**MDS472C: NATURAL LANGUAGE PROCESSING**

**OBSERVATION NOTEBOOK**

**Name: Neelanjan Dutta**

**Roll No: 2448040**

**Class:** **4MDS**

**Contents**

[**LAB No. 1: Text Processing and Regular Expression** 3](#_Toc201867183)

[**LAB No. 2:  Edit Distance and Applications** 26](#_Toc201867184)

**DATE:** **19/06/25**

# **LAB No. 1: Text Processing and Regular Expression**

**Question 1:**

**Installing NLTK**, NLTK.book, and practicing the NLP Environment using exercises 1 and 2 from the given link. [Language Processing and Python (nltk.org)](https://www.nltk.org/book/ch01.html).

**Draft Plan:**

**Program description:**

This lab focuses on setting up the NLTK (Natural Language Toolkit) library in a Python environment and practicing basic NLP (Natural Language Processing) tasks using exercises from the NLTK documentation. The main objectives are:

* Install NLTK and download necessary datasets (NLTK.book).
* Practice basic NLP environment setup and usage.
* Perform simple text processing and calculations as per the exercises.

**Program Logic:**

* **Library Installation:** Used pip to install the NLTK library.
* **Data Download:** Used nltk.download the book to download the required NLTK datasets and corpora.
* **Environment Setup:** Imported NLTK and necessary modules.
* **Basic NLP Tasks:** Used Python interpreter as a calculator and computed combinatorial problems (e.g., number of possible strings from a given alphabet).
* **Algorithms:** Simple arithmetic and combinatorial calculations.
* **Data Types:** Integers, strings, variables.
* **Key Libraries:** NLTK, Python built-in libraries.

**Program:**

!pip install nltk

import nltk

nltk.download('book')

from nltk.book import \*

**Exercise 1:** Try using the Python interpreter as a calculator and typing expressions like 12 / (4 + 1).

**Code:**

numerator = 12

denominator = 4+1

result = numerator/denominator

print ("The value of the result is:”, result)

**Exercise 2:** Given an alphabet of 26 letters, there are 26 to the power 10, or 26 \*\* 10, ten-letter strings we can form. That works out to 141167095653376. How many hundred-letter strings are possible?

**Code:**

alphabet\_size = 26

strings = 100

total\_combinations = alphabet\_size \*\* strings

print ("The total number of 100 letter strings possible are:”, total\_combinations)

**Test Case with Actual and Expected Input/Output**

|  |  |  |  |
| --- | --- | --- | --- |
| Test Cases | Input | Actual Output | Expected Output |
| Exercise 1 (Calculator) | **numerator = 12, denominator = 4 + 1** | **2.4** | **2.4** |
| Exercise 2 (String Combinations) | **alphabet\_size = 26, strings = 100** | **26\*\*100** | **Large number (26^100)** |

**Question 2:**

**Text Processing (Basics)**  
(a) Define a paragraph as a string value. :  
(b) Write a program to print the number of total words and total unique words in the paragraph.  
(c) Write a program to find the frequency of all words and display the most and least frequent words.  
(d) Write a program to find the longest word in the paragraph

**Draft Plan:**

**Program Description:**

The objective is to analyze a given paragraph by performing basic text processing tasks:

1. **Count total words and unique words** in the paragraph.
2. **Determine the frequency of each word** and identify the most and least frequent words.
3. **Find the longest word** in the paragraph.  
   The program should process the paragraph, handle punctuation, and display the required outputs clearly.

**Program Logic:**

* **Input:** A paragraph (multi-line string).
* **Preprocessing:** Remove punctuation and convert text to lowercase for uniform processing.
* **Tokenization:** Split the paragraph into individual words.
* **Counting:**
  + Calculate the total number of words.
  + Count unique words using a set.
* **Frequency Analysis:**
  + Use collections.Counter to count word frequencies.
  + Identify the most and least frequent words.
* **Longest Word:**
  + Iterate through the words to find the word with the maximum length.
* **Output:** Display the results for each task.

