DATE:20-06-25

LAB NO:1

Q1. Installing NLTK, NLTK.book and Practice the NLP Environment using the exercises 1 and 2 from the given link. Language Processing and Python (nltk.org)

Program Description

This exercise begins by installing the NLTK library and downloading its “book” collection, allowing Python to load and explore sample texts. With the NLP environment verified, you then use the interpreter as a basic calculator—evaluating expressions such as `12 / (4 + 1)`—and apply Python’s exponentiation operator to a combinatorial problem: first computing `26**10` to confirm the number of ten-letter strings over a 26-letter alphabet, then evaluating `26**100` to determine the total number of possible hundred-letter strings.

Program Logic

Libraries

- `nltk` (to install and load the “book” corpus)
- No external libraries needed for arithmetic operations

Data Types

- **Strings** for file paths and corpus names
- **Integers** for exponentiation (`26**10`, `26**100`)
- **Floats** for division results (`12 / (4 + 1)`)

PROGRAM:

```
import nltk
```

Generate | Code | Markdown

```
nltk.download()
```

```
showing info https://raw.githubusercontent.com/nltk/nltk_data/gh-pages/index.xml

True
```

```
from nltk.book import *
```

✓ 21.4s

```
*** Introductory Examples for the NLTK Book ***
Loading text1, ..., text9 and sent1, ..., sent9
Type the name of the text or sentence to view it.
Type: 'texts()' or 'sents()' to list the materials.
text1: Moby Dick by Herman Melville 1851
text2: Sense and Sensibility by Jane Austen 1811
text3: The Book of Genesis
text4: Inaugural Address Corpus
text5: Chat Corpus
text6: Monty Python and the Holy Grail
text7: Wall Street Journal
text8: Personals Corpus
text9: The Man Who Was Thursday by G . K . Chesterton 1908
```

```
#Q1. Try using the Python interpreter as a calculator, and typing expressions like 12 / (4 + 1).
result = 12 / (4 + 1)
print(result)
```

✓ 0.0s

```
2.4
```

Generate | Code | Markdown

```
#Q2.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?
print(26**100)
```

✓ 0.0s

```
3142930641582938830174357788501626427282669988762475256374173175398995908420104023465432599069702289330964075081611719197835869803511992549376
```

TEST CASES:

Test Case	Action	Expected Output
Calculator	**12 / (4 + 1)**	**12 / (4 + 1) = 2.4**
Hundred-letter strings	**26 ** 100**	**26**100 = 3142930641582938830174357788501626427282669988762475256374173175398995908420104023465432599069702289330964075081611719197835869803511992549376**

Q2.Text Processing (Basics)

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

Program Description

This program performs basic text processing on a given paragraph. It calculates:

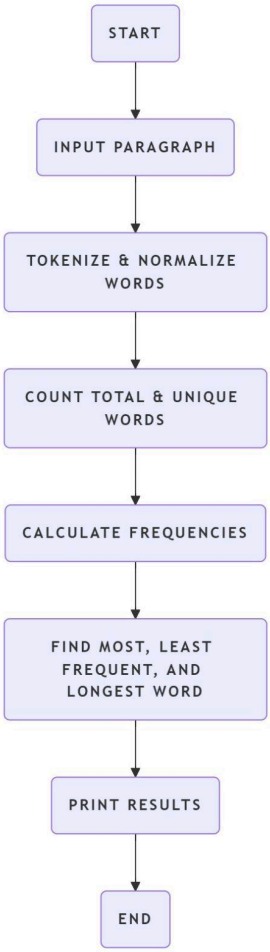
- Total number of words
- Total number of unique words
- Frequency of all words
- Most and least frequent word
- Longest word in the paragraph

Program Logic

Algorithm:

1. Accept a paragraph as a string.
2. Tokenize the paragraph into words using nltk.word_tokenize.
3. Normalize the words (convert to lowercase, remove punctuation).
4. Count total words and unique words using len() and set().
5. Use nltk.FreqDist or collections.Counter to calculate word frequencies.
6. Find:
    - The most frequent word using .most_common(1)
    - The least frequent using .most_common()[-1]
    - The longest word using max() with key=len

FLOWCHART



Preprocessing

- Convert the text to lowercase (to normalize case).
- Remove punctuation for clean word analysis.
- Split the paragraph into words using `.split()`.

