COMP3074 Human-AI Interaction
Lab 0: Preparatory work and basic string processing

Jérémie Clos

version 4 - 2023

Contents

1 Introduction

This is an optional lab that you can do at home, as a Python refresher and/or introduction. The goal of this lab is to introduce you to basic processing and parsing of text in Python. This lab script is made up of multiple sections, with code, explanations, and problems for you to experiment with as you read. I encourage you to do those tasks and play around with the code in order to become familiar with Python and its features.

WARNING: copy and pasting from a PDF file can sometimes mess up code alignment and/or quotation marks around Strings. If you copy from the PDF and it does not work, make sure to check this.

2 Prelude - getting acquainted with Python

Because this is not a programming course, we expect that you have some programming background. Experience with Python is not strictly necessary, because we will be using relatively basic features of the language, the most complex of which would be something like list comprehensions. The first lab is intentionally simple, so that students with little Python experience can take the week to become familiar with the language.

2.1 Installing the Computer Science Windows Virtual Desktop (WVD) client

If you are using your own machine, follow the steps described on the University's website[1], for whichever operating system your machine is running.

Once the setup is complete, you can launch the WVD "Computer Science Desktop" from the list of workspaces.

Launch the cmd prompt by typing "cmd" in the Search box located in the lower left corner of the WVD. To launch the Python interpreter from the prompt, simply type

```
python
```

2.2 Installing Python (optional)

Note: you can skip this step if you are using the CS WVD as described in Section 2.1.

To install Python, I would recommend installing the Anaconda[2] distribution. The reason is that it contains a set of packages, which can be very useful.

2.3 Installing an IDE (optional)

Note: you can skip this step if you are using the CS WVD as described in Section 2.1.

Three schools of thought:

- I don't care about IDEs. In this case, feel free to use your text editor of choice. Notepad++[3] is always a good choice, but there are others depending on your operating system.
- I like lightweight IDEs. In this case, VSCode[4] and Spyder[5] are both very suitable. VSCode is more modular and extensible. Spyder is more interactive.
- I cannot live without IDEs. In this case, PyCharm Community Edition[6] is probably your

[1] `https://www.nottingham.ac.uk/it-services/computers/virtual.aspx`
[2] `https://www.anaconda.com/products/individual`
[3] `https://notepad-plus-plus.org/downloads`
[4] `https://code.visualstudio.com`
[5] `https://www.spyder-ide.org`
[6] `https://www.jetbrains.com/pycharm/download`

best bet.

Choose whichever you prefer. It does not matter.

2.4 Running a Python script

```
python scriptname.py
```

It is fairly easy. There is not more to it, at least in the context of this module.

2.5 Installing packages (optional)

Note: you can skip this step if you are using the CS WVD as described in Section 2.1.

Installing packages is typically done using pip, the default package manager for the PyPI (Python Package Index) repository. For example, let us install BeautifulSoup, a library for parsing HTML pages. You can see from the official PyPI website of the package [7] of the package that its alias is beautifulsoup4, and therefore you can install it with the following command:

```
pip install beautifulsoup4
```

Once installed, you can import packages with the import command, and optionally alias them with the as command :

```python
import bs4 as bs
```

You can also import direct objects from the library with the from...import command, as follows:

```python
from bs4 import BeautifulSoup as bsoup
```

2.6 Learning Python

There are many ways to learn Python, depending on your current background and your familiarity with C-type programming languages. There would not be much interest in me copying a Python tutorial into this document, so I will provide useful links instead.

My personal recommendations are the following.

- hackingscience.org[8] provides a set of 50+ exercises that will get you up to speed on some basic programming in Python.
- The official Python tutorial[9] is excellent and touches on most, if not all, features. The essentials sections are 1, 2, 3, 4, 5, 7, 10.5, 10.7
- The official documentation[10] is extremely instructive and a good webpage to bookmark for future use.

