is unstructured data and contains human-readable text. Before you can analyze that data programmatically, you first need to preprocess it. In this tutorial, you'll take your first look at the kinds of **text preprocessing** tasks you can do with NLTK so that you'll be ready to apply them in future projects. You'll also see how to do some basic **text analysis** and create **visualizations**.

If you're familiar with the basics of using Python and would like to get your feet wet with some NLP, then you've come to the right place.

By the end of this tutorial, you'll know how to:

- **Find text** to analyze
- **Preprocess** your text for analysis
- **Analyze** your text
- Create **visualizations** based on your analysis

Let's get Pythoning!

Getting Started With Python's NLTK

The first thing you need to do is make sure that you have Python installed. For this tutorial, you'll be using Python 3.9. If you don't yet have Python installed, then check out Python 3 Installation & Setup Guide to get started.

Once you have that dealt with, your next step is to install NLTK with `pip`. It's a best practice to install it in a virtual environment. To learn more about virtual environments, check out Python Virtual Environments: A Primer.

For this tutorial, you'll be installing version 3.5:

Shell

```
$ python -m pip install nltk==3.5
```

In order to create visualizations for named entity recognition, you'll also need to install NumPy and Matplotlib:

Shell

```
$ python -m pip install numpy matplotlib
```

If you'd like to know more about how `pip` works, then you can check out What Is Pip? A Guide for New Pythonistas. You can also take a look at the official page on installing NLTK data.

Tokenizing

By **tokenizing**, you can conveniently split up text by word or by sentence. This will allow you to work with smaller pieces of text that are still relatively coherent and meaningful even outside of the context of the rest of the text. It's your first step in turning unstructured data into structured data, which is easier to analyze.

When you're analyzing text, you'll be tokenizing by word and tokenizing by sentence. Here's what both types of tokenization bring to the table:

- **Tokenizing by word:** Words are like the atoms of natural language. They're the smallest unit of meaning that still makes sense on its own. Tokenizing your text by word allows you to identify words that come up particularly often. For example, if you were analyzing a group of job ads, then you might find that the word "Python" comes up often. That could suggest high demand for Python knowledge, but you'd need to look deeper to know more.

- **Tokenizing by sentence:** When you tokenize by sentence, you can analyze how those words relate to one another and see more context. Are there a lot of negative words around the word "Python" because the hiring manager doesn't like Python? Are there more terms from the domain of herpetology than the domain of software development, suggesting that you may be dealing with an entirely different kind of python than you were expecting?

Here's how to import the relevant parts of NLTK so you can tokenize by word and by sentence:

Python >>>

```
>>> from nltk.tokenize import sent_tokenize, word_tokenize
```

Now that you've imported what you need, you can create a string to tokenize. Here's a quote from *Dune* that you can use:

Python >>>

```
>>> example_string = """
... Muad'Dib learned rapidly because his first training was in how to learn.
... And the first lesson of all was the basic trust that he could learn.
... It's shocking to find how many people do not believe they can learn,
... and how many more believe learning to be difficult."""
```

You can use `sent_tokenize()` to split up `example_string` into sentences:

Python >>>

```
>>> sent_tokenize(example_string)
["Muad'Dib learned rapidly because his first training was in how to learn.",
'And the first lesson of all was the basic trust that he could learn.',
"It's shocking to find how many people do not believe they can learn, and how many more believe learning to b
```

Tokenizing `example_string` by sentence gives you a list of three strings that are sentences:

1. `"Muad'Dib learned rapidly because his first training was in how to learn."`
2. `'And the first lesson of all was the basic trust that he could learn.'`
3. `"It's shocking to find how many people do not believe they can learn, and how many more believe learning to be difficult."`

Now try tokenizing `example_string` by word:

Python >>>

```
>>> word_tokenize(example_string)
["Muad'Dib",
 'learned',
 'rapidly',
 'because',
 'his',
 'first',
 'training',
 'was',
 'in',
 'how',
 'to',
 'learn',
 '.',
 'And',
 'the',
 'first',
 'lesson',
 'of',
 'all',
 'was',
 'the',
 'basic',
 'trust',
 'that',
 'he',
 'could',
 'learn',
 '.',
 'It',
 "'s",
 'shocking',
 'to',
 'find',
 'how',
 'many',
 'people',
 'do',
 'not',
 'believe',
 'they',
 'can',
 'learn',
 ',',
 'and',
 'how',
 'many',
 'more',
 'believe',
 'learning',
 'to',
 'be',
 'difficult',
 '.']
```

You got a list of strings that NLTK considers to be words, such as:

- `"Muad'Dib"`
- `'training'`
- `'how'`

But the following strings were also considered to be words:

- `"'s"`
- `','`
- `'.'`

See how "It's" was split at the apostrophe to give you 'It' and "'s", but "Muad'Dib" was left whole? This happened because NLTK knows that 'It' and "'s" (a contraction of "is") are two distinct words, so it counted them separately. But "Muad'Dib" isn't an accepted contraction like "It's", so it wasn't read as two separate words and was left intact.

Filtering Stop Words

Stop words are words that you want to ignore, so you filter them out of your text when you're processing it. Very common words like 'in', 'is', and 'an' are often used as stop words since they don't add a lot of meaning to a text in and of themselves.

Here's how to import the relevant parts of NLTK in order to filter out stop words:

Python >>>

```
>>> nltk.download("stopwords")
>>> from nltk.corpus import stopwords
>>> from nltk.tokenize import word_tokenize
```

Here's a quote from Worf that you can filter:

Python >>>

```
>>> worf_quote = "Sir, I protest. I am not a merry man!"
```

Now tokenize worf_quote by word and store the resulting list in words_in_quote:

Python >>>

```
>>> words_in_quote = word_tokenize(worf_quote)
>>> words_in_quote
['Sir', ',', 'protest', '.', 'merry', 'man', '!']
```

You have a list of the words in worf_quote, so the next step is to create a set of stop words to filter words_in_quote. For this example, you'll need to focus on stop words in "english":

Python >>>

```
>>> stop_words = set(stopwords.words("english"))
```

Next, create an empty list to hold the words that make it past the filter:

Python >>>

```
>>> filtered_list = []
```

You created an empty list, filtered_list, to hold all the words in words_in_quote that aren't stop words. Now you can use stop_words to filter words_in_quote:

Python >>>

```
>>> for word in words_in_quote:
...    if word.casefold() not in stop_words:
...         filtered_list.append(word)
```

You iterated over words_in_quote with a for loop and added all the words that weren't stop words to filtered_list. You used .casefold() on word so you could ignore whether the letters in word were uppercase or lowercase. This is worth doing because stopwords.words('english') includes only lowercase versions of stop words.

