03

Python for (Corpus) Linguists

Python Programming for Linguists

Ingo Kleiber, 2022

Notebook and Solutions

We are going to solve the exercises in an (almost) empty notebook. This will, most likely, become a bit messy ….

Therefore, after this workshop, I'll provide you (via *GitHub*) with a clean(er) documented solutions notebook.

Also: Have a look at the additional notebooks and exercises. Some of the problems/exercises will be discussed and solved there in more detail.

This Session

1. Concordancer
2. N-Grams
3. Frequency Analysis
4. Computing Basic Statistics
5. Basic Collocation Analysis
6. NLTK Stemming, Lemmatization, and WordNet
7. spaCy Tagging
8. Parsing XML
9. Web Scraping
10. Keyword Analysis

→ For some of these exercises, we will find **two solutions**. First, we will be using **well-established libraries and tools** (e.g., NLTK and spaCy). Then we are going to implement solutions in **plain (more-or-less) Python** in order to understand how these things work under the hood.

Some New Tools / Hints

- Importing
- Classes and Methods
- List Comprehensions
- Pandas and DataFrames
- Enumerate
- TextDirectory (Refresher)
- Bonus: ftfy

There will also be some *additional new things* that we will explore while solving the exercises.

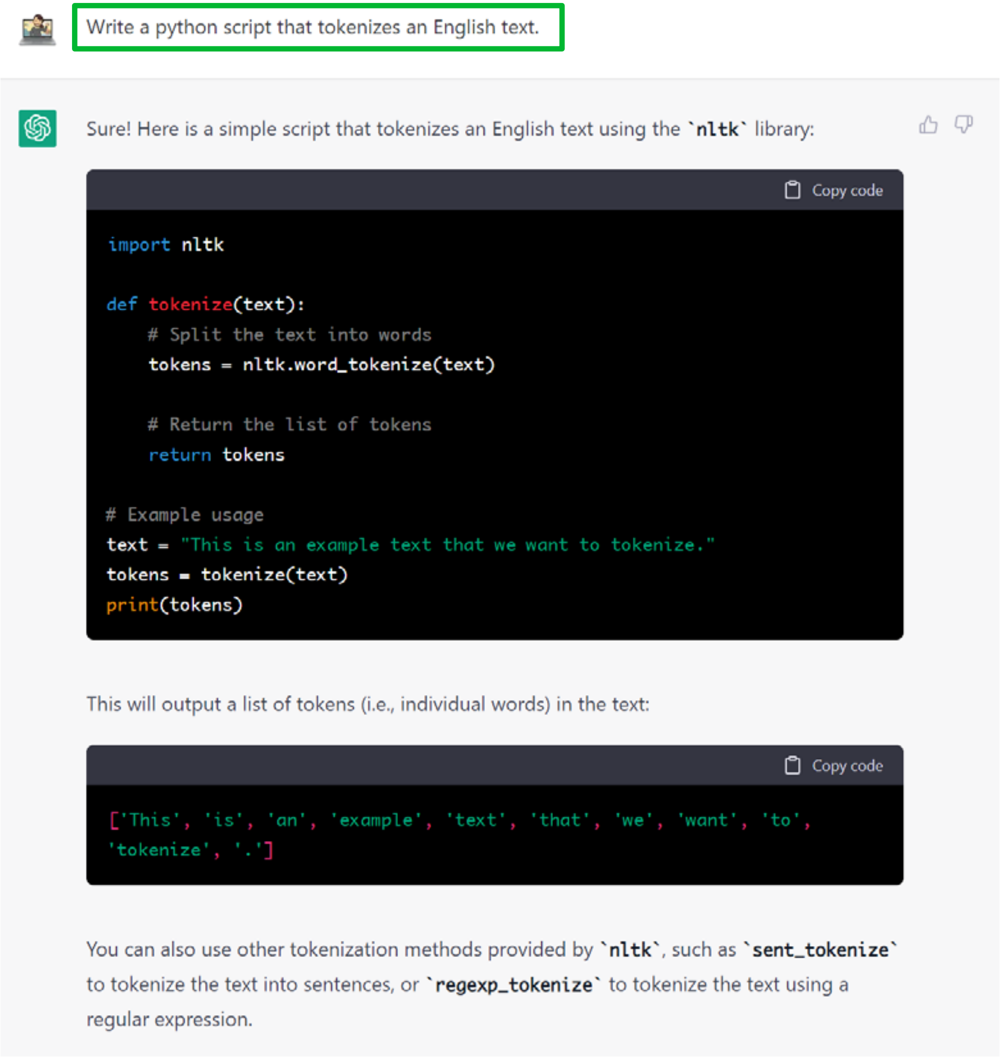
ChatGPT and Similar LLMs

Many *Large Language Models* (LLMs) have the capacity to produce (working) code based on a prompt – a sort of instruction.

Most recently (as of December 2022), OpenAI's ChatGPT has demonstrated incredible capabilities in both "writing" and "coding."

However, keep in mind: While extremely powerful and helpful, these systems are unreliable, and there are significant legal and ethical questions still unanswered!

Prompt



Importing

```
import nltk
nltk.stem.PorterStemmer()
```

Importing the whole library

```
from nltk.stem import PorterStemmer
PorterStemmer()
```

Importing just a specific thing

```
import pandas as pd
pd.DataFrame()
```

Importing the whole thing under a shorthand. This is very useful if you use something very often.

Classes and Methods

```
class Word():

  def __init__(self, word):
    self.word = word
    self.length = len(word)

  def reverse(self):
    self.word = self.word[::-1]
```

Classes are basically “blueprints” for objects

```
new_word = Word('cat')

  new_word.reverse()
  new_word.word

another_word = Word('dog')
```

Object/Instantiation

List Comprehensions

