REVA UNIVERSITY
Bengaluru, India
Established as per the section 2(f) of the UGC Act, 1956,
Approved by AICTE, New Delhi

Natural Language Processing

UNIT-2

Processing Raw Text

UNIT-2: Processing Raw Text

- The goal of this chapter is :

1. How can we write programs to access text from local files and from the Web, in order to get hold of an unlimited range of language material?
2. How can we split documents up into individual words and punctuation symbols, so we can carry out the some kinds of analysis we did with text corpora in earlier chapters?
3. How can we write programs to produce formatted output and save it in a file?

- In order to address above questions, we will be covering key concepts in NLP, including tokenization and stemming.
- Along the way you will consolidate your Python knowledge and learn about strings, files, and regular expressions.
- Since so much text on the Web is in HTML format, we will also see how to dispense with markup.

• Accessing Text from the Web and from Disk

- **Electronic Books**
- A small sample of texts from Project Gutenberg appears in the NLTK corpus collection. interested
- However, you may be in analyzing other texts from Project Gutenberg.
- You can browse the catalog of 25,000 free online books at http://www.gutenberg.org/catalog/, and obtain a URL to an ASCII text file. Although 90% of the texts in Project Gutenberg are in English, it includes material in over 50 other languages, including Catalan, Chinese, Dutch, Finnish, French, German, Italian, Portuguese, and Spanish (with more than 100 texts each).
- Text number 2554 is an English translation of *Crime and Punishment*, and we can access it as follows.

- >>> from urllib import urlopen
- >>> url = "http://www.gutenberg.org/files/2554/2554.txt"
- >>> raw = urlopen(url).read()
- >>> type(raw)
- <type 'str'>
- >>> len(raw) 1176831
- >>> raw[:75]

'The Project Gutenberg EBook of Crime and Punishment, by Fyodor Dostoevsky\r\n'

- The variable raw contains a string with 1176831 characters. This is the raw content of the book, including many details we are not interested in, such as whitespace, line breaks, and blank lines.
- Notice the\r and \n in the opening line of the file, which is how Python displays the special carriage return and line-feed characters.
- For our language processing, we want to break up the string into words and punctuation. This step is called **tokenization**, and it produces our familiar structure, a list of words and punctuation.

- >>> tokens = nltk.word_tokenize(raw)
- >>> type(tokens)
- <type 'list'>
- >>> len(tokens) 255809
- >>> tokens[:10]

['The', 'Project', 'Gutenberg', 'EBook', 'of', 'Crime', 'and', 'Punishment', ',', 'by']

- Notice that NLTK was needed for tokenization, but not for any of the earlier tasks of opening a URL and reading it into a string.
- If we now take the further step of creating an NLTK text from this list, we can carry out all of the other linguistic processing we saw in Chapter 1, along with the regular list operations, such as slicing:
- >>> text = nltk.Text(tokens)
- >>> type(text)
- <type 'nltk.text.Text'>
- >>> text[1020:1060]

- ['CHAPTER', 'I', 'On', 'an', 'exceptionally', 'hot', 'evening', 'early', 'in',
- 'July', 'a', 'young', 'man', 'came', 'out', 'of', 'the', 'garret', 'in',
- 'which', 'he', 'lodged', 'in', 'S', '.', 'Place', 'and', 'walked', 'slowly',
- ',', 'as', 'though', 'in', 'hesitation', ',', 'towards', 'K', '.', 'bridge', '.']
- >>> text.collocations()
- Katerina Ivanovna; Pulcheria Alexandrovna; Avdotya Romanovna; Pyotr Petrovitch; Project Gutenberg; Marfa Petrovna; Rodion Romanovitch; Sofya Semyonovna; Nikodim Fomitch; did not; Hay Market; Andrey Semyonovitch; old woman; Literary Archive; Dmitri Prokofitch; great deal; United States; Praskovya Pavlovna; Porfiry Petrovitch; ear rings
- Notice that *Project Gutenberg* appears as a collocation. This is because each text downloaded from Project Gutenberg contains a header with the name of the text, the author, the names of people who scanned and corrected the text, a license, and so on. Some- times this information appears in a footer at the end of the file. We cannot reliably detect where the content begins and ends, and so have to resort to manual inspection of the file, to discover unique strings that mark the beginning and the end, before trimming raw to be just the content and nothing else:

```
>>> raw.find("PART I")
5303
>>> raw.rfind("End of Project Gutenberg's Crime")
1157681
>>> raw = raw[5303:1157681]
>>> raw.find("PART I")
0
```

- The find() and rfind() ("reverse find") methods help us get the right index values to use for slicing the string.
- We overwrite raw with this slice, so now it begins with "PART I" and goes up to (but not including) the phrase that marks the end of the content.

- **Dealing with HTML**
- Much of the text on the Web is in the form of HTML documents. You can use a web browser to save a page as text to a local file.
- However, if you’re going to do this often, it’s easiest to get Python to do the work directly. The first step is the same as before, using urlopen.
- For fun we’ll pick a BBC News story called “Blondes to die out in 200 years,” an urban legend passed along by the BBC as established scientific fact:
- >>> url = "http://news.bbc.co.uk/2/hi/health/2284783.stm"
- >>> html = urlopen(url).read()
- >>> html[:60]

'<!doctype html public "-//W3C//DTD HTML 4.0 Transitional//EN'

- You can type print html to see the HTML content in all its glory, including meta tags, an image map, JavaScript, forms, and tables

- Getting text out of HTML is a sufficiently common task that NLTK provides a helper function nltk.clean_html(), which takes an HTML string and returns raw text. We can then tokenize this to get our familiar text structure:
- >>> raw = nltk.clean_html(html)
- >>> tokens = nltk.word_tokenize(raw)
- >>> tokens
- ['BBC', 'NEWS', '|', 'Health', 'Blondes', "'", 'to', 'die', 'out', ...]
- This still contains unwanted material concerning site navigation and related stories. With some trial and error you can find the start and end indexes of the content and select the tokens of interest, and initialize a text as before.
- >>> tokens = tokens[96:399]
- >>> text = nltk.Text(tokens)
- >>> text.concordance('gene')

they say too few people now carry the gene for blondes to last beyond the next tw t blonde hair is caused by a recessive gene . In order for a child to have blonde to have blonde hair , it must have the gene on both sides of the family in the gra there is a disadvantage of having that gene or by chance . They don ' t disappear ondes would disappear is if having the gene was a disadvantage and I do not think

- **Processing Search Engine Results**
- The Web can be thought of as a huge corpus of unannotated text. Web search engines provide an efficient means of searching this large quantity of text for relevant linguistic examples.
- The main advantage of search engines is size: since you are searching such a large set of documents, you are more likely to find any linguistic pattern you are interested in.
- Furthermore, you can make use of very specific patterns, which would match only one or two examples on a smaller example, but which might match tens of thousands of examples when run on the Web.
- A second advantage of web search en- gines is that they are very easy to use.
- Thus, they provide a very convenient tool for quickly checking a theory, to see if it is reasonable. See Table 3-1 for an example.