Alternatively, you could use a list comprehension to make a list of all the words in your text that aren't stop words:

Python >>>

```
>>> filtered_list = [
...     word for word in words_in_quote if word.casefold() not in stop_words
... ]
```

When you use a list comprehension, you don't create an empty list and then add items to the end of it. Instead, you define the list and its contents at the same time. Using a list comprehension is often seen as more Pythonic.

Take a look at the words that ended up in `filtered_list`:

Python >>>
```
>>> filtered_list
['Sir', ',', 'protest', '.', 'merry', 'man', '!']
```

You filtered out a few words like `'am'` and `'a'`, but you also filtered out `'not'`, which does affect the overall meaning of the sentence. (Worf won't be happy about this.)

Words like `'I'` and `'not'` may seem too important to filter out, and depending on what kind of analysis you want to do, they can be. Here's why:

- `'I'` is a pronoun, which are context words rather than content words:
  - **Content words** give you information about the topics covered in the text or the sentiment that the author has about those topics.
  - **Context words** give you information about writing style. You can observe patterns in how authors use context words in order to quantify their writing style. Once you've quantified their writing style, you can analyze a text written by an unknown author to see how closely it follows a particular writing style so you can try to identify who the author is.
- `'not'` is technically an adverb but has still been included in NLTK's list of stop words for English. If you want to edit the list of stop words to exclude `'not'` or make other changes, then you can download it.

So, `'I'` and `'not'` can be important parts of a sentence, but it depends on what you're trying to learn from that sentence.

Stemming

Stemming is a text processing task in which you reduce words to their root, which is the core part of a word. For example, the words "helping" and "helper" share the root "help." Stemming allows you to zero in on the basic meaning of a word rather than all the details of how it's being used. NLTK has more than one stemmer, but you'll be using the Porter stemmer.

Here's how to import the relevant parts of NLTK in order to start stemming:

Python >>>
```
>>> from nltk.stem import PorterStemmer
>>> from nltk.tokenize import word_tokenize
```

Now that you're done importing, you can create a stemmer with `PorterStemmer()`:

Python >>>
```
>>> stemmer = PorterStemmer()
```

The next step is for you to create a string to stem. Here's one you can use:

Python >>>
```
>>> string_for_stemming = """
... The crew of the USS Discovery discovered many discoveries.
... Discovering is what explorers do."""
```

Before you can stem the words in that string, you need to separate all the words in it:

Python >>>

```python
>>> words = word_tokenize(string_for_stemming)
```

Now that you have a list of all the tokenized words from the string, take a look at what's in `words`:

Python >>>

```python
>>> words
['The',
 'crew',
 'of',
 'the',
 'USS',
 'Discovery',
 'discovered',
 'many',
 'discoveries',
 '.',
 'Discovering',
 'is',
 'what',
 'explorers',
 'do',
 '.']
```

Create a list of the stemmed versions of the words in `words` by using `stemmer.stem()` in a list comprehension:

Python >>>

```python
>>> stemmed_words = [stemmer.stem(word) for word in words]
```

Take a look at what's in `stemmed_words`:

Python >>>

```python
>>> stemmed_words
['the',
 'crew',
 'of',
 'the',
 'uss',
 'discoveri',
 'discov',
 'mani',
 'discoveri',
 '.',
 'discov',
 'is',
 'what',
 'explor',
 'do',
 '.']
```

Here's what happened to all the words that started with `'discov'` or `'Discov'`:

Original word	Stemmed version
`'Discovery'`	`'discoveri'`
`'discovered'`	`'discov'`
`'discoveries'`	`'discoveri'`
`'Discovering'`	`'discov'`

Those results look a little inconsistent. Why would `'Discovery'` give you `'discoveri'` when `'Discovering'` gives you `'discov'`?

Understemming and overstemming are two ways stemming can go wrong:

1. **Understemming** happens when two related words should be reduced to the same stem but aren't. This is a false negative.
2. **Overstemming** happens when two unrelated words are reduced to the same stem even though they shouldn't be. This is a false positive.

The Porter stemming algorithm dates from 1979, so it's a little on the older side. The **Snowball stemmer**, which is also called **Porter2**, is an improvement on the original and is also available through NLTK, so you can use that one in your own projects. It's also worth noting that the purpose of the Porter stemmer is not to produce complete words but to find variant forms of a word.

Fortunately, you have some other ways to reduce words to their core meaning, such as lemmatizing, which you'll see later in this tutorial. But first, we need to cover parts of speech.

Tagging Parts of Speech

Part of speech is a grammatical term that deals with the roles words play when you use them together in sentences. Tagging parts of speech, or **POS tagging**, is the task of labeling the words in your text according to their part of speech.

In English, there are eight parts of speech:

Part of speech	Role	Examples
Noun	Is a person, place, or thing	mountain, bagel, Poland
Pronoun	Replaces a noun	you, she, we
Adjective	Gives information about what a noun is like	efficient, windy, colorful
Verb	Is an action or a state of being	learn, is, go
Adverb	Gives information about a verb, an adjective, or another adverb	efficiently, always, very
Preposition	Gives information about how a noun or pronoun is connected to another word	from, about, at
Conjunction	Connects two other words or phrases	so, because, and
Interjection	Is an exclamation	yay, ow, wow

Some sources also include the category **articles** (like "a" or "the") in the list of parts of speech, but other sources consider them to be adjectives. NLTK uses the word **determiner** to refer to articles.

Here's how to import the relevant parts of NLTK in order to tag parts of speech:

Python >>>

```
>>> from nltk.tokenize import word_tokenize
```

Now create some text to tag. You can use this Carl Sagan quote:

Python >>>

```
>>> sagan_quote = """
... If you wish to make an apple pie from scratch,
... you must first invent the universe."""
```

Use `word_tokenize` to separate the words in that string and store them in a list:

Python >>>

```
>>> words_in_sagan_quote = word_tokenize(sagan_quote)
```

Now call `nltk.pos_tag()` on your new list of words:

Python >>>

```
>>> import nltk
>>> nltk.pos_tag(words_in_sagan_quote)
[('If', 'IN'),
 ('you', 'PRP'),
 ('wish', 'VBP'),
 ('to', 'TO'),
 ('make', 'VB'),
 ('an', 'DT'),
 ('apple', 'NN'),
 ('pie', 'NN'),
 ('from', 'IN'),
 ('scratch', 'NN'),
 (',', ','),
 ('you', 'PRP'),
 ('must', 'MD'),
 ('first', 'VB'),
 ('invent', 'VB'),
 ('the', 'DT'),
 ('universe', 'NN'),
 ('.', '.')]
```

All the words in the quote are now in a separate tuple, with a tag that represents their part of speech. But what do the tags mean? Here's how to get a list of tags and their meanings:

Python >>>

```
>>> nltk.help.upenn_tagset()
```

The list is quite long, but feel free to expand the box below to see it.