```
numbers = [10, 20, 30]

times_ten = [n * 10 for n in numbers]
```

List Comprehension

This is equal to:

```
times_ten = []
for n in numbers:
    times_ten.append(n * 10)
```

List Comprehensions

```
lol = [
        ['A', 1],
        ['B', 2],
        ['C', 3]
]

only_first_element= [n[1] for n in lol]
```

[1, 2, 3]

Enumerate

```
l = ['A', 'B', 'C']

for index, value in enumerate(l):
    print(index, value)
```

0 A
1 B
2 C

ftfy – Fixing Unicode

ftfy by Robyn Speer is an incredibly simple (to use) and useful tool for fixing problems with Unicode.

ftfy.fix_text(‘âœ” No problems‘)

'✔ No problems'

Pandas and DataFrames

Pandas is a very powerful data analysis and manipulation tool/library. The key component are `DataFrame` objects which are essentially very powerful tables.

Axis 1 (rows)

Index

Columns

Document	Tokens	Sentiment
0	1000	0.2
1	2000	0.3
2	3000	0.8
3	3000	

Axis 0 (columns)

df = pd.DataFrame(...)

df['Tokens'].mean()

→ *2250.0*

TextDirectory Refresher

TextDirectory is a library that is useful when working with multiple text files in one directory. We can filter files based on various criteria and also run transformations (e.g., transforming the corpus to lowercase) on the texts.

wikipedia = textdirectory.TextDirectory(directory='data/wikipedia', autoload=True)	Load all files in the directory `data/wikipedia`.
wikipedia.filter_by_random_sampling(10)	Reduce the selection to 10 randomly sampled files
wikipedia.stage_transformation(['transformation_lowercase'])	Schedule/stage that all files (texts) are being transformed to lowercase
text = wikipedia.aggregate_to_memory()	Run the transformation and aggregate all documents into one string

Exercises

Exercise 8 – Concordancer

Write a basic concordancer that can generate concordances based on a given file and a given search term. If you want to challenge yourself, try to format the concordances in KWIC format.

RegEx-Based Approach

We will use a **regular expression** to find all instances of the search term as well as 25 characters before and after (left and right).

Token-Based Approach

We will **tokenize** the text so that we can define a window/span in terms of tokens (words) instead of characters. We will then generate a left and right window to print KWIC concordances.

join

We can use `.join()` to turn an *iterable* into a string.

Iterables are sequences that can be iterated over using, for example, a for-loop.

