Text processing

- Text processing in Python involves manipulating and analyzing text data.
- Python offers a wide array of built-in functions, libraries, and tools to perform text processing tasks such as string manipulation, regular expressions, and natural language processing.
- Text Processing is an essential task in NLP as it helps to clean and transform raw data into a suitable format used for analysis or modeling.
- NLTK (Natural Language Toolkit) library is used for NLP.

Installing NLTK Pack

- pip install nltk

- “**Preferred Installer Program**” or PIP Installs Packages.
- It is a command-line utility that installs, reinstalls, or uninstalls PyPI packages with one simple command: pip
- pip is the **package installer** for Python.

TOKENIZATION

- In Python tokenization basically refers to splitting up a larger body of text into smaller lines, words or even creating words for a non-English language.
- The various tokenization functions in-built into the nltk module itself.

- Tokenization: Tokenization is the process of splitting text into individual words or tokens.

```
from nltk.tokenize import word_tokenize, sent_tokenize
# Word Tokenization
text = "Hello, world! How are you?"
words = word_tokenize(text)
print(words)  # Output: ['Hello', ',', 'world', '!', 'How',
  'are', 'you', '?']

# Sentence Tokenization
sentences = sent_tokenize(text)
print(sentences)  # Output: ['Hello, world!', 'How are
  you?']
```

Line Tokenization

- In the below example divide a given text into different lines by using the function sent_tokenize.

For Example:

```
import nltk
sentence_data = "The First sentence is about Python. The Second:
about Django. You can learn Python,Django and Data Ananlysis
here. " nltk_tokens = nltk.sent_tokenize(sentence_data)
print (nltk_tokens)
```

Word Tokenzitaion

- Tokenize the words using word_tokenize function available as part of nltk.

For Example:

```
import nltk
word_data = "It originated from the idea that there are readers who prefer learning new skills from the comforts of their drawing rooms"
nltk_tokens = nltk.word_tokenize(word_data)
print (nltk_tokens)
```

REMOVE DEFAULT STOPWORDS:

- Stopwords are words that do not contribute to the meaning of a sentence.
- Hence, they can safely be removed without causing any change in the meaning of the sentence.
- The NLTK library has a set of stopwords and can use these to remove stopwords from text and return a list of word tokens.

- Removing Stopwords: Stopwords are common words that are usually removed in text processing (e.g., "and", "the").

```
from nltk.corpus import stopwords
# Download stopwords
import nltk
nltk.download('stopwords')

stop_words = set(stopwords.words("english"))

text = "This is a sample sentence, showing off the stop words filtration."
words = word_tokenize(text)
filtered_words = [word for word in words if word.lower() not in
    stop_words]
print(filtered_words)  # Output: ['This', 'sample', 'sentence', ',', 'showing',
    'stop', 'words', 'filtration', '.']
```

STEMMING:

- Stemming is the process of getting the root form of a word.
- Stem or root is the part to which inflectional affixes (-ed, -ize, -de, -s, etc.) are added.
- The stem of a word is created by removing the prefix or suffix of a word.
- So, stemming a word may not result in actual words.

- If the text is not in tokens, then need to convert it into tokens.
- After converted strings of text into tokens, can convert the word tokens into their root form.
- There are mainly three algorithms for stemming.
- These are the **Porter Stemmer**, the **Snowball Stemmer** and the **Lancaster Stemmer**.
- Porter Stemmer is the most common among them.

- Stemming:

```
from nltk.stem import PorterStemmer
ps = PorterStemmer()
words = ["running", "runner", "ran", "runs"]
stemmed_words = [ps.stem(word) for word in
   words]
print(stemmed_words)  # Output: ['run', 'runner',
   'ran', 'run']
```

LEMMATIZATION:

- Like stemming, lemmatization also converts a word to its root form.
- The only difference is that lemmatization ensures that the root word belongs to the language.
- We will get valid words if we use lemmatization.
- In NLTK, we use the WordNetLemmatizer to get the lemmas of words.
- Need to provide a context for the lemmatization.
- So, we add the part-of-speech as a parameter.

```
from nltk.stem import WordNetLemmatizer
# Download WordNet data
nltk.download('wordnet')

lemmatizer = WordNetLemmatizer()
words = ["running", "runner", "ran", "runs"]
lemmatized_words = [lemmatizer.lemmatize(word,
   pos='v') for word in words]
print(lemmatized_words)  # Output: ['run', 'runner',
   'run', 'run']
```

Regular Expressions

- A Regular Expression or RegEx is a special sequence of characters that uses a **search pattern** to find a string or set of strings.
- It can detect the presence or absence of a text by matching it with a particular pattern and also can split a pattern into one or more sub-patterns.
- Python has a built-in module named "**re**" that is used for regular expressions in Python.

import re

METACHARACTERS

MetaCharacters	Description	Example
\	Used to drop the special meaning of character following it	"\d"
[ ]	Represent a character clas	"[a-m]"
^	Matches the beginning	"^hello"
$	Matches the end	"planet$"
.	Matches any character except newline	"he..o"
|	Means OR (Matches with any of the characters separated by it.	"falls|stays"
?	Matches zero or one occurrence	"he.?o"
*	Any number of occurrences (including 0 occurrences)	"he.*o"
+	One or more occurrences	"he.+o"
{}	Indicate the number of occurrences of a preceding regex to match.	"he.{2}o"
()	Enclose a group of Regex	

Special Characters

Special Sequence	Description	Example	
\A	Matches if the string begins with the given character	\Afor	for geeks
			for the world
\b	Matches if the word begins or ends with the given character. \b(string) will check for the beginning of the word and (string)\b will check for the ending of the word.	\bge	geeks
			get
\B	It is the opposite of the \b i.e. the string should not start or end with the given regex.	\Bge	together
			forge
\d	Matches any decimal digit, this is equivalent to the set class [0-9]	\d	123
			Gee1
\D	Matches any non-digit character, this is equivalent to the set class [^0-9]	\D	geeks
			geek1

SPECIAL CHARACTERS

Special Sequence	Description	Example	
\s	Matches any whitespace character.	\s	gee ks
			a bc a
\S	Matches any non-whitespace character	\S	a bd
			abcd
\w	Matches any alphanumeric character, this is equivalent to the class [a-zA-Z0-9_].	\w	123
			geeKs4
\W	Matches any non-alphanumeric character.	\W	>$
			gee<>
\Z	Matches if the string ends with the given regex	ab\Z	abcdab
			abababab

RegEx Functions

Function	Description
re.findall()	finds and returns all matching occurrences in a list
re.compile()	Regular expressions are compiled into pattern objects
re.split()	Split string by the occurrences of a character or a pattern.
re.sub()	Replaces all occurrences of a character or pattern with a replacement string.
re.escape()	Escapes special character
re.search()	Searches for first occurrence of character or pattern

1. re.findall()

- Return all non-overlapping matches of pattern in string, as a list of strings.
- Finding all occurrences of a pattern

```
import re
string = """Hello my Number is 123456789 and
          my friend's number is 987654321"""
regex = '\d+’
match = re.findall(regex, string)
print(match)
```

Output

```
['123456789', '987654321']
```

2. re.compile()

- Regular expressions are compiled into pattern objects, which have methods for various operations such as searching for pattern matches or performing string substitutions.
- **find and list all lowercase letters from ‘a’ to ‘e’**

```
import re
p = re.compile('[a-e]’)
print(p.findall("Aye, said Mr. Gibenson Stark"))

Output
['e', 'a', 'd', 'b', 'e', 'a']
```

2. re.compile()

```
import re
p = re.compile('\d')
print(p.findall("I went to him at 11 A.M. on 4th July 1886"))

p = re.compile('\d+')
print(p.findall("I went to him at 11 A.M. on 4th July 1886"))
```

Output

```
['1', '1', '4', '1', '8', '8', '6']
['11', '4', '1886']
```

2. re.compile()

```
import re
p = re.compile('ab*')
print(p.findall("ababbaabbb"))
```