3 Accessing text and NLTK resources

NLTK comes with several integrated datasets[11].

```python
import nltk
nltk.download('gutenberg')
print(nltk.corpus.gutenberg.fileids())
```

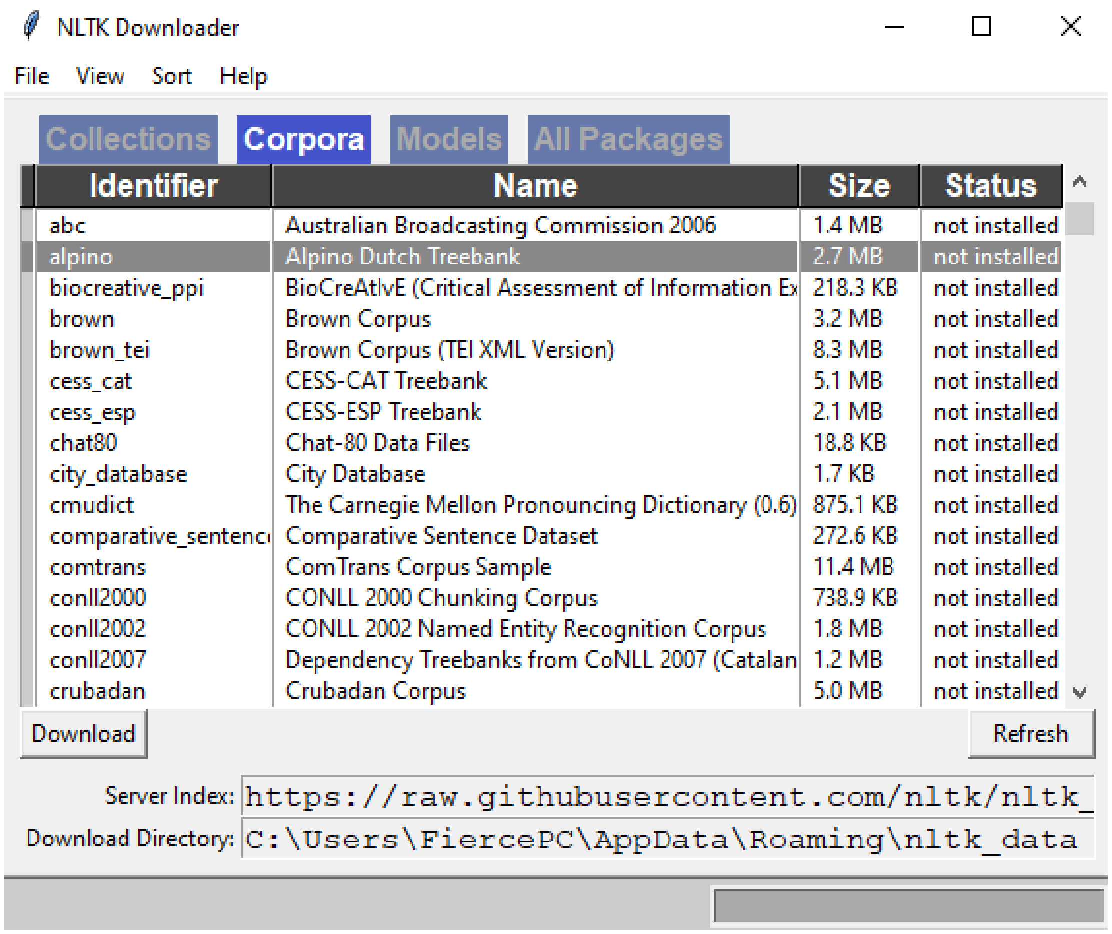
[7] https://pypi.org/project/beautifulsoup4

[8] https://www.hackinscience.org

[9] https://docs.python.org/3/tutorial

[10] https://docs.python.org/3

[11] You can see the list here: http://www.nltk.org/nltk_data.