Google hits	adore	LOVE	like	prefer
absolutely	289,000	905,000	16,200	644
definitely	1,460	51,000	158,000	62,600
ratio	198:1	18:1	1:10	1:97

Table 3-1. Google hits for collocations: The number of hits for collocations involving the words absolutely *or* definitely, *followed by one of* adore, love, like, *or* prefer. *(Liberman, in LanguageLog, 2005)*

Processing RSS Feeds

- The blogosphere is an important source of text, in both formal and informal registers.
- With the help of a third-party Python library called the *Universal Feed Parser*, freely downloadable from [*http://feedparser.org/*](http://feedparser.org/),
- we can access the content of a blog, as shown here:

```
>>> import feedparser
>>> llog = feedparser.parse("http://languagelog.ldc.upenn.edu/nll/?feed=atom")
>>> llog['feed']['title'] u'Language Log'
>>> len(llog.entries) 15
>>> post = llog.entries[2]
>>> post.title u"He's My BF"
>>> content = post.content[0].value
>>> content[:70]
u'<p>Today I was chatting with three of our visiting graduate students f'
>>> nltk.word_tokenize(nltk.html_clean(content))
>>> nltk.word_tokenize(nltk.clean_html(llog.entries[2].content[0].value))
[u'Today', u'I', u'was', u'chatting', u'with', u'three', u'of', u'our', u'visiting',
u'graduate', u'students', u'from', u'the', u'PRC', u'.', u'Thinking', u'that', u'I',
u'was', u'being', u'au', u'courant', u',', u'I', u'mentioned', u'the', u'expression',
u'DUI4XIANG4', u'\u5c0d\u8c61', u'("', u'boy', u'/', u'girl', u'friend', u'"', ...]
```

- Note that the resulting strings have a u prefix to indicate that they are Unicode strings .With some further work, we can write programs to create a small corpus of blog posts, and use this as the basis for our NLP work

Reading Local Files

•In order to read a local file, we need to use Python's built-in `open()` function, followed b the `read()` method. Supposing you have a file *document.txt*, you can load its contents lik this:

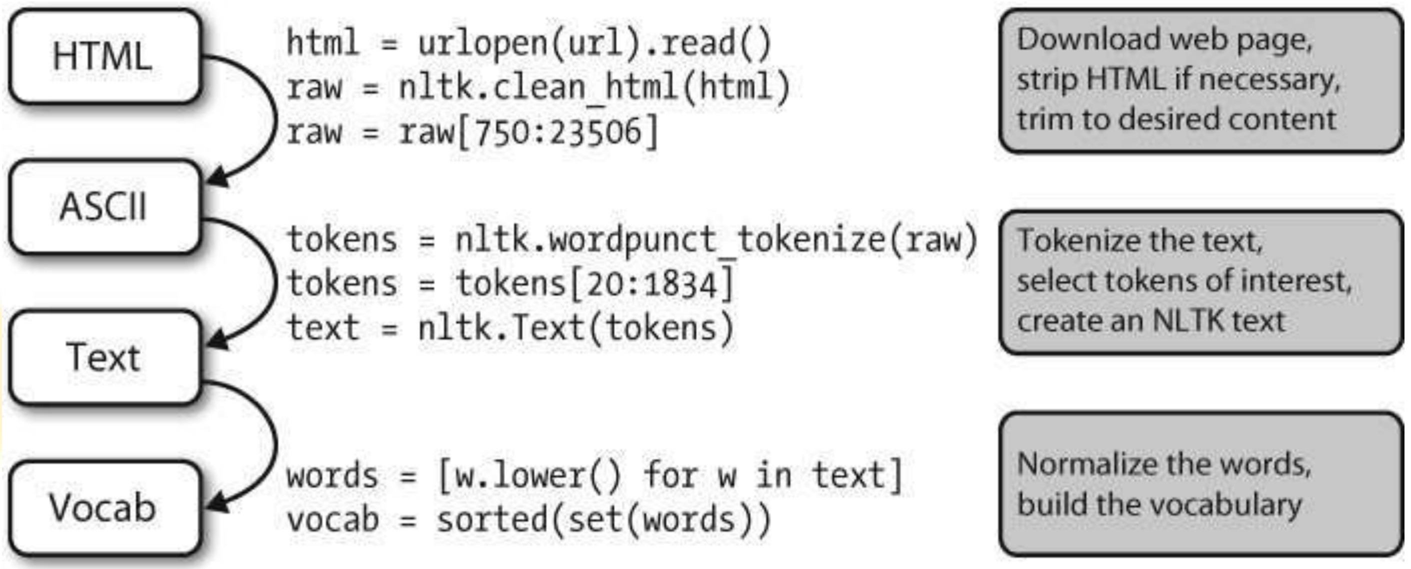
```
•>>> f = open('document.txt')
•>>> raw = f.read()
```

- Another possible problem you might have encountered when accessing a text file is the newline conventions, which are different for different operating systems.
- The built-in `open()` function has a second parameter for controlling how the file is opened: `open('do cument.txt', 'rU')`. `'r'` **means to open the file for reading** (the default), and `'U'` **stands for "Universal"**, which lets us ignore the different conventions used for marking new- lines.
- Assuming that you can open the file, there are several methods for reading it.
- The read() method creates a string with the contents of the entire file:

- >>> f.read()
- 'Time flies like an arrow.\nFruit flies like a banana.\n'
- Recall that the '\n' characters are **newlines**; this is equivalent to pressing Enter on a keyboard and starting a new line.
- We can also read a file one line at a time using a for loop:
- >>> f = open('document.txt', 'rU')
- >>> for line in f:
- ...   print line.strip() Time flies like an arrow. Fruit flies like a banana.
- Here we use the strip() method to remove the newline character at the end of the input line.
- NLTK's corpus files can also be accessed using these methods. We simply have to use nltk.data.find() to get the filename for any corpus item. Then we can open and read it in the way we just demonstrated:
- >>> path = nltk.data.find('corpora/gutenberg/melville-moby_dick.txt')
- >>> raw = open(path, 'rU').read()

- **Extracting Text from PDF, MSWord, and Other Binary Formats**
- ASCII text and HTML text are human-readable formats.
- Text often comes in binary formats—such as PDF and MSWord—that can only be opened using specialized soft- ware.
- Third-party libraries such as pypdf and pywin32 provide access to these formats. Extracting text from multicolumn documents is particularly challenging.
- For one-off conversion of a few documents, it is simpler to open the document with a suitable application, then save it as text to your local drive, and access it as described below.
- If the document is already on the Web, you can enter its URL in Google's search box.
- The search result often includes a link to an HTML version of the document, which you can save as text.

