NLTK Tutorial: Basics

Edward Loper

Table of Contents

1. Goals

The Natural Language Toolkit (NLTK) defines a basic infrastructure that can be used to build NLP programs in Python. It provides:

- Basic classes for representing data relevant to natural language processing.
- Standard interfaces for performing tasks, such as tokenization, tagging, and parsing.
- Standard implementations for each task, which can be combined to solve complex problems.

This tutorial introduces NLTK, with an emphasis on tokens and tokenization.

2. Accessing NLTK

NLTK consists of a set of Python *modules*, each of which defines classes and functions related to a single data structure or task. Before you can use a module, you must *import* its contents. The simplest way to import the contents of a module is to use the "`from` *`module`* `import *`" command. For example, to import the contents of the `nltk.token` module, which is discussed in this tutorial, type:

```
>>> from nltk.token import *
```

A disadvantage of the "`from` *`module`* `import *`" command is that it does not specify what objects are imported; and it is possible that some of the import objects will unintentionally cause conflict. To avoid this disadvantage, you can explicitly list the objects you wish to import. For example, to import the `Token` and `Location` classes from the `nltk.token` module, type:

```
>>> from nltk.token import Token, Location
```

Another option is to import the module itself, rather than its contents. For example, to import the `nltk.token` module, type:

```
>>> import nltk.token
```

Once a module is imported, its contents can be accessed using fully qualified dotted names:

```
>>> nltk.token.Token('dog')
'dog'@[?]
>>> nltk.token.Location(3,5)
@[3:5]
```

For more information about importing, see any Python textbook.

3. Words

There are a number of reasonable ways to represent words in Python. Perhaps the simplest is as string values, such as `'dog'`; this is how words are typically represented when using NLTK.

```
>>> words = ['the', 'cat', 'climbed', 'the', 'tree']
['the', 'cat', 'climbed', 'the', 'tree']
```

However, NLTK also allows for other representations. For example, you could store words as integers, with some mapping between integers and words.

3.1. Types and Tokens

The term "word" can actually be used in two different ways: to refer to an individual occurrence of a word; or to refer to an abstract vocabulary item. For example, the phrase "my dog likes his dog" contains five occurrences of words, but only four vocabulary items (since the vocabulary item "dog" appears twice). In order to make this distinction clear, we will use the term *word token* to refer to occurrences of words, and the term *word type* to refer to vocabulary items.

The terms *token* and *type* can also be applied in other domains. For example, a *sentence token* is an individual occurrence of a sentence; but a *sentence type* is an abstract sentence, without context. If someone repeats a sentence twice, they have uttered two sentence tokens, but only one sentence type. When the kind of token or type is obvious from context, we will simply use the terms *token* and *type*.

In NLTK, `Token`s are constructed from their types, using the `Token` constructor, which is defined in the `nltk.token` module:

```
>>> from nltk.token import *
>>> my_word_type = 'dog'
'dog'
>>> my_word_token = Token(my_word_type)
'dog'@[?]
```

The reason that the token is displayed this way will be explained in the next two sections.

3.2. Text Locations

Text locations specify regions of texts, using a *start index* and an *end index*. A location with start index *s* and end index *e* is written `@[`*s*`:`*e*`]`, and specifies the region of the text beginning at *s*, and including everything up to (but not including) the text at *e*. `Location`s are created using the `Location` constructor, which is defined in the `nltk.token` module:

```
>>> from nltk.token import *
>>> my_loc = Location(1, 5)
@[1:5]
>>> another_loc = Location(0, 2)
@[0:2]
>>> yet_another_loc = Location(22, 22)
@[22:22]
```

Note that a text location does *not* include the text at its end location. This convention may seem unintuitive at first, but it has a number of advantages. It is consistent with Python's slice notation (e.g., `x[1:3]` specifies elements 1 and 2 of x). [1] It allows text locations to specify points between tokens, instead of just ranges; for example, `Location(3,3)` specifies the point just before the text at index 3. And it simplifies arithmetic on indices; for example, the length of `Location(5,10)` is `10-5`, and two locations are contiguous if the start of one equals the end of the other.

