***Implementation Date : 20th June, 2025***

***Submission Date : 20th June, 2025***

***Lab1: Text Processing and Regular Expression***

**Program Question 1 :**

Installing NLTK, NLTK.book and practice the NLP Environment using the exercises :

1. Try using the Python interpreter as a calculator, and typing expressions like

12 / (4 + 1).

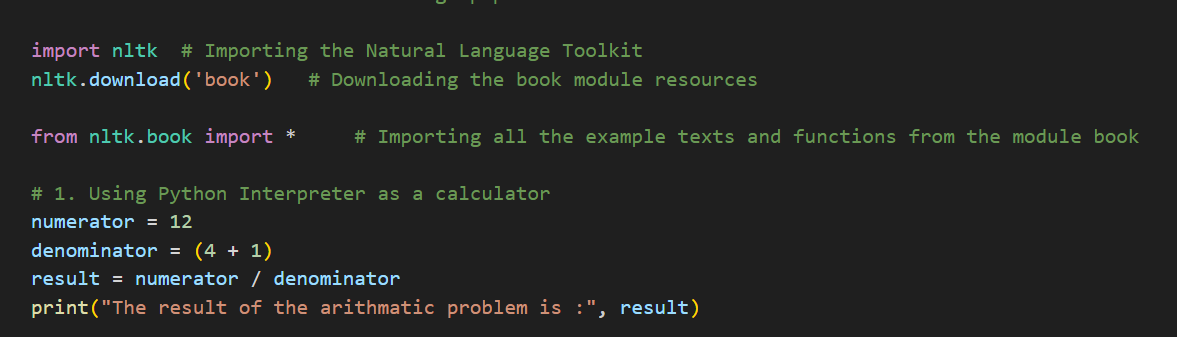
1. Given an alphabet of 26 letters, there are 26 to the power 10, or 26 \*\* 10, ten-letter strings we can form. That works out to 141167095653376. How many hundred-letter strings are possible?

**Observations :**

* 1. ***Program Description :*** This program evaluates a basic arithmetic expression

(12 / (4 + 1)) using Python. It helps learners get comfortable with the Python interpreter, syntax, and arithmetic operations.

* 1. ***Requirement :***
* Use Python to:
* Perform a calculation: 12 / (4 + 1)
* Store the result in a variable
* Print the result clearly
* Ensure the program is structured and readable
* Extend testing to other expressions (test cases)
  1. ***Program Logic :***
* Assign integer values to numerator and denominator
* Use arithmetic operators: +, /
* Perform the division and store it in a result variable
* Display the result using print()
* Use simple built-in data types: int, float
  1. ***Python Source Code :***



* 1. ***Test Cases :***

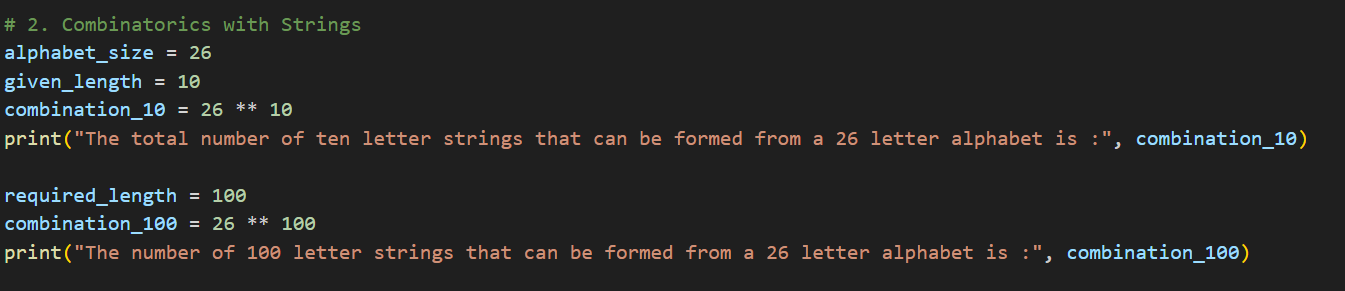
|  |  |  |
| --- | --- | --- |
| Test Cases | Code | Expected Output |
| 1 | 12 / (4 + 1) | 2.4 |
| 2 | 10 / 2 | 5.0 |
| 3 | 9 / (2 + 1) | 3.0 |
| 4 | (3 + 6) / 3 | 3.0 |
| 5 | 100 / (25 + 5) | 3.33… |
| 6 | (8 \* 2) / 4 | 4.0 |
| 7 | (15 – 5) / 5 | 2.0 |
|  |  |  |

* 1. ***Program Description :*** This sub-part calculates how many different strings of a

given length can be formed using the 26-letter English alphabet. It illustrates

how Python handles large exponentiation and supports combinatorial logic.