**Algorithm:**

1. **Input paragraph.**
2. **Remove punctuation and convert to lowercase.**
3. **Split into words.**
4. **Count total and unique words.**
5. **Count the frequency of each word.**
6. **Find the most and least frequent words.**
7. **Find the longest word.**
8. **Display results**

**Libraries:**

* string (for punctuation handling)
* collections.Counter (for frequency counting)

**Data Types:**

* str (paragraph, words)
* list (word list)
* set (unique words)
* Counter (word frequencies)

**Program:**

1. **Define a string containing a paragraph as the value.**

paragraph = """

Global warming is the phenomenon of a gradual increase in the temperature near the earth’s surface.

This phenomenon has been observed over the past one or two centuries. This change has disturbed the climatic pattern of

the earth. However, the concept of global warming is quite controversial but the scientists have provided relevant data

in support of the fact that the temperature of the earth is rising constantly.

There are several causes of global warming, which have a negative effect on humans, plants and animals.

These causes may be natural or might be the outcome of human activities.

In order to curb the issues, it is very important to understand the negative impacts of global warming.

"""

1. **Write a program to print the number of total words and total unique words in the paragraph.**

import string

# Converting paragraph into a list of words

# Here we would remove punctuation and convert to lowercase

cleaned\_paragraph = paragraph.translate(str.maketrans('', '', string.punctuation)).lower()

# Split the cleaned paragraph into individual words

words = cleaned\_paragraph.split()

**# Finding the total words and unique words**

total\_words = len(words)

unique\_words = len(set(words))

print("Total number of words:", total\_words)

print("Total number of unique words:", unique\_words)

1. **Find the frequency of all words and also display the most and least frequent word**

from collections import Counter

word\_count = Counter(words)

# Cleaning any punctuation and converting to lowercase

cleaned\_paragraph = paragraph.translate(str.maketrans('', '', string.punctuation)).lower()

# Splitting into words

words = cleaned\_paragraph.split()

print("\nWord Frequencies:")

for word in word\_count:

print(word, ":", word\_count[word], end = " ")

**# Most frequent word**

most\_frequent = max(word\_count.values())

for word in word\_count:

if word\_count[word] == most\_frequent:

print("Most frequent word:")

print(word,":",word\_count[word])

break

**# Least frequent word**

print("Least frequent word:")

for word in word\_count:

if word\_count[word] == 1:

print(word, end = ",")

1. **Find the longest word in the paragraph.**

longest\_word = ""

for word in word\_count:

if len(word) > len(longest\_word):

longest\_word = word

print("The longest word in the paragraph is:", longest\_word)

**Test Case with Actual and Expected Input/Output**

|  |  |  |  |
| --- | --- | --- | --- |
| Test Case | Input | Actual Output | Expected Output |
| Total words, Unique words | Given paragraph | 116 73 | 116 73 |
| Most frequent word | Given paragraph | the: 14 | the: 14 |
| Longest word | Given paragraph | controversial | controversial |

**Question 3:**

**Regular Expression**

Solve exercises 2.1 and 2.2 from the textbook of Speech and Language Processing by Daniel Jurafsky and team

**Draft Plan:**

**Program Description:**

This program uses **Python regular expressions (re module)** to solve problems from Jurafsky’s NLP textbook. The tasks include pattern matching for alphabetic strings, repeated words, specific word placements, word boundaries, sentence structure, and multiple constraints. It focuses on writing and testing regex patterns for various natural language scenarios.

**Program Logic:**

**Program Logic:**

**Tools Used:**

* re module from Python: to compile, match, and search patterns.
* re.fullmatch(), re.search(): to verify complete or partial matches.
* re.IGNORECASE: to match regardless of case.

**Regex Patterns and Logic:**

1. **Alphabetic strings (2.1.1):**  
   Pattern: ^[A-Za-z]+$
   * Matches only strings with letters (upper/lower), no numbers or symbols.
2. **Lowercase strings ending in 'b' (2.1.2):**  
   Pattern: ^[a-z]\*b$
   * All lowercase letters, ending specifically in b.
3. **All as surrounded by bs (2.1.3):**  
   Pattern: ^(b|bab)\*$
   * Every a must be in a bab pattern. Only b and bab sequences allowed.
4. **Consecutive repeated words (2.2.1):**  
   Pattern: \b(\w+)\s+\1\b
   * Checks for same word repeated with space (like "the the").
5. **String starting with integer, ending with word (2.2.2):**  
   Pattern: ^\d+.\*\b([A-Za-z]+)\b$
   * Begins with digits, ends with an alphabetic word.
6. **Contains both grotto and raven (2.2.3):**  
   Patterns: \bgrotto\b and \braven\b
   * Uses re.search() to find both words as complete matches, not substrings.
7. **First word in a sentence (2.2.4):**  
   Pattern: \b\w+
   * Captures the first word (skips quotes/punctuation if present).