To create a text location specifying the text at a single index, you can use the `Location` constructor with a single argument. For example, the fourth word in a text could be specified with `loc1`:

```
>>> loc1 = Location(4)      # location width = 1
@[4]
```

NLTK uses the shorthand notation `@[s]` for locations whose width is one. Note that `Location(s)` is equivalent to `Location(s, s+1)`, *not* `Location(s, s)`:

```
>>> loc2 = Location(4, 5)   # location width = 1
@[4]
>>> loc3 = Location(4, 4)   # location width = 0
@[4:4]
```

3.2.1. Units

The start and end indices can be based on a variety of different *units*, such as character number, word number, or sentence number. By default, the unit of a text location is left unspecified, but locations can be explicitly tagged with information about what unit their indices use:

```
>>> my_loc = Location(1, 5, unit='w')
@[1w:5w]
>>> another_loc = Location(3, 72, unit='c')
@[3c:72c]
>>> my_loc = Location(6, unit='s')
@[6s]
>>> my_loc = Location(10, 11, unit='s')
@[10s]
```

Unit labels take the form of case-insensitive `strings`. Typical examples of unit labels are `'c'` (for character number), `'w'` (for word number), and `'s'` (for sentence number).

3.2.2. Sources

A text location may also be tagged with a *source*, which gives an indication of where the text was derived from. A typical example of a source would be a `string` containing the name of the file from which the element of text was read.

```
>>> my_loc = Location(1, 5, source='foo.txt')
@[1:5]@'foo.txt'
>>> another_loc = Location(3, 72, unit='c', source='bar.txt')
@[3c:72c]@'bar.txt'
>>> my_loc = Location(6, unit='s', source='baz.txt')
@[6s]@'baz.txt'
```

By default, a text location's source is unspecified.

Sometimes, it is useful to use text locations as the sources for other text locations. For example, we could specify the third character of the fourth word of the first sentence in the file `foo.txt` with `char_loc`:

```
>>> sentence_loc = Location(0, unit='s', source='foo.txt')
@[0s]@'foo.txt'
>>> word_loc = Location(3, unit='w', source=sentence_loc)
@[3w]@[0s]@'foo.txt'
>>> char_loc = Location(2, unit='c', source=word_loc)
@[2c]@[3w]@[0s]@'foo.txt'
```

Note that the location indexes are zero-based, so the first sentence starts at an index of zero, not one.

3.3. Tokens and Locations

As discussed above, a text token represents a single occurrence of a text type. In NLTK, a token is defined by a type, together with a location at which that type occurs. A token with type *t* and location `@[`*l*`]` can be written as *t*`@[`*l*`]`. Tokens are constructed with the `Token` constructor:

```
>>> token1 = Token('hello', Location(0, unit='w'))
'hello'@[0w]
>>> token2 = Token('world', Location(1, unit='w'))
'world'@[1w]
```

Two tokens are only equal if both their type and their location are equal:

```
>>> token1 = Token('hello', Location(0, unit='w'))
'hello'@[0w]
>>> token2 = Token('hello', Location(1, unit='w'))
'world'@[1w]
>>> token3 = Token('world', Location(0, unit='w'))
'world'@[1w]
>>> token4 = Token('hello', Location(0, unit='w'))
'world'@[1w]
>>> token1 == token2
0
>>> token1 == token3
0
>>> token1 == token4
1
```

When a token's location is unknown or unimportant, the special location `None` may be used. A token with type *t* and location `None` is written as *t*`@[?]`. If a token's location is not specified, it defaults to `None`:

```
>>> token1 = Token('hello', None)
'hello'@[?]
>>> token2 = Token('world')
'world'@[?]
```

A Token with a location of `None` is not considered to be equal to any other token. In particular, even if two tokens have the same type, and both have a location of `None`, they are not equal:

```
>>> token1 = Token('hello')
'hello'@[?]
>>> token2 = Token('hello')
'hello'@[?]
>>> token1 == token2
0
```

To access a token's type, use its `type` member function; and to access a token's location, use its `loc` member function:

```
>>> token1 = Token('hello', Location(0, unit='w'))
'hello'@[0w]
>>> token1.type()
'hello'
>>> token1.loc()
@[0w]
```

To access a location's start index, use its `start` member function; and to access a location's end index, use its `end` member function:

```
>>> loc1 = Location(0, 8, unit='w')
@[0w:8w]
>>> loc1.start()
0
>>> loc1.end()
8
```

To access a location's unit index, use its `unit` member function; and to access a location's source index, use its `source` member function:

```
>>> loc1 = Location(3, unit='w', source='foo.txt')
@[3w]@'foo.txt'
>>> loc1.unit()
'w'
>>> loc1.source()
'foo.txt'
```

For more information about tokens and locations, see the reference documentation for the `nltk.token` module.