* 1. ***Requirement :***
* Define the size of the alphabet: 26 (A-Z)
* Compute and print:
* Total 10-letter strings → 26 \*\* 10
* Total 100-letter strings → 26 \*\* 100
* Use variables and display meaningful output
* Confirm Python handles large numbers gracefully
  1. ***Program Logic :***
* Use variable alphabet\_size = 26
* Apply exponentiation (\*\*) to compute total combinations
* Store results in combination\_10 and combination\_100
* Display using print() statements
* Use Python’s arbitrary-precision integers (int type)
* No need for external libraries beyond print and math operators.
  1. ***Python Source Code :***



* 1. ***Test Cases :***

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test Cases | Alphabet Size | String Length | Expression | Expected Output |
| 1 | 26 | 1 | 26 \*\* 1 | 26 |
| 2 | 26 | 2 | 26 \*\* 2 | 676 |
| 3 | 26 | 5 | 26 \*\* 5 | 11881376 |
| 4 | 26 | 10 | 26 \*\* 10 | 141167095653376 |
| 5 | 26 | 20 | 26 \*\* 20 | ≈1.99e+28 |
| 6 | 26 | 50 | 26 \*\* 50 | ≈8.03e+70 |
| 7 | 26 | 100 | 26 \*\* 100 | ≈2.03e+141 |
|  |  |  |  |  |

***Program Question 2 : Text Processing (Basics)***

1. Define a string containing a paragraph as the value.
2. Write a program to print the number of total words and total unique words in the paragraph.
3. Find the frequency of all words and also display the most and least frequent word.
4. Find the longest word in the paragraph.

**Observations :**

***Program Description :*** This program demonstrates basic natural language processing (NLP) operations on a paragraph using Python and NLTK. The tasks include:

* Lowercasing and tokenizing text
* Counting total and unique words
* Calculating word frequencies
* Finding the most and least frequent words
* Identifying the longest word.

***Requirement*** :

* Import necessary libraries (nltk, collections.Counter).
* Define a paragraph as a string.
* Preprocess the paragraph:
* Convert to lowercase
* Tokenize
* Remove punctuation
* Perform the following operations:
* Tokenization and preprocessing
* Count total and unique words
* Display word frequencies, most frequent, and least frequent words
* Find the longest word

***Program Logic :***

**Libraries Used**

* nltk: For text tokenization
* collections.Counter: For word frequency counting

**Data Preparation**

* Paragraph: A multiline string representing a descriptive passage
* Lowercase Conversion: Ensures case-insensitive analysis
* Tokenization: Splits the paragraph into words and punctuation using nltk.word\_tokenize()
* Filtering Words: Retain only alphabetic tokens using isalpha() to remove punctuation

**Step-by-Step Logic for Each Sub-Part :**

***a) Preprocessing***

* Convert the paragraph to lowercase (lower()).
* Tokenize the paragraph into words using nltk.word\_tokenize().
* Filter out non-alphabetic tokens using a list comprehension.

***b) Counting Words***

* total\_words: len(words) gives total number of valid (filtered) words.
* unique\_words: Convert words to a set, then use len(set(words)) to count unique words.

***c) Word Frequency***

* Use Counter(words) to count frequency of each word.
* most\_common(1)[0] gives the most frequent word.
* min(..., key=item[1]) gives the least frequent word.

***d) Longest Word***

* Use max(words, key=len) to find the word with the maximum length.

***Python Source Code :***

# Q2. Text Processing (Basics)

# Importing the required libraries

import nltk

from collections import Counter

nltk.download('punkt\_tab')      # Downloading the 'punkt\_tab' module

# a)  Define a string containing a paragraph as the value.

paragraph = """A bright, sunny day is a welcome sight, filling the world with warmth and light.