**Program**

**2.1 Write regular expressions for the following languages.**

**(a) the set of all alphabetic strings**

import re

# All alphabetic strings (uppercase + lowercase allowed)

pattern = r'^[A-Za-z]+$' # + to ensure at least one character

test1 = "Helloworld"

print(bool(re.fullmatch(pattern,test1)))

test2 = "Hello123"

print(bool(re.fullmatch(pattern,test2)))

* 1. **the set of all lower case alphabetic strings ending in a b**

# Lowercase strings ending with 'b'

pattern = r'^[a-z]\*b$' # \* here would help with zero or more lowercase letters

test1 = "abcb"

print(bool(re.fullmatch(pattern, test1)))

test2 = "abcB"

print(bool(re.fullmatch(pattern, test2)))

* 1. the set of all strings from the alphabet {a,b} such that each **a** is immediately preceded by and immediately followed by a **b**

pattern = r'^(b|bab)\*$'

# Test cases

print(bool(re.fullmatch(pattern, "")))

print(bool(re.fullmatch(pattern, "bab")))

print(bool(re.fullmatch(pattern, "babbabbab"))) #multiple valid bab

print(bool(re.fullmatch(pattern, "bbbabbabbb"))) # all a’s are inside bab

print(bool(re.fullmatch(pattern, "aba")))

print(bool(re.fullmatch(pattern, "ba")))

print(bool(re.fullmatch(pattern, "abbab")))

print(bool(re.fullmatch(pattern, "babbbababa"))) #last a is not surrounded by b

**2.2 Write regular expressions for the following languages. By “word”, we mean an alphabetic string separated from other words by whitespace, any relevant punctuation, line breaks, and so forth.**

* 1. **the set of all strings with two consecutive repeated words (e.g., “Humbert Humbert” and “the the” but not “the bug” or “the big bug”);**

# Repeated words with space in between (case insensitive)

pattern = r'\b(\w+)\s+\1\b'

test1 = "Humbert Humbert"

print(bool(re.fullmatch(pattern, test1)))

test2 = "the big bug"

print(bool(re.fullmatch(pattern, test2))

* 1. **all strings that start at the beginning of the line with an integer and at the end of the line with a word;**

# Integer at start, word at end, entire line

pattern = r'^\d+.\*\b([A-Za-z]+)\b$'

test1 = "123 buy now"

print(bool(re.fullmatch(pattern, test1)))

test2 = "Hello 123"

print(bool(re.fullmatch(pattern, test2)))

* 1. **all strings that have both the word grotto and the word raven in them (but not, e.g., words like grottos that merely contain the word grotto);**

# Checking both words exist as whole words

pattern\_grotto = r'\bgrotto\b'

pattern\_raven = r'\braven\b'

# List of test sentences

test\_cases = [

"the grotto and the raven",

"raven in the grotto",

"grottos and ravens",

"the raven flew"

]

# Check each sentence

for s in test\_cases3:

found = (re.search(pattern\_grotto, s, re.IGNORECASE) and

re.search(pattern\_raven, s, re.IGNORECASE))

print(s, "->", bool(found))

* 1. **write a pattern that places the first word of an English sentence in a register. Deal with punctuation.**

pattern = r'\b\w+'

text = '"Hello, how are you?" she asked.'

match = re.search(pattern,text)

if match:

print("First word is:", match.group())

else:

print("No match found")