4. Texts

Many natural language processing tasks involve analyzing texts of varying sizes, ranging from single sentences to very large corpora. There are a number of ways to represent texts using NLTK. The simplest is as a single `string`. These strings are typically loaded from files:

```
>>> text_str = open('corpus.txt').read()
'Hello world.  This is a test file.\n'
```

It is often more convenient to represent a text as a `list` of `Token`s. These `lists` are typically created using a *tokenizer*, such as `WSTokenizer` (which splits words apart based on whitespace):

```
>>> text_tok_list = WSTokenizer().tokenize(text_str)
['Hello'@[0w], 'world.'@[1w], 'This'@[2w], 'is'@[3w],
 'a'@[4w], 'test'@[5w], 'file.'@[6w]]
```

Texts can also be represented as sets of word tokens or sets of word types:

```
>>> text_tok_set = Set(*text_tok_list)
{'This'@[2w], 'a'@[4w], 'Hello'@[0w], 'world.'@[1w],
 'is'@[3w], 'file.'@[6w], 'test'@[5w]}
```

Note: For example, this representation might be convenient for a search engine which is trying to find all documents containing some keyword. (For more information on why a "`*`" is used in the call to the Set constructor, see the Set tutorial.)

5. Tokenization

As mentioned in the previous section, it is often useful to represent a text as a list of tokens. The process of breaking a text up into its constituent tokens is known as *tokenization*. Tokenization can occur at a number of different levels: a text could be broken up into paragraphs, sentences, words, syllables, or phonemes. And for any given level of tokenization, there are many different algorithms for breaking up the text. For example, at the word level, it is not immediately clear how to treat such strings as "can't," "$22.50," "New York," and "so-called."

NLTK defines a general interface for tokenizing texts, the `TokenizerI` class. This interface is used by all tokenizers, regardless of what level they tokenize at or what algorithm they use. It defines a single method, `tokenize`, which takes a `string`, and returns a list of `Token`s.

5.1. NLTK Interfaces

`TokenizerI` is the first "interface" class we've encountered; at this point, we'll take a short digression to explain how interfaces are implemented in NLTK.

An *interface* gives a partial specification of the behavior of a class, including specifications for methods that the class should implement. For example, a "comparable" interface might specify that a class must implement a comparison method. Interfaces do not give a complete specification of a class; they only specify a minimum set of methods and behaviors which should be implemented by the class. For example, the `TokenizerI` interface specifies that a tokenizer class must implement a `tokenize` method, which takes a `string`, and returns a list of `Token`s; but it does not specify what other methods the class should implement (if any).

The notion of "interfaces" can be very useful in ensuring that different classes work together correctly. Although the concept of "interfaces" is supported in many languages, such as Java, there is no native support for interfaces in Python.

NLTK therefore implements interfaces using classes, all of whose methods raise the `NotImplementedError` exception. To distinguish interfaces from other classes, they are always named with a trailing "`I`". If a class implements an interface, then it should be a subclass of the interface. For example, the `WSTokenizer` class implements the `TokenizerI` interface, and so it is a subclass of `TokenizerI`.

5.2. The whitespace tokenizer

A simple example of a tokenizer is the `WSTokenizer`, which breaks a text into words, assuming that words are separated by whitespace (space, enter, and tab characters). We can use the `WSTokenizer` constructor to build a new whitespace tokenizer:

```
>>> tokenizer = WSTokenizer()
```

Once we have built the tokenizer, we can use it to process texts:

```
>>> tokenizer.tokenize(text_str)
['Hello'@[0w], 'world.'@[1w], 'This'@[2w], 'is'@[3w],
 'a'@[4w], 'test'@[5w], 'file.'@[6w]]
```

However, this tokenizer is not ideal for many tasks. For example, we might want punctuation to be included as separate tokens; or we might want names like "New York" to be included as single tokens.

