7. Text processing (pt. 1) LING 471

Assignment 1

- Grades released
- Please remember to commit, push, and then include a link in your Canvas submission
- Email me if you have any questions

Assignment 2

- Due in 1 week
  - Apr 28, 11:59 PM
- Should more or less be able to complete after today's lecture

Learning outcomes

- Explain why **text preprocessing** is necessary in NLP
- Reuse code via **Python modules and packages**
- Use **regular expressions** for pattern matching Perform basic tokenization
- Understand how modern NLP systems **tokenize** text (e.g., subword tokenization used in LLMs)

Code reuse

Code reuse

- We have been manually writing lots of new code for all of our activities
  - Without intentional practices, new code introduces new bugs
- Better general goal: write as little new code as you can
  - Alternatively stated, **reuse** as much existing code as you can

Ways to reuse code

- Use Python's built-in functions and modules
- Previously written code (new installable packages and your own historical code)
- Community resources
  - E.g., StackOverflow, Reddit, maybe GitHub
- LLMs (ChatGPT, Claude, Copilot, etc.)
  - Important caution: LLM outputs may be incorrect or insecure
  - Always test and understand code before using it

More on code reuse

- Many times, the code you look up is the only obvious way to do something and fine to copy and paste
  - E.g., finding out sorting is done with `list.sort()` is like finding out addition is written as 2+2
- Sometimes will also point out a standard module built into Python you can use
  - Unless a task is explicitly assigned as part of your homework, don't write new code until you are convinced there is no built-in module for it

Modules

Modules

- Modules are just other Python files
- You can **import** modules in other Python files
- To import a module
  - It must exist on your computer
  - You need to be able to tell Python where to find the module
    - This is sometimes harder than it sounds!

The basic `import` statement

- Modules are loaded with the import statement
- Easiest construction:
  - `import MODULE`
  - import – keyword for importing
  - MODULE – the module you want to import
  - This style of import requires you to use the module name when using what it provides

```
import math # imports math module
math.sqrt(4) # takes sq. root of 4

import sys # imports sys module
sys.argv # accesses program
parameters
```

More `import` statements: `from`

- Selecting specific parts of a module to import
  - `from MODULE import SUBPART`
  - `from` – keyword
  - `MODULE` – module to import
  - `import` – keyword for importing
  - `SUBPART` – part of module to import

```
from math import sqrt
sqrt(4) # take sq. root of 4
```

More `import` statements: `as`

- Can rename imported modules
  - `import MODULE as NAME`
  - import – import keyword
  - MODULE – module to import
  - as – keyword for renaming imported module
  - NAME – what to rename MODULE to
- Can be combined with **`from`** to import subpart as well

```
import math as m
m.sqrt(4) # sq. root of 4

from math import sqrt as sq_root
sq_root(4) # sq. root of 4
```

Programming activities

- Import the math module and try using the following functions on whatever numbers you want: sin, cos, sqrt, pow, log, floor
  - Try any other functions or variables from the documentation that you think are interesting
- Make a list with the numbers 1 up to and including 10 and assign it to a variable
  - Import the statistics module and try using the following functions: mean, median, stdev
  - Call the `sum` function over your list and then divide the result by the length of the list. Does it match the output of `statistics.mean`?
  - Try any other functions or variables from the documentation that you think are interesting

Regular expressions

Text processing and pattern matching

- A lot of the text processing we do in linguistics requires some kind of pattern matching
  - E.g., remove all punctuation, make all letters lower case, split at whitespace, find all singular and plural forms of this noun, remove all instances of this word, etc.
- There are a wide variety of tools we can use to accomplish this goal
- Some of them are more **performant** or **widely applicable** than others

Simple finding and replacing

- We have already seen simple pattern matching in strings by using the **`in`** operator
- Can count all or some occurrences of a pattern with the **`count`** method
- Can replace all or some occurrences of a pattern with the **`replace`** method
- These methods are preferable for **simple** tasks

```
'ello' in 'hello' # True
'halo' in 'hello' # False

'hello'.count('l') # 2
'hello'.count('a') # 0

'hello'.replace('el', 'a') # 'halo'
'hello'.replace('eel', 'a') # 'hello'
'hello'.replace('l', '') # 'heo'
```