**Test Case with Actual and Expected Input/Output**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test Case | Input | Pattern/Logic | Expected Output | Actual Output |
| 1 | "Helloworld" | ^[A-Za-z]+$ | TRUE | TRUE |
| 2 | "Hello123" | ^[A-Za-z]+$ | FALSE | FALSE |
| 3 | "abcb" | ^[a-z]\*b$ | TRUE | TRUE |
| 4 | "abcB" | ^[a-z]\*b$ | FALSE | FALSE |
| 5 | "bab", "babbabbab" | ^(b|bab)\*$ | TRUE | TRUE |
| 6 | "aba", "ba" | ^(b|bab)\*$ | FALSE | FALSE |
| 7 | "Humbert Humbert" | \b(\w+)\s+\1\b | TRUE | TRUE |
| 8 | "the big bug" | \b(\w+)\s+\1\b | FALSE | FALSE |
| 9 | "123 buy now" | ^\d+.\*\b([A-Za-z]+)\b$ | TRUE | TRUE |
| 10 | "Hello 123" | ^\d+.\*\b([A-Za-z]+)\b$ | FALSE | FALSE |
| 11 | "raven in the grotto" | \braven\band\bgrotto\b | TRUE | TRUE |
| 12 | "grottos and ravens" | \braven\band\bgrotto\b | FALSE | FALSE |
| 13 | '"Hello, how are you?"' | \b\w+ | First word: Hello | Hello |

**ADDITIONAL QUESTIONS:**

**Practice Q1. Text Normalization**

Perform EDA operation for the text file containing a chapter of a text book or story book.

**Draft Plan :**

**Program Description:**

The objective of this program is to perform basic Exploratory Data Analysis (EDA) and text normalization on a given text file (e.g., a chapter from a storybook or textbook). The goal is to clean the text, analyze its structure, and extract useful information such as word counts, sentence counts, and frequently occurring words. A word cloud is also generated to visually represent the most frequent terms in the text.

**Program Logic:**

**Input:** A plain text file named "story.txt" containing a chapter of a book.

Text Normalization Steps:

**Convert to lowercase:** Standardizes all words to the same case for accurate comparison.

**Remove punctuation:** Eliminates symbols like commas and periods using string.punctuation and str.translate().

**Tokenize into words:** Splits the cleaned text into individual words using .split().

**EDA Operations Performed:**

**Total characters:** Counted from the original text using len(text).

**Total words:** Counted after cleaning and splitting.

**Unique words:** Calculated using set() to remove duplicates.

**Total lines:** Found by splitting the text on newline characters.

**Total sentences:** Detected using sent\_tokenize() from the NLTK library.

**Word Frequency Analysis:** Used collections.Counter to count how often each word appears. Displayed the top 10 most frequent words.

**Word Length Analysis:** Longest word found by checking word lengths using a loop and shortest word found by comparing all words using a conditional check.

**Visualization:** A **Word Cloud** was generated using the WordCloud class to visually show word frequencies. Displayed using matplotlib.pyplot.

**Libraries Used:**

**string**: for punctuation removal.

**collections.Counter**: for word frequency analysis.

**nltk.sent\_tokenize**: for sentence segmentation.

**wordcloud**: for generating a word cloud.

**matplotlib.pyplot**: for displaying the word cloud.

**Data Types Used:**

**String** for reading and processing the text.

**List** for storing words and lines.

**Set** for finding unique words.

**Dictionary (Counter)** for word frequency.

**Integers** for counts (characters, lines, words).

**NoneType / String** for shortest/longest word comparison.

**Python Program –**

# **Loading the file**

with open("chapter\_1.txt", 'r') as file: text = file.read()

# Cleaning the text

# Convert to lowercase lower\_case\_text = text.lower() # Remove punctuation

no\_punct\_text = lower\_case\_text.translate(str.maketrans('', '', string.punctuation)) # Split into words

words = no\_punct\_text.split()

# Basic Preprocessing

# Total characters total\_characters = len(text)

print("Total characters:", total\_characters)

# Total words (after cleaning) total\_words = len(words) print("Total words:", total\_words) # Unique words

total\_unique\_words = len(set(words)) print("Total unique words:", total\_unique\_words) # Total lines

lines = text.split('\n') print("Total lines:", len(lines)) # Total sentences

sentences = sent\_tokenize(text) print("Total sentences:", len(sentences))

# Word Frequency Analysis

word\_freq = Counter(words)

print("\n Top 10 most frequent words:")

for word, count in word\_freq.most\_common(10): print(f"{word}: {count}",end =", ")

# Longest and Shortest Words

longest\_word = "" for word in words:

if len(word) > len(longest\_word): longest\_word = word

print("Longest word:", longest\_word) shortest\_word = None

for word in words:

if shortest\_word is None or len(word) < len(shortest\_word): shortest\_word = word

print("Shortest word:", shortest\_word)