5.3. The regular expression tokenizer

The `RETokenizer` is a more powerful tokenizer, which uses a regular expression to determine how text should be split up. This regular expression specifies the format of a valid word. For example, if we wanted to mimic the behavior or `WSTokenizer`, we could define the following `RETokenizer`:

```
>>> tokenizer = RETokenizer(r'[^\s]+')
>>> tokenizer.tokenize(example_text)
['Hello.'@[0w], "Isn't"@[1w], 'this'@[2w], 'fun?'@[3w]]
```

(The regular expression `\s` matches any whitespace character.)

To define a tokenizer that includes punctuation as separate tokens, we could use:

```
>>> regexp = r'\w+|[^\w\s]+'
'\w+|[^\w\s]+'
>>> tokenizer = RETokenizer(regexp)
>>> tokenizer.tokenize(example_text)
['Hello'@[0w], '.'@[1w], 'Isn'@[2w], "'"@[3w], 't'@[4w],
 'this'@[5w], 'fun'@[6w], '?'@[7w]]
```

The regular expression in this example will match *either* a sequence of alphanumeric characters (letters and numbers); *or* a sequence of punctuation characters.

There are a number of ways we might want to improve this regular expression. For example, it currently breaks the string "$22.50" into four tokens; but we might want it to include this as a single token. One approach to making this change would be to add a new clause to the tokenizer's regular expression, which is specialized for handling strings of this form:

```
>>> example_text = 'That poster costs $22.40.'
>>> regexp = r'(\w+)|(\$\d+\.\d+)|([^\w\s]+)'
'(\w+)|(\$\d+\.\d+)|([^\w\s]+)'
>>> tokenizer = RETokenizer(regexp)
>>> tokenizer.tokenize(example_text)
['That'@[0w], 'poster'@[1w], 'costs'@[2w], '$22.40'@[3w], '.'@[4w]]
```

Of course, more general solutions to this problem are also possible, using different regular expressions.

6. Example: Processing Tokenized Text

In this section, we show how you can use NLTK to examine the distribution of word lengths in a document. This is meant to be a simple example of how the tools we have introduced can be used to solve a simple NLP problem. The distribution of word lengths in a document can give clues to other properties, such as the document's style, or the document's language.

We present three different approaches to solving this problem; each one illustrates different techniques, which might be useful for other problems.

6.1. Word Length Distributions 1: Using a List

To begin with, we'll need to extract the words from a corpus that we wish to test. We'll use the `WSTokenizer` to tokenize the corpus:

```
>>> corpus = open('corpus.txt').read()
>>> tokens = WSTokenizer().tokenize(corpus)
```

Now, we will construct a list `wordlen_count_list`, which gives the number of words that have a given length. In particular, `wordlen_count_list[`*`i`*`]` is the number of words whose length is *`i`*.

When constructing this list, we must be careful not to try to add a value past the end of the list. Therefore, whenever we encounter a word that is longer than any previous words, we will add enough zeros to `wordlen_count_list` that we can store the occurrence of the new word:

```
>>> wordlen_count_list = []
>>> for token in tokens:
...     wordlen = len(token.type())
...     # Add zeros until wordlen_count_list is long enough
...     while wordlen >= len(wordlen_count_list):
...         wordlen_count_list.append(0)
...     # Increment the count for this word length
...     wordlen_count_list[wordlen] += 1
```

We can plot the results, using the `Plot` class, defined in the `nltk.draw.plot` module:

```
>>> Plot(wordlen_count_list)
```

Note: We are currently using a fairly simple class to plot functions. We will likely replace it with a more advanced plotting system in the future.