These include, for example, *lists*, *sets*, and *strings*.

tokens = [‘The’, ‘cat’, ‘is’, ‘grey’]

s1 = ‘ ‘.join(tokens)   *The cat is grey*

s2 = ‘-‘.join(tokens)   *The-cat-is-grey*

Slicing Tokens

text_tokenized = [‘the’, ‘cat’, ‘is’, ‘grey’, ‘and’, ‘likes’, ‘mice’]

search_word = ‘grey’

lr = 2

Let’s call this index (for the search term) *id*

0	1	2	3	4	5	6
the	cat	is	**grey**	and	likes	mice

text_tokenized[id – lr: id]
1 3

text_tokenized[id + 1: id + lr]
4 5

Exercise 9 – N-Grams

Write a function that produces all n-grams based on a given text file and an *n*. *Hint:* The NLTK provides a fairly easy solution to generating n-grams.

NLTK Approach

NLTK has an `ngram` method that allows us to generate n-grams very easily.

Plain Old Python

In order to generate n-grams ourselves, we need to know that the number of n-grams will be *the number of tokens + 1 – n*. Once we know how many n-grams there are, we can create a loop that appends the n-grams, which we get by slicing the tokenized text, to a list of n-grams.

Plain Old Python

text = 'I really like Python, it is pretty awesome.'

n = 3

There are **six trigrams** here.

6

```
for i in range(no_of_ngrams):
    print(tokenized_text[i:i+n])
```

i	**tokenized_text[i:i+n]**
0	['I', 'really', 'like']
1	['really', 'like', 'Python']
2	['like', 'Python', 'it']
3	['Python', 'it', 'is']
…	…

1 2 3

['I', 'really', 'like', 'Python', 'it', 'is', 'pretty', 'awesome']

1: 1+n

1: 4

Exercise 10 – Frequency Analysis

Write a script that generates a frequency table for a given text. The list should contain all types and their frequencies. *Hint:* Have a look at Python's Counter capabilities.

Counter Approach

We can also use Python's `Counter` to count all elements in an iterable (i.e., a list of tokens).

NLTK Approach

We can use *NLTK*'s `FreqDist` to generate frequency distributions of tokenized texts.

spaCy Approach

After creating a *spaCy* document (see Exercise 14), we can use the `.count_by()` method to get frequency distributions.

Counter

`Counter` can be used to count hashable objects (e.g., a list). The resulting counter object behaves a lot like a dictionary and contains the individual elements as well as their counts.

numbers = [1, 1, 2, 3, 3, 4]

counts = Counter(numbers)

counts[1] → *2*

counts.most_common(2) → *[(1, 2), (3, 2)]*

spaCy Documents

A *spaCy* `Doc` is a sequence of `Token` objects.

The `Vocab` (of a `Language`) contains all `Lexeme` objects and other shared data.

[2]
```python
import spacy
```

[19]
```python
language = spacy.load('en_core_web_sm')

print(type(language))
```

```
<class 'spacy.lang.en.English'>
```

[25]
```python
lexeme = language.vocab.__getitem__('Hello')

print(type(lexeme))
print(lexeme.text, lexeme.orth)
```

```
<class 'spacy.lexeme.Lexeme'>
Hello 15777305708150031551
```

[24]
```python
document = language('Hello World')
print(type(document))
print(document)
```

```
<class 'spacy.tokens.doc.Doc'>
Hello World
```

[23]
```python
token = document[0]
print(type(token))
print(token)
```

```
<class 'spacy.tokens.token.Token'>
Hello
```

Exercise 11 – Computing Basic Statistics

Write a script that generates the following statistics for a given search term and a set of text files (a corpus): The absolute and relative frequencies; the mean frequency; the standard deviation. Also try to plot the frequency distribution across files.

Basic Approach

We define two functions for getting the absolute and relative frequencies of a given text. Then we are using a third function to generate frequencies for a number of texts which we will store in a list. Finally, we can use Python's `statistics` functions to get the required statistics.

Pandas DataFrame Approach

After getting the **vocabulary** of the corpus, we use one of the functions from above to populate two frequency tables. Then we create *Pandas* DataFrames from these tables.