# Word cloud

from wordcloud import WordCloud

wordcloud = WordCloud(width=800, height=400, background\_color='white').generate(' '.join(words))

plt.figure(figsize=(10, 5)) plt.imshow(wordcloud, interpolation='bilinear') plt.axis("off")

plt.title("Word Cloud")

plt.show()

# Word cloud

from wordcloud import WordCloud

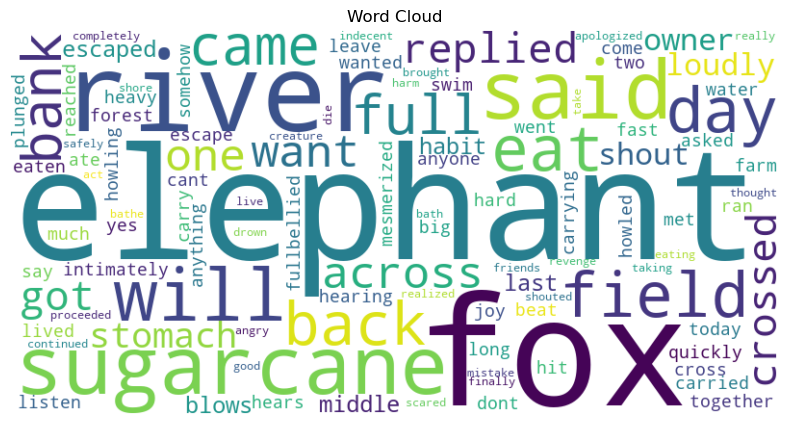
wordcloud = WordCloud(width=800, height=400, background\_color='white').generate(' '.join(words))

plt.figure(figsize=(10, 5)) plt.imshow(wordcloud, interpolation='bilinear') plt.axis("off")

plt.title("Word Cloud") plt.show()

**Test Case with Actual and Expected Input/Output**

|  |  |
| --- | --- |
| Operation | Output |
| Total characters | 2219 |
| Total words | 418 |
| Total unique words | 171 |
| Total lines | 24 |
| Total sentences | 36 |
| Top 10 frequent words | the: 61, elephant: 20, fox: 19, to: 16, and: 14, river: 9, i: 8, a: 7, of: 7, not: 6, |
| Longest word | fullbellied |
| Shortest word | a |

**Word Cloud**

**Practice Q2. From TextBook Exercises: Steve Bird and team**

**24. Write expressions for finding all words in text6 that meet the conditions listed below. The result should be in the form of a list of words: ['word1', 'word2', ...].**

Ending in ize

Containing the letter z

Containing the sequence of letters pt

Having all lowercase letters except for an initial capital (i.e., titlecase)

**Draft Plan -**

**Program Description:**

The goal of this program is to perform word-level filtering operations on text6 from the NLTK book corpus (which contains the text of *Monty Python and the Holy Grail*). The task is based on Exercise 24 from the NLTK textbook by Steven Bird and team. The objective is to extract words that match specific conditions related to spelling, letter sequences, and casing.

**Program Logic:**

**Corpus Used:** text6 from NLTK’s nltk.book collection. It must be imported using: from nltk.book import \*

**Data Type:**

**set(text6)** is used to remove duplicate words.

**list** is used to collect and return results for each condition.

**Libraries Used:**

nltk.book to access predefined texts (specifically text6).

No external libraries are needed for this task.

**Logic for Each Condition:**

**(a) Words ending in ‘ize’:**

Used **str.endswith('ize')** to check if each word ends with “ize”. Words are taken from set(text6) to avoid duplicates. Result stored in a list and printed.

**(b) Words containing the letter ‘z’:**

Used 'z' in word to check if the word contains the character ‘z’. Matches both lowercase and capital ‘z’ by default (though most words are lowercase in this corpus).

**(c) Words containing the sequence ‘pt’:**

Used 'pt' in word to match this exact letter sequence anywhere in the word.

**(d) Words with only the first letter capitalized (Titlecase):**

Used word.istitle() which returns True only if the first character is uppercase and the rest are lowercase. This is useful to identify proper nouns or sentence-start words.