The complete code for this example, including necessary `import` statements, is:

```
from nltk.token import WSTokenizer
from nltk.draw.plot import Plot

# Extract a list of words from the corpus
corpus = open('corpus.txt').read()
tokens = WSTokenizer().tokenize(corpus)

# Count up how many times each word length occurs
wordlen_count_list = []
for token in tokens:
     wordlen = len(token.type())
     # Add zeros until wordlen_count_list is long enough
     while wordlen >= len(wordlen_count_list):
         wordlen_count_list.append(0)
     # Increment the count for this word length
     wordlen_count_list[wordlen] += 1

Plot(wordlen_count_list)
```

6.2. Word Length Distributions 2: Using a Dictionary

We have been examining the function from word lengths to token counts. In this example, the range of the function (i.e., the set of word lengths) is ordered and relatively small. However, we often wish to examine functions whose ranges are not so well behaved. In such cases, dictionaries can be a powerful tool. The following code uses a dictionary to count up the number of times each word length occurs:

```
>>> wordlen_count_dict = {}
>>> for token in tokens:
...     word_length = len(token.type())
...     if wordlen_count_dict.has_key(word_length):
...         wordlen_count_dict[word_length] += 1
...     else:
...         wordlen_count_dict[word_length] = 1
```

To plot the results, we can use a list of (wordlen, count) pairs. This is simply the `items` of the dictionary:

```
>>> points = wordlen_count_dict.items()
>>> Plot(points)
```

The complete code for this example, including necessary `import` statements, is:

```
from nltk.token import WSTokenizer
from nltk.draw.plot import Plot

# Extract a list of words from the corpus
corpus = open('corpus.txt').read()
tokens = WSTokenizer().tokenize(corpus)

# Construct a dictionary mapping word lengths to token counts
wordlen_count_dict = {}
for token in tokens:
    word_length = len(token.type())
    if wordlen_count_dict.has_key(word_length):
        wordlen_count_dict[word_length] += 1
    else:
        wordlen_count_dict[word_length] = 1

# Construct a list of (wordlen, count) and plot the results.
points = wordlen_count_dict.items()
Plot(points)
```

6.3. Word Length Distributions 3: Using a Frequency Distribution

The `nltk.probability` module defines two interfaces, `FreqDistI` and `ProbDistI`, for modeling frequency distributions and probability distributions, respectively. In this example, we use a frequency distribution to find the relationship between word lengths and token counts.

We will use a `SimpleFreqDist`, which is a simple (but sometimes inefficient) implementation of the `FreqDistI` interface. For this example, three methods of `SimpleFreqDist` are relevant:

- `inc(`*`sample`*`)` increments the frequency of a given sample.
- `samples()` returns a list of the samples covered by a frequency distribution.
- `count(`*`sample`*`)` returns the number of times a given sample occurred.

First, we construct the frequency distribution for the word lengths:

```
>>> wordlen_freqs = SimpleFreqDist()
```

```
>>> for token in tokens:
...     wordlen_freqs.inc(len(token.type()))
```

Next, we extract the set of word lengths that were found in the corpus:

```
>>> wordlens = wordlen_freqs.samples()
```

Finally, we construct a list of (wordlen, count) pairs, and plot it:

```
>>> points = [(wordlen, wordlen_freqs.count(wordlen))
...           for wordlen in wordlens]
>>> Plot(points)
```

> **Note:** The expression `[... for ... in ...]` is called a "list comprehension." For more information on list comprehensions, see the "New Python Features" tutorial.

The complete code for this example, including necessary `import` statements, is:

```
from nltk.token import WSTokenizer
from nltk.draw.plot import Plot
from nltk.probability import SimpleFreqDist

# Extract a list of words from the corpus
corpus = open('corpus.txt').read()
tokens = WSTokenizer().tokenize(corpus)

# Construct a frequency distribution of word lengths
wordlen_freqs = SimpleFreqDist()
for token in tokens:
    wordlen_freqs.inc(len(token.type()))

# Exctract the set of word lengths found in the corpus
wordlens = wordlen_freqs.samples()

# Construct a list of (wordlen, count) and plot the results.
points = [(wordlen, wordlen_freqs.count(wordlen))
          for wordlen in wordlens]
Plot(points)
```

For more information about frequency distributions, see the Probability Tutorial.

Index

Notes

1. But unlike Python slices, text locations do *not support negative indexes.*