Lists and Sets

Sets, in the mathematical sense, are well-defined collections of distinct elements.

```
list_with_duplicate = ['A', 'B', 'B', 'C']
s = set(list_with_duplicate)      → {'A', 'B', 'C'}
```

In Python, sets are **unordered** and only **contain unique elements**.

While sets can be used for many things (especially when leveraging set theory), we will simply use them to turn a *list of tokens* (containing duplicates) into a *set of types*.

Vocabulary

In NLP it is very common to store the **vocabulary** (essentially a list/set of types) in a data structure separate from everything else. Aside from some other benefits, this avoids duplication and reduces memory cost. *Here's a very simplistic example:*

Index	**0**	**1**	**2**	**3**	**4**
vocabulary =	['the', 'grey', 'cat', 'is', 'black']				

```
v = vocabulary
sentence = [ v[0], v[2], v[3], v[4] ]
```

Vocabulary

In NLP it is very common to store the **vocabulary** (essentially a list/set of types) in a data structure separate from everything else. Aside from some other benefits, this avoids duplication and reduces memory cost. *Here's a very simplistic example:*

Index	**0**	**1**	**2**	**3**	**4**
vocabulary =	['the', 'grey', 'cat', 'is', 'black']				

We can use the same vocabulary to index frequency tables.

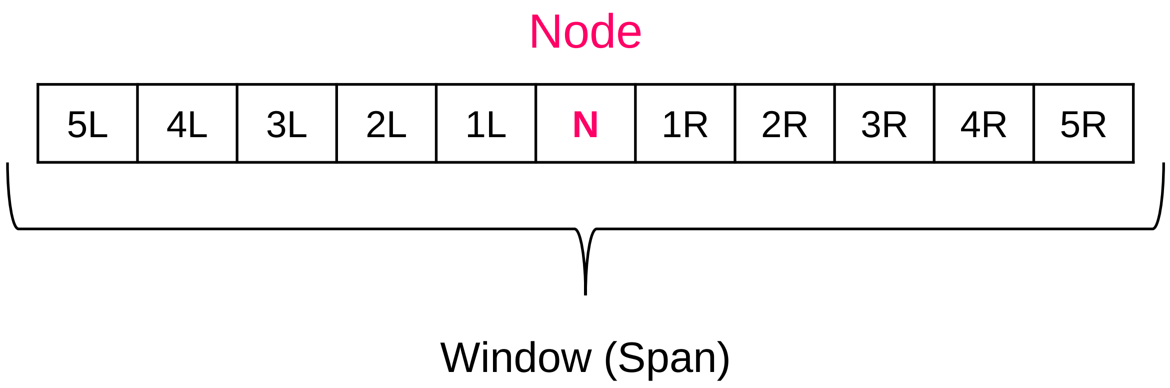
document = 'The cat is grey. The cat is **black**'

frequencies = {0: 2, 1: 1, 2: 2, 3: 2, **4**: 1}

Type *vocabulary[i]*	**Frequency**	
0	2	*the*
1	1	*grey*
2	2	*cat*
3	2	*is*
4	1	*black*

Exercise 12 – Basic Collocation Analysis

Write a function that allows you to find collocates of a given word in a given text file. If you want to do this from scratch, you will have to implement a collocation/association measure of your choice.



Which words (tokens) appear frequently within the node in the window?

NLTK Approach

We are using *NLTK* to generate "Collocations". However, these collocations are somewhat different from what we are used to in CL.

From Scratch

We are implementing the 'traditional' approach to collocation using MI scores.

Basic Collocation Analysis

Node

5L	4L	3L	2L	1L	**N**	1R	2R	3R	4R	5R

Usually, we consider **three things**: a node word, a possible collocate (candidate), and a specific window.