Identifier	Name	Size	Status
abc	Australian Broadcasting Commission 2006	1.4 MB	not installed
alpino	Alpino Dutch Treebank	2.7 MB	not installed
biocreative_ppi	BioCreAtIvE (Critical Assessment of Information Ex	218.3 KB	not installed
brown	Brown Corpus	3.2 MB	not installed
brown_tei	Brown Corpus (TEI XML Version)	8.3 MB	not installed
cess_cat	CESS-CAT Treebank	5.1 MB	not installed
cess_esp	CESS-ESP Treebank	2.1 MB	not installed
chat80	Chat-80 Data Files	18.8 KB	not installed
city_database	City Database	1.7 KB	not installed
cmudict	The Carnegie Mellon Pronouncing Dictionary (0.6)	875.1 KB	not installed
comparative_sentenc	Comparative Sentence Dataset	272.6 KB	not installed
comtrans	ComTrans Corpus Sample	11.4 MB	not installed
conll2000	CONLL 2000 Chunking Corpus	738.9 KB	not installed
conll2002	CONLL 2002 Named Entity Recognition Corpus	1.8 MB	not installed
conll2007	Dependency Treebanks from CoNLL 2007 (Catalan	1.2 MB	not installed
crubadan	Crubadan Corpus	5.0 MB	not installed

Server Index: https://raw.githubusercontent.com/nltk/nltk_
Download Directory: C:\Users\FiercePC\AppData\Roaming\nltk_data

Figure 1: The package and resource management system of NLTK

But more than that, NLTK comes with an absurd amount of resources and an even more absurd way to download them: a pre-packaged GUI. Run the following program and find out:

```python
import nltk
nltk.download_gui() #  use nltk.download_shell() if you prefer a text interface
```

The graphical user interface shown in Figure 1 should appear.

This lets you pick and choose which corpus to download as well as pre-existing machine learning models. Altogether, they can be quite heavy, so it was deemed necessary to make the library modular and let you pick and choose what you need.

If we are not satisfied with our current corpora, we can use the URLLib library to easily download new documents:

```python
from urllib import request
url = "http://example.org"
raw = request.urlopen(url).read().decode('utf8')
print(raw)  # html code of the page
```

Feel free to try it out with your favourite websites and marvel in horror at the ridiculous amount of JavaScript code they are riddled with. This is partly why your phone battery is dying if you spend too much time using it to browse the Web. Be very careful if you are putting the request in a loop and accessing the same website many times; most good providers have bot detection tools that will ban your IP for some time to protect themselves. And if you are accessing from the university, you might be banning other people with you!

4 Basic parsing

4.1 Accessing data

We need something to parse, and putting example strings can only go so far. We can acquire data either from existing text files or by downloading stuff from the Web.

4.1.1 Reading online files

Let us download an e-book from the Gutenberg online library. I will work with Mary Shelley's Frankenstein, but feel free to take your pick from the Gutenberg library[12] as they are all provided in text format (.txt).

```python
import nltk
from urllib import request
url = "http://www.gutenberg.org/files/84/84-0.txt"  # Frankenstein, by Mary
    Wollstonecraft (Godwin) Shelley
content = request.urlopen(url).read().decode('utf8', errors='ignore')
```

We set the encoding to utf-8 because it's a common encoding that works most of the time, but Python has built-in functions to deal with many different encodings, which you can read about in the official documentation. It is important that we set *errors* to 'ignore' so that the entire process does not stop if Python encounters one character it cannot parse. On the other hand, it might mean that we end up with some blank characters in our file, which is a risk we will almost always have to take when dealing with data.

4.1.2 Reading local files

The `open` command lets us open a file in different modes (writing, reading, writing + reading) and with different encodings. The scripts below assume that there is an existing document.txt file in the same folder as your Python script that we are opening in "read" mode (the `'r'`) and with the utf-8 encoding.

```python
f = open('document.txt', 'r', encoding='utf-8')
content = f.read()
print(len(content))  # print the length of the text
```

However, this technically leaves us open to problems: you would need to remember to `f.close()` to free the handle on the file. A better and more Pythonic way of doing things is the `with...as` statement.

```python
with open('document.txt', 'r', encoding='utf-8') as f:
    content = f.read()
    print(len(content))  # print the length of the text
```

This ensures that Python releases its handle on the file once the processing block is done and results in cleaner, better code. What about reading line-by-line? It is not much different:

```python
with open('document.txt', 'r', encoding='utf-8') as f:
    for line in f:
        print(line)  # print the current line
```

[12]Accessible at `http://www.gutenberg.org/ebooks/bookshelf`

4.1.3 Working with a database

To make an application with persistent knowledge from session to session, you might (but do not have to) end up having to store and retrieve structured data, in which case a database would be helpful. Fortunately, Python integrates native access to SQLite databases, which do not require the installation of any extraneous software.

```python
import sqlite3
connection = sqlite3.connect('database.db')
```

