22-06-2023

**Chapter 3: Processing Raw Text (Part 1)**

1. **Accessing Text from the Web and from Disk:**
2. Electronic Books:

For 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. Texts found on the Web may contain unwanted material, and there may not be an automatic way to remove it. But with a small amount of extra work we can extract the material we need.

1. 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, then access this as described in the later section on files. 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 URL open.

1. 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 engines 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.

Unfortunately, search engines have some significant shortcomings. First, the allowable range of search patterns is severely restricted. Unlike local corpora, where you write programs to search for arbitrarily complex patterns, search engines generally only allow you to search for individual words or strings of words, sometimes with wildcards. Second, search engines give inconsistent results, and can give widely different figures when used at different times or in different geographical regions. When content has been duplicated across multiple sites, search results may be boosted. Finally, the markup in the result returned by a search engine may change unpredictably, breaking any patternbased method of locating particular content (a problem which is ameliorated by the use of search engine APIs).

1. 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/, we can access the content of a blog.

1. Reading Local Files:

In order to read a local file, we need to use Python’s built-in open () function, followed by the read () method.

To check that the file that you are trying to open is really in the right directory, use IDLE’s Open command in the File menu; this will display a list of all the files in the directory where IDLE is running. An alternative is to examine the current directory from within Python.

1. Extracting Text from PDF, MS word, 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 software. 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.

1. Capturing user input:

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.

1. **Strings: Text Processing at the lowest level:**

Topics under this section such as:

* Basic operations with Strings
* Printing Strings
* Accessing Individual Characters
* Accessing substrings
* Additional operations on strings

All these topics have been practically implemented in the notebook (python file)

1. **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 \uXXXX, 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.
* 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.**
* 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.

1. **Regular expressions for detecting word patterns:**

Regular expressions give us a more powerful and flexible method for describing the character patterns we are interested in.

This topic has been practically implemented in the notebook.

1. **Useful Applications of Regular Expressions:**

**1. Extracting Word Pieces:** The `re.findall()` method is used to find all non-overlapping matches of a regular expression in a word. For example, finding all the vowels in a word and counting them.

**2. Finding Sequences of Two or More Vowels:** The text demonstrates finding sequences of two or more vowels in words and determining their relative frequency. It uses the `re.findall()` method in combination with a conditional frequency distribution (`nltk.FreqDist`) to achieve this.

**3. Manipulating Dates:** The text presents a task where the W3C Date Time Format is given, and the goal is to convert the date string '2009-12-31' into a list of integers [2009, 12, 31]. It asks to replace the "?" in the given Python code with a regular expression to achieve the desired result.

**4. Gluing Word Pieces:** Regular expressions can be used to extract specific parts of words and then join them together. The example provided shows how to extract initial vowel sequences, final vowel sequences, and all consonants from words, and then join them using `re. findall ()` and `''.join()`.

**5. Conditional Frequency Distributions:** Regular expressions can be combined with conditional frequency distributions to perform operations on word pieces. The text demonstrates extracting consonant-vowel sequences from words and creating a conditional frequency distribution (`nltk. ConditionalFreqDist`) to tabulate the frequencies of each pair.

**6. Finding Word Stems:** The text explains the concept of word stemming, which involves removing word endings to focus on the word stems or lemmas. It demonstrates a simple stemming approach using regular expressions to strip off known suffixes from words.

**7. Searching Tokenized Text:** Regular expressions can be used to search across multiple words in a text. The text presents examples of using special regular expressions with NLTK's `findall()` method to search for specific patterns in a text, such as finding instances of "a man" or three-word phrases ending with "bro".

**8. NLTK Tools and Applications:** The text mentions NLTK's built-in stemmers and provides an example of using NLTK's stemmer to stem words in a given text.