1. Find all instances of node in the corpus
2. For each instance, count the appearances of the candidate in the given window
3. Calculate an MI (Mutual Information) score for the candidate
4. Repeat this process for all possible candidates (= every word in the vocabulary)
5. Report the 'top' candidates based on MI-score and frequency

$$MI = \log_2 \frac{O_{11}}{E_{11}} = \log_2 \frac{5}{0.5} = \mathbf{10}$$

W2 (Node) W1 (Candidate)

	W_1 ***Present***	W_1 ***Absent***	***Totals***
W_2 ***Present***	**5** (O_{11})	45 (O_{12})	**50** (R_1)
W_2 ***Absent***	35 (O_{21})	9,915 (O_{22})	9,550 (R_2)
Totals	**40** (C_1)	9,960 (C_2)	**10,000** (N)

	W_1 ***Present***	W_1 ***Absent***
W_2 ***Present***	$E_{11} = \frac{R_1 * C_1}{N} = 0.5$	$E_{12} = \frac{R_1 * C_2}{N} = 99.6$
W_2 ***Absent***	$E_{21} = \frac{R_2 * C_1}{N} = 38.2$	$E_{22} = \frac{R_2 * C_2}{N} = 9511.8$

N: *Tokens in the corpus*
R_1: *Frequency of* W_2
C_1: *Frequency of* W_1
O_{11}: *Frequency of the candidate in the window*

Exercise 13 – NLTK Stemming, Lemmatization, and WordNet

Use NLTK to stem and lemmatize the following words. Use the PorterStemmer, the LancasterStemmer, and the WordNetLemmatizer and compare your results. What are the pros and cons of these approaches?

words = ['connection', 'become', 'caring', 'are', 'women', 'driving']

Of course, feel free to add more examples! Since you already have WordNet, try to find the synonyms for *fantastic* using WordNet.

Stemming and Lemmatizing

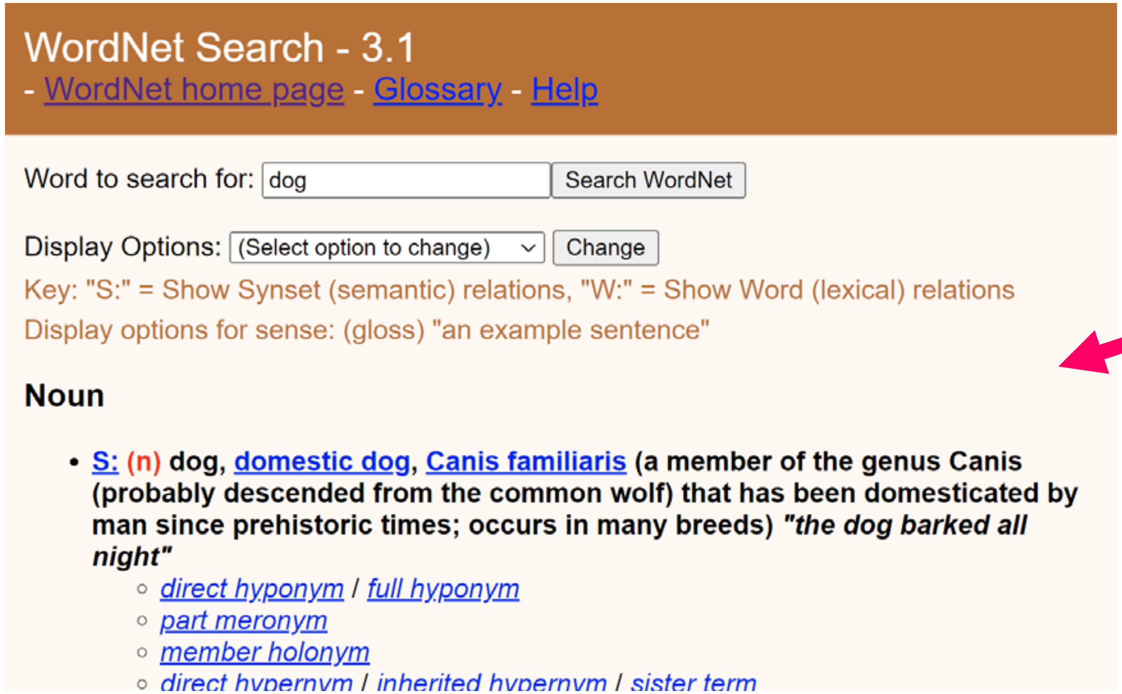
We are using *NLTK* to compare three stemmers and/or lemmatizers. After looking at them qualitatively, we are testing how fast they can lemmatize a large number of words.