**Python Program -**

**(a.) Ending in ize**

result = []

for word in set(text6):

if word.endswith('ize'): result.append(word)

print("Words ending in 'ize':", result)

**(b.) Containing the letter z**

result = []

for word in set(text6): if 'z' in word:

result.append(word) print("Words containing 'z':", result)

**(c.) Containing the sequence of letters pt**

result = []

for word in set(text6): if 'pt' in word:

result.append(word) print("Words containing 'pt':", result)

**(d.) Having all lowercase letters except for an initial capital (i.e., titlecase)**

result = []

for word in set(text6): if word.istitle():

result.append(word) print("Titlecase words:", result)

25. Define *sent* to be the list of words ['she', 'sells', 'sea', 'shells', 'by', 'the', 'sea', 'shore']. Now write code to perform the following tasks:

(a) Print all words beginning with sh

(b) Print all words longer than four characters

**Draft Plan** -

**Program Description:**

This program performs simple filtering operations on a given list of words. The goal is to extract and print specific words from the list sent = ['she', 'sells', 'sea', 'shells', 'by', 'the', 'sea', 'shore'] based on two conditions:

Words that begin with the substring ‘sh’

Words that are longer than 4 characters

**Program Logic**

**Input Data:** A predefined list of strings named sent, representing a short sentence split into words.

**Tasks Performed:**

**Filter words starting with "sh"** using the built-in method startswith('sh').

**Filter words longer than 4 characters** by checking the length of each word using len(word) > 4.

**Iteration Method:** A simple for loop is used to iterate over each word in the list.

**Output:** Words satisfying the given condition are printed directly using print().

**Libraries Used:** No external libraries are used — only built-in Python functions.

Data Types Used:

list: to store the words (sent)

str: for individual words

int: for word length comparison

**Python program -**

**(a.) Print all words beginning with sh**

sent = ['she', 'sells', 'sea', 'shells', 'by', 'the', 'sea', 'shore']

for word in sent:

if word.startswith('sh'):

print(word)

**(b.) Print all words longer than four characters for word in sent:**

for word in sent:

if len(word) > 4:

print(word)

26. **What does the following Python code do? sum(len(w) for w in text1). Can you use it to work out the average word length of a text?**

**Answer:**

text1 is a list of words (comes from from nltk.book import \*).

len(w) gives the length of each word w in the list.

sum(...) adds up the lengths of all the words.

So, it calculates the total number of characters in the entire text1, excluding spaces and punctuation between words.

Yes, we can use it to calculate the average word length by dividing it by the total number of words.

i.e The avg word length is; **average = sum(len(w) for w in text1) / len(text1)**

where;

sum(len(w) for w in text1) = total number of characters in all words and len(text1) = total number of words

**27. Define a function called vocab\_size(text) that has a single parameter for the text, and which returns the vocabulary size of the text.**

**Draft Plan** -

**Program Description:**

The purpose of this program is to define a function that calculates the vocabulary size of a given text. The vocabulary size is the total number of unique words present in the text.

**Program Logic:**

**Function Name:** vocab\_size(text)

**Parameter:** text **–** a list of words (e.g., NLTK texts like text1, text2, etc.)

**Logic:** Convert the list of words into a set to remove all duplicates. Return the length of the set, which gives the total count of distinct words.

**Example Use:** Called with text1 (Moby Dick) to find its vocabulary size.

Libraries Used: None. Only built-in Python functions like set() and len() are used.

**Data Types Used:**

list: input text (sequence of words)

set: to store unique words

int: for the vocabulary cont (length of the set)

**Python program and Output** –

def vocab\_size(text):

unique\_words = set(text) # remove duplicates

return len(unique\_words)

# Vocabulary size of text1 (Moby Dick)

print("Vocabulary size of text1:", vocab\_size(text1))

**Output:** Vocabulary size of text1: 19317

28. **Define a function percent(word, text) that calculates how often a given word occurs in a text, and expresses the result as a percentage.**

**Draft Plan** -

**Program Description:**

This program defines a function percent(word, text) that calculates the percentage frequency of a given word in a text. It shows how often a specific word appears relative to the total number of words, and returns the result rounded to two decimal places.

**Program Logic:**

**Function Name:** percent(word, text)

**Parameters:**

word: the word to be searched.

text: a list of words (e.g., NLTK text like text2).

**Logic:** Use text.count(word) to count the number of times the word appears in the text. Use len(text) to get the total number of words. Calculate percentage and round the result to two decimal places using round().