More general finding and replacing: Regular expressions

- More complicated finding and replacing can be accomplished using what are known as **regular expressions**
- Basically, they are a special language used to match patterns in text
- Not all patterns can be matched with regular expressions
  - A full Python program cannot be parsed with them, for example
- But **a lot** of patterns can be

- **Usage note**: in formal language theory, "regular expression" has a more restricted implementation than in most modern programming languages
  - Some individuals distinguish between a "regular expression" (formal) and a "**regex**" (modern)
  - Other individuals (like me, usually) use these terms as exact synonyms

Regex standards

- Regex conventions are fairly standard across programming languages
  - Not so much in applications (compare formats in Microsoft Office, Vim, etc.)
- Usually hail from either the POSIX standard or the Perl standard (or a related variant)
  - Python's implementation is close that of Perl
- Always need to check the regex documentation for the specific tool you are using, though!

The `re` module

- Python has a built-in regex module called **`re`**
  - The tutorial in the assigned reading used this module
- You will need to read the documentation to use it appropriately
- Must be imported to be used appropriately

```
import re
# find the plural nouns in the
sentence
re.findall(r'(\w+s|es)', 'the
puppers and kitties played in the
yard')

# ['puppers', 'kitties']

# this regex only reliably works for
this specific English sentence
```

Some common regex patterns

- Literal: family
  - Just matches "family"
- Disjunction (or): famil(y|ies)
- Character classes: [Tt]he, bec[oa]me, [A-z], [0-9]
  - Match one item in the specified set
- One or more times: [0-9]**+**
- Zero or more times: [0-9]*
- Zero or one time: colou**?**r
- Any character: **.** (the dot/period)
- Escape special characters and treat them literally: ***** (match "*" instead of match zero or more times)
- Special characters:
  - \w (characters that can make words)
  - \s (whitespace characters)

Raw strings

- You can tell Python to treat all the characters in a string literally by putting "r" immediately before the string
  - i.e., treat all backslashes literally
- This is called a **raw** string in Python
  - Commonly confused as a “regex string”
- Many regex searches work fine without doing this, but beware the backslash plague

```
r'C:\Users' # string is "C:\Users"
'C:\Users' # error: no \U escape in strings
'C:\\Users' # string is "C:\Users"
re.findall('\\w\\d+\\w', 'abc123def') # ['c123d']
re.findall(r'\w\d+\w', 'abc123def') # ['c123d']
re.findall('\w\d+\w', 'abc123def') # ['c123d']
```

Useful functions in `re` module

- re.sub – substitutes the matched patterns with a different string
- re.findall – finds all instances of the pattern in the string
- re.finditer – like findall but returns an interator instead of a list
- Generally, read the docs!

`re` — Regular expression operations

Source code: Lib/re.py

This module provides regular expression matching operations similar to those found in Perl.

Both patterns and strings to be searched can be Unicode strings (`str`) as well as 8-bit strings (`bytes`). However, Unicode strings and 8-bit strings cannot be mixed: that is, you cannot match a Unicode string with a byte pattern or vice-versa; similarly, when asking for a substitution, the replacement string must be of the same type as both the pattern and the search string.

Regular expressions use the backslash character (`'\'`) to indicate special forms or to allow special characters to be used without invoking their special meaning. This collides with Python's usage of the same character for the same purpose in string literals; for example, to match a literal backslash, one might have to write `'\\\\'` as the pattern string, because the regular expression must be `\\`, and each backslash must be expressed as `\\` inside a regular Python string literal. Also, please note that any invalid escape sequences in Python's usage of the backslash in string literals now generate a `DeprecationWarning` and in the future this will become a `SyntaxError`. This behaviour will happen even if it is a valid escape sequence for a regular expression.

Tips for working with regexes…

- Regexes will break your brain on a regular basis
  - Remember that **it is natural** to want to **throw** your computer **out the window** when writing regexes
- Try validating the pattern on a small string first
- Try using regex101.com to test the regexes
- Find a good cheat sheet
- Try adding more backslashes (seriously!)
- Be patient with yourself! :)

Programming activities

- Assign the string "The passcode is 8675309. Please remember it." to a variable
- Use `re.findall` or `re.sub` to do the following:
  - Extract just the passcode
  - Change the passcode to "555-5555"
  - Extract all words that start with "p"
  - Extract the first word of each sentence
  - Delete the word "Please" and capitalize "remember" afterward
    - This can be done with just a single replacement!
  - Change each "." to "!"