WordNet Synsets

We are using *NLTK* to access *WordNet* data. More precisely, we are accessing the synsets for *fantastic* in order to find possible synonyms.

WordNet and Synsets

[WordNet](#) is a lexical database for English in which words “are grouped into sets of cognitive synonyms,” so-called synsets. These express distinct concepts.



[14]

```python
import nltk
from nltk.corpus import wordnet

nltk.download('wordnet')
nltk.download('omw-1.4')
```

[21]

```python
wordnet.synset('dog.n.01').lemmas()
```

```
[Lemma('dog.n.01.dog'),
 Lemma('dog.n.01.domestic_dog'),
 Lemma('dog.n.01.Canis_familiaris')]
```

Exercise 14 – spaCy Tagging

Use spaCy to automatically tag/annotate a text file of your choice for PoS, NERs, and Universal Dependencies.

Here we are using *spaCy* and a small **language model** (*en_core_web_sm*) to tag a given text. After creating a *spaCy* document – using the model – we can loop over the tokens (and entities) to access their tags.

Using *displaCy*, *spaCy's* visualizer library, we can also generate graphs for the dependencies.

Exercise 15 – Parsing XML

Write a function that allows you to extract all elements with a given attribute from an XML file.
For example, the function should be able to produce the following output for the file data/xml/bnc_style.xml and the attribute pos="VERB": have, bought

RegEx-Based Approach

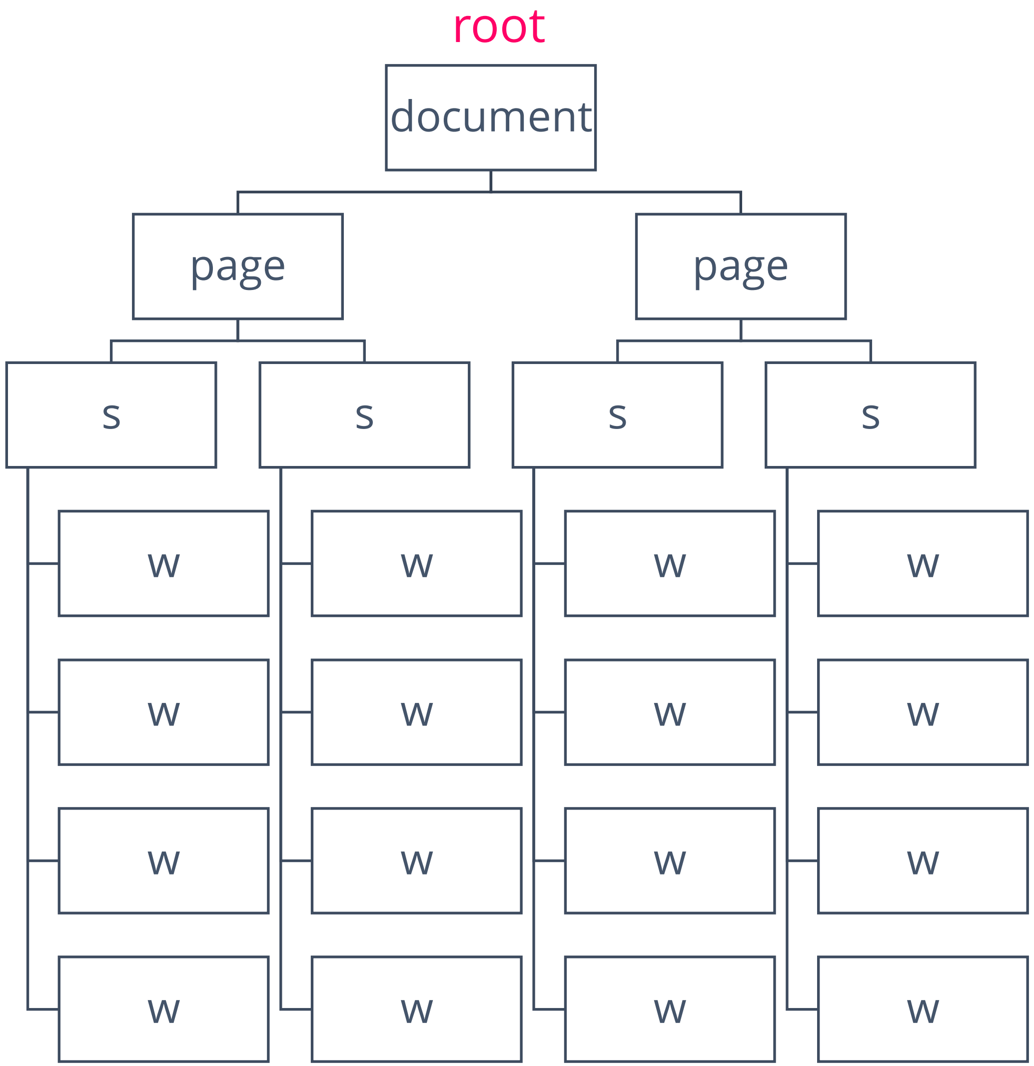
We are using a rather simple regular expression to find XML elements that contain the desired attribute and value. This solution, while being very straightforward, is not very robust if, for example, the underlying XML changes slightly.