Operation	Logic / Function Used
Total number of words	Use `len()` on the word list.
Total number of unique words	Convert list to `set()` and use `len()`.
Frequency of all words	Use `collections.Counter` to count occurrences.
Most frequent word	Use `.most_common(1)` from `Counter`.
Least frequent word	Filter words with frequency = 1 (hapaxes).
Longest word	Use `max(words, key=len)` to find the word with max length.

Libraries Used

- `string` — for punctuation removal.
- `collections.Counter` — to count word frequencies.

Test Cases

Test Case Description	Input	Expected Output
Normal paragraph	`"The sun shines. The sun sets."`	Total words: 6Unique: 4Most: "the" (2)Least: "shines"/"sets"/"sun" (1)Longest: "shines" or "sets"
Case-insensitive check	`"Python python PYTHON."`	Total: 3Unique: 1Most: "python" (3)Least: —Longest: "python"
Punctuation handling	`"Hi! Hello, world."`	Total: 3Unique: 3Most: all equal (1)Least: allLongest: "hello"
All unique words	`"One two three four"`	Total: 4Unique: 4Most: all equal (1)Least: allLongest: "three"
Equal frequency tie	`"cat cat dog dog"`	Total: 4Unique: 2Most: "cat"/"dog" (2)Least: —Longest: "cat"/"dog"
Longest word check	`"tiny enormousness"`	Total: 2Unique: 2Longest: "enormousness"
Empty paragraph	`""`	Total: 0Unique: 0Most: —Least: —Longest: ""
Non-alphabetic characters	`"123 $$ wow!"`	Total: 1Unique: 1Most: "wow" (1)Longest: "wow"
Repeated long word	`"elephant elephant elephant"`	Total: 3Unique: 1Most: "elephant" (3)Longest: "elephant"

PROGRAM

```
#QUESTION 2:Text Processing (Basics)


from nltk.tokenize import word_tokenize
from collections import Counter
import string

# a)Define a string containing a paragraph as the value.
paragraph = """
The curator of the Louvre, Jacques Saunière, staggered through the vaulted
archway of the museum's Grand Gallery.
He clutched his chest, his fingers smeared with blood. Beneath the eerie
glow of the gallery lights, priceless
paintings gazed down with eternal stillness. Somewhere behind him,
footsteps echoed. Panic surged. Saunière
turned, slipping on the polished floor. As he fell, memories flooded his
mind—cryptic symbols, secret societies,
hidden truths buried in history. He knew his time was short. In trembling
strokes, he wrote a message—his last clue.
"""


# Step 1: Tokenize
tokens = word_tokenize(paragraph)

# Step 2: Normalize - lowercase and remove punctuation
words = [word.lower() for word in tokens if word.isalpha()]


#b)Write a program to print the number of total words and total unique
words in the paragraph.
# Step 3: Count total and unique words
total_words = len(words)
unique_words = len(set(words))


#c)Find the frequency of all words and also display the most and least
frequent word.
# Step 4: Frequency
```

```python
freq = Counter(words)

# Step 5: Most and least frequent words
most_freq = freq.most_common(1)[0]
least_freq = freq.most_common()[-1]


#d)Find the longest word in the paragraph.
# Step 6: Longest word
longest_word = max(words, key=len)

# Step 7: Output
print(f"Total words: {total_words}")
print(f"Unique words: {unique_words}")
print(f"Most frequent word: {most_freq}")
print(f"Least frequent word: {least_freq}")
print(f"Longest word: {longest_word}")
```

```
Total words: 84
Unique words: 66
Most frequent word: ('the', 7)
Least frequent word: ('clue', 1)
Longest word: staggered
```

Q3. Regular Expression

Solve exercise 2.1 and 2.2 from Text book of Speech and Language Processing of Daniel Jurafsky and team

Program Description

This program uses regular expressions to match different types of strings based on specific language rules. Each sub-question defines a unique pattern, such as detecting repeated words, words ending in specific letters, or filtering text with alphabet-only characters. The program applies these patterns to a list of sample inputs and checks whether they match or not, helping to understand how regex can be used for pattern recognition in text processing.