The sky is a brilliant blue, and white, fluffy clouds drift lazily across the expanse, casting a gentle shadow here and there.

The sun shines warmly, and the air feels fresh and clean, carrying the scent of blooming flowers and the sound of chirping birds."""

lower\_paragraph = paragraph.lower()     # Converting the paragraph into lowercase

tokens = nltk.word\_tokenize(lower\_paragraph)        # Tokenizing the paragraph into words

words = [word for word in tokens if word.isalpha()]     # Filtering out the punctuations and keeping only the alphabetic words

# b) Write a program to print the number of total words and total unique words in the paragraph.

# Number of total words in the paragraph

total\_words = len(words)

print("The total number of words in the paragraph is :", total\_words)

# Number of unique words in the paragraph

unique\_words = len(set(words))

print("The number of unique words in the paragraph is :", unique\_words)

# c) Find the frequency of all words and also display the most and least frequent word.

# Frequency of all words

word\_frequencies = Counter(words)

print("\n Word Frequencies :")

for word, freq in word\_frequencies.items():

    print(f"{word}: {freq}")

# Most frequent word

most\_frequent = word\_frequencies.most\_common(1)[0]

print("The most frequent word in the paragraph is :")

print(f"{most\_frequent[0]} : {most\_frequent[1]} times")

# Least frequent word

least\_frequent = min(word\_frequencies.items(), key=lambda item: item[1])

print("\nLeast Frequent Word:")

print(f"{least\_frequent[0]}: {least\_frequent[1]} time")

# d) Find the longest word in the paragraph.

longest\_word = max(words, key=len)

print("\nLongest Word:")

print(longest\_word)

***Test Cases :***

* Preprocessing and Tokenization :

|  |  |  |
| --- | --- | --- |
| **Test Cases** | **Input** | **Expected Output** |
| 1 | Capital letters | All words converted to lowercase |
| 2 | Punctuation present | Tokens like , or . removed |
| 3 | Words with hyphens like "w ell-being" | Hyphen removed (if filtered properly) |
| 4 | "The the THE" | All converted to "the" |
| 5 | Empty paragraph | No tokens |
| 6 | "123 and dogs!" | "and", "dogs" kept; "123" removed |
| 7 | Only punctuations | No words returned |

* Total and Unique Word Count :

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Cases** | **Input** | **Expected Total** | **Expected Unique** |
| 1 | Single repeated word | 5 | 1 |
| 2 | All unique words | 5 | 5 |
| 3 | Mixed repetition | 10 | 6 |
| 4 | Empty string | 0 | 0 |
| 5 | Numbers and symbols | 0 | 0 |
| 6 | Case insensitive check | “The the THE” → 3 total | 1 unique |
| 7 | Proper paragraph (given) | Matches actual | Matches actual |

* Word Frequencies :

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case** | **Word List** | **Most Frequent** | **Least Frequent** |
| 1 | ["sun", "sun", "moon"] | sun: 2 | moon: 1 |
| 2 | ["a", "b", "a", "b", "c"] | a/b: 2, c: 1 | c: 1 |
| 3 | All unique | All freq = 1 | Any word with 1 |
| 4 | One word repeated | That word: count | That word: same |
| 5 | Empty list | None | None |
| 6 | Word appears most | Found with Counter.most\_common |  |
| 7 | Word appears least | Found with min() |  |

* Longest Word :

|  |  |  |
| --- | --- | --- |
| **Test Case** | **Word List** | **Expected Output** |
| 1 | ["sun", "flower", "sky"] | flower |
| 2 | ["a", "bb", "ccc", "dddd"] | dddd |
| 3 | All words same length | First longest word |
| 4 | One word | That word |
| 5 | Empty list | Error or none |
| 6 | Long compound words | Matches longest |
| 7 | Words with symbols filtered out | Based on clean list |
|  |  |  |

**Program Question 3 : Regular Expression**

2.1. Write regular expressions for the following languages :

1.  the set of all alphabetic strings

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

3.  the set of all strings from the alphabet a,b such that each a is immediately preceded by and immediately followed by a 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”);

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

3. 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);

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

**Observations :**

***Program Description :*** This program demonstrates the use of regular expressions to solve two types of pattern recognition problems:

* Language patterns involving character rules (alphabet-only, lowercase ending with b, etc.)
* Text pattern recognition within sentences (repeated words, patterns across start and end of string, etc.)

