# CS463: Lab 2: Processing Raw Text with Python

# Introduction

The most important source of texts is undoubtedly the Web. It's convenient to have existing text collections to explore, such as the corpora from project Gutenberg (is a library of over 60,000 free eBooks). However, you probably have your own text sources in mind, and need to learn how to access them.

The goal of this lab is to answer the following questions:

* 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?
* How can we split documents up into individual words and punctuation symbols, so we can carry out the same kinds of analysis we did with text corpora in earlier chapters?
* How can we write programs to produce formatted output and save it in a file?

**Useful String Methods**: operations on strings in addition to the string tests shown

| **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.title() | 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 |

## Accessing Text from the Web and from Disk

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 section on files below. 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:essing Text from the Web and from Disk

## A small sample of texts from Project Gutenberg appears in the NLTK corpus collection. However, you may be interested in analyzing other texts from Project Gutenberg.

from urllib import request

nltk.download('punkt')

url = "http://www.gutenberg.org/files/2554/2554-0.txt"

response = request.urlopen(url)

raw = response.read().decode('utf8')

tokens = nltk.word\_tokenize(raw)

tokens[:5]

## output:

[nltk\_data] Downloading package punkt to /root/nltk\_data...

[nltk\_data] Unzipping tokenizers/punkt.zip.

['\ufeffThe', 'Project', 'Gutenberg', 'eBook', 'of']

### 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 section on files below. 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: To get text out of HTML we will use a Python library called BeautifulSoup,

from bs4 import BeautifulSoup

url = "http://news.bbc.co.uk/2/hi/health/2284783.stm"

html = request.urlopen(url).read().decode('utf8')

raw = BeautifulSoup(html, 'html.parser').get\_text()

tokens = nltk.word\_tokenize(raw)

tokens[:5]

### Output:

### ['BBC', 'NEWS', '|', 'Health', '|']

### 

### 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. Suppose you have a file document.txt, you can load its contents like this:

f = open('document.txt')

raw = f.read()

## Regular Expressions for Detecting Word Patterns

## Many linguistic processing tasks involve pattern matching. For example, we can find words ending with *ed* using endswith('ed'. Regular expressions give us a more powerful and flexible method for describing the character patterns we are interested in. To use regular expressions in Python we need to import the re library using: import re. We also need a list of words to search; we'll use the Words Corpus again. We will preprocess it to remove any proper names.

import re

import nltk

from nltk.corpus import words

nltk.download('words')

wordlist = words.words()

wordlist = [w for w in wordlist if re.search('ed$', w)]

wordlist[:5]

### Output:

### 

**Basic Regular Expression Meta-Characters**, 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 |

**Question**: The . **wildcard** symbol matches any single character. Suppose we have room in a crossword puzzle for an 8-letter word with  *n* as its third letter and *t* as its sixth letter. Write a python code to find list of word that math the previous criteria

### Finding Word Stems

### When we use a web search engine, we usually don't mind (or even notice) if the words in the document differ from our search terms in having different endings. A query for laptops finds documents containing laptop and vice versa. Indeed, laptop and laptops are just two forms of the same dictionary word (or lemma). For some language processing tasks we want to ignore word endings, and just deal with word stems. Here, re.findall() just gave us the suffix even though the regular expression matched the entire word.

**X=re.findall(r'^.\*(ing|ly|ed|ious|ies|ive|es|s|ment)$', 'processing')**

**Output**:

This is because the parentheses have a second function, to select substrings to be extracted. If we want to use the parentheses to specify the scope of the disjunction, but not to select the material to be output, we have to add ?:

**re.findall(r'^.\*(?:ing|ly|ed|ious|ies|ive|es|s|ment)$', 'processing')**

**output**:

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')**

**Output**:

**Question**:

Write a stem method which take a word as input and return its stem.

### Normalizing Text

In earlier program examples we have often converted text to lowercase before doing anything with its words, e.g. set(w.lower() for w in text). By using lower(), we have **normalized** the text to lowercase so that the distinction between *The* and *the* is ignored. Often we want to go further than this, and strip off any affixes, a task known as stemming. A further step is to make sure that the resulting form is a known word in a dictionary, a task known as lemmatization.

### Stemmers

|  |  |
| --- | --- |
| **from nltk import word\_tokenize**   |  | | --- | | **raw = 'listening, hardly, wlaks, walked'**  **tokens = nltk.word\_tokenize(raw)**  **porter = nltk.PorterStemmer()**  **[porter.stem(t) for t in tokens]** | |

### Lemmatization

The WordNet lemmatizer only removes affixes if the resulting word is in its dictionary. This additional checking process makes the lemmatizer slower than the above stemmers. Notice that it doesn't handle *lying*, but it converts *women* to *woman*.

**wnl = nltk.WordNetLemmatizer()**

**[wnl.lemmatize(t) for t in tokens]**

## 

## Regular Expressions for Tokenizing Text

raw="In earlier stages of life, I'm having 23.56$ and a big house."

w1=re.split(r' ', raw)

print(w1)

w2=re.split(r'\W+', raw)

print(w2)

w3= re.findall(r"\w+(?:[-']\w+)\*|'|[-.(]+|\S\w\*", raw)

print(w3)

**Output**:

### NLTK's Regular Expression Tokenizer

The function nltk.regexp\_tokenize() is similar to re.findall() (as we've been using it for tokenization). However, nltk.regexp\_tokenize() is more efficient for this task, and avoids the need for special treatment of parentheses. For readability we break up the regular expression over several lines and add a comment about each line. The special (?x) "verbose flag" tells Python to strip out the embedded whitespace and comments.

text = 'That U.S.A. poster-print costs $12.40...'

pattern = r'''(?x)     # set flag to allow verbose regexps

 (?:[A-Z]\.)+       # abbreviations, e.g. U.S.A.

 | \w+(?:-\w+)\*       # words with optional internal hyphens

 | \$?\d+(?:\.\d+)?%? # currency and percentages, e.g. $12.40, 82%

 | \.\.\.             # ellipsis

 | [][.,;"'?():-\_`]   # these are separate tokens; includes ], [

 '''

nltk.regexp\_tokenize(text, pattern)

**Output**:

## Writing Results to a File

We have seen how to read text from files. It is often useful to write output to files as well. The following code opens a file output.txt for writing, and saves the program output to the file.

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | **output\_file = open('output.txt', 'w')**  **words = set(nltk.corpus.genesis.words('english-kjv.txt'))**  **for word in sorted(words):**  **print(word, file=output\_file)** | |