Program Logic

Libraries Used:

- `re`: Python's built-in regular expressions library

Approach:

- Define **regex patterns** for each sub-question (a–h)
- Prepare **test inputs** for each case
- Check which strings match each pattern
- Print output indicating success or failure of the match

TEST CASES:

(a) All alphabetic strings

Regex: ^[A-Za-z]+$

→ Matches strings with only letters (no spaces, digits, or punctuation)

Input	Match	Reason
`"hello"`	✅ Yes	All lowercase letters
`"HelloWorld"`	✅ Yes	Mixed case
`"hi123"`	❌ No	Contains digits
`"hi!"`	❌ No	Contains punctuation
`"hi there"`	❌ No	Contains space

(b) Lowercase alphabetic strings ending in 'b'

Regex: ^[a-z]*b$

Input	Match	Reason
`"bob"`	✅ Yes	Lowercase, ends with 'b'
`"ab"`	✅ Yes	Ends with 'b'
`"abc"`	❌ No	Does not end with 'b'
`"Bob"`	❌ No	Has uppercase 'B'

(c) Two consecutive repeated words (e.g., "the the")
Regex (word boundary-based): \b(\w+)\s+\1\b

Input	Match	Reason
`"the the"`	✅ Yes	Word repeated
`"Humbert Humbert"`	✅ Yes	Valid repeated word
`"the big bug"`	❌ No	No repeated word
`"thethe"`	❌ No	No space between words

(d) Each 'a' is immediately preceded and followed by 'b' in a string of a's and b's

Input	Match	Reason
`"bab"`	✅ Yes	'a' is surrounded by 'b's
`"baab"`	❌ No	First 'a' is followed by another 'a'
`"babbab"`	✅ Yes	All 'a's have 'b' before and after

(f) Strings that start with an integer and end with a word

Regex: ^\d+.*\b([A-Za-z]+)\b$

Input	Match	Reason
`"42 hello"`	✅ Yes	Starts with number, ends with word
`"1. The world"`	✅ Yes	Starts with number, ends with word
`"abc 123"`	❌ No	Starts with word, ends with number

(g) Contains both 'grotto' and 'raven' (as full words)

Input	Match	Reason
`"the grotto and the raven"`	✅ Yes	Contains both words
`"raven in the grotto"`	✅ Yes	Valid
`"grottos are scary"`	❌ No	'grottos' ≠ 'grotto'
`"grotto"`	❌ No	Missing 'raven'

(h) Place the first word of a sentence in a register (capture it)

Regex (capture first word): ^([A-Z][a-z]*)