***Requirement :***

2.1: Character-Level Language Patterns

* Match all strings made up only of alphabetic characters.
* Match lowercase strings ending with the letter b.
* Match strings over alphabet {a, b} where every a is surrounded by b’s.

2.2: Word-Level Sentence Patterns

* Match sentences with two consecutive repeated words.
* Match strings that start with an integer and end with a word.
* Match strings containing both the word grotto and raven.
* Extract the first word of a sentence (ignoring leading punctuation/space).

***Program Logic :***

* **Libraries and Tools :**
* re (Regular Expressions): Used to match specific patterns in strings using regex syntax.
* re.fullmatch() / re.search() / re.match(): Used depending on whether the match is required over the full string or a part.
* **2.1 Logic**
* Pattern 1 – ^[A-Za-z]+$ (Alphabetic strings only):
  + - Anchored start (^) and end ($)
    - [A-Za-z]+: One or more alphabetic characters
* Pattern 2 – ^[a-z]\*b$ (Lowercase ending in b):
  + - Lowercase letters only
    - Ends specifically with b
* Pattern 3 – ^(b\*(ab)+b\*)\*$ (Every a surrounded by b):
  + - Zero or more valid groups of ab with surrounding bs
    - Only a inside valid groups
* **2.2 Logic**
* Pattern 1 – \b(\w+)\s+\1\b (Repeated words):
  + - Uses capturing group for a word
    - Checks for immediate repetition using \1
    - Case-insensitive (re.IGNORECASE)
* Pattern 2 –^\d+.\*\b[a-zA-Z]+\b$ ( Start with integer, end with word):
  + - ^\d+: Starts with one or more digits
    - .\*: Any characters in between
    - Ends with a word using word boundary \b
* Pattern 3 – : (?=.\*\bgrotto\b)(?=.\*\braven\b).\* (Contains both grotto and raven):
  + Lookaheads used to ensure both words are present
  + Word boundaries (\b) to avoid partial matches like "grottos"
* Pattern 4 – ^\s\*([A-Za-z]+)\b (First word in a sentence):
  + - Skips leading whitespace or punctuation
    - Captures the first actual word

***Python Source Code :***

# Q3. Regular Expression

# 2.1. Write regular expressions for the following languages :

import re

# 1) the set of all alphabetic strings

pattern1 = r'^[A-Za-z]+$'

test\_strings1 = ["Hello", "Python3", "duma", "123", "DUMA"]

print("\nAlphabetic strings only :")

for string in test\_strings1:

    if re.fullmatch(pattern1, string):

        print(f"'{string}' is valid")

    else:

        print(f"'{string}' is not valid")

# 2)  the set of all lower case alphabetic strings ending in a b

pattern2 = r'^[a-z]\*b$'

test\_strings2 = ["ab", "cab", "bb", "hello", "aaB", "b", "xyzb"]

print("\nLowercase alphabetic strings ending in 'b'")

for string in test\_strings2:

    if re.fullmatch(pattern2, string):

        print(f"'{string}' is valid")

    else:

        print(f"'{string}' is not valid")

# 3) the set of all strings from the alphabet a,b such that each a is immediately preceded by and immediately followed by a b

pattern3 = r'^(b\*(ab)+b\*)\*$'  # every 'a' must be between 'b's

test\_strings3 = ["bab", "bbabb", "ab", "a", "baba", "b", "aa", "bb"]

print("\nEach 'a' must be immediately preceded and followed by 'b'")

for string in test\_strings3:

    if re.fullmatch(pattern3, string):

        print(f"'{string}' is valid")

    else:

        print(f"'{string}' is not valid")

# 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”);

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

# 3. 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);

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

# Q1: Two consecutive repeated words

print("Repeated words")

test\_sentences1 = [

    "Humbert Humbert loved Lolita.",

    "the the dog barked",

    "the big bug",

    "go go go",

    "hello Hello"

]

pattern1 = r'\b(\w+)\s+\1\b'

for sentence in test\_sentences1:

    match = re.search(pattern1, sentence, re.IGNORECASE)

    if match:

        print(f"Match: '{match.group()}' in → '{sentence}'")

    else:

        print(f"No match → '{sentence}'")