POS Tags and Their Meanings Show/Hide

Here's a summary that you can use to get started with NLTK's POS tags:

Tags that start with	Deal with
`JJ`	Adjectives
`NN`	Nouns
`RB`	Adverbs
`PRP`	Pronouns

Tags that start with	Deal with
VB	Verbs

Now that you know what the POS tags mean, you can see that your tagging was fairly successful:

- `'pie'` was tagged `NN` because it's a singular noun.
- `'you'` was tagged `PRP` because it's a personal pronoun.
- `'invent'` was tagged `VB` because it's the base form of a verb.

But how would NLTK handle tagging the parts of speech in a text that is basically gibberish? Jabberwocky is a nonsense poem that doesn't technically mean much but is still written in a way that can convey some kind of meaning to English speakers.

Make a string to hold an excerpt from this poem:

Python >>>

```
>>> jabberwocky_excerpt = """
... 'Twas brillig, and the slithy toves did gyre and gimble in the wabe:
... all mimsy were the borogoves, and the mome raths outgrabe."""
```

Use `word_tokenize` to separate the words in the excerpt and store them in a list:

Python >>>

```
>>> words_in_excerpt = word_tokenize(jabberwocky_excerpt)
```

Call `nltk.pos_tag()` on your new list of words:

Python >>>

```
>>> nltk.pos_tag(words_in_excerpt)
[("'T", 'NN'),
 ('was', 'VBD'),
 ('brillig', 'VBN'),
 (',', ','),
 ('and', 'CC'),
 ('the', 'DT'),
 ('slithy', 'JJ'),
 ('toves', 'NNS'),
 ('did', 'VBD'),
 ('gyre', 'NN'),
 ('and', 'CC'),
 ('gimble', 'JJ'),
 ('in', 'IN'),
 ('the', 'DT'),
 ('wabe', 'NN'),
 (':', ':'),
 ('all', 'DT'),
 ('mimsy', 'NNS'),
 ('were', 'VBD'),
 ('the', 'DT'),
 ('borogoves', 'NNS'),
 (',', ','),
 ('and', 'CC'),
 ('the', 'DT'),
 ('mome', 'JJ'),
 ('raths', 'NNS'),
 ('outgrabe', 'RB'),
 ('.', '.')]
```

Accepted English words like `'and'` and `'the'` were correctly tagged as a conjunction and a determiner, respectively. The gibberish word `'slithy'` was tagged as an adjective, which is what a human English speaker would probably assume from the context of the poem as well. Way to go, NLTK!

Lemmatizing

Now that you're up to speed on parts of speech, you can circle back to lemmatizing. Like stemming, **lemmatizing** reduces words to their core meaning, but it will give you a complete English word that makes sense on its own instead of just a fragment of a word like `'discoveri'`.

Note: A **lemma** is a word that represents a whole group of words, and that group of words is called a **lexeme**.

For example, if you were to look up the word "blending" in a dictionary, then you'd need to look at the entry for "blend," but you would find "blending" listed in that entry.

In this example, "blend" is the **lemma**, and "blending" is part of the **lexeme**. So when you lemmatize a word, you are reducing it to its lemma.

Here's how to import the relevant parts of NLTK in order to start lemmatizing:

Python

```
>>> from nltk.stem import WordNetLemmatizer
```

Create a lemmatizer to use:

Python

```
>>> lemmatizer = WordNetLemmatizer()
```

Let's start with lemmatizing a plural noun:

Python

```
>>> lemmatizer.lemmatize("scarves")
'scarf'
```

`"scarves"` gave you `'scarf'`, so that's already a bit more sophisticated than what you would have gotten with the Porter stemmer, which is `'scarv'`. Next, create a string with more than one word to lemmatize:

Python

```
>>> string_for_lemmatizing = "The friends of DeSoto love scarves."
```

Now tokenize that string by word:

Python

```
>>> words = word_tokenize(string_for_lemmatizing)
```

Here's your list of words:

Python

```
>>> words
['The',
 'friends',
 'of',
 'DeSoto',
 'love'
 'scarves',
 '.']
```

Create a list containing all the words in `words` after they've been lemmatized:

Python >>>

```
>>> lemmatized_words = [lemmatizer.lemmatize(word) for word in words]
```

Here's the list you got:

Python >>>

```
>>> lemmatized_words
['The',
 'friend',
 'of',
 'DeSoto',
 'love',
 'scarf',
 '.'
```

That looks right. The plurals `'friends'` and `'scarves'` became the singulars `'friend'` and `'scarf'`.

But what would happen if you lemmatized a word that looked very different from its lemma? Try lemmatizing `"worst"`:

Python >>>

```
>>> lemmatizer.lemmatize("worst")
'worst'
```

You got the result `'worst'` because `lemmatizer.lemmatize()` assumed that [`"worst"` was a noun](https://www.merriam-webster.com/dictionary/worst). You can make it clear that you want `"worst"` to be an adjective:

Python >>>

```
>>> lemmatizer.lemmatize("worst", pos="a")
'bad'
```

The default parameter for `pos` is `'n'` for noun, but you made sure that `"worst"` was treated as an adjective by adding the parameter `pos="a"`. As a result, you got `'bad'`, which looks very different from your original word and is nothing like what you'd get if you were stemming. This is because `"worst"` is the superlative form of the adjective `'bad'`, and lemmatizing reduces superlatives as well as comparatives to their lemmas.

Now that you know how to use NLTK to tag parts of speech, you can try tagging your words before lemmatizing them to avoid mixing up homographs, or words that are spelled the same but have different meanings and can be different parts of speech.

Chunking

While tokenizing allows you to identify words and sentences, **chunking** allows you to identify **phrases**.

Note: A **phrase** is a word or group of words that works as a single unit to perform a grammatical function. **Noun phrases** are built around a noun.

Here are some examples:

- "A planet"
- "A tilting planet"
- "A swiftly tilting planet"

Chunking makes use of POS tags to group words and apply chunk tags to those groups. Chunks don't overlap, so one instance of a word can be in only one chunk at a time.

Here's how to import the relevant parts of NLTK in order to chunk:

Python >>>
```
>>> from nltk.tokenize import word_tokenize
```

Before you can chunk, you need to make sure that the parts of speech in your text are tagged, so create a string for POS tagging. You can use this quote from *The Lord of the Rings*:

Python >>>
```
>>> lotr_quote = "It's a dangerous business, Frodo, going out your door."
```

Now tokenize that string by word:

Python >>>
```
>>> words_in_lotr_quote = word_tokenize(lotr_quote)
>>> words_in_lotr_quote
['It',
 "'s",
 'a',
 'dangerous',
 'business',
 ',',
 'Frodo',
 ',',
 'going',
 'out',
 'your',
 'door',
 '.']
```

Now you've got a list of all of the words in `lotr_quote`.