**Example Output:** Used to find how frequently 'a' appears in text2.

**Libraries Used:** None. All operations are done using basic Python functions.

**Data Types Used:**

str: for the target word.

list: for the text input.

int: for word count and total length.

float: for percentage value.

**Python program and Output** –

def percent(word, text):

count = text.count(word) # number of times word appears total = len(text) # total number of words

return round(100 \* count / total, 2)

print("Percent of 'the' in text2:", percent('a', text2), "%")

**Output:** Percent of 'the' in text2: 1.44 %

**DATE:** **26/06/25**

# **LAB No. 2:  Edit Distance and Applications**

**Question 2: Edit distance (Implementation)**

2.6 Implement a minimum edit distance algorithm and use your hand-computed results to check your code.

str1 **=** "leda"

str2 **=** "deal"

distance **=** levenshtein\_distance(str1, str2)

print(f"The edit distance between '{str1}' and '{str2}' is {distance}")

**Draft Plan:**

**Program Description:**

The objective of this program is to calculate the **Levenshtein Distance** (also called Minimum Edit Distance) between two input strings.  
Levenshtein Distance represents the **minimum number of single-character edits** (insertions, deletions, or substitutions) required to convert one string into another.

This algorithm is widely used in applications such as:

* **Spell checking**
* **DNA sequence comparison**
* **Natural Language Processing**
* **Plagiarism detection**

**Program Logic:**

**Concept Used:**

**Dynamic Programming** is used to compute the edit distance efficiently.  
A 2D matrix, dp, is created where dp[i][j] represents the edit distance between the first i characters of str1 and the first j characters of str2.

**Algorithm Steps:**

1. **Initialize a (m+1) x (n+1) matrix**, where m and n are the lengths of the two strings.
2. Fill the **first row and column** with increasing integers (represents insertion/deletion from empty string).
3. Use the recurrence:
4. if str1[i - 1] == str2[j - 1]:
5. cost = 0 #No cost if characters matches
6. else:
7. cost = 1 #Cost of 1 if charaters do not match
8. deletion = dp[i - 1][j] + 1
9. insertion = dp[i][j - 1] + 1
10. substitution = dp[i - 1][j - 1] + cost
11. After filling the table, the **final answer is in dp[m][n]**.
12. The matrix is printed to visualize the steps.

**Libraries Used:**

No external libraries used; only built-in Python features.

**Data Types:**

* str for input strings
* int for index calculations and matrix elements
* list for the 2D matrix

**Program (Python code):**

# Levenshtein Distance (Minimum Edit Distance)

#Defining the function to compute edit distance

def levenshtein\_distance(str1, str2):

m = len(str1)

n = len(str2)

#Creating a (m+1) x (n+1) matrix

dp = []

for i in range(m + 1):

row = []

for j in range(n + 1):

row.append(0)

dp.append(row)

#Initializing first row and column

for i in range(m + 1):

dp[i][0] = i

for j in range(n + 1):

dp[0][j] = j

#Filling the matrix using edit distance rules

for i in range(1, m + 1):

for j in range(1, n + 1):

if str1[i - 1] == str2[j - 1]:

cost = 0 #No cost if characters matches

else:

cost = 1 #Cost of 1 if charaters do not match

deletion = dp[i - 1][j] + 1

insertion = dp[i][j - 1] + 1

substitution = dp[i - 1][j - 1] + cost

dp[i][j] = min(deletion, insertion, substitution)

#Edit distance matrix

print("\nEdit Distance Matrix:")

print(" ", end="")

for ch in str2:

print(f"{ch:>3}", end="")

print()

for i in range(m + 1):

if i == 0:

print(" ", end=" ")

else:

print(str1[i - 1], end=" ")

for j in range(n + 1):

print(f"{dp[i][j]:>3}", end="")

print()

#Final edit distance

return dp[m][n]

#Give input to get the distance

str1 = input("Enter the first string: ")

str2 = input("Enter the second string: ")

distance = levenshtein\_distance(str1, str2)

print("\nThe edit distance between '{}' and '{}' is: {}".format(str1, str2, distance))