# Q2: Start with integer and end with word

print("\nStarts with integer and ends with word")

test\_sentences2 = [

    "123 the",

    "4567 hello there",

    "98 Done!",

    "Start 123",

    "12"

]

pattern2 = r'^\d+.\*\b[a-zA-Z]+\b$'

for sentence in test\_sentences2:

    match = re.fullmatch(pattern2, sentence)

    if match:

        print(f"Valid → '{sentence}'")

    else:

        print(f"Invalid → '{sentence}'")

# Q3: Contains both 'grotto' and 'raven'

print("\nContains both 'grotto' and 'raven'")

test\_sentences3 = [

    "The raven sat outside the dark grotto.",

    "There was a grotto but no raven here.",

    "Ravens flew past grottos.",

    "The grotto had nothing in it.",

    "grotto raven"

]

pattern3 = r'\b(?=.\*\bgrotto\b)(?=.\*\braven\b).\*'

for sentence in test\_sentences3:

    match = re.search(pattern3, sentence, re.IGNORECASE)

    if match:

        print(f"Valid → '{sentence}'")

    else:

        print(f"Invalid → '{sentence}'")

# Q4: First word of the sentence

print("\nFirst word of sentence")

test\_sentences4 = [

    " Hello, how are you?",

    "Start the machine.",

    "  wait! I’ll do it.",

    "123 Go",

    "?Who knows."

]

pattern4 = r'^\s\*([A-Za-z]+)\b'

for sentence in test\_sentences4:

    match = re.match(pattern4, sentence)

    if match:

        print(f"First word: '{match.group(1)}' from → '{sentence}'")

    else:

        print(f"No first word found → '{sentence}'")

***Test Cases :***

2.1. Language Patterns :

* + - 1. Alphabetic Strings

|  |  |  |
| --- | --- | --- |
| **Test Cases** | **Input** | **Expected** |
| 1 | "Hello" | Valid |
| 2 | "Python3" | Not valid |
| 3 | "duma" | Valid |
| 4 | "123" | Not valid |
| 5 | "DUMA" | Valid |
| 6 | "hello!" | Not valid |
| 7 | "ValidOne" | Valid |

* + - 1. Lowercase ending in b

|  |  |  |
| --- | --- | --- |
| **Test Cases** | **Input** | **Expected** |
| 1 | "ab" | Valid |
| 2 | "cab" | Valid |
| 3 | "bb" | Valid |
| 4 | "hello" | Not valid |
| 5 | "aaB" | Not valid |
| 6 | "b" | Valid |
| 7 | "xyzb" | Valid |

* + - 1. a surrounded by b’s

|  |  |  |
| --- | --- | --- |
| **Test Cases** | **Input** | **Expected** |
| 1 | "bab" | Valid |
| 2 | "bbabb" | Valid |
| 3 | "ab" | Valid |
| 4 | "a" | Not valid |
| 5 | "baba" | Not valid |
| 6 | "b" | Valid |
| 7 | "aa" | Not valid |

2.2. Sentence Patterns :

1. Repeated Words

|  |  |  |
| --- | --- | --- |
| **Test Cases** | **Sentence** | **Match** |
| 1 | "Humbert Humbert loved Lolita" | Humbert Humbert (matched) |
| 2 | "the the dog barked" | the the (matched) |
| 3 | "the big bug" | Not matched |
| 4 | "go go go" | go go (matched) |
| 5 | "hello Hello" | hello Hello (matched) |
| 6 | "Bug bug bug" | bug bug (matched) |
| 7 | "repeat Repeat" | repeat Repeat (matched) |

1. Starts with int, ends with words

|  |  |  |
| --- | --- | --- |
| **Test Cases** | **Sentence** | **Match** |
| 1 | "123 the" | Valid |
| 2 | "4567 hello there" | Valid |
| 3 | "98 Done!" | Invalid |
| 4 | "Start 123" | Invalid |
| 5 | "12" | Valid |
| 6 | "1 one" | Valid |
| 7 | "987 finish!" | Invalid |