- **Capturing User Input**
- Sometimes we want to capture the text that a user inputs when she is interacting with our program. To prompt the user to type a line of input, call the Python function raw_input(). After saving the input to a variable, we can manipulate it just as we have done for other strings.
- >>> s = raw_input("Enter some text: ")
- Enter some text: On an exceptionally hot evening early in July
- >>> print "You typed", len(nltk.word_tokenize(s)), "words." You typed 8 words.
- Figure 3-1 summarizes what we have covered in this section, including the process of building a vocabulary that we saw in Chapter 1. (One step, normalization, will be discussed in Section 3.6.)


HTML
ASCII
Text
Vocab
Download web page, strip HTML if necessary, trim to desired content
Tokenize the text, select tokens of interest, create an NLTK text
Normalize the words, build the vocabulary

```
html = urlopen(url).read()
raw = nltk.clean_html(html)
raw = raw[750:23506]

tokens = nltk.wordpunct_tokenize(raw)
tokens = tokens[20:1834]
text = nltk.Text(tokens)

words = [w.lower() for w in text]
vocab = sorted(set(words))
```

Figure 3-1. The processing pipeline: We open a URL and read its HTML content, remove the markup and select a slice of characters; this is then tokenized and optionally converted into an `nltk.Text` *object; we can also lowercase all the words and extract the vocabulary*

- There's a lot going on in this pipeline. To understand it properly, it helps to be clear about the type of each variable that it mentions.
- We find out the type of any Python object *x* using `type(`*x*`)`; e.g., `type(1)` is `<int>` since `1` is an integer.

When we load the contents of a URL or file, and when we strip out HTML markup, we are dealing with strings, Python's `<str>` data type (we will learn more about strings in Section 3.2):

```
>>> raw = open('document.txt').read()
>>> type(raw)
<type 'str'>
```

When we tokenize a string we produce a list (of words), and this is Python's <list> type. Normalizing and sorting lists produces other lists:

```
>>> tokens = nltk.word_tokenize(raw)
>>> type(tokens)
<type 'list'>
>>> words = [w.lower() for w in tokens]
>>> type(words)
<type 'list'>
>>> vocab = sorted(set(words))
>>> type(vocab)
<type 'list'>
```

•The type of an object determines what operations you can perform on it. So, for example, we can append to a list but not to a string:

```
>>> vocab.append('blog')
>>> raw.append('blog')

Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: 'str' object has no attribute 'append'
```

- Similarly, we can concatenate strings with strings, and lists with lists, but we cannot concatenate strings with lists:
- >>> query = 'Who knows?'
- >>> beatles = ['john', 'paul', 'george', 'ringo']
- >>> query + beatles

```
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'list' objects
```

- **Strings: Text Processing at the Lowest Level**

- The contents of a word, and of a file, are represented by programming languages as a fundamental data type known as a **string**.
- In this section, we explore strings in detail, and show the connection between strings, words, texts, and files.
- Strings are specified using single quotes or double quotes as shown in the fol lowing code example.
- If a string contains a single quote, we must backslash-escape the quote
- Python knows a literal quote character is intended, or else put the string in double quotes Otherwise, the quote inside the string will be interpreted as a close quote, and the Python interpreter will report a syntax error:

- >>> monty = 'Monty Python'
- >>> monty
- 'Monty Python'
- >>> circus = "Monty Python's Flying Circus"
- >>> circus
- "Monty Python's Flying Circus"
- >>> circus = 'Monty Python\'s Flying Circus'
- >>> circus
- "Monty Python's Flying Circus"
- >>> circus = 'Monty Python's Flying Circus'
- File "<stdin>", line 1
- circus = 'Monty Python's Flying Circus'
- SyntaxError: invalid syntax

- Sometimes strings go over several lines. Python provides us with various ways of entering them. In the next example, a sequence of two strings is joined into a single string. We need to use backslash or parentheses so that the interpreter knows that the statement is not complete after the first line.
- >>> couplet = "Shall I compare thee to a Summer's day?"\

..."Thou are more lovely and more temperate:"

- >>> print couplet

Shall I compare thee to a Summer's day?Thou are more lovely and more temperate:

- >>> couplet = ("Rough winds do shake the darling buds of May,"

..."And Summer's lease hath all too short a date:")

- >>> print couplet

Rough winds do shake the darling buds of May,And Summer's lease hath all too short a date:

- Unfortunately these methods do not give us a newline between the two lines of the sonnet.

Instead, we can use a triple-quoted string as follows:

- >>> couplet = """Shall I compare thee to a Summer's day?
- ... Thou are more lovely and more temperate:"""
- >>> print couplet

Shall I compare thee to a Summer's day?

Thou are more lovely and more temperate:

- >>> couplet = '''Rough winds do shake the darling buds of May,
- ... And Summer's lease hath all too short a date:'''
- >>> print couplet

Rough winds do shake the darling buds of May,

And Summer's lease hath all too short a date:

- Now that we can define strings, we can try some simple operations on them. First let’s look at the + operation, known as **concatenation**
- It produces a new string that is a copy of the two original strings pasted together end-to-end. Notice that concatenation doesn’t do anything clever like insert a space between the words. We can even multiply strings

- >>> 'very' + 'very' + 'very‘
- 'veryveryvery'
- >>> 'very' * 3
- 'veryveryvery‘

- **Printing Strings**
- So far, when we have wanted to look at the contents of a variable or see the result of a calculation, we have just typed the variable name into the interpreter. We can also see the contents of a variable using the print statement:
- >>> print monty Monty Python
- Notice that there are no quotation marks this time. When we inspect a variable by typing its name in the interpreter, the interpreter prints the Python representation of its value. Since it’s a string, the result is quoted.
- However, when we tell the interpreter to print the contents of the variable, we don’t see quotation characters, since there are none inside the string.