Parsing Approach (LXML)

Here we are using an XML library (*LXML*) to parse and then navigate the XML structure/tree. We can also use **XPath** to navigate the document comfortably.

XML as a Tree

```xml
<document>
    <page pg_nr="1">
        <s>
            <w pos="determiner">The</w>
            <w pos="noun">flower</w>
            <w pos="verb">was</w>
            <w pos="adjective">red.</w>
        </s>
        <s>
            <w pos="pronoun">It</w>
            <w pos="verb">smelled</w>
            <w pos="preposition">of</w>
            <w pos="noun">summer.</w>
        </s>
    </page>
    <page pg_nr="2">
        <s>
            <w pos="pronoun">She</w>
            <w pos="verb">enjoyed</w>
            <w pos="det">the</w>
            <w pos="noun">trip.</w>
        </s>
        <s>
            <w pos="pronoun">They</w>
            <w pos="verb">took</w>
            <w pos="det">a</w>
            <w pos="noun">bus.</w>
        </s>
    </page>
</document>
```



XML XPath

XPath is a query language used for selecting nodes in XML documents.

/page[@pg_nr='2']/s[2]/w[1]

/ Select from the root node
@ Select attribute

```xml
<document>
    <page pg_nr="1">
        <s>
            <w pos="determiner">The</w>
            <w pos="noun">flower</w>
            <w pos="verb">was</w>
            <w pos="adjective">red.</w>
        </s>
        <s>
            <w pos="pronoun">It</w>
            <w pos="verb">smelled</w>
            <w pos="preposition">of</w>
            <w pos="noun">summer.</w>
        </s>
    </page>
    <page pg_nr="2">
        <s>
            <w pos="pronoun">She</w>
            <w pos="verb">enjoyed</w>
            <w pos="det">the</w>
            <w pos="noun">trip.</w>
        </s>
        <s>
            <w pos="pronoun">They</w>
            <w pos="verb">took</w>
            <w pos="det">a</w>
            <w pos="noun">bus.</w>
        </s>
    </page>
</document>
```

/page[@pg_nr='2']

s[2]
w[1]

/page[@pg_nr='2‘]/']/s[2]

/page[@pg_nr='2‘]/']/s[2]/w[1]

Exercise 16 – Web Scraping

Write a function that scrapes the text from a given website. The function should take a URL as its input and return the text present on the given website (e.g., Wikipedia). If you want to challenge yourself even further, try to remove boilerplate (everything that is not the main text) from the result.