Tokenization

Tokenization

- Assume you are given a large string of text
  - This could be a program, or maybe just some writing from somewhere
- You want to perform some kind of analysis
  - E.g., find variable names and numbers, count words, tag part of speech, etc.
- To perform many analyses, you need to break up the large string
- That is, you need to **tokenize** the string
- Often the first step when doing NLP!

Tokens

- **Token** is a very general term
- Requires a precise definition of possible tokens
- Examples:
  - "words are strings of characters separated by spaces"
  - "variables are strings of letters, numbers, and _ that cannot start with a number"
  - "a (is a valid token"

Cheap tokenization: `str.split`

- For English and languages with similar orthographies (spelling systems), words are separated by spaces
- So, you can just separate a string into words with the **`split()`** function we talked about, right?
- What happens if you have punctuation, though?
- Should capital letters matter? (They do to Python!)

```
'the brown cow'.split()
# ['the', 'brown', 'cow']

'The brown cow eats.'.split()
# ['The', 'brown', 'cow', 'eats.']
```

Slightly better tokenization: regex

- We can instead tokenize with regex! Is this powerful enough for us?
- Will allow us to get rid of punctuation and capital letters
- Regexes can be inefficient to run
  - You can also end up with many steps like on the right…
- This also only really works for languages with spelling and punctuation like English, though
  - Even then, how do you handle English contractions and possessives?

```
text = "The brown cow ate.”

text = text.lower()
text = re.sub(r'[^\w\s]', '', text)
tokens = re.findall(r'\w+', text)

print(tokens)

# ['the', 'brown', 'cow', 'ate']
```

Statistical tokenization

- Instead of trying to formulate every possible pattern for tokenization ourselves, we can ask a computer to do it
  - Sort of…
- Writing your own code to do this can be challenging
  - Certainly, it's beyond what we've covered so far
- In general, you want to use someone else's code for this (i.e., a module!)

NLTK

- A popular package for processing language is the Natural Language Toolkit (NLTK)
- Can be installed via command line with ”python –m pip install nltk"
- Has a lot of useful functions you can call, some of which create statistical models you can train on language data
- We may use this NLTK time to time in this course, but there are other modules we will use heavily as well
- It comes with a free ebook that teaches you to use it

Tokenization with NLTK

- NLTK makes it easy to tokenize sentences
- Just use the functions in the `nltk.tokenize` submodule!
- Need to download a pre-trained model the first time you try it:

```
import nltk
nltk.download('punkt')
nltk.download('punkt_tab')
```

- Mac users may need to follow instructions here too

```
s ='Some people write money as $2.44, but
others write it as 2.44$.'

from nltk.tokenize import wordpunct_tokenize

wordpunct_tokenize(s)
# ['Some', 'people', 'write', 'money', 'as',
'$', '2', '.', '44', ',', 'but', 'others',
'write','it', 'as', '2', '.', '44', '$.']


from nltk.tokenize import word_tokenize

word_tokenize(s)
['Some', 'people', 'write', 'money', 'as',
'$', '2.44', ',', 'but', 'others', 'write',
'it', 'as','2.44', '$', '.']
```

Other tokenizers

- Different NLP systems break texts into tokens in different ways

Tokenizer	Typical use
split()	simple scripts
regex/NLTK	traditional NLP
spaCy	NLP pipeline
subword tokenizers (BPE, WordPiece)	modern NLP models (BERT, GPT, etc.)

- Important idea: tokens used by more recent NLP models are not always words

Programming activities

- Download NLTK and use it to tokenize a sentence/string
- Try to come up with a sentence that NLTK's function does not correctly tokenize
  - Try messing with capitalization, foreign characters/words, abbreviations, contractions, colloquial language, etc.
- If you want, try to compare with other tokenizers: spaCy, tiktoken, etc.