Input	Match	Captured	Reason
`"Hello, how are you?"`	✅ Yes	"Hello"	Starts with a capital word
`"this is fine."`	❌ No	—	Does not start with capital
`"Wow! That's great."`	✅ Yes	"Wow"	"Wow" is the first word

```
import re

print("\n Regular Expression Tester\n")

# a. the set of all alphabetic strings.
regex_a = r'^[A-Za-z]+$'
test_cases_a = ["Hello", "world", "123", "hello123", "hello_world"]
print("# a. All alphabetic strings")
for test in test_cases_a:
    print(f"  {test!r} ->", "Match" if re.fullmatch(regex_a, test) else
"No match")

print("\n" + "-"*60)

# b. the set of all lowercase alphabetic strings ending in a b.
regex_b = r'^[a-z]*b$'
test_cases_b = ["b", "ab", "aab", "abc", "BB", "bbb"]
print("# b. Lowercase alphabetic strings ending in 'b'")
for test in test_cases_b:
    print(f"  {test!r} ->", "Match" if re.fullmatch(regex_b, test) else
"No match")

print("\n" + "-"*60)

# c. Strings with two consecutive repeated words
regex_c = r'\b(\w+)\s+\1\b'
test_cases_c = ["Humbert Humbert", "the the", "the bug", "a a", "yes no"]
print("# c. Two consecutive repeated words")
for test in test_cases_c:
    print(f"  {test!r} ->", "Match" if re.search(regex_c, test) else "No
match")

print("\n" + "-"*60)

# d. Strings from {a, b} where each 'a' is surrounded by 'b'
regex_d = r'^(b*ab+)*b*$'
test_cases_d = ["bab", "bbabb", "ab", "a", "ba", "abba", "bbabbbab"]
print("# d. Strings from {a, b} where each 'a' is surrounded by 'b'")
for test in test_cases_d:
```

```
    print(f"  {test!r} ->", "Match" if re.fullmatch(regex_d, test) else
"No match")

print("\n" + "-"*60)

# f. Strings that start with an integer and end with a word
regex_f = r'^\d+.*\b([A-Za-z]+)\b$'
test_cases_f = ["1000 hello", "3 days remain", "1", "start 100", "99
bottles"]
print("# f. Start with integer and end with a word")
for test in test_cases_f:
    print(f"  {test!r} ->", "Match" if re.match(regex_f, test) else "No
match")

print("\n" + "-"*60)

# g. Strings with both "grotto" and "raven"
regex_g = r'\b(grotto)\b.*\b(raven)\b|\b(raven)\b.*\b(grotto)\b'
test_cases_g = ["The raven found the grotto.", "grotto and raven",
"grottos and ravens", "raven in the cave", "just a grotto"]
print("# g. Strings with both 'grotto' and 'raven'")
for test in test_cases_g:
    print(f"  {test!r} ->", "Match" if re.search(regex_g, test) else "No
match")

print("\n" + "-"*60)
```

```
Q2.1 Regular Expression Tester

# a. All alphabetic strings
  'Hello' -> Match
  'world' -> Match
  '123' -> No match
  'hello123' -> No match
  'hello_world' -> No match

------------------------------------------------------------
# b. Lowercase alphabetic strings ending in 'b'
  'b' -> Match
  'ab' -> Match
  'aab' -> Match
  'abc' -> No match
  'BB' -> No match
  'bbb' -> Match

------------------------------------------------------------
# c. Two consecutive repeated words
  'Humbert Humbert' -> Match
  'the the' -> Match
  'the bug' -> No match
  'a a' -> Match
  'yes no' -> No match
```

```
# d. Strings from {a, b} where each 'a' is surrounded by 'b'
   'bab' -> Match
   'bbabb' -> Match
   'ab' -> Match
   'a' -> No match
   'ba' -> No match
   'abba' -> No match
   'bbabbbab' -> Match

 ------------------------------------------------------------
 # f. Start with integer and end with a word
   '1000 hello' -> Match
   '3 days remain' -> Match
   '1' -> No match
   'start 100' -> No match
   '99 bottles' -> Match

 ------------------------------------------------------------
 # g. Strings with both 'grotto' and 'raven'
   'The raven found the grotto.' -> Match
   'grotto and raven' -> Match
   'grottos and ravens' -> No match
   'raven in the cave' -> No match
   'just a grotto' -> No match
```

2.2 Implement an ELIZA-Iike program, using substitutions such as those described on page IO. You may choose a different domain than a Rogerian psychologist, if you wish, although keep in mind that you would need a domain in which your program can legitimately do a lot of simple repeating-back.

Program Description

This program simulates an ELIZA-style chatbot, inspired by one of the earliest natural language processing systems. It uses simple pattern matching and substitution rules to reply to user inputs in a conversational style, mimicking a virtual therapist. When the user enters a sentence, the bot reflects parts of the sentence and responds with a suitable template. The conversation continues until the user types "exit", "quit", or "bye".