1. Contains both grotto and raven

|  |  |  |
| --- | --- | --- |
| **Test Cases** | **Sentence** | **Match** |
| 1 | "The raven sat outside the dark grotto." | Valid |
| 2 | "There was a grotto but no raven here." | Valid |
| 3 | "Ravens flew past grottos." | Invalid |
| 4 | "The grotto had nothing in it." | Invalid |
| 5 | "grotto raven" | Valid |
| 6 | "a raven and a grotto" | Valid |
| 7 | "grotto-raven connection" | Valid |
|  |  |  |

1. First word of sentence

|  |  |  |
| --- | --- | --- |
| **Test Cases** | **Sentence** | **First Word** |
| 1 | " Hello, how are you?" | Hello |
| 2 | "Start the machine." | Start |
| 3 | " wait! I’ll do it." | wait |
| 4 | "123 Go" | Go |
| 5 | "?Who knows." | Who |
| 6 | "Hello there." | Hello |
| 7 | " Finally!" | Finally |

**Program Question : Practice Question 1 : Text Normalization**

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

**Observations :**

***Program Description :*** This program performs text normalization and exploratory data analysis (EDA) on a .txt file containing a chapter from a textbook or storybook. The analysis includes line and character counts, lowercasing, punctuation removal, word-level statistics, and frequency distribution of words

***Requirements :***

* Read a .txt file containing text.
* Normalize the text by:
* Converting to lowercase
* Removing punctuation
* Perform EDA:
  + Count total lines, characters, words, and unique words
  + Identify the longest word
  + Display 10 most frequent and 10 least frequent words

***Program Logic :***

* **Libraries Used**
* string: For accessing punctuation list
* collections.Counter: For word frequency analysis
* **Step-by-Step Breakdown**

1. Load the Text File
   * Open the file using open() in read mode with UTF-8 encoding.
   * Read all lines into a list lines for line-based stats.
   * Reset pointer with seek(0) to read entire content as raw\_text.
2. Basic Statistics
   * Count total lines using len(lines)
   * Count total characters using len(raw\_text)
3. Text Normalization
   * Convert text to lowercase: text\_lower = raw\_text.lower()
   * Remove punctuation using translate(str.maketrans(...))
   * Tokenize into words using split() (splits by whitespace)
4. Word Statistics
   * Total words: len(words)
   * Unique words: len(set(words))
   * Frequency distribution: Counter(words)
   * Most frequent: word\_freq.most\_common(10)
   * Least frequent: last 10 of sorted most\_common
   * Longest word: max(words, key=len)
5. Display Results
   * Print all the above findings in a clean, readable format

***Python Source Code :***

# Practice Q1. Text Normalization

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

import string

from collections import Counter

# Step 1: Load the file

with open("Chapter.txt", "r", encoding="utf-8") as file:

    lines = file.readlines()

    file.seek(0)

    raw\_text = file.read()

# Step 2: Basic statistics

num\_lines = len(lines)

num\_chars = len(raw\_text)

# Step 3: Text normalization

text\_lower = raw\_text.lower()

text\_no\_punct = text\_lower.translate(str.maketrans("", "", string.punctuation))

words = text\_no\_punct.split()

# Step 4: Word statistics

total\_words = len(words)

unique\_words = len(set(words))

word\_freq = Counter(words)

most\_common = word\_freq.most\_common(10)

least\_common = word\_freq.most\_common()[-10:]

longest\_word = max(words, key=len)

# Step 5: Print all EDA results

print("---- Text EDA Report ----")

print(f"Total Lines: {num\_lines}")

print(f"Total Characters: {num\_chars}")

print(f"Total Words: {total\_words}")

print(f"Unique Words: {unique\_words}")

print(f"Longest Word: {longest\_word}")

print("\nTop 10 Most Frequent Words:")

for word, count in most\_common:

    print(f"{word}: {count}")

print("\n10 Least Frequent Words:")

for word, count in least\_common:

    print(f"{word}: {count}")

***Test Cases :***

|  |  |  |
| --- | --- | --- |
| **Test Case** | **File Content Sample** | **Expected Output** |
| 1 | Empty file | 0 lines, 0 characters, 0 words |
| 2 | Single sentence with punctuation | Line count = 1, punctuation removed |
| 3 | Paragraph with repeated words | Most frequent word shown correctly |
| 4 | Text with upper and lowercase words | All normalized to lowercase |
| 5 | Text with numeric characters | Numeric tokens retained unless manually filtered |
| 6 | Text with multiple newlines | Correct line count shown |
| 7 | Long words like "uncharacteristically" | Longest word correctly identified |