The next step is to tag those words by part of speech:

Python >>>
```
>>> nltk.download("averaged_perceptron_tagger")
>>> lotr_pos_tags = nltk.pos_tag(words_in_lotr_quote)
>>> lotr_pos_tags
[('It', 'PRP'),
 ("'s", 'VBZ'),
 ('a', 'DT'),
 ('dangerous', 'JJ'),
 ('business', 'NN'),
 (',', ','),
 ('Frodo', 'NNP'),
 (',', ','),
 ('going', 'VBG'),
 ('out', 'RP'),
 ('your', 'PRP$'),
 ('door', 'NN'),
 ('.', '.')]
```

You've got a list of tuples of all the words in the quote, along with their POS tag. In order to chunk, you first need to define a chunk grammar.

Note: A **chunk grammar** is a combination of rules on how sentences should be chunked. It often uses regular expressions, or **regexes**.

For this tutorial, you don't need to know how regular expressions work, but they will definitely come in handy for you in the future if you want to process text.

Create a chunk grammar with one regular expression rule:

Python >>>

```
>>> grammar = "NP: {<DT>?<JJ>*<NN>}"
```

`NP` stands for noun phrase. You can learn more about **noun phrase chunking** in Chapter 7 of *Natural Language Processing with Python—Analyzing Text with the Natural Language Toolkit*.

According to the rule you created, your chunks:

1. Start with an optional (`?`) determiner (`'DT'`)
2. Can have any number (`*`) of adjectives (`JJ`)
3. End with a noun (`<NN>`)

Create a **chunk parser** with this grammar:

Python >>>

```
>>> chunk_parser = nltk.RegexpParser(grammar)
```

Now try it out with your quote:

Python >>>

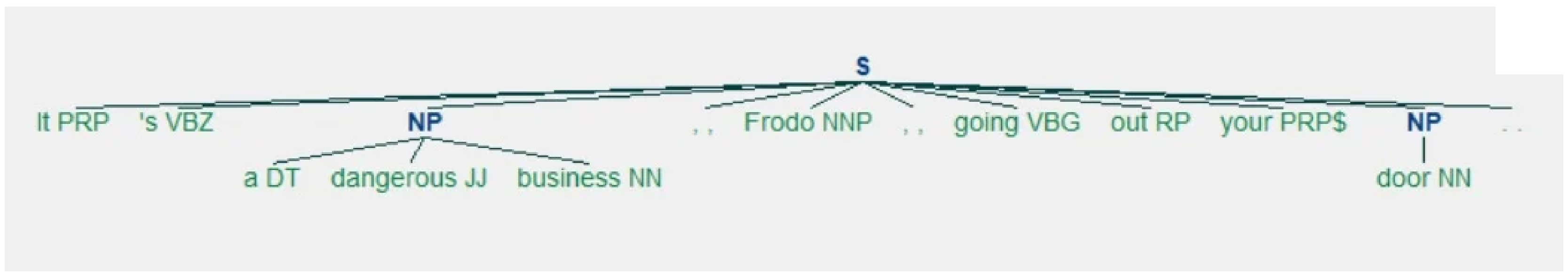
```
>>> tree = chunk_parser.parse(lotr_pos_tags)
```

Here's how you can see a visual representation of this tree:

Python >>>

```
>>> tree.draw()
```

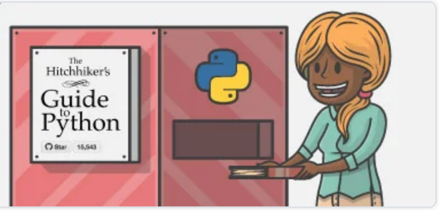
This is what the visual representation looks like:



You got two noun phrases:

1. `'a dangerous business'` has a determiner, an adjective, and a noun.
2. `'door'` has just a noun.

Now that you know about chunking, it's time to look at chinking.



Chinking

Chinking is used together with chunking, but while chunking is used to include a pattern, **chinking** is used to exclude a pattern.

Let's reuse the quote you used in the section on chunking. You already have a list of tuples containing each of the words in the quote along with its part of speech tag:

Python >>>

```
>>> lotr_pos_tags
[('It', 'PRP'),
 ("'s", 'VBZ'),
 ('a', 'DT'),
 ('dangerous', 'JJ'),
 ('business', 'NN'),
 (',', ','),
 ('Frodo', 'NNP'),
 (',', ','),
 ('going', 'VBG'),
 ('out', 'RP'),
 ('your', 'PRP$'),
 ('door', 'NN'),
 ('.', '.')]
```

The next step is to create a grammar to determine what you want to include and exclude in your chunks. This time, you're going to use more than one line because you're going to have more than one rule. Because you're using more than one line for the grammar, you'll be using triple quotes (`"""`):

Python >>>

```
>>> grammar = """
... Chunk: {<.*>+}
...        }<JJ>{"""
```

The first rule of your grammar is `{<.*>+}`. This rule has curly braces that face inward (`{}`) because it's used to determine what patterns you want to include in you chunks. In this case, you want to include everything: `<.*>+`.

The second rule of your grammar is `}<JJ>{`. This rule has curly braces that face outward (`}{`) because it's used to determine what patterns you want to exclude in your chunks. In this case, you want to exclude adjectives: `<JJ>`.

Create a chunk parser with this grammar:

Python >>>

```
>>> chunk_parser = nltk.RegexpParser(grammar)
```

Now chunk your sentence with the chink you specified:

Python >>>

```
>>> tree = chunk_parser.parse(lotr_pos_tags)
```

You get this tree as a result:

Python >>>

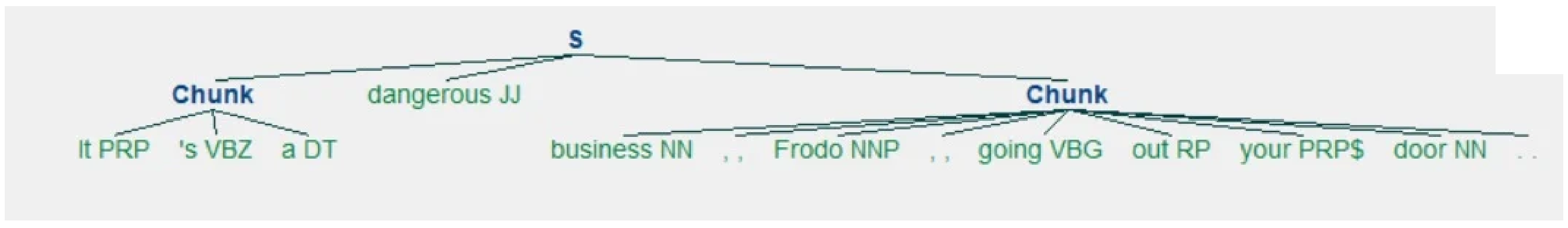
```
>>> tree
Tree('S', [Tree('Chunk', [('It', 'PRP'), ("'s", 'VBZ'), ('a', 'DT')]), ('dangerous', 'JJ'), Tree('Chunk', [('
```