These two lines would import the sqlite3 library from the Python standard library, and connect to a file named database.db, and create it if it does not already exist. This connection is stored in an object named "connection". You can then use that object to perform operations on the database.

```python
1  cursor = connection.cursor()
2
3  cursor.execute('''CREATE TABLE Corpus
4                 (documentId text, documentContent text, documentTopic text)''')
5
6  cursor.execute("INSERT INTO Corpus VALUES ('doc1','This is an example document
       for the corpus', 'example')")
7
8  connection.commit()
9
10 connection.close()
```

Line 1 opens a cursor from the connection. Line 3 executes a table creation instruction to create a table in our database. Line 6 inserts one row of data into that table. Line 8 commits the transaction described before, and line 10 closes the connection. The file containing the data is persistent and can then be reused in the latter code, but the CREATE TABLE instruction erases the content of current tables, so be wary of that. This is not a course on databases, but a simple googling of SQL and SQLite/Python should yield a large number of tutorials and help pages on how to use it effectively in Python.

If you want to look at the content of your DB file, you can either use SELECT statements from code, or use specialised software such as SQLite browser[13].

5 The preprocessing pipeline

5.1 Preprocessing

Now that we know how to get text, let us do something with it.

5.1.1 Dealing with HTML

If you downloaded some HTML files, you can use BeautifulSoup to remove the tags and extract the text only. Assuming you have text containing html tags in a character string htmlString, the code would look like this:

```python
from bs4 import BeautifulSoup
content = BeautifulSoup(htmlString, 'html.parser').get_text()
```

5.1.2 Tokenisation, stopword removal, and casing normalisation

This brings about some notions discussed in Lecture 2, on the NLP pipeline. Now we are starting to actually use NLTK! NLTK has many handy lists of words, one of which is a stop-word list.

[13]https://sqlitebrowser.org

You can download it with the nltk.download('stopwords') command and use it by importing stopwords from nltk.corpus. Once imported, you can call the collection with stopwords.word(). In order to be able to remove stopwords, we need to do some basic tokenisation of the text.

```python
import nltk
nltk.download('stopwords')

from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize

text = "Artificial intelligence is cool but I am not too keen on Skynet."
text_tokens = word_tokenize(text)
tokens_without_sw = [word.lower() for word in text_tokens if not word in
    stopwords.words()]
print(tokens_without_sw)
filtered_sentence = (" ").join(tokens_without_sw)
print(filtered_sentence)
```

We can also use NLTK to rebuild a sentence in a way that will be easy to parse by the library, using the following function:

```python
text = nltk.Text(tokens_without_sw)
print(text.count('cool'))  # count the number of occurrences of the word 'cool'
```

It will build an object of type Text, which allows complex exploration operations such as counting, collocation discovery, etc. You can see the additional features of the Text object in the official NLTK documentation[14]. Note that we use word.lower() when building our list of tokens in order to take care of casing: it is important that tokens which are meant to be the same, look the same. For example, "artificial" and "Artificial" should both point to the same token, and as such consistent casing needs to be applied. What about "is" and "am"? It stands to reason that they should also point towards the same token, which is where **stemming** or **lemmatisation** come into play.

5.1.3 Stemming versus lemmatisation

"One cat jumps, two cats jump"

Why is it cat in the first part, but cats in the second part? Why is it jumps in the first part, but jump in the second? Words are modified based on usage, and those little variations of words are called **inflections**. Inflections on verbs are called conjugation, and inflections on nouns are called declensions. Some languages have a lot of declensions, such as Russian, while some only have a few like English. Some, like French, kept some form of declensions from their Latin roots but simplified them so much that we don't even call them declensions anymore. Finally, other languages such as standard Chinese have almost no declension at all and all tokens have a single grammatical form.

Declensions give us extra information about the meaning of the word through its grammatical role, in exchange for a bit of extra complexity. However useful those inflections can be for us humans, they are terrible for machines, because a machine has no way to know that cats and cat refer to the same animal or that jumps and jump are merely two forms of the same verb. This greatly increases the number of potential tokens the machine might encounter, since we need to plan for each possible inflection, which is a problem.