**Program Question : Practice Question 2 : 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', ...].

a. Ending in ize

b. Containing the letter z

c. Containing the sequence of letters pt

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

**Observation :**

***Program Description :*** Write expressions to extract specific patterns from text6 in the form of a list:

a) Words ending in "ize"

b) Words containing the letter "z"

c) Words containing "pt"

d) Words that are **titlecase** (first letter capital only)

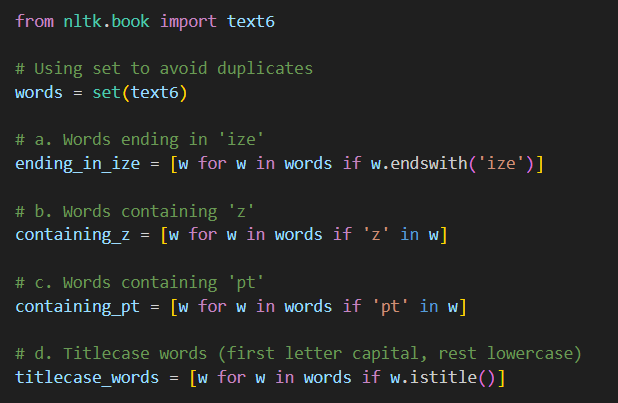
***Requirements :***

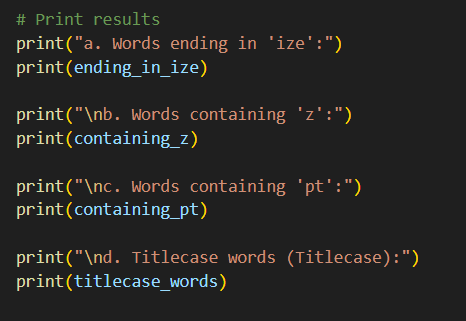
* Use nltk.book.text6 which loads the chat corpus.
* Convert the word list into a **set** to remove duplicates.
* Apply string methods like endswith, in, istitle, etc., in list comprehensions.

***Program Logic :***

* set(text6): Removes duplicates for efficiency.
* w.endswith("ize"): Filters words that end in "ize".
* "z" in w: Filters words that contain the letter "z".
* "pt" in w: Looks for substring "pt" inside the word.
* w.istitle(): Captures words in **Titlecase** (like "Hello").

***Python Source Code :***





***Test Cases :***

|  |  |  |
| --- | --- | --- |
| **Test Case** | **Word** | **Matches** |
| 1 | "finalize" | a |
| 2 | "crazy" | b |
| 3 | "empty" | c |
| 4 | "Apartment" | c, d |
| 5 | "Zebra" | b, d |
| 6 | "Optimize" | a, d |
| 7 | "normal" | None |

**Program Question : Practice Question 2 : From TextBook Exercises: Steve Bird and team.**

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.

**Observation :**

***Program Description :***

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

a) Find all words starting with 'sh'

b) Find all words with length > 4

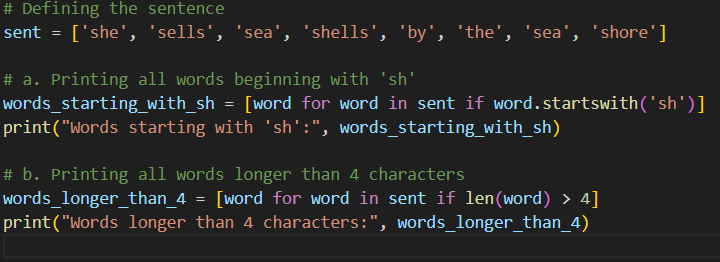
***Requirements :***

Use list comprehensions with startswith() and len().