In this case, `('dangerous', 'JJ')` was excluded from the chunks because it's an adjective (`JJ`). But that will be easier to see if you get a graphic representation again:

Python >>>

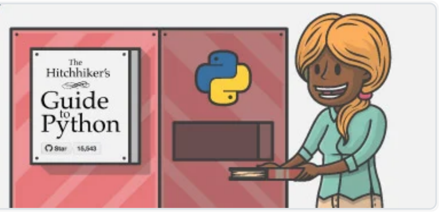
```
>>> tree.draw()
```

You get this visual representation of the `tree`:



Here, you've excluded the adjective `'dangerous'` from your chunks and are left with two chunks containing everything else. The first chunk has all the text that appeared before the adjective that was excluded. The second chunk contains everything after the adjective that was excluded.

Now that you know how to exclude patterns from your chunks, it's time to look into named entity recognition (NER).



Using Named Entity Recognition (NER)

Named entities are noun phrases that refer to specific locations, people, organizations, and so on. With **named entity recognition**, you can find the named entities in your texts and also determine what kind of named entity they are.

Here's the list of named entity types from the NLTK book:

NE type	Examples
ORGANIZATION	Georgia-Pacific Corp., WHO
PERSON	Eddy Bonte, President Obama
LOCATION	Murray River, Mount Everest
DATE	June, 2008-06-29
TIME	two fifty a m, 1:30 p.m.
MONEY	175 million Canadian dollars, GBP 10.40
PERCENT	twenty pct, 18.75 %
FACILITY	Washington Monument, Stonehenge
GPE	South East Asia, Midlothian

You can use `nltk.ne_chunk()` to recognize named entities. Let's use `lotr_pos_tags` again to test it out:

Python

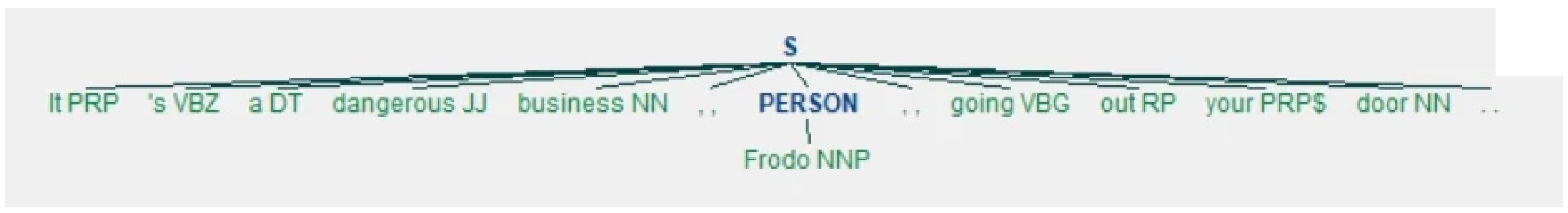
```python
>>> nltk.download("maxent_ne_chunker")
>>> nltk.download("words")
>>> tree = nltk.ne_chunk(lotr_pos_tags)
```

Now take a look at the visual representation:

Python

```python
>>> tree.draw()
```

Here's what you get:

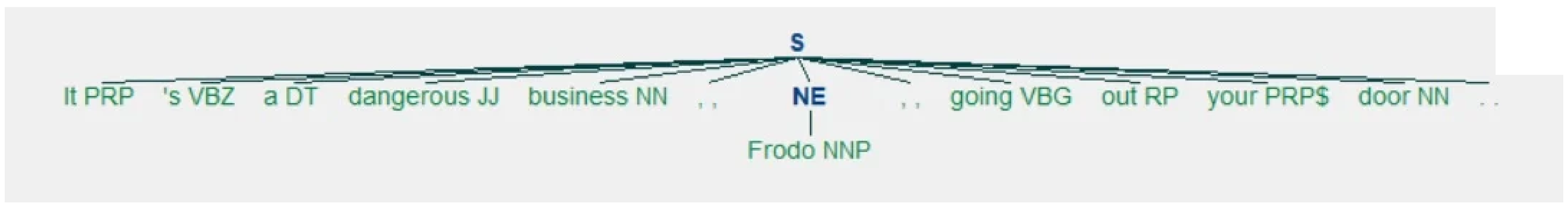


See how `Frodo` has been tagged as a `PERSON`? You also have the option to use the parameter `binary=True` if you just want to know what the named entities are but not what kind of named entity they are:

Python >>>

```
>>> tree = nltk.ne_chunk(lotr_pos_tags, binary=True)
>>> tree.draw()
```

Now all you see is that `Frodo` is an `NE`:



That's how you can identify named entities! But you can take this one step further and extract named entities directly from your text. Create a string from which to extract named entities. You can use this quote from *The War of the Worlds*:

Python >>>

```
>>> quote = """
... Men like Schiaparelli watched the red planet—it is odd, by-the-bye, that
... for countless centuries Mars has been the star of war—but failed to
... interpret the fluctuating appearances of the markings they mapped so well.
... All that time the Martians must have been getting ready.
...
... During the opposition of 1894 a great light was seen on the illuminated
... part of the disk, first at the Lick Observatory, then by Perrotin of Nice,
... and then by other observers. English readers heard of it first in the
... issue of Nature dated August 2."""
```

Now create a function to extract named entities:

Python >>>

```
>>> def extract_ne(quote):
...     words = word_tokenize(quote, language=language)
...     tags = nltk.pos_tag(words)
...     tree = nltk.ne_chunk(tags, binary=True)
...     return set(
...         " ".join(i[0] for i in t)
...         for t in tree
...         if hasattr(t, "label") and t.label() == "NE"
...     )
```

With this function, you gather all named entities, with no repeats. In order to do that, you tokenize by word, apply part of speech tags to those words, and then extract named entities based on those tags. Because you included `binary=True`, the named entities you'll get won't be labeled more specifically. You'll just know that they're named entities.

Take a look at the information you extracted:

Python >>>

```
>>> extract_ne(quote)
{'Lick Observatory', 'Mars', 'Nature', 'Perrotin', 'Schiaparelli'}
```

You missed the city of Nice, possibly because NLTK interpreted it as a regular English adjective, but you still got the following:

- **An institution:** `'Lick Observatory'`
- **A planet:** `'Mars'`
- **A publication:** `'Nature'`
- **People:** `'Perrotin'`, `'Schiaparelli'`

That's some pretty decent variety!

Getting Text to Analyze

Now that you've done some text processing tasks with small example texts, you're ready to analyze a bunch of texts at once. A group of texts is called a **corpus**. NLTK provides several **corpora** covering everything from novels hosted by Project Gutenberg to inaugural speeches by presidents of the United States.