Two types of method in order to reduce the number of different tokens are **stemming** and **lemmatisation**. They operate differently but achieve similar things: removing inflections from words.

Stemming The goal of stemming is to recover the stem of a word, which is the common part of all inflections of that word. Because languages have their own idiosyncrasies, stemming

[14] `https://www.nltk.org/api/nltk.html#nltk.text.Text`

algorithms are not portable from one to another. For English, the best known stemmers are the Porter, Snowball, and Lancaster stemmers. You can experiment with them in `https://text-processing.com/demo/stem/` and they are all accessible in NLTK[15]. All stemmers are not 100% accurate and use a set of heuristics to guess the most likely stem of a word, and as such have different levels of aggressiveness when cutting down inflections.

```
from nltk.stem.porter import PorterStemmer
from nltk.stem.snowball import SnowballStemmer
from nltk.tokenize import word_tokenize

p_stemmer = PorterStemmer()
sb_stemmer = SnowballStemmer('english')
sentence = "This is a test sentence, and I am hoping it doesn't get chopped up
    too much."
print(sentence)
for token in word_tokenize(sentence):
    print(p_stemmer.stem(token))
    print(sb_stemmer.stem(token))
    print("---")
```

Lemmatisation The main difference between lemmatisation and stemming is that lemmatisation is dictionary-based: it tries to uncover the morphological root (dictionary form) of a word instead of its stem, which means it is slower but gives more meaningful results. As such, it also requires downloading some extra resources by running the download command at least once.

```
from nltk.stem.wordnet import WordNetLemmatizer
from nltk.tokenize import word_tokenize

nltk.download('punkt')
nltk.download('wordnet')  # These two lines only need to be run once
lemmatiser = WordNetLemmatizer()
sentence = "I am writing a few words, and I am hoping they don't get chopped up
    too much."
print(sentence)
for token in word_tokenize(sentence):
    print(lemmatiser.lemmatize(token))
```

You will notice that the lemmatiser does very poorly on verbs. The reason for that is that it needs to be given additional information as a second argument of the lemmatize(token) method, the part-of-speech tag (noun, verb, adjective, etc.), in order to lemmatise correctly. Its default behaviour is to assume everything is a noun. Luckily, NLTK also gives us a handy way of determining the part of speech of a token! But that requires downloading an additional resource, the `average_perceptron_tagger`.

```
from nltk.stem.wordnet import WordNetLemmatizer
from nltk.tokenize import word_tokenize
nltk.download('wordnet')
nltk.download('averaged_perceptron_tagger')
nltk.download('universal_tagset')

lemmatiser = WordNetLemmatizer()
sentence = "This is a test sentence, and I am hoping it doesn't get chopped up
    too much."

post = nltk.pos_tag(word_tokenize(sentence), tagset='universal')
print(post)
```

This code will yield the following result:

[15]`https://www.nltk.org/howto/stem.html`

Tag	Meaning	Example
ADJ	Adjective	good, bad, hot, ...
ADP	Adposition	into, on, of, ...
ADV	Adverb	very, really, not, ...
CONJ	Conjunction	and, or, but, ...
DET	Article	the, a, every, ...
NOUN	Noun	apple, dog, car, ...
NUM	Numeral	one, 21, fifty five, ...
PRT	Particle	at, on, out, ...
PRON	Pronoun	she, he, I, ...
VERB	Verb	drink, sleep, is
.	Punctuation	. , !
X	Other	boom, ersatz, gr8, ...

Table 1: A set of common Part-of-Speech tags

```
[('This', 'DET'), ('is', 'VERB'), ('a', 'DET'), ('test', 'NOUN'), ('sentence', '
    NOUN'), (',', '.'), ('and', 'CONJ'), ('I', 'PRON'), ('am', 'VERB'), ('hoping'
    , 'VERB'), ('it', 'PRON'), ('does', 'VERB'), ("n't", 'ADV'), ('get', 'VERB'),
     ('chopped', 'VERB'), ('up', 'PRT'), ('too', 'ADV'), ('much', 'ADV'), ('.', '
    .')]
```

All tokens have been tagged with their parts of speech. The following figure explains what each tag means:

However, all is not done, because the POS tags that the lemmatiser wants are different from the POS tags that the POS tagger provides (why would anything be easy after all?). Now all we need to do is map those POS tags to POS tags that the WordNetLemmatizer takes as input (a for adjectives, n for nouns, v for verbs, and r for adverbs) and then let the default behaviour happen for the other tags. An easy way to do so is by doing that mapping in a Python dictionary, as shown in the following code:

```python
from nltk.stem.wordnet import WordNetLemmatizer
from nltk.tokenize import word_tokenize
nltk.download('wordnet')
nltk.download('averaged_perceptron_tagger')
nltk.download('universal_tagset')
lemmatiser = WordNetLemmatizer()
sentence = "I am writing a few words, and I am hoping they don't get chopped up
    too much."

posmap = {
    'ADJ': 'a',
    'ADV': 'r',
    'NOUN': 'n',
    'VERB': 'v'
}

post = nltk.pos_tag(word_tokenize(sentence), tagset='universal')
print(post)
for token in post:
    word = token[0]
    tag = token[1]
    if tag in posmap.keys():
        print(lemmatiser.lemmatize(word, posmap[tag]))
    else:
        print(lemmatiser.lemmatize(word))
    print("---")
```

The resulting lemmatisation is still not perfect, but it can recognise verbs and it does not cut down nouns as much as the average stemmer. However, you will notice that it takes a bit more time.

5.1.4 String processing and pattern matching

Like all modern languages, Python has an extensive array of pattern matching tools that you can use when looking for specific information in text. For example, if we wanted to count the number of gerunds in a document, a naïve but effective way of doing so would be as follows:

```python
from nltk.tokenize import word_tokenize
sentence = "This is a test sentence, and I am hoping it doesn't get chopped up
    too much."
tokens = word_tokenize(sentence)
count = 0
for token in tokens:
    if token.endswith("ing"):
        count += 1
print(count)
```

Python also gives you a large collection of methods and constants to deal with character strings, which you can read about in the official documentation[16]. If you need to push the analysis further, it also gives you access to powerful regular expressions[17] for more advanced pattern matching.

Table 2 (reproduced from Chapter 3 of the NLTK book) gives a summary of the most useful string processing methods.

Method	Functionality
`s.find(t)`	returns the index of the **first** instance of string `t` inside `s`, or -1 if not found
`s.rfind(t)`	returns the index of the **last** instance of string `t` inside `s`, or -1 if not found
`s.index(t)`	returns the index of the **first** instance of string `t` inside `s`, or `ValueError` if not found
`s.rindex(t)`	returns the index of the **last** instance of string `t` inside `s`, or `ValueError` if not found
`s.join(text)`	combines the words of the `text` into a string using `s` as the joining character
`s.split(t)`	splits `s` into a list wherever a `t` is found
`s.splitlines()`	splits `s` into a list of strings, one per line
`s.lower()`	returns a lowercased version of the string `s`
`s.upper()`	returns an uppercased version of the string `s`
`s.title()`	returns a titlecased version of the string `s`
`s.strip()`	returns a copy of the string `s` without the leading or trailing whitespace
`s.replace(t, u)`	replaces instances of `t` with `u`

Table 2: Useful string processing methods, reproduced from the NLTK book

[16] https://docs.python.org/3/library/string.html
[17] https://docs.python.org/3/library/re.html

6 The I/O loop

The core of an interactive system is to react to user input. In this exercise, you will create a basic chatbot that interacts with the user in a continuous loop until the user types ”exit” to end the session. This will be done using a while loop in Python, and you will utilise some functions from the Natural Language Toolkit (NLTK) to process the user's input.

Start by importing necessary libraries:

```python
import nltk
```

Next, define a function to process user input. For simplicity, we will create a function that tokenises the user's input into words:

```python
def process(user_input):
    tokens = nltk.word_tokenize(user_input)
    return tokens
```

Now, create your main IO loop using a while loop. In this loop, ask the user for input, process the input using your process function, and output the result back to the user. Exit the loop when the user types ”exit”:

```python
def main():
    while True:
        user_input = input("You: ")
        if user_input.lower() == 'exit':
            print("Goodbye!")
            break
        processed_input = process(user_input)
        print(f"Bot: You said {processed_input}")

if __name__ == "__main__":
    main()
```

Now, when you run your script, you should see a prompt asking for your input. Type some text and watch as your chatbot tokenises your input into words using NLTK. When you are done, type ”exit” to end the session.

7 Exercises

Exercises are basic tasks that can help you better understand the content. You are not expected to finish everything in the lab.

1. Write a Python function that changes the American spelling of words like *color*, *favor*, *behavior*, etc. to its British spelling of (e.g., *col**our***, *fav**our***, *behavi**our***) using only slice and concatenation operations.
2. The slice notation can be used to remove morphological endings in words. For example, `'programming'[:-4]` removes the last 4 characters of `programming`, leaving `program`. Use the slice notation to remove the affixes from these words (we've inserted a hyphen to indicate the affix boundary, but omit this from your strings): `dish-es`, `run-ning`, `nation-ality`, `un-do`, `pre-heat`.
3. We can specify a "step" size for the slice. The following returns every second character within the slice: `monty[6:11:2]`. It also works in the reverse direction: `monty[10:5:-2]`. Try these for yourself, then experiment with different step values. What happens if you ask the interpreter to evaluate `monty[::-1]`?
4. Define a string `raw` containing a sentence of your own choosing. Now, split `raw` on some character other than space.
5. Create a variable `words` containing a list of words. Experiment with `words.sort()` and `sorted(words)`. What is the difference?
6. Write code to access a webpage and extract some text from it. Print it to the console. For example, access a weather site and extract the forecast top temperature for your town or city today.
7. Look at the stopword list (you can print `stopwords.words('English')`): what problems do you think having a fixed stopword list could cause? How would you overcome them? Can you build a better stopword list?
8. Find a set of documents and try to download/parse them. What is the proportion of nouns vs. verbs vs. adjectives in an average novel?
9. [⋆] You can use the `vocab()` method to access the vocabulary of a `Text` object in the form of a dictionary, where the key is the token and the value is its frequency in a given text. Use this to build a function `topN(S,N)` that, given as input a string of characters S and a number N, returns the top N tokens with decreasing frequency.
10. [⋆] Expand on your IO loop to add a feature that responds to the user by telling them the number of different words they used. Make sure to count after lemmatisation in order to make sure that all inflections are counted in the same word.
11. [⋆] Expand on your IO loop to add a feature which asks the user their name and saves it in a file (or alternatively a database). Upon reloading the program, it should use this file and confirm the user's name (e.g., "am I still talking to username?").

8 Tasks

Tasks are more time-consuming than exercises and allow you to go much deeper into content mastery. We do not expect you to finish all of them in the lab, but they can be good toy problems to practise with.

1. [$\star$] Write an application which downloads books from the Gutenberg repository and stores them in a file or in an SQLite database.
2. [$\star\star$] Add a function to parse all books in the file/database and print their name and the number of words they contain.
3. [$\star\star$] Add a function to produce the 100 most frequent and 100 least frequent words in all the books of your file/database and their frequency, by decreasing order of frequency.
4. [$\star\star$] Building on one of the previous exercises, write a function that uses a conditional statement and the slice notation to remove the affixes of plural words and verbs ending in `-ing`. Try to take care of multiple cases, e.g. plural in ”s”, plural in ”es”, verbs ending in -ing with double consonant (e.g., running -¿ run, winning -¿ win), etc.
5. [$\star\star$] Compute the document frequency (DF) of those 100 words, where DF is defined as $DF(n, N) = \frac{n}{N}$ where n is the number of documents containing the word at least once, and N is the total number of documents in your corpus. What words have the highest document frequency?
6. [$\star\ \star\ \star$] Using what you have learnt so far, write a function `plagiarise(text)` which uses a dictionary of synonymous expressions and a set of transformation rules in order to paraphrase arbitrary text.
7. [$\star\ \star\ \star$] How would you detect such deviously manipulated plagiarism?