**Test cases:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test case | Input (str 1) | Input (str 2) | Actual output | Expected output |
| Test case 1 | **kitten** | **sitting** | **3** | **3** |
| Test case 2 | **sunday** | **saturday** | **3** | **3** |
| Test case 3 | **school** | **schoo** | **1** | **1** |
| Test case 4 | **python** | **typhoon** | **3** | **3** |
| Test case 5 | **a** | **a** | **0** | **0** |
| Test case 6 | **leda** | **deal** | **3** | **3** |
| Test case 7 | **drive** | **brief** | **3** | **3** |
| Test case 8 | **drive** | **divers** | **3** | **3** |
| Test case 9 | **distance** | **editing** | **5** | **5** |

**Question 3: Implement Sequence Alignment**

Write a program to align the given sequence of input text A and B

Input:

**Text A: AGGCTATCACCTGACCTCCAGGCCGATGCCC**

**Text B: TAGCTATCACGACCGCGGTCGATTTGCCCGAC**

**Output:**

**-AGGCTATCACCTGACCTCCAGGCCGA--TGCCC---**

**TAG-CTATCAC--GACCGC--GGTCGATTTGCCCGAC**

**Draft Plan:**

**Program Description:**

This program performs **sequence alignment** of two DNA strings (or general text strings) using a **dynamic programming algorithm** that minimizes the number of insertions, deletions, and substitutions.

It outputs the aligned sequences with gaps (-) to show **how the characters are matched or edited** to align the sequences optimally.

**Program Logic:**

**Algorithm:**

* **Dynamic Programming Table** (Edit Distance Matrix) is used to compute the minimum number of operations.
* **Backtracking** is applied to reconstruct the aligned sequences, inserting - where characters are added or removed.
* **Alignment output** helps visualize the match/mismatch/gap between two sequences.

**Python program:**

def align\_sequences(A, B):

n = len(A)

m = len(B)

# Create a DP table

dp = [[0 for \_ in range(m + 1)] for \_ in range(n + 1)]

# Initialize base cases

for i in range(n + 1):

dp[i][0] = i # all deletions

for j in range(m + 1):

dp[0][j] = j # all insertions

# Fill the DP table

for i in range(1, n + 1):

for j in range(1, m + 1):

if A[i - 1] == B[j - 1]:

cost = 0

else:

cost = 1

dp[i][j] = min(

dp[i - 1][j] + 1, # deletion

dp[i][j - 1] + 1, # insertion

dp[i - 1][j - 1] + cost # substitution/match

)

# Backtrace to build aligned strings

aligned\_A = []

aligned\_B = []

i, j = n, m

while i > 0 and j > 0:

if A[i - 1] == B[j - 1] or dp[i][j] == dp[i - 1][j - 1] + 1:

# Match or substitution

aligned\_A.append(A[i - 1])

aligned\_B.append(B[j - 1])

i -= 1

j -= 1

elif dp[i][j] == dp[i - 1][j] + 1:

# Deletion (gap in B)

aligned\_A.append(A[i - 1])

aligned\_B.append('-')

i -= 1

else:

# Insertion (gap in A)

aligned\_A.append('-')

aligned\_B.append(B[j - 1])

j -= 1

# Fill remaining gaps if any

while i > 0:

aligned\_A.append(A[i - 1])

aligned\_B.append('-')

i -= 1

while j > 0:

aligned\_A.append('-')

aligned\_B.append(B[j - 1])

j -= 1

# Reverse both sequences to get final alignment

aligned\_A.reverse()

aligned\_B.reverse()

# Join as strings

return ''.join(aligned\_A), ''.join(aligned\_B)

# Example

A = input("enter first string: ")

B = input("enter second string: ")

a\_aligned, b\_aligned = align\_sequences(A, B)

print(a\_aligned)

print(b\_aligned)

**Test Cases:**

**P.T.O.**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test Cases | | String 1 | String 2 | Aligned string 1 | Aligned string 2 |
| Test case 1 | **AGGCTATCACCTGACCTCCAGGCCGATGCCC** | | **TAGCTATCACGACCGCGGTCGATTTGCCCGAC** | **AGGCTATCACCTGACCTCCAGGCCGA--TG-CC--C** | **TAGCTATCA-C-GACC-GC-GGTCGATTTGCCCGAC** |
| Test case 2 | **abc** | | **abc** | **abc** | **abc** |
| Test case 3 | **drive** | | **divers** | **drive--** | **d-ivers** |