**Laboratory work 9 (NLTK)**

**Python 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 is how we represent text in Python, in this case the opening sentence of *Moby Dick*:

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

>>>

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

>>> sent1

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

>>> len(sent1)

4

>>> lexical\_diversity(sent1)

1.0

>>>

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 saying 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', '.']

>>>

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:

>>> ['Monty', 'Python'] + ['and', 'the', 'Holy', 'Grail']

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

We don’t have to literally type the lists either; we can use short names that refer to predefined 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']

**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**.

**Variables**

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

>>>

**Computing with language**

Let us return to our exploration of the ways we can bring our computational resources to bear on large quantities of text. 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. 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:]

**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. 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.

Since we often need frequency distributions in language processing, NLTK provides built-in support for them. Let us use a FreqDist to find the 50 most frequent words of *Moby Dick*. Try to work out what is going on here, then read the explanation that follows.

>>> fdist1 = FreqDist(text1)

>>> fdist1

<FreqDist with 260819 outcomes>

>>> vocabulary1 = fdist1.keys()

>>> vocabulary1[:50]

[',', 'the', '.', 'of', 'and', 'a', 'to', ';', 'in', 'that', "'", '-', 'his', 'it', 'I', 's', 'is', 'he', 'with', 'was', 'as', '"', 'all', 'for', 'this', '!', 'at', 'by', 'but', 'not', '--', 'him', 'from', 'be', 'on', 'so', 'whale', 'one', 'you', 'had', 'have', 'there', 'But', 'or', 'were', 'now', 'which', '?', 'me', 'like']

>>> fdist1['whale']

906

>>>

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. These 50 words account for nearly half the book!



**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. This means “the set of all *w* such that *w* is an element of *V* (the vocabulary) and *w* has property *P*.”

(1) a. {*w* | *w* ∈ *V* & *P*(*w*)}

b. [w for w in V 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:

>>> V = set(text1)

>>> long\_words = [w for w in V if len(w) > 15]

>>> sorted(long\_words)

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

>>>

**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 thatthey are resistant to substitution with words that have similar senses; for example, *maroon wine* sounds very odd.

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

>>> bigrams(['more', 'is', 'said', 'than', 'done'])

[('more', 'is'), ('is', 'said'), ('said', 'than'), ('than', 'done')]

>>>

Exercises

1. Take three texts in NLTK corpus and find the longest one and the shortest one
2. Create a list of tokens and then sort it
3. Find 100 most frequently used words in two texts from NLTK corpus
4. Create your own lists and append words to them
5. Create bigrams of words from the list