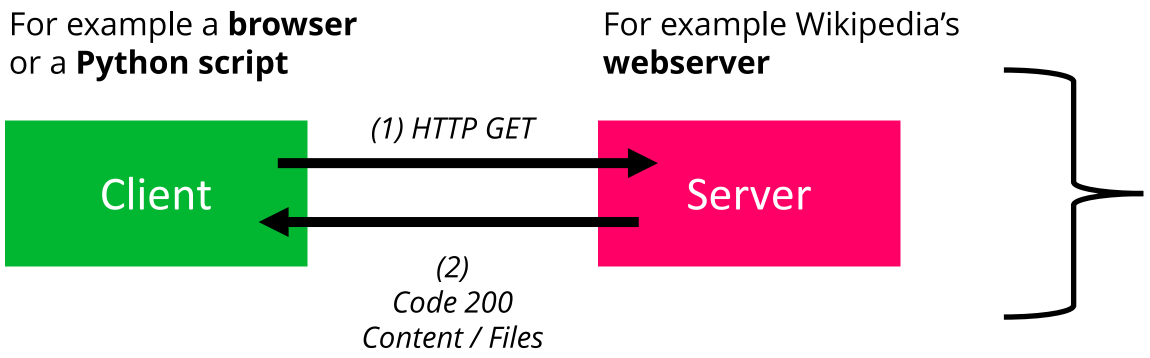
HTML and BeautifulSoup Parsing

The first function will use *requests* to get the HTML for the article. We are then using *BeautifulSoup* to parse the HTML and only return the content of the *bodyContent* div of the Wikipedia article.

HTML and jusText

The second function also retrieves the HTML using *requests*. Instead of parsing the site ourselves, we are using *jusText* to identify non-boilerplate paragraphs which we then combine into one string.

HTTP GET and *Requests*



In order to retrieve a website, a **client** (e.g., a browser) sends an HTTP GET request to a **webserver**. The server then responds with the website by sending **HTML** and possibly other content/files such as images.

The server also sends a status code indicating whether the requests worked.

We can use the `Requests` library to send HTTP requests:

r = requests.get('https://en.wikipedia.org/wiki/Linguistics')

r.status_code → *200*
r.text → *text/HTML*
r.content → *Binary/non-text content*

Code	**Meaning**
200	OK
403	Forbidden
404	Not Found
5XX	Server Error
…	…

Exercise 17 – Putting Everything Together (Keyword Analysis)

The ultimate goal of this exercise is to write a system which can perform basic (comparative) keyword analysis on two corpora.

1. Use your web scraper to build a small Wikipedia corpus of about three to five articles. Ideally, they will belong to a similar topic, e.g., politics.
2. Find a suitable reference corpus to compare your Wikipedia corpus with.
3. Use your new skills to generate frequency lists for both corpora.
4. Implement any keyness statistic (e.g., simple maths or log-likelihood) and determine the keywords.

Hint: To download the COCA sampler, run the following command in a Google Colab cell:

```
!cd python-programming-for-linguists/2020/data && sh download_coca.sh
```

This will download and extract the *COCA sampler* to your `/data/corpora/coca` folder.

Shared Vocabulary

For our comparative analysis, we need to be able to compare the frequencies for all types in both corpora. A reliable way of doing this is to use a shared vocabulary; alternatively, one could assign 0 if the type is not found in the other corpus during the comparison.

Target

Type_A, Type_B

Reference

Type_A, Type_C

Shared Vocabulary

Type_A
Type_B
Type_C

Frequency Table

Vocab	Target	Reference
Type_A	4	3
Type_B	6	**0**
Type_C	**0**	7
...		

Lambda Functions

aka. Anonymous Functions

Lambda functions are very powerful but quite hard to comprehend. On the surface level, and we will not go any deeper, these are functions without a name. They are used when we only require a function for a short period of time.

```
x = lambda a: a + 10
x(5)    → 15
```

We're only going to use them once!

They are, for example, useful when `.apply`-ing functions to a DataFrame.

Simple Maths Parameter

The *k* parameter works almost as a filter. The lower we set the parameter, the more low-frequency items we will identify as keywords.

$$SMP = \frac{RF_T + k}{RF_R + k} \qquad k = 100 \qquad Relative\ Frequency$$

See Kilgarriff, Adam. (2009). Simple Maths for Keywords. In Proceedings of the Corpus Linguistics Conference, Liverpool, July.