In order to analyze texts in NLTK, you first need to import them. This requires `nltk.download("book")`, which is a pretty big download:

Python >>>

```
>>> nltk.download("book")
>>> 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 now have access to a few linear texts (such as *Sense and Sensibility* and *Monty Python and the Holy Grail*) as well as a few groups of texts (such as a chat corpus and a personals corpus). Human nature is fascinating, so let's see what we can find out by taking a closer look at the personals corpus!

This corpus is a collection of personals ads, which were an early version of online dating. If you wanted to meet someone, then you could place an ad in a newspaper and wait for other readers to respond to you.

If you'd like to learn how to get other texts to analyze, then you can check out Chapter 3 of *Natural Language Processing with Python – Analyzing Text with the Natural Language Toolkit*.

Using a Concordance

When you use a **concordance**, you can see each time a word is used, along with its immediate context. This can give you a peek into how a word is being used at the sentence level and what words are used with it.

Let's see what these good people looking for love have to say! The personals corpus is called `text8`, so we're going to call `.concordance()` on it with the parameter `"man"`:

Python >>>

```
>>> text8.concordance("man")
Displaying 14 of 14 matches:
 to hearing from you all . ABLE young man seeks , sexy older women . Phone for
ble relationship . GENUINE ATTRACTIVE MAN 40 y . o ., no ties , secure , 5 ft .
ship , and quality times . VIETNAMESE MAN Single , never married , financially
ip . WELL DRESSED emotionally healthy man 37 like to meet full figured woman fo
 nth subs LIKE TO BE MISTRESS of YOUR MAN like to be treated well . Bold DTE no
eeks lady in similar position MARRIED MAN 50 , attrac . fit , seeks lady 40 - 5
eks nice girl 25 - 30 serious rship . Man 46 attractive fit , assertive , and k
 40 - 50 sought by Aussie mid 40s b / man f / ship r / ship LOVE to meet widowe
discreet times . Sth E Subs . MARRIED MAN 42yo 6ft , fit , seeks Lady for discr
woman , seeks professional , employed man , with interests in theatre , dining
 tall and of large build seeks a good man . I am a nonsmoker , social drinker ,
lead to relationship . SEEKING HONEST MAN I am 41 y . o ., 5 ft . 4 , med . bui
 quiet times . Seeks 35 - 45 , honest man with good SOH & similar interests , f
 genuine , caring , honest and normal man for fship , poss rship . S / S , S /
```

Interestingly, the last three of those fourteen matches have to do with seeking an honest man, specifically:

1. `SEEKING HONEST MAN`
2. `Seeks 35 - 45 , honest man with good SOH & similar interests`
3. `genuine , caring , honest and normal man for fship , poss rship`

Let's see if there's a similar pattern with the word `"woman"`:

Python >>>

```
>>> text8.concordance("woman")
Displaying 11 of 11 matches:
at home . Seeking an honest , caring woman , slim or med . build , who enjoys t
thy man 37 like to meet full figured woman for relationship . 48 slim , shy , S
rry . MALE 58 years old . Is there a Woman who would like to spend 1 weekend a
 other interests . Seeking Christian Woman for fship , view to rship . SWM 45 D
ALE 60 - burly beared seeks intimate woman for outings n / s s / d F / ston / P
ington . SCORPIO 47 seeks passionate woman for discreet intimate encounters SEX
le dad . 42 , East sub . 5 " 9 seeks woman 30 + for f / ship relationship TALL
personal trainer looking for married woman age open for fun MARRIED Dark guy 37
rinker , seeking slim - medium build woman who is happy in life , age open . AC
. O . TERTIARY Educated professional woman , seeks professional , employed man
 real romantic , age 50 - 65 y . o . WOMAN OF SUBSTANCE 56 , 59 kg ., 50 , fit
```

The issue of honesty came up in the first match only:

Shell

```
Seeking an honest , caring woman , slim or med . build
```

Dipping into a corpus with a concordance won't give you the full picture, but it can still be interesting to take a peek and see if anything stands out.

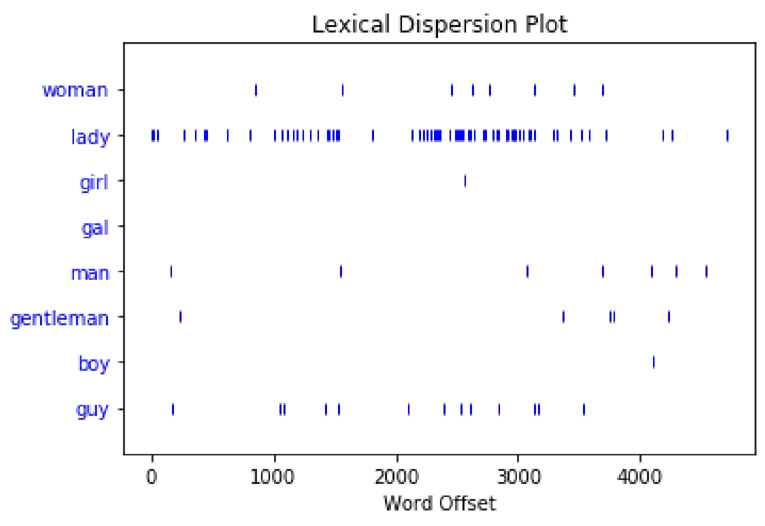
Making a Dispersion Plot

You can use a **dispersion plot** to see how much a particular word appears and where it appears. So far, we've looked for `"man"` and `"woman"`, but it would be interesting to see how much those words are used compared to their synonyms:

Python >>>