***Program Logic :***

* Loop through list sent
* Use startswith('sh') to find ['she', 'shells', 'shore']
* Use len(word) > 4 to find ['sells', 'shells', 'shore']

***Python Source Code :***



***Test Cases :***

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Case** | **Input Word** | | **a) Starts with 'sh'** | **b) Length > 4** |
| 1 | "she" | Valid | | Not Valid |
| 2 | "shells" | Valid | | Valid |
| 3 | "sea" | Not Valid | | Not Valid |
| 4 | "sells" | Not Valid | | Valid |
| 5 | "shore" | Valid | | Valid |

**Program Question : Practice Question 2 : From TextBook Exercises: Steve Bird and team.**

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?

**Observations :**

***Program Description :*** This is a generator expression that goes through each word w in text1 and calculates the length of each word using len(w) and adds up all those lengths using sum(). We can use this generator expression to work out the average word length of a text.

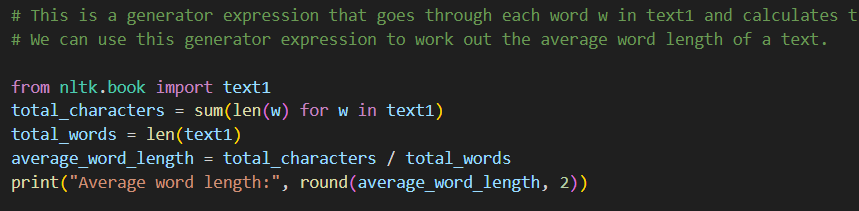
***Requirements :***

* Use sum() to calculate total characters
* Divide by len(text1) for average.

***Problem Logic :***

* text1: List of words from "Moby Dick"
* len(w): Character count per word
* sum(...): Total character count
* average = total\_chars / total\_words

***Python Source Code :***



***Test Cases :***

|  |  |  |
| --- | --- | --- |
| **Test Case** | **Text1 Subset** | **Average Word Length** |
| 1 | ["cat", "dog"] | 3.0 |
| 2 | ["the", "whale"] | 4.0 |
| 3 | [] | Error (zero div) |
| 4 | ["I", "am"] | 1.5 |
| 5 | Full text1 | ~4.3 |

**Program Question : Practice Question 2 : From TextBook Exercises: Steve Bird and team.**

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.

**Observations :**

***Problem Description :*** Create a function to return vocabulary size (distinct words) of a text.

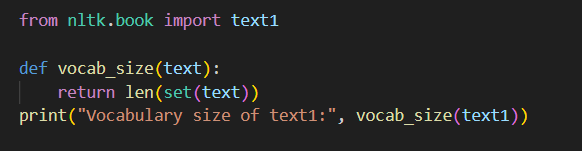
***Requirements :***

* Input: text (list of words)
* Output: len(set(text))

***Problem Logic :***

* set(text) removes duplicates
* len() gives vocabulary size

***Python Source Code :***



***Test Cases :***

|  |  |  |
| --- | --- | --- |
| **Test Case** | **Input Text** | **Output** |
| 1 | ["cat", "dog", "cat"] | 2 |
| 2 | ["hello", "hello", "hi"] | 2 |
| 3 | [] | 0 |
| 4 | ["apple", "banana", "grape"] | 3 |
| 5 | text1 | ~19000 |

**Program Question : Practice Question 2 : From TextBook Exercises: Steve Bird and team.**

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.

**Observations :**

***Problem Description :*** Create a function that calculates how often a word appears in a text, expressed as a percentage.

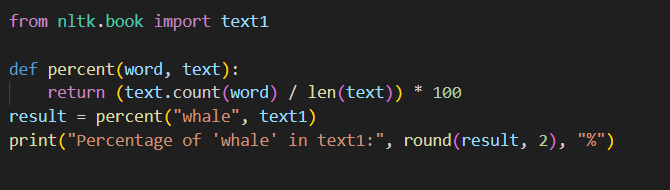
***Requirements :***

* text.count(word) counts occurrences
* Divide by len(text) and multiply by 100

***Problem Logic :***

(word frequency / total words) \* 100

***Python Source Code :***



***Test Cases :***

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case** | **Input Text** | **Word** | **Output %** |
| 1 | ["a", "b", "a", "a"] | "a" | 75.0 |
| 2 | ["a", "b", "c"] | "d" | 0.0 |
| 3 | ["a", "b", "b", "b", "b"] | "b" | 80.0 |
| 4 | [] | "any" | Error |
| 5 | text1 | "whale" | ~0.42% |

***Evaluation Comments :***