- The print statement allows us to display more than one item on a line in various ways, as shown here:
- >>> grail = 'Holy Grail'
- >>> print monty + grail

 Monty PythonHoly Grail

- >>> print monty, grail

 Monty Python Holy Grail

- >>> print monty, "and the", grail

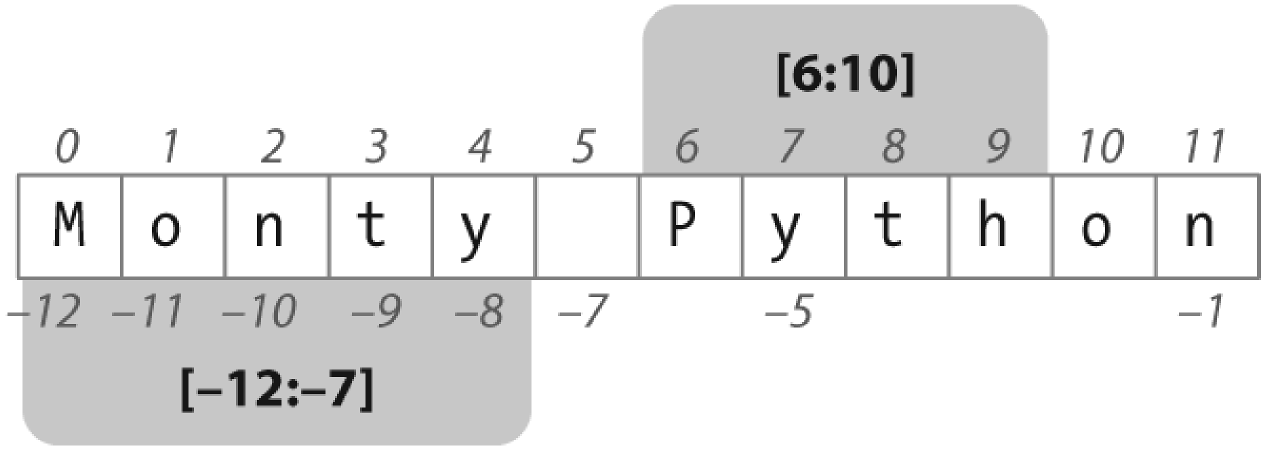
Monty Python and the Holy Grail

- **Accessing Individual Characters**
- As we saw in Section 1.2 for lists, strings are indexed, starting from zero. When we index a string, we get one of its characters (or letters). A single character is nothing special—it's just a string of length 1.
- >>> monty[0] 'M'
- >>> monty[3] 't'
- >>> monty[5] ‘ ’

- As with lists, if we try to access an index that is outside of the string, we get an error:
- >>> monty[20]
- Traceback (most recent call last):
- File "<stdin>", line 1, in ? IndexError: string index out of range
- Again as with lists, we can use negative indexes for strings, where -1 is the index of the last character.
- Positive and negative indexes give us two ways to refer to any position in a string.
- In this case, when the string had a length of 12, indexes 5 and -7 both refer to the same character (a space). (Notice that 5 = len(monty) - 7.)
- >>> monty[-1]
- 'n'
- >>> monty[5] ' '
- >>> monty[-7] ' '

- We can write for loops to iterate over the characters in strings. This print statement ends with a trailing comma, which is how we tell Python not to print a newline at the end.
- >>> sent = 'colorless green ideas sleep furiously'
- >>> for char in sent:
- ...   print char,
- ...
- c o l o r l e s s   g r e e n   i d e a s   s l e e p   f u r i o u s l y
- We can count individual characters as well. We should ignore the case distinction by normalizing everything to lowercase, and filter out non-alphabetic characters:
- >>> from nltk.corpus import gutenberg
- >>> raw = gutenberg.raw('melville-moby_dick.txt')
- >>> fdist = nltk.FreqDist(ch.lower() for ch in raw if ch.isalpha())
- >>> fdist.keys()
- ['e', 't', 'a', 'o', 'n', 'i', 's', 'h', 'r', 'l', 'd', 'u', 'm', 'c', 'w',
- 'f', 'g', 'p', 'b', 'y', 'v', 'k', 'q', 'j', 'x', 'z']

- This gives us the letters of the alphabet, with the most frequently occurring letters listed first .
- You might like to visualize the distribution using fdist.plot(). The relative character frequencies of a text can be used in automatically identifying the language of the text.
- **Accessing Substrings**
- A substring is any continuous section of a string that we want to pull out for further processing. We can easily access substrings using the same slice notation we used for lists (see Figure 3-2). For example, the following code accesses the substring starting at index 6, up to (but not including) index 10:
- >>> monty[6:10]
- 'Pyth'
-



- Here we see the characters are 'P', 'y', 't', and 'h', which correspond to monty[6] to monty[9] but not monty[10].

This is because a slice *starts* at the first index but finishes *one before* the end index.

- We can also slice with negative indexes—the same basic rule of starting from the start index and stopping one before the end index applies; here we stop before the space character.
- >>> monty[-12:-7]
- 'Monty'
- As with list slices, if we omit the first value, the substring begins at the start of the string.
- If we omit the second value, the substring continues to the end of the string:
- >>> monty[:5]
- 'Monty'
- >>> monty[6:]
- 'Python'

- We test if a string contains a particular substring using the in operator, as follows:
- >>> phrase = 'And now for something completely different'
- >>> if 'thing' in phrase:
- ... print 'found "thing"'

found "thing"

- We can also find the position of a substring within a string, using find():
- >>> monty.find('Python')
- 6
- **More Operations on Strings**
- Python has comprehensive support for processing strings. A summary, including some operations we haven’t seen yet, is shown in Table 3-2. For more information on strings,
- type help(str) at the Python prompt

Method	Functionality
s.find(t)	Index of first instance of string t inside s (-1 if not found)
s.rfind(t)	Index of last instance of string t inside s (-1 if not found)
s.index(t)	Like s.find(t), except it raises ValueError if not found
s.rindex(t)	Like s.rfind(t), except it raises ValueError if not found
s.join(text)	Combine the words of the text into a string using s as the glue
s.split(t)	Split s into a list wherever a t is found (whitespace by default)
s.splitlines()	Split s into a list of strings, one per line
s.lower()	A lowercased version of the string s
s.upper()	An uppercased version of the string s s.titlecase() A titlecased version of the string s
s.strip()	A copy of s without leading or trailing whitespace
s.replace(t, u)	Replace instances of t with u inside s