Program Logic

Libraries Used:

- `re`: Used to perform pattern matching using regular expressions.
- `random`: Used to randomly select a response from the list of possible responses for a pattern**.**

Test Cases

User Input	Expected Output (Example)
I need advice	Why do you need advice? / Would it really help you to get advice?
I can't sleep	How do you know you can't sleep? / Perhaps you could sleep if you tried.
I am anxious	Did you come to me because you are anxious? / How long have you been anxious?
Why don't you help me?	Do you really think I don't help me? / Perhaps eventually I will help me.
What should I do?	Why do you ask that? / What do you think?
I lost my phone	Please tell me more. / Why do you say that you lost your phone?
quit	ELIZA: Goodbye! Take care. (Chat exits)

nonsense input	Please tell me more. / Can you elaborate on that?

PROGRAM

```
import re

import random



# Reflection dictionary for pronoun changes

reflections = {

    "i am": "you are",

    "i was": "you were",

    "i": "you",

    "my": "your",

    "me": "you",

    "am": "are",

    "you are": "I am",

    "you're": "I'm",

    "your": "my",

    "yours": "mine",

    "you": "I"
```

```
}


# Basic substitution rules (pattern, response)
patterns = [
    (r'I need (.*)',
     ["Why do you need {0}?",
      "Would it really help you to get {0}?",
      "Are you sure you need {0}?"]),

    (r'Why don\'?t you ([^\?]*)\??',
     ["Do you really think I don't {0}?",
      "Perhaps eventually I will {0}.",
      "Do you really want me to {0}?"]),

    (r'I can\'?t (.*)',
     ["How do you know you can't {0}?",
      "Perhaps you could {0} if you tried.",
      "What would it take for you to {0}?"]),

    (r'I am (.*)',
     ["Did you come to me because you are {0}?",
      "How long have you been {0}?",
      "How do you feel about being {0}?"]),
```

```
    (r'(.*)\?',
     ["Why do you ask that?",
      "What do you think?",
      "Can you answer your own question?"]),

    (r'(.*)',
     ["Please tell me more.",
      "Let's change focus a bit… Tell me about your family.",
      "Can you elaborate on that?",
      "How does that make you feel?",
      "Why do you say that {0}?"])
]

# Function to substitute reflections
def reflect(fragment):
    words = fragment.lower().split()
    reflected = [reflections.get(word, word) for word in words]
    return ' '.join(reflected)

# ELIZA main response generator
def eliza_response(user_input):
    for pattern, responses in patterns:
```

```
        match = re.match(pattern, user_input, re.IGNORECASE)
        if match:
            groups = match.groups()
            reflected_groups = [reflect(g) for g in groups]
            response = random.choice(responses)
            return response.format(*reflected_groups)
    return "I'm not sure I understand you."


# Chat loop
print("ELIZA: Hello, I'm your virtual therapist. How can I help you
today?")
while True:
    user_input = input("You: ")
    if user_input.lower() in ['exit', 'quit', 'bye']:
        print("ELIZA: Goodbye! Take care.")
        break
    response = eliza_response(user_input)
    print("ELIZA:", response)
```

```
ELIZA: Hello, I'm your virtual therapist. How can I help you today?
ELIZA: Would it really help you to get help?
ELIZA: How long have you been tired?
ELIZA: Perhaps eventually I will listen.
ELIZA: Why do you say that your friends ignore you?
ELIZA: Can you elaborate on that?
ELIZA: Can you elaborate on that?
ELIZA: How does that make you feel?
ELIZA: Please tell me more.
ELIZA: Let's change focus a bit… Tell me about your family.
ELIZA: Please tell me more.
ELIZA: Why do you say that no?
ELIZA: Why do you say that no?
ELIZA: Why do you say that yes?
ELIZA: Please tell me more.
ELIZA: Can you elaborate on that?
ELIZA: Can you elaborate on that?
ELIZA: Please tell me more.
ELIZA: Why do you say that breathe?
ELIZA: Goodbye! Take care.
```