```
>>> text8.dispersion_plot(
...     ["woman", "lady", "girl", "gal", "man", "gentleman", "boy", "guy"]
... )
```

Here's the dispersion plot you get:



Each vertical blue line represents one instance of a word. Each horizontal row of blue lines represents the corpus as a whole. This plot shows that:

- `"lady"` was used a lot more than `"woman"` or `"girl"`. There were no instances of `"gal"`.
- `"man"` and `"guy"` were used a similar number of times and were more common than `"gentleman"` or `"boy"`.

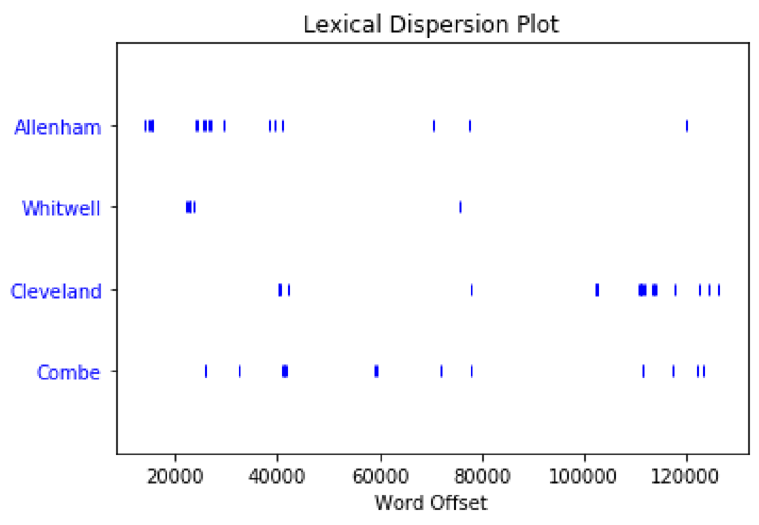
You use a dispersion plot when you want to see where words show up in a text or corpus. If you're analyzing a single text, this can help you see which words show up near each other. If you're analyzing a corpus of texts that is organized chronologically, it can help you see which words were being used more or less over a period of time.

Staying on the theme of romance, see what you can find out by making a dispersion plot for *Sense and Sensibility*, which is `text2`. Jane Austen novels talk a lot about people's homes, so make a dispersion plot with the names of a few homes:

Python >>>

```python
>>> text2.dispersion_plot(["Allenham", "Whitwell", "Cleveland", "Combe"])
```

Here's the plot you get:



Apparently Allenham is mentioned a lot in the first third of the novel and then doesn't come up much again. Cleveland, on the other hand, barely comes up in the first two thirds but shows up a fair bit in the last third. This distribution reflects changes in the relationship between Marianne and Willoughby:

- **Allenham** is the home of Willoughby's benefactress and comes up a lot when Marianne is first interested in him.
- **Cleveland** is a home that Marianne stays at after she goes to see Willoughby in London and things go wrong.

Dispersion plots are just one type of visualization you can make for textual data. The next one you'll take a look at is frequency distributions.

Remove ads

Making a Frequency Distribution

With a **frequency distribution**, you can check which words show up most frequently in your text. You'll need to get started with an `import`:

Python >>>

```
>>> from nltk import FreqDist
```

FreqDist is a subclass of `collections.Counter`. Here's how to create a frequency distribution of the entire corpus of personals ads:

Python >>>

```
>>> frequency_distribution = FreqDist(text8)
>>> print(frequency_distribution)
<FreqDist with 1108 samples and 4867 outcomes>
```

Since `1108` samples and `4867` outcomes is a lot of information, start by narrowing that down. Here's how to see the `20` most common words in the corpus:

Python >>>

```
>>> frequency_distribution.most_common(20)
[(',', 539),
 ('.', 353),
 ('/', 110),
 ('for', 99),
 ('and', 74),
 ('to', 74),
 ('lady', 68),
 ('-', 66),
 ('seeks', 60),
 ('a', 52),
 ('with', 44),
 ('S', 36),
 ('ship', 33),
 ('&', 30),
 ('relationship', 29),
 ('fun', 28),
 ('in', 27),
 ('slim', 27),
 ('build', 27),
 ('o', 26)]
```

You have a lot of stop words in your frequency distribution, but you can remove them just as you did earlier. Create a list of all of the words in `text8` that aren't stop words:

Python >>>

```
>>> meaningful_words = [
...     word for word in text8 if word.casefold() not in stop_words
... ]
```

Now that you have a list of all of the words in your corpus that aren't stop words, make a frequency distribution:

Python >>>

```
>>> frequency_distribution = FreqDist(meaningful_words)
```

Take a look at the `20` most common words:

Python >>>

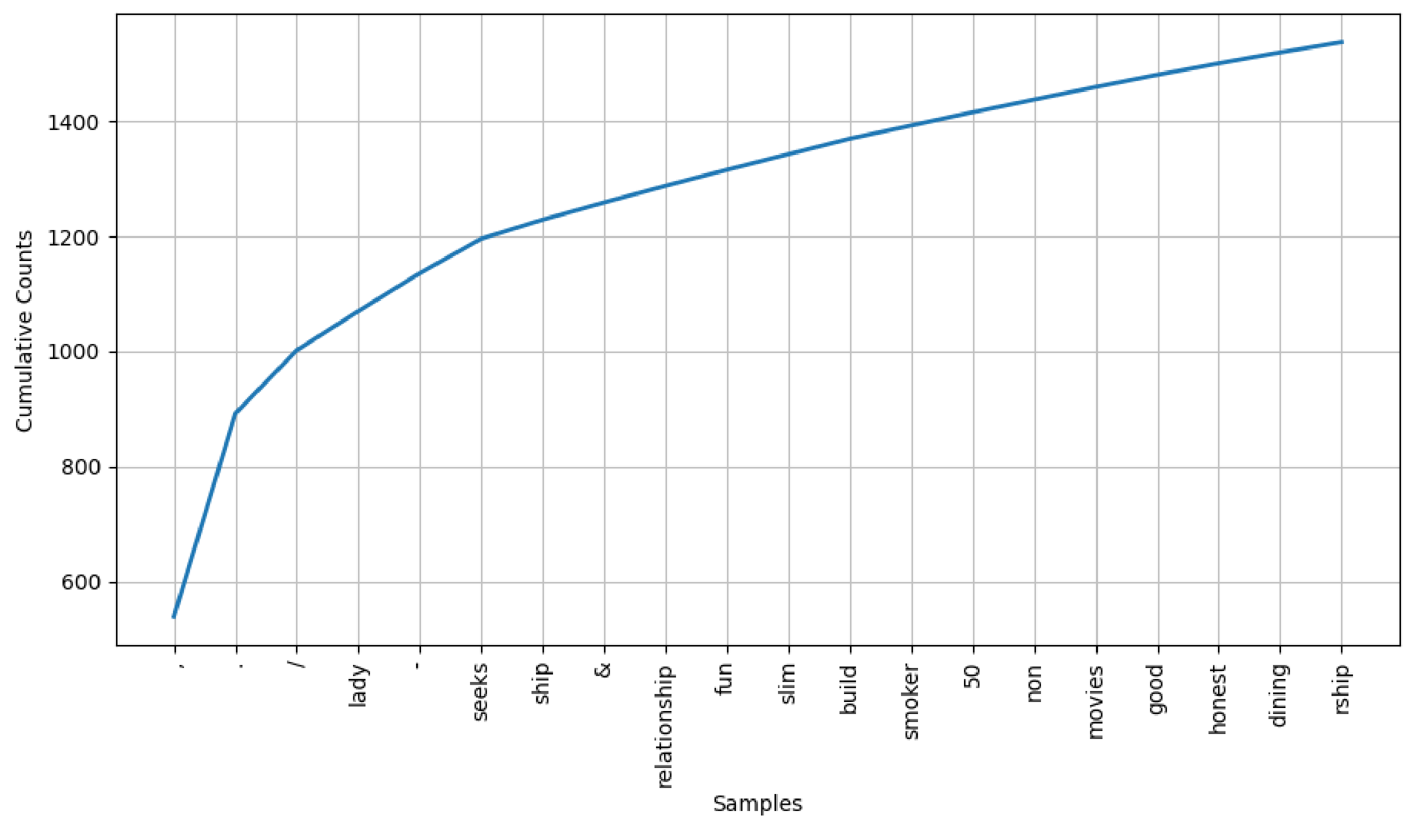
```
>>> frequency_distribution.most_common(20)
[(',', 539),
 ('.', 353),
 ('/', 110),
 ('lady', 68),
 ('-', 66),
 ('seeks', 60),
 ('ship', 33),
 ('&', 30),
 ('relationship', 29),
 ('fun', 28),
 ('slim', 27),
 ('build', 27),
 ('smoker', 23),
 ('50', 23),
 ('non', 22),
 ('movies', 22),
 ('good', 21),
 ('honest', 20),
 ('dining', 19),
 ('rship', 18)]
```

You can turn this list into a graph:

Python >>>

```
>>> frequency_distribution.plot(20, cumulative=True)
```

Here's the graph you get:



Some of the most common words are:

- `'lady'`
- `'seeks'`
- `'ship'`
- `'relationship'`
- `'fun'`
- `'slim'`
- `'build'`
- `'smoker'`
- `'50'`

- `'non'`
- `'movies'`
- `'good'`
- `'honest'`

From what you've already learned about the people writing these personals ads, they did seem interested in honesty and used the word `'lady'` a lot. In addition, `'slim'` and `'build'` both show up the same number of times. You saw `slim` and `build` used near each other when you were learning about concordances, so maybe those two words are commonly used together in this corpus. That brings us to collocations!

Finding Collocations

A **collocation** is a sequence of words that shows up often. If you're interested in common collocations in English, then you can check out *The BBI Dictionary of English Word Combinations*. It's a handy reference you can use to help you make sure your writing is idiomatic. Here are some examples of collocations that use the word "tree":

- Syntax tree
- Family tree
- Decision tree

To see pairs of words that come up often in your corpus, you need to call `.collocations()` on it:

Python

```
>>> text8.collocations()
would like; medium build; social drinker; quiet nights; non smoker;
long term; age open; Would like; easy going; financially secure; fun
times; similar interests; Age open; weekends away; poss rship; well
presented; never married; single mum; permanent relationship; slim
build
```

`slim build` did show up, as did `medium build` and several other word combinations. No long walks on the beach though!

But what would happen if you looked for collocations after lemmatizing the words in your corpus? Would you find some word combinations that you missed the first time around because they came up in slightly varied versions?

If you followed the instructions earlier, then you'll already have a `lemmatizer`, but you can't call `collocations()` on just any data type, so you're going to need to do some prep work. Start by creating a list of the lemmatized versions of all the words in `text8`:

Python

```
>>> lemmatized_words = [lemmatizer.lemmatize(word) for word in text8]
```

But in order for you to be able to do the linguistic processing tasks you've seen so far, you need to make an NLTK text with this list:

Python

```
>>> new_text = nltk.Text(lemmatized_words)
```

Here's how to see the collocations in your `new_text`:

Python >>>

```
>>> new_text.collocations()
medium build; social drinker; non smoker; long term; would like; age
open; easy going; financially secure; Would like; quiet night; Age
open; well presented; never married; single mum; permanent
relationship; slim build; year old; similar interest; fun time; Photo
pls
```

Compared to your previous list of collocations, this new one is missing a few:

- `weekends away`
- `poss rship`

The idea of `quiet nights` still shows up in the lemmatized version, `quiet night`. Your latest search for collocations also brought up a few news ones:

- **`year old`** suggests that users often mention ages.
- **`photo pls`** suggests that users often request one or more photos.

That's how you can find common word combinations to see what people are talking about and how they're talking about it!

Conclusion

Congratulations on taking your first steps with **NLP**! A whole new world of unstructured data is now open for you to explore. Now that you've covered the basics of text analytics tasks, you can get out there are find some texts to analyze and see what you can learn about the texts themselves as well as the people who wrote them and the topics they're about.

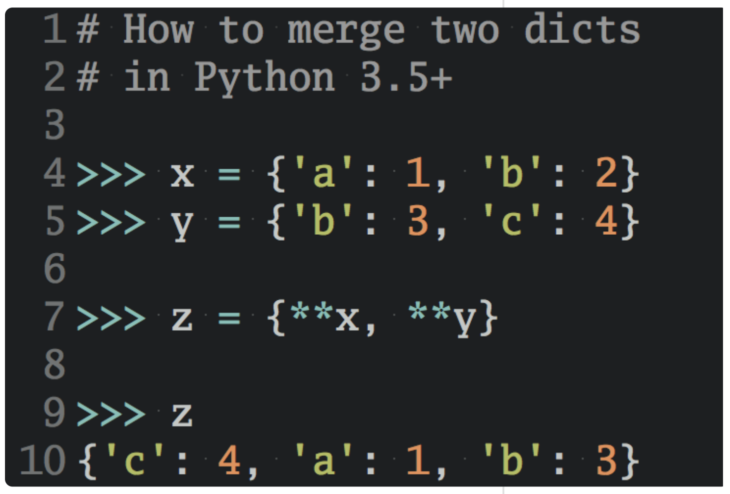
Now you know how to:

- **Find text** to analyze
- **Preprocess** your text for analysis
- **Analyze** your text
- Create **visualizations** based on your analysis

For your next step, you can use NLTK to analyze a text to see whether the sentiments expressed in it are positive or negative. To learn more about sentiment analysis, check out Sentiment Analysis: First Steps With Python's NLTK Library. If you'd like to dive deeper into the nuts and bolts of **NLTK**, then you can work your way through *Natural Language Processing with Python—Analyzing Text with the Natural Language Toolkit*.

Now get out there and find yourself some text to analyze!

Mark as Completed



```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```

Send Me Python Tricks »

About **Joanna Jablonski**

Joanna is the Executive Editor of Real Python. She loves natural languages just as much as she loves programming languages!

» More about Joanna

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

Aldren

David

Geir Arne

Jacob

What Do You Think?

Rate this article:

Tweet Share Share Email

What’s your #1 takeaway or favorite thing you learned? How are you going to put your newfound skills to use? Leave a comment below and let us know.

Commenting Tips: The most useful comments are those written with the goal of learning from or helping out other students. Get tips for asking good questions and get answers to common questions in our support portal.

Looking for a real-time conversation? Visit the Real Python Community Chat or join the next “Office Hours” Live Q&A Session. Happy Pythoning!

Keep Learning

Related Tutorial Categories: basics data-science