Output

```
['ab', 'abb', 'a', 'abbb']
```

3. re.split()

- Split string by the occurrences of a character or a pattern, upon finding that pattern, the remaining characters from the string are returned as part of the resulting list.
- **Syntax :**

  re.split(pattern, string, maxsplit=0, flags=0)
- **pattern** denotes the regular expression,
- **string** is the given string in which pattern will be searched for and in which splitting occurs,
- **maxsplit** if not provided is considered to be zero '0', and if any nonzero value is provided, then at most that many splits occur. If maxsplit = 1, then the string will split once only, resulting in a list of length 2.
- **The flags** are very useful and can help to shorten code, they are not necessary parameters, eg: flags = re.IGNORECASE, in this split, the case, i.e. the lowercase or the uppercase will be ignored.

3. re.split()

```
from re import split

print(split('\W+', 'Words, words , Words'))
print(split('\W+', "Word's words Words"))
print(split('\W+', 'On 12th Jan 2016, at 11:02 AM'))
print(split('\d+', 'On 12th Jan 2016, at 11:02 AM’))
```

Output

```
['Words', 'words', 'Words']
['Word', 's', 'words', 'Words']
['On', '12th', 'Jan', '2016', 'at', '11', '02', 'AM']
['On ', 'th Jan ', ', at ', ':', ' AM']
```

4. re.sub()

- The ‘sub’ in the function stands for SubString, a certain regular expression pattern is searched in the given string (3rd parameter), and upon finding the substring pattern is replaced by repl (2nd parameter), count checks and maintains the number of times this occurs.
- Syntax:

  re.sub(pattern, repl, string, count=0, flags=0)

- Example:
- First statement replaces all occurrences of ‘ub’ with ‘~*’ (case-insensitive): 'S~*ject has ~*er booked already'.
- Second statement replaces all occurrences of ‘ub’ with ‘~*’ (case-sensitive): 'S~*ject has Uber booked already'.
- Third statement replaces the first occurrence of ‘ub’ with ‘~*’ (case-insensitive): 'S~*ject has Uber booked already'.
- Fourth replaces ‘AND’ with ‘ & ‘ (case-insensitive): 'Baked Beans & Spam’.

4. re.sub()

```
import re
print(re.sub('ub', '~*', 'Subject has Uber booked already',
          flags=re.IGNORECASE))
print(re.sub('ub', '~*', 'Subject has Uber booked already'))
print(re.sub('ub', '~*', 'Subject has Uber booked already',
          count=1, flags=re.IGNORECASE))
print(re.sub(r'\sAND\s', ' & ', 'Baked Beans And Spam',
          flags=re.IGNORECASE))
```

Output

```
S~*ject has ~*er booked already
S~*ject has Uber booked already
S~*ject has Uber booked already
Baked Beans & Spam
```

5. re.escape()

- Returns string with all non-alphanumerics backslashed, this is useful if you want to match an arbitrary literal string that may have regular expression metacharacters in it.
- re.escape() is used to escape special characters in a string, making it safe to be used as a pattern in regular expressions. It ensures that any characters with special meanings in regular expressions are treated as literal characters.

re.escape(string)

5. re.escape()

```
import re

# Pattern
pattern = 'https://www.google.com/'

# Using escape function to escape metacharacters
result = re.escape( pattern)

# Printing the result
print("Result:", result)
```

Output

```
Result: https://www\.google\.com/
```

6. re.search()

- This method either returns None (if the pattern doesn't match), or a re.MatchObject contains information about the matching part of the string. This method stops after the first match, so this is best suited for testing a regular expression more than extracting data.
- This code uses a regular expression to search for a pattern in the given string. If a match is found, it extracts and prints the matched portions of the string.

6. re.search()

```
import re

line = "Cats are smarter than dogs"
matchObj = re.search( r'than', line)

print (matchObj.start(), matchObj.end())
print ("matchObj.group() : ", matchObj.group())
print(res)
```

Output

```
17 21
matchObj.group() :  than
```