- **The Difference Between Lists and Strings**
- Strings and lists are both kinds of **sequence**. We can pull them apart by indexing and slicing them, and we can join them together by concatenating them. However, we cannot join strings and lists:
- >>> query = 'Who knows?'
- >>> beatles = ['John', 'Paul', 'George', 'Ringo']
- >>> query[2] 'o'
- >>> beatles[2] 'George'
- >>> query[:2] 'Wh'
- >>> beatles[:2] ['John', 'Paul']
- >>> query + " I don't" "Who knows? I don't"
- >>> beatles + 'Brian'
- Traceback (most recent call last):
- File "<stdin>", line 1, in <module>
- TypeError: can only concatenate list (not "str") to list
- >>> beatles + ['Brian']
- ['John', 'Paul', 'George', 'Ringo', 'Brian']

- When we open a file for reading into a Python program, we get a string corresponding to the contents of the whole file. If we use a for loop to process the elements of this string, all we can pick out are the individual characters—we don’t get to choose the granularity. By contrast, the elements of a list can be as big or small as we like: for example, they could be paragraphs, sentences, phrases, words, characters. So lists have the advantage that we can be flexible about the elements they contain, and corre- spondingly flexible about any downstream processing. Consequently, one of the first things we are likely to do in a piece of NLP code is tokenize a string into a list of strings (Section 3.7). Conversely, when we want to write our results to a file, or to a terminal, we will usually format them as a string (Section 3.9).
- Lists and strings do not have exactly the same functionality. Lists have the added power that you can change their elements:
- >>> beatles[0] = "John Lennon"
- >>> del beatles[-1]
- >>> beatles
- ['John Lennon', 'Paul', 'George']

- On the other hand, if we try to do that with a *string*—changing the 0th character in
- query to 'F'—we get:
- >>> query[0] = 'F'
- Traceback (most recent call last):
- File "<stdin>", line 1, in ?
- TypeError: object does not support item assignment
- This is because strings are **immutable**: you can’t change a string once you have created it.
- However, lists are **mutable**, and their contents can be modified at any time. As a result, lists support operations that modify the original value rather than producing a new value.

- **Text Processing with Unicode**

- Our programs will often need to deal with different languages, and different character sets.
- If you live in Europe you might use one of the extended Latin character sets, containing such characters as “ø” for Danish and Norwegian, “ő” for Hungarian, “ñ” for Spanish and Breton, and “ň” for Czech and Slovak.
- In this section, we will give an overview of how to use Unicode for processing texts that use non-ASCII character sets.

- **What Is Unicode?**
- Unicode supports over **a million characters**. Each character is assigned a number, called a **code point**. In Python, code points are written in the form \u*XXXX*, where *XXXX* is the number in four-digit hexadecimal form.
- Within a program, we can manipulate Unicode strings just like normal strings. However, when Unicode characters are stored in files or displayed on a terminal, they must be encoded as a stream of bytes.
- Some encodings (such as ASCII and Latin-2) use a single byte per code point, so they can support only a small subset of Unicode, enough for a single language. Other encodings (such as UTF-8) use multiple bytes and can represent the full range of Unicode characters.
- Text in files will be in a particular encoding, so we need some mechanism for translating it into Unicode—translation into Unicode is called **decoding**. Conversely, to write out Unicode to a file or a terminal, we first need to translate it into a suitable encoding— this translation out of Unicode is called **encoding**, and is illustrated in Figure 3-3

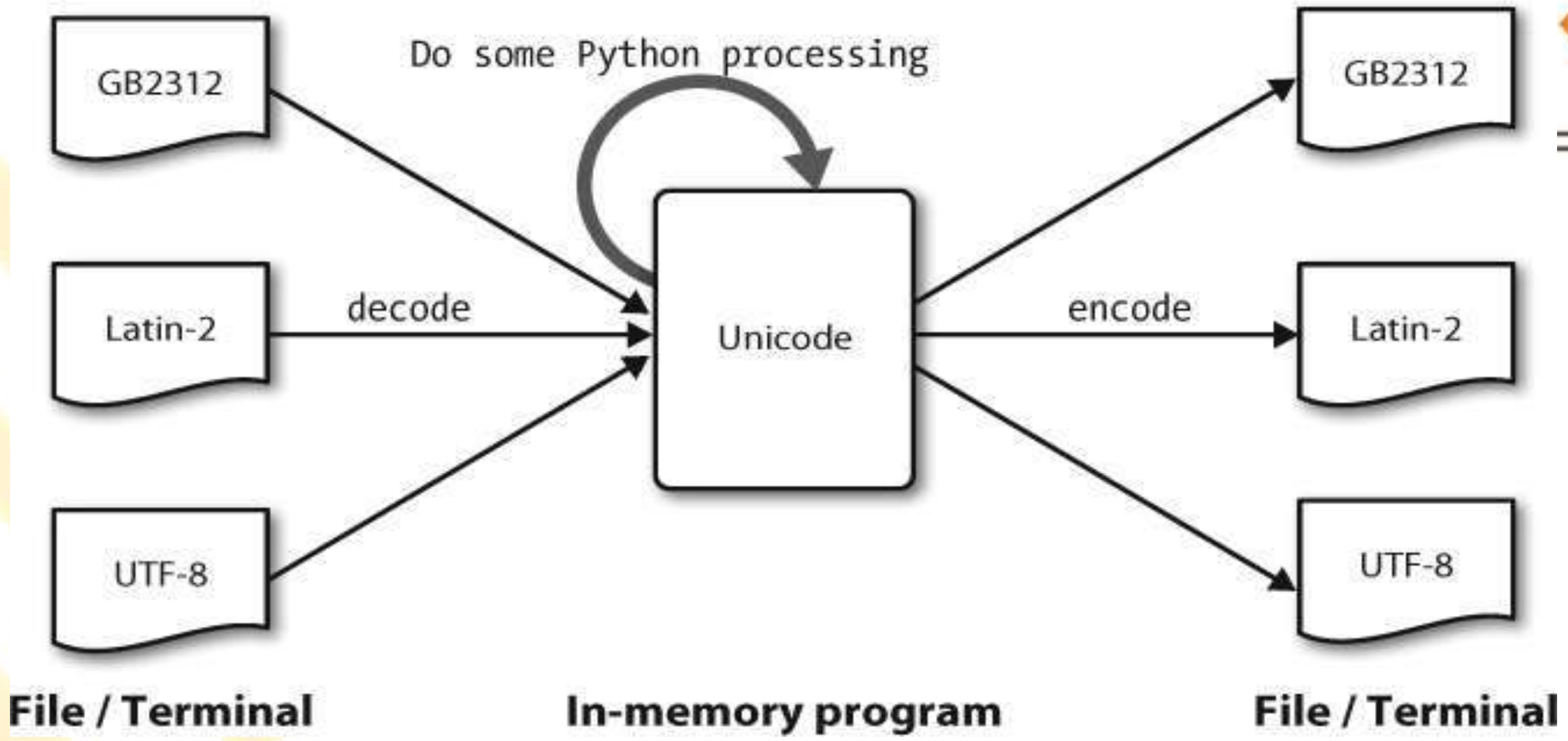


Figure 3-3. Unicode decoding and encoding.

From a Unicode perspective, characters are abstract entities that can be realized as one or more **glyphs**. Only glyphs can appear on a screen or be printed on paper. A font is a mapping from characters to glyphs.

- **Extracting Encoded Text from Files**
- Let's assume that we have a small text file, and that we know how it is encoded. For example, *polishlat2.txt*, as the name suggests, is a snippet of Polish text.
- This file is encoded as Latin-2, also known as ISO-8859-2. The function nltk.data.find() locates the file for us.
- >>> **path = nltk.data.find('corpora/unicode_samples/polishlat2.txt')**
- The Python codecs module provides functions to read encoded data into Unicode strings, and to write out Unicode strings in encoded form.
- The **codecs.open() function** takes an encoding parameter to specify the encoding of the file being read or written. So let's import the codecs module, and call it with the encoding 'latin2' to open our Polish file as Unicode:
- **>>> import codecs**
- >>> **f = codecs.open(path, encoding='latin2')**
- Note that we can write Unicode-encoded data to a file using
- f = codecs.open(path, 'w', encoding=**'utf-8').**
-

- Text read from the file object f will be returned in Unicode.
- The Python-specific encoding unicode_escape is a dummy encoding that converts all non-ASCII characters into their \u*XXXX* representations. Code points above the ASCII 0–127 range but below 256 are represented in the two-digit form \x*XX*.
- >>> for line in f:
- ...   line = line.strip()
- ...   print line.encode('unicode_escape')
- **Pruska Biblioteka Pa\u0144stwowa. Jej dawne zbiory znane pod nazw\u0105 "Berlinka" to skarb kultury i sztuki niemieckiej. Przewiezione przez**
- **Niemc\xf3w pod koniec II wojny \u015bwiatowej na Dolny \u015al\u0105sk, zosta\u0142y odnalezione po 1945 r. na terytorium Polski. Trafi\u0142y do Biblioteki Jagiello\u0144skiej w Krakowie, obejmuj\u0105 ponad 500 tys. zabytkowych archiwali\xf3w, m.in. manuskrypty Goethego, Mozarta, Beethovena, Bacha.**
- The first line in this output illustrates a Unicode escape string preceded by the \u escape string, namely \u0144. The relevant Unicode character will be displayed on the screen as the glyph ń. In the third line of the preceding example, we see \xf3, which corre- sponds to the glyph ó, and is within the 128–255 range.

- In Python, a Unicode string literal can be specified by preceding an ordinary string literal with a u, as in u'hello'. Arbitrary Unicode characters are defined using the
- \u*XXXX* escape sequence inside a Unicode string literal. We find the integer ordinal of a character using ord(). For example:
- >>> ord('a') 97
- The hexadecimal four-digit notation for 97 is 0061, so we can define a Unicode string literal with the appropriate escape sequence:
- >>> a = u'\u0061'
- >>> a
- u'a'
- >>> print a a
- The module unicodedata lets us inspect the properties of Unicode characters. In the following example, we select all characters in the third line of our Polish text outside the ASCII range and print their UTF-8 escaped value, followed by their code point integer using the standard Unicode convention (i.e., prefixing the hex digits with U+), followed by their Unicode name.
-

- >>> import unicodedata
- >>> lines = codecs.open(path, encoding='latin2').readlines()
- >>> line = lines[2]
- >>> print line.encode('unicode_escape')
- Niemc\xf3w pod koniec II wojny \u015bwiatowej na Dolny \u015al\u0105sk, zosta\u0142y\n
- >>> for c in line:
- ...   if ord(c) > 127:
- ...   print '%r U+%04x %s' % (c.encode('utf8'), ord(c), unicodedata.name(c)) '\xc3\xb3' U+00f3 LATIN SMALL LETTER O WITH ACUTE
- '\xc5\x9b' U+015b LATIN SMALL LETTER S WITH ACUTE '\xc5\x9a' U+015a LATIN CAPITAL LETTER S WITH ACUTE '\xc4\x85' U+0105 LATIN SMALL LETTER A WITH OGONEK '\xc5\x82' U+0142 LATIN SMALL LETTER L WITH STROKE

- Alternatively, you may need to replace the encoding 'utf8' in the example by 'latin2', again depending on the details of your system.
- The next examples illustrate how Python string methods and the re module accept Unicode strings.
- >>> line.find(u'zosta\u0142y')
- 54
- >>> line = line.lower()
- >>> print line.encode('unicode_escape')
- niemc\xf3w pod koniec ii wojny \u015bwiatowej na dolny \u015bl\u0105sk, zosta\u0142y\n
- >>> import re
- >>> m = re.search(u'\u015b\w*', line)
- >>> m.group()
- u'\u015bwiatowej’
- NLTK tokenizers allow Unicode strings as input, and correspondingly yield Unicode strings as output.
- >>> nltk.word_tokenize(line)
- [u'niemc\xf3w', u'pod', u'koniec', u'ii', u'wojny', u'\u015bwiatowej',
- u'na', u'dolny', u'\u015bl\u0105sk', u'zosta\u0142y']

- **Regular Expressions for Detecting Word Patterns**
- Regular expressions give us a more powerful and flexible method for describing the character patterns
- Many linguistic processing tasks involve pattern matching. For example, we can find words ending with ***ed*** using **endswith('ed').**
- **Using Basic Metacharacters:**
- Let's find words ending with *ed* using the regular expression «ed$». We will use the **re.search(p, s) function** to check whether the pattern p can be found somewhere inside the string s. We need to specify the characters of interest, and use the dollar sign, which has a special behavior in the context of regular expressions in that it matches the end of the word:
- `>>> [w for w in wordlist if re.search('ed$', w)]`
- **['abaissed', 'abandoned', 'abased', 'abashed', 'abatised', 'abed', 'aborted', ...]**
- The . **wildcard** symbol matches any single character. Suppose we have room in a crossword puzzle for an eight-letter word, with ***j* as its third letter** and ***t* as its sixth letter**. In place of each blank cell we use a period:

- >>> [w for w in wordlist if re.search('^..j..t..$', w)]
- ['abjectly', 'adjuster', 'dejected', 'dejectly', 'injector', 'majestic', ...]
- Finally, the ? symbol specifies that the previous character is optional.

  Thus «^e-?mail$»

will match both *email* and *e-mail*.

We could count the total number of occurrences of this word (in either spelling) in a text using

sum(1 for w in text if re.search('^e-? mail$', w)).

- **Ranges and Closures:**
  The **T9** system is used for entering text on mobile phones (see Figure 3-5). Two or more words that are entered with the same sequence of keystrokes are known as **textonyms**
- For example, both ***hole*** and ***golf*** are entered by pressing the **sequence 4653**. What other words could be produced with the same sequence? Here we use the regular expression «^[ghi][mno][jlk][def]$»:
- >>> [w for w in wordlist if re.search('^[ghi][mno][jlk][def]$', w)]

  ['gold', 'golf', 'hold', 'hole']

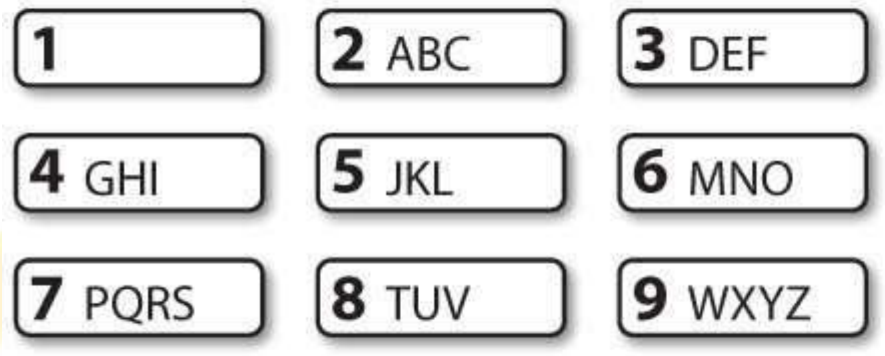


Figure 3-5. T9: Text on 9 keys.

The first part of the expression, «`^[ghi]`», matches the start of a word followed by *g*, *h*, or *i*. The next part of the expression, «`[mno]`», constrains the second character to be *m*, *n*, or *o*. The third and fourth characters are also constrained. Only four words satisfy all these constraints. Note that the order of characters inside the square brackets is not significant, so we could have written «`^[hig][nom][ljk][fed]$`» and matched the same words.

Let's explore the + symbol a bit further. Notice that it can be applied to individual letters, or to bracketed sets of letters:

```
>>> chat_words = sorted(set(w for w in nltk.corpus.nps_chat.words()))
>>> [w for w in chat_words if re.search('^m+i+n+e+$', w)]
['miiiiiiiiiiiiinnnnnnnnnnneeeeeeeeee', 'miiiiiinnnnnnnnnneeeeeeee', 'mine',
'mmmmmmmmiiiiiiiiinnnnnnnnneeeeeeee']
>>> [w for w in chat_words if re.search('^[ha]+$', w)]
['a', 'aaaaaaaaaaaaaaaaa', 'aaahhhh', 'ah', 'ahah', 'ahahah', 'ahh',
'ahhahahaha', 'ahhh', 'ahhhh', 'ahhhhhh', 'ahhhhhhhhhhhhhh', 'h', 'ha', 'haaa',
'hah', 'haha', 'hahaaa', 'hahah', 'hahaha', 'hahahaa', 'hahahah', 'hahahaha', ...]
```

- It should be clear that + simply means “one or more instances of the preceding item,” which could be an individual character like m, a set like [fed], or a range like [d-f].
- Now let’s replace + with *, which means “zero or more instances of the preceding item.”
- The regular expression «^m*i*n*e*$» will match everything that we found using «^m+l+n+e+$», but also words where some of the letters don’t appear at all, e.g., *me*, *min*, and *mmmmm*. Note that the + and * symbols are sometimes referred to as **Kleene clo- sures**, or simply **closures**
- The ^ operator has another function when it appears as the first character inside square brackets. For example, «[^aeiouAEIOU]» matches any character other than a vowel. We can search the NPS Chat Corpus for words that are made up entirely of non-vowel characters using «^[^aeiouAEIOU]+$» to find items like these: :):):), grrr, cyb3r, and zzzzzzzz. Notice this includes non-alphabetic characters.
- examples of regular expressions being used to find tokens that match a particular pattern, illustrating the use of some new symbols: \, {}, (), and |.

- >>> wsj = sorted(set(nltk.corpus.treebank.words()))
- >>> [w for w in wsj if re.search('^[0-9]+\.[0-9]+$', w)]
- ['0.0085', '0.05', '0.1', '0.16', '0.2', '0.25', '0.28', '0.3', '0.4', '0.5',
- '0.50', '0.54', '0.56', '0.60', '0.7', '0.82', '0.84', '0.9', '0.95', '0.99',
- '1.01', '1.1', '1.125', '1.14', '1.1650', '1.17', '1.18', '1.19', '1.2', ...]
- >>> [w for w in wsj if re.search('^[A-Z]+\$$', w)] ['C$', 'US$']
- >>> [w for w in wsj if re.search('^[0-9]{4}$', w)]
- ['1614', '1637', '1787', '1901', '1903', '1917', '1925', '1929', '1933', ...]
- >>> [w for w in wsj if re.search('^[0-9]+-[a-z]{3,5}$', w)]
- ['10-day', '10-lap', '10-year', '100-share', '12-point', '12-year', ...]
- >>> [w for w in wsj if re.search('^[a-z]{5,}-[a-z]{2,3}-[a-z]{,6}$', w)] ['black-and-white', 'bread-and-butter', 'father-in-law', 'machine-gun-toting', 'savings-and-loan']
- >>> [w for w in wsj if re.search('(ed|ing)$', w)]
- ['62%-owned', 'Absorbed', 'According', 'Adopting', 'Advanced', 'Advancing', ...]

- The metacharacters we have seen are summarized in Table 3-3.
- *Table 3-3. Basic regular expression metacharacters, including wildcards, ranges, and closures*
- **Operator Behavior**
- . Wildcard, matches any character
- ^abc Matches some pattern *abc* at the start of a string abc$ Matches some pattern *abc* at the end of a string [abc] Matches one of a set of characters
- [A-Z0-9] Matches one of a range of characters
- ed|ing|s Matches one of the specified strings (disjunction)
- * Zero or more of previous item, e.g., a*, [a-z]* (also known as *Kleene Closure*)
- + One or more of previous item, e.g., a+, [a-z]+
- ? Zero or one of the previous item (i.e., optional), e.g., a?, [a-z]?
- {n} Exactly *n* repeats where *n* is a non-negative integer
- {n,} At least *n* repeats
- {,n} No more than *n* repeats
- {m,n} At least *m* and no more than *n* repeats
- a(b|c)+ Parentheses that indicate the scope of the operators
-

Useful Applications of Regular Expressions

- The previous examples all involved searching for words *w* that match some regular expression *regexp* using re.search(regexp, w). Apart from checking whether a regular expression matches a word, we can use regular expressions to extract material from words, or to modify words in specific ways.
- **Extracting Word Pieces**
- The re.findall() ("find all") method finds all (non-overlapping) matches of the given regular expression. Let's find all the vowels in a word, then count them:
- >>> word = 'supercalifragilisticexpialidocious'
- >>> re.findall(r'[aeiou]', word)
- ['u', 'e', 'a', 'i', 'a', 'i', 'i', 'i', 'e', 'i', 'a', 'i', 'o', 'i', 'o', 'u']
- >>> len(re.findall(r'[aeiou]', word)) 16

- Let’s look for all sequences of two or more vowels in some text, and determine their relative frequency:
- >>> wsj = sorted(set(nltk.corpus.treebank.words()))
- >>> fd = nltk.FreqDist(vs for word in wsj
- ...   for vs in re.findall(r'[aeiou]{2,}', word))
- >>> fd.items()
- [('io', 549), ('ea', 476), ('ie', 331), ('ou', 329), ('ai', 261), ('ia', 253),
- ('ee', 217), ('oo', 174), ('ua', 109), ('au', 106), ('ue', 105), ('ui', 95),
- ('ei', 86), ('oi', 65), ('oa', 59), ('eo', 39), ('iou', 27), ('eu', 18), ...]

- **Finding Word Stems:**
- There are various ways we can pull out the stem of a word. Here's a simple-minded approach that just strips off anything that looks like a suffix:
- `>>> def stem(word):`
- `...     for suffix in ['ing', 'ly', 'ed', 'ious', 'ies', 'ive', 'es', 's', 'ment']:`
- `...     if word.endswith(suffix):`
- `...     return word[:-len(suffix)]`
- `...     return word`
- Although we will ultimately use NLTK's built-in stemmers, it's interesting to see how we can use regular expressions for this task. Our first step is to build up a disjunction of all the suffixes. We need to enclose it in parentheses in order to limit the scope of the disjunction.
- `>>> re.findall(r'^.*(ing|ly|ed|ious|ies|ive|es|s|ment)$', 'processing') ['ing']`
- `>>> re.findall(r'^.*(?:ing|ly|ed|ious|ies|ive|es|s|ment)$', 'processing') ['processing']`

- However, we'd actually like to split the word into stem and suffix. So we should just parenthesize both parts of the regular expression:
- >>> re.findall(r'^(.*)(ing|ly|ed|ious|ies|ive|es|s|ment)$', 'processing') [('process', 'ing')]
- This looks promising, but still has a problem. Let's look at a different word, *processes*:
- >>> re.findall(r'^(.*)(ing|ly|ed|ious|ies|ive|es|s|ment)$', 'processes') [('processe', 's')]
- The regular expression incorrectly found an *-s* suffix instead of an *-es* suffix. This dem- onstrates another subtlety: the star operator is "greedy" and so the .* part of the ex- pression tries to consume as much of the input as possible. If we use the "non-greedy" version of the star operator, written *?, we get what we want:
- >>> re.findall(r'^(.*?)(ing|ly|ed|ious|ies|ive|es|s|ment)$', 'processes') [('process', 'es')]
- This works even when we allow an empty suffix, by making the content of the second parentheses optional:
- >>> re.findall(r'^(.*?)(ing|ly|ed|ious|ies|ive|es|s|ment)?$', 'language') [('language', '')]

- This approach still has many problems (can you spot them?), but we will move on to define a function to perform stemming, and apply it to a whole text:
- `>>> def stem(word):`
- `...     regexp = r'^(.*?)(ing|ly|ed|ious|ies|ive|es|s|ment)?$'`
- `...     stem, suffix = re.findall(regexp, word)[0]`
- `...     return stem`
- `...`
- `>>> raw = """DENNIS: Listen, strange women lying in ponds distributing swords`
- `... is no basis for a system of government. Supreme executive power derives from`
- `... a mandate from the masses, not from some farcical aquatic ceremony."""`
- `>>> tokens = nltk.word_tokenize(raw)`
- `>>> [stem(t) for t in tokens]`
- `['DENNIS', ':', 'Listen', ',', 'strange', 'women', 'ly', 'in', 'pond',`
- `'distribut', 'sword', 'i', 'no', 'basi', 'for', 'a', 'system', 'of', 'govern',`
- `'.', 'Supreme', 'execut', 'power', 'deriv', 'from', 'a', 'mandate', 'from',`
- `'the', 'mass', ',', 'not', 'from', 'some', 'farcical', 'aquatic', 'ceremony', '.']`

- **Searching Tokenized Text**
- You can use a special kind of regular expression for searching across multiple words in a text (where a text is a list of tokens). For example, "<a> <man>" finds all instances of *a man* in the text. The angle brackets are used to mark token boundaries, and any whitespace between the angle brackets is ignored (behaviors that are unique to NLTK's findall() method for texts).
- In the following example, we include <.*>, which will match any single token, and enclose it in parentheses so only the matched word (e.g., *monied*) and not the matched phrase (e.g., *a monied man*) is produced. The second example finds three-word phrases ending with the word *bro*
- The last example finds sequences of three or more words starting with the letter *l*
- >>> from nltk.corpus import gutenberg, nps_chat
- >>> moby = nltk.Text(gutenberg.words('melville-moby_dick.txt'))
- >>> moby.findall(r"<a> (<.*>) <man>") monied; nervous; dangerous; white; white; white; pious; queer; good; mature; white; Cape; great; wise; wise; butterless; white; fiendish; pale; furious; better; certain; complete; dismasted; younger; brave; brave; brave; brave
- >>> chat = nltk.Text(nps_chat.words())
- >>> chat.findall(r"<.*> <.*> <bro>")
- you rule bro; telling you bro; u twizted bro
- >>> chat.findall(r"<l.*>{3,}")
- lol lol lol; lmao lol lol; lol lol lol; la la la la la; la la la; la la la; lovely lol lol love; lol lol